Alokace dvourozměrného pole, ladění programu a realokace

IZP-cv08

Ing. Jakub Husa

Vysoké Učení Technické v Brně, Fakulta informačních technologií
Božetěchova 1/2. 612 66 Brno - Královo Pole
ihusa@fit.vut.cz

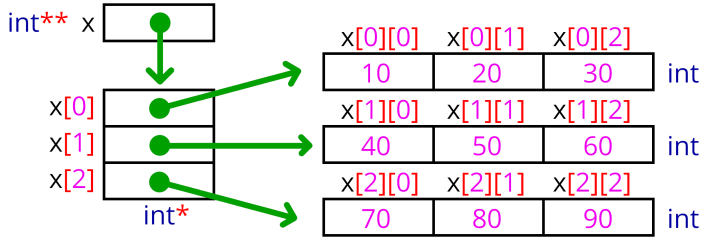
20. listopadu 2023

Alokace dvourozměrného pole

Dvourozměrné (2D) pole je polem několika jednorozměrných (1D) polí:

- Každé z polí vyžaduje samostatné volání funkce malloc (a free).
- K prvkům pole přistupujeme dvojitými hranatými závorkami ([] []).
- Řádky dynamicky alokovaného pole nemusejí mít stejnou délku.

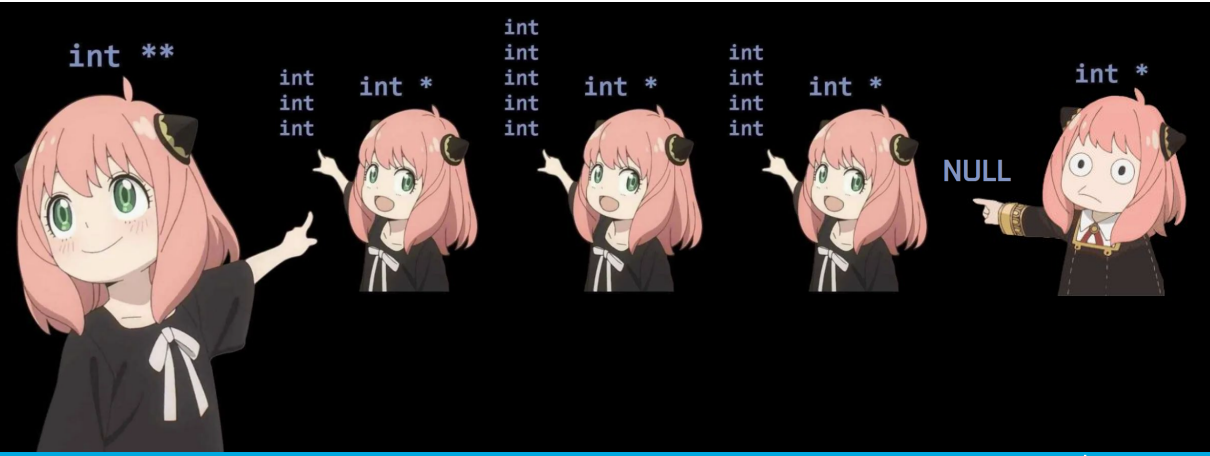
```
1  int radky = 3, sloupce = 3;                  //pocet radku a sloupcu pole
2  int** x = malloc(sizeof(int*) * radky);      //alokujeme 2D pole (radky)
3  for(int i = 0; i < radky; i++)               //pro kazdy radek
4      x[i] = malloc (sizeof(int) * sloupce); //alokujeme 1D pole (sloupce)
5
6  for(int i = 0; i < radky; i++)               //pro kazdy radek
7      for(int j = 0; j < sloupce; j++)         //pro kazdy sloupece
8          x[i][j] = (i*sloupce+j) * 10 + 10; //prvkum pole pocitame hodnotu
```



Problémy s alokací 2D pole

Pokud některá z alokací selže, všechny předcházející alokace musíme uvolnit:

```
1  int** x = malloc(sizeof(int*) * 4); //2D pole ctyr 1D poli
2  x[0] = malloc(sizeof(int) * 3);     //1D pole tri  celych cisel
3  x[1] = malloc(sizeof(int) * 5);     //1D pole peti celych cisel
4  x[2] = malloc(sizeof(int) * 4);     //1D pole ctyr celych cisel
5  x[3] = malloc(sizeof(int) * -1);    //1D pole ktere se nepodari alokovat
```



Ošetření alokace 2D pole

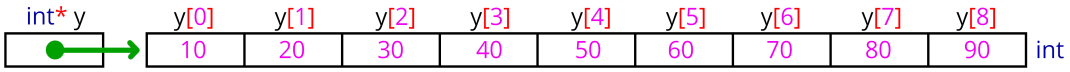
Všechna 1D pole uvolňujeme dříve než 2D pole které je obsahuje:

```
1  int radky = 3, sloupce = 3;                  //pocet radku a sloupcu pole
2
3  int** x = malloc(sizeof(int*) * radky);      //alokujeme 2D pole (radky)
4  if (x != NULL)                               //pokud alokace neselhala
5  {
6      for(int i = 0; i < radky; i++)           //pro kazdy radek
7      {
8          x[i] = malloc(sizeof(int)*sloupce);//alokujeme sloupce
9          if (x[i] == NULL)                    //pokud alokace selhala
10         {
11             for (int j = i-1; j >= 0; j--) //pro vsechny predchozi radky
12             {
13                 free(x[j]);                  //uvolnujeme radek
14             }
15             free(x);                         //uvolnujeme 2D pole
16             break;                           //koncime cyklus alokace
17         }
18     }
19 }
```

