PB071 – Principy nízkoúrovňového programování

Textové řetězce, const, argumenty main, funkční ukazatel

Slidy pro komentáře (děkuji!):
https://drive.google.com/file/d/1zdbtY1QHL6JcXgadNzijXjPjWpInOmL-/view?usp=sharing

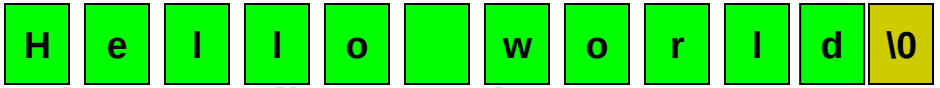
Textové řetězce

Pole znaků, řetězce

- Jak uchovat a tisknout jeden znak?
  - (umíme, char)
  - jak uchovat a tisknout celé slovo/větu?
- Nepraktická možnost – nezávislé znaky
  - `char first = 'E'; char second = 'T';`
  - `printf("%c%c", first, second);`
- Praktičtější možnost – pole znaků
  - `char sentence[10];`
  - `for (int i = 0; i < sizeof(sentence); i++) printf("%c", sentence[i]);`
- Jak ukončit/zkrátit existující větu?
  - `\0` binární nula - speciální znak jako konec

Řetězce v C

- Řetězce v C jsou pole znaků ukončených binární nulou
- `"Hello world"`



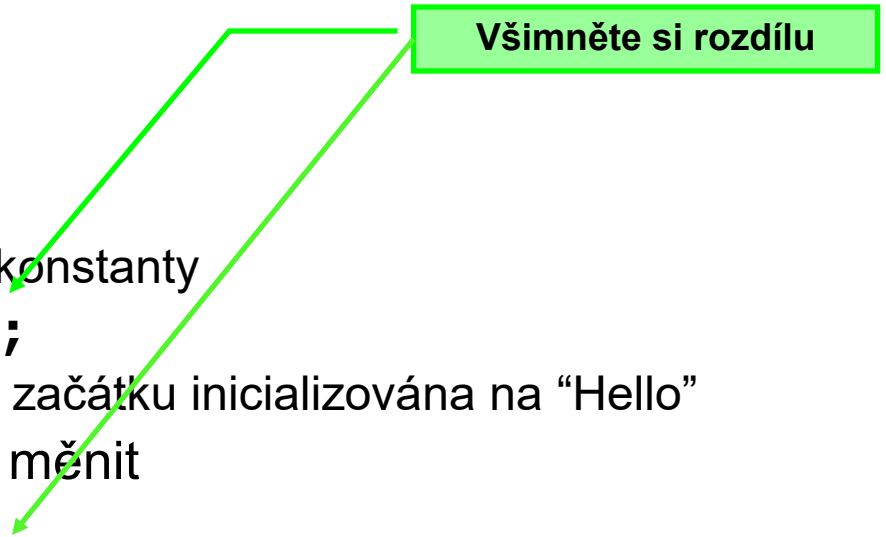
- Díky ukončovací nule nemusíme udržovat délku pole
  - pole znaků procházíme do výskytu koncové nuly
  - řetězec lze “zkrátiť” posunutím nuly
  - vzniká riziko jejího přepsání (viz. dále)
- Deklarace řetězců
  - `char myString[100]; // max. 99 znaků + koncová nula`
  - `char myString2[] = "Hello"; // délka pole dle konstanty, viz dále`
- Od C99 lze i široké (wide) znaky/řetězce
  - `wchar_t myWideString[100]; // max. 99 širokých (typicky Unicode) znaků + koncová nula`

Řetězcová konstanta v C

- Je uzavřená v úvozovkách `""` ("retezcova_konstanta")
- Je uložena v statické sekci programu
- Obsahuje koncovou nulu (binární 0, zápis 0 nebo \0)
  - pozor, ‘0’ NENÍ binární nula (je to ascii znak s hodnotou 48)
- Příklady
  - “” (pouze koncová 0)
  - “Hello” (5 znaků a koncová nula)
  - “Hello world”, “Hello \t world”
- Konstanty pro široké řetězce s mají předřazené `L`
  - `L"Děsně šťavňaťoučké"`
  - `sizeof(L"Hello world") == sizeof("Hello world") * sizeof(wchar_t)`
- Pozn.: Lokalizace programu se ale typicky řeší jinak
  - V kódu unikátní čísla řetězců, samotné řetězce jinde (soubor, res)

Inicializace řetězců

- Pozor na rozdíl inicializace řetězec a pole
  - `char answers[]={'a','y','n'};`
    - nevloží koncovou nulu
  - `char answers2[]="ayn";`
    - vloží koncovou nulu
- Pozor na rozdíl ukazatel a pole
  - `char* myString = "Hello";`
    - ukazatel na pozici řetězcové konstanty
  - `char myString[50] = "Hello";`
    - nová proměnná typu pole, na začátku inicializována na “Hello”
- Pozor, řetězcové konstanty nelze měnit
  - `char* myString = "Hello";`
  - `myString[5] = 'y'; // špatně`
  - vhodné použít `const char* myString = "Hello";`



