

# PB071 – Programování v jazyce C

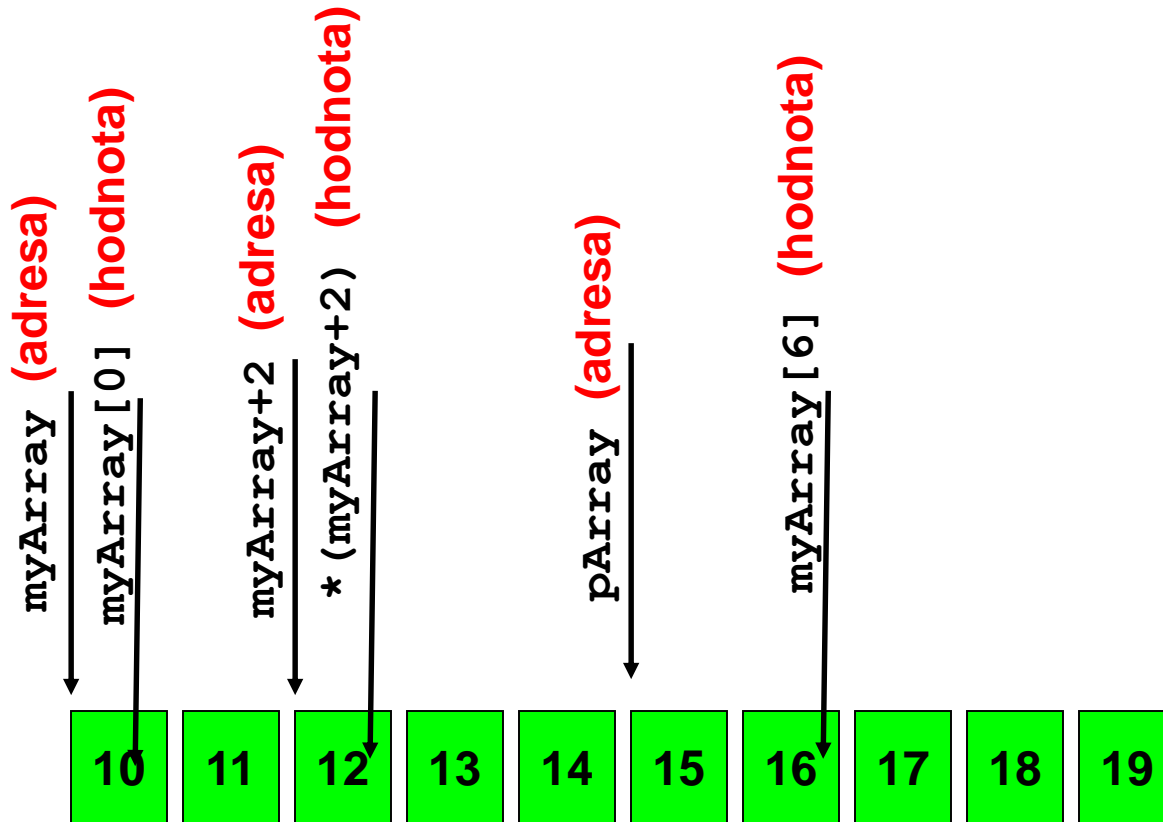
Vícerozměrné pole, textové řetězce,  
const, argumenty main, funkční  
ukazatel

# Organizační – vnitrosemestrálka

- Úterý 11.4.2017 20:00, D1, D3
- Je nutné se přihlásit v ISu
- Forma je odpovědník s 5 možnými odpověďmi

# Ukazatelová aritmetika - ilustrace

```
int myArray[10];  
for (int i = 0; i < 10; i++) myArray[i] = i+10;  
int* pArray = myArray + 5;
```



# Multipole

# Vícerozměrné pole

```
int array[2][4];
```

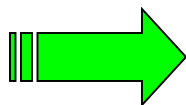
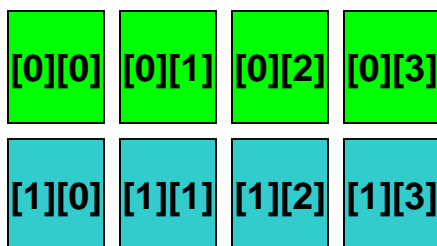
[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]

