Ukazatele, dynamická alokace

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Přednášky byly připraveny s pomocí materiálů, které vyrobili Ladislav Vágner, Pavel Strnad, Martin Mazanec

Programování v C++, B6B36PJC
03/2017, Lekce 4

https://cw.fel.cvut.cz/wiki/courses/b6b36pjc/start

Ukazatele

- Ukazatel je abstrakce paměťové adresy ve vyšším programovacím jazyce.
- Proměnná typu ukazatel obsahuje adresu, na které se nachází zpřístupňovaná data:
  - proměnná,
  - objekt,
  - funkce,
  - jiný ukazatel.

Ukazatele

- Deklarace proměnné typu ukazatel:

```
int *dataPtr;
```

- Vyhradí v paměti prostor pro adresu:
  - typicky 4B pro 32-bit prostředí,
  - typicky 8B pro 64-bit prostředí.
- Proměnná **`dataPtr`** je ukazatel, který může ukazovat na hodnotu typu **`int`**.
- Deklarací není nastavena adresa – **`dataPtr`** zatím ukazuje "někam do paměti".

Ukazatele - operace

- Dereference (*) – zpřístupnění místa, kam ukazatel směřuje
- Adresa, reference (&) – získání adresy paměťového místa (pořízení ukazatele)

```
int a = 10;
int *p_a = &a; // p_a ukazuje na a
int x = *p_a;  // x je nyní rovno 10
```

Ukazatele - inicializace

```
int ival = 1024;
int *pi = nullptr; // ekvivalentní s pi = NULL, pi = 0
                   // podobné NULL referenci v Javě
int *pi2 = &ival;  // pi2 inicializováno na adresu ival
int *pi3;   // ok, pi3 je neinicializováno
pi = pi2;   // pi a pi2 ukazují na ival
pi2 = NULL; // pi2 je nyní NULL
std::string *ps, str; /* ps je ukazatel na string,
                         str je string */
std::string *ps1, *ps2; /* ps1 i ps2 jsou ukazatele na
                           string */
```

Ukazatele

- Vždy ukazatel inicializujte na adresu proměnné, pokud to je možné.
- Pokud to není možné, inicializujte ukazatel na nulu (`nullptr`, `NULL`, `0`) – pak lze kontrolovat, zda je ukazatel platný.
- Statisticky je neinicializovaný ukazatel jedna z nejčastějších chyb – překladač ji neodhalí.

Ukazatele – hodnoty

- Ukazatel může být inicializován na:
  - na "nulovou" adresu (`0`, `NULL`, `nullptr`),

```
double *pd = nullptr;
```

  - na adresu objektu stejného typu jako je ukazatel,

```
double d = 6.022e23;
double *pd = &d;
```

  - na jiný platný ukazatel stejného typu,

```
double *pd2 = pd;
```

  - na adresu jeden prvek za polem (pro potřeby iterace – nesmí být dereferencován).

Ukazatele – void *

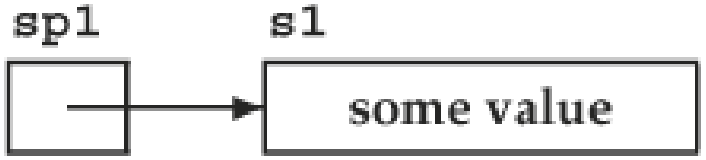
- Může ukazovat na objekt jakéhokoli typu

```
double pi = 3.141592;
void *p_pi = &pi;
```

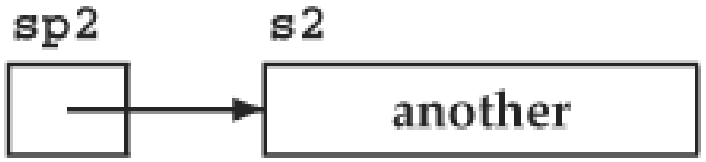
- Abychom mohli přistupovat k objektu na který ukazuje, musíme ho nejdříve přetypovat (viz další část přednášky).

Dereference

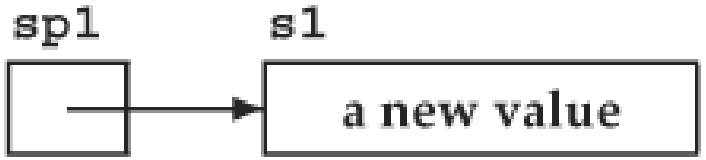
```
string s1("some value");
string *sp1 = &s1;
```



