Funkce

Funkce a procedury v C/C++

- definice funkce

```
typ identifikator(parametry)
{
  tělo funkce
}
```

```
int obsah(int x, int y)
{
  return x*y;
}
```

– hodnota se vrací pomocí **`return`**

- ve funkci/proceduře lze deklarovat lokální proměnnou:

```
int obsah(int x, int y)
{
  int S;
  S = x*y;
  return S;
}
```

- procedura
  - funkce, která nevrací hodnotu

```
void tiskni(int x)
{
  printf("%+06d",x);
}
```

- Funkce, která vrací **int** (v C)

```
moje_funkce(int x)
{
  …
}
```

- procedura/funkce bez parametrů

```
void tiskni_podtrzeni(void)
{
  printf("---------------");
}
```

Rozdíly v C a C++

- **v C++** se musí explicitně deklarovat vždy návratový typ
  - `moje_funkce(int x)` způsobí chybu při překladu v C++
- v C++ funkce musí vracet hodnotu (**`return`** nesmí chybět) – chyba při překladu, v C pouze varování
- v C++ musí existovat prototyp funkce

Prototyp funkce (hlavička)

- deklarace

```
type identifikátor(parametry);
```

  bez těla funkce

- prototypy jsou typicky v hlavičkových souborech

```
int obsah(int x, int y);
void tiskni(int x);
void tiskni_podtrzeni(void);
```

- prototyp v C

```
void tiskni_podtrzeni();
```

  znamená, že o parametrech se nic neříká

- prototyp v C

```
void print_underlines(void);
```

  znamená, že funkce parametry nemá

- **oba prototypy v C++ znamenají to samé: funkce parametry nemá**

- v jazyce C se přeloží (objeví se pouze varování, že je volána funkce sum bez prototypu), ale překlad v C++ skončí s chybou:

```
void main(void)
{ int x;
  x= sum(3,4);
}
int sum(int a, int b)
{
  return a+b;
}
```

- v jazyce C++ musím zapsat takto:

```
int sum(int a, int b)
{
  return a+b;
}
void main(void)
{ int x;
  x= sum(3,4);
}
```

- nebo takto:

```
int sum(int a, int b); //prototyp

void main(void)
{ int x;
  x= sum(3,4);
}

int sum(int a, int b)
{
  return a+b;
}
```

Předávání parametrů funkcím

Otázka:

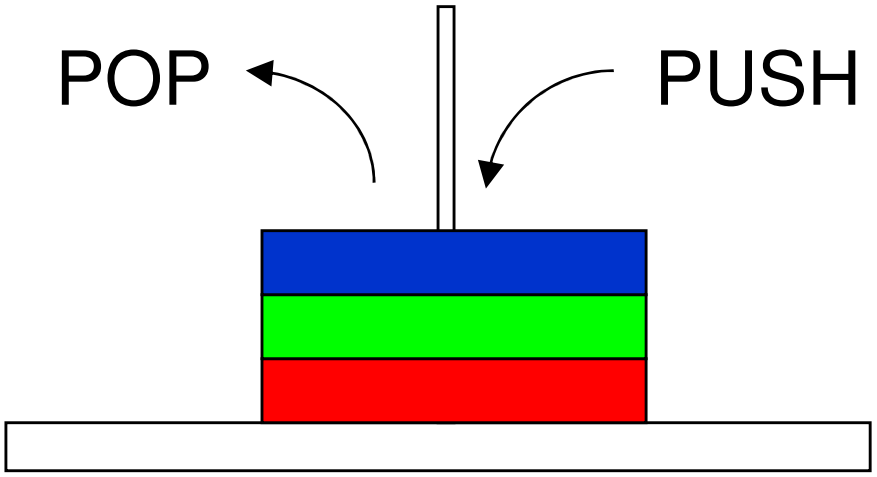
Jak se předávají skutečné parametry funkcí?

Odpověď: Přes **zásobník**.

