

PB071 – Programování v jazyce C

Funkce, modulární programování,
paměť a ukazatel, jednorozměrné pole

Funkce, deklarace, definice

Funkce - koncept

- Důležitý prvek jazyka umožňující zachytit rozdělení řešeného problému na podproblémy
- Logicky samostatná část programu (podprogram), která zpracuje svůj vstup a vrátí výstup
 - výstup v návratové hodnotě
 - nebo pomocí vedlejších efektů
 - změna argumentů, globálních proměnných, souborů...

```
int main(void) {  
    // ...  
    for (int fahr = low;  
         celsius = 5.0 /  
    }  
    return 0;  
}
```

```
float f2c(float fahr) {  
    return F2C_RATIO * (fahr - 32);  
}  
  
int main(void) {  
    // ...  
    for (int fahr = low; fahr <= high; fahr += step) {  
        float celsius = f2c(fahr);  
    }  
    // ...  
}
```

implementace
funkce

volání funkce

Funkce - deklarace

- Známe již funkci main

- `int main() { return 0; }`

- Funkce musí být deklarovány před prvním použitím

- Před prvním příkazem v souboru obsahující zavolání funkce

- Dva typy deklarace

- předběžná deklarace
- deklarace a definice zároveň

```
void foo1(void);  
int foo2(void);  
float foo3(int a);  
float foo3(int);  
void foo4(int a, float* b);  
  
// volání  
foo1();  
x = foo2();  
x = foo3(15);
```

- Deklarace funkce

- `návratový_typ` `jmeno_funkce(arg1, arg2, arg3...);`
- lze i bez uvedení jmen proměnných (pro prototyp nejsou důležité)

Funkce - definice

- Implementace těla funkce (kód)
- Uzavřeno v bloku pomocí {}
 - pozor, lokální proměnné zanikají
- Funkce končí provedením posledního příkazu
 - lze ukončit před koncem funkce příkazem **return**
 - funkce vracející hodnotu musí volat **return** hodnota;
- Lze deklarovat a definovat zároveň

```
float f2c(float fahr); // declaration only
----
float f2c(float);      // declaration only
----
float f2c(float fahr) { // declaration and definition
    float celsius = (5.0 / 9.0) * (fahr - 32);
    return celsius;
}
```

Funkce - argumenty

- Funkce může mít vstupní argumenty
 - žádné, fixní počet, proměnný počet
 - lokální proměnné v těle funkce
 - mají přiřazen datový typ (typová kontrola)



Argumenty funkce se v C vždy předávají hodnotou

- při zavolání funkce s parametrem se vytvoří lokální proměnná X
- hodnota výrazu E při funkčním volání se zkopíruje do X
- začne se provádět tělo funkce

```
float f2c(float fahr) {  
    return (5.0 / 9.0) * (fahr - 32);  
}  
int main(void) {  
    int a = 10;  
    float celsius1 = f2c(100);  
    float celsius2 = f2c(a);  
    return 0;  
}
```

implementace
funkce, fahr je
lokální proměnná

volání funkce s
argumentem 100,
100 je výraz

Způsob předávání funkčních argumentů

- Argument funkce je její lokální proměnná
 - lze ji číst a využívat ve výrazech
 - lze ji ve funkci měnit (pokud není `const`)
 - na konci funkce proměnná zaniká
- Pořadí předávání argumentů funkci není definováno
 - tedy ani pořadí vyhodnocení výrazů před funkčním voláním
 - `int i = 5; foo(i, ++i) → foo(5, 6) nebo foo(6,6)?`
- Problém: jak přenést hodnotu zpět mimo funkci?
 - využití návratové hodnoty funkce
 - využití argumentů předávaných hodnotou ukazatele

Funkce – návratová hodnota

- Funkce nemusí vracet hodnotu

- deklaruujeme pomocí void

```
void div(int a, int b) {  
    printf("%d", a / b);  
}
```

- Funkce může vracet jednu hodnotu

- klíčové slovo **return** a hodnota

```
int div(int a, int b) {  
    int div = a / b;  
    return div;  
}
```

- Jak vracet více hodnot?

- využití globálních proměnných (většinou nevhodné)
- strukturovaný typ na výstup (struct, ukazatel)
- modifikací vstupních parametrů (předání ukazatelem)

```
int div = 0;  
int rem = 0;  
void divRem(int a, int b) {  
    div = a / b;  
    rem = a % b;  
}  
int main() { divRem(7, 3); return 0; }
```

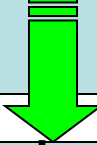

Hlavičkové soubory

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)
 - Jaké argumenty bude funkce přijímat a jaké hodnoty vrátet
 - implementace do zdrojového souboru (*.c)
 - Jak přesně je tělo funkce implementované

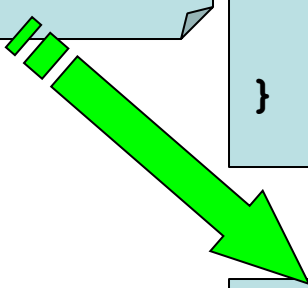
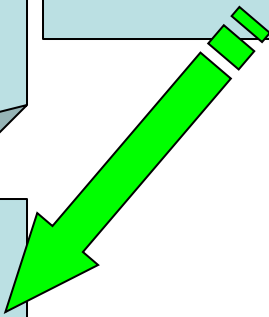
main.c

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



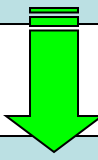
library.h

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



library.c

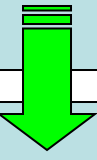
```
#include "library.h"
int foo(int a) {
    return a + 2;
}
```



gcc -E main.c → main.i

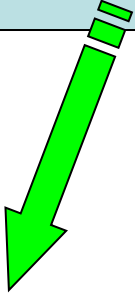
```
int foo(int a);

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



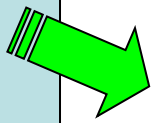
gcc -E, gcc-S, as → library.s

assembler code
implementace funkce foo()



gcc -S, as → main.o

assembler code
adresa funkce foo() zatím
nevyplněna



gcc main.o library.s → main.exe

spustitelná binárka

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

Dělení kódu na části - ukázka

