Typ ukazatel, strukturované datové typy

Jitka Kreslíková, Aleš Smrčka

2021

Fakulta informačních technologií
Vysoké učení technické v Brně

IZP – Základy programování

Typ ukazatel, strukturované datové typy

- Datový typ ukazatel
- Alokace paměti
- Datový typ pole, řetězce
- Funkce - předávání parametrů

Datový typ ukazatel

- Deklarace typu ukazatel
- Reference, dereference ukazatele
- Nulový ukazatel
- Inicializace ukazatele
- Obecný ukazatel

Datový typ ukazatel

- Obecný formát definice proměnné typu ukazatel:

  *bázový_typ *jméno_proměnné;*

```
int *integer_ptr;
// integer_ptr je ukazatel na int
```

- Do proměnné se ukládá adresa paměti
- Bázový typ
  - určuje velikost odkazované paměti
  - libovolný datový typ, funkce, void

Datový typ ukazatel - operátory

- & referenční
  - vrací adresu proměnné operandu.

```
int x = 10;
int *px; // (int *) je datový typ
px = &x; // reference; px,&x - adresa
```

- * dereferenční
  - vrací hodnotu uloženou na adrese operandu (hodnota získaná odkazem)
  - *px - místo v paměti o rozměru sizeof(int)

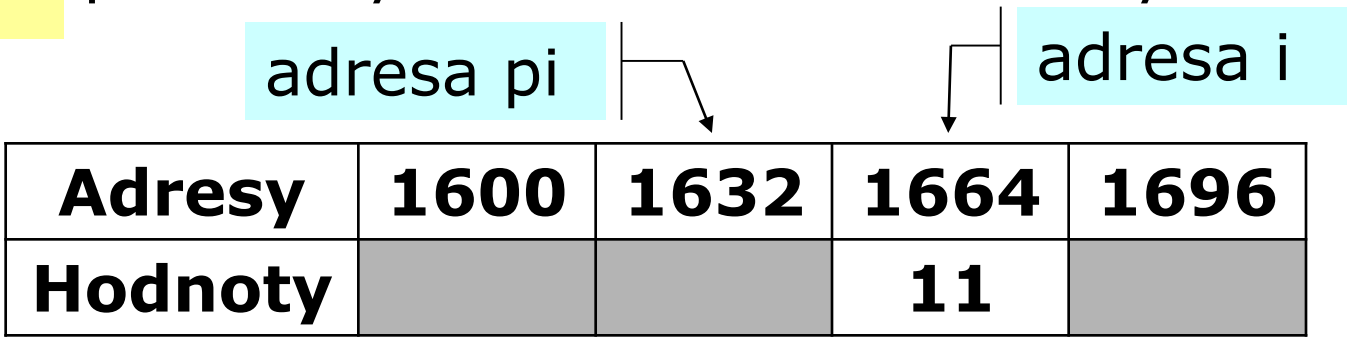
```
int y = *px; // dereference = odkaz
```

Reference, dereference ukazatele

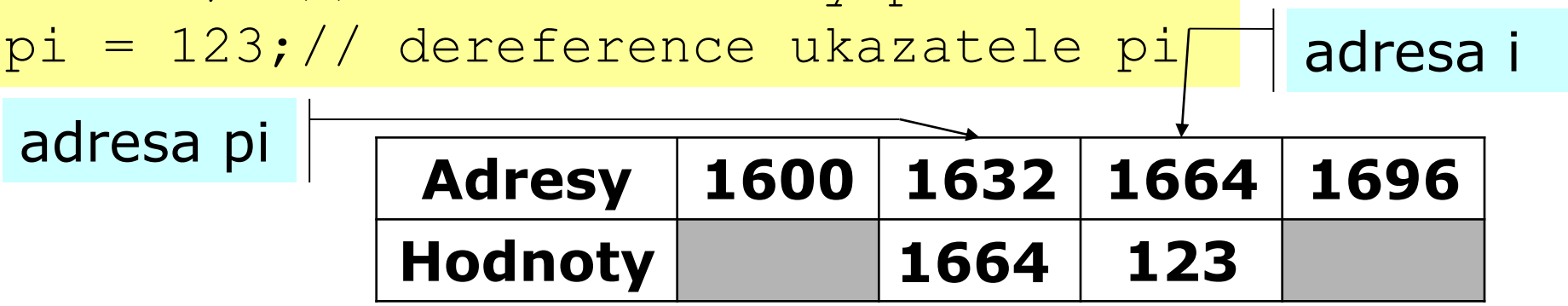
Příklad: Zápis hodnoty 123 do proměnné *i* přes ukazatel.

```
int i = 11;
int *pi;
```

Situace po inicializaci *i*, obsah okolních paměťových míst není definovaný:



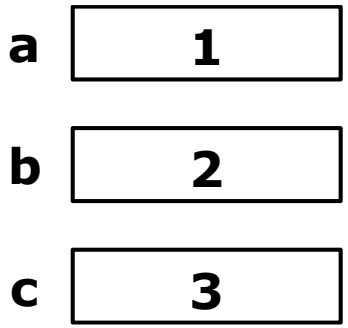

```
pi = &i;  // získání adresy proměnné i
*pi = 123;// dereference ukazatele pi
```

