Y36PJC Programování v jazyce C/C++

Funkce, makra, preprocesor

Ladislav Vagner

Dnešní přednáška

- Funkce v C/C++:
  - předávání parametrů,
  - výstupní parametry,
  - přetěžování funkcí,
  - inline funkce.
- Makra.
- Preprocesor.

Minulá přednáška

- Příkazy v C/C++.
- Řídící konstrukce.
- Vstup a výstup:
  - C,
  - C++,
  - soubory,
  - textové a binární soubory.

Funkce

- Realizace dílčího uceleného algoritmu.
- Rozdělení velkého problému na menší celky.
- Přehlednost programu.
- Zamezení opakujícího se kódu.
- Program v C:
  - tvořen funkcemi,
  - funkce se navzájem mohou volat,
  - funkci **`main`** (**`WinMain`**) je předáno řízení při spuštění.
- Program v C++:
  - obdoba C, menší význam funkcí, větší význam tříd a metod,
  - zápis metod je podobný zápisu funkcí.

Funkce

Deklarace funkce:

```
<typ> name ( <parametry> );
```

- Deklarace informuje překladač, že funkce existuje, jak se jmenuje, jaké má parametry a jaký je typ návratové hodnoty.
- Deklarace se používá pro funkce, které jsou implementované "mimo" náš program - např. knihovní funkce, funkce OS, jiný modul programu.
- Deklarace funkcí se většinou umisťují do hlavičkových souborů.

Funkce

Definice funkce:
`<typ> name ( <parametry> ) { <tělo> }`

- Definice funkce je deklarace, která navíc nese vlastní implementaci.
- Definice se zpravidla neumisťují do hlavičkových souborů (patří do .c či .cpp souborů).
- Deklarací téže funkce může v daném programu existovat více, definice právě jedna.
- Definice funkcí nelze vnořovat do sebe (funkce nemůže být definována uvnitř jiné funkce).

Funkce

Parametry funkcí, návratový typ:

- Seznam **`<typ> <identifikátor>`**, oddělený čárkou.
- Funkce bez parametrů – prázdný seznam nebo (lépe) uvést klíčové slovo **`void`**.
- Parametry stejného typu – musí se rozepsat.
- Návratový typ – **`void`** znamená proceduru.
- Vracená hodnota – výraz za **`return`**.
- Pokud funkce nevrací **`void`**, musí obsahovat **`return`** a vracet výraz správného typu.

Funkce

Příklad:

```
unsigned int gcd ( unsigned int a, unsigned int b )
 {
   while ( a != b )
    if ( a > b )
      a -= b;
     else
      b -= a;
   return ( a );
 }

unsigned int scm ( unsigned int a, unsigned int b )
 {
   return ( a * b / gcd ( a, b ) );
 }
```

Výstupní parametry funkcí

Předávání parametrů:

- Parametry jsou předávané hodnotou.
- Volající tedy má vlastní kopii, která se po jeho ukončení odstraní.
- Takto lze předávat pouze vstupní parametry.
- Výstupní a vstupně/výstupní parametry – předávat pomocí ukazatelů nebo referencí (pouze C++).
- Vstupní parametry předávané ukazatelem či referencí – označit **`const`**.
- Má smysl předávat vstupní parametry jinak než hodnotou?

Výstupní parametry funkcí

Příklad (C i C++):

```
void timeToHMS ( int t, int H, int M, int S ) // !!
 {
   H = t / 3600;
   M = ( t / 60 ) % 60;
   S = t % 60;
 }

int main ( int argc, char * argv[] )
 {
   int t, h, m, s;
   cin >> t;
   timeToHMS ( t, h, m, s );
   cout << h << ":" << m << ":" << s <<endl;
   return ( 0 );
 }
```

Výstupní parametry funkcí

Příklad (C i C++):

```
void timeToHMS ( int t, int * H, int * M, int * S )
 {
   *H = t / 3600;
   *M = ( t / 60 ) % 60;
   *S = t % 60;
 }

int main ( int argc, char * argv[] )
 {
   int t, h, m, s;
   cin >> t;
   timeToHMS ( t, &h, &m, &s );
   cout << h << ":" << m << ":" << s <<endl;
   return ( 0 );
 }
```

