Ukazatele a pole

204 Deklarace jednorozměrného pole s inicializací

Chceme-li pole v deklaraci inicializovat, zapíšeme seznam inicializátorů jednotlivých prvků do složených závorek:

```
#define N 5
int A[N] = {1, 2, 3, 4, 5};
```

Uvedeme-li méně inicializátorů, než kolik má pole prvků (v našem případě méně než `N`), doplní si je překladač nulami. To znamená, že zápis

```
int A[N] = {1, 2, 3, };
```

znamená totéž co

```
int A[N] = {1, 2, 3, 0, 0};
```

a to – vzdor tvrzení některých učebnic – i pro lokální automatická pole.

Uvedeme-li více inicializátorů, než kolik má pole prvků, ohlásí překladač chybu.

205 Jaké prvky má pole?

Deklarujeme-li pole `A` zápisem

```
int A[N] = {1, 2, 3, 0, 0};
```

bude mít toto pole prvky `A[0]`, `A[1]`, …, `A[N-1]`. Prvek s indexem `N` už není jeho součástí. Pole v C a v C++ jsou vždy indexována od nuly.

206 Inicializace pole nulami

Chceme-li vyplnit celé pole nulami, použijeme prázdný inicializátor:

```
int A[N] = {};
```

znamená totéž co

```
int A[N] = {0, 0, 0, 0, 0};
```

207 Čárka na konci seznamu inicializátorů

Na konci seznamu inicializátorů může být zbytečná čárka. Zápis

```
int A[N] = {1, 2, 3, 4, 5, };
```

není chybný.

Toto pravidlo usnadňuje generování seznamu inicializátorů pomocí maker nebo jiných programů.

208 Známe inicializátory, neznáme počet prvků

Generujeme-li deklaraci pole pomocí maker nebo generujeme-li části svého programu pomocí generátoru zdrojového kódu, může se stát, že nebudeme vědět, kolik prvků pole bude mít, známe však jeho inicializátory. Pak můžeme konstantu udávající počet prvků v deklaraci vynechat. Například takto:

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

Překladač si spočítá počet inicializátorů a z něj odvodí počet prvků.

209 Kolik prvků má pole?

Jestliže v deklaraci jednorozměrného pole neuvedeme počet prvků, můžeme ho později zjistit pomocí operátoru `sizeof`:

```
int A[] = {1, 2, 3, 4, 5, };
int pocet = sizeof(A)/sizeof(int);
```

V místě, kde překladač vidí definici pole `A`, vrátí operátor `sizeof` počet bajtů, které toto pole zabírá. Prvky pole jsou uloženy vždy v souvislé řadě za sebou, a proto když počet bajtů pro celé pole vydělíme velikostí jednoho prvku, dostaneme počet prvků.

210 Inicializujeme jen některé prvky (jen C99)

Chceme-li inicializovat jen několik počátečních prvků, použijeme postup popsaný v tipu 204.

Jazyk C podle standardu z roku 1999 dovoluje inicializovat jen vybrané prvky pole, které nemusí tvořit souvislou řadu počínaje prvním prvkem. K tomu použijeme tzv. pojmenované inicializátory, které mají v případě pole tvar

`[`*index*`] =` *hodnota*

To znamená, že chceme-li inicializovat v poli `A` jen prvky s indexy 1 a 3, napíšeme

```
int A[] = {[1] = 6, [3] = 77 };
```

Poznamenejme, že pojmenované inicializátory lze kombinovat s nepojmenovanými (takovými, jaké používáme v ostatních tipech).

211 Pole s nekonstantním počtem prvků

Potřebujeme-li pole, v němž počet prvků není zadán konstantním výrazem, ale je výsledkem výpočtu, musíme v C90 a v C++ použít dynamicky alokované pole:

```
void f(int n)
{
   double *B = new double[n];  // V C++
   // double *B = (double*)malloc(n*sizeof(double)); // V C
   // a další zpracování
}
```

Proměnná `B` je ukazatel na `double`; ukazatele lze, jak jistě víte, indexovat stejně jako pole.

Takto vytvořené pole nelze ovšem inicializovat způsobem popsaným v tipu 204. Nelze také zjistit jeho pojmenovaný inicializátor (viz tip 210).

