

# Řetězce

## 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 a Martin Mazanec

© Karel Richta, 2015

Programování v C++, A7B36PJC

10/2015, Lekce 10

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



# Řetězce v C <string.h> (<cstring>)

```
size_t strlen(char *Retezec);
char *strcpy(char *Cil, char *Zdroj);
char *strncpy(char *Cil, char *Zdroj, size_t MaxDelka);
char *strcat(char *Cil, char *Zdroj);
char *strncat(char *Cil, char *Zdroj, size_t MaxDelka);
int strcmp(char *Prvni, char *Druhy);
int strncmp(char *Prvni, char *Druhy, size_t N);
char *strchr(char *Retezec, int Znak);
char *strrchr(char *Retezec, int Znak);
char *strstr(char *Retezec, char *Podretezec);
void *memcpy(void *Cil, void *Zdroj, size_t Delka);
void *memmove(void *Cil, void *Zdroj, size_t Delka);
int memcmp(void *Prvni, void *Druhy, size_t Delka);
void *memchr(void *Adresa, int Byte, size_t Delka);
void *memset(void *Adresa, int Byte, size_t Delka);
Makro: NULL - jako celočíselná konstanta (0 nebo 0L), nebo hodnota typu
        nullptr_t (nullptr) - (void*)0
Typ: size_t - typ pro délku libovolného objektu (jako unsigned int)
```

# Řetězce jako třída String s operátory

```
class String {
    int pocet; // aktuální délka řetězce
    char *znaky; // ukazatel na dyn. alokované pole
public:
    String(); // vytvoří prázdný řetězec
    String(const String&); // vytvoří řetězec jako kopii jiného
    String(const char *); // konverze z C-Stringu
    ~String(); // destruktork
    int delka() const; // vrátí délku řetězce
    String& operator=(const String&); // přiřazení řetězců
    friend ostream& operator<<(ostream&, const String&); // výstup řetězce do
        // streamu
    bool operator>=(const String&) const; // porovnání řetězců
    String operator+(const String&) const; // sřetězení
    operator const char *() const; // konverze na C-String
private:
    String(const String&, const String&); // podpora sřetězení
};
```

# Třída String s operátory

- přiřazení:

```
String& String::operator=(const String& p) {  
    if (this != &p) {  
        delete [] znaky;  
        pocet = p.pocet;  
        znaky = new char[pocet+1];  
        strcpy(znaky, p.znaky);  
    }  
    return *this;  
}
```

- výstup do datového proudu:

```
ostream& operator<<(ostream& str, const String& p) {  
    return str << p.znaky;  
}
```

# Třída String s operátory

- konverze na C-string:

```
String::operator const char *() const {  
    return znaky;  
}
```

- zpřístupnění i-tého znaku (pro čtení):

```
char String::operator[](int i) const {  
    if (i < 0 || i >= pocet) throw ChybnyIndex();  
    return znaky[i];  
}
```

- zpřístupnění i-tého znaku (pro změnu):

```
char& String::operator[](int i) {  
    if (i < 0 || i >= pocet) throw ChybnyIndex();  
    return znaky[i];  
}
```

# Třída String s operátory

- použití:

```
int main() {
    String A, B("abc"), C(B);
    A = B; // přiřazení
    C = A + "de"; // sřetězení, konverze
    if (C >= B) // porovnání
        cout << C << " je větší nebo rovno " << B;
    else
        cout << C << " je menší než " << B;
    cout << endl << "Délka " << C << " je: " << C.delka() << endl;
    return 0;
}
```

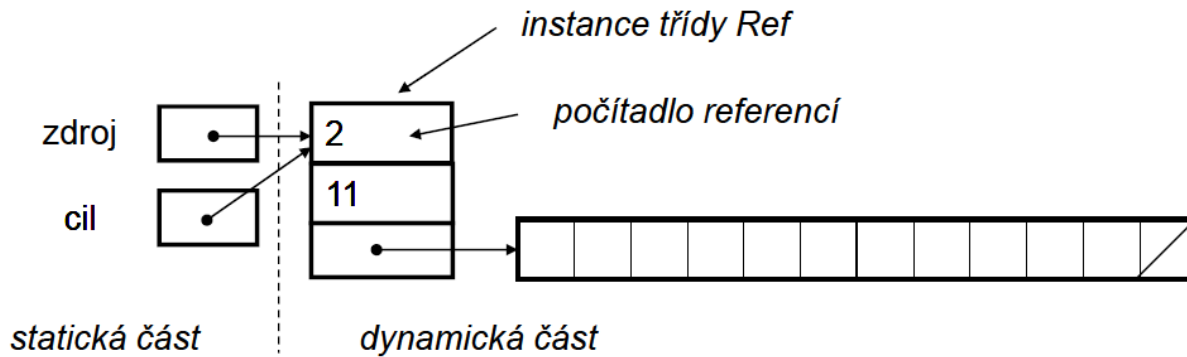
Výstup:

abcde je větší nebo rovno abc

Délka abcde je: 5

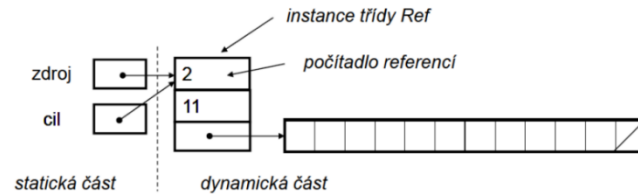
# Počítané reference

- Sdílení dynamické části:  
je třeba vyřešit dva problémy: modifikaci a destrukci  
**cil = zdroj;**



# Počítané reference

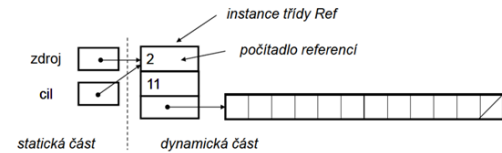
```
class Ref {  
public:  
    int reference;  
    int pocet;  
    char *znaky;  
public:  
    Ref(const char *str) {  
        reference = 1; pocet = strlen(str);  
        znaky = new char[pocet+1];  
        strcpy(znaky, str);  
    }  
    Ref(int delka) {  
        reference = 1; pocet = delka;  
        znaky = new char[pocet+1];  
    }  
    ~Ref() {  
        delete [] znaky; pocet = 0;  
        reference = 0; znaky = 0;  
    }  
};  
Ref prazdny("");
```





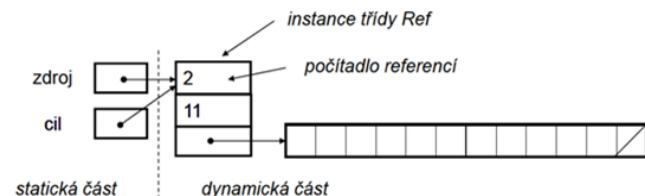
# Počítané reference

```
class String {
    Ref *ref;
    String(const String& l, const String& p); // konstruktor pro sretezeni
public:
    String() { // prazdny retezec
        ref = &prazdny; prazdny.reference++;
    }
    String(const char *str) { // C-string
        ref = new Ref(str);
    }
    String(const String& str) { // kopie jineho retezce
        ref = str.ref; ref->reference++;
    }
    ~String() { // destruktork
        ref->reference--;
        if (ref->reference == 0) { delete ref; ref = 0; }
    }
    int delka() const { // délka řetězce
        return ref->pocet;
    }
    operator const char *() const { // konverze na C-string
        return ref->znaky;
    }
}
```



# Počítané reference

```
String& operator=(const String& p) { // přiřazení
    if (ref != p.ref) {
        this->~String(); // explicitní volání destrukturu!
        ref = p.ref;
        ref->reference++;
    }
    return *this;
}
friend String operator+(const String& l, const String& p) { // sřetězení
    return String(l, p);
}
char operator[](int i) const; // vrať i-tý znak - později
void zmen(int i, char nový); // změň i-tý znak - později
bool operator>=(const String& p) const { // větší nebo rovno
    return strcmp(ref->znaky, p.ref->znaky) >= 0;
}
bool operator==(const String& p) const { // rovnost
    return strcmp(ref->znaky, p.ref->znaky) == 0;
}
};
class ChybnýIndex {}; // třída pro výjimku
```



# Počítané reference

```
String::String(const String& l, const String& p) {
    ref = new Ref(l.ref->pocet + p.ref->pocet);
    strcpy(ref->znaky, l.ref->znaky);
    strcat(ref->znaky, p.ref->znaky);
}

char String::operator[](int i) const {
    if (i < 0 || i >= ref->pocet) throw ChybnyIndex();
    return ref->znaky[i];
}

void String::zmen(int i, char novy) {
    if (i < 0 || i >= ref->pocet) throw ChybnyIndex();
    if (ref->reference > 1) {
        ref->reference--;
        ref = new Ref(ref->znaky); // vytvoříme dynamickou kopii
    }
    ref->znaky[i] = novy;
}
```