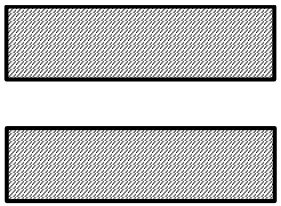


Reference, dereference ukazatele

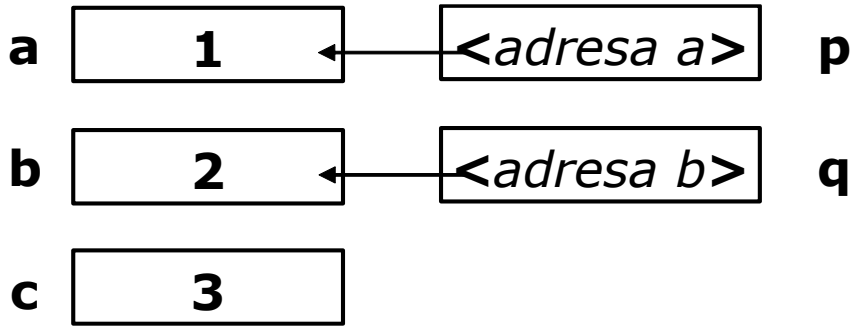
Příklad: dva ukazatele mohou ukazovat na stejné místo v paměti.

```
int a = 1;
int b = 2;
int c = 3;
int *p;
int *q;
```

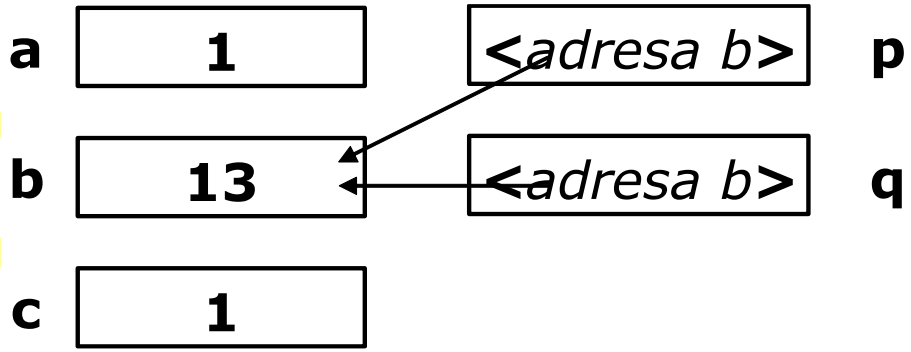





```
p = &a; // reference a
q = &b; // reference b
```




```
c = *p; // dereference
p = q;  // přiřadí ukazatel
*p = 13;// přepíše obsah b
```



Nulový ukazatel

- Konstanta NULL ze <stdio.h>
  - lze přiřadit každému ukazateli
  - pro test, že ukazatel nikam neukazuje
- Zásady bezpečného programování
  - ukazatel musí obsahovat bezpečnou adresu
  - nepoužívané ukazatele nastavit na NULL
  - testovat ukazatele zda nejsou NULL

```
// definice ukazatele a inicializace na NULL
int *pi = NULL;
    .
    .
// test, zda ukazatel někam ukazuje
if (pi != NULL)
  *pi = 20;
```

Inicializace ukazatele

Příklad: definice ukazatele s inicializací.

```
int i = 7;
int *pi = &i;
```

- Použití ukazatele bez inicializace
  - častá **chyba** začátečníků
  - zápis do nealokované paměti ➔ **havárie**

Příklad: zápis do nedefinovaného místa paměti. Takto ne:

```
float *px, *py;
*px = 12.5;
*py = 54.1;
```

Inicializace ukazatele

Příklad: chybná reference ukazatele – takto nelze:

```
int q;          // sizeof(int) ~ 4B
double *fp;     // sizeof(double) ~ 8B
fp = &q;        //překladač vypisuje varování
*fp = 100.23; //přepíše paměť sousedící s q
```

- Bázový typ
  - určuje velikost odkazované paměti
  - počet bajtů při kopírování paměti a při nepřímém porovnávání
  - Nikdy vzájemně nepřiřazovat ukazatele různých typů (až na výjimky)!

Konstantní ukazatel, ukazatel na konstantu

- Konstantní ukazatel
  - nelze měnit, odkazovanou paměť ano
- Ukazatel na konstantu
  - lze měnit, odkazovanou paměť ne

```
int i;
int *pi;                  // ukazatel
int * const CP = &i; // konstantní ukazatel
const int CI = 7;     // celočíselná konstanta
// neinicializovaný ukazatel na konstantu
const int *pci;
// konstantní ukazatel na konstantu
const int * const CPC = &CI;
```

Obecný ukazatel

- Typový ukazatel
  - umožňuje typovou kontrolu
- Obecný ukazatel (void *)
  - neumožňuje typovou kontrolu
  - ale lze jej přetypovat na jakýkoli typový ukazatel ➔ kompatibilní se všemi ukazateli
  - nelze použít dereferenční operátor ➔ pro praktické použití nutno přetypovat
  - pro situace, kdy je potřeba měnit typ ukazatele za běhu programu podle kontextu řešené úlohy (jde o speciální případy!)

Obecný ukazatel

Příklad: datový typ void a přetypování ukazatele.

```
int inum = 10;
float fnum = 3.14;
void *pgeneric = &inum;
//přetypování je nutné, inum je nyní 20
*(int *)pgeneric = 20;

pgeneric = &fnum;
//nastaví fnum na 2.72
*(float *)pgeneric = 2.72;
```

Konverze ukazatelů, adresová aritmetika

- Konverze ukazatelů
  - pomocí operátoru přetypování
  - potenciálně nebezpečná operace – programátor na sebe bere veškerou zodpovědnost za rizika!
- Adresová aritmetika
  - nad ukazateli fungují aditivní a relační operátory
  - využitelné jako náhrada indexování (většinou ale na úkor přehlednosti)
  - není náplní tohoto kurzu

