X36PJC
6. přednáška

Pole, ukazatele

Minulá přednáška

- Namespace
- Datové typy z STL
  - String
  - Vector
  - Iterace (Iterátory)

Pole

- Složený datový typ
- Umožňuje ukládat prvky stejného typu.
- Nemůžeme libovolně měnit jeho velikost.
- Velikost pole se specifikuje při jeho deklaraci.
- Pole si nepamatuje svoji velikost.
- Prvky v poli jsou indexovány od nuly.

Pole

- Přístup k jednotlivým prvkům pomocí indexu.
- Alokace pole:
  - statická, pokud je velikost známa v době překladu,
  - dynamická.

```
int  stat_pole [5]; //statická alokace
int  *dyn_pole = new int[5](); //dynamická alokace
```

Statická Pole

- Nové pole je definováno
  - identifikátorem (název),
  - typem uložených prvků a
  - velikostí.
- Typ prvku pole – libovolný, kromě:
  - referencí,
  - funkcí (ale mohou být ukazatele na funkci).
- Prvkem pole může být opět pole (vícerozměrná pole).

Statická Pole - Velikost

- Velikost musí být známa v době překladu
- Může být dána konstantím výrazem, konstantou, hodnotou z výčtu (enumerator)

Statická Pole - Příklad

const unsigned buf_size = 512, max_files = 20;

int staff_size = 27; // nekonstantní

const unsigned sz = get_size(); // konstantní hodnota, není známa v době překladu

char input_buffer[buf_size]; // **ok:** Proč?

fileTable[max_files + 1]; // **ok**: Proč?

double salaries[staff_size]; // **chyba**: Proč?

int test_scores[get_size()]; // **chyba**: Proč?

int vals[sz]; // **chyba**: Proč?

Statická Pole - Inicializace

- Pole můžeme inicializovat explicitně takto

  const unsigned int velikost = 3;

  int pole[velikost] = {1, 2, 3};

- Při explicitní inicializaci nemusíme uvádět velikost pole

  int pole[] = {1, 2, 3};

Statická Pole - Inicializace

- Pokud pole neinicializujeme pak záleží kde je pole alokováno.
- Prvky pole alokovaného **mimo tělo funkce** jsou inicializovány na “0” (platí pro vestavěné typy).
- Prvky pole alokovaného **uvnitř funkce** nejsou inicializovány (platí pro vestavěné typy).
- Pokud pole uchovává prvky typu třída pak
  - prvky pole jsou inicializovány implicitním konstruktorem, pokud třída nemá implicitní konstruktor pak musí být prvky inicializovány explicitně.

Statická Pole – Pole znaků

- Pole typu char mají speciální inicializátor – řetězec

  char ca1[] = {'C', '+', '+'}; // velikost pole 3

  char ca2[] = {'C', '+', '+', '\0'}; // velikost pole 4

  char ca3[] = "C++"; // velikost pole 4

- Pole ca2 a ca3 jsou stejná, pole ca1 nelze vytvořit pomocí řetězce

Ukazatele

- Ukazatel je abstrakce adresy ve vyšším programovacím jazyce.
- Proměnná 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:
  - typ. 4B pro 32-bit prostředí,
  - typ. 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 (&) – získání adresy paměťového místa (pořízení ukazatele)

  int a = 10;

  int *p_a = &a; // p_a ukazuje na a

  int x = *p_a; // proměnná x je inicializována hodnotou p_a

Ukazatele - Inicializace

int ival = 1024;

int *pi = 0; // pi je incilizováno na 0 – null 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 = 0; // pi2 je NULL

Ukazatele - Incializace

string *ps, str; //ps je ukazatel na string, str je string

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 (NULL, 0) – pak lze kontrolovat.
- Statisticky je neinicializovaný ukazatel jedna z nejčastějších chyb – překladač ji neodhalí.

Ukazatel - Hodnoty

- Ukazatel může být inicializován na:
  - konstantní výraz s hondotou „0“ – NULL,
  - na adresu objektu stejného typu jako je ukazatel,
  - na adresu o jeden prvek za polem (pro potřeby iterace – nesmí být dereferencován)
  - na jiný platný pointer stejného typu

Ukazatele – void *

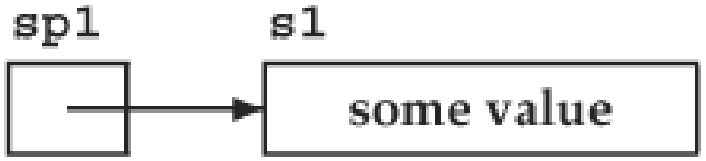
- Může ukazovat na objekt jakéhokoli typu

  double pi = 3.14.1592;

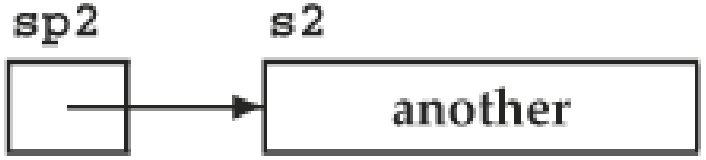
  void *p_pi = π
- Abychom mohli přistupovat k objektu na který ukazuje, musíme ho nejdříve přetypovat (viz další část přednášky)

Dereference

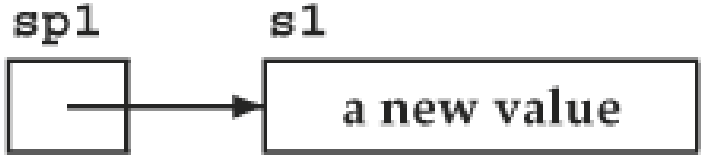
```
string s1("some value");
string *sp1 = &s1;
```