- Pravoúhlé pole  $N \times M$ 
  - stejný počet prvků v každém řádku
  - `int array[N][M];`
  - (pole ukazatelů o délce  $N$ , v každém adresa pole `intů` o délce  $M$ )
- Přístup pomocí operátoru `[]` pro každou dimenzi
  - `array[7][5] = 1;`
- Lze i vícerozměrné pole
  - v konkrétním rozměru bude stejný počet prvků
  - `int array[10][20][30][7];`
  - `array[1][0][15][4] = 1;`

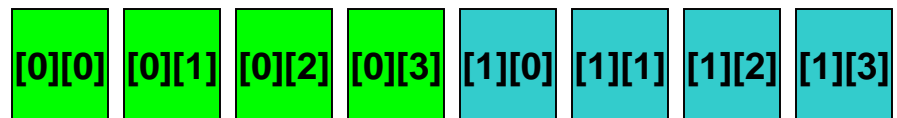
# Reprezentace vícerozměrného pole jako 1D

- Více rozměrné pole lze realizovat s využitím jednorozměrného pole
- Fyzická paměť není N-rozměrná
  - => překladač musí rozložit do 1D paměti
- Jak implementovat pole `array[2][4]` v 1D?
  - indexy v 0...3 jsou prvky `array2D[0][0...3]`
  - indexy v 4...7 jsou prvky `array2D[1][0...3]`
  - `array2D[row][col] → array1D[row * NUM_COLS + col]`
    - `NUM_COLS` je počet prvků v řádku (zde 4)

```
int array2D[2][4];
```



```
int array1D[8];
```

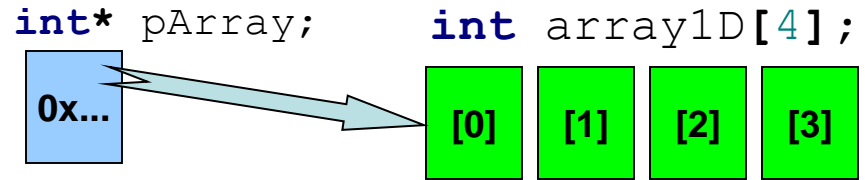


# Pro zamyšlení

- Lze takto realizovat trojrozměrné pole?
  - `int array3D[NUM_X][NUM_Y][NUM_Z];`
- `array3D[x][y][z] → ?`
  - `int array1D[NUM_X*NUM_Y*NUM_Z];`
  - `array1D[x*(NUM_Y*NUM_Z) + y*NUM_Z + z];`
- Proč?



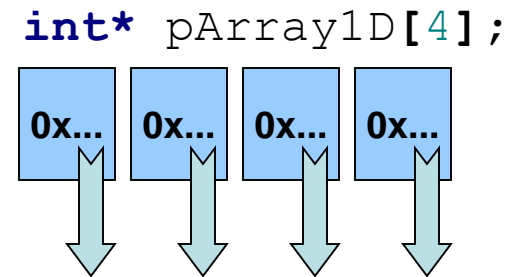
# Nepravoúhlé pole



- Pole, které nemá pro všechny “řádky” stejný počet prvků
  - dosahováno typicky pomocí dynamické alokace (později)
  - lze ale i pomocí statické alokace