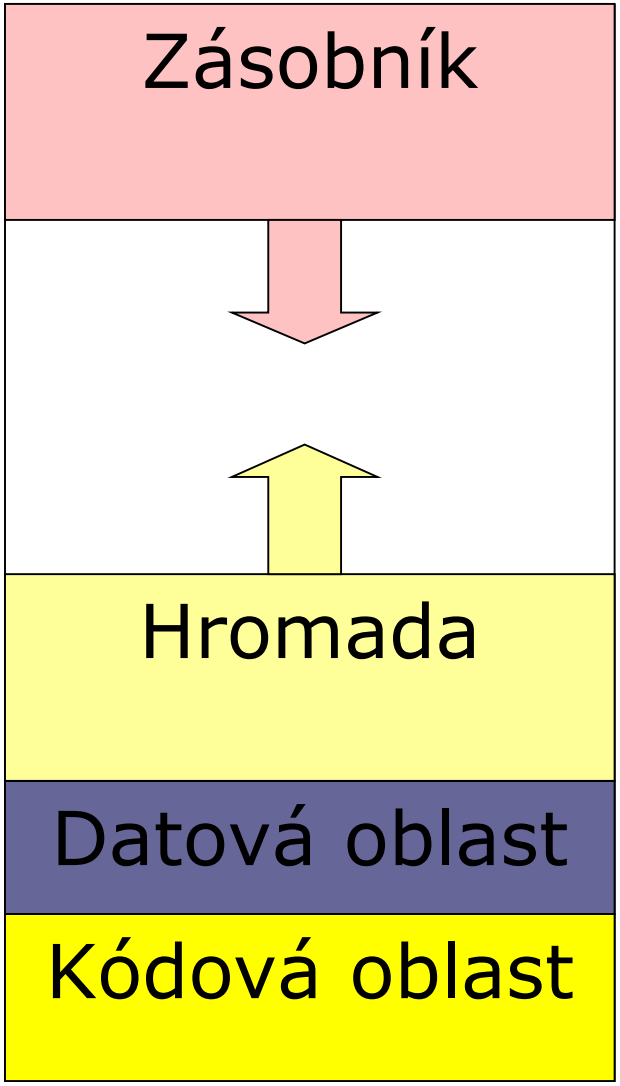
Vícenásobný nepřímý odkaz

- Ukazatel odkazující na jiný ukazatel
  - funkce – předávání ukazatelů odkazem
  - tvorba složitějších dynamických datových struktur (seznam, zásobník, strom, …)
  - Např. při vytváření vícerozměrných polí

Příklad: ukazatel může ukazovat i na jiný ukazatel.

```
int i = 12;
int *pi = &i;
int **ppi = &pi; //ukazatel na ukazatel na int
int j = **ppi;   //dvojitá dereference
```

Alokace paměti

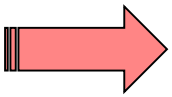


- Paměťové nároky proměnných
  - sizeof(typ) + zarovnání
- Paměťový prostor
  - Zásobník (stack) – lokální proměnné + parametry funkcí
  - Hromada (heap) – dynamické proměnné
  - Dat. oblast – globální proměnné + konstanty
  - Kódová oblast – kód programu

Alokace paměti

- Statická alokace
  - v datové oblasti programu
  - překladač musí znát předem velikost dat
  - během zavádění programu do paměti
  - modifikátor paměťové třídy – viditelnost
    - auto – implicitní pro lokální proměnné

```
auto int i;
auto int j = 12;
```



```
int i;
int j = 12;
```

    - extern – implicitní pro globální proměnné

```
//  soubor S1.C
int suma;
```

```
//  soubor S2.C
extern int suma;
```

Alokace paměti

- static – není implicitní definice, proměnné jsou uloženy v datové oblasti

```
int x = 5;
static int j = 12;
```

- register – proměnná je uložena pouze v registru

```
register int j;
```

- Dynamická alokace
  - zásobník + hromada
  - teoreticky lze využít celou paměť
  - za běhu programu

Alokace paměti

- Dynamická (automatická) alokace na zásobníku
  - lokální proměnné, parametry funkcí
  - automaticky při volání funkcí
  - při skončení funkce se automaticky uvolní
- Dynamická alokace na hromadě
  - na hromadu nelze přistupovat pomocí proměnné ➔ nutno použít ukazatel
  - Programátor je zodpovědný za správnou alokaci i uvolnění (dealokaci) paměti

Alokace na hromadě v jazyce C

- Jazyk C nemá syntaktické prostředky pro práci s hromadou.
- Přidělování paměti provádí funkce *malloc* z rozhraní <stdlib.h>

```
void *malloc(size_t size);
```

- size – počet alokovaných **bajtů**
- vrací obecný ukazatel nebo NULL, pokud došlo k chybě (např. není dostatek paměti)

Alokace na hromadě v jazyce C

- Nutné používat spolu se sizeof!

Příklad: na všech platformách alokuje správně velkou paměť.

```
int *pi = malloc(sizeof(int));
```

- Vždy je třeba otestovat návratovou hodnotu.

```
if ((pi = malloc(sizeof(int))) == NULL)
{ // zpracuj chybu
  return ERR_MALLOC;
}
```

Alokace na hromadě v jazyce C

- Nikdy nesmíme ztratit ukazatel na alokovanou paměť!
- Ztráta ukazatele → **memory leak** → ztracenou paměť nelze získat zpět!

Příklad: ztráta ukazatele na alokovanou paměť.

```
int *pi = malloc(sizeof(int));
...
pi = pj; // ztráta původního ukazatele
```

Uvolnění paměti

- S pamětí je nutno dobře hospodařit ➔ nepoužívanou paměť uvolnit – free().

```
void free(void *ptr);
```

- Platí pravidlo:
  - ***Ke každému volání funkce malloc() patří jedno volání funkce free().***

```
int *pi = malloc(sizeof(int));
...
free(pi);
```

Datový typ pole

- Deklarace pole.
- Přiřazení, kopie pole.
- Řetězce.
- Vícerozměrná pole.
- Inicializace pole.
- Pole s neurčenou velikostí.
- Alokace dynamického pole.
- Použití ukazatelů pro práci s poli.

