Programovací jazyk C(++)

2. Ukazatele, pole, řetězce

Miroslav Spousta, 2004

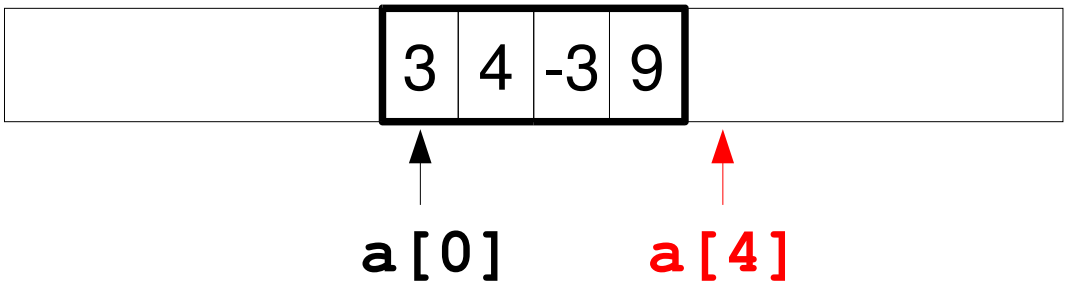
```
static struct vm_area_struct * unmap_fixup(struct mm_struct *mm,
        struct vm_area_struct *area, unsigned long addr, size_t len,
        struct vm_area_struct *extra)
{
        struct vm_area_struct *mpnt;
        unsigned long end = addr + len;

        area->vm_mm->total_vm -= len >> PAGE_SHIFT;
        if (area->vm_flags & VM_LOCKED)
                area->vm_mm->locked_vm -= len >> PAGE_SHIFT;

        /* Unmapping the whole area. */
        if (addr == area->vm_start && end == area->vm_end) {
                if (area->vm_ops && area->vm_ops->close)
                        area->vm_ops->close(area);
                if (area->vm_file)
                        fput(area->vm_file);
                kmem_cache_free(vm_area_cachep, area);
                return extra;
        }
```

C++

Pole

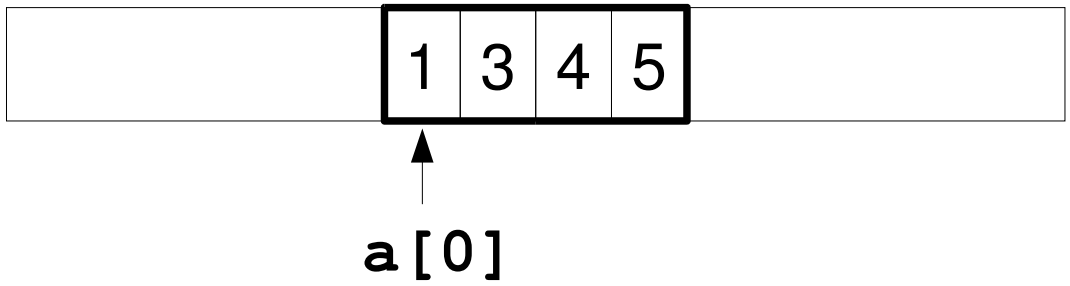
- pole je několik proměnných stejného typu za sebou v paměti
- k jednotlivým prvkům se přistupuje pomocí indexů
- indexy pole vždy začínají od 0 (!)
- při přístupu do pole se nekontrolují meze!
- definice: **`int a[4];`**