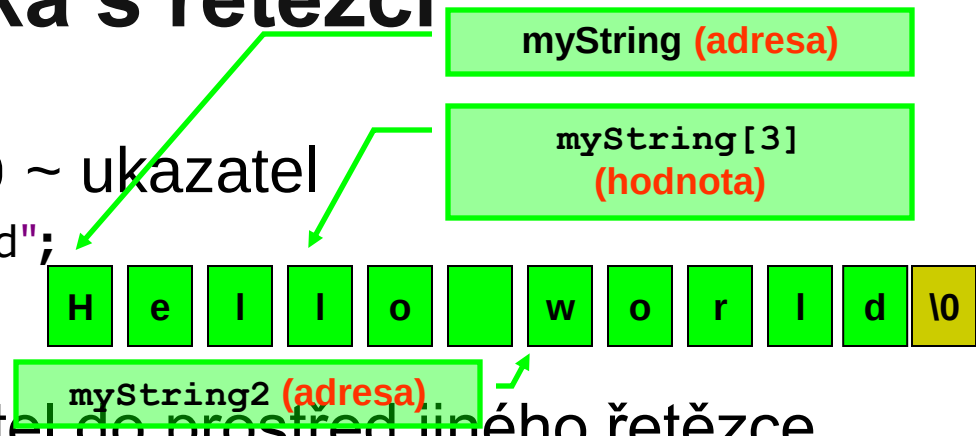
Inicializace řetězců

- Proměnnou můžeme při vytváření inicializovat
  - doporučený postup, v opačném případě je počátečním obsahem předchozí „smetí“ z paměti
  - inicializace výrazem, např. konstantou (`int a = 5;`)
- Pole lze také inicializovat
  - `int array[5] = {1, 2};`
  - zbývající položky bez explicitní hodnoty nastaveny na 0
  - `array[0] == 1`, `array[1] == 2`, `array[2] == 0`
- Řetězec je pole znaků ukončené nulou, jak inicializovat?
  - jako pole: `char myString[] = {'W', 'o', 'r' , 'l' , 'd', \0};`
  - pomocí konstanty: `char myString2[] = "World";`
  - `sizeof(myString) == sizeof("Hello") == 6 * sizeof(char)`
    - Pro zjištění velikosti řetězce ale lépe použít strlen() ze standardní knihovny

Jak manipulovat s řetězci?

1. Jako s ukazatelem
   - využití ukazatelové aritmetiky, operátory +, * ....
2. Jako s polem
   - využití operátoru []
3. Pomocí knihovních funkcí
   - hlavičkový soubor <string.h>
   - strcpy(), strcmp() ...

Ukazatelová aritmetika s řetězci



- Řetězec ~ pole znaků s `\0` ~ ukazatel
  - `char myString[] = "Hello world";`
  - `myString[4]; *(myString + 4)`
  - `myString + 6 => "world"`
- Můžeme uchovávat ukazatel do prostřed jiného řetězce
  - `char* myString2 = myString + 6; // myString2 contains "world"`
- Můžeme řetězec ukončit vložením nuly
  - `myString[5] = 0;`

- Pozor, řetězce v C nemění automaticky svou velikost
  - nedochází k automatickému zvětšení řetězce na potřebnou délku
  - nekorektní použití může vést k zápisu za konec pole
- `myString1 + myString2` je sčítání ukazatelů řetězců
  - nikoli jejich řetězení (jak je v C++/Javě pro typ `string`)

Kdy použít který zápis?

- Zápis v kódu by měl vyjadřovat zamýšlené použití
  - I když pro překladač to bude stejné
  - Programátor by měl z kódu pochopit sémantiku požití
- `myString[4];`
  - Pokud pracujeme s daty myšlenkově jako s polem
- `*(myString + 4)`
  - Pokud pracujeme s daty myšlenkově jako s ukazatelem

Knihovní funkce pro práci s řetězci

- Hlavičkový soubor string.h
  - #include <string.h>
- Kompletní dokumentace dostupná např. na http://www.cplusplus.com/reference/clibrary/cstring/
- Nejdůležitější funkce
  - `strcpy, strcat, strlen, strcmp, strchr, strstr...`
  - výpis řetězce: `puts(myString),printf("%s",myString);//stdio.h`
- Obecná pravidla
  1. funkce předpokládají korektní C řetězec ukončený nulou
  2. funkce modifikující řetězce (strcpy, strcat...) očekávají dostatečný paměťový prostor v cílovém řetězci
     - jinak zápis za koncem alokovaného místa a poškození
  3. při modifikaci většinou dochází ke korektnímu umístění koncové nuly
     - pozor na výjimky

Jak číst dokumentaci?

strcpy, strcpy_s

Defined in header <string.h>

```
char *strcpy( char *dest, const char *src );                          (1)  (until C99)
char *strcpy( char *restrict dest, const char *restrict src );             (since C99)
errno_t strcpy_s(char *restrict dest, rsize_t destsz, const char *restrict src);  (2)  (since C11)
```

1) Copies the null-terminated byte string pointed to by src, including the null terminator, to the character array whose first element is pointed to by dest.

The behavior is undefined if the dest array is not large enough. The behavior is undefined if the strings overlap. The behavior is undefined if either dest is not a pointer to a character array or src is not a pointer to a null-terminated byte string.

2) Same as (1), except that it may clobber the rest of the destination array with unspecified values and that the following errors are detected at runtime and call the currently installed constraint handler function:

- src or dest is a null pointer
- destsz is zero or greater than RSIZE_MAX
- destsz is less or equal `strnlen_s(src, destsz)`; in other words, truncation would occur
- overlap would occur between the source and the destination strings

The behavior is undefined if the size of the character array pointed to by dest <= `strnlen_s(src, destsz)` < destsz; in other words, an erroneous value of destsz does not expose the impending buffer overflow.

As with all bounds-checked functions, strcpy_s is only guaranteed to be available if `__STDC_LIB_EXT1__` is defined by the implementation and if the user defines `__STDC_WANT_LIB_EXT1__` to the integer constant 1 before including string.h.

Parameters

dest	-	pointer to the character array to write to
src	-	pointer to the null-terminated byte string to copy from
destsz	-	maximum number of characters to write, typically the size of the destination buffer

Return value

1) returns a copy of dest

2) returns zero on success, returns non-zero on error. Also, on error, writes zero to `dest[0]` (unless dest is a null pointer or destsz is zero or greater than RSIZE_MAX).

Notes

strcpy_s is allowed to clobber the destination array from the last character written up to destsz in order to improve efficiency: it may copy in multibyte blocks and then check for null bytes.

The function strcpy_s is similar to the BSD function strlcpy, except that

- strlcpy truncates the source string to fit in the destination (which is a security risk)
- strlcpy does not perform all the runtime checks that strcpy_s does
- strlcpy does not make failures obvious by setting the destination to a null string or calling a handler if the call fails.