212 Pole objektového typu

Podívejme se na deklaraci

```
#define N 50
T pole[N];
```

kde `T` je objektový typ. I když neobsahuje explicitní inicializaci, objekty nemohou zůstat neinicializované, a proto se pro každý z prvků zavolá konstruktor bez parametrů. *To znamená, že takovouto deklaraci můžeme v programu napsat pouze v případě, že typ* `T` *má bezparametrický konstruktor, který je v místě deklarace přístupný.*

213 Pole objektového typu inicializované skalárními hodnotami

Tentokrát se podíváme na deklaraci

```
#define N 50
T pole[N] = {1, 2, 3};
```

kde `T` je objektový typ. Tato inicializace je správná, jestliže má typ `T` konstruktor s jedním parametrem typu `int` nebo typu, který lze implicitními konverzemi na `int` převést. První tři prvky budou inicializovány tímto konstruktorem a jako parametr mu budou předány hodnoty uvedené v seznamu inicializátorů. Zbylé prvky budou inicializovány bezparametrickým konstruktorem. Všechny použité konstruktory musí být v místě deklarace přístupné.

214 Pole objektového typu inicializované instancemi

Chceme-li inicializovat pole objektového typu `T` již hotovými instancemi `a`, `b` (a případně dalšími), můžeme napsat

```
#define N 50
T pole[N] = {a, b, c};
```

Tyto instance se do odpovídajících prvků pole okopírují pomocí kopírovacího konstruktoru typu `T`. Kopírovací konstruktor musí být v místě deklarace přístupný.

215 Explicitní volání konstruktoru v inicializaci pole objektového typu


pokročilý

Jestliže nechceme kvůli inicializaci pole objektového typu `T` vytvářet samostatné instance, můžeme jako inicializátory použít přímá volání konstruktoru:

```
#define N 50
T pole[N] = {T(1,7), T(), T(4)};
```

Pro první prvek se použije konstruktor se dvěma parametry, pro druhý bezparametrický konstruktor a pro třetí konstruktor s jedním parametrem. Zbylé prvky budou inicializovány konstruktorem bez parametrů. Všechny použité konstruktory musí být v místě deklarace přístupné.

216 Pole s nekonstantním počtem prvků (jen C99)


pokročilý

Jazyk C podle standardu z roku 1999 dovoluje zadat v deklaraci lokálního automatického pole počet prvků nekonstantním výrazem. Například takto:

```
void f(int n)
{
   double B[n];                        // Jen C99
  // a další zpracování
}
```

217 Inicializace vícerozměrného pole


začátečník

I vícerozměrná pole lze v deklaraci inicializovat. Přitom vyjdeme ze skutečnosti, že jazyky C a C++ považují vícerozměrné pole za jednorozměrné pole, jehož prvky jsou opět pole. (Např. dvourozměrné pole – matice – je pole, jehož prvky jsou jednorozměrná pole – řádky matice.) To znamená, že do složených závorek zapíšeme inicializátory jednotlivých prvků, což jsou opět pole – tentokrát jednorozměrná. Například takto

```
int B[2][3] = {{1,2,3},{4,5,6}};
```

`B` je pole složené se dvou prvků – polí o třech prvcích typu `int`.

Vynecháme-li některé inicializátory, doplní si je překladač nulami.

218 První index v deklaraci vícerozměrného pole můžeme někdy vynechat


pokročilý

Jestliže vícerozměrné pole v deklaraci inicializujeme, můžeme vynechat první index a překladač si potřebnou konstantu dopočítá podle počtu inicializátorů. Například deklarace

```
int B[][3] = {{1,2,3},{4,5,6}};
```

zavádí dvourozměrné pole, které má dva řádky a tři sloupce, tedy stejné pole jako deklarace v tipu 213.

219 Literál typu pole (jen C99)

V C++ a v C90 nelze použít literál – přímo zapsanou konstantu – typu pole. Jazyk C99 to umožňuje. Literál typu pole má v C99 tvar

`(`*typ_pole*`){`*seznam_inicializátorů*`}`

To znamená, že můžeme napsat např.

```
int *p = (int[]){ 4, 5, 6 }   // jen C99
```

220 Přístup k prvkům pole

Chceme-li pracovat s jedním prvkem jednorozměrného pole, použijeme jeho index – tedy jeho pořadové číslo počítané od 0. Je-li `A` pole o `n` prvcích deklarované zápisem

```
int A[n];
```

přiřadíme prvku s indexem `j` hodnotu příkazem

```
A[j] = 7;
```

Máme-li vícerozměrné pole, musí mít každý index své vlastní lomené závorky. *Při přístupu k prvkům se nekontroluje, zda index leží v rozmezí daném deklarací pole.*

221 Indexování je symetrické

Operátor indexování je komutativní – můžeme zaměnit pořadí operandů. To znamená, že místo `A[i]` můžeme napsat `i[A]`. Oba zápisy znamenají totéž.