- v jazyce C se parametry předávají jen hodnotou, tj. kopií na zásobník
- „výstupní“ parametry musíme realizovat pomocí ukazatelů

Zásobník (stack)

- datová struktura LIFO (last in – first out)
- odebírá se naposledy uložená hodnota
- operace:
  - PUSH (uložení hodnoty na vrchol zásobníku)
  - POP (odebrání hodnoty z vrcholu zásobníku)



- zásobník se využívá v programech také pro odložení mezivýsledků, automaticky se využívá při volání procedur a funkcí
- instrukce CALL *adr*
  - automaticky se uloží na vrchol zásobníku návratová adresa, provede se skok na adresu *adr*
- instrukce RET
  - návratová adresa se odebere ze zásobníku, na ní se provede návrat

```
int obsah(int a,int b)
{
  return a*b;
}
```

funkce obsah

1234: ...

1240: MUL EAX,EBX

1242: RET

```
void main(void)
{
  x = obsah(3,4);
}
```

hlavní program

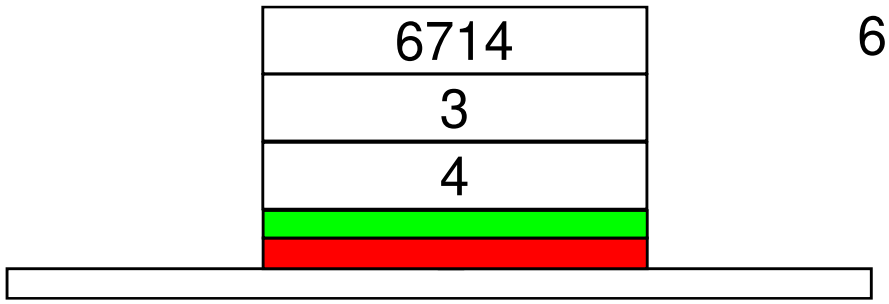
6709: push 4

6710: push 3

6711: call 1234

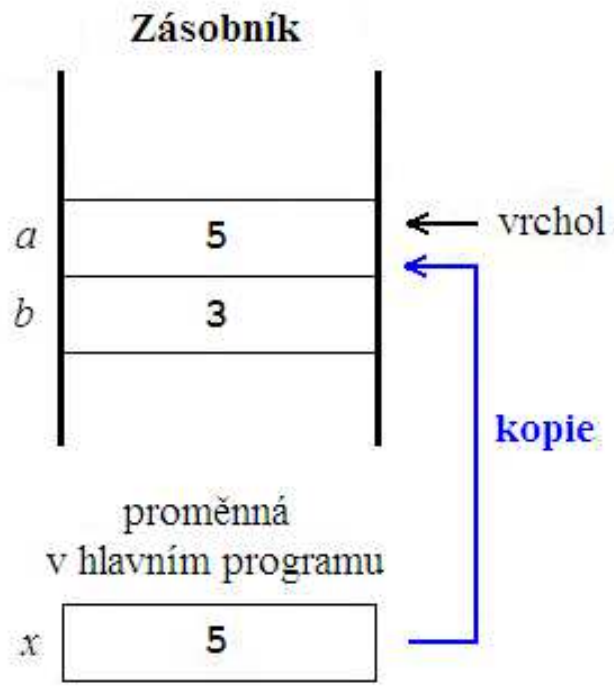
6714: pop

6716: pop

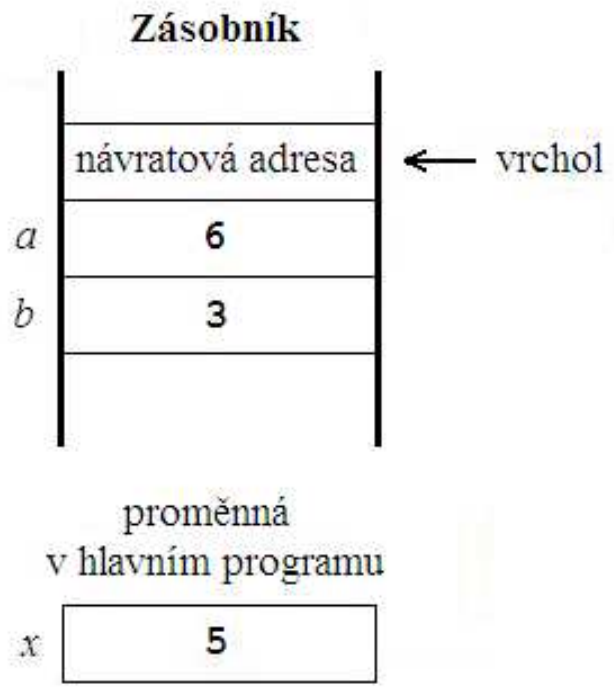


```
void tisk(int a, int b)
{  a++;
   printf("%d\t%d",a,b);
 }
void main(void)
{  int x = 5;
   tisk(x,3);
}
```

Před voláním procedury



Před ukončením procedury



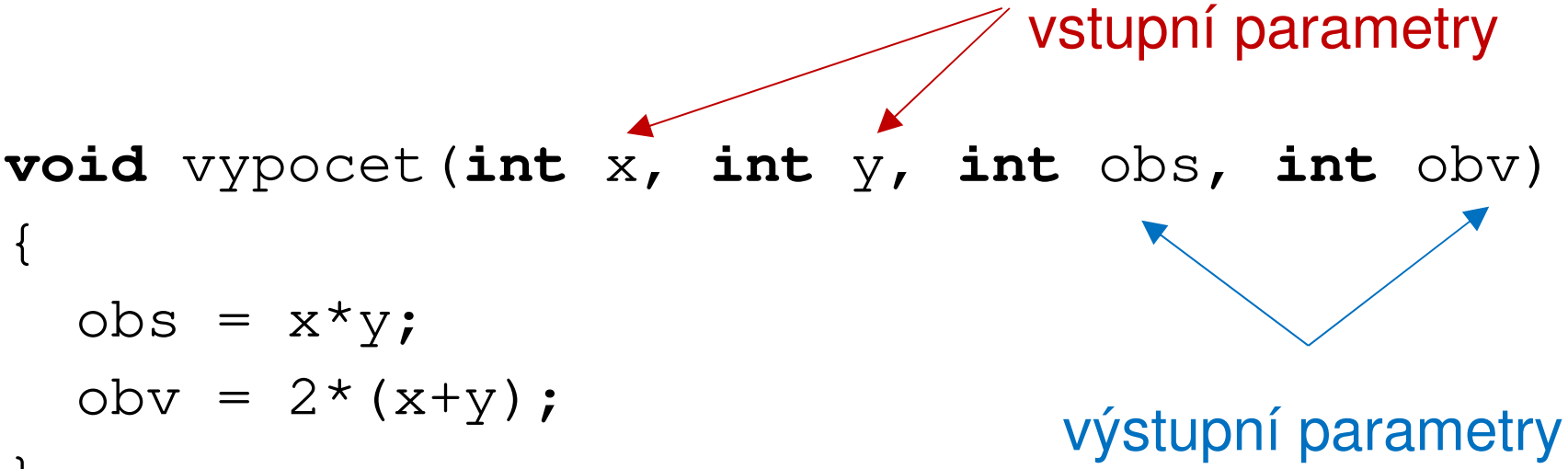
Naprogramujeme dvě funkce pro výpočet obsahu a obvodu obdélníka:

```
int obsah(int x, int y)
{
  return x*y;
}

int obvod(int x, int y)
{
  return 2*(x+y);
}
```

Chtěli bychom naprogramovat pouze jeden podprogram (funkci, proceduru), která vypočítá obě hodnoty najednou