Although strcpy_s prohibits truncation due to potential security risks, it's possible to truncate a string using bounds-checked strncpy_s instead.

Example

Run this code

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int mai
{
    cha
```

Example

Run this code

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *src = "Take the test.";
//  src[0] = 'M' ; // this would be undefined behavior
    char dst[strlen(src) + 1]; // +1 to accomodate for the null terminator
    strcpy(dst, src);
    dst[0] = 'M'; // OK
    printf("src = %s\ndst = %s\n", src, dst);

#ifdef __STDC_LIB_EXT1__
    set_constraint_handler_s(ignore_handler_s);
    int r = strcpy_s(dst, sizeof dst, src);
    printf("dst = \"%s\", r = %d\n", dst, r);
    r = strcpy_s(dst, sizeof dst, "Take even more tests.");
    printf("dst = \"%s\", r = %d\n", dst, r);
#endif
}
```

Possible output:

```
src = Take the test.
dst = Make the test.
dst = "Take the test.", r = 0
dst = "", r = 22
```

References

- C11 standard (ISO/IEC 9899:2011):
  - 7.24.2.3 The strcpy function (p: 363)
  - K.3.7.1.3 The strcpy_s function (p: 615-616)
- C99 standard (ISO/IEC 9899:1999):
  - 7.21.2.3 The strcpy function (p: 326)
- C89/C90 standard (ISO/IEC 9899:1990):
  - 4.11.2.3 The strcpy function

See also

strncpy strncpy_s (C11)	copies a certain amount of characters from one string to another (function)
memcpy memcpy_s (C11)	copies one buffer to another (function)
wcscpy (C95) wcscpy_s (C11)	copies one wide string to another (function)

Převzato z https://en.cppreference.com/w/c/string/byte/strcpy

Jak číst dokumentaci?

strcpy, strcpy_s

Defined in header <string.h>

```
char *strcpy( char *dest, const char *src );                          (1)  (until C99)
char *strcpy( char *restrict dest, const char *restrict src );        (1)  (since C99)
errno_t strcpy_s(char *restrict dest, rsize_t destsz, const char *restrict src);   (2)  (since C11)
```

1) Copies the null-terminated byte string pointed to by src, including the null terminator, to the character array whose first element is pointed to by dest.

The behavior is undefined if the dest array is not large enough. The behavior is undefined if the strings overlap. The behavior is undefined if either dest is not a pointer to a character array or src is not a pointer to a null-terminated byte string.

2) Same as (1), except that it may clobber the rest of the destination array with unspecified values and that the following errors are detected at runtime and call the currently installed constraint handler function:

- src or dest is a null pointer
- destsz is zero or greater than RSIZE_MAX
- destsz is less or equal strnlen_s(src, destsz); in other words, truncation would occur
- overlap would occur between the source and the destination strings

The behavior is undefined if the size of the character array pointed to by dest <= strnlen_s(src, destsz) < destsz; in other words, an erroneous value of destsz does not expose the impending buffer overflow.

As with all bounds-checked functions, strcpy_s is only guaranteed to be available if __STDC_LIB_EXT1__ is defined by the implementation and if the user defines __STDC_WANT_LIB_EXT1__ to the integer constant 1 before including string.h.

Parameters

dest	-	pointer to the character array to write to
src	-	pointer to the null-terminated byte string to copy from
destsz	-	maximum number of characters to write, typically the size of the destination buffer

Return value

1) returns a copy of dest

2) returns zero on success, returns non-zero on error. Also, on error, writes zero to dest[0] (unless dest is a null pointer or destsz is zero or greater than RSIZE_MAX).

Notes

strcpy_s is allowed to clobber the destination array from the last character written up to destsz in order to improve efficiency: it may copy in multibyte blocks and then check for null bytes.

The function strcpy_s is similar to the BSD function strlcpy, except that

- strlcpy truncates the source string to fit in the destination (which is a security risk)
- strlcpy does not perform all the runtime checks of strcpy_s
- strlcpy does not make failures obvious by setting the destination to a null string or calling a handler if the call fails.

Although strcpy_s prohibits truncation due to potential security risks, it's possible to truncate a string using bounds-checked strncpy_s instead.

Example

Run this code

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *src = "Take the test.";
```

Example

Run this code

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *src = "Take the test.";
//  src[0] = 'M' ; // this would be undefined behavior
    char dst[strlen(src) + 1]; // +1 to accomodate for the null terminator
    strcpy(dst, src);
    dst[0] = 'M';
    printf("src = %s\ndst = %s\n", src, dst);

#ifdef __STDC_LIB_EXT1__
    set_constraint_handler_s(ignore_handler_s);
    int r = strcpy_s(dst, sizeof dst, src);
    printf("dst = \"%s\", r = %d\n", dst, r);
    r = strcpy_s(dst, sizeof dst, "Take even more tests.");
    printf("dst = \"%s\", r = %d\n", dst, r);
#endif
}
```

Possible output:

```
src = Take the test.
dst = Make the test.
dst = "Take the test.", r = 0
dst = "", r = 22
```

References

- C11 standard (ISO/IEC 9899:2011):
  - 7.24.2.3 The strcpy function (p: 363)
  - K.3.7.1.3 The strcpy_s function (p: 615-616)
- C99 standard (ISO/IEC 9899:1999):
  - 7.21.2.3 The strcpy function (p: 326)
- C89/C90 standard (ISO/IEC 9899:1990):
  - 4.11.2.3 The strcpy function

See also

strncpy strncpy_s (C11)	copies a certain amount of characters from one string to another (function)
memcpy memcpy_s (C11)	copies one buffer to another (function)
wcscpy (C95) wcscpy_s (C11)	copies one wide string to another (function)
strdup (dynamic memory TR)	allocate a copy of a string (function)

C++ documentation for strcpy

jméno funkce

deklarace funkce (funkční parametry, návratová hodnota...

