PB071 – Principy nízkoúrovňového programování

Funkce, modulární programování, paměť a ukazatel

Slidy pro komentáře (děkuji!): **https://drive.google.com/file/d/15O0epfoXDiwWfgpq_8caEeWZoKiukzdl/view?usp=sharing**

http://dkdavis.weebly.com

Neopisujte

- Co je OK
  - Společně diskutujete ústně možnosti řešení
  - Společně diskutujete včetně využití tabule, z tabule ale neopisujete ani si ji nefotíte (pochopení v hlavě)
  - Necháte si poradit od konzultanta
- Co není OK
  - Pošlete svůj kód kolegovi
    - Např. sdílíte svoje IDE s domácím úkolem přes Discord!
  - Najdete kód na internetu a zkopírujete ho (Ctrl+C, přepis)
  - Napíšete kus kódu kolegovi nebo diktujete co má psát
  - … (pokud si nejste jisti, zeptejte se nás)

Obsah přednášky

- Jak dělit kód na spravovatelné části
  - Funkce, deklarace, definice, předávání argumentů
- Jak dělit projekt na spravovatelné části
  - Moduly, hlavičkové soubory
- Jak uložit, najít, a odkazovat objekty v paměti
  - Realizace objektů v paměti, zásobník, ukazatel * &, předávání objektů do funkcí
- Jednorozměrné pole
  - Objekty v paměti těsně za sebou a práce s nimi

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

```
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);
```

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
  - (globální proměnná - nepoužívejte)
  - (vedlejším efektem funkce – např. výpis na obrazovku, packet…)

Funkce – návratová hodnota

- Funkce nemusí vracet hodnotu
  - deklarujeme 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, int * div, int * rem) {
    *div = a / b;
    *rem = a % b;
}
int main() { divRem(7, 3); return 0; }
```

```
nt div = 0;
nt rem = 0;
oid divRem(int a, int b) {
    div = a / b;
    rem = a % b;
}
nt 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 (knihovních) funkcí
  - např. knihovna pro práci se vstupem a výstupem
- Logicky související funkce můžeme vyčlenit do samostatných souborů
  - deklarace knihovních funkcí do hlavičkového souboru (*.h)
    - Jaké argumenty bude funkce přijímat a jaké hodnoty vracet
  - implementace do zdrojového souboru (*.c)
    - Jak přesně je tělo funkce implementované

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 (knihovních) funkcí
  - např. knihovna pro práci se vstupem a výstupem
- Logicky související funkce můžeme vyčlenit do samostatných souborů
  - deklarace knihovních funkcí do hlavičkového souboru (*.h)
    - Jaké argumenty bude funkce přijímat a jaké hodnoty vracet
  - implementace do zdrojového souboru (*.c)
    - Jak přesně je tělo funkce implementované

Hlavičkové soubory – rozhraní (*.h)

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

- 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 překladu
  - (preprocessing -E)
  - obsah hlavičkového souboru vloží namísto #include
  - ochranné makro #ifndef HEADERNAME_H

stdio.h

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

Implementace funkcí z rozhraní (*.c)

- 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 –E, gcc-S, as → library.o

```
assembler code
implementace funkce foo()
```

gcc –S, as → main.o

```
assembler code
adresa funkce foo() zatím
nevyplněna
```

gcc main.o library.o → 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 fahrenheita
       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 fahrenheita
               printf("%3d \t %6.2f \n", fahr, celsius);
       }
       return 0;
}
```

Dělení kódu na části - ukázka

1. Převod F na C vyjmeme 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
```