- ale funkce vrací pouze jednu hodnotu

- **řešení:** výstupní parametry procedury

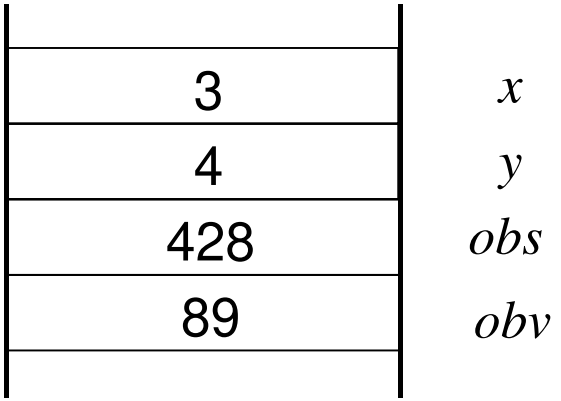


Je to dobře? Není!

```
void vypocet(int x, int y, int obs,
  int obv)
{
   obs = x*y;
   obv = 2*(x+y);
}

void main(void)
{
   int obsah,obvod;
   vypocet(3,4,obsah,obvod);
}
```

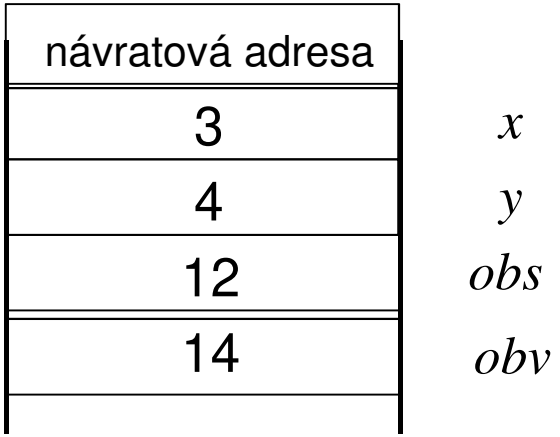
```
void vypocet(int x, int y,
        int obs, int obv)
{
  obs = x*y;
  obv = 2*(x+y);
}

void main(void)
{
  int obsah,obvod;
  vypocet(3,4,obsah,obvod);
}
```

Před voláním procedury



obsah: 428 obvod: 89

Před ukončením procedury



obsah: 428 obvod: 89

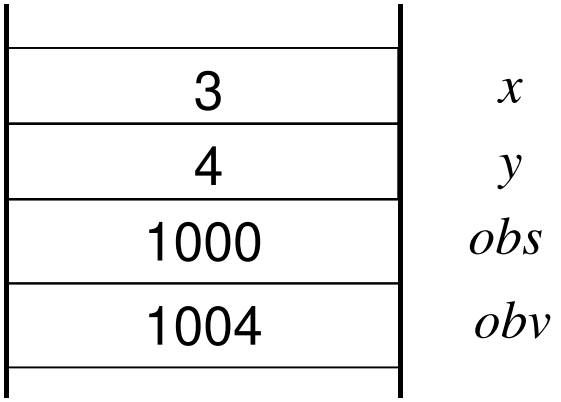
- je třeba, aby procedura výpočet nepracovala s kopiemi parametrů *obsah*, *obvod* na zásobníku, ale s "originálem", tj. s proměnnými *obsah, obvod*
  - na zásobník se musí předat nikoliv kopie hodnoty proměnné, ale odkaz na skutečné parametry, tj. odkaz na proměnné *obsah, obvod*
    - odkaz = adresa proměnných, kde jsou umístěny v paměti
  - některé programovací jazyky (Pascal) mají možnost definovat tzv. parametry volané odkazem
  - jazyk C tuto možnost nemá, realizuje se parametrem typu ukazatel

```
void vypocet(int x, int y, int *obs, int *obv)
{
  *obs = x*y;
  *obv = 2*(x+y);
}

void main(void)
{
  int obsah, obvod;

  vypocet(3,4,&obsah,&obvod);
}
```

```
void vypocet(int x, int y,
             int *obs, int *obv)
{
   *obs = x*y;
   *obv = 2*(x+y);
}

void main(void)
{
   int obsah,obvod;

vypocet(3,4,&obsah,&obvod);
}
```