popis chování

detaily k parametrům, jejich očekávaný tvar

ukázka použití, velmi přínosné

Odkazy ve standardu

příbuzné a související funkce, přínosné, pokud zobrazená funkce nesplňuje naše požadavky

Nejdůležitější funkce pro řetězce

- `strlen(a)` – délka řetězce bez koncové nuly
  - strlen(“Hello”) == 5
- `strcpy(a,b)` – kopíruje obsah řetězce b do a
  - vrací ukazatel na začátek a
  - a musí být dostatečně dlouhý pro b
- `strcat(a,b)` – připojí řetězec b za a
  - nutná délka a: strlen(a) + strlen(b) + 1; (koncová nula)
- `strcmp(a,b)` – porovná shodu a s b
  - vrací nulu, pokud jsou shodné (znaky i délka), !0 jinak
  - vrací > 0 pokud je a větší než b, < 0 pokud je b > a

Nejdůležitější funkce pro řetězce

- **`strchr(a,c)`** – hledá výskyt znaku `c` v řetězci `a`
  - pokud ano, tak vrací ukazatel na první výskyt c, jinak NULL
- **`strstr(a,b)`** – hledá výskyt řetězce `b` v řetězci `a`
  - pokud ano, tak vrací ukazatel na první výskyt b, jinak NULL
- Pro manipulaci se širokými znaky jsou dostupné analogické funkce v hlavičkovém souboru **`wchar.h`**
  - Wide Character String (WCS)
  - název funkce obsahuje **wcs** namísto **str**
  - např. `wchar_t *wcscpy(wchar_t*, const wchar_t *);`

Testování znaků v řetězci <ctype.h>

Klasifikace znaků - výsledek je nenulové číslo (true) nebo 0 (false)	
`isalpha(znak)`	Je znak písmeno (malé nebo velké)?
`isdigit(znak)`	Je znak dekadická číslice?
`isxdigit(znak)`	Je znak hexadecimální číslice (0-9, a-f, A-F)?
`isalnum(znak)`	Je znak písmeno nebo číslice?
`ispunct(znak)`	Je znak speciální (tisknutelný, ale ani písmeno ani číslice)?
`isprint(znak)`	Je znak tisknutelný (písmeno, číslice, speciální nebo mezera)?
`isgraph(znak)`	Má znak grafickou podobu (písmeno, číslice, speciální, ale ne mezera)?
`isspace(znak)`	Jde o bílý znak (mezeru, tabulátor, nový řádek, návrat vozíku, vertikální tabulátor, nová stránka)?
`iscntrl(znak)`	Jde o řídicí znak (v kódu ASCII jsou to znaky s kódem <32 nebo =127)?
`isupper(znak)`	Je znak velké písmeno?
`islower(znak)`	Je znak malé písmeno?
Převod znaků - z malých písmen na velká nebo naopak (jen jediný znak, ne řetězec!)	
`toupper(znak)`	Je-li znak malé písmeno, je výsledek odpovídající písmeno velké, jinak je vrácen znak původní
`tolower(znak)`	Je-li znak velké písmeno, je výsledek odpovídající písmeno malé, jinak je vrácen znak původní

For the first set, here is a map of how the original 127-character ASCII set is considered by each function (an x indicates that the function returns true on that character)

ASCII values	characters	iscntrl	isspace	isupper	islower	isalpha	isdigit	isxdigit	isalnum	ispunct	isgraph	isprint
0x00 .. 0x08	NUL, (other control codes)	x										
0x09 .. 0x0D	(white-space control codes: '\t','\f','\v','\n','\r')	x	x									
0x0E .. 0x1F	(other control codes)	x										
0x20	space (' ')		x									x
0x21 .. 0x2F	!"#$%&'()*+,-./									x	x	x
0x30 .. 0x39	01234567890						x	x	x		x	x
0x3a .. 0x40	:;<=>?@									x	x	x
0x41 .. 0x46	ABCDEF			x		x		x	x		x	x
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ			x		x			x		x	x
0x5B .. 0x60	[\]^_`									x	x	x
0x61 .. 0x66	abcdef				x	x		x	x		x	x
0x67 .. 0x7A	ghijklmnopqrstuvwxyz				x	x			x		x	x
0x7B .. 0x7E	{|}~									x	x	x
0x7F	(DEL)	x										

Převzato z http://www.cplusplus.com/reference/clibrary/cctype/ a http://www.fi.muni.cz/usr/jkucera/pb071/sl4.htm

Časté problémy

```
char myString1[] = "Hello";
strcpy(myString1, "Hello world");
puts(myString1);
```

- Nedostatečně velký cílový řetězec
  - např. strcpy

```
char myString2[] = "Hello";
myString2[strlen(myString2)] = '!'
```

- Chybějící koncová nula
  - následné funkce nad řetězcem nefungují korektně
  - vznikne např. nevhodným přepisem (N+1 problém)
- Nevložení koncové nuly
  - strncpy

```
char myString3[] = "Hello";
strncpy(myString3, "Hello world", strlen(myString3));
```

- Délka/velikost řetězce bez místa pro koncovou nulu
  - strlen

```
char myString4[] = "Hello";
char myString5[strlen(myString4)];
strcpy(myString5, myString4);
```

- **if (**myString4 == "Hello"**) { }**
  - chybné, porovnává hodnotu ukazatelů, nikoli obsah řetězců
  - nutno použít `if (strcmp(myString, "Hello") == 0) { }`
- Sčítání řetězců pomocí + (sčítá ukazatele namísto obsahu)

Pole řetězců

- Pole ukazatelů na řetězce
- Typicky nepravoúhlé (řetězce jsou různé dlouhé)
- Použití často pro skupinu konstantních řetězců
  - např. dny v týdnu
  - `char* dayNames[] = {"Pondeli", "Utery", "Streda"};`
    - pole se třemi položkami typu char*
    - dayNames`[0]` ukazuje na konstantní řetězec "Pondeli"
- Co způsobí `strcpy(dayNames[0], "Ctvrtek");`?
  - zapisujeme do paměti s konstantním řetězcem
  - pravděpodobně způsobí pád, je vhodné nechat kontrolovat překladačem (klíčové slovo `const` – viz. dále)