```
#include <stdio.h>
#define F2C_RATIO (5.0 / 9.0)
int main(void) {
    int fahr = 0;           // promenna pro stupne fahrenheitu
    float celsius = 0;      // promenna pro stupne celsia
    int dolni = 0;          // pocatecni mez tabulky
    int horni = 300;        // horni mez
    int krok = 20;          // krok ve stupnich tabulky

    // vypiseme vyzvu na standardni vystup
    printf("Zadejte pocatecni hodnotu: ");
    // precteme jedno cele cislo ze standardniho vstupu
    scanf("%d", &dolni);

    for (fahr = dolni; fahr <= horni; fahr += krok) {
        celsius = F2C_RATIO * (fahr - 32);
        // vypise prevod pro konkretni hodnotu fahrenheitu
        printf("%3d \t %6.2f \n", fahr, celsius);
    }
    return 0;
}
```

Dělení kódu na části - ukázka

1. Převod F na C vyjmemme z mainu do samostatné funkce
 - Vhodnější, ale stále neumožňuje využít funkci f2c v jiném projektu
 - Řešení: konverzní funkce přesuneme do samostatné knihovny
2. Deklarace funkce (signatura funkce)
 - `float f2c(float fahr);`
 - Rozhraní funkčnosti pro výpočet převodu
 - Přesuneme do samostatného souboru *.h (converse.h)
3. Definice funkce (tělo funkce)

```
float f2c(float fahr) {  
    return F2C_RATIO * (fahr - 32);  
}
```

 - Přesuneme do samostatného souboru *.c (converse.c)
4. Původní main.c upravíme na využití fcí z converse.h
 - `#include "converse.h"`
5. Překládáme včetně converse.c (`gcc main.c converse.c`)

converse.h

```
#ifndef CONVERSE_H
#define CONVERSE_H
float f2c(float a);
#endif
```

Měla by být definice
F2C_RATIO v hlavičkovém
souboru? Proč (ne)?

main.c

```
#include "converse.h"
int main() {
    float celsius = f2c(20);
    return 0;
}
```

converse.c

```
#include "converse.h"
#define F2C_RATIO (5.0 / 9.0)
float f2c(float fahr) {
    return F2C_RATIO * (fahr - 32);
}
```

```
gcc -std=c99 .... -o binary main.c converse.c
spustitelná binárka
```

Poznámky k provazování souborů

- Soubory s implementací se typicky nevkládají
 - tj. nepíšeme `#include "library.c"`
- Principiálně ale s takovým vložením není problém
 - preprocesor nerozlišuje, jaký soubor vkládáme
 - pokud bychom tak udělali, vloží se celé implementace stejně jako bychom je napsali přímo do vkládajícího souboru
- 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`

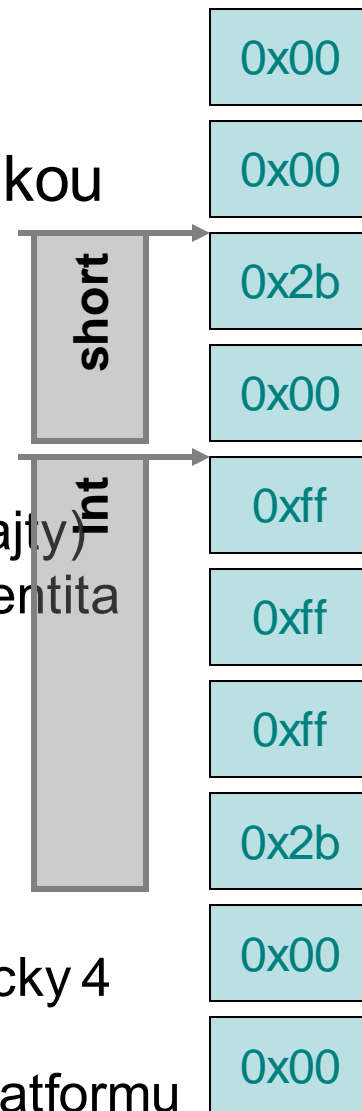
PB071 Prednaska 03 – Funkce a soubory



Realizace objektů v paměti, ukazatel

Paměť

- Paměť obsahuje velké množství slotů s fixní délkou
 - adresovatelné typicky na úrovni 8 bitů == 1 bajt
- V paměti mohou být umístěny entity
 - proměnné, řetězce...
- Entita může zabírat více než jeden slot
 - např. proměnná typu short zabírá na x86 16 bitů (2 bajty)
 - adresa entity v paměti je dána prvním slotem, kde je entita umístěna
 - zde vzniká problém s Little vs. Big endian
- Adresy a hodnoty jsou typicky uváděny v hexadecimální soustavě s předponou 0x
 - 0x0a == 10 dekadicky, 0x0f == 15 dekadicky
 - adresy na x86 zabírají pro netypované ukazatele typicky 4 bajty, na x64 8 bajtů
 - na konkrétní délky nespolehejte, ověřte pro cílovou platformu



Organizace paměti

- Instrukce (program)
 - nemění se
- Statická data (static)
 - většina se nemění, jsou přímo v binárce
 - globální proměnné (mění se)
- Zásobník (stack)
 - mění se pro každou funkci (stack-frame)
 - lokální proměnné
- Halda (heap)
 - mění se při každém malloc, free
 - dynamicky alokované prom.

Celková paměť programu

Instrukce

```
...  
push %ebp 0x00401345  
mov %esp,%ebp 0x00401347  
sub $0x10,%esp 0x0040134d  
call 0x415780  
...
```

Statická a glob. data