Tzv. Ochranná makra proti násobnému vložení obsahu stejného hlavičkového souboru

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);
}
```

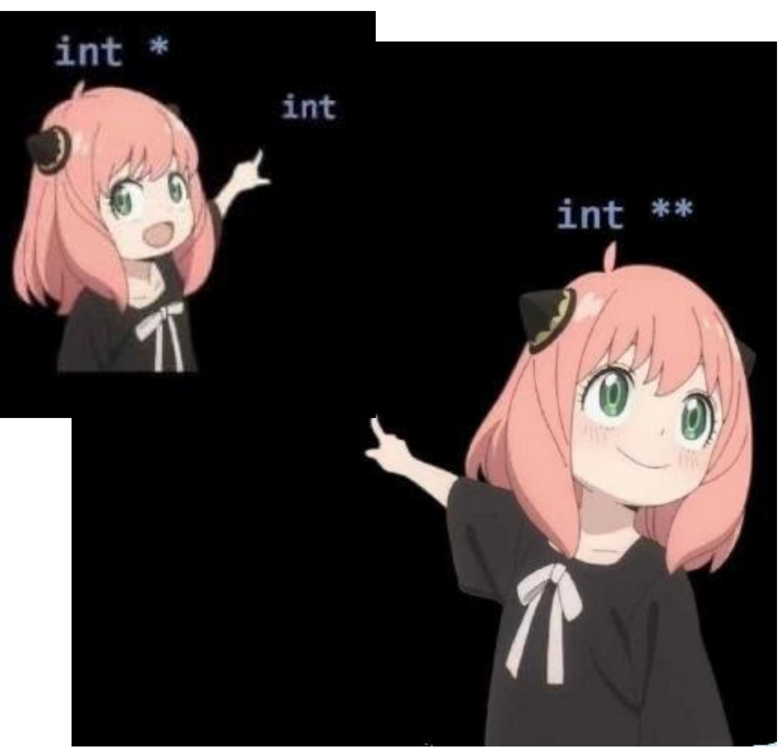
gcc –std=c99 .... –o binary **main.c converse.c**

`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

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 nespoléhejte, 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.

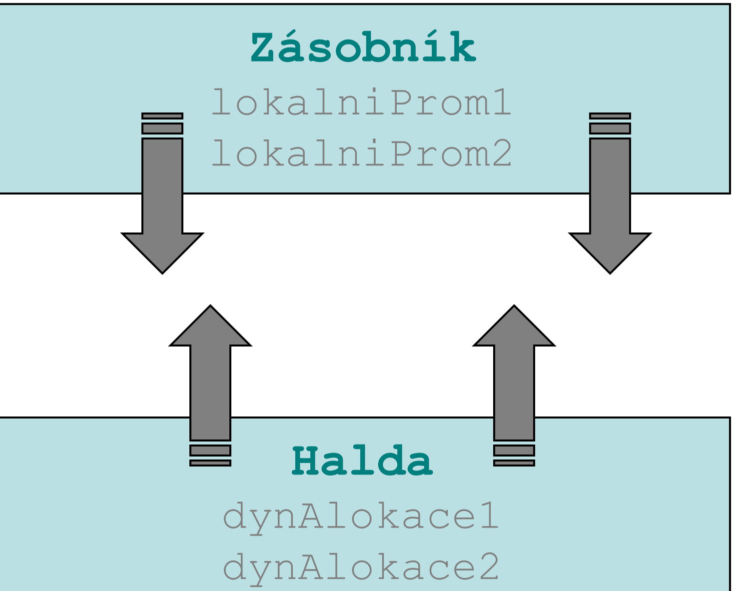
Celková paměť programu

Instrukce

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

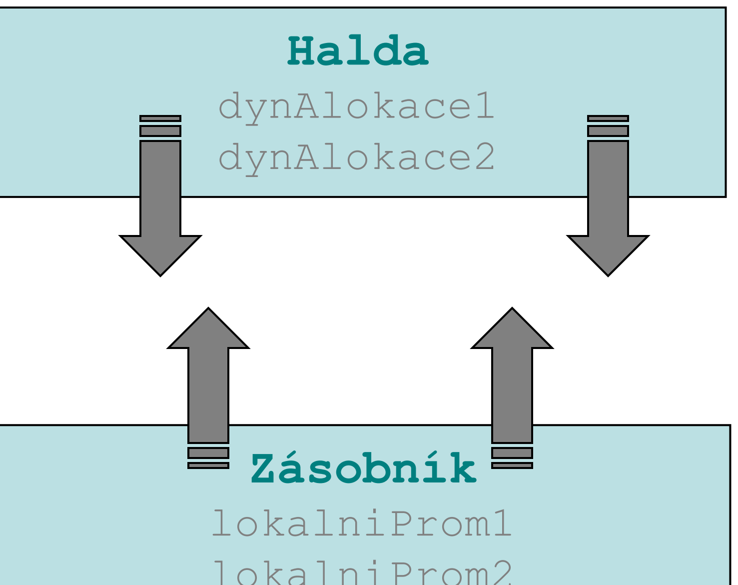
Statická a glob. data

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

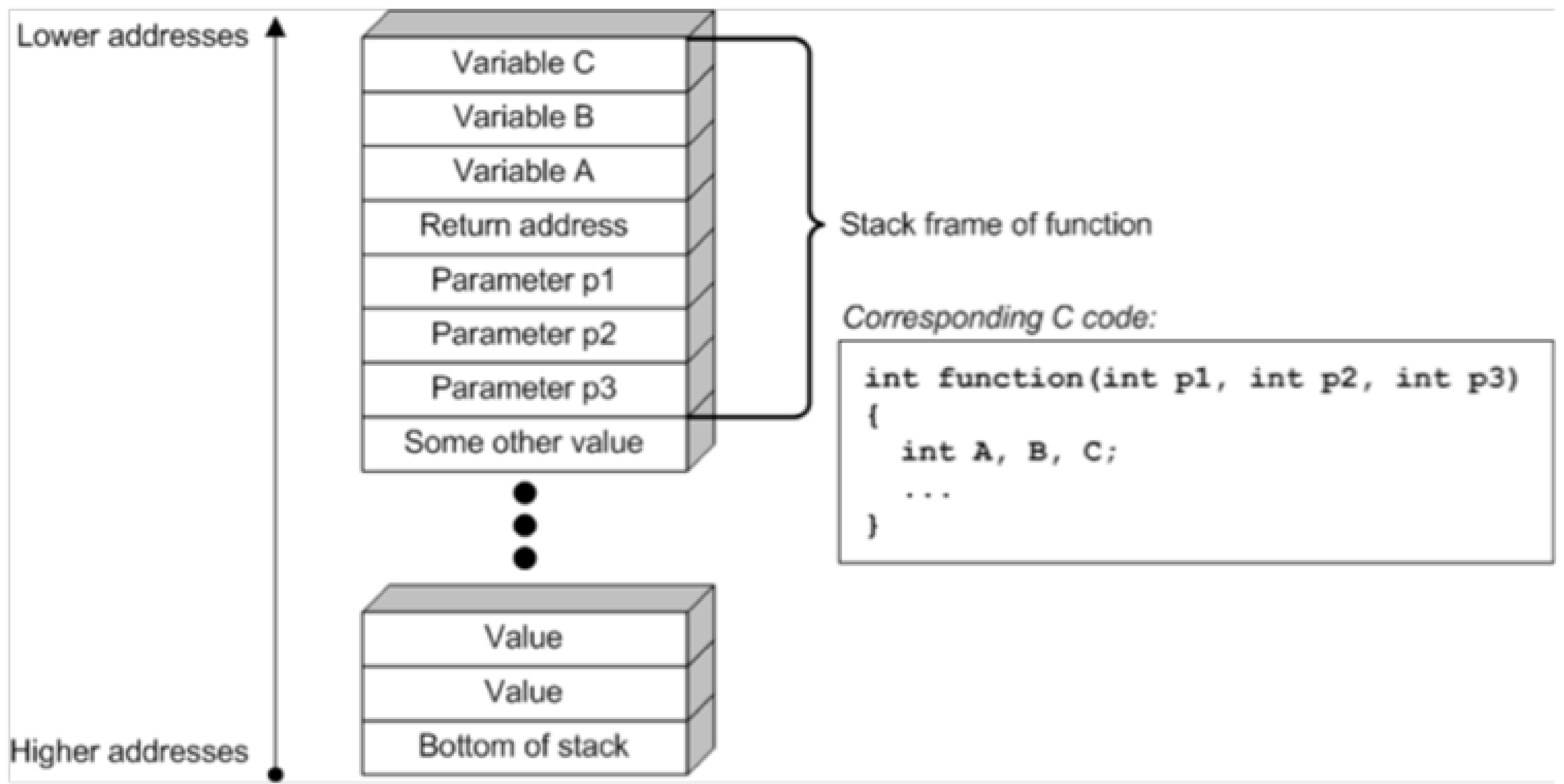


Organizace paměti

- 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.



Rámec na zásobníku (stack frame)



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

...
3. položka
2. položka
1. položka
Dno

Směr růstu zásobníku a haldy

- Učebnicové zobrazení růstu zásobníku je „zespodu nahoru“
- Při každém zavolání funkce se vytváří nový rámec na zásobníku (stack frame) s proměnnými
- Co to ale znamená pro adresy umístěných položek?
  - Bude mít 1. položka číselně vyšší nebo nižší adresu než 2. položka?
- Záleží na překladači a platformě Application Binary Interface (ABI)
- Růst směrem dolů (k nule) je častější
  - x86/x64, PowerPC, MIPS, SPARC, EE, Cell SPU
- Lze snadno zjistit (porovnání adres proměnných)
  - `int a = 0; printf("%-20s: %p\n", "a", &a);`
  - Lokální proměnná v main() a následně v zavolané funkci foo()
  - Pozice haldy (dyn. alokace) dle adresy z malloc()
- https://stackoverflow.com/questions/1677415/does-stack-grow-upward-or-downward

Test směru růstu zásobníku

