# Řetězce

## PB161 Programování v jazyce C++

Nikola Beneš

30. dubna 2024

# Reprezentace textu

## Kódování znaků

- zobrazení znaků na číselné hodnoty
  - případně na jednotky paměti (bajty)
- historicky spousta různých
  - ASCII, EBCDIC
  - ISO-8859-1, ISO-8859-2, ISO-8859-3, ...
- v moderním světě: *Unicode*
  - UTF-8, UTF16, UTF32, ...

# C++ a Unicode

## Podpora pro Unicode v C++

- nějaká ano, ale oproti jiným moderním jazykům velmi slabá
- na některých místech prakticky žádná
  - kategorie znaků
  - vstup / výstup
  - formátování
  - regulární výrazy
- snad to bude v budoucích standardech lepší...

## Důsledek

- na seriózní práci s texty použijte nějakou knihovnu třetí strany

# Co je to „znak“?

**Grafém** (*grapheme*)

- to, co čtenář považuje za základní grafický prvek písma

**Kódový bod** (*code point*) – atomický prvek textu v Unicode

- jeden grafém může být složen z více kódových bodů
- některé kódové body nejsou součástí grafémů

**Kódová jednotka** (*code unit*) – jednotka v konkrétním kódování

- UTF-8: 8 bitů, UTF-16: 16 bitů, UTF-32: 32 bitů
- jeden kódový bod může být složen z více kódových jednotek

**Bajt** (*byte*)

- nejmenší adresovatelná jednotka paměti
  - moderní architektury – typicky oktet, tj. 8 bitů

# Znaky v C++

`char`

- jeden bajt; znaménkovost platformově závislá
- `signed` a `unsigned` varianty jsou samostatné typy

`wchar_t`

- „široký znak“; platformově závislý typ
  - Linux: znaménkový, 32 bitů, UTF-32
  - Windows: neznaménkový, 16 bitů, UTF-16

`char8_t`, `char16_t`, `char32_t`

- neznaménkové číselné typy; rozsah alespoň 8 / 16 / 32 bitů
- určeny pro UTF-8 / UTF-16 / UTF-32

*Všimněte si:* Všechny zmíněné typy jsou definovány *přímo v jazyce* (tedy ne ve standardní knihovně).

# Znakové literály

chars.cpp

`'x'`

- typu `char`, je-li znak reprezentovaný jedním bajtem (v kódování zdrojového souboru)
- (implementace mohou podporovat vícebajtové znakové literály, ty jsou pak typu `int`)

`u8'x'` – typu `char8_t`; kódový bod Unicode v rozsahu 0–127

`u'x'` – typu `char16_t`; kódový bod Unicode v rozsahu 0–65535

`U'x'` – typu `char32_t`; libovolný kódový bod Unicode

`L'x'` – typu `wchar_t`; závisí na platformě

Escape sekvence:
https://en.cppreference.com/w/cpp/language/escape

# Odbočka: std::byte

`std::byte`

- jako `unsigned char`, ale nepodporuje aritmetické operace, pouze bitové (&, |, ^, ~, <<, >>)
- určený pro práci s bajty čistě jako s kolekcí bitů

*Pro zajímavost:*

- typy `char`, `unsigned char` a `std::byte` (a *jen* tyto typy) je možné podle standardu použít ke zkoumání bajtové reprezentace jiných objektů
  - např. pomocí `reinterpret_cast` (mimo záběr předmětu)

# Řetězcové literály

strings.cpp

`"nějaký text"`

- je typu `const char[N]`, kde `N` je délka textu[1] *v bajtech* + 1
  - implicitně se konvertuje v hodnotovém kontextu (*decay*) na `const char*`
- staticky alokované (Céčkové) pole, poslední bajt má hodnotu 0
  - žije jako globální konstanta

`u8"text"`, `u"text"`, `U"text"`, `L"text"`

- podobně jako výše s typem `char8_t` / `char16_t` / `char32_t` / `wchar_t`
- kódování je UTF-8 / UTF-16 / UTF-32 / platformově závislé

[1] počet bajtů / kódových jednotek nutných pro reprezentaci celého textu, *včetně případných znaků s hodnotou 0 uvnitř*; tj. není to `std::strlen`

# Práce s řetězci ve standardní knihovně

std_string.cpp

## `std::string`

- kontejner podobný `std::vector<char>`, ale
  - má vždy prvek na indexu `size()` s hodnotou 0
  - má jiná pravidla pro invalidaci iterátorů / referencí (viz dále)
  - má specifické řetězcové metody a operátor `+`
- různé druhy inicializace
  - zejména z `const char*` – implicitní konverze
- může obsahovat „znaky“ s hodnotou 0 uprostřed
  - ale konverze z `const char*` se zastaví na první 0

- ve skutečnosti alias pro `std::basic_string<char>`
- podobně existují `std::u8string`, `std::u16string`, `std::u32string`, `std::wstring`

# Řetězcové literály pro std::string

std_string.cpp

```
"text"s
```

- k použití potřebujeme
  `using namespace std::string_literals;`[2]
  - jako obvykle, ideálně uvnitř bloku
- má sémantiku podobnou `std::string{ "text" }`
  - ale nezastaví se na první 0
  - délka výsledného řetězce odpovídá délce literálu
- podobně pro `std::u8string` atd.