  pole o čtyř prvcích s indexy 0 … 3



Inicializace pole

- pole je možné inicializovat:
- definice: `int a[4] = { 1, 3, 4, 5 }`



- prvků v inicializaci musí být méně nebo rovno počtu prvků v poli

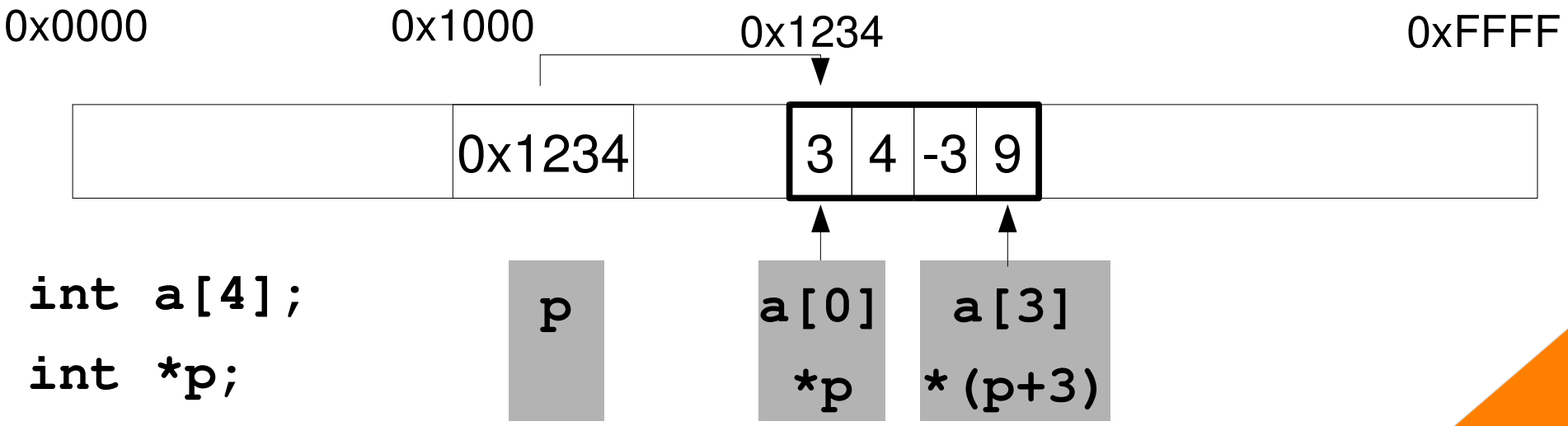
  je-li jich méně, kompilátor doplní zbývající nulami

- je možné vynechat počet prvků pole: `int a[] = { 1, 3, 4, 5 }`
- celé pole není možné přiřazovat (!)

  přiřadit po prvcích v cyklu nebo přiřazovat struktury

Ukazatele

- neboli pointery, vyšší dívčí jazyka C, jeho největší síla
- proměnná, která ukazuje na jinou proměnnou, např. **`int *p;`**
- obecnější než pole
- ukazatel má přiřazen typ proměnné, na kterou ukazuje (případně **`void`**)
- pointerová aritmetika (něco jako indexace pole)



Ukazatele

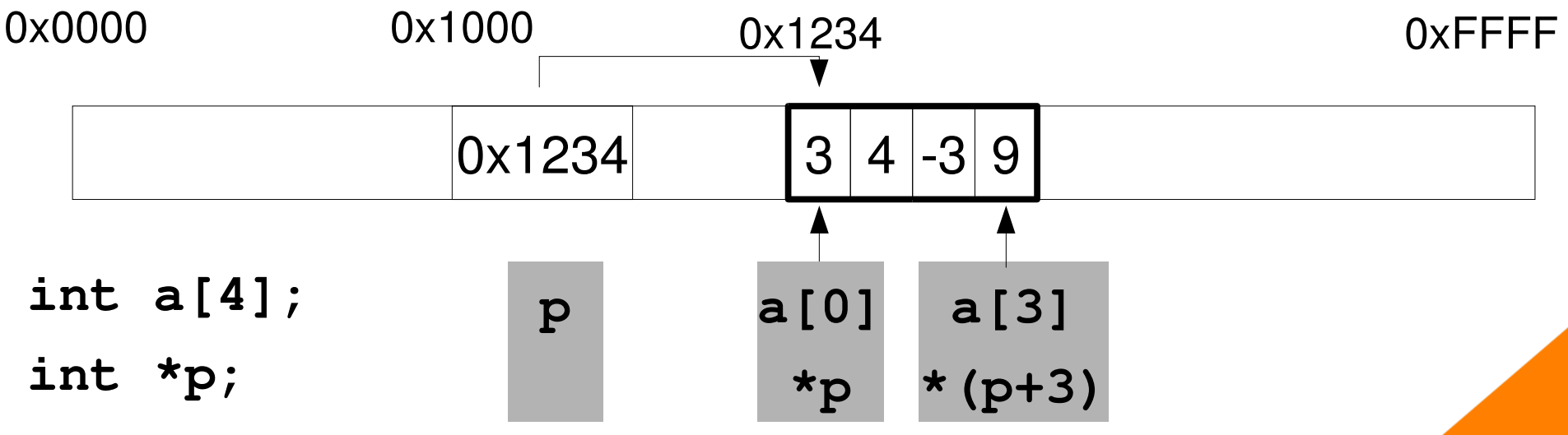
- ukazatel míří někam do paměti (na nějakou adresu)

  nebo má hodnotu 0 (NULL)

  adresa proměnné se zjistí operátorem **&**, např.: **`int i; int *p = &i;`**

- ukazatelé jsou záměnné s polem: pole se dá převést na ukazatel a zpět:

  `p = a; a[3] == *(p+3)`



Příklad

```
/* vytiskne obsah pole prvků typu int */
#include <stdio.h>

void char *printint(int *p, int n)
{
   int i;

   for (i = 0; i < n; i++) {
      printf("%d ", p[i]);
   }
}
```