Demo – problémy s řetězci

```
void demoStringProblems() {
        char myString1[] = "Hello";
        strcpy(myString1, "Hello world");
        puts(myString1);


        char myString2[] = "Hello";
        myString2[strlen(myString2)] = '!';
        puts(myString2);

        char myString3[] = "Hello";
        strncpy(myString3, "Hello world", sizeof(myString3));

        char myString4[] = "Hello";
        char myString5[strlen(myString4)];
        strcpy(myString4, myString5);

        char myString6[] = "Hello";
        if (myString6 == "Hello") { }

        char* dayNames[] = {"Pondeli", "Utery", "Streda"};
        puts(dayNames[0]);
        strcpy(dayNames[0], "Ctvrtek");
}
```

Umí C aritmetiku s mixovanými typy? ☺

- “-0.5” + 1 == ?
- Proč funguje?
- Kdy funguje?

Klíčové slovo const

Klíčové slovo *const*

- Zavedeno pro zvýšení robustnosti kódu proti nezáměrným implementačním chybám
- Motivace:
  - označit proměnnou, která nesmí být změněna
  - typicky konstanta, např. počet měsíců v roce
- A chceme mít kontrolu přímo od překladače!
- Explicitně vyznačujeme proměnnou, která nebude měněna
  - jejíž hodnota by neměla být měněna
  - argument, který nemá být ve funkci měněn
- Široce používaný koncept (Java *final*, C++ *const*)

Klíčové slovo *const* - ukázka

```
void konstConstantDemo(const int* pParam) {
    //const int a, b = 0; // error, uninitialized const 'a'
    const int numMonthsInYear = 12;
    printf("Number of months in year: %d", numMonthsInYear);

    numMonthsInYear = 13;   // error: assignment of read-only variable
   *pParam = 1; // error: assignment of read-only location
}
```

Klíčové slovo *const*

- Používejte co nejčastěji
  - zlepšuje typovou kontrolu a celkovou robustnost
  - kontrola že omylem neměníme konstantní objekt
  - umožňuje lepší optimalizaci překladačem
  - dává dalšímu programátorovi dobrou představu, jaká bude hodnota konstantní „proměnné“ v místě použití
- Proměnné s **`const`** jsou defaultně lokální v daném souboru
  - Nelze je (defaultně) používat z jiných souborů (aby nedošlo k nezáměrné kolizi)
  - (viditelnost lze zvětšit pomocí **`extern`**)

Řetězcové literály

- `printf("ahoj");`
  - řetězec “ahoj” je uložen ve statické sekci
  - je typu char*, ale zároveň ve statické sekci

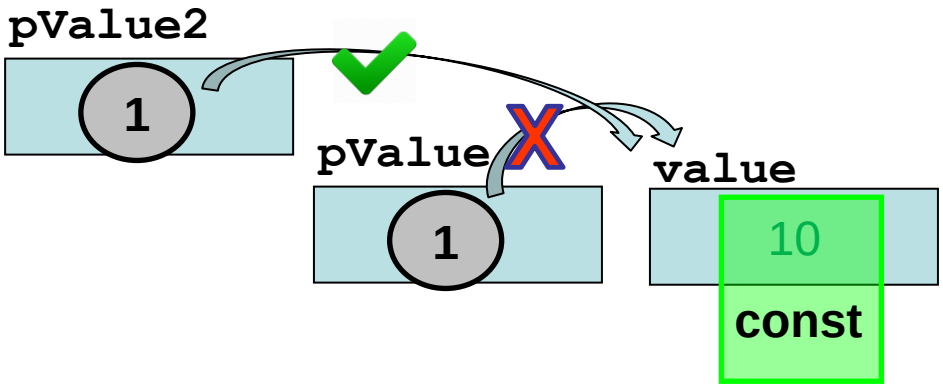
```
char* konstReturnValueDemo() {return "Unmodifiable string"; }
const char* konstReturnValueDemo2() {return "Unmodifiable string"; }

void demoConst() {
    char* value = konstReturnValueDemo();
    value[1] = 'x';          // read-only memory write - problem
    char* value2 = konstReturnValueDemo2(); // error: invalid conversion
}
```

- Používejte tedy **`const char*`**
  - překladač hlídá pokus o zápis do statické sekce
  - **`const char*`** dayNames**`[] = {`**"Pondeli", "Utery", "Streda"**`};`**

const ukazatel

- Konstantní je pouze hodnota označené proměnné
  - platí i pro ukazatel včetně dereference
  - **int** value **=** 10**; const int*** pValue **= &**value**;**
- Není konstantní objekt proměnnou odkazovaný
  - const je „plytké“
  - konstantní proměnnou lze modifikovat přes nekonstantní ukazatel




```
const int value = 10;
const int* pValue = &value;
int* pValue2 = &value;

value = 10;     // error
*pValue = 10;   // error
*pValue2 = 0x1234; // possible
```

Varianty const

- **int** value = 0;
- **const int** value2 = 0;
- **const int** * pValue = &value;
- **int * const** pValue2 = &value;
- **const int * const** pValue3 = &value;

- value = 10; //OK
- value2 = 10; //ERROR
- *pValue = 10; //ERROR
- pValue = &value; //OK
- *pValue2 = 10; //OK
- pValue2 = &value; //ERR
- *pValue3 = 10; //ERR
- pValue3 = &value; //ERR

Konstantní proměnná typu int

Ukazatel na proměnnou typu const int

Konstantní ukazatel na proměnnou typu int

Konstantní ukazatel na proměnnou typu const int

Pomůcka

- *“const se aplikuje na typ vlevo, pokud to nejde, tak na typ vpravo”*
- Příklad («» označuje, na co se bude aplikovat):
  «const T» * var
  «T const» * var
  T «* const» var
  «const T» «* const» var
  «const T const» * var (chyba)