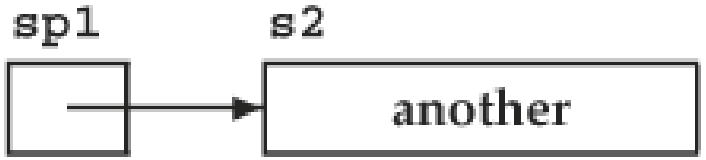

```
string s2("another");
string *sp2 = &s2;
```




```
// assign through sp1
// value in s1 changed
*sp1 = "a new value";
```




```
// assign to sp1
// sp1 points to a different object
sp1 = sp2;
```



zdroj C++ Primer, 4th Edition, Addison-Wesley

Ukazatele vs. reference

- Reference musí vždy odkazovat na nějaký objekt.
- Přiřazení do reference mění odkazovaný objekt. Nemůžeme referenci „donutit“ odkazovat na jiný objekt.

```
int ival = 1024, ival2 = 2048;
int &ri = ival, &ri2 = ival2;
ri = ri2; // změna hodnoty ival
```

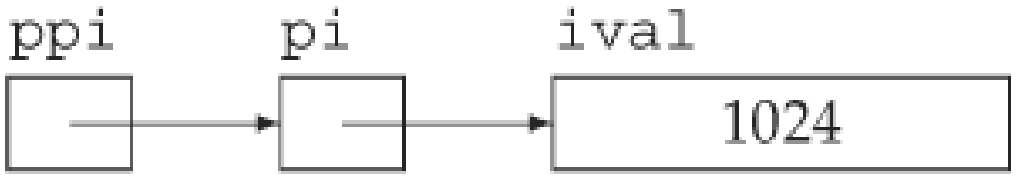
- Ukazatel můžeme pomocí přiřazení změnit – může ukazovat na jiný objekt.

```
int *pi = &ival, *pi2 = &ival2;
pi = pi2; // pi i pi2 ukazují na stejný objekt
```

Ukazatele na ukazatele

- Můžeme jednoduše vytvořit ukazatel na ukazatel:

```
int ival = 1024;
int *pi = &ival; // pi ukazuje na ival
int **ppi = &pi; // ppi ukazuje na ukazatel na ival
```



Aritmetika s ukazateli

- Pro ukazatele lze použít aritmetické operace, dovolené kombinace a typ výsledku: jsou

  T + int -> **T****

  T - int -> **T****

  T - **T*** -> int*

- Přičtení *n* k ukazateli typu *T** znamená jeho změnu o *n*-násobek délky typu *T*, podobně odečtení a rozdíl ukazatelů.
- Příklady:

```
int a[10], *p = a;
*(p + 3) = 10;/* do a[3] se uloží 10 */
/* vynulování pole a */
for (p = &a[0]; p <= &a[9]; p++) *p = 0;
/* nebo */
for (p = &a[0]; p <= &a[9]; *p++ = 0);
/* nebo */
for (p = a; p <= a + 9; *p++ = 0);
```

Ukazatele a pole

- Pokud použijeme identifikátor pole ve výrazu, automaticky dojde ke konverzi na ukazatel na první prvek pole.

```
int i_arr[] = { 0, 5, 10 };
int *p1 = i_arr;      /* p1 ukazuje na 1. prvek
                         pole i_arr, tj. i_arr[0] */
int *p2 = &i_arr[2]; /* p2 ukazuje na 3. prvek
                         pole i_arr, tj. i_arr[2] */
```

Ukazatele a pole

- Místo indexu můžeme k prvkům pole přistupovat pomocí pointerové aritmetiky.
- Můžeme vypočítat ukazatel pomocí přičítání a odčítání.

```
int i_arr[] = { 0, 5, 10 };
int *p1 = i_arr;      /* p1 ukazuje na 1. prvek
                         pole i_arr, tj. i_arr[0] */
int *p2 = &i_arr + 2;/* p2 ukazuje na 3. prvek
                         pole i_arr, tj. i_arr[2] */
```

- Odečtem dvou ukazatelů zjistíme kolik prvků daného typu je mezi nimi.

```
ptrdiff_t n = p2 – p1; // n je 2
```

Ukazatele a pole

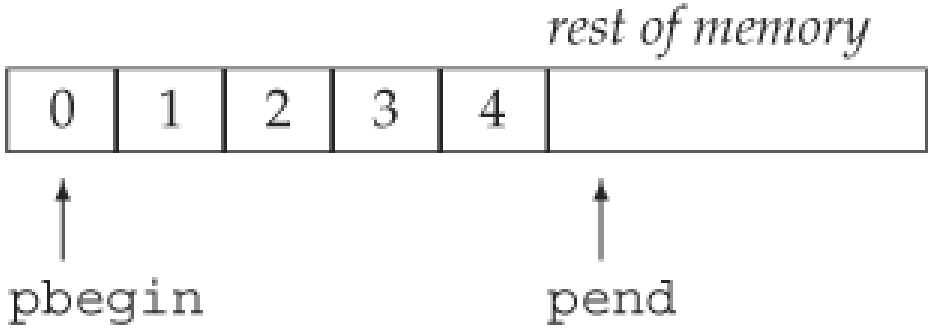
- Pokud chceme získat hodnotu prvku pole:

```
int val = *(pi + 3); // deref. ukazatele pi + 3
int val = *pi + 3; // deref. pi a k tomu + 3
```

- Závorky jsou nutné z důvodu priority!