Datový typ pole

- Složený (agregovaný) typ
  - Označuje skupinu hodnot
  - Pole, struktura, union
- Pole
  - Kolekce hodnot (prvků) stejného typu
  - V C – spojitá oblast paměti
- Prvek pole
  - Přístup pomocí identifikátoru pole a indexu

Datový typ pole

- Pro deklaraci jednorozměrného pole se používá obecný formát:

typ_pole	*jméno_pole*	*[velikost];*

typ prvků pole

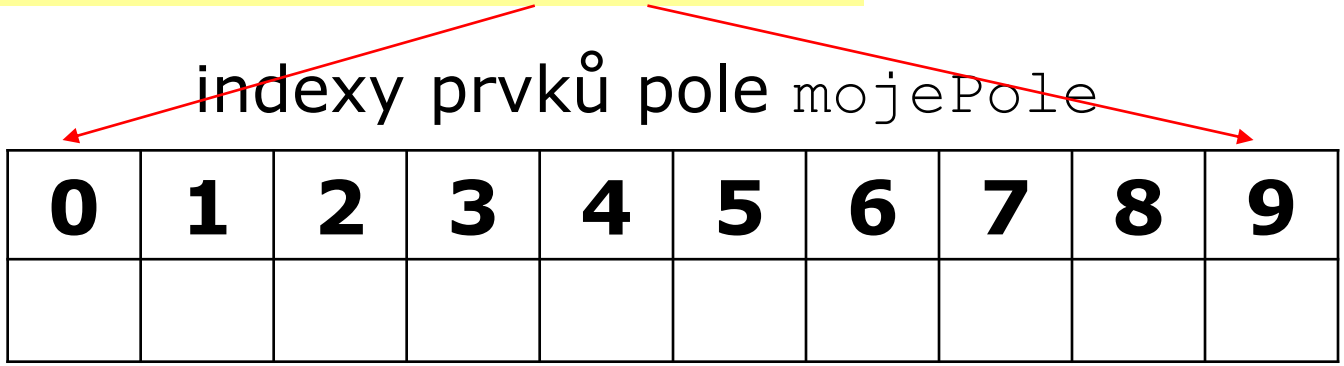
identifikátor pole

kladný počet prvků pole

Datový typ pole

Příklad: deklarace pole o deseti prvcích.

```
int mojePole[10];
```



- Prvek pole lze získat ***indexováním***.
- V C všechna pole začínají indexem 0
  - První prvek: mojePole[0]

Datový typ pole

- Obecný formát přístupu k prvku pole:
  *jméno_pole**[**výraz**]***

```
mojePole[0]  = 100; //první prvek

mojePole[1]  = 5  //druhý prvek
```

- Jednorozměrné pole – souvislá paměť
- První prvek – nejnižší adresa

Datový typ pole

Příklad: naplní prvky pole hodnotou indexu.

```
int pole[5];
for(int i=0; i<5; i++)
  pole[i] = i;
```

pole[0]	pole[1]	pole[2]	pole[3]	pole[4]
0	1	2	3	4

- Prvek pole je L-hodnota

Datový typ pole

Příklad: naplní pole sqrs druhými mocninami čísel od 1 do 10 a pak je vypíše.

```
#include <stdio.h>
#define N 10
int main(void)
{
  int sqrs[N];
  for(int i=1; i<=N; i++) // tak ne!
    sqrs[i-1] = i*i;

  for(int i=0; i<N; i++)  // tak ano
    printf("%d ", sqrs[i]);
  return 0;
}
```

Meze polí

- Jazyk C nekontroluje meze polí!
  - Musí ohlídat programátor
  - Indexace mimo meze pole ➔ zhroucení programu nebo poškození vnitřních dat ➔ podstata bezp. chyby ***buffer-overflow***!

Příklad: načtení celého čísla za hranicí pole.

```
int count[5];
scanf("%d", &count[9]);
// nelze!, překladač to nehlídá
```

Přiřazení, kopie pole

- V C nelze přiřazovat pole – neexistuje operátor přiřazení pro pole
  - Neplést s přiřazováním ukazatelů!

Příklad: nelze přiřadit najednou celé pole jinému poli.

```
char prvni[10], druhe[10];
.
.    // naplnění prvků pole prvni hodnotami
.
druhe = prvni; // nelze
```

Přiřazení, kopie pole

- Kopie – cyklus, memcpy()

Příklad: naplní pole *prvni* čísly 1 až 10 a pak je zkopíruje do pole *druhe* a zobrazí.

```
int prvni[POCET], druhe[POCET];
for(int i=1; i <= POCET; i++) //?!?
  prvni[i-1] = i;
for(int i=0; i < POCET; i++)
  druhe[i] = prvni[i];
for(int i=0; i < POCET; i++)
  printf("%d ", druhe[i]);
```

Použití řetězců

- Textový řetězec = pole typu char
- V C – řetězec ukončen znakem '\0'
- Pole musí být minimálně o znak delší než řetězec.
- Řetězcové konstanty jsou ukončeny nulou automaticky

0	1	2	3	4
'A'	'd'	'a'	'm'	'\0'

ASCII Code [on line, cit. 2019-10-12]

Použití řetězců

Příklad: čte řetězec zadaný z klávesnice. Pak vypíše obsah řetězce po znacích.

```
char str[N];
printf("Zadejte řetězec znaků:\n");
fgets(str, N, stdin);// nikdy ne gets - hrozí přetečení !!!
                                                    // [HePa13, str. 206]
int i = 0;
while(str[i] != '\0')
{
  printf("%c", str[i]);
  i++;
}
```