```
temp_in_main          : 0x00F3FD84
temp1_in_foo1         : 0x00F3FCA4
temp2_in_foo2         : 0x00F3FBC4
Fnc foo1()            : 0x00701294
globalVar             : 0x0070A014
```

```c
#include <stdio.h>
int globalVar = 1;
void foo2() {
  int temp2 = 0;
  printf("%-20s: 0x%p\n", "temp2_in_foo2", &temp2);
}
void foo1() {
  int temp1 = 0;
  printf("%-20s: 0x%p\n", "temp1_in_foo1", &temp1);
  foo2();
}
int main() {
  int temp = 0;
  printf("%-20s: 0x%p\n", "temp_in_main", &temp);
  printf("%-20s: 0x%p\n", " foo1()", &foo1);
  foo1();
  printf("%-20s: 0x%p\n", "Fnc foo1()", &foo1);
  printf("%-20s: 0x%p\n", "globalVar", &globalVar);
  return 0;
}
```

Proměnná – opakování

- Každé místo v paměti má svou adresu
- Pomocí názvu proměnné můžeme číst nebo zapisovat do této paměti
  - např. `promenna = -213;`
- Překladač nahrazuje jméno proměnné její adresou
  - typicky relativní k zásobníku
  - movl `$0xffffff2b,0xc(%esp)`

instrukce assembleru pro zápis do paměti