- Ukazatel na pole

- adresa prvního prvku
  - `int array[4]; int* pArray = array;`

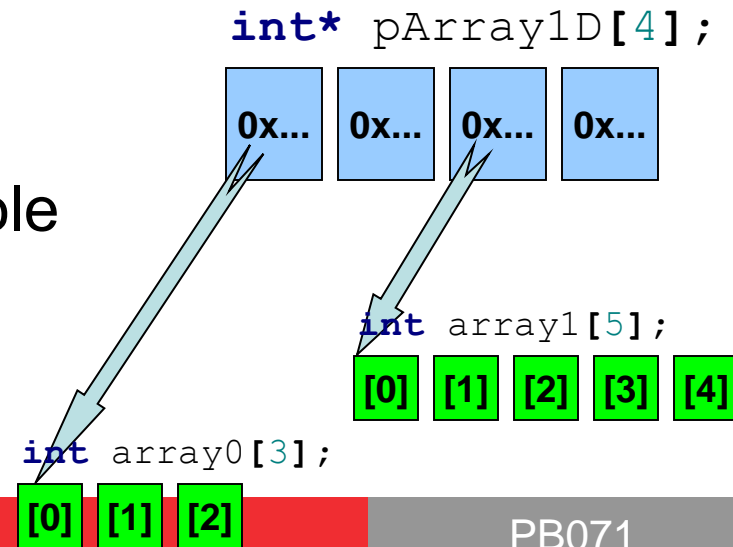


- Pole ukazatelů

- pole položek typu ukazatel
  - `int* pArray1D[4];`

- Do položek `pArray1D` přiřadíme pole

- `pArray1D[0] = array0;`
  - `pArray1D[2] = array1;`



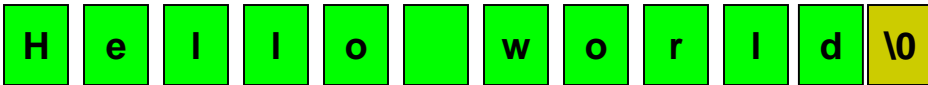


# Textové řetězce

# 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 “zkrátit” 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 unicode znaků + koncová nula

# Řetězcová konstanta v C

- Je uzavřena 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 (unicode) řetězce mají předřazené **L**
  - `L"Děsně šťavňatoučké"`
  - `sizeof(L"Hello world") == sizeof("Hello world") * sizeof(wchar_t)`

# 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"**;

Všimněte si rozdílu

# 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 x sizeof(char)`

# Jak manipulovat s řetězci?

## 1. Jako s ukazatelem

- využití ukazatelové aritmetiky, operátory +, \* ....

## 2. Jako s polem

- využití operátoru []

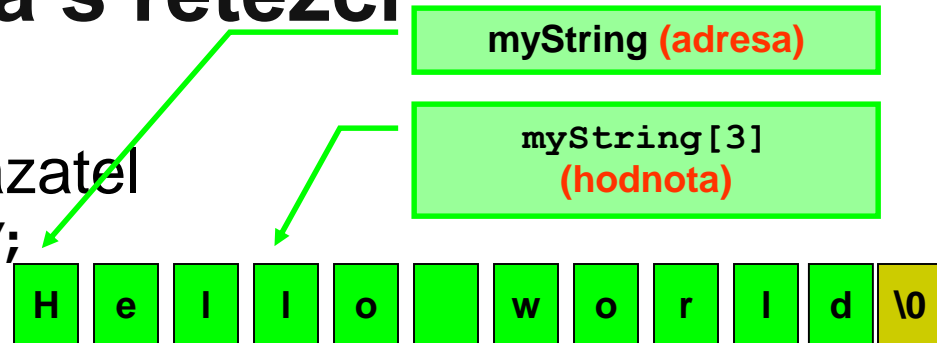
## 3. Pomocí knihovných funkcí

- hlavičkový soubor <string.h>
- strcpy(), strcmp() ...

# Ukazatelová aritmetika s řetězci

- Řetězec ~ pole znaků ~ 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 vložení nuly

- `myString[5] = 0;`



- Pozor, řetězce v C nemění automatickou 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 např. v C++ nebo Javě pro typ `string`)



# Knihovny funkce pro práci s řetězci

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

# Jak číst dokumentaci?

Forum
Reference
C Library
IOstream Library
Strings library
STL Containers
STL Algorithms
Miscellaneous
C Library
cassert (assert.h)
cctype (ctype.h)
cerrno (errno.h)
cfloat (float.h)
ciso646 (iso646.h)
climits (limits.h)
locale (locale.h)
cmath (math.h)
csetjmp (setjmp.h)
csignal (signal.h)
cstdarg (stdarg.h)
cstdarg (stdarg.h)
csdlib (stdlib.h)
cstring (string.h)
ctime (time.h)
cstring (string.h)
functions:
memchr
memcmp
memcpy
memmove
memset
strcat
strchr
strcmp
strcoll
strcpy
strncpy
strcspn
strerror
strlen
strncat
strncpy
strpbrk
strrchr
strspn
strstr
strtok
strxfrm
macros:
NULL
types:
size_t
Insure++
Runtime memory
analysis and error
detection tool for

## strcpy

```
char * strcpy ( char * destination, const char * source );
```

### Copy string

Copies the C string pointed by *source* into the array pointed by *destination*, including the terminating null character.

To avoid overflows, the size of the array pointed by *destination* shall be long enough to contain the same C string as *source* (including the terminating null character), and should not overlap in memory with *source*.