To platí i pro vícerozměrná pole Například místo `A[1][2]` můžeme napsat `2[1[A]]`.

Podobné zápisy někdy používají programátoři zvyklí na asembler.

222 Pole se skoro vždy konvertuje na ukazatel

V jazycích C a C++ platí pravidlo, že pole se ve výrazech konvertuje na ukazatel na první prvek, a to vždy s výjimkou následujících situací:

1. Použijeme-li na ně operátor `sizeof`; aplikujeme-li operátor `sizeof` na pole, dostaneme velikost pole, nikoli velikost ukazatele.
2. Použijeme-li na ně operátor `&`; aplikujeme-li operátor získání adresy na pole, dostaneme ukazatel na pole, nikoli ukazatel na ukazatel. (Rozdíl je v typu výsledku, nikoli v číselné hodnotě.)
3. Použijeme-li pole k inicializaci reference.

Poslední možnost se týká pouze C++.

223 Pole lze přiřadit ukazateli

Je-li `p` ukazatel na typ `T`, lze mu přiřadit hodnotu představující pole typu `T`. Například

```
int A [] = {0, 1, 2, 3, 4};
int *p = A;
```

Proměnná `p` bude obsahovat adresu prvního prvku pole `A`, tedy `&A[0]`.

224 Používáme proměnnou, na kterou ukazuje ukazatel

Chceme-li použít proměnnou, na kterou ukazuje daný ukazatel, musíme ho dereferencovat pomocí operátoru `*`. Dereferencovaný ukazatel může stát i na levé straně přiřazení (operátor `*` vytváří l-hodnotu).

Jestliže např. platí deklarace

```
int A[] = {1, 2, 3, 6, 8};
int *p = A;
```

znamená přiřazení

```
*p = 54;
```

totéž co

```
A[0] = 54;
```

neboť `p` ukazuje na první složku pole `A`.

225 Ukazatel indexujeme jako pole

Jazyky C a C++ předpokládají, že každý ukazatel na typ `T` (jiný než `void`) je vlastně ukazatel na první prvek jednorozměrného pole typu `T`. To znamená, že každý ukazatel můžeme indexovat.

Je-li `p` ukazatel na `int`,

```
int A [] = {0, 1, 2, 3, 4};
int *p = A;
```

můžeme napsat

```
p[2] = 654;
```

a tento příkaz bude znamenat totéž co

```
A[2] = 654;
```

226 Jak vytvořit ukazatel nikam

Přiřadíme-li ukazateli hodnotu 0, říkáme tím, že tento ukazatel neukazuje na žádnou platnou adresu.

V jazyce C používáme místo hodnoty 0 zpravidla konstantu (makro) `NULL` definované v `<stddef.h>`. Standard však přesně neříká, jakého typu `NULL` je (může to být mj. nula typu `int`, `long` nebo `void*`), a proto dáváme v C++ přednost literálu 0.

227 Adresová aritmetika (aritmetika ukazatelů)

Jazyky C a C++ umožňují provádět s ukazateli některé běžné aritmetické operace: přičíst k ukazateli celé číslo, odečíst dva ukazatele, použít na ně operátory ++ a - -.

Všechny tyto operace mají smysl, pouze když jde o ukazatele na prvky pole.

Dále můžeme ukazatele porovnávat pomocí relačních operátorů == a !=. Tyto operace mají smysl pro jakékoli ukazatele.

Ukazatele na prvky polí a struktur z jazyka C (nikoli objektových typů) lze porovnávat také pomocí relačních operátorů <, <=, > a >=.

228 Co znamená rovnost nebo nerovnost ukazatelů?

Dva ukazatele jsou si rovny, jestliže ukazují na totéž místo v paměti. Jde-li o ukazatele na proměnné, znamená to, že oba ukazatele ukazují na tutéž proměnnou.

Pozor, v případě instancí objektových typů neplatí opačné tvrzení: Adresa jednoho objektu může být vyjádřena dvěma (i více) různými ukazateli. Jinak řečeno, *když si dva ukazatele na instance nejsou rovny, neplyne z toho ještě, že ukazují na různé objekty*.

