

# Struktury, funkce, reference

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Přednášky byly připraveny i s pomocí materiálů, které vyrobili Ladislav Vágner, Pavel Strnad a Martin Mazanec

© Karel Richta, Martin Hořeňovský, Aleš Hrabalík, 2022

Programování v C++, B6B36PJC

10/2022, Lekce 3

<https://cw.fel.cvut.cz/wiki/courses/b6b36pjc/start>



# Reprezentace složitějších dat

- Dosud jsme pracovali se základními datovými typy.
- Způsob uložení triviálních informací není v C a C++ určen striktně, jen jsou stanovena určitá pravidla a závisí na počítači a operačním systému.
- Nyní se budeme zabývat strukturováním dat.

## **Odvozené (strukturované) typy:**

- pole
- struktura (záznam)
- třída
- union (sjednocení)

# Odvozené (strukturované) typy

- **Pole**
  - jednorozměrná pole prvků libovolného typu (kromě funkce)
  - indexováno vždy od 0
- **Struktura** (záznam)
  - obsahuje pojmenované položky různých typů uložené za sebou, implicitně přístupné
- **Třída**
  - obsahuje pojmenované položky různých typů uložené za sebou, implicitně nepřístupné zvenku
- **Union** (sjednocení)
  - pojmenované položky různých typů uložené "přes" sebe

# Pole

- Homogenní složený objekt, jeho prvky jsou v paměti uloženy za sebou a odkazovány číselnými indexy.

```
int chlebicky[35];
```

```
float kremrole[35];
```

- K jednotlivým prvkům přistupujeme pomocí operace [ ] (indexace pole).
- Rozsah indexů je od 0 do N-1, kde N je velikost pole – počet prvků v poli.

```
chlebicky[0] = 15;
```

```
kremrole[34] = chlebicky[0];
```

- Vícerozměrná pole se vytváří jako pole polí.

```
char pohary[7][5];
```

```
pohary[6][4] = 'c';
```

# Pole (pokračování)

- Pole prvků typu T se implicitně konvertuje na T\*, tj. ukazatel na T. Toho se často využívá k předávání polí (např. předání pole funkci).
- Operátor indexace pole lze použít i na ukazatele.

```
void fce(int* chleb, float* krem) {  
    chleb[0] = 15;  
    krem[34] = chleb[0];  
}
```

```
int main() {  
    int chlebicky[35];  
    float kremrole[35];  
    fce(chlebicky, kremrole);  
}
```

# Příklady na inicializaci polí

```
int P[3] = { 1, 2, 3 };
/* inicializací lze zadat počet prvků pole */
int Q[] = { 1, 2, 3 };
int PP[4][3] = {{1, 3, 5}, {2, 4, 6}, {3, 5, 7}, {4, 6, 8}};
/* následující inicializace má stejný efekt */
int PP[4][3] = { 1, 3, 5, 2, 4, 6, 3, 5, 7, 4, 6, 8 };
/* inicializace nemusí být úplná */
int QQ[][3] = { { 1 }, { 2 }, { 3 }, { 4 } };
/* pole má 4 řádky, explicitně inicializován je jen 1. sloupec;
ostatní jsou vynulovány - chybějící prvky se vždy vynulují */
char Msg[] = "text";
char Msg[5] = { 't', 'e', 'x', 't', '\0' }; /* totéž */
```

# Odbočka: konstantní výrazy

- Konstantní výraz je výraz vyhodnotitelný v době překladač
- Nesmí obsahovat vedlejší efekty
- Použití:
  - počet prvků pole
  - inicializace statických objektů
  - návěští v přepínači atd.

```
int pole[10];           // 10 je konstanta
int pole[5 + 7];       // součet konstant je konstanta
const int velikost = 20;
int pole[velikost];    // v pořádku, velikost je const
int pole[velikost + 10]; // OK, výraz je const
```

# Odbočka: konstantní výrazy (2)

- Konstantní výraz je výraz vyhodnotitelný v době překladu
- Nesmí obsahovat vedlejší efekty
- Použití:
  - počet prvků pole
  - inicializace statických objektů
  - návěští v přepínači atd.

```
int pole[6 - 7]; // chyba, konstanta je záporná
int velikost2 = 20;
int pole[velikost2]; // chyba, velikost2 není const
int get_size() { ... }
int pole[get_size()]; // chyba, výsledek funkcí není známý
                       // během kompilace
char Msg[4] = "text"; // chyba, nevejde se konečná \0
```



# Struktury

- Syntaxe:

`struct` značka { seznam popisů položek };



Středník je  
nutný!

