

PB071 – Programování v jazyce C

Vícerozměrné pole, hlavičkové
soubory, řetězce, const

Organizační

Organizační

- Studentští poradci – využívejte
 - místnost B011 – časy na hlavním webu
- Odevzdání prvního příkladu
 - konec odevzdávání 11.3.2014 (zítra, půlnoc)
 - je nutné odevzdat NAOSTRO!
 - Pokud nemáte záznam do ISu od cvičícího do 4 dnů, ozvěte se!

Neopisujte



- Škodíte především sami sobě
 - začněte pracovat včas, ať máte čas na řešení "záseků"
- Provádíme automatickou kontrolu opisu u všech odevzdaných příkladů
 - každý s každým
 - každý s řešeními z minulých let (pokud je podobný příklad)
 - u podezřelých příkladů probíhá manuální kontrola
- V případě opsání jsou potrestáni oba účastníci
- Nedávejte kódy k "nahlédnutí" kamarádům
 - vč. PasteBin, Facebook apod...

Práce s poli

Jednorozměrné pole - opakování

- Jednorozměrné pole lze implementovat pomocí ukazatele na souvislou oblast v paměti
 - `int array[10];`
- Jednotlivé prvky pole jsou v paměti za sebou
- Proměnná typu pole obsahuje adresu prvního prvku pole
 - `int *pArray = array;`
 - `int *pArray = &array[0];`
- Indexuje se od 0
 - n-tý prvek je na pozici `[n-1]`
- Pole v C **nehlídá** přímo meze při přístupu!
 - `int array[10]; array[100] = 1;`
 - pro kompilátor OK, může nastat výjimka při běhu (ale nemusí!)

Jednorozměrné pole – proměnná délka

- Dynamická alokace – bude probíráno později
 - proměnná typu ukazatel `int*` `pArray`;
 - místo na pole alokujeme (a odebíráme) pomocí speciálních funkcí na [haldě](#)
 - `malloc()`, `free()`
- Deklarace pole s variabilní délkou
 - variable length array (VLA)
 - až od C99
 - alokuje se na [zásobníku](#)
 - není nutné se starat o uvolnění (lokální proměnná)
 - nedoporučuje se pro příliš velká pole (použít haldu)

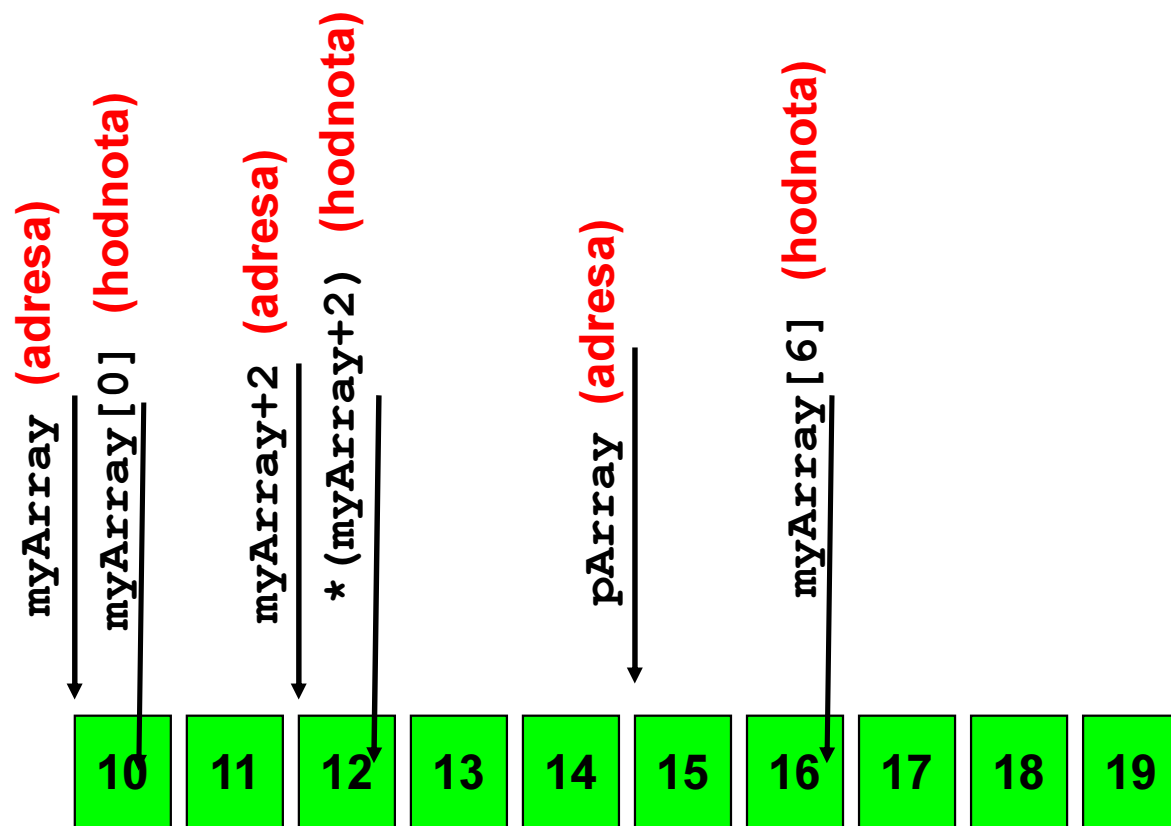
```
int    arraySize = 20;
scanf("%d", &arraySize);
int    arrayLocal[arraySize];
arrayLocal[10] = 9;
```