Předání const ukazatele do funkce

```
void foo(const int* carray) {
    carray[0] = 1; // error: assignment of read-only location '*carray'
    int* tmp = carray; //warning: initialization discards qualifiers
                       //from pointer target type
    tmp[0] = 1; // possible, but unwanted! (array was const, tmp is not)
}
int main() {
    int array[10];
    foo(array);
    return 0;
}
```

Argumenty funkce main()

Motivace

- *Jak může program přijímat vstupní data/info?*

- Standardní vstup (klávesnice nebo přesměrování)
- Otevření souboru (disk, stdin…)
- Sdílená paměť (sdílená adresa, virtuální soubor)
- Příkazová řádka (argumenty při spuštění)
- Zprávy (message queue)
- Síťová komunikace (sokety)
- Přerušení, semafory…

Argumenty funkce `main`

- Funkce main může mít tři verze:
  - `int main(void);`
    - bez parametrů
  - `int main(int argc, char *argv[]);`
    - parametry předané programu při spuštění
    - `**argv == *argv[]`
  - `int main(int argc, char **argv, char **envp);`
    - navíc proměnné prostředí,
- ***`binary.exe -test -param1 hello "hello world"`***
- `argc` obsahuje počet parametrů
  - Pokud argc > 0, tak je první cesta ke spuštěnému programu
- `argv[]` je pole řetězců, každý obsahuje jeden parametr
  - `argv[0]` ... cesta k programu
  - `argv[1]` ... první parametr
  - Poslední položka argv[argc] == NULL

Parametry funkce `main` - ukázka

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[], char *envp[]) {
  if (argc == 1) {
     printf("No extra parameters given\n");
  }
  else {
     for (int i = 0; i < argc; i++) {
        printf("%d. parameter: '%s'\n", i, argv[i]);

        if(strcmp(argv[i], "-param1") == 0) {
           printf("   Parameter '-param1' detected!\n");
        }
     }
  }
  // Print environmental variables. Number is not given,
  // but envp ends with NULL (==0) pointer
  printf("\nEnvironment parameters:\n");
  int i = 0;
  while (envp[i] != NULL) printf("%s\n", envp[i++]);

  return EXIT_SUCCESS;
}
```

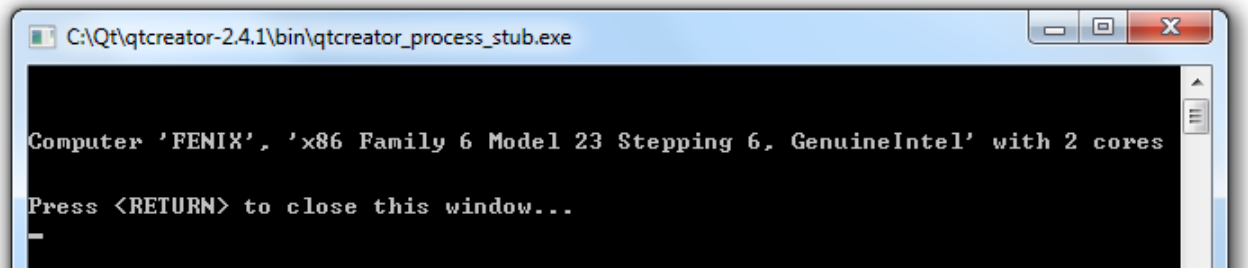
zpracování parametrů spuštění

zpracování proměnných prostředí

Demo: Využití proměnných prostředí

- Jednotlivé položky ve formátu NAZEV=hodnota
  - `PROCESSOR_ARCHITECTURE=x86`
  - lze manuálně parsovat řetězec
  - lze využít funkci **`getenv()`** funkce v stdlib.h
  - poslední položka obsahuje ukazatel na NULL
- Řetězce z **`envp`** nemodifikujte
  - Jsou z proměnných systému kopírovány do vašeho programu a po jeho konci zanikají

```
int main(int argc, char *argv[], char *envp[]) {
    const char* host = getenv("COMPUTERNAME");
    const char* cpu = getenv("PROCESSOR_IDENTIFIER");
    const char* cores = getenv("NUMBER_OF_PROCESSORS");
    printf("Computer '%s'
    return EXIT_SUCCESS;
}
```



Funkční ukazatel

Funkční ukazatel - motivace

- Antivirus umožňuje provést analýzu souborů
  - funkce `int Analyze(const char* filePath);`
- Antivirus běží na pozadí a analyzuje všechny soubory při jejich čtení nebo zápisu na disk
- Jak ale antivirus zjistí, že má soubor analyzovat?
  - trvalé monitorování všech souborů je nemožné
  - vložení funkce Analyze() do všech programů nemožné
- Obsluha souborového systému čtení i zápis zná
  - antivirus si může zaregistrovat funkci `Analyze` na událost čtení/zápis
  - Jak registrovat? → funkční ukazatel (callback)
- Událostmi řízené programování (Event-driven)

Funkční ukazatele

- Funkční ukazatel obsahuje adresu umístění kódu funkce
  - namísto běžné hodnoty je na adrese kód funkce
- Ukazatel na funkci získáme pomocí operátoru &
  - `&Analyze // bez kulatých závorek`
- Ukazatel na funkci lze uložit do proměnné
  - `návratový_typ (*jméno_proměnné)(typy argumentů)`
  - `např. int (*pAnalyze)(const char*) = &Analyze;`
- Ukazatel na funkci lze zavolat jako funkci
  - `pAnalyze("C:\\autoexec.bat");`
- Ukazatel na funkci může být parametr jiné funkce
  - `void foo(int neco, int (*pFnc)(float, float));`

Funkční ukazatel - signatura

- Důležitá je signatura funkce
  - typ a počet argumentů, typ návratové hodnoty, volací konvence
  - jméno funkce není důležité
- Do funkčního ukazatele s danou signaturou můžeme přiřadit adresy všech funkcí se stejnou signaturou