229 Který prvek má vyšší index?

Máme-li dva ukazatele `p` a `q` na *prvky téhož pole*, plyne ze vztahu

```
p > q
```

že prvek, na který ukazuje `p`, má vyšší index než prvek, na který ukazuje `q`.

230 Která složka struktury je definována dříve?

Máme-li dva ukazatele `p` a `q` na *složky téže struktury*, plyne ze vztahu

```
p > q
```

že složka, na kterou ukazuje `p`, je v deklaraci této struktury zapsána dříve než složka, na kterou ukazuje `q`.

231 Chceme s ukazatelem přejít na následující prvek pole

Máme ukazatel `q` na prvek pole typu `T`. Chceme-li získat ukazatel na následující prvek téhož pole, použijeme operaci ++:

```
++q;
```

Ke stejnému výsledku by vedl i příkaz

```
q +=1;
```

Tato operace způsobí přechod na následující prvek, nikoli zvětšení adresy uložené v proměnné `q` o 1. (Adresa se zvětší o `sizeof(T)`). Musíme si ovšem být jisti, že pole následující prvek obsahuje.

232 Chceme s ukazatelem přejít na předcházející prvek pole

Máme ukazatel `q` na prvek pole typu `T`. Chceme-li získat ukazatel na předcházející prvek téhož pole, použijeme operaci - -:

```
--q;
```

Ke stejnému výsledku by vedl i příkaz

```
q -=1;
```

Tato operace způsobí přechod na předcházející prvek, nikoli zmenšení adresy uložené v proměnné `q` o 1. (Adresa se zmenší o `sizeof(T)`). Musíme si ovšem být jisti, že pole předcházející prvek obsahuje.

233 Chceme s ukazatelem přejít o m prvků dál

Máme ukazatel `q` na prvek pole a chceme získat ukazatel na prvek, který leží o `m` prvků dál (v kterémkoli směru). Pak stačí přičíst číslo `m` k ukazateli `q`:

```
q = q + m;
```

I tentokrát si však musíme být jisti, že pole odpovídající prvek obsahuje. Poznamenejme, že tentokrát se adresa změní o `m*sizeof(T)`.

234 O kolik se liší indexy daných prvků?

Máme ukazatele `p` a `q`, které ukazují na prvky téhož pole typu `T`. Chceme vědět, o kolik se liší jejich indexy. K tomu nám stačí odečíst tyto dva ukazatele:

```
int rozdil_indexu = q - p;
```

Rozdíl ukazatelů vrátí jako výsledek rozdíl adres vydělený velikostí prvku.

235 Překopírování obsahu jednorozměrného pole

Pole je v jazycích C a C++ nízkoúrovňová konstrukce, pro niž není definováno přiřazení, které by umožňovalo přenést obsah pole jako celku jediným příkazem. Chceme-li tedy přenést všechny hodnoty z pole `A` do odpovídajících prvků pole `B`, můžeme použít cyklus

```
for(int i = 0; i < N; i++)
   B[i] = A[i];
```

236 Překopírování obsahu jednorozměrného pole pomocí adresové aritmetiky

Chceme-li okopírovat obsah pole do jiného pole, můžeme použít i adresovou aritmetiku. Jsou-li `A` a `B` pole, deklarovaná příkazy

```
#define N 25

int A[N], B[N];
```

můžeme obsah pole `A` překopírovat do pole `B` cyklem

```
for(int* p = A, *q = B; p < &A[N]; ++p, ++q)
   *q = *p;
```

Výraz `&A[N]` není chybný, i když prvek `A[N]` v poli `A` neexistuje. Standard jazyka C výslovně dovoluje získat jeho adresu za účelem porovnání (nesmíme ji ovšem dereferencovat).

237 Kopírování obsahu pole, když se zdrojové a cílové pole nepřekrývá

Chceme-li překopírovat obsah pole `zdroj` do pole `cil` *a víme-li, že se tato dvě pole v paměti počítače nepřekrývají*, můžeme použít knihovní funkci `memcpy()` deklarovanou v hlavičkovém souboru `<string.h>`. Tato funkce očekává po řadě ukazatel na cílové pole, ukazatel za zdrojové pole a počet kopírovaných bajtů.