# Vektory

- Příklad: třída pro 3D vektorový prostor nad reálnými čísly

```
const int dim = 3;
typedef double Skalar;
class Vektor {
    Skalar slozky[dim];
public:
    Vektor();
    Vektor(Skalar x, Skalar y, Skalar z);
    Vektor operator-() const;
    Vektor operator+(const Vektor& p) const;
    Skalar operator*(const Vektor& p) const;
    Skalar operator[](unsigned index) const;
    Skalar& operator[](unsigned index);
    Skalar delka() const;
    friend Vektor operator*(Skalar k, const Vektor& v);
    friend ostream& operator<<(ostream&, const Vektor& v);
    bool operator==(const Vektor& p) const;
};
```

# Vektory (pokr.)

- konstruktor prázdného vektoru:

```
Vektor::Vektor() {  
    for (int i = 0; i < dim; i++)  
        slozky[i] = 0;  
}  
  
Vektor::Vektor(Skalar x, Skalar y, Skalar z) {  
    slozky[0] = x; slozky[1] = y; slozky[2] = z;  
}
```

- opačný vektor:  $-\vec{u}$  unární operátor => bez parametrů

```
Vektor Vektor::operator-() const {  
    Vektor vysledek;  
    for (int i = 0; i < dim; i++) vysledek.slozky[i] = -slozky[i];  
    return vysledek;  
}
```

# Vektory (pokr.)

- součet vektorů:

```
Vektor Vektor::operator+(const Vektor& p) const {  
    Vektor vysledek;  
    for (int i = 0; i < dim; i++)  
        vysledek.slozky[i] = slozky[i] + p.slozky[i];  
    return vysledek;  
}
```

- skalární součin:

```
Skalar Vektor::operator*(const Vektor& p) const {  
    Skalar vysledek = 0;  
    for (int i = 0; i < dim; i++)  
        vysledek += slozky[i] * p.slozky[i];  
    return vysledek;  
}
```

# Vektory (pokr.)

- indexace (i-tá složka):

```
Skalar Vektor::operator[](unsigned index) const {  
    if (index >= dim) throw "chybny index";  
    return slozky[index];  
} // použije se na konstantní vektory
```

```
Skalar& Vektor::operator[](unsigned index) {  
    if (index >= dim) throw "chybny index";  
    return slozky[index];  
} // pro nekonstantní vektory; umožňuje modifikaci
```

- velikost vektoru (délka):

```
Skalar Vektor::delka() const {  
    return sqrt(*this * *this); // sqrt vyžaduje #include <cmath>  
}
```

# Vektory (pokr.)

- násobení vektoru skalárem:

```
Vektor operator*(Skalar k, const Vektor& v) {  
    Vektor vysledek;  
    for (int i = 0; i < dim; i++) vysledek.slozky[i] = k * v.slozky[i];  
    return vysledek;  
}
```

- výstup vektoru:

```
ostream& operator<<(ostream& str, const Vektor& v) {  
    str << "[" << v.slozky[0];  
    for (int i = 1; i < dim; i++) str << "," << v.slozky[i];  
    return str << "];"  
}
```



# Vektory (pokr.)

- použití:

$$\vec{b}_1 = [1, 0, 0], \quad \vec{b}_2 = [0, 1, 0], \quad \vec{b}_3 = [0, 0, 1],$$
$$\vec{x} = -\vec{b}_1 + 3\vec{b}_2 + 2\vec{b}_3, \quad \vec{y} = -\vec{x}$$

```
int main() {
    const Vektor b1(1, 0, 0), b2(0, 1, 0), b3(0, 0, 1);
    Vektor x = -b1 + 3 * b2 + 2 * b3;
    Vektor y = -x;
    y[2] = 5;
    cout << "Vektory x=" << x << " a y=" << y;
    if (x * y == 0) cout << " jsou kolmé." << endl;
    else cout << " nejsou kolmé." << endl;
    return 0;
}
```

Výstup:

Vektory x=[-1,3,2] a y=[1,-3,5] jsou kolmé.

# Knihovna `<string>` pro C++

- Dva základní typy:
  - `basic_string` - generická šablona pro řetězce
  - `char_traits` - generická šablona pro množinu znaků
- Generické instance `char_traits`:
  - `char` - základní znaková sada (velikosti 1 byte)
  - `wchar_t` - sada širokých znaků (stejná velikost, znaménko a zarovnání, jako jiný celočíselný typ)
  - `char16_t` - reprezentuje 16-bitové kódové jednotky (stejná velikost, znaménko a zarovnání, jako celočíselný typ `uint_least16_t`)
  - `char32_t` - reprezentuje 32-bitové kódové jednotky (stejná velikost, znaménko a zarovnání, jako jiný celočíselný typ `uint_least32_t`)
- Generické instance `basic_string`:
  - `string` - řetězce znaků (`char`)
  - `wstring` - řetězce širokých znaků (`wchar_t`)
  - `u16string` - řetězce 16-bitových znaků
  - `u32string` - řetězce 32-bitových znaků