- Popisy položek mají podobný tvar jako deklarace
  - struktura nesmí obsahovat sama sebe (objekt svého typu)
  - struktura smí obsahovat ukazatel na objekt svého typu

- Příklad:

```
struct Osoba {  
    std::string jmeno;  
    std::string prijmeni;  
    int rok_narozeni;  
}; // definice struktury  
Osoba Jan;  
// deklarace proměnné typu Osoba
```

# Příklady struktur

- struktura pro binární strom nebo dvousměrně zřetězený seznam:

```
struct TNode {  
    std::string slovo;  
    int pocet;  
    TNode *levy, *pravy;  
};
```

- anonymní struktura:

```
struct { int prvni, druha; } p1[9], *p2;
```

- vzájemně odkazované struktury:

```
struct S1; /* neúplná deklarace, lze použít jen pro  
vytvoření ukazatelů a referencí */  
struct S2 { int obsah; S1* dalsi; };  
struct S1 { long obsah; S2* dalsi; };
```

## Příklad: Tabulka zaměstnanců

```
const int MAXZAM = 30;

struct Osoba {
    unsigned int ID;
    std::string jmeno;
    std::string prijmeni;
    double plat;
}; // definice struktury Osoba

Osoba tab[MAXZAM]; // tabulka zaměstnanců
```

## Příklad: Inicializace tabulky

```
const int MAXZAM = 30;

struct Osoba {
    unsigned int ID;
    std::string jmeno;
    std::string prijmeni;
    double plat;
}; // definice struktury Osoba

Osoba tab[MAXZAM] = {
    { 1, "Karel", "Novák", 10000.0 },
    { 2, "Jana", "Nováková", 8000.0 },
    { 3, "Dáša", "Poláková", 14000.0 },
}; // tabulka zaměstnanců
```

# Selekce

- Zpřístupnění položky struktury, třídy nebo unionu.
- Přímá selekce:  
`X . položka` kde X je výraz typu **struct**, **class** nebo **union**
- Nepřímá selekce (přes ukazatel):  
`X -> položka` kde X je výraz typu ukazatel na **struct**, **class** nebo **union**
- `X -> položka` je zkratkou za `(*X) . položka`
- Příklad:

```
struct { int a; char b; } x, *px = &x;  
x.a = 1;  
x.b = 'a';  
px->a = 4;  
px->b = 'd';  
(*px).b = 'd'; // totéž
```

# Výčtový typ

- Syntaxe:

```
enum značka { seznam literálů };
```

- Příklad:

```
enum Color { Red, Green, Blue };
```

- Literály jsou synonyma celočíselných hodnot

```
/* Red = 0, Green = 1, Blue = 2 */
```

- Takto zavedená jména jsou globální konstanty – nelze tedy v různých výčtech používat stejná jména.

# Výčtový typ (pokračování)

- Literálu lze explicitně přiřadit hodnotu:

```
enum Masky { Nula, Jedna, Dva, Ctyri = 4, Osm = 8 };  
/* Dva = 2, Ctyri = 4, Osm = 8 */
```

```
enum Sign { Minus = -1, Zero, Plus };  
/* Minus = -1, Zero = 0, Plus = 1 */
```

```
enum SelfRef { E1, E2, E3 = 5, E4, E5 = E4 + 10, E6 };  
/* E4 = 6, E5 = 16, E6 = 17 */
```

- Výčtový typ je (obvykle) kompatibilní s `int` i co do délky vnitřní reprezentace:

```
enum Sign s; /* totéž co int s; */
```

# Výčtový typ jako třída

- V C++ je v zájmu bezpečnosti možno zavést výčtový typ jako třídu:

```
enum class Colors { red, blue, green };
```

- Hodnoty typu **Colors** nejsou kompatibilní s typem **int**, ani implicitně konvertované na **int** – jsou to prostě hodnoty typu **Colors**.
- Každá hodnota výčtové třídy se musí uvádět spolu s kvalifikátorem:

```
Colors mycolor;
```

```
mycolor = Colors::blue;
```

```
if (mycolor == Colors::green) mycolor = Colors::red;
```

- Hodnoty výčtových tříd mohou být reprezentovány různými základními typy, dle rozsahu výčtu. Např.:

```
enum class EyeColor : char { blue, green, brown };
```

- Pak **EyeColor** je typ se stejnou velikostí jako **char** (1 byte) (ale nikoliv kompatibilní s **char**).