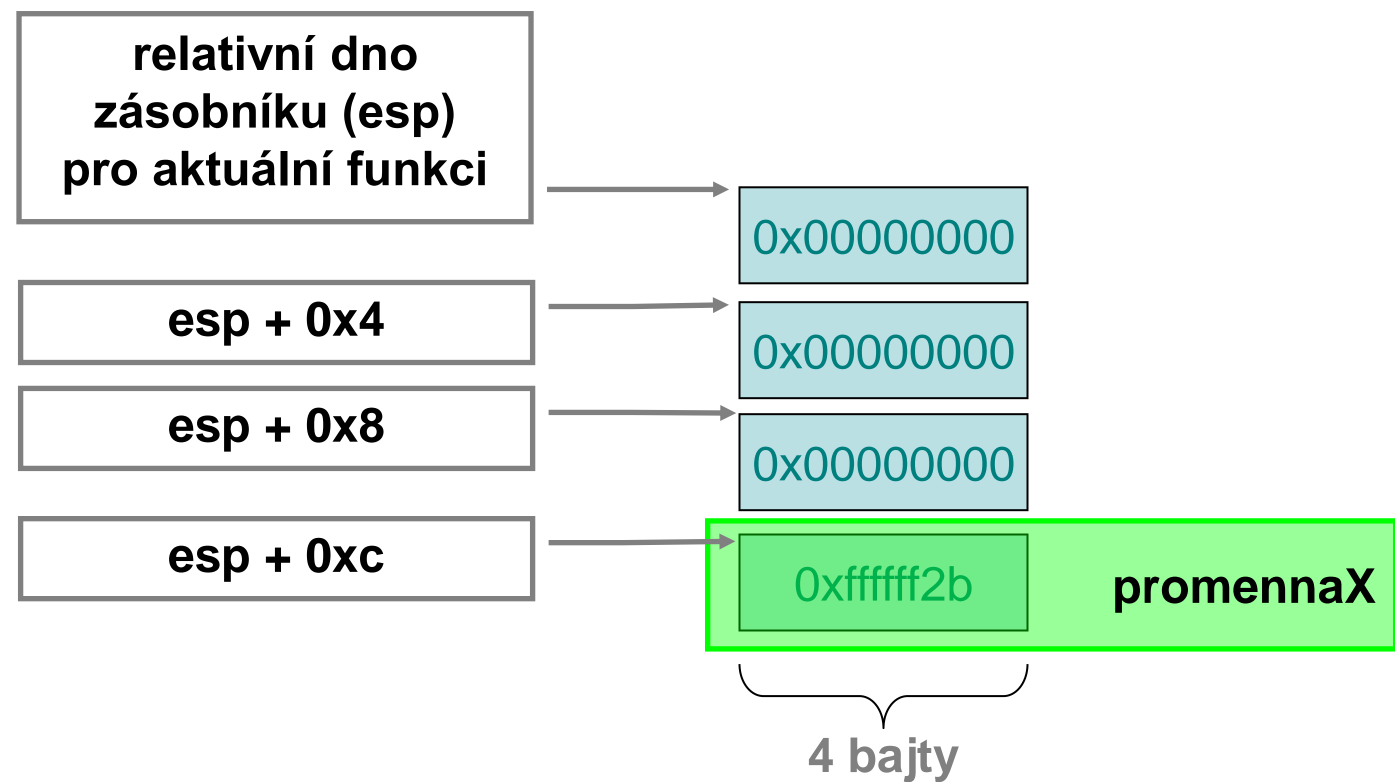
relativní adresa proměnné k adrese zásobníku

-213 hexadecimálně

počáteční adresa zásobníku pro danou funkci

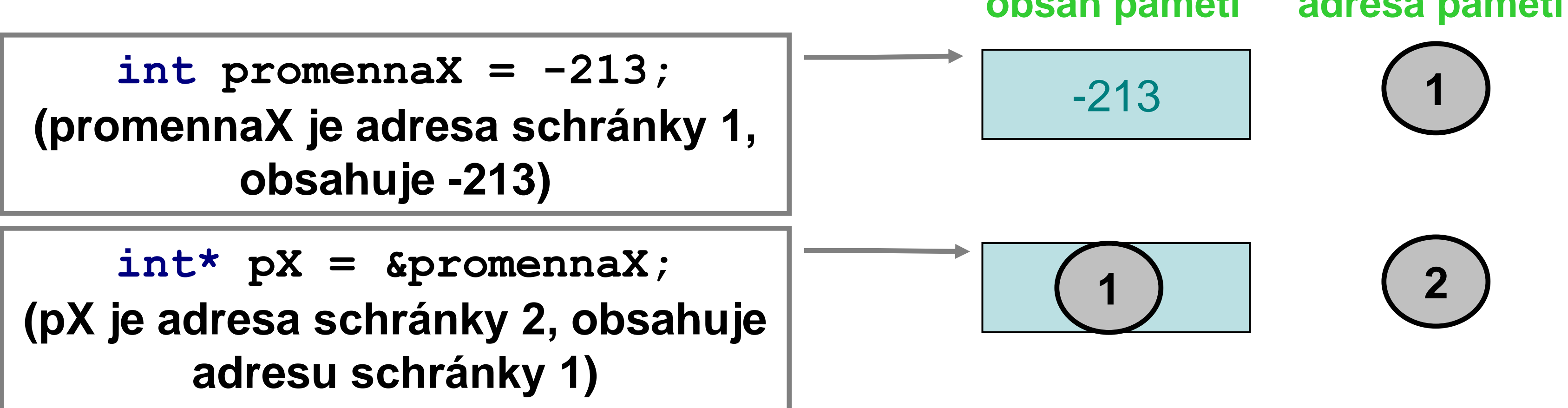
Proměnná na zásobníku

- C kód: `int promennaX = -213;`
- Asembler kód: `movl $0xffffff2b,0xc(%esp)`



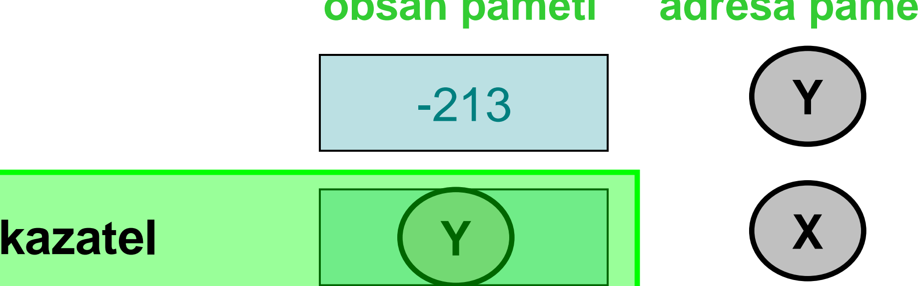
Operátor &

- 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;`

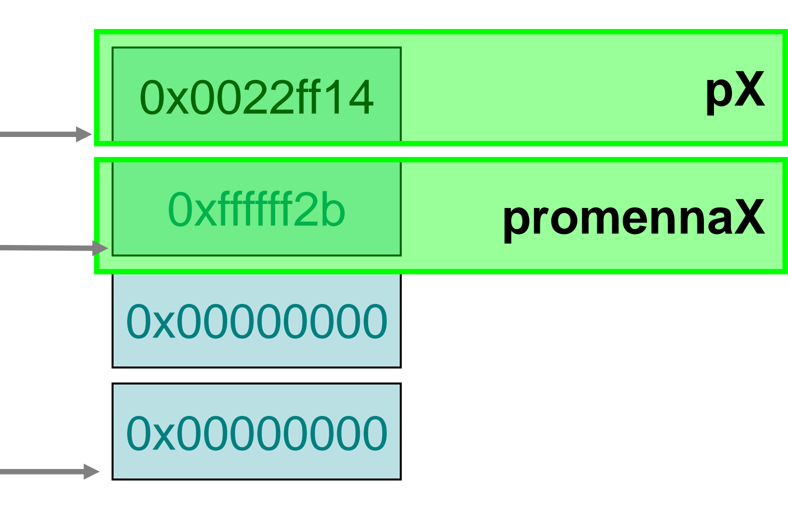
operátor & vrátí adresu argumentu (zde promennaX)

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

esp + 0xc
absolutní adresa 0x0022ff18

esp + 0x8
absolutní adresa 0x0022ff14

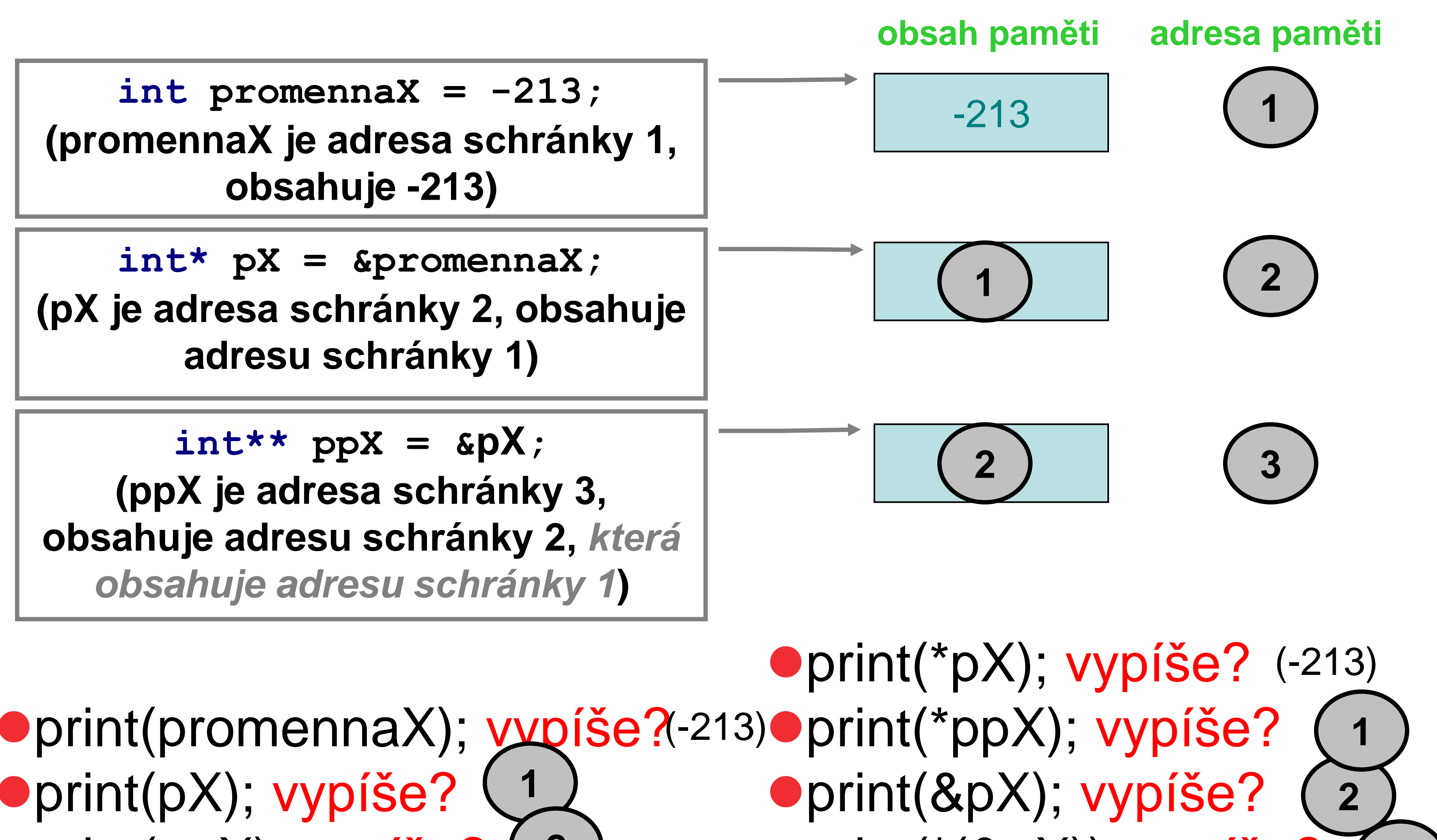
relativní dno zásobníku (esp)
pro aktuální funkci
absolutní adresa 0x0022ff0c



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



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

Signal received

The inferior stopped because it received a signal from the Operating System.