Výpis pole

```
int a[] = { 1, 2, 3, 4, 5 };
int *pbegin = a;
int *pend = a + 5; // Mohu dereferencovat pend?
for (int *p = pbegin; p != pend; p++) {
   std::cout << *p << std::endl;
}
```



Výpis pole

- Ukazatel `pbegin` ukazuje na 1. prvek pole `a` – `a[0]`.
- Ukazatel `pend` ukazuje **za** poslední prvek pole `a`.
- `pend` slouží jako zarážka pro iteraci polem, nesmí být dereferencován!
- Ukazatele slouží pro pole podobně jako iterátory pro kontejnery v STL.

Ukazatele na konstantní objekty

- Pokud chceme deklarovat ukazatel na konstantní objekt, musí i ten být *const*.

```
const double *cptr;
*cptr = 42; // chyba
const double pi = 3.14;
double *ptr = &pi; // chyba: ptr není const
const double *cptr = &pi; /* ok: cptr je ukazatel
                             na const.*/
double dval = 3.14;
cptr = &dval; /* ok: nemůžeme ale měnit hodnotu
                       dval přes cptr */
```

Konstantní ukazatele

- Konstantní ukazatel nemůžeme po inicializaci změnit (nemůžeme změnit na co ukazuje).
- Musí být při deklaraci inicializován.
- Můžeme měnit objekt na který ukazuje.

```
int errNumb = 0;
int *const curErr = &errNumb; /* curErr je
                      konstantní ukazatel */
curErr = curErr; // chyba: curErr je const
if (*curErr) { errorHandler(); *curErr = 0; } // ok
```

Konstantní ukazatel na konstantní objekt

- Kombinace předchozích dvou případů.
- Při deklaraci musí být inicializován na konstantní objekt.
- Nesmíme měnit na jaký objekt ukazuje ani objekt na který ukazuje.

```
const double pi = 3.14159;
// pi_ptr je const a ukazuje na const
const double *const pi_ptr = &pi;
```

Ukazatele a typedef

- Použití typedef může vést k nečekaným typům.

```
using pstring = string*;
const pstring cstr;
```

- Jaký je typ proměnné cstr?
  a) `const string *`
  b) `string * const`
  c) `const string * const`

Ukazatele a typedef

```
using pstring = string*;
```

- `using` definuje typ `pstring` jako `string*`

```
const pstring cstr;
```

- `const` je modifikátor pro typ `string *`, což je ukazatel na `string`. **Správně je tedy b).**

```
string* const cstr;
```

Řetězce ve stylu C

- Jsou v C++ především z důvodu kompatibility s C.
- Jedná se o pole prvků typu char, zakončené "nulou" `'\0'`.
- Lze je inicializovat pomocí řetězce.

```
char r1[] = "Ahoj"; // řetězec ve stylu C
char r2[] = { 'A', 'h', 'o', 'j' }; /* není řetězec
        ve stylu C, chybí „nula“ na konci */
char r3[] = { 'A', 'h', 'o', 'j', '\0' };
       // řetězec ve stylu C
```

Řetězce ve stylu C

- Často se používají následujícím způsobem.

```
const char *cp = "some value";
while (*cp) {
        // dělej něco s *cp
        ++cp;
}
```

- Nikdy nezapomeňte na „nulu“ na konci pole.
  - Proč?

Řetězce ve stylu C

`strlen(s)`	Returns the length of `s`, not counting the null.
`strcmp(s1, s2)`	Compares `s1` and `s2` for equality. Returns 0 if `s1 == s2`, positive value if `s1 > s2`, negative value if `s1 < s2`.
`strcat(s1, s2)`	Appends `s2` to `s1`. Returns `s1`.
`strcpy(s1, s2)`	Copies `s2` into `s1`. Returns `s1`.
`strncat(s1, s2,n)`	Appends `n` characters from `s2` onto `s1`. Returns `s1`.
`strncpy(s1, s2, n)`	Copies `n` characters from `s2` into `s1`. Returns `s1`.

```
#include <cstring>
```

Knihovní funkce, zdroj C++ Primer, 4th Edition, Addison-Wesley

Řetězce ve stylu C

- Nikdy neporovnávejte přímo, místo toho použijte funkci strcmp (nebo ještě lépe strncmp).
- Nikdy nezapomeňte ukončovací znak, knihovní funkce jeho přítomnost nekontrolují.
  - Ani nemohou kontrolovat – potřebovaly by vědět, jak je řetězec dlouhý.
- Volající je zodpovědný za správnou velikost polí.
- Doporučujeme přečíst si sekci 4.3 v C++ Primer.