Výstupní parametry funkcí

Příklad (pouze C++):

```
void timeToHMS ( int t, int & H, int & M, int & S )
 {
   H = t / 3600;
   M = ( t / 60 ) % 60;
   S = t % 60;
 }

int main ( int argc, char * argv[] )
 {
   int t, h, m, s;
   cin >> t;
   timeToHMS ( t, h, m, s );
   cout << h << ":" << m << ":" << s <<endl;
   return ( 0 );
 }
```

Přetížené funkce

Přetěžování funkcí (jen C++):

- Dvě (více) funkcí může být stejného jména.
- Musí být ale odlišitelné počtem či typem parametrů.
- Nelze přetěžovat na základě návratové hodnoty.
- Pravidla pro párování typů parametrů:
  - přesná shoda,
  - roztažení,
  - standardní konverze,
  - uživatelská konverze.
- Přetěžování může být zrádné.

Přetížené funkce

Příklad přetížené funkce:

```
int abs ( int x )
 { return ( x >= 0 ? x : -x ); }

double abs ( double x )
 { return ( x >= 0 ? x : -x ); }

cout << abs ( -10 ) << " " << abs ( -10.5 );
```

Přetížené funkce

Pravidla pro párování typů parametrů:

- Přesná shoda.
- Roztažení (promotion) – nedochází ke ztrátě rozsahu ani přesnosti:
  - `char -> int`
  - `short -> int`
  - `float -> double`
- Standardní konverze – může dojít ke ztrátě přesnosti nebo rozsahu:
  - `int -> float`
  - `float -> int`
  - `int -> double`
  - `double -> int`
  - …
- Uživatelská konverze – konstruktorem uživ. konverze.

Přetížené funkce

Pravidla pro párování parametrů primitivních typů:

	char	u char	short	u short	int	u int	long	u long	long long	u long long	float	double	l double
char	E	S	S	S	P	S	S	S	S	S	S	S	S
u char	S	E	S	S	P	S	S	S	S	S	S	S	S
short	S	S	E	S	P	S	S	S	S	S	S	S	S
u short	S	S	S	E	P	S	S	S	S	S	S	S	S
int	S	S	S	S	E	S	S	S	S	S	S	S	S
u int	S	S	S	S	S	E	S	S	S	S	S	S	S
long	S	S	S	S	S	S	E	S	S	S	S	S	S
u long	S	S	S	S	S	S	S	E	S	S	S	S	S
long long	S	S	S	S	S	S	S	S	E	S	S	S	S
u long long	S	S	S	S	S	S	S	S	S	E	S	S	S
float	S	S	S	S	S	S	S	S	S	S	E	P	S
double	S	S	S	S	S	S	S	S	S	S	S	E	S
l double	S	S	S	S	S	S	S	S	S	S	S	S	E

Přetížené funkce

```
void foo ( int a, double b );
void foo ( double a, int b );

foo ( 1,2 );       // chyba
foo ( 2.0, 3.0 ); // chyba
foo ( 'a', 5 );    // chyba


void bar ( float a );
void bar ( long double b );

bar ( 10 );        // chyba
bar ( 12.0 );      // chyba
bar ( 12.0f );     // ok, bar ( float )
```

Implicitní parametry funkce

- V deklaraci funkce lze nastavit výchozí hodnoty parametrů.
- Výchozí hodnoty se použijí, pokud je funkce volaná s méně skutečnými parametry.
- Implicitní parametry lze nastavovat pouze zprava.
- Výběr volané funkce – pravidla pro přetěžování + implicitní parametry.

```
void foo ( int a, int b = 3 );
foo ( 1 );     // 1, 3
foo ( 2, 4 ); // 2, 4
```

Implicitní parametry funkce

```
void foo ( double a, int b = 10 );
void foo ( long int a );

foo ( 12 );   // chyba - nejednoznacne
foo ( 'z' );  // chyba - nejednoznacne
foo ( 15.3 ); // foo ( double, int )
foo ( 3.0f ); // foo ( double, int )
```