Signal name : SIGSEGV
Signal meaning : 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, vždy inicializujte
  - Zde je již jasná chyba

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

Nulový ukazatel (NULL)

- Dříve uvedená ukázka používala inicializaci s 0
  - `int* ptr = 0;`
- Lze použít, ale lepší by bylo vyjádřit, že vkládáme nulový ukazatel, ne číslo 0
  - `int* ptr = NULL;`
  - Dostupné v hlavičkové souboru stdio.h
    - `#define NULL ((void *)0)`
- Čitelnější a sémanticky čistší
  - `if (ptr == NULL) {…}`

Samostudium

- Detailnější rozbor zásobníku a haldy
  - http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory
  - https://web.archive.org/web/20170829060314/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

Argumenty předávané hodnotou ukazatele

- S využitím ukazatelů můžeme předat výstup přes vstupní argumenty
- Pokud je vstupním argumentem ukazatel:
  1. vznikne lokální proměnná P typu ukazatel
  2. předává se hodnotou, do P zkopíruje se hodnota (== adresa, např. X)
  3. pomocí operátoru dereference * můžeme modifikovat paměť na adrese X (*P == X)
  4. lokální proměnná P na konci funkce zaniká
  5. hodnota na adrese X ale zůstává modifikována
- Změna provedená ve funkci zůstává po jejím ukončení

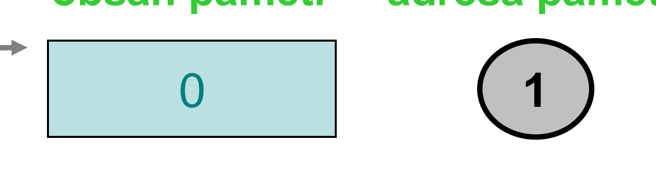
```
void foo(int* P) {
  *P = 3;
}
int main() {
  int x = 0;
  foo(&x);
  return 0;
}
```

Předáváním hodnotou ukazatele -

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

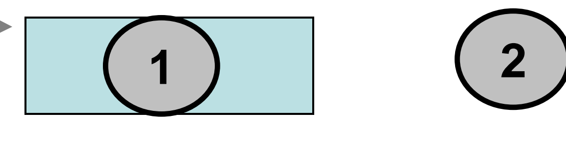
obsah paměti | adresa paměti

`int x = 0;`
(x je adresa schránky 1, obsahuje hodnotu 0)

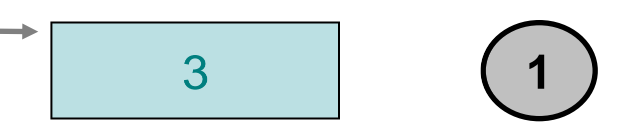


`void foo(int* P);`
(P je adresa schránky 2, kam se uloží argument funkce foo()

`foo(&x);`
(&x vrátí adresu schránky 1, ta se uloží do lokální proměnné P)



`*P = 3;`
(*P vrátí obsah schránky 2, tedy adresu 1)



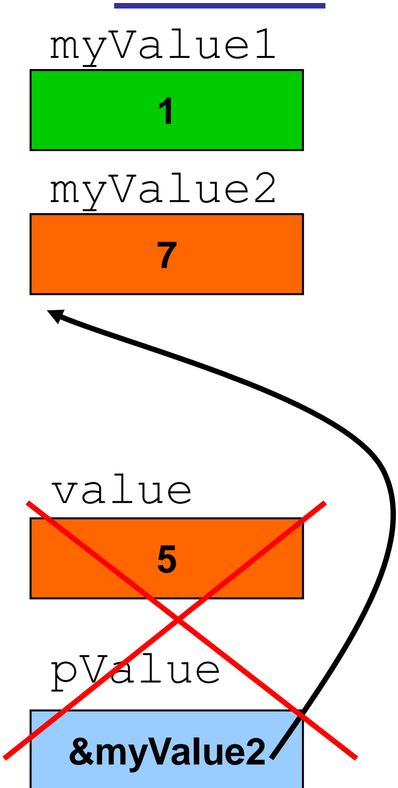
Předávání hodnotou a hodnotou ukazatele