```
int Analyze (const char* filePath) {…}
int Ahoj (const char* filePath) {…}
int main(void) {
  int (*pFnc)(const char*) = &Analyze;
  pFnc("c:\\autoexec.bat");
  pFnc = &Ahoj; // mozne take jako:  pFnc = Ahoj;
  pFnc("c:\\autoexec.bat");
  return 0;
}
```

- Neodpovídající signatury kontroluje překladač

Funkční ukazatel - využití

- Podpora „pluginů“ v systému (např. antivirus)
  - plug-in zaregistruje svůj callback na žádanou událost
    - např. antivirus na událost „přístup k souboru na disku“
  - při výskytu události systém zavolá zaregistrovaný callback
- V systému lze mít několik antivirů
  - seznam funkčních ukazatelů na funkce
  - seznam zaregistrovaných funkcí „`Analyze()`“
- Jak systém pozná, že není žádný antivirus?
  - žádný zaregistrovaný callback

Funkční ukazatel - využití

- Podpora abstrakce (např. I/O zařízení)
  - program si nastaví funkční ukazatel pro výpis hlášení
    - na obrazovku, na display, na tiskárnu, do souboru...
  - provádí volání nastaveného ukazatele, nikoli výběr funkce dle typu výstupu
  - (simulace pozdní vazby známé z OO jazyků)

```
//printScreen(), printLCD(), printPrinter(), printFile()...
void (*myPrint)(const char*);

myPrint = &printLCD;

myPrint("Hello world");
```

Funkční ukazatel - využití

- Aplikace různých funkcí na interval prvků
  - např. simulace *for_each* známého z jiných jazyků
  - přehlednější a rychlejší než switch nad každým prvkem

```
int add2(int value) { return value + 2; }
int minus3(int value) { return value - 3; }

void for_each(int* array, int arrayLen, int(*fnc)(int)) {
    for (int i = 0; i < arrayLen; i++) {
        array[i] = fnc(array[i]);
    }
}
int main(){
    const int arrayLen = 10;
    int myArray[10] = {0};
    for_each(myArray, arrayLen, &add2);
    for_each(myArray, arrayLen, &minus3);

    return 0;
}
```

Funkční ukazatele – konvence volání

- Při funkčním volání musí být
  - někdo zodpovědný za úklid zásobníku (lokální proměnné)
  - definováno pořadí argumentů
- Tzv. konvence volání
  - uklízí volající funkce (cdecl) – default pro C/C++ programy
  - uklízí volaná funkce (stdcall) – např. Win32 API
- GCC: `void foo(int a, char b) __attribute__((cdecl));`
- Microsoft, Borland: `void __cdecl foo(int a, char b);`
- Při nekompatibilní kombinaci dochází k porušení zásobníku
- Více viz.
  - http://en.wikipedia.org/wiki/X86_calling_conventions
  - http://cdecl.org/

Samostudium – detaily funkčních ukazatelů

- Lars Engelfried, The Function Pointer Tutorials
- http://www.newty.de/fpt/index.html
- http://www.newty.de/fpt/zip/e_fpt.pdf

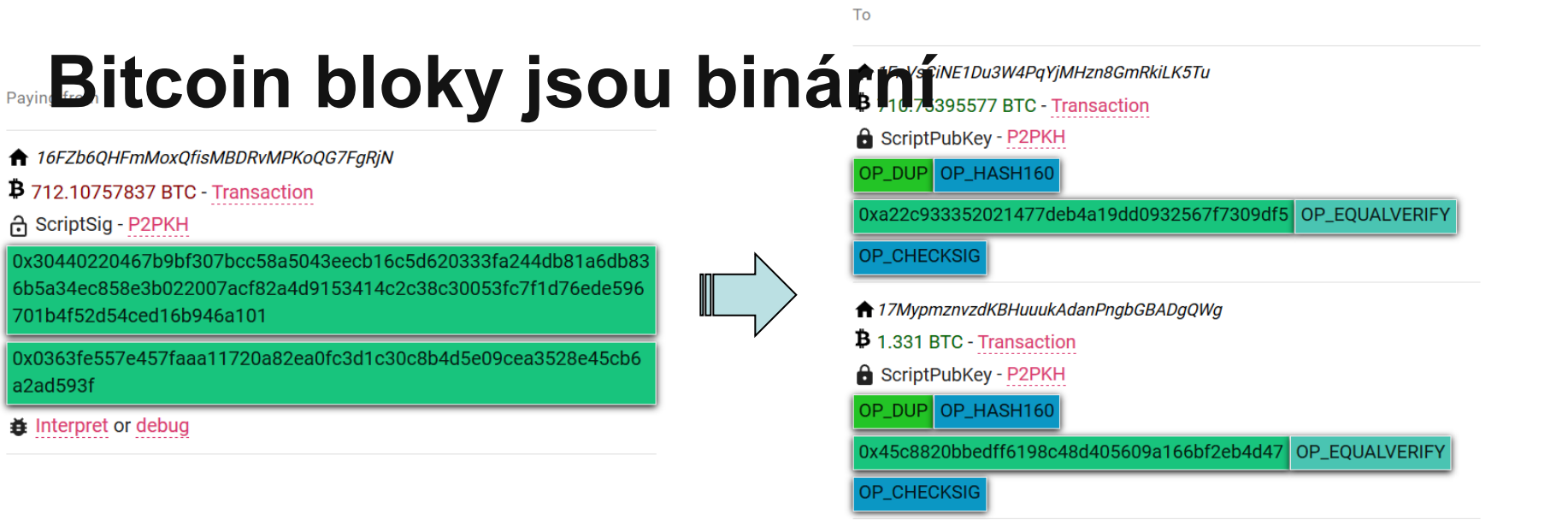
Shrnutí

- Řetězce – pozor na koncovou nulu
- Pole a řetězce – pozor na nechtěný přepis paměti
- `const` – zlepšuje čitelnost kódu a celkovou bezpečnost
- Argumenty funkce main() umožňují přijmout informace z příkazové řádky
- Funkční ukazatel – funkce může být také argument
  - Předáme místo, kde je zápis toho, co se má pustit

DÍKY ZA VÁŠ ČAS A V PONDĚLÍ ZASE NA VIDĚNOU

Bonus ☺

Bitcoin bloky jsou binární