```
"Hello",  
"World"  
{0xde 0xad 0xbe 0xef}
```

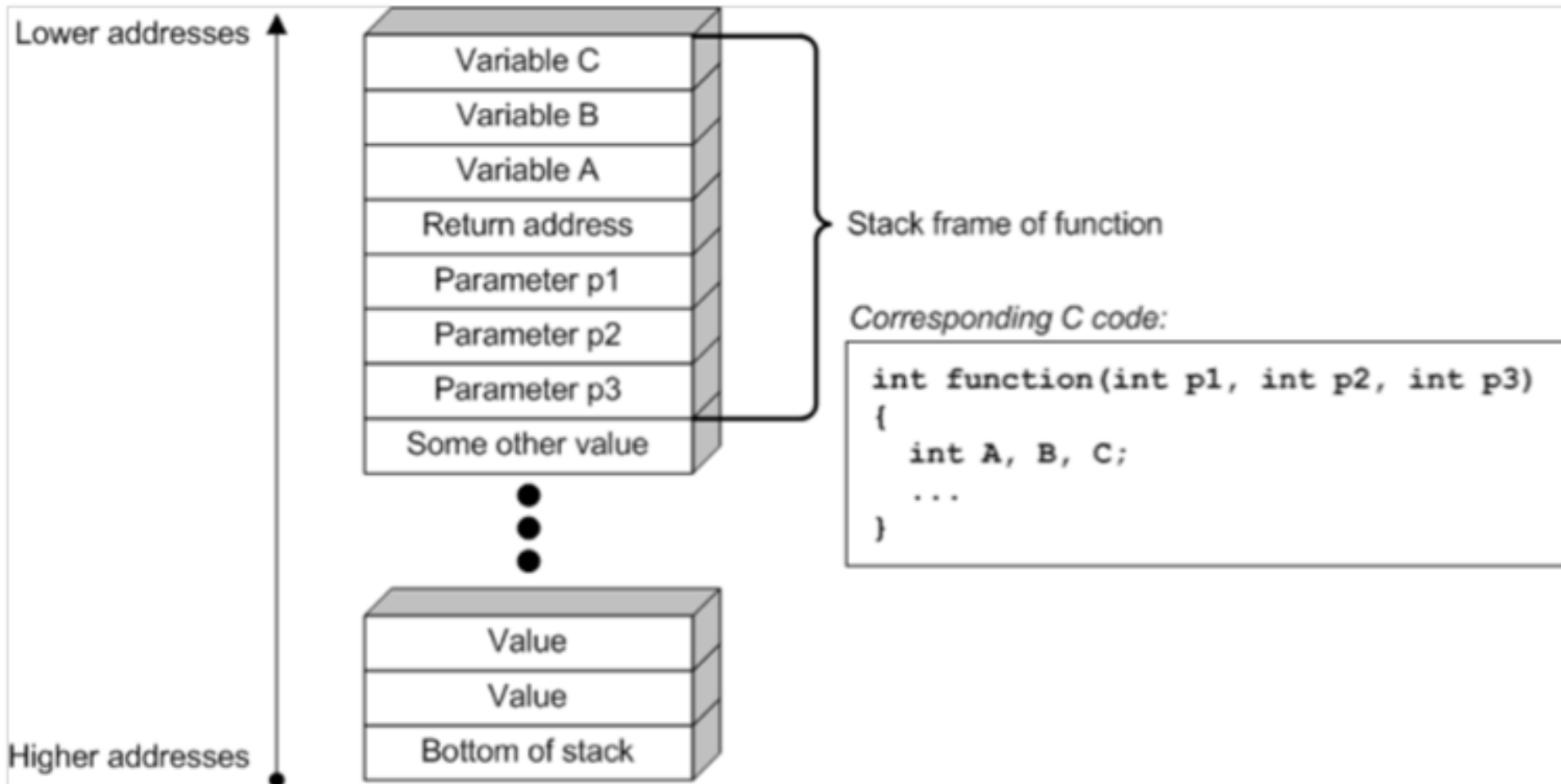
Zásobník