# 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

# 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
```

# Volání funkce

- Syntaxe:

**$F ( X_1, X_2, \dots X_n )$**

kde  $F$  je identifikátor funkce (nebo ukazatel na funkci) a  $X_1, \dots X_n$  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 typů 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);
```

# 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;  
}
```

# 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 vrátí nejvýše jeden výsledek – funkční hodnotu (zpravidla přes zásobník). Pokud vrátí **void** – nevrací nic (je to procedura).
- Pokud chceme, aby funkce vracela více hodnot, můžeme použít `std::tuple`, nebo 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 disciplínu.

# 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

- 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, můžeme 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 – musí to být l-hodnota).



# Příklad: Parametry typu reference

```
void mul_by_2(int& a, int& b, int& c) {  
    a *= 2;  
    b *= 2;  
    c *= 2;  
}  
  
int main() {  
    int x = 1, y = 3, z = 7;  
    mul_by_2(x, y, z); // po provedeni plati  
                        // x == 2, y == 6, z == 7  
    mul_by_2(2, y, z); // toto se nezkompile,  
                        // 2 není lvalue, ale rvalue  
}
```

# 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);
```

# 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 změna 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

- Příklad – spojení řetězců:

```
#include <string>
using namespace std;
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 a. To samé platí pro b.
- U parametrů volaných referencí se místo toho pracuje s původním řetězcem. Výhodou je, že nedochází k potenciálně nákladným kopiím:

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

- Protože funkce s parametry volanými referencí mohou modifikovat své parametry, je bezpečnější použít:

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

# 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 <stdio>

void foo(int x, int y) {}

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

Ve Windows vytiskne:  
barfoo  
Na Linuxu:  
foobar

# Implicitní hodnoty parametrů

```
#include <iostream>
```

```
int divide(int a, int b = 2) { // b má implicitní hodnotu 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

```
// výpočet faktoriálu
```

```
#include <iostream>
```

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

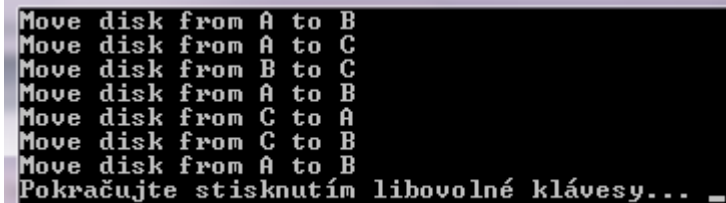
```
int main() {  
    long number = 9;  
    std::cout << number << "! = " << factorial(number);  
    return 0;  
}
```