Ukazatelová aritmetika

- Aritmetické operátory prováděné nad ukazateli
- Využívá se faktu, že `array[X]` je definováno jako `*(array + X)`
- Operátor `+` přičítá k adrese na úrovni prvků pole
 - nikoli na úrovni bajtů!
 - obdobně pro `-`, `*`, `/`, `++` ...
- Adresu počátku pole lze přiřadit do ukazatele

Ukazatelová aritmetika - ilustrace

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



Demo ukazatelová aritmetika

```
void demoPointerArithmetic() {
    const int arrayLen = 10;
    int myArray[arrayLen];
    int* pArray = myArray; // value from variable myArray is assigned to variable pArray
    int* pArray2 = &myArray; // wrong, address of variable array,
                             //not value of variable myArray (warning)

    for (int i = 0; i < arrayLen; i++) myArray[i] = i;

    myArray[0] = 5; // OK, first item in myArray
    *(myArray + 0) = 6; // OK, first item in myArray
    //myArray = 10; // wrong, we are modifying address itself, not value on address

    pArray = myArray + 3; // pointer to 4th item in myArray
    //pArray = 5; // wrong, we are modifying address itself, not value on address
    *pArray = 5; // OK, 4th item
    pArray[0] = 5; // OK, 4th item
    *(myArray + 3) = 5; // OK, 4th item
    pArray[3] = 5; // OK, 7th item of myArray

    pArray++; // pointer to 5th item in myArray
    pArray++; // pointer to 6th item in myArray
    pArray--; // pointer to 5th item in myArray

    int numItems = pArray - myArray; // should be 4 (myArray + 4 == pArray)
}
```

Ukazatelová aritmetika - otázky

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

- Co vrátí myArray[10]?
- Co vrátí myArray[3]?
- Co vrátí myArray + 3?
- Co vrátí *(pArray – 2) ?
- Co vrátí pArray - myArray ?

Typické problémy při práci s poli

- Zápis do pole bez specifikace místa
 - `int array[10]; array = 1;`
 - proměnnou typu pole nelze naplnit (rozdíl oproti ukazateli)
- Zápis těsně za konec pole, častý “N+1” problém
 - `int array[N]; array[N] = 1;`
 - v C pole se indexuje od 0
- Zápis za konec pole
 - např. důsledek ukazatelové aritmetiky nebo chybného cyklu
 - `int array[10]; array[someVariable + 5] = 1;`
- Zápis před začátek pole
 - méně časté, ukazatelová aritmetika
- Čtení/zápis mimo alokovanou paměť může způsobit pád nebo nežádoucí změnu jiných dat (která se zde nachází)
 - `int array[10]; array[100] = 1; // runtime exception`