Inline funkce

- Volání funkce představuje nenulovou režii (příprava záznamů na zásobníku, ukládání registrů CPU, zneplatnění cache, rozpracovaných instrukcí, ...).
- Citelné zejména u krátkých a často volaných funkcí.
- Optimalizace – rozpis instrukcí funkce v místě použití.
- Klíčové slovo **`inline`**.
- Inline funkce nesmí být rekurzivní.
- Inline funkce se zpravidla umisťují do hlavičkových souborů.

```
inline int max2 ( int a, int b )
 {
   return a > b ? a : b;
 }
```

Funkce `main`

- Vstupní bod programu (předáno řízení po spuštění).
- Návratová hodnota - signalizace úspěchu programu:
  - 0 ok,
  - 1, 2, 3, … rozlišení příčiny neúspěchu.
- Vstupní parametry:
  - **`argc`** počet parametrů na příkazové řádce+1,
  - **`argv`** pole parametrů z příkazové řádky
  - **`argv[0]`** jméno spuštěného programu
  - **`argv[1]`** první parametr,
  - …
  - **`argv[argc-1]`** poslední parametr.

Preprocesor

- První část kompilátoru C/C++.
- Zajišťuje vkládání hlavičkových souborů, podmíněný překlad a makra.
- Programovatelný – příkazy preprocesoru jsou uvozeny znakem **#**.

Preprocesor

`#include`:

- Vloží na aktuální místo zdrojového souboru obsah jiného souboru.
- Lze použít i vnořeně.
- Tvar buď **`#include <filename>`** (ze systémového adresáře) nebo **`#include "filename"`** (z aktuálního adresáře).
- Typické použití – vkládání hlavičkových souborů.

Příklady:

```
#include <iostream>
#include "parser/mylexan.h"
```

Preprocesor

`#define` - zavedení substituce:

```
#define MAX_SIZE 200
```

- Identifikátor **`MAX_SIZE`** bude ve všech výskytech v programu nahrazen číslem **`200`**.
- Zavede de-facto konstantu.
- Za **`#define`** není středník.
- Substituce lze skládat, ale ne v kruhu:

```
#define PI 3.1415926535
#define PI_HALF (PI / 2.0)
```

Preprocesor

`#define` - zavedení makra:

```
#define SQR(X) ((X)*(X))
```

- Zavede makro, které se rozvine na druhou mocninu zadaného výrazu.
- X je parametrem makra.
- V zápisu **`SQR(X)`** nesmí být mezery.
- Za **`#define`** opět není středník.

Příklad:

```
a = SQR(20);      // a = ((20)*(20))
b = SQR(a - 20); // b = ((a-20)*(a-20))
```

Preprocesor

Makra jsou zrádná:

```
#define MAX        100 ;   // pozor, zde je ;
#define SQR1(X)    X*X
#define SQR2(X)    (X)*(X)
#define SQR3(X)    ((X)*(X))
#define MAX2(X,Y)  ((X)>(Y)?(X):(Y))

a = MAX + 200;      // a = 100, syntaxe OK
b = SQR1(10+10);    // b = 120 ne 400
c = ~ SQR2(10);     // c = -110 ne -101
d = ~ SQR3(10+10); // ok, d = -401
i = 10;
m = MAX2(i++,5);    // m = 11, i = 12
```

Preprocesor

Makra vs. inline funkce:

- Poskytují podobnou funkcionalitu:
  - opakující se kód je psán pouze 1x,
  - odpadá režie volání funkce.
- Makra jsou v C i v C++, inline funkce pouze v C++.
- Makra vyžadují opatrnost při psaní a používání.
- Pokud to lze, je lepší použít inline funkci.

Preprocesor

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

Podmíněný překlad:

- Blok zdrojového kódu může být začleněn / nezačleněn do překladu.
- Lze použít např. pro odstranění ladicích částí kódu při finálním překladu.

Příklad:

```
#define LINUX
…
#ifdef LINUX
 sleep ( 1 );
#else
 Sleep ( 1000 );
#endif
```

Dotazy...

Děkuji za pozornost.