[2] nebo jen `using namespace std::literals;`

# Řetězcové operace

string_ops.cpp

## Hledání

- metody `find`, `rfind`, `find_first_of`, …
  - parametr specifikující počáteční pozici (typu `std::size_t`)
  - vrací index nebo hodnotu `std::string::npos` (nenalezeno)
- metody `starts_with`, `ends_with`
  - vrací `bool`

## Podřetězec – metoda `substr`

- interval zadaný pozicí a délkou
- vytvoří novou hodnotu typu `std::string`
  - nesdílí znaky s původním řetězcem

## Nahrazení – metoda `replace`

- interval zadaný pozicí a délkou nebo dvěma iterátory
- nahradí úsek řetězce zadaným řetězcem

# Konverze čísel

conversion.cpp

## Čísla → `std:string`

- funkce `std::to_string`
- pro `std::wstring` máme `std::to_wstring`
- detailnější formátování příště

## `std::string` → čísla

- funkce `std::stoi`, `std::stol` apod.
- čtou ze začátku, dokud to jde
- druhý (nepovinný) parametr: ukazatel na `std::size_t`
  - není-li *null*, do odkazovaného objektu se zapíše pozice, kde se zastavilo čtení
- třetí (nepovinný) parametr: báze (jen pro celočíselné typy)
- fungují i pro `std::wstring`
  - (pro `u`-verze zatím nic…)

# Konverze čísel

to_from_chars.cpp

## `std::to_chars`, `std::from_chars`

- rychlá nízkoúrovňová konverze
  - mnohem rychlejší než alternativy
  - užitečná tam, kde potřebujeme konvertovat často
- nealokuje paměť, je třeba mít připravené místo
- nevyhazuje výjimky, vrací dvojici
  - ukazatel za poslední znak, chybový kód
- `std::to_chars` nezapisuje za konec čísla znak s hodnotou 0
- funguje pouze pro řetězce z `char`ů
- omezené možnosti formátování pro `float` / `double`
- záruka konverze „tam a zpět“
  - konverze čísla na řetězec a zpět vede na úplně stejnou hodnotu, i pro `float` / `double` (bez zadání přesnosti)

# Invalidace

**Zachování platnosti iterátorů / referencí** do `std::string`

- nemodifikující operace
  - `const` metody
  - funkce, které berou `const std::string&`
- operátor `[]`
- metody `at`, `data`, `front`, `back`, `begin`, `rbegin`, `end`, `rend`

*Všechny ostatní metody / funkce mohou zneplatnit* ***všechny*** *iterátory / reference do dotčeného objektu typu* `std::string`.

# Odbočka: Small String Optimization

sso.cpp

## Optimalizace pro malé řetězce

- umožněná standardem (pravidly pro invalidaci)
- typicky používaná moderními implementacemi knihovny
- dostatečně „malé“ řetězce jsou drženy přímo *uvnitř* objektu typu `std::string`, pro velké se použije dynamická alokace

## Pro zajímavost

- libstdc++ (gcc), MSVC
  - 15 znaků typu `char`, `char8_t`
  - 7 znaků typu `char16_t`
  - 3 znaky typu `char32_t`
- libc++ (clang)
  - 22 / 10 / 4
- https://www.youtube.com/watch?v=kPR8h4-qZdk (fbstring)

# Pohled na řetězec

`std::string_view`

- (existují všechny verze `u`- i `w`)
- pohled na řetězec nebo jeho část; jen pro čtení
  - implementace: ukazatel + délka
- snadno kopírovatelný
  - kopie neznamená kopii řetězce
  - není vlastníkem řetězce
- má levnou operaci `substr`
  - vrací nový `std::string_view`
- funguje s `std::string` i Céčkovými řetězci
  - `std::string` má operátor konverze na `std::string_view`
- v principu může fungovat i s jinými implementacemi řetězců
  - drží-li znaky v souvislém úseku paměti
  - stačí implementovat operátor konverze

sv_init.cpp

## Inicializace

- bez parametrů – prázdný `std::string_view`
- Céčkovým řetězcem `const char*`
  - konec indikovaný první hodnotou 0
- ukazatelem do pole (`const char*`) a délkou
  - pohled na jeden „znak“ pomocí `{ &c, 1 }`
- dvěma iterátory
  - musí to být iterátory do souvislého úseku paměti (`std::array`, `std::vector`, `std::string`)
  - dereference musí vracet `const char&`

- vše funguje i pro ostatní druhy „znaků“

# Pohled na řetězec

string_view.cpp

## Použití `std::string_view`

- uvažujte o `std::string_view` jako o ukazateli
- předávejte vždy hodnotou
- zajistěte, že původní řetězec bude žít dostatečně dlouho
- invalidace – jako u iterátorů / referencí

## Operace se `std::string_view`

- nemodifikující jako `std::string`
  - `find` apod., operátor `[]`, …
- `std::string_view` je iterovatelný objekt
- `substr`, `remove_prefix`, `remove_suffix`
  - levné, nevytvářejí nový řetězec
  - (kratší) pohledy do původního řetězce

# Řetězcové literály pro `std::string_view`

```
"text"sv
```

- k použití opět potřebujeme
  `using namespace std::string_view_literals;`[3]
- vytvoří `std::string_view` odkazující se na zadaný literál
- připomenutí: řetězcové literály jsou staticky alokované,
  tj. žijí až do konce programu
- podobně pro `std::u8string_view` atd.