Tutoriál v češtině

- Programování v jazyku C
 - <http://www.sallyx.org/sally/c/>

Multipole

Vícerozměrné pole

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

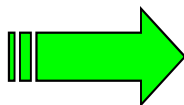
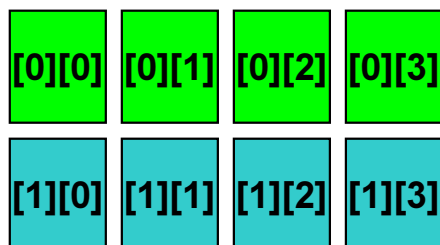
[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]

- Pravoúhlé pole N x 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;`

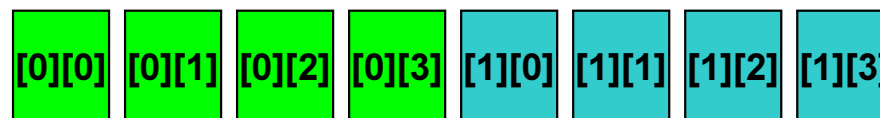
Reprezentace vícerozměrného pole jako 1D

- 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];
```

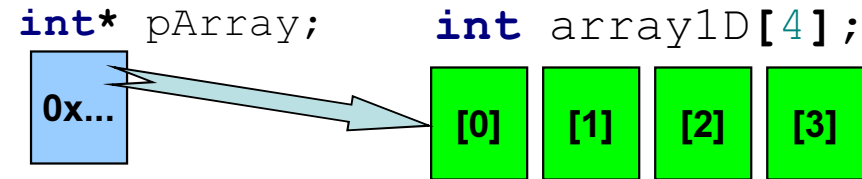


Pro zamyšlení

- 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?



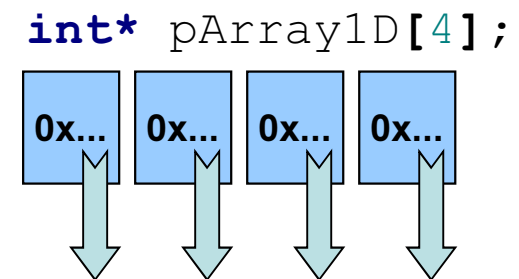
Nepravoúhlé pole



- 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;

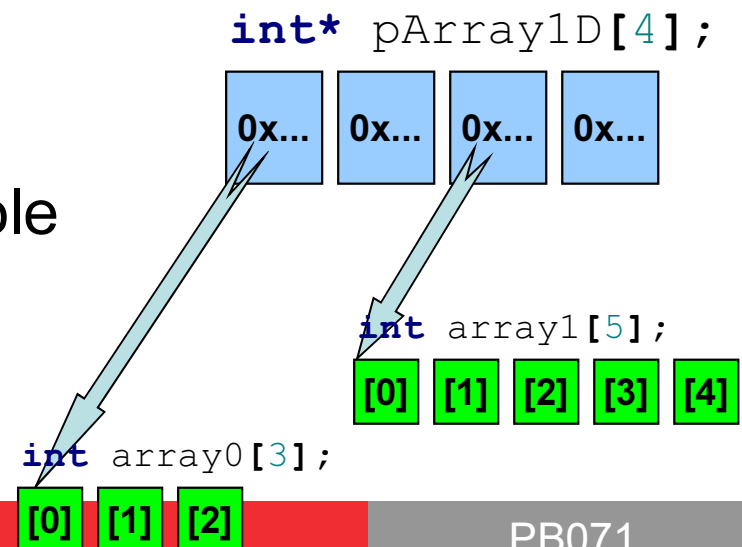


- Pole ukazatelů

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

- Do položek pArray1D přiřadíme pole

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



Hlavičkové soubory

Modulární programování

- Program typicky obsahuje kód, který lze použít i v jiných programech
 - např. výpočet faktoriálu, výpis na obrazovku...
 - snažíme se vyhnout cut&paste duplikaci kódu
- Opakovaně použitelné části kódu vyčleníme do samostatných (knihovných) funkcí
 - např. knihovna pro práci se vstupem a výstupem
- Logicky související funkce můžeme vyčlenit do samostatných souborů
 - deklarace knihovných funkcí do hlavičkového souboru (*.h)
 - implementace do zdrojového souboru (*.c)

Hlavičkové soubory – rozhraní (*.h)

- Obsahuje typicky deklarace funkcí
 - může ale i implementace
- Uživatelův program používá hlavičkový soubor pomocí direktivy preprocesoru **#include**
 - #include <soubor.h> - hledá se na standardních cestách
 - #include "soubor.h" – hledá se v aktuálních cestách
 - #include "../tisk/soubor.h" – hledá se o adresář výše
- Zpracováno během 1. fáze překladač
 - (preprocessing -E)
 - obsah hlavičkového souboru vloží namísto #include
 - ochranné makro #ifndef HEADERNAME_H