Vícerozměrná pole

- Dvourozměrné pole
  - pole jednorozměrných polí

Příklad: deklarace dvourozměrného pole

	0	1	2	3	4
0					
1					
2					
3					

```
#define RADKY 4
#define SLOUPCE 5
.
.
float matice[RADKY][SLOUPCE];
```

Vícerozměrná pole

Příklad: zaplní pole součiny indexů a pak zobrazí pole po řádcích.

```
int matice[RADKY][SLOUPCE];
for(int r=0; r < RADKY; r++)
  for(int s=0; s < SLOUPCE; s++)
    matice[r][s] = r*s;

for(int r=0; r < RADKY; r++)
{
  for(int s=0; s < SLOUPCE; s++)
  { printf("%d ", matice[r][s]); }

  printf("\n");
}
```

Inicializace pole

- Obecný formát inicializace polí pomocí inicializátoru:

*typ jméno_pole**[**velikost**]**=**{**seznam-hodnot**};**

- seznam hodnot – typově kompatibilní konstanty oddělené čárkami

Příklad: pětiprvkové celočíselné pole inicializováno mocninami čísel 1 až 5.

```
int pole[5] = {1, 4, 9, 16, 25};
```

Inicializace pole

Příklad: tříprvkové znakové pole inicializováno znaky 'A' 'B' 'C'.

```
char a[3] = {'A', 'B', 'C'};
```

- Textové řetězce lze inicializovat jednodušeji – překladač doplní znak '\0'

Příklad: pětiprvkové pole, do kterého je uložen řetězec.

```
char name1[5] = "Adam";
char name2[]  = "Adam";
```

0	1	2	3	4
'A'	'd'	'a'	'm'	'\0'

Inicializace pole

- Vícerozměrná pole – závorkovat
  - jinak překladač vypíše varování

```
int mesice[4][3] =
      {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};

int mesice[4][3] =
{
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

Inicializace pole

- Není nutné vyjmenovat všechny prvky
  - ISO C99
  - Neuvedené prvky mají hodnotu 0
  - Neinicializované pole má nedefinované hodnoty

```
#define N 5
int pole0[N];                    // ? ? ? ? ?
int pole1[N] = {0};              // 0 0 0 0 0
int pole2[N] = {5};              // 5 0 0 0 0
int pole3[N] = {1, [3]=2};       // 1 0 0 2 0
```

Pole s neurčenou velikostí

- Nejlevější rozměr není třeba u inicializovaných polí specifikovat
  - Překladač doplní počet prvků podle inicializátoru

Příklad: osmiprvkové pole inicializované hodnotami mocniny čísla 2.

```
int pwr[] = {1, 2, 4, 8, 16, 32, 64, 128};
```

Příklad: pole s neurčenou velikostí pro uložení textu výzvy.

```
char prompt[] = "Zadejte svoje jmeno: ";
```

Pole s neurčenou velikostí

Příklad: vícerozměrné pole s neurčenou velikostí

```
int  mesice[][3] =
{
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9},
   {10, 11, 12}
};
```

Globální, lokální deklarace pole a inicializace

- Globální pole
  - Rozměr – jen symbolické konstanty (#define)
  - Lze použít inicializátor
- Lokální (automatické) pole
  - Rozměr – i celočíselným výrazem
  - Inicializátor lze použít pouze u polí s rozměrem specifikovaným symbolickou konstantou

```
#define N 5
//prvky lze vyjmenovat
int pole1[N] = {3, 4, 1, 0, 2};
```

Globální, lokální deklarace pole a inicializace

- Pole s rozměrem specifikovaným proměnnou
  - nelze použít inicializátor
  - inicializace cyklem, memcpy

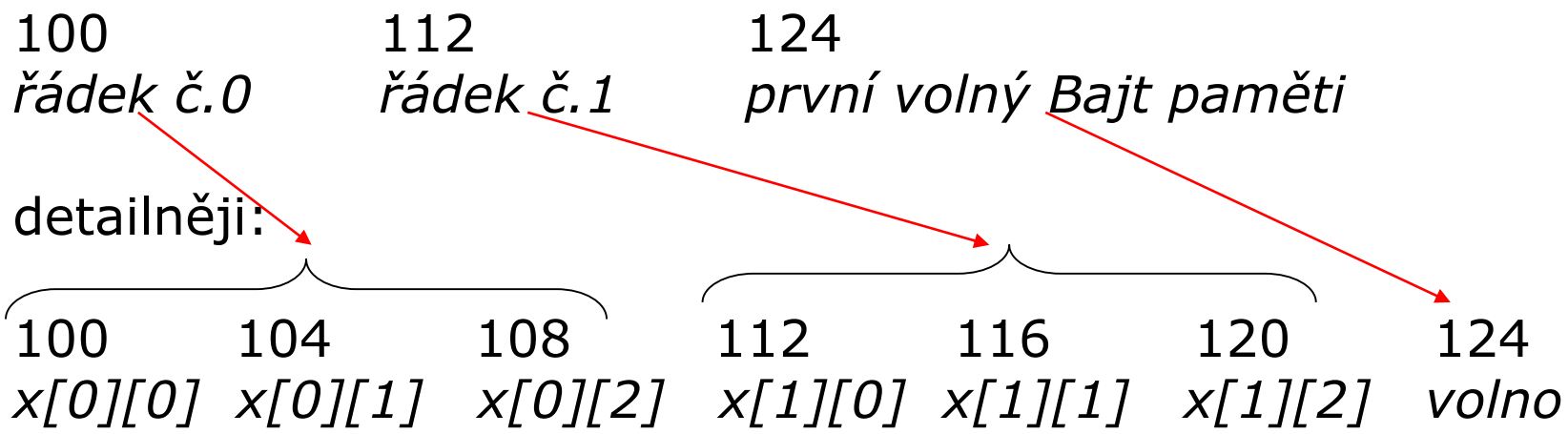
```
void funkce(int n)
{
  int pole[n]; //nelze použít inicializátor
  for (int i = 0; i < n; i++)
    pole[i] = 0;
...
}
```

Alokace polí
(uložení vícerozměrných polí v paměti)

- 2D statické pole
  - v paměti souvisle po řádcích