```
lokálníProm1  
lokálníProm2
```

Halda

```
dynAlokace1  
dynAlokace2
```

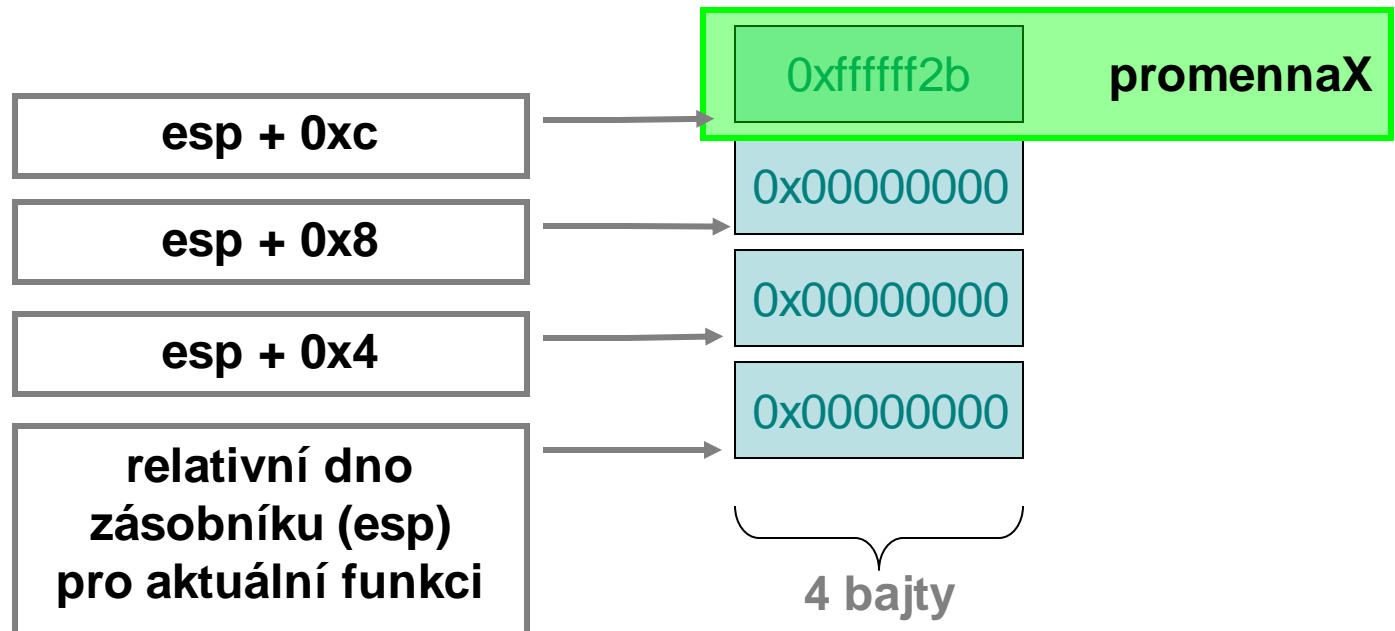
Rámec na zásobníku (stack frame)



<http://www.drdoobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832#>

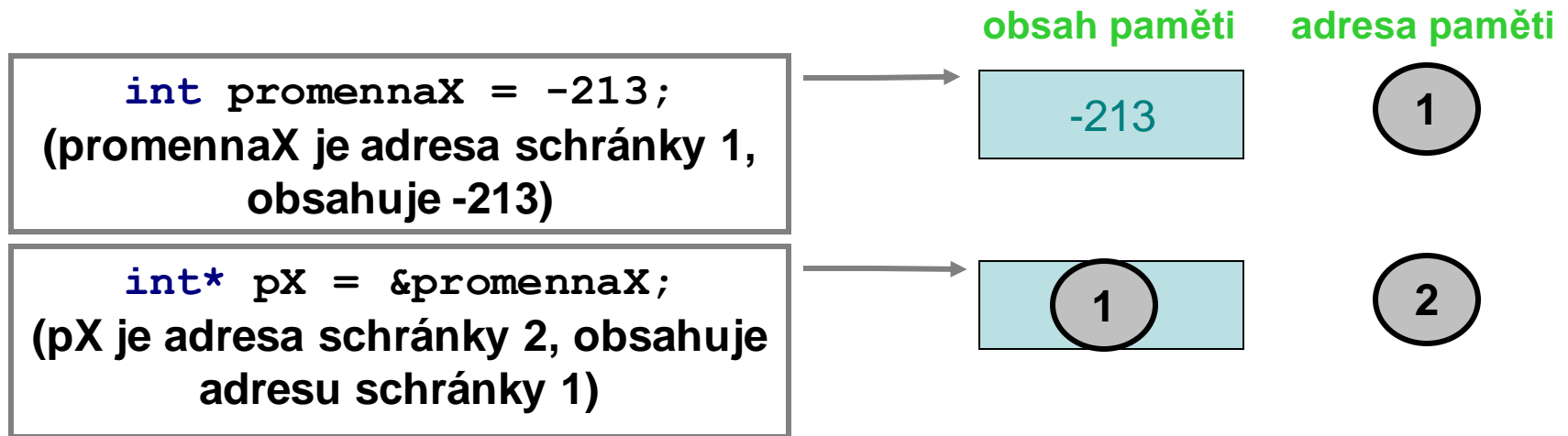
Proměnná na zásobníku

- POZOR: Zásobník je zde znázorněn „učebnicovým“ způsobem (růst „nahoru“)
- C kód: `int promennaX = -213;`
- Asembler kód: `movl $0xffffffff2b,0xc(%esp)`



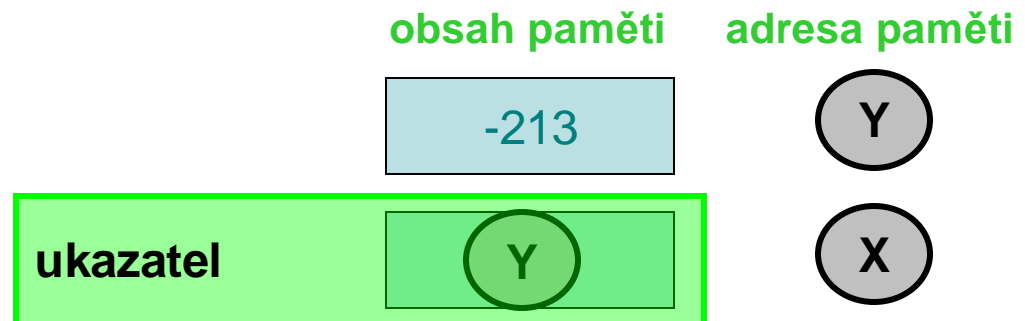
Operátor &

- Operátor `&` vrací adresu svého argumentu
 - místo v paměti, kde je argument uložen
- Výsledek lze přiřadit do ukazatele
 - `int promennaX = -213;`
 - `int* pX = &promennaX;`



Proměnná typu ukazatel

- Proměnná typu ukazatel je stále proměnná
 - tj. označuje místo v paměti s adresou X
 - na tomto místě je uložena další adresa Y
- Pro kompletní specifikaci proměnné není dostačující
 - chybí datový typ pro data na adrese Y
 - Jak se mají bity paměti na adrese Y interpretovat?
 - datový typ pro data na adrese X je ale známý
 - je to adresa
- Neinicializovaný ukazatel
 - v paměti na adrese X je „smetí“ (typicky velké číslo, ale nemusí být)
- Nulový ukazatel (`int* a = 0;`)
 - v paměti na adrese X je 0
- Pozor na `int* a, b;`
 - `a` je `int*`, `b` jen `int`



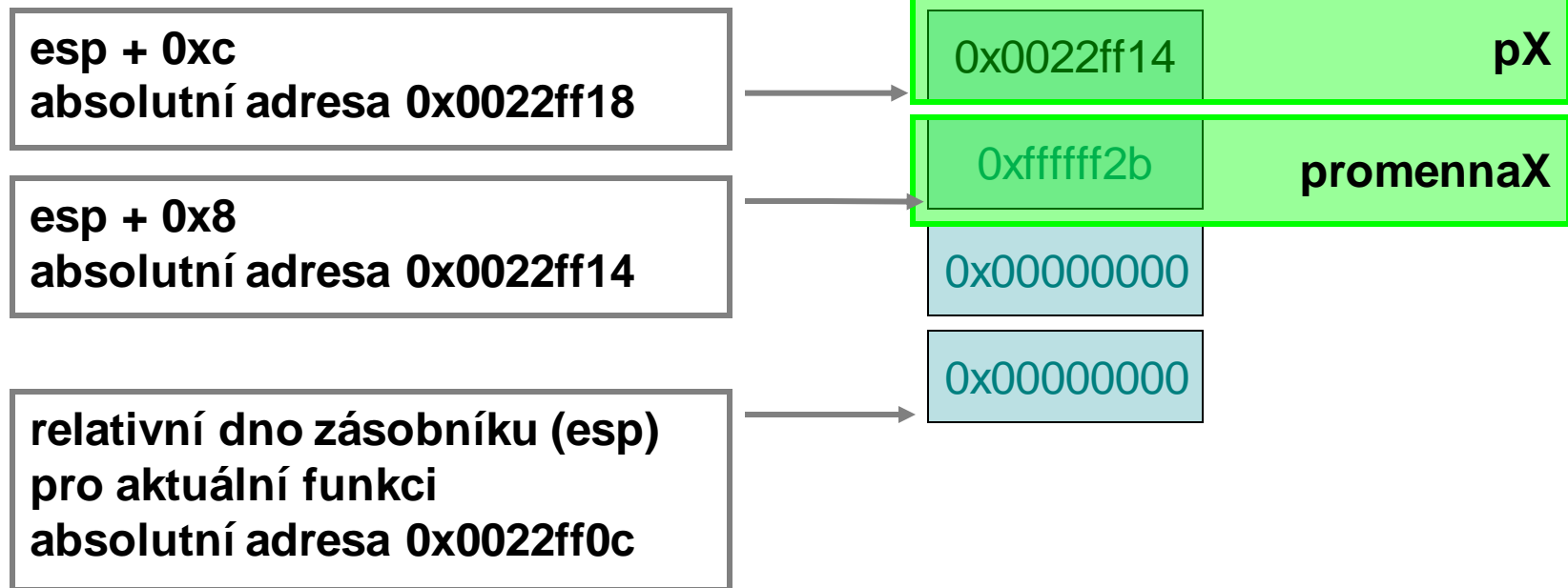
Proměnná typu ukazatel na typ

● C kód: `int* pX = &promennaX;`

operátor & vrátí adresu argumentu (zde promennaX)

● Asembler kód:

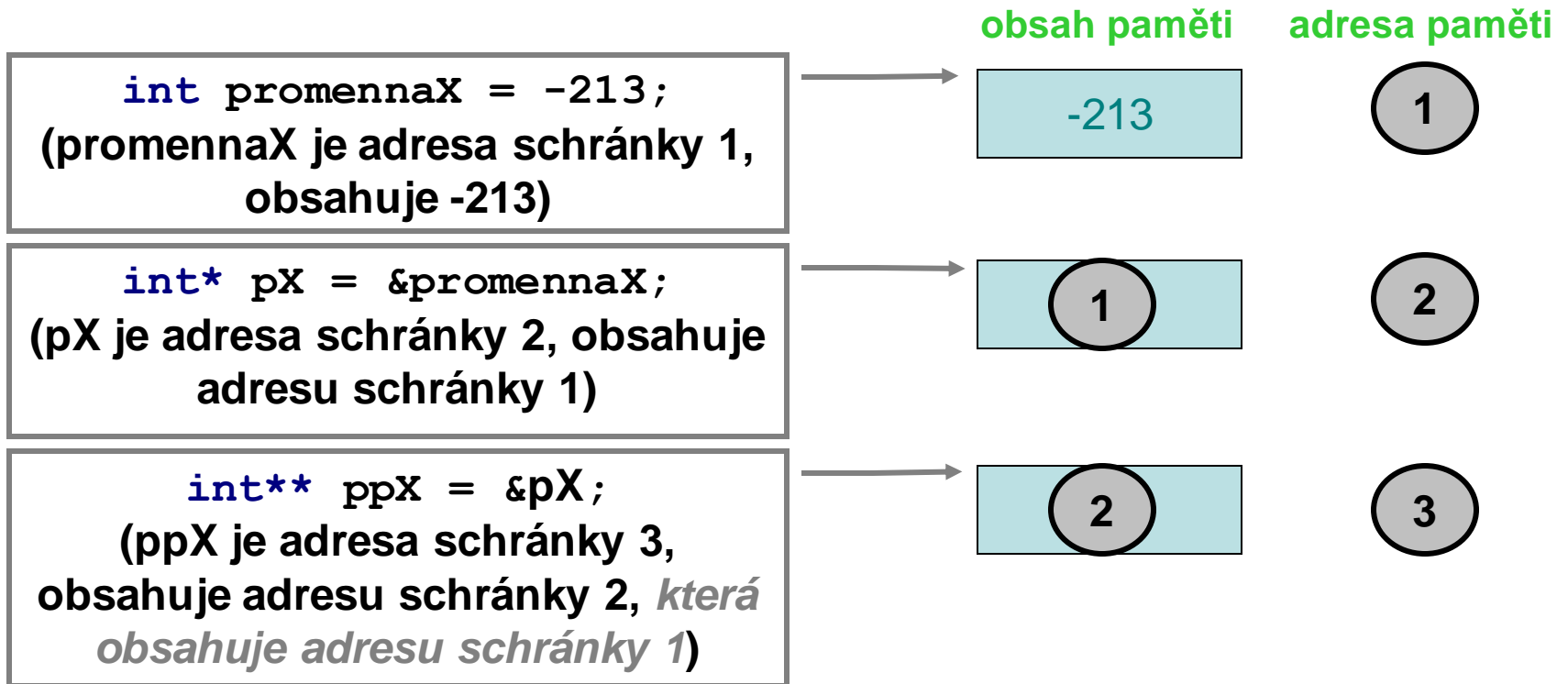
- `lea 0x8(%esp), %eax`
- `mov %eax, 0xc(%esp)`



Operátor dereference *

- Pracuje nad proměnnými typu ukazatel
 - “podívej” se na adresu kterou najdeš v proměnné X jako na hodnotu typu Y