```
#include <stdio.h>
int main(void) {
    printf("Hello world");
    return 0;
}
```

stdio.h

```
#ifndef _STDIO_H_
#define _STDIO_H_
int fprintf (FILE *__stream, const char *__format, ...)
// ...other functions
#endif /* MYLIB_H_ */
```

Implementace funkcí z rozhraní

- Obsahuje implementace funkcí deklarovaných v hlavičkovém souboru
- Uživatel vkládá hlavičkový soubor, nikoli implementační soubor
 - dobrá praxe oddělení rozhraní od konkrétného provedení (implementace)
- Dodatečný soubor je nutné zahrnout do kompilace
 - gcc -std=c99 -o binary **file1.c file2.c ... fileN.c**
 - hlavičkové soubory explicitně nezahrnujeme, proč?
 - jsou již zahrnuty ve zdrojáku po zpracování #include

main.c

```
#include "library.h"
int main() {
    foo(5);
}
```

library.h

```
int foo(int a);
```

library.c

```
#include "library.h"
int foo(int a) {
    return a + 2;
}
```

gcc -E main.c → main.i

```
int foo(int a);

int main() {
    foo(5);
}
```

gcc -S, as → main.o

assembler code
adresa funkce foo() zatím
nevyplněna

gcc -E, gcc-S, as → library.s

assembler code
implementace funkce foo()

gcc main.o library.s → main.exe

spustitelná binárka

gcc -std=c99 ... -o binary main.c library.c

Poznámky k provazování souborů

- Soubory s implementací se typicky nevkládají
 - tj. nepíšeme `#include "library.c"`
 - pokud bychom tak udělali, vloží se celé implementace stejně jako bychom je napsali přímo do vkládajícího souboru
- Principiálně ale s takovým vložením není problém
 - preprocesor nerozlišuje, jaký soubor vkládáme
- Hlavičkovým souborem slíbíme překladači existenci funkcí s daným prototypem
 - implementace v separátním souboru, provazuje *linker*
- Do hlavičkového souboru se snažte umisťovat jen opravdu potřebné další `#include`
 - pokud někdo chce použít váš hlavičkový soubor, musí zároveň použít i všechny ostatní `#include`

Textové řetězce

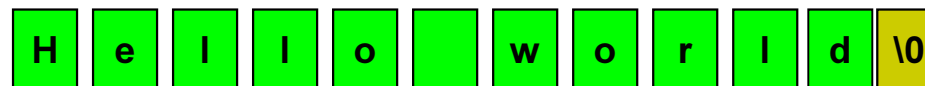
Pole znaků, řetězce

- 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 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 (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 []

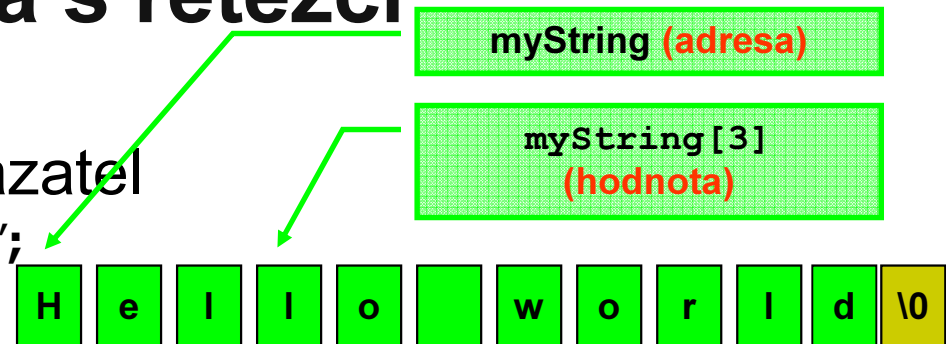
3. Pomocí knihovných funkcí

- 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 prostředí 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`)

Knihovnní funkce pro práci s řetězci

- Hlavičkový soubor `string.h`
 - `#include <string.h>`
- Komplettní 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

memcpy

memmove

memset

strcat

strchr

strcmp

strcpy

strncpy

strerror

strlen

strncat

strncpy

strpbrk

strchr

strspn

strstr

strtok

strxfrm

macros:

NULL

types:

size_t

Insure++

Runtime memory

analysis and error

detection tool for

strcpy

function

<cstring>