```
02000000 01 cf8adb46bf0cf23bea6d1a17a1b0527313cbefcc6a2206ef3b31
a4a7ba9d4bcd 00000000 6a473044022046 7b9bf307bcc58a5043eecb16c5
d620333fa244db81a6db836b5a34ec858e3b022007acf82a4d9153414c2c3
8c30053fc7f1d76ede596701b4f52d54ced16b946a101210363fe557e457fa
aa11720a82ea0fc3d1c30c8b4d5e09cea3528e45cb6a2ad593f feffffff 02 f97
76c8c10000000 1976a914a22c933352021477deb4a19dd0932567f7309df
588ace0f1ee0700000000 1976a91445c8820bbedff6198c48d405609a166b
f2eb4d4788ac68700700
```

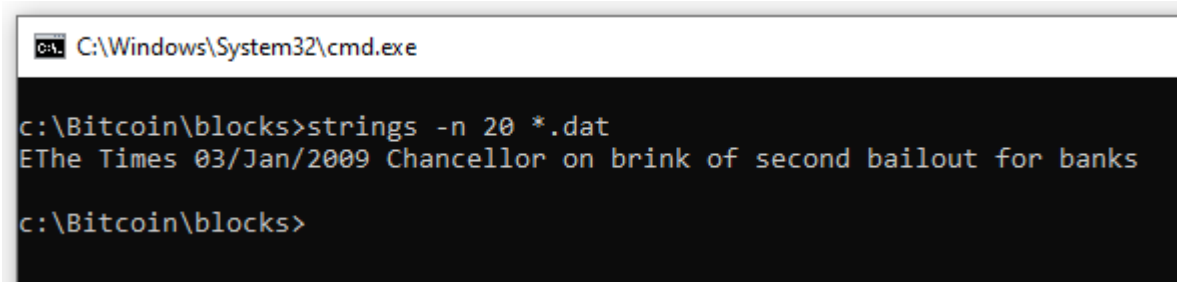
https://nioctib.tech/#/transaction/f2f398dace996dab12e0cfb02fb0b59de0ef0398be393d90ebc8ab397550370b

Řetězce v Bitcoinu

- Bitcoin bloky obsahují převážně binární data
  - hash, podpis, lock/unlock script, txid, hodnota…
- Může ale obsahovat i lidsky čitelné řetězce
  - Nejedná se typicky o standardní Bitcoin transakci
  - Stále binární, ale aby mělo v ASCII kódování význam
- Jak si sami vyzkoušíte?
  - Nainstalujte si Bitcoin full node (https://bitcoincore.org/en/download/)
  - Spusťte bitcoin-qt, nechte synchronizovat alespoň část blockchainu
    - Soubory blk00xxx.dat obsahují části blockchainu po cca 134MB
  - V adresáři /Bitcoin/blocks/ spusťte **`strings -n 20 *.dat`**
    - Najde všechny tisknutelné řetězce o délce alespoň 20 znaků

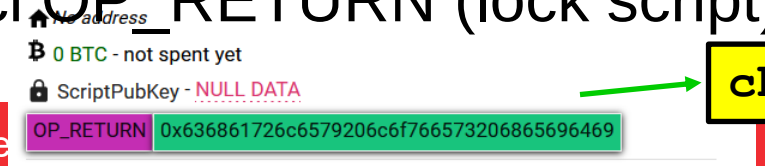
Řetězce v Bitcoinu

```
c:\Bitcoin\blocks>strings -n 20 *.dat
EThe Times 03/Jan/2009 Chancellor on brink of second bailout for banks
```



Kde všude mohou být lidsky čitelné řetězce?

- Variabilní část tzv. coinbase transakce (unlock scriptSig)
  - Typicky identifikace minera (např. `²�¯s ��÷1a`/Foundry USA Pool #dropgold/��-xuž�������-`)
  - Lze i jiné řetězce: `The Times 03/Jan/2009 Chancellor on…`
  - Musí vložit miner (pouze ten vytváří coinbase transakci)
- Různý „hack“ standardních položek běžných transakcí
  1. Platba na neexistující, ale „čitelnou“ adresou
  2. Vanity adresy - opakovaně generovaná adresa dokud není žádoucí prefix (https://github.com/samr7/vanitygen) `$ ./vanitygen 1Love`
     - Address: 1LoveRg5t2NCDLUZh6Q8ixv74M5YGVxXaN
  3. Hodnota převáděných satoshies tak, že „náhodou“ dělá řetězec…
- Data v instrukci OP_RETURN (lock script)
  - Max. 40 bajtů



Slavné řetězce

- Genesis blok [blok 0]
  - “The Times 03/Jan/2009 Chancellor on brink of second bailout for banks”
- Bitcoin 2020 poslední před halving [blok 629999]
  - NYTimes 09/Apr/2020 With $2.3T Injection, Fed's Plan Far Exceeds 2008 Rescue
- OP_RETURN: Mark of the Beast [blok 666666]
  - “Do not be overcome by evil, but overcome evil with good - Romans 12:21”
- Elon Musk “In retrospect, it was inevitable” [blok 668197]
- …

Řetězce v Bitcoinu

- Pokud necháte chvíli bitcoin-qt synchronizovat…

```
c:\Bitcoin\blocks>strings -n 20 *.dat
```

- blk00003.dat
- Dokážete zjistit, jak bylo ascii art vloženo?
  - A kolik to stálo?

https://blockstream.info/tx/930a2114cdaa86e1fac46d15c74e81c99ee1d4150ff9d48e76cb0697d8e1d72