- Zpřístupní hodnotu, na kterou ukazatel ukazuje
 - nikoli vlastní hodnotu ukazatele (což je adresa)
 - ukazatel ale může ukazovat na hodnotu, která se interpretuje zase jako adresa (např. `int**`)
- Příklady (pseudosyntaxe, `=>` označuje změnu typu výrazu):
 - `&int => int*`
 - `*(int*) => int`
 - `*(int**) => int*`
 - `** (int**) => int`
 - `*(&int) => int`
 - `** (&&int) => int`
- Pokud je dereference použita jako l-hodnota, tak se mění odkazovaná hodnota, nikoli adresa ukazatele
 - `int* pX; pX = 10;` (typicky špatně – nechceme měnit ukazatel)
 - `int* pX; *pX = 10;` (typicky OK – chceme měnit hodnotu, která je na adrese na kterou pX ukazuje)

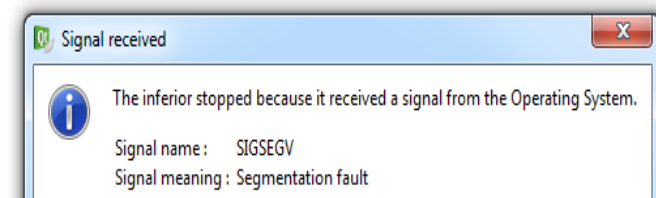
Proměnné a ukazatele - ukázka



- `print(promennaX);` vypíše? (-213)
- `print(pX);` vypíše? 1
- `print(ppX);` vypíše? 2
- `print(*pX);` vypíše? (-213)
- `print(*ppX);` vypíše? 1
- `print(&pX);` vypíše? 2
- `print(*(&pX));` vypíše? 1

Segmentation fault

- Proměnná typu ukazatel obsahuje adresu
 - `int promennaX = -213; int* pX = &promennaX;`
- Adresa nemusí být přístupná našemu programu
 - `pX = 12345678; // nejspíš není naše adresa`
- Při pokusu o přístup mimo povolenou paměť výjimka
 - `*pX = 20;`
 - segmentation fault



Více ukazatelů na stejné místo v paměti

- **Není principiální problém**
 - `int promennaX = -213;`
 - `int* pX = &promennaX;`
 - `int* pX2 = &promennaX;`
 - `int* pX3 = &promennaX;`
- **Všechny ukazatele mohou místo v paměti měnit**
 - `*pX3 = 1; *pX = 10;`
- **Je nutné hlídat, zda si navzájem nepřepisují**
 - logická chyba v programu
 - problém při použití paralelních vláken

Častý problém

- V čem je problém s následujícím kódem?