```
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

strncpy	Copy characters from string (function)
memcpy	Copy block of memory (function)

jméno funkce

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 požadavky

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

Úvod do C, 10.3.2014

PB071

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 (mezeru, 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") { }

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

- 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");  
}
```


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;  
}
```

Shrnutí

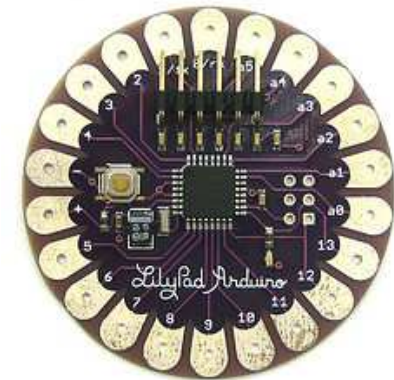
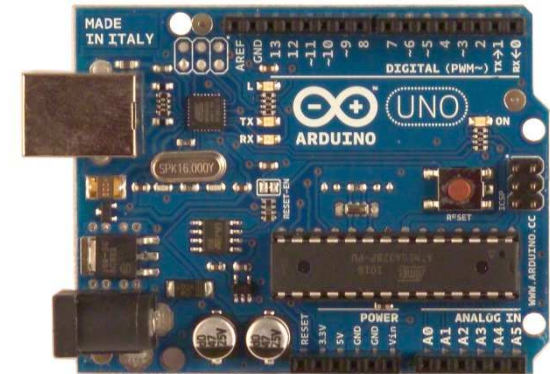
- Pole – nutné chápat souvislost s ukazatelem
- Řetězce – pozor na koncovou nulu
- Pole a řetězce – pozor na nechtěný přepis paměti
- **const** – zlepšuje čitelnost kódu a celkovou bezpečnost