```
int x[2][3]; // alokuje v paměti 2*3 prvků
```

Např. počáteční adresa pole je 100, pak je obsazení paměti:



Alokace dynamického pole

- Dynamické pole na zásobníku (ISO C99)
  - lokální pole ➔ nelze vracet z funkce!
  - o alokaci a uvolnění se stará překladač (pole zanikne při ukončení funkce)
  - nelze detekovat nedostatek paměti!
  - zásobník mívá omezenou velikost!
  - používat opatrně (nebo vůbec)

```
void praceNaPoli(int delka){
  int pole[delka][delka+2];
  ...
}
```

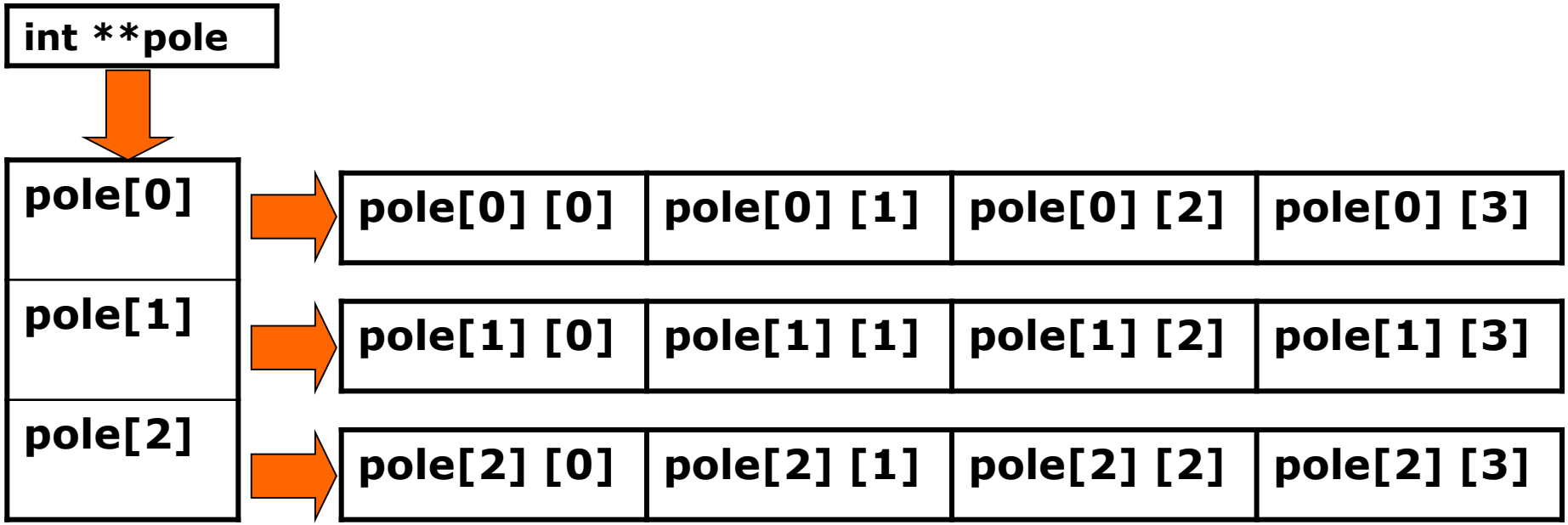
Alokace dynamického pole

- Dynamické pole na hromadě
  - může přežít konec funkce
  - lze alokovat více než na zásobníku + detekovat nedostatek paměti
  - příbuznost polí a ukazatelů
  - inicializovat lze cyklem nebo *memset()*

```
int *pole = malloc(delka * sizeof(int));
if (pole == NULL) chyba();
for (int i=0; i < delka; i++)
  pole[i] = hodnota;
...
free(pole);
```

Alokace dynamického pole

- Dynamická, vícerozměrná pole
  - pomocí vícenásobného ukazatele



  - další možnosti – viz literatura

Alokace dynamického pole

Příklad: alokace dvourozměrného pole.

```
// nejprve pole ukazatelů na int
int **pole = malloc(RADKU*sizeof(int *));
if (pole == NULL) chyba();
// potom jednotlivé řádky
for (int i = 0; i < RADKU; i++)
{
  pole[i] = malloc(SLOUPCU*sizeof(int));
  if (pole[i] == NULL) chyba();
}
// nyní lze pole běžně používat
pole[1][2] = 10;
```

Alokace dynamického pole

Příklad: takto to nejde, protože překladač nemá informaci o rozměrech pole.

```
int **pole = malloc(RADKU*SLOUPCU*sizeof(int));
pole[1][2] = 10; // chyba za běhu !!

// chyba při překladu !!
int pole[][] = malloc(RADKU*SLOUPCU*sizeof(int));
```

- Dealokace
  - opačný algoritmus – nejprve řádky, pak vektor ukazatelů

Použití ukazatelů pro práci s poli

- Samotný identifikátor pole ~ konstantní ukazatel na první prvek
  - lze používat adresovou aritmetiku → raději ne

Příklad: zobrazení prvních třech prvků pole pomocí ukazatelové aritmetiky.