Dynamicky alokovaná paměť

- Data uložená v automaticky alokované proměnné existují pouze do konce bloku.
- Pokud chceme, aby data přežila i konec svého bloku, pak je musíme alokovat dynamicky.
- Dynamickou alokaci provádíme pomocí operátoru new.
- Operátor new alokuje paměť dle velikosti daného typu, zavolá odpovídající konstruktor a vrátí ukazatel na paměť alokovanou na haldě (heap).

```
int* ptr = new int; // int bude vytvořen s nespecifikovanou hodnotou
std::vector<int>* vecptr = new std::vector<int>(123);
                                    // vektor bude obsahovat 123 nul
```

Dynamicky alokovaná paměť

- Pokud dynamicky alokovanou paměť přestaneme potřebovat, musíme ji uvolnit (dealokovat) pomocí operátoru `delete`
  - Pokud to neuděláme, paměť zůstane alokovaná a nemůžeme ji znovu použít

```
int* ptr = new int; // int bude vytvořen s nespecifikovanou hodnotou
std::vector<int>* vecptr = new std::vector<int>(123);
                                    // vektor bude obsahovat 123 nul

// do stuff

delete ptr;
delete vecptr;
```

Dynamicky alokovaná pole

- Pokud chceme vytvořit pole s velikostí, která není známá v době překladu, musíme ho dynamicky alokovat
  - Opakovanou realokací pak můžeme dané pole i zvětšovat dle potřeby
- Pole se alokuje pomocí operátoru new[ ]
  - Operátor new[ ] nepodporuje konstruktory s argumenty
  - A uvolňuje (dealokuje) pomocí operátoru delete[]

```
int* ptr = new int[10]; // ptr je ukazatel na první z 10 intů
                        // všechny prvky pole jsou neinicializované

// do stuff

delete[] ptr;
```

Dynamicky alokovaná paměť – `malloc`

- Paměť se dá též alokovat pomocí C funkce `malloc`.
  - Následně se dealokuje funkcí `free`
- `malloc` nezná typy a konstruktory, jednoduše alokuje n bytů paměti.
  - Tudíž se nedá použít pro třídy.

```
int* ptr = static_cast<int*>(  // malloc vrací void*, ne int*
       malloc(2 * sizeof(int)) // 2* velikost intu v bytech
     ); // ptr bude ukazovat na kus paměti, kam se vejdou 2 inty

// do stuff

free(ptr);
```

Příklad: dynamicky alokovaná pole

```
#include <iostream>
int main() {
    std::cout << "Jak dlouhy retezec se chystas napsat? ";
    int delka; std::cin >> delka;
    char* retezec = new char[delka + 1]; // proč + 1?
    std::cout << "Tak ho napis: ";
    std::cin >> retezec;
    std::cout << "Doufam, ze jsi ho nenapsal moc dlouhy.\n";
    std::cout << "Napsal jsi: " << retezec << "\n";
    delete[] retezec; // tuto paměť už nepotřebujeme
}
```

Vícerozměrná pole

- V C++ neexistuje vícerozměrné pole jako datový typ, místo toho používáme pole polí.

```
int ia[3][4]; /* ia je pole 3 prvků, každý prvek
                 obsahuje pole 4 prvků typu int */
```

- Inicializace pole:

```
int ia[3][4] = { { 0, 1, 2, 3 }, { 4, 5, 6, 7 },
               { 8, 9, 10, 11 } };
int ia[3][4] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                 11 };
```

Ukazatelé a vícerozměrná pole

- Vícerozměrná pole:

```
int ia[3][4]; /* pole velikosti 3, každý prvek je
                 pole int velikosti 4 */
int(*ip)[4] = ia; // ip je ukazatel na pole 4 intů
ip = &ia[2]; // ia[2] je pole 4 intů
```

- Pozor na závorky!

```
int *ip[4]; // pole 4 ukazatelů na int
int(*ip)[4]; // ukazatel na pole 4 intů
```

- Doporučujeme číst deklaraci zevnitř ven: **`ip`** je ukazatel na **`int[4]`**

Typedef a vícerozměrná pole

- Typedef nám zjednoduší práci s vícerozměrnými poli.

```
int ia[3][4] = { {11,12,13,14}, {21,22,23,24}, {31,32,33,34} };

using int_array = int[4]; // nebo: typedef int int_array[4];

for (int_array *p = ia; p != ia + 3; ++p) {
    for (int *q = *p; q != *p + 4; ++q) {
        std::cout << *q << " ";
    }
    std::cout << "\n";
}
```

```
11 12 13 14
21 22 23 24
31 32 33 34
```

The End