# 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;
}
```



```
Move disk from A to B
Move disk from A to C
Move disk from B to C
Move disk from A to B
Move disk from C to A
Move disk from C to B
Move disk from A to B
Pokračujte stisknutím libovolné klávesy... _
```

# 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. 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;

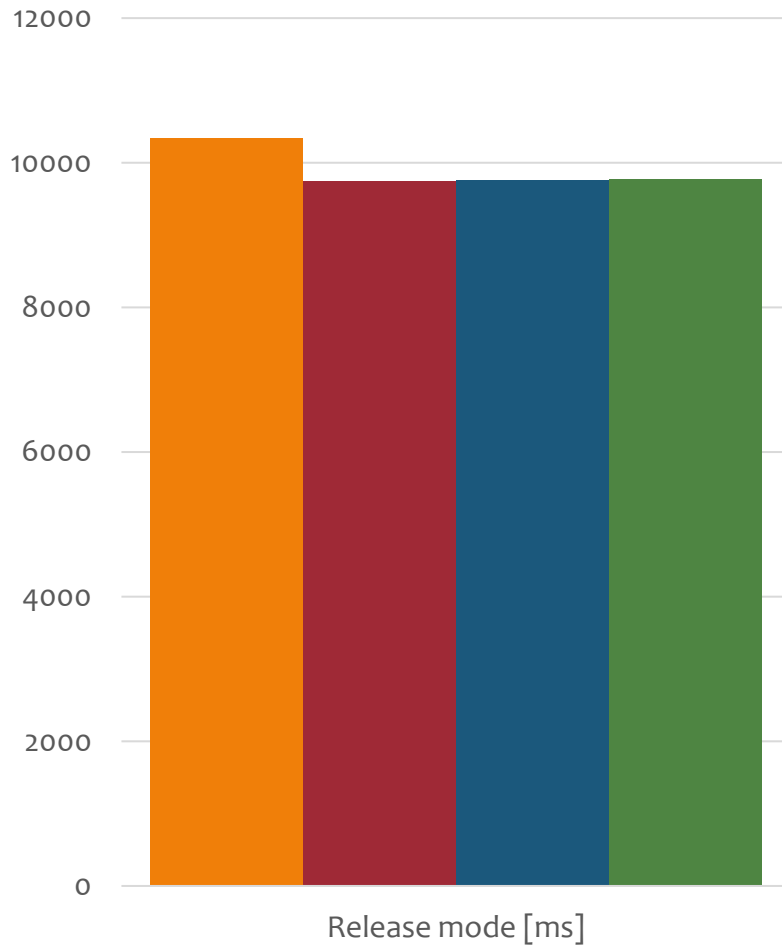
double tmdiff(chrono::time_point<clk> t) {
    return chrono::duration<double, milli>(clk::now() - t).count();
}

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

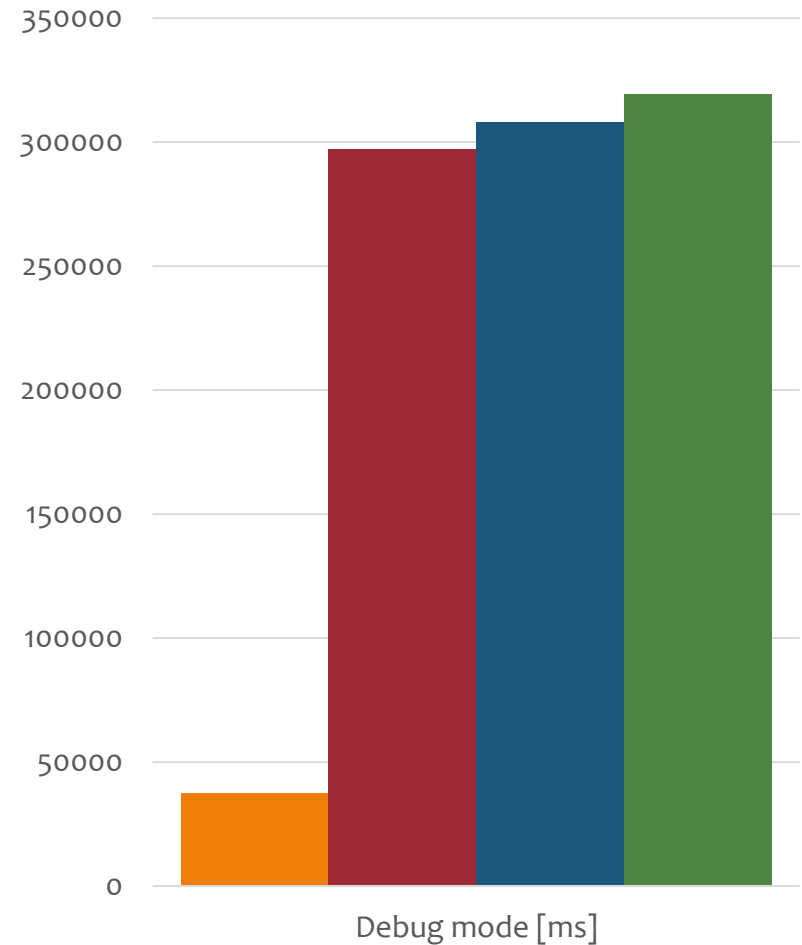
inline ll max1(ll x, ll y) { return x > y ? x : y; }
ll max2(ll x, ll y) { return x > y ? x : y; }
ll max3(const ll& x, const ll& y) { return x > y ? x : y; }

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



- Makro
- Inline funkce
- Funkce
- Funkce s referencemi



- Makro
- Inline funkce
- Funkce
- Funkce s referencemi

# 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 praxi.
- 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
<code>0</code>	úspěšný konec
<code>EXIT_SUCCESS</code>	úspěšný konec
<code>EXIT_FAILURE</code>	neúspěšný konec, chyba

**Konec**