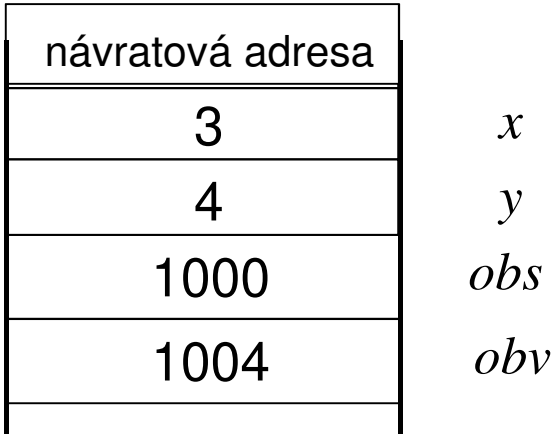
1000 a 1004 jsou adresy proměnných obsah, obvod v paměti počítače

Před voláním procedury



1000: obsah: 428 1004 obvod: 89

Před ukončením procedury



1000: obsah: 12 1004 obvod: 14

Poznámka

- lokální proměnné funkcí jsou také alokovány na zásobníku v momentě vstupu do funkce (tzv. proměnné ve třídě auto – automatické proměnné)

- v jazyce C++ je zaveden typ **reference**, který umožňuje psát v C++ výstupní parametry funkcí jednodušeji, podobně jako např. v Pascalu parametry volané odkazem (var parametry)

Typ reference

- proměnná typu reference na *typ T* je synonymem pro jinou proměnnou

```
int  i = 0;
int& ir = i;          // ir ~ i
ir = 2;               // i = 2
```

- proměnná typu reference na *T* musí být inicializovaná, a to proměnnou typu *T*

```
int &ir;  // chyba, ir není inicializována
```

```
float f;
const int ci = 10;

int &ir1 = f;   // chyba, f není typu int
int &ir2 = ci;  // chyba, ci je konstanta
```

- proměnná typu reference po celou dobu existence referencuje stejnou proměnnou (referencuje = odkazuje se na ni)
- typ reference se používá pro parametry procedur a funkcí (nahrazuje parametry volané odkazem)

Konstanta typu reference

- konstanta typu reference na *T* ~ synonymum pro neměnnou hodnotu typu *T*
- konstanta typu reference na *T* může být inicializována konstantou nebo proměnnou typu *T* nebo typu *T1*, jestliže *T1* je kompatibilní vzhledem k přiřazení s *T*. Ve druhém případě se vytvoří dočasný objekt, do kterého se zkonvertuje hodnota daná inicializačním výrazem

```
const int max = 100;
float f = 3.14;
const int& rmax = max;
const int& rf = f;
```
//referencuje se dočasný objekt s hodnotou 3

```
rmax = 10;
```
// chyba, rmax je konstanta

```
rf = 5;
```
// chyba, rf je konstanta

```
int& rmax1 = max;
```
// chyba, rmax1 není konstanta

```
int& rf1 = f;
```
// chyba, rf1 není konstanta

Parametr typu reference

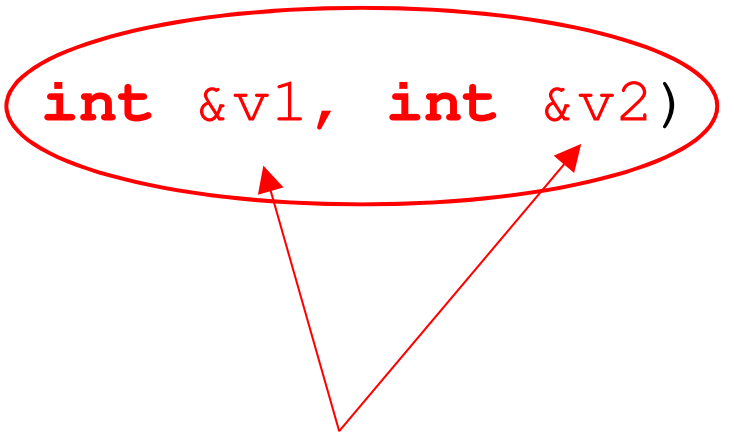
- v procedurách a funkcích nahrazuje parametry volané odkazem
- parametr typu reference představuje ve funkci synonymum *skutečného* parametru a umožňuje hodnotu *skutečného* parametru *změnit*