## Použití

- kdekoli bychom chtěli použít řetězcový literál, ale s výhodami
  funkcí / metod pro `std::string_view`

[3] nebo jen `using namespace std::literals;`

# Odbočka: Parametry příkazové řádky

args.cpp

```
int main(int argc, char** argv)
```

- `argc` je počet parametrů
- `argv` je ukazatel na začátek pole Céčkových řetězců
  - `argv[0]` je název spuštěného programu

- výhodné použití `std::string_view`
  - operátor `==`, literál `""sv` a implicitní konverze

```
if (argv[i] == "-h"sv || argv[i] == "--help"sv) {
    // ...
}
```

# Pozor na záměnu

bad_string_view.cpp

```
std::string_view sv1 = "some text"sv;
std::string_view sv2 = "some text";
std::string_view sv3 = "some_text"s;
```

- první dva řádky jsou OK
  - `std::string_view` se odkazuje na řetězcové literály
- třetí řádek je špatný
  - vznikne dočasný objekt typu `std::string`
  - `sv3` se odkazuje do tohoto dočasného objektu
  - použití `sv3` kdykoli později vede k nedefinovanému chování

# Řetězcové literály a algoritmy

sv_algo.cpp

`"..."` vs. `"..."sv` v algoritmech / `for` cyklech

- `"..."` je Céčkové pole obsahující koncovou nulu
  - `std::end("...")` ukazuje *za koncovou nulu*
  - `std::size("...")` je *délka pole*
  - iterace pomocí `for` / v algoritmech zahrnuje koncovou nulu
- `"..."sv` je typu `std::string_view`
  - `std::end("..."sv)` ukazuje *za poslední znak*
  - `std::size("..."sv)` je délka řetězce
  - iterace pomocí `for` / v algoritmech *nezahrnuje* koncovou nulu
- `"..."s` podobně jako `"..."sv`

Důsledek: v algoritmech chcete spíše `"..."sv`.

# Doporučení pro řetězcové parametry

`std::string&`

- chceme řetězec modifikovat

`std::string_view`

- chceme řetězec pouze číst
- nebo z něj chceme vyrábět podřetězce, opět pouze pro čtení
- výhoda: bude fungovat i pro Céčkové řetězce (a případně jiné)

`std::string`

- chceme kopii (celého) řetězce
- typicky proto, že si ji chceme někam uložit, např. do položky
  - uložení provedeme pomocí `std::move`
- výhoda: může dojít k vynechání kopií, volající může použít `std::move`

# Pohled do pole

`std::span<T>`

- pohled do pole objektů typu `T`
- funguje podobně jako `std::string_view`, ale
  - pro libovolné typy `T`
  - nemusí být jen pro čtení
  - nemá specifické „řetězcové“ metody

`std::span<T, N>`

- totéž, ale s fixní délkou
- může mít jednodušší implementaci
  - jen ukazatel místo dvojice (ukazatel, délka)

span.cpp

## Inicializace

- bez parametrů – prázdný `std::span`
- Céčkovým polem
- `std::array<T, N>`
- iterátorem a počtem prvků
- dvěma iterátory
- iterovatelným objektem (*range*)
- konverze `std::span<T, N>` → `std::span<T>`

- iterovatelný objekt musí být souvislým úsekem paměti, iterátory musí ukazovat do takového objektu

# Konverze mezi druhy řetězců

utf_convert.cpp

Konverze `std::string` ↔ `std::u32string` apod.

- existuje `std::wstring_convert`
  - použití je trochu komplikované
  - od C++17 je tato část knihovny *deprecated*
  - žádná oficiální alternativa zatím neexistuje
- alternativně si napište vlastní
  - formáty UTF-{8, 16, 32} jsou docela jednoduché
  - napsat si konverzní funkci byste měli zvládnout

```
std::u32string to_utf32(std::string_view s) {
    std::wstring_convert<std::codecvt_utf8<char32_t>,
                         char32_t> conv;
    auto* start = s.data();
    return conv.from_bytes(start, start + s.size());
}
```

# Vstup a výstup s řetězci

## Výstup

- pomocí `std::cout` a operátoru `<<`
- funguje pro `std::string`, `std::string_view`
- pro `std::wstring` máme `std::wcout`
- (pro `u`-verze nic…)

## Vstup

- vstupní proud `std::cin` (případně `std::wcin`)
- operátor `>>` načte *jedno slovo*
  - ignoruje bílé znaky ze začátku
  - potom čte, dokud nenarazí na další bílý znak
- načtení celého řádku pomocí
  `std::getline(std::cin, line)`
  - `line` je proměnná / reference typu `std::string`

*Více uvidíme příště…*