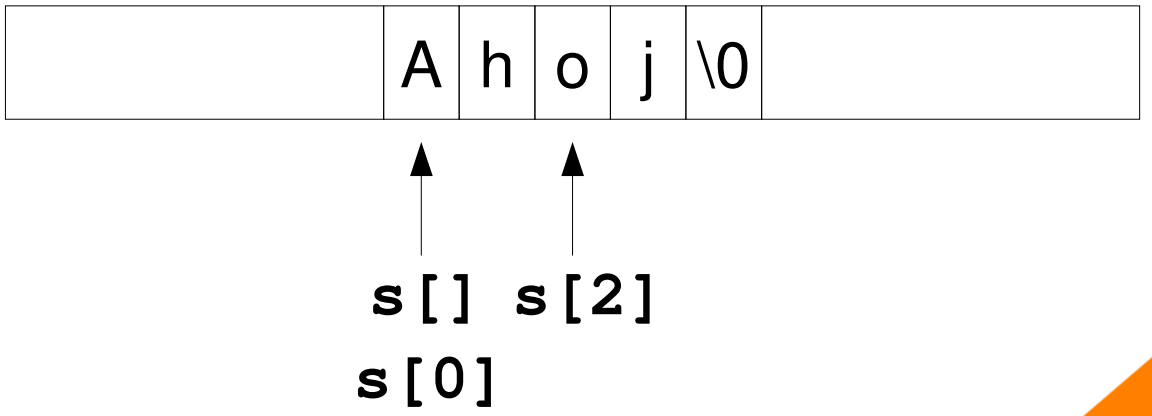
Řetězce

- konstanty: vždy v uvozovkách, v apostrofech jen znaky (typ **`char`**)
- řetězec je pole znaků (**`char[]`**)
- řetězec je zakončený znakem NUL (**`'\0'`**)

  místo potřebné pro řetězec je o 1 větší, než maximální počet znaků!

- v C++ je pro řetězce speciální třída

```
char s[] = "Ahoj";
```



Řetězce: funkce

- `#include <string.h>`

  hlavičkový soubor pro řetězcové funkce

- `char *strcpy(char *kam, char *co)`

  zkopíruje řetězec z adresy co na adresu kam, vrátí odkaz na kam

- `char *strncpy(char *kam, char *co, size_t kolik)`

  jako strcpy, ale kopíruje max. kolik bajtů (pozor na NUL na konci), vrátí odkaz na kam

- `int strcmp(const char *prvy, const char *druhy)`

  lexikograficky porovná dva řetězce, vrátí -1, 0, 1

- `size_t strlen(const char *s)`

  vrátí velikost řetězce (bez ukončovacího znaku NUL)

Znaky: funkce

- `#include <ctype.h>`

  hlavičkový soubor pro standardní konverzní funkce

- protoptyp: `int funkce(int c)`

- `isalnum(), isalpha(), isascii(), isblank(), iscntr(), isdigit(), isgraph(), islower(), isprint(), ispunc(), isspace(), isupper(), isxdigit()`

  funkce otestují, zda argument odpovídá zadanému kritériu

- `tolower(), toupper()`

  zkonvertuje velké znaky na malé (nebo obráceně)

  vrací zkonvertovaný znak

Typové konverze

- explicitní: operátor přetypování: `int i; sin((double)i);`
- implicitní: dělá kompilátor
- před vykonáním opreace se provede:

```
char, short int => int
unsigned char, unsigned short => unsigned int
```

- pokud mají dva argumenty operace různý typ, ten z menší prioritou se převede

```
int => unsigned ing          unsigned long => float
unsigned int => long         float => double
long => unsigned long        double => long double
```

Dynamická alokace paměti

- alokace paměti:

  globální paměť (statická): globální proměnné, statické lokální proměnné

  na **zásobníku**: lokální proměnné

  dynamická: ze zváštní oblasti (heap), je potřeba ji „ručně“ uvolnit

- v jazyce C++: operátory **`new, delete`**

  používá se především ve spojitosti s OOP (volá se konstruktor, případně destruktor)

- v jazyce C: **`malloc(), free()`**

```
#include <stdlib.h>
#include <stdio.h>
char *s; s = malloc(10); strcpy(s, “123456789”);
puts(s); free(s);
```

Funkce main()

- prototyp: **`main(int argc, char *argv[])`**

  argc je počet argumentů, první argument je název programu (!)

  používá se především ve spojitosti s OOP (volá se konstruktor, případně destruktor)

  `argc` je počet argumentů, **`argv`** je pole pointerů na char (stringů)

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   int i;
   for (i = 0; i < argc; i++) {
      puts(argv[i]); puts("\n");
}
return 0;
```