```
int a[10] = {1, 20, 3, 40, 5, 60, 7, 80, 9, 100};
int *p = a; // přiřadí do p adresu začátku pole
// zobrazí  první, druhý a třetí prvek
printf("%d %d %d\n", *p, *(p+1), *(p+2));
// zobrazí stejné prvky pomocí indexů
printf("%d %d %d\n", a[0], a[1], a[2]);
```

Použití ukazatelů pro práci s poli

- U vícerozměrných polí je indexování jednodušší než adresová aritmetika

Příklad: Přístup k prvku dvourozměrného pole pomocí ukazatele.

```
float hotovost[4][3];
float *p = *hotovost;
```

*(p + (2*3) + 1) ~
p[2*3+1] ~
hotovost[2][1].

	0	**1**	**2**
0	*0*	*1*	*2*
1	*3*	*4*	*5*
2	*6*	*7*	*8*
3	*9*	*10*	*11*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*

Proč je zde *hotovost a ne hotovost nebo **hotovost?

Použití ukazatelů pro práci s poli

Příklad: ukazatel lze indexovat jakoby to bylo pole.

```
char str[] = "Ukazatele jsou bezva.\n";
char *p = str;

// cyklus trvá dokud se nenajde znak s kódem 0
int i=0;
while(p[i] != '\0')
  printf("%c", p[i++]);
```

- Hodnotu ukazatele tvořeného jménem pole nelze změnit!

Vytvoření polí ukazatelů

Příklad: pole ukazatelů na *int*, které má 20 prvků.

```
int *pa[20];
```

Příklad: adresa celočíselné proměnné mojePromenna se přiřadí devátému prvku pole.

```
pa[8] = &mojePromenna;
```

Příklad:pro přístup k hodnotě za ukazatelem je nutné použít dereferenční operátor.

```
*pa[2] = 100;
```

Použití ukazatelů pro práci s poli

Příklad: pole (s neurčenou velikostí) ukazatelů.

```
enum errcodes {ERR_NOER, ERR_PROFF, ERR_PAPER};
void error(int err_num)
{
  static const char *p[] =
  {
    "Bez chyby!",
    "Tiskarna neni zapnuta.",
    "Chybi papir."
  };
  fprintf(stderr, "%s\n",p[err_num]);
}
...
  error(ERR_PAPER);
```

Funkce a předávání parametrů

- Existují dva druhy předávání parametrů
  - Liší se tím, co se děje s argumentem při volání
  - Jazyk C syntakticky tyto dva druhy nerozlišuje
- Předávání hodnotou
  - Při volání – kopie argumentu do parametru
  - Parametr – lokální proměnná
  - Změna hodnoty ve funkci se vně neprojeví
  - Lze předávat P-hodnoty (tedy i konstanty)

Funkce a předávání parametrů

- Předávání odkazem
  - Formální parametr propojen s argumentem pomocí ukazatele
  - Změna hodnoty ve funkci se projeví i vně
  - Argument musí být L-hodnotou
  - Jazyk C syntakticky nezná – je nutné předávat ukazatele
    - Při volání je nutné zajistit předání ukazatele – u jednoduchých typů pomocí referenčního operátoru (&)
    - Změna hodnoty ve funkci – dereferencí (*)

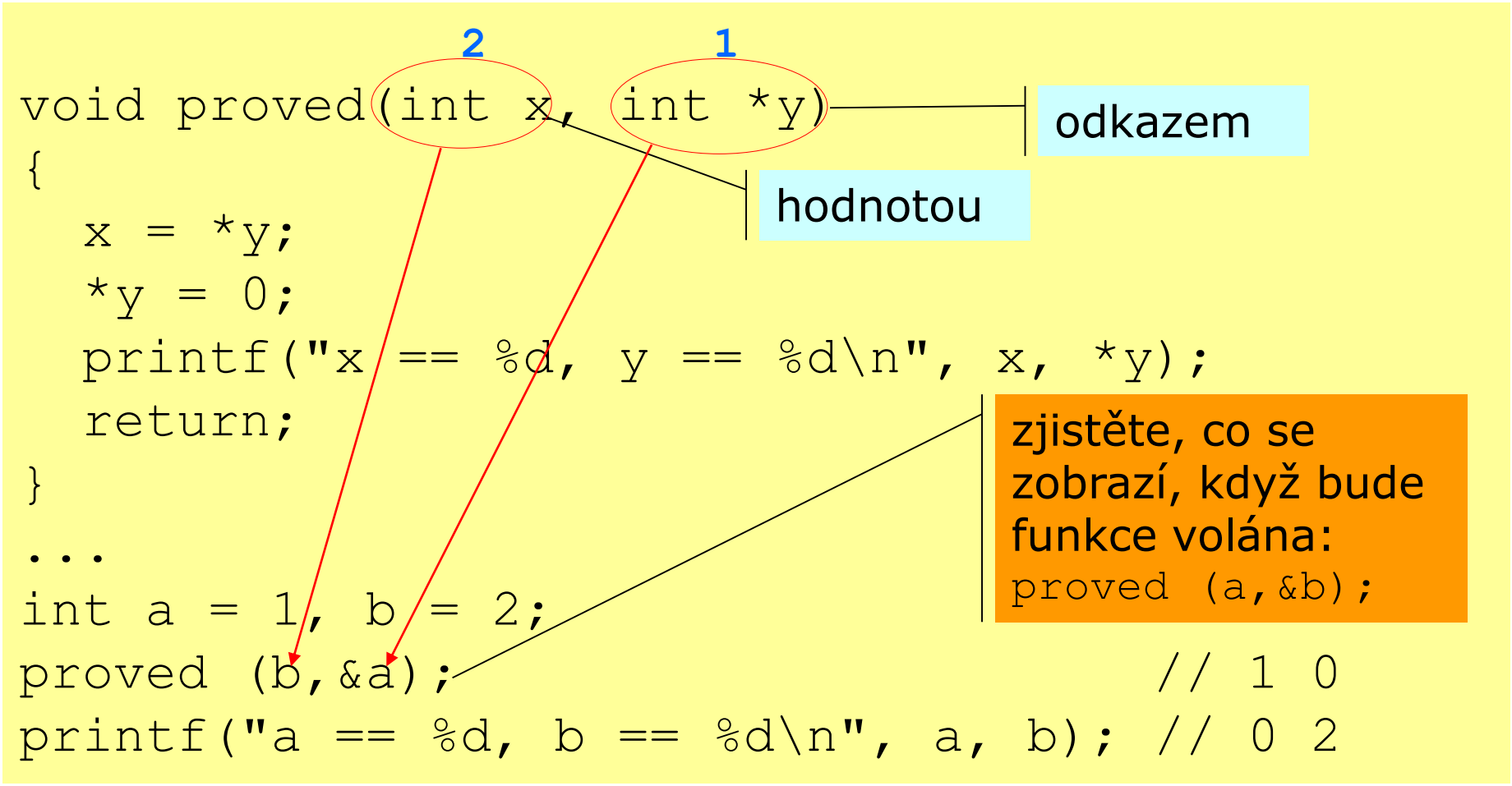
Funkce a předávání parametrů

Příklad: záměna dvou proměnných.