```
int* a, b;  
a[0] = 1;  
b[0] = 1;
```

- Specifikace ukazatele * se vztahuje jen k první proměnné (a) ne již ke druhé (b)
 - Ekvivalentní k `int *a; int b;`
 - Raději deklarujte a i b na samostatném řádku
- Ukazatel není inicializován => smetí v paměti
 - Inicializujte
 - Zde je již jasná chyba

```
int* a = null;  
int* b = null;  
a[0] = 1;  
b[0] = 1;
```

Samostudium

- Detailnější rozbor zásobníku a haldy
 - <http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory>
 - <http://www.inf.udec.cl/~leo/teoX.pdf>

Problém s předáváním hodnotou

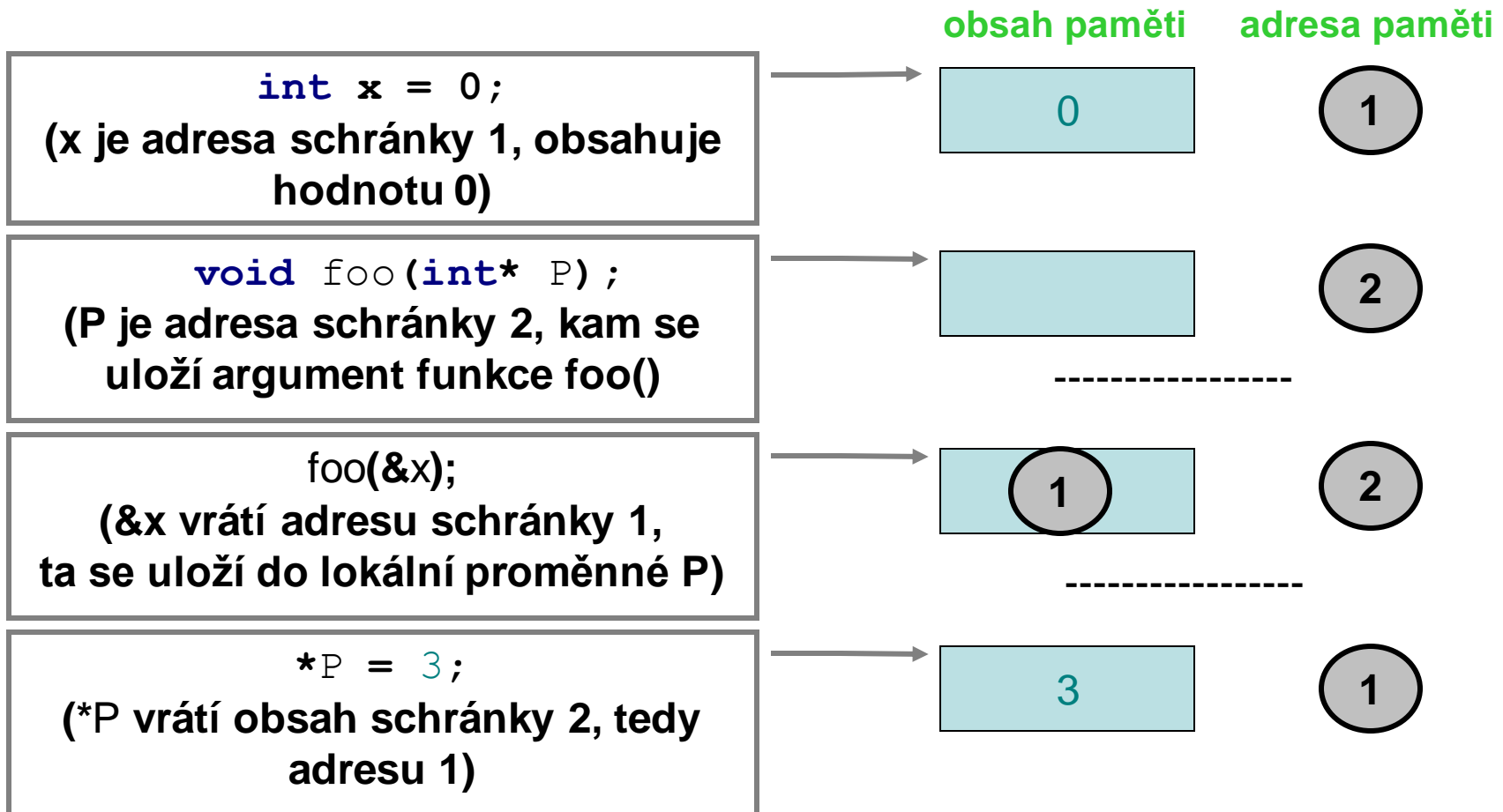
```
void foo(int X) {  
    X = 3;  
    // X is now 3  
}  
  
int main() {  
    int variable = 0;  
    foo(variable);  
    // ☹ x is (magically) back to 0  
    return 0;  
}
```