```
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

```
void valuePassingDemo(int value, int* pValue) {
...
   value = 5;
003F1821  mov        dword ptr [value],5
   *pValue = 7;
003F1828  mov        eax,dword ptr [pValue]
003F182B  mov        dword ptr [eax],7
}

int main(void) {
...
   int myValue1 = 1;
00031895  mov        dword ptr [myValue1],1
   int myValue2 = 1;
0003189C  mov        dword ptr [myValue2],1

   valuePassingDemo(myValue1, &myValue2);
000318A3  lea        eax,[myValue2]
000318A6  push       eax
000318A7  mov        ecx,dword ptr [myValue1]
000318AA  push       ecx
000318AB  call       valuePassingDemo (03138Eh)
000318B0  add        esp,8

...
}
```

https://godbolt.org/

Problém s předáváním hodnotou

- Proměnná X je lokální ve funkci, její změna se nepropaguje
  - Proměnná one zůstane pořád rovna 1
- `return ++X;` (Prefixový operátor)
  - výsledek ++X je hodnota X po inkrementu a to se vrátí pomocí return (takže ret1 == 2)
- `return X++;` (Postfixový operátor)
  - Výsledek X++ je hodnota X před inkrementem a to se vrátí pomocí return (takže ret2 == 1)
  - X se inkrementuje, ale protože je lokální proměnná, tak hned v zápětí na konci funkce zaniká

```
int foo1(int X) {
   return ++X;
}
int foo2(int X) {
   return X++;
}
int main() {
   int one = 1;
   int ret1 = foo1(one);
   int ret2 = foo2(one);
   return 0;
}
```

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: `jméno_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]`

myArray (adresa)
myArray[0] (hodnota)
myArray[6] (hodnota)
myArray[9] (hodnota)
myArray[10] (problem)

10	11	12	13	14	15	16	17	18	19

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, ne 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; // undefined behavior,
                                   // likely runtime exception
                                   // (like SIGSEGV)
    ```

Připomenutí: `sizeof(type) == sizeof(`proměnná toho `typu)`
    ```

Pole vs. hodnota

- Jaký je rozdíl mezi ukazatelem na místo v paměti obsahujícím hodnotu typu integer a proměnnou typu pole integerů?
  - `int myInt = -213;`
  - `int* pMyInt = &myInt;`
  - `int myIntArray[100];`
- Jméno pole bez `[]` vrací adresu na začátek pole
- `int*` a `int[10]` jsou rozdílné datové typy
  - Obě proměnné s těmito typy vrátí ukazatel do paměti
  - Velikost celého pole `int[10]` je známa v době překladu
- Co vrací `&myIntArray` ?
  - Ověřte si samostatně ve vlastním kódu

Jaktože `&array == array` ?

```
int main() {
    char     array[10];
    char*    array2 = 0;

    printf("array=%p\n", array);
    printf("&array=%p\n", &array);
    printf("array2=%p\n", array2);
    printf("&array2=%p\n", &array2);
    array2 = array;
    printf("array2=%p\n", array2);
    printf("&array2=%p\n", &array2);

    printf("tmp=%p, %d\n", array+1); // address of second element
    printf("tmp=%p, %d\n", &array+1);// == array + sizeof(char) * 10
    return 0;
}
```

```
array=0x0028FF18
&array=0x0028FF18
array2=0x00000000
&array2=0x0028FF14

array2=0x0028FF18
&array2=0x0028FF14
```

&array je ukazatel na typ `char (*) [10]` takže při inkrementu se skočí na další prvek – ale prvkem je celé pole (10x sizeof(char))!

- Jméno pole `array` se obvykle vyhodnotí na adresu prvního prvku pole (pozor, závisí na překladači)
  - `&array == array`
  - `Ale array+1 != &array+1` (pokud je pole delší jak jeden element)
- http://stackoverflow.com/questions/2528318/c-how-come-an-arrays-address-is-equal-to-its-value

Shrnutí

- Funkce
  - Podpora strukturovaného programování
- Ukazatel
  - Principiálně jednoduchá věc, ale časté problémy
- Předávání hodnotou resp. hodnotou ukazatele
  - Důležitý koncept, realizace v paměti

Top questions (1)

Anonymous 0

Pokud přeložím program překladačem gcc na Linuxu, bude spustitelný i na Windows?

Join at **slido.com**

#pb071_2024

- **Pokud budete mít k přednášce dotaz, tak jej vložte na slido.com (#pb071_2024)**

DÍKY ZA VÁŠ ČAS A V PONDĚLÍ ZASE NA VIDĚNOU