Platí-li deklarace

```
#define N 10
int zdroj[N] = {1,2,3,};
int cil[N];
```

můžeme přenést obsah pole zdroj do pole cil příkazem

```
memcpy(cil, zdroj, N*sizeof(int));
```

238 Kopírování obsahu pole, když se zdrojové a cílové pole překrývá

Chceme-li překopírovat obsah pole `zdroj` do pole `cil` *a víme-li, že se tato dvě pole v paměti počítače mohou překrývat*, použijeme knihovní funkci `memmove()` deklarovanou v hlavičkovém souboru `<string.h>`. Tato funkce očekává po řadě ukazatel na cílové pole, ukazatel za zdrojové pole a počet kopírovaných bajtů.

Máme pole `A` typu `int` deklarované příkazem

```
#define N 10
int zdroj[N] = {1,2,3,};
```

a chceme prvky s indexy 1 – 4 „posunout“ o jeden prvek doprava. K tomu můžeme použít příkaz

```
memmove(zdroj+1, zdroj, 4*sizeof(int));
```

Pole `A` pak bude obsahovat hodnoty 1,1,2,3,0,…

239 První nulový prvek v jednorozměrném poli pomocí adresové aritmetiky

Máme jednorozměrné pole `D` o délce `N` obsahující čísla typu `double`, víme, že toto pole obsahuje alespoň jednu nulovou hodnotu a chceme najít první z nich. K tomu můžeme použít příkazy

```
double *p = D;
while(*p) p++;
// zpracování p
```

240 První nulový prvek v jednorozměrném poli pomocí indexování

Podívejme se znovu na úkol najít první nulový prvek v jednorozměrném poli, který jsme řešili v tipu 239. Samozřejmě můžeme týž problém řešit i „klasicky“, tedy pomocí indexů:

```
int i;       // Tuto proměnnou potřebujeme i po skončení cyklu
for(i = 0; i < N; i++)
   if(!D[i]) break;
// zpracování i
```

Druhá možnost je bezpečnější: Kdyby se stalo, že pole `D` neobsahuje žádný záporný prvek, měli bychom jistotu, že cyklus skončí a my to podle hodnoty `i` můžeme zjistit.

241 První záporný prvek ve dvourozměrném poli pomocí indexování

Máme dvourozměrné pole (matici) `M` o `m` řádcích a `n` sloupcích. Toto pole chceme prohledat po řádcích a najít indexy jeho prvního záporného prvku. K tomu nám poslouží příkazy

```
int i, j; // Tyto proměnné potřebujeme i po skončení cyklu
for(i = 0; i < m; i++)
   for(j = 0; j < n; j++)
   {
      if(M[i][j] < 0) goto ven;
   }
ven: // zpracování i a j
```

Poznamenejme, že jde o jedno z mála tolerovaných použití příkazu `goto`.

242 Výpis všech záporných prvků ve dvourozměrném poli pomocí adresové aritmetiky

Máme dvourozměrné pole (matici) `M` o `m` řádcích a `n` sloupcích

```
int M[m][n];
```

Toto pole chceme prohledat po řádcích a vypsat hodnoty všech jeho záporných prvků.

K tomu můžeme použít příkazy

```
for(int(*p)[n] = M; p < &M[m]; p++)
   for(int *q = *p; q < *p+n; q++)
      if(*q<0)
      {
         cout << *q << endl;
      }
```

`M` je dvourozměrné pole, tedy pole složené z jednorozměrných polí o délce `n`. To znamená, že se automaticky konvertuje na ukazatel na první prvek – tedy na jednorozměrné pole o délce `n` prvků. Proto jako parametr vnějšího cyklu použijeme proměnnou `p` typu `int(*)[n]`.

Přičteme-li k této proměnné 1, bude to znamenat přesun na další prvek pole `M`, tedy na další řádek matice `M`.

Dereferencujeme-li `p`, dostaneme jednorozměrné pole – tedy řádek matice `M`. Toto pole se ovšem automaticky konvertuje na ukazatel na první prvek, tedy na první celé číslo v daném řádku.

243 Omezené (restringované) ukazatele v C99

V deklaraci ukazatele v C99 můžeme použít klíčové slovo `restrict`:

```
int * restrict a;
```

Tím v podstatě říkáme, že na objekt, na který `a` ukazuje, sice mohou ukazovat i jiné ukazatele, ale měnit ho budeme pouze pomocí `a`.

Nevznikne tedy tzv. *aliasing*, kdy hodnota proměnné může být zároveň měněna z několika míst v programu pomocí několika ukazatelů. To umožňuje překladači agresivnější optimalizaci.

Poznamenejme, že klíčové slovo má z hlediska syntaxe stejné postavení jako modifikátory `const` nebo `volatile`.