Bonus

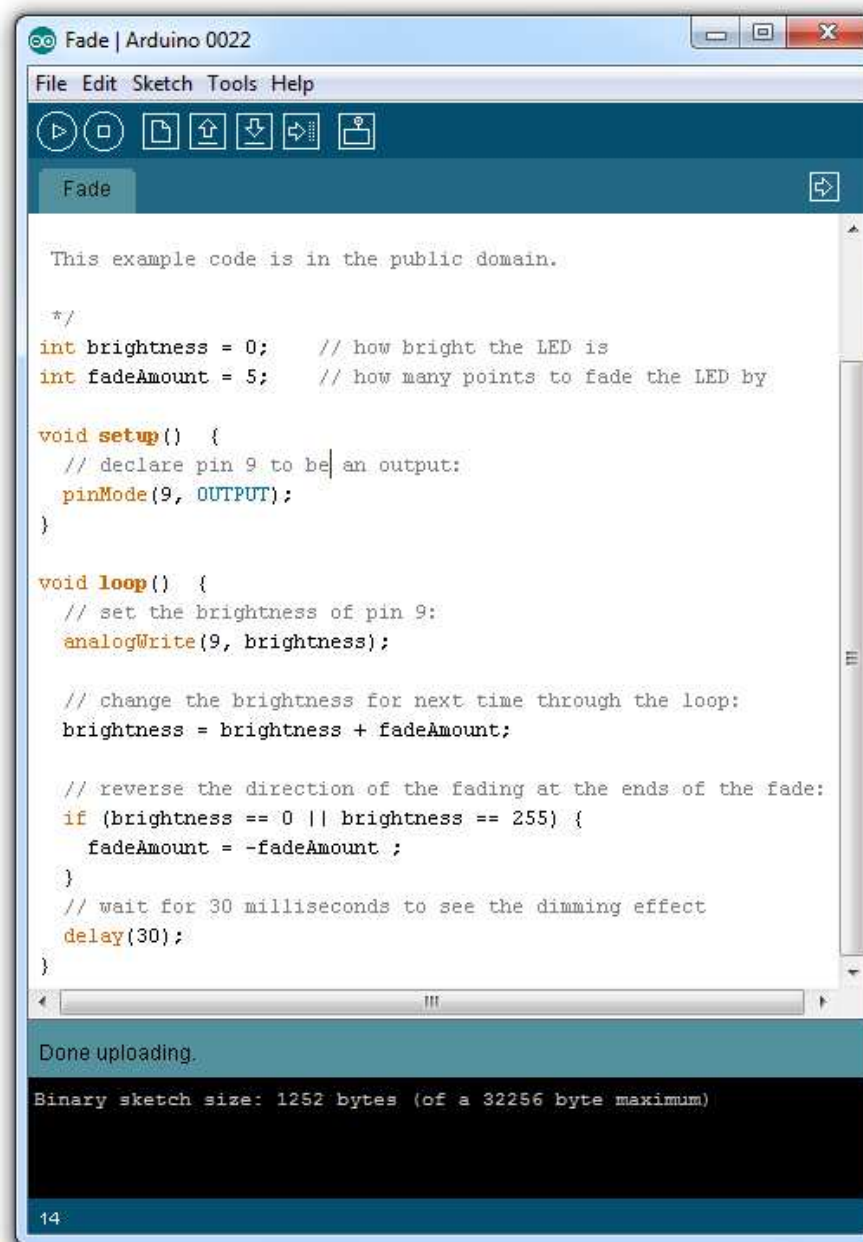
Arduino.cc



- Open-source hardware projekt
 - <http://arduino.cc/en/Main/Hardware>
 - + free vývojové prostředí
 - + velké množství existujících projektů
 - cena cca 650Kč (např. hwkitchen.com)
- Alternativní klony
 - <http://jeelabs.com/products/jeenode> (500Kč)
 - včetně rádiové komunikace
- Programování v C
 - vše připraveno, velice snadné (vyzkoušejte!)
- Projekty na
 - ovládání LED, sensory, roboti, Ethernet, Bluetooth...



Arduino Uno - Fade



```
Fade | Arduino 0022
File Edit Sketch Tools Help

This example code is in the public domain.

*/
int brightness = 0;    // how bright the LED is
int fadeAmount = 5;    // how many points to fade the LED by

void setup() {
  // declare pin 9 to be an output:
  pinMode(9, OUTPUT);
}

void loop() {
  // set the brightness of pin 9:
  analogWrite(9, brightness);

  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;

  // reverse the direction of the fading at the ends of the fade:
  if (brightness == 0 || brightness == 255) {
    fadeAmount = -fadeAmount ;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}

Done uploading.
Binary sketch size: 1252 bytes (of a 32256 byte maximum)

14
```

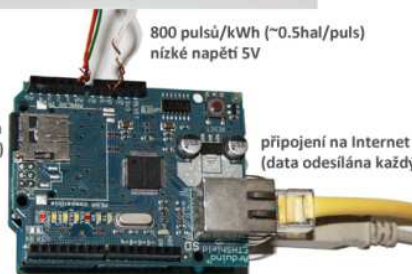
Elektroměr

Elektroměr DTS 353-L 2T X/5A 7M (~2000Kč)



800 pulsů/kWh (~0.5hal/puls)
nízké napětí 5V

paměťová karta
(uložení měření)



připojení na Internet
(data odesílána každých 30 sekund)

Platforma Arduino Uno (625Kč)
+ Ethernet shield (875Kč)