```
void zamen(int *pa, int *pb)
{
  int pom = *pa; *pa = *pb; *pb = pom;
}
int main(void)
{
  int i = 7, j = 3;
  printf("i == %d, j == %d\n", i, j);
  zamen(&i, &j); // pozor,nesmí chybět &
  printf("i == %d, j == %d\n", i, j);
  return 0;
}
```

Funkce a předávání parametrů

Příklad: parametry předávané odkazem a hodnotou



Funkce a předávání parametrů

Příklad: načte zadaný počet čísel ze *stdin* a uloží je do pole.

```
int nactiPole(int pole[], int delka){
  int i = 0;
  while(i < delka && scanf("%d", &pole[i]) == 1)
    i++;
  if(i != delka) return ERR_SHORT_ARRAY;
  return EOK;
}
...
int *pole = malloc(N*sizeof(int));
int errcode = nactiPole(pole, N);
...
free(pole);
```

Funkce a předávání parametrů

- Předávání polí
  - V C: pole ~ ukazatel
  - Vždy se předá ukazatel, pole se na zásobník nekopíruje

Příklad: přičte hodnotu n ke všem prvkům pole.

```
void prictiN(int pole[], int delka, int n)
{
  for(int i = 0; i < delka; i++)
  { pole[i] += n; }
}
...
prictiN(pole, DELKA, 5); // zde není &pole
```

Funkce a předávání parametrů

- Alternativní prototyp stejné funkce

```
void prictiN(int *pole, int delka, int n);
```

- Ukazatel a pole nejsou shodné typy
  - Datový typ pole uchovává informace navíc
  - Např. u vícerozměrných polí uchovává informaci o vyšších dimenzích pole, pro správný výpočet polohy indexovaného prvku

Funkce a předávání parametrů

- Předávání ukazatele odkazem
  - Syntakticky stejné, jako u jiných typů předávaných odkazem.

Příklad: Alokuje nové pole a vrací ho přes parametr.

```
int alokuj(int **pole, int delka){
  *pole = malloc(delka * sizeof(int));
  if (*pole == NULL) return ERR_MEM;
  for(int i = 0; i < delka; i++)
  { (*pole)[i] = 0; } // () kvůli prioritám
  return EOK;
}
...
int *pole;
int err = alokuj(&pole, 5); // zde je &pole!
```

Typ ukazatel, strukturované datové typy

Kontrolní otázky

1. Co je ukazatel?
2. Jaké jsou ukazatelové operátory a jaká je jejich funkce?
3. Vysvětlete, co je nulový ukazatel a k čemu slouží.
4. Charakterizujte statickou alokaci paměti.
5. Charakterizujte dynamickou alokaci paměti.
6. Co je pole?
7. Jak se provede kopie jednoho pole do jiného pole?
8. Jaké jsou možnosti předávání argumentů funkcím?
9. Jaká je výhoda použití ukazatelů místo indexování polí?
10. Jaký je vzájemný vztah polí a ukazatelů?
11. Proč lze při deklaraci vícerozměrného pole s inicializátorem nebo jako parametru funkce vynechat nejlevější rozměr pole?
12. Proč nelze při deklaraci pole jako globální proměnné použít pro specifikaci jeho rozměrů proměnné?
13. Lze v nějakém případě použít při deklaraci pole proměnné pro specifikaci jeho rozměrů. Pokud ano, které to jsou a proč je to v těchto případech možné?
14. Existuje nějaké principiální omezení jazyka C, které mu znemožňuje automaticky hlídat meze polí?

Úkoly k procvičení

1. Deklarujte ukazatel na double.
2. Inicializujte celočíselné pole nazvané items hodnotami od 1 do 10.
3. Deklarujte pole s neurčenou velikostí. Pole bude obsahovat mocniny čísla 3 až do čísla 729.
4. Napište program, který předává funkci ukazatel na celočíselnou proměnnou. Uvnitř funkce přiřaďte proměnné hodnotu -5. Po návratu z funkce ukažte, že proměnná obsahuje skutečně hodnotu -5 tak, že ji vypíšete.
5. Schematicky znázorněte, jak bude v paměti vypadat dvourozměrné pole nad typem int, alokované staticky.
6. Schematicky znázorněte, jak bude v paměti vypadat dvourozměrné pole nad typem int, alokované dynamicky na hromadě.