# Konverze v knihovně <string>

## Konverze z řetězců **string**:

- stoi - konverze string na int
- stol - konverze string na long int
- stoul - konverze string na unsigned long
- stoll - konverze string na long long
- stoull - konverze string na unsigned long long
- stof - konverze string na float
- stod - konverze string na double
- stold - konverze string na long double

## Konverze na řetězce **string**:

- to\_string - konverze numerické hodnoty na string
- to\_wstring - konverze numerické hodnoty na string

# Iterátory v knihovně `<string>`

## Iterátory:

- `begin` – vrací iterátor na začátek
- `end` – vrací iterátor za konec
- `rbegin` – vrací iterátor na poslední znak (reverse beginning) – reverzní iterátory iterují pozpátku
- `rend` – vrací iterátor před první znak (reverse end)
- `cbegin` – vrací `const_iterator` na začátek
- `cend` – vrací `const_iterator` za konec
- `crbegin` – vrací `const_reverse_iterator` na poslední znak
- `crend` – vrací `const_reverse_iterator` před první znak

# Příklad

```
// string::crbegin/crend
#include <iostream>
#include <string>

int main ()
{
    std::string str ("lorem ipsum");
    for (auto rit=str.crbegin(); rit!=str.crend(); ++rit)
        std::cout << *rit;
    std::cout << '\n';

    return 0;
}
```

muspi merol

# Kapacita řetězců v knihovně `<string>`

- `size` – délka řetězce
- `length` – počet znaků
- `max_size` – maximální velikost řetězce
- `resize` – změna velikosti řetězce
- `capacity` – velikost alokované paměti
- `reserve` – požadavek na změnu kapacity
- `clear` – vymaže řetězec
- `empty` – test na prázdnost řetězce
- `shrink_to_fit` – nastaví kapacitu dle aktuální délky

# Příklad

```
// string::shrink_to_fit
#include <iostream>
#include <string>

int main ()
{
    std::string str (100,'x');
    std::cout << "1. capacity of str: " << str.capacity() << '\n';

    str.resize(10);
    std::cout << "2. capacity of str: " << str.capacity() << '\n';

    str.shrink_to_fit();
    std::cout << "3. capacity of str: " << str.capacity() << '\n';

    return 0;
}
```

1. capacity of str: 100
2. capacity of str: 100
3. capacity of str: 10

# Přístup k elementům v knihovně <string>

- **operator[]** – indexace
- **at()** – totéž
- **back()** – poslední znak
- **front()** – první znak

```
int main () {  
    std::string str ("Test string");  
    for (unsigned i=0; i<str.length(); ++i) {  
        std::cout << str.at(i);  
    }  
    cout << endl;  
    cout << str.front() << endl;  
    cout << str.back() << endl;  
}
```

```
Test string  
T  
g
```



# Další možnosti knihovny <string>

## Modifikátory:

- **operator+=**
- **append**
- **push\_back**
- **assign**
- **insert**
- **erase**
- **replace**
- **swap**
- **pop\_back**

## Práce s řetězci:

- **c\_str**
- **data**
- **get\_allocator**
- **copy**
- **find**
- **rfind**
- **find\_first\_of**
- **find\_last\_of**
- **find\_first\_not\_of**
- **find\_last\_not\_of**
- **substr**
- **compare**

## Další možnosti:

- **operator+**
- **relational operators**
- **swap**
- **operator>>**
- **operator<<**
- **getline**

**The End**

# Definice funkce - starý styl (K&R C)

- Hlavička obsahuje paměťovou třídu, typ návratové hodnoty, jméno funkce a seznam jmen formálních parametrů.
- Formální parametry typu `int` není třeba deklarovat:

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

- Volání funkcí definovaných starým způsobem se nekontroluje.
- Proto nepoužívat !!!

# Starý styl deklaráce funkce (K&R C)

- Starý styl deklaráce funkce v C (a jen v C) - bez specifikace parametrů - s **libovolným počtem parametrů**:

```
long F();
```

```
int *G();
```

- Nebezpečné a nepoužívat, neboť při volání takto deklarované funkce se nekontroluje počet a typy parametrů !
- **Implicitní deklaráce** - při volání nedeklarované funkce F se implicitně předpokládá deklaráce:

```
extern int F();
```

- Nebezpečné, vyhýbat se!