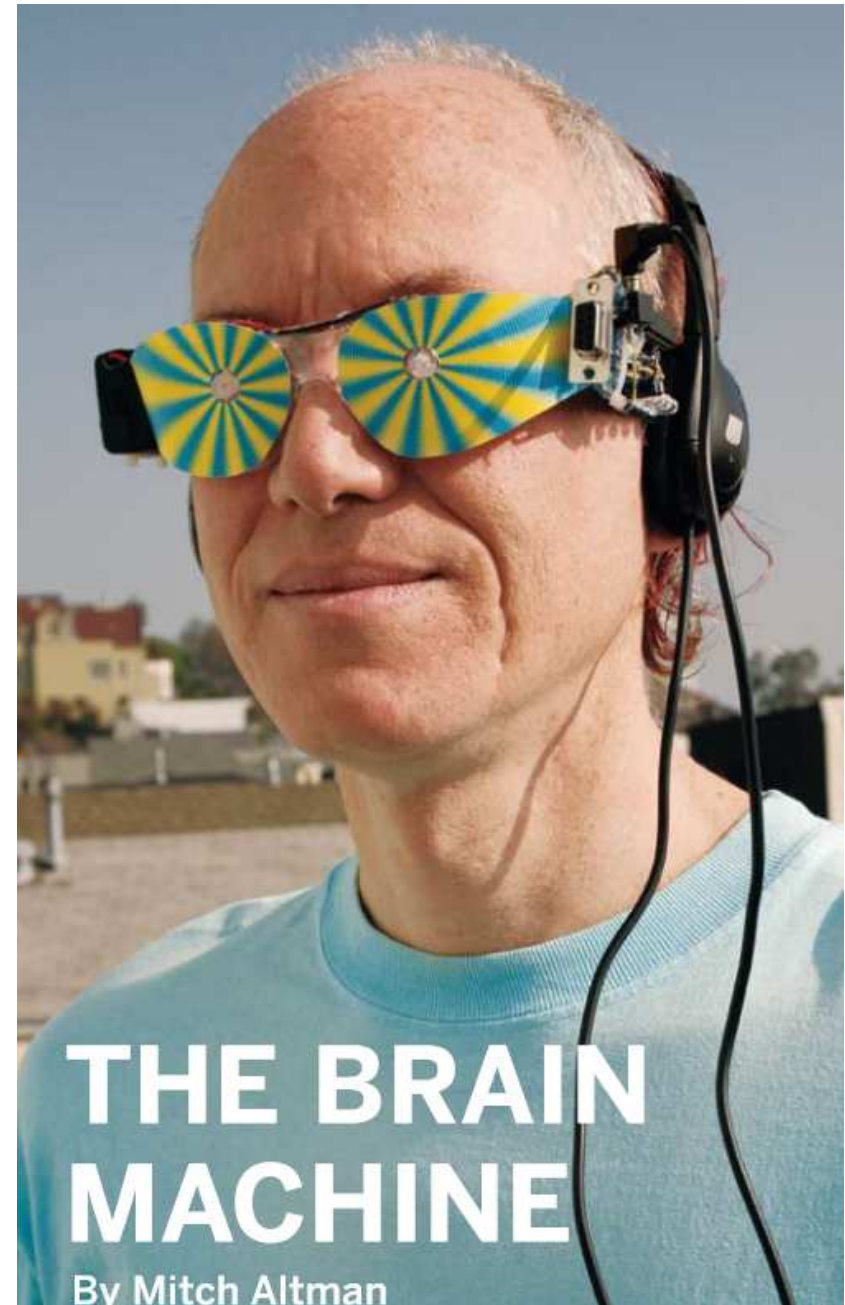
Cosm portál pro sběr
a vizualizaci dat (zdarma)



Přesnost měření na úrovni 0.5 halíře
Zobrazování spotřeby v reálném čase přes Internet
Záloha měření na paměťovou kartu
Celková cena cca 3700Kč

Brain machine

- Brain machine project (M. Altman)
 - http://makezine.com/images/store/MAKE_V10_BrainMachine_Project_F2.pdf
- Založeno na Arduinu
 - <http://low.li/story/2011/01/arduino-brain-machine>



Inspirace?

- “Zaujalo mě Arduino, které jste nám ukazoval v Céčku, tak jsem si říkal, že ho zkusím použít. Napadlo mně postavit si ...”