```
void foo(int* P) {  
    *P = 3;  
}  
  
int main() {  
    int x = 0;  
    foo(&x);  
    // x is 3  
    return 0;  
}
```

- Po zániku lokální proměnné se změna nepropaguje mimo tělo funkce
- Řešením je předávat adresu proměnné
 - a modifikovat hodnotu na této adrese, namísto lokální proměnné

Předáváním hodnotou ukazatele

```
void foo(int* P) {  
    *P = 3;  
}  
  
int main() {  
    int x = 0;  
    foo(&x);  
    return 0;  
}
```



Předávání hodnotou a hodnotou ukazatele

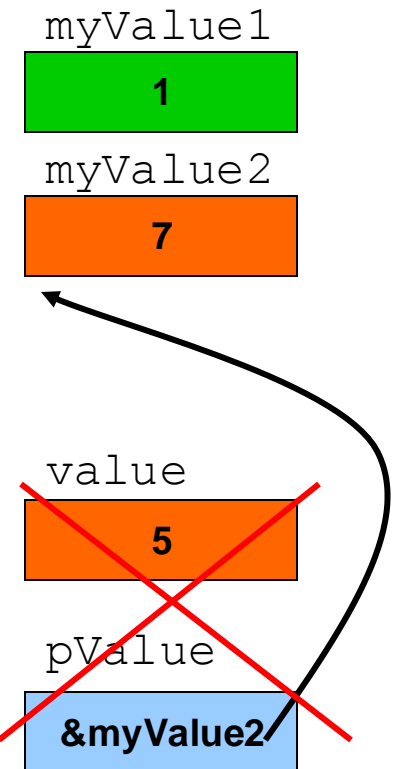
Zásobník

```
int main() {
    int myValue1 = 1;
    int myValue2 = 1;

    valuePassingDemo(myValue1, &myValue2);

    return 0;
}

void valuePassingDemo(int value, int* pValue) {
    value = 5;
    *pValue = 7;
}
```



- Proměnná `value` i `pValue` zaniká, ale zápis do `myValue2` zůstává

PB071 Prednaska 03 – Ukazatel, předávání argumentů funkce

The image shows the Kahoot! logo in a large, white, 3D-style font with a slight shadow. 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 bottom-left half (the last two columns) is green. The top-right half (the first two columns) is light blue, and the bottom-right half (the last two columns) is a darker green. The text 'Kahoot!' is positioned in the center, spanning across the middle of the grid.

Kahoot!

Jednorozměrné pole

Jednorozměrné pole

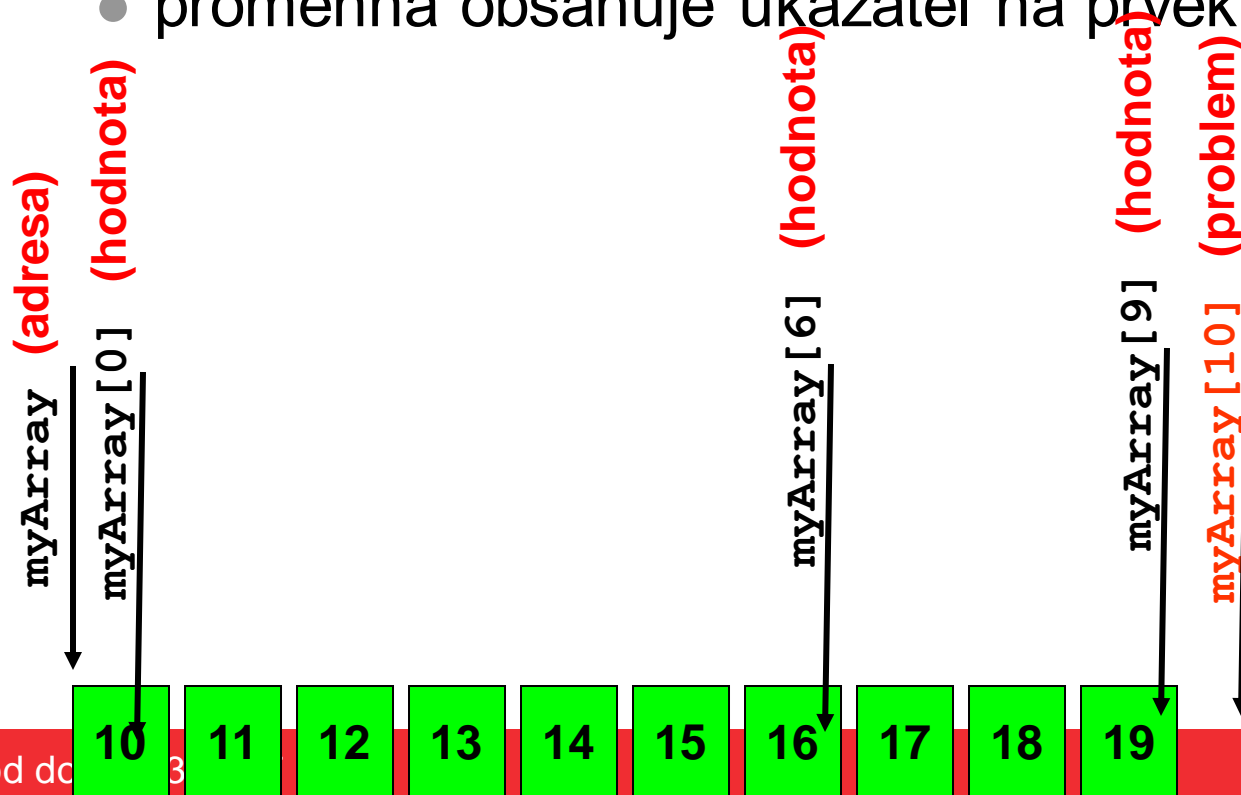
- Jednorozměrné pole lze implementovat pomocí ukazatele na souvislou oblast v paměti
 - jednotlivé prvky pole jsou v paměti za sebou
 - první prvek umístěn na paměťové pozici uchovávané ukazatelem
- Prvky pole jsou v paměti kontinuálně za sebou
- Deklarace: `datový_typ jméno_proměnné[velikost];`
 - velikost udává počet prvků pole

0x00	0x00	0x2b	0x00	0xff	0xff	0xff	0x2b	0x00	0x00
------	------	------	------	------	------	------	------	------	------