- styl C pomocí ukazatelů

```
void vypocet1(int a, int b, int *v1, int *v2)
{
   *v1 = a*b;
   *v2 = 2*(a+b);
}
```

- styl C++ pomocí parametrů typu reference

```
void vypocet2(int a, int b, int &v1, int &v2)
{
   v1 = a*b;
   v2 = 2*(a+b);
}
```



```
void main(void)
{
  int obsah, obvod;

  vypocet1(3,4,&obsah,&obvod);
  vypocet2(3,4,obsah,obvod);
}
```

Pole jako parametr funkcí

- pole jsou předávána jako ukazatel na počátek
  - počet prvků se musí předat jako zvláštní parametr
- deklaraci lze provést dvěma způsoby:

```
int maxpole(int n, int *pole)
{  int i;
   int max = pole[0];
   for(i=0;i<n;i++)
    if (pole[i]>max) max = pole[i];
   return max;
}
```

```
int maxpole(int n, int pole[])
{  int i;
   int max = pole[0];
   for(i=0;i<n;i++)
    if (pole[i]>max) max = pole[i];
   return max;
}

void main(void)
{  int maximum;
   int p[10]={5,4,8,7,15,5,9,8,7,10};
   maximum = maxpole(10,p);
   }
```

Dvoudimensionální pole

- statické:

```
int maxpole(int n, int m, int arr[][20])
{
   …
}
```

- dynamické:

```
int maxpole(int n, int m, int **arr)
{
   …
}
```

Přetěžování funkcí v C++

- v klasickém jazyku C se funkce rozlišují jen identifikátorem, tj. nelze definovat více funkcí se stejným identifikátorem, ale jinými parametry
- nevýhoda - existence různě pojmenovaných podobných funkcí, např.:

```
int abs(int x);
double fabs(double x);
```

- v C++ můžeme funkci *přetížit*, tj. rozlišit i počtem a typem parametrů

- tj. lze deklarovat funkci se stejným jménem, ale různými parametry (různými typy, s různým počtem)

```
int abs(int x);

double abs(double x);
```

- *funkci nelze přetížit* jen návratovou hodnotou - proč?

```
void tisk(int x)
{
  printf("%d",x);
}

void tisk(char x)
{
  printf("%c",x);
}
```

- při volání přetížené funkce se vyvolá ta, jejíž parametry se nejlépe spárují se skutečnými parametry
- párování (matching) parametrů:
  1. přesná shoda typů
  2. shoda typů po roztažení (promotion)

     char $\rightarrow$ int
     short $\rightarrow$ int
     enum $\rightarrow$ int
     float $\rightarrow$ double

3. shoda typů po standardní konverzi

int $\rightarrow$ float
float $\rightarrow$ int
int $\rightarrow$ unsigned
…
int $\rightarrow$ long

4. shoda typů po uživatelské konverzi (přetypování)

- pokud nejlepší párování má více než jedna funkce, ohlásí se chyba (při překladu) - potíže nastávají hlavně, jsou-li skutečným parametrem konstanty

Příklady:

```
void f(float, int);
void f(int, float);
f(1,1);                                 // chyba
f(1.2,1.3);                             // chyba

void f(long);
void f(float);
f(1.1);                                         // chyba
f(1.1F);                                        //
f(float)

void f(unsigned);
void f(int);
void f(char);
unsigned char uc;
f(uc);                                          // f(int)
```