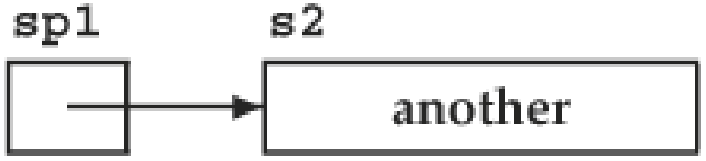

```
string s2("another");
string *sp2 = &s2;
```




```
// 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

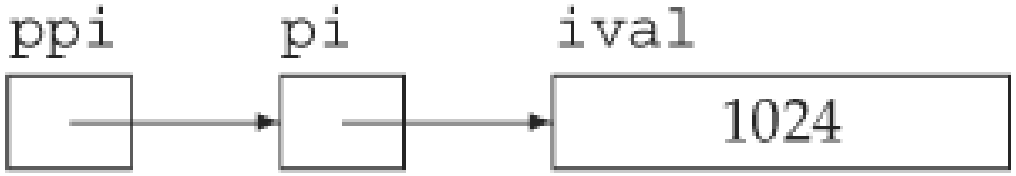
Ukazatele na ukazatele

- Můžeme jednoduše vytvořit ukazatel na ukazatel

  int ival = 1024;

  int *pi = &ival; // pi ukazuje na ival

  int **ppi = π // ppi ukazuje na ukazatel na ival



Ukazatele a Pole

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

  int int_arr[] = {0,5,10};

  int *int_p = int_arr; // int_p ukazuje na int_arr[0]

  int_p = &int_arr[2]; // int_p ukazuje na 3. prvkek v poli int_arr

Ukazatele a Pole

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

  int i_arr[] = {1,2,3,4,5};

  int *ip = i_arr; // ip ukazuje na i_arr[0]

  int *ip2 = ip + 3; // ip2 ukazuje na i_arr[3]
- Odečtem dvou ukazatelů zjistíme kolik prvů je mezi nimi.

  ptrdiff_t n = ip2 – ip;

Ukazatele a Pole

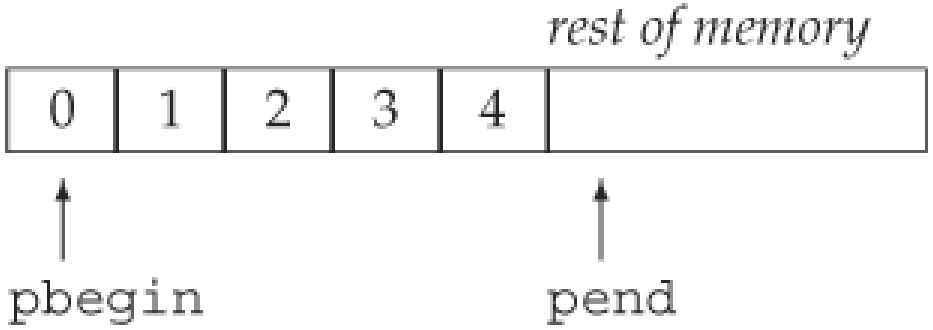
- Pokud chceme získat hodnotu prvku pole.

  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?
for(int *p=pbegin; p!=pend; p++) {
    cout << *p << endl;
}
```



Výpis Pole

- Ukazatel pbegin ukazuje na 1. prvek pole a – a[0].
- Ukazatel pend ukazuje za poslední prvek pole a.
- 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 = π // chyba: ptr není const

  const double *cptr = π // 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 být obtížně čitelné.

  typedef string *pstring;

  const pstring cstr;

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

Ukazatele a typedef

typedef string *pstring;

- typedef 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

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 „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.
- Nikdy nezapomeňte ukončovací znak, knihovní funkce jeho přítomnost nekontrolují ani nemohou.
- Volající je zodpovědný za správnou velikost polí.
- Doporučuji přečíst si sekci 4.3 v C++ Primer

Dynamicky alokovaná pole

- Staticky alokované pole má několik nevýhod:
  1. velikost pole musí být známa při překladu
  2. pole existuje pouze do konce bloku, kde bylo alokováno (často nežádoucí)
  3. jeho velikost je po vytvoření neměnná
- Dynamicky alokované pole (DAP) odstraňuje nevýhody 1 a 2.
- DAP musí být explicitně uvolněno programátorem.

DAP

- DAP alokujeme pomocí operátoru new.

  // pole 10 neinicializovaných prvků typu int

  int *pia = new int[10];
- Operátor new vrací ukazatel na první prvek nově vytvořeného pole.
- Pole je vytvořeno na haldě (heap).
- Pokud je typem prvků třída, pak je zavolán pro každý z prvků implicitní konstruktor.

DAP

- Pokud je typ prvků primitivní typ, pak jsou neinicializovány.
- O inicializaci prvků na implicitní hodnotu můžeme požádat překladač takto:

  int *pi = new int[10]();
- Pokud pole již nepotřebujeme uvolníme ho:

  delete [] pi;

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};

Ukazatele a Vícerozměrná Pole

int ia[3][4]; // pole velikosti 3, každý prvek je pole int velikosti 4

int (*ip)[4] = ia; // ip ukazuje na pole 4 intů

ip = &ia[2]; // ia[2] je pole 4 intů

- Pozor na závorky!

  int *ip[4]; // pole ukazatelů na int

  int (*ip)[4]; // ukazatel na pole 4 intů
- Doporučuji číst definici zevnitř ven: ip je ukazatel na int[4]

Typedef a Vícerozměrná pole

- Typedef nám zjednoduší práci s vícerozměrnými poli.

```
typedef int int_array[4];

int_array *ip = ia;
for (int_array *p = ia; p != ia + 3; ++p)
    for (int *q = *p; q != *p + 4; ++q)
        cout << *q << endl;
```

Děkuji Vám za pozornost