### Parameters

*destination*

Pointer to the destination array where the content is to be copied.

*source*

C string to be copied.

### Return Value

*destination* is returned.

### Example

```
1 /* strcpy example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str1[]="Sample string";
8     char str2[40];
9     char str3[40];
10    strcpy (str2,str1);
11    strcpy (str3,"copy successful");
12    printf ("str1: %s\nstr2: %s\nstr3: %s\n",str1,str2,str3);
13    return 0;
14 }
```

Output:

```
str1: Sample string
str2: Sample string
str3: copy successful
```

### See also

<a href="#">strncpy</a>	Copy characters from string (function)
<a href="#">memcpy</a>	Copy block of memory (function)

jméno funkce

function  
<cstring>

deklarace funkce (funkční  
parametry, návratová hodnota...

popis chování

detaily k parametrům, jejich  
očekávaný tvar

ukázka použití, velmi přínosné

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

Převzato z <http://www.cplusplus.com/reference/cstring/strcpy/>

# 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 vrátí ukazatel na první výskyt c, jinak NULL
- **strstr(a, b)** – hledá výskyt řetězce **b** v řetězci **a**
  - pokud ano, tak vrátí ukazatel na první výskyt b, jinak NULL
- Pro manipulaci se širokými znaky jsou dostupné analogické funkce v hlavičkovém souboru **wchar.h**
  - název funkce obsahuje **wcs** namísto **str**
  - např. `wchar_t *wcsncpy(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 (mezera, tabulátor, nový řádek, návrat vozíku, vertikální tabulátor, nová stránka)?
isctrl (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	isctrl	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	ABCDEFG			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										

# 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

- `if (myString4 == "Hello") { }`

- chybné, porovnává hodnotu ukazatelů, nikoli obsah řetězců
- nutno použít `if (strcmp(myString, "Hello") == 0) { }`

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

- 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 řetězcí

```
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");  
}
```



# PB071 Prednaska 04 – Multipole a retezce



# Klíčové slovo `const`

# Klíčové slovo *const*

- Zavedeno pro zvýšení robustnosti kódu proti nezáměrným implementačním chybám
- Motivace:
  - potřebujeme 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

# 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 lokální v daném souboru

# Řetězcové literály

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

```
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 = 10;   // possible
```

# 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, semaforey...

# 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

# Parametry funkce `main` - ukázka

```
#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;
}
```

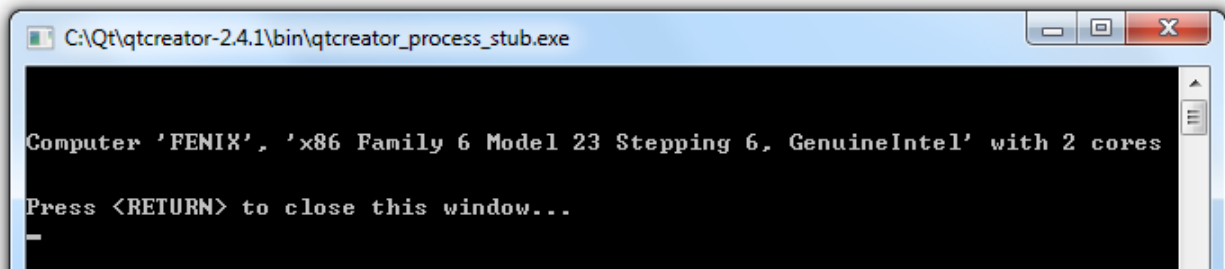
zpracování  
parametrů  
spuštění

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', '%s' with %s cores\n", host, cpu, cores);  
    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[arrayLen] = {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://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](http://www.newty.de/fpt/zip/e_fpt.pdf)

# PB071 Prednaska 04 – Funkcni ukazatele

The image shows the Kahoot! logo in a white, bold, sans-serif font with a slight 3D effect. The logo is centered over a background consisting of a 4x4 grid of squares. The top-left half of the grid (the first two columns) is orange, and the top-right half (the last two columns) is light blue. The bottom half of the grid (the last two rows) is green. The logo is positioned such that it spans across the middle of the grid, with the 'K' and 'a' on the orange background and the 'h', 'o', 'o', 't', and '!' on the blue and green backgrounds.

**Kahoot!**

# Shrnutí

- Pole – nutné chápat souvislost s ukazatelem
- 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