```
int myArray[10];  
for (int i = 0; i < 10; i++) {myArray[i] = i+10;}
```

Jednorozměrné pole

- Syntaxe přístupu: `jmeno_proměnné[pozice]` ;
 - na prvek pole se přistupuje pomocí operátoru `[]`
 - indexuje se od 0, tedy n-tý prvek je na pozici `[n-1]`
 - proměnná obsahuje ukazatel na prvek `[0]`



Zjištění velikosti pole

- Jak zjistit, kolik prvků se nachází v poli?
- Jak zjistit, kolik bajtů je potřeba na uložení pole?
- Jazyk C obecně neuchovává velikost pole
 - Např. Java uchovává prostřednictvím `pole.length`
- Velikost pole si proto musíme pamatovat
 - Dodatečná proměnná
- (V některých případech lze využít operátor **sizeof**)
 - Pozor, nefunguje u ukazatelů - vrátí velikost ukazatele, ne pole
 - Pozor, funguje jen u pole deklarovaného s pevnou velikostí
 - Pozor, nefunguje u pole předaného do funkce
 - Nespoléhejte, pamatujte si velikost v separátní proměnné.

Využití operátoru sizeof()

- `sizeof(pole)`
 - velikost paměti obsazené polem v bajtech
 - funguje jen pro statická pole, jinak velikost ukazatele
- `sizeof(ukazatel)`
 - velikost ukazatele (typicky 4 nebo 8 bajtů)

 Pozor, pole v C nehlídá meze!


- čtení/zápis mimo alokovanou paměť může způsobit pád nebo nežádoucí změnu jiných dat
- `int array[10]; array[100] = 1; // runtime exception`

Práce s poli

Jednorozměrné pole

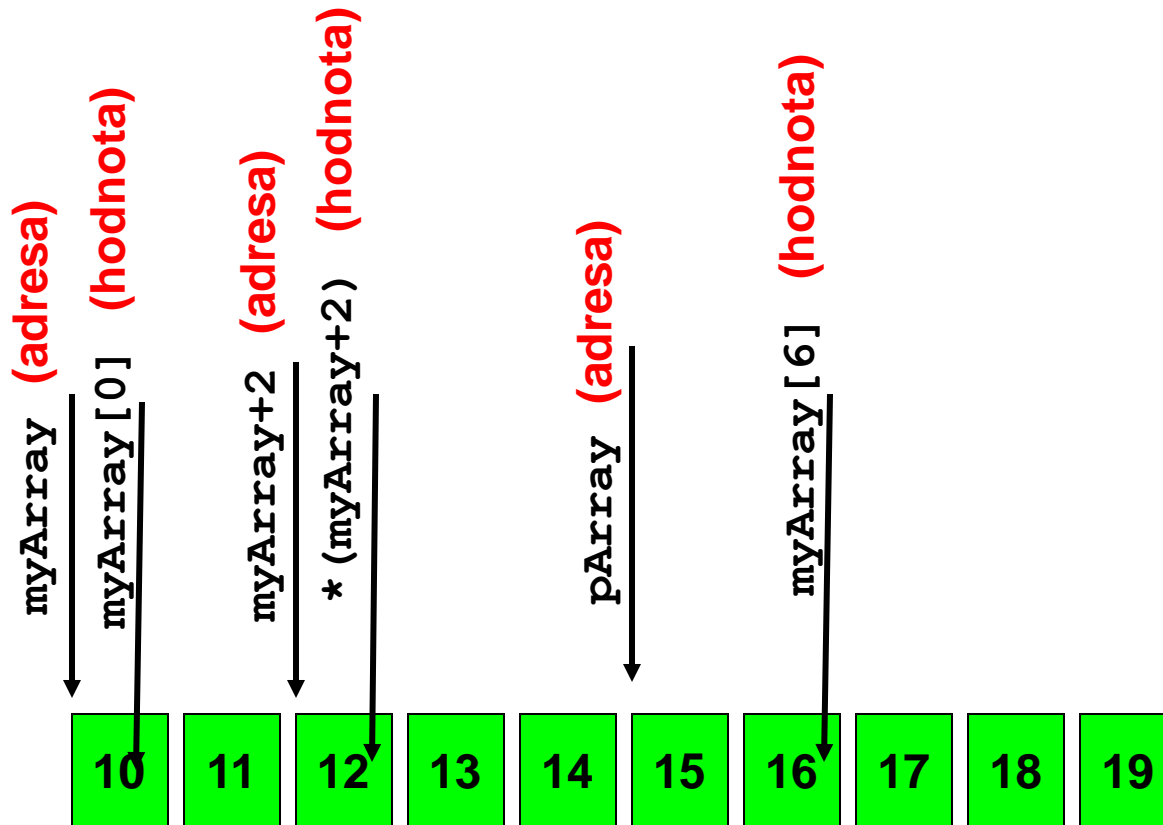
- Jednorozměrné pole lze implementovat pomocí ukazatele na souvislou oblast v paměti
 - `int array[10];`
- Jednotlivé prvky pole jsou v paměti za sebou
- Proměnná typu pole obsahuje adresu prvního prvku pole
 - `int *pArray = array;`
 - `int *pArray = &array[0];`
- Indexuje se od 0
 - n-tý prvek je na 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!)

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/>

Shrnutí

- Funkce
 - Podpora strukturovaného programování
- Ukazatel
 - Principiálně jednoduchá věc, ale časté problémy
- Předávání hodnotou resp. ukazatelem
 - Důležitý koncept, realizace v paměti
- Pole
 - Opakování datového prvku v paměti za sebou
 - Přístup pomocí ukazatele na první prvek