Místo 2D pole můžeme použít 1D pole se složitějším indexováním:

- K alokaci stačí jedno volání funkce malloc (a free).
- K prvkům pole přistupujeme jedněmi hranatými závorkami ([]).
- Jako kombinovaný index používáme index řádku * počet sloupců + index sloupce.

```
1  int radky = 3, sloupce = 3;                  //pocet radku a sloupcu pole
2  int* y = malloc(sizeof(int)*radky*sloupce);//alokujeme jedno velke 1D pole
3
4  for(int i = 0; i < radky; i++)               //pro kazdy radek
5      for(int j = 0; j < sloupce; j++)         //pro kazdy sloupec
6          y[i*sloupce+j]=(i*sloupce+j)*10+10;//prvkum pole pocitame hodnotu
```



Vyzkoušejte si:

- Napište program který jako argumenty příkazové řádky dostane dvě celá čísla, představující počet řádků a sloupců dvourozměrného pole znaků.
- Dynamicky alokujte jednorozměrné pole a načtěte do něj vstup.
- Načtené řádky vypište na výstup v obráceném pořadí.
- Nezapomeňte uvolnit alokovanou paměť a ošetřit neplané přístupy od paměti.

Například:

- `./main 2 2` => (ABCD) => CD
  => AB
- `./main 2 3` => (EFGHIJ) => HIJ
  => EFG
- `./main 3 2` => (KLMNOP) => OP
  => MN
  => KL

A	B
C	D

E	F	G
H	I	J

K	L
M	N
O	P

Ladění programu

Ladění je postup systematického hledání chyb v programu:

- Pro ladění programu ve vývojovém prostředí používáme debugger.
- Debugger umožňuje běh programu krokovat po jednotlivých příkazech, a sledovat hodnoty proměnných a zásobník volání funkcí.

Postup ladění v prostředí Code::Blocks:

- V menu Debug -> Debugging windows nastavíme co chceme sledovat.
  Watches zobrazuje aktuální hodnoty všech proměnných a parametrů funkce.
  Call Stack ukazuje jak je právě vykonávaná funkce zavolána.
- Před řádkem o kterém si myslíme že způsobuje problémy kliknutím napravo od čísla řádku nastavíme breakpoint, na kterém se vykonávání programu zastaví.
- Tlačítkem Debug / Continue (F8) spustíme ladění.
- Tlačítky Next line (F7) a Step into (Shift-F7) krokujeme běh programu.

Na procvičení

Vyzkoušejte si:

- Sledujte Watches a Call Stack při krokování následujícího programu.

```
1  int max(int delka, int pole[])        //definicie funkce
2  {
3      int max = pole[0];                //pocatecnihodnota maxima
4      for(int i = 0; i < delka; i++)    //pro vsechny prvky pole
5          if (max < pole[i])            //pokud najdeme prvek ktery je vetsi
6              max = pole[i];            //aktualizujeme hodnotu maxima
7      return max;                       //vracime maximum
8  }
9
10 int main()                            //zacatek programu
11 {
12     int poleA[5] = {1, 4, 3, 5, 2};  //vytvarime poleA
13     int poleB[4] = {10, 30, 20, 40}; //vytvarime poleB
14
15     printf("Maximum poleA je %i\n", max(5, poleA)); //vypis
16     printf("Maximum poleB je %i\n", max(4, poleB)); //vypis
17     return 0;                         //konec programu
18 }
```

Detekování úniků paměti

V Unixu můžeme úniky paměti detekovat programem Valgrind:

- Valgrind kolem našeho programu vytvoří obálku která mu umožní detekovat jestli náš program před ukončením uvolnil veškerou alokovanou paměť:
- Programy které chceme debugovat překládáme s parametrem `-g` :

```
gcc -g main.c -o main
valgrind --leak-check=full --show-leak-kinds=all ./main
```

- Pokud k žádným únikům nedošlo Valgrind by měl vypsat:

```
All heap blocks were freed -- no leaks are possible
```

- V opačném případě Valgrind vypíše kterou alokaci jsme zapomněli uvolnit:

```
40 bytes in 1 blocks are definitely lost in loss record 1 of 1
at 0x4C29B0D: malloc (vg_replace_malloc.c:299)
by 0x400556: main (main.c:8) <- neuvolnena alokace
```

  Problém nastal ve funkci main v souboru main.c na řádku 8.

Hlášení no leaks are possible znamená že v daném běhu nemohlo dojít k únikům:

- **NEZNAMENÁ** že k nim nemůže dojít pro jiné argumenty nebo vstupní data!
- Kromě úniků Valgind detekuje také neplané přístupy do paměti.
- Ve Windows je detekování úniků paměti složitější.

Vyzkoušejte si:

- Na serveru merlin.fit.vutbr.cz pomocí Valgind debugujte následující program který načte posloupnost tří celých čísel a ověří jestli je rostoucí.
- Ošetřete neúspěšnou alokaci a opravte úniky a neplané přístupy do paměti.

```
1  int main()                          //zacatek programu
2  {
3      int* pole = malloc(3);          //alokujeme pole tri prvku
4      for(int i = 0; i < 3; i++)      //pro vsechny prkvy pole
5          scanf("%i", &pole[i]);      //nacitame vstup
6      bool rostouci = true;           //posloupnost povazujeme za rostouci
7      for(int i = 0; i < 3; i++)      //pro vsechny prvky pole
8      {                               //kontrolujeme sousedni prvky
9          if (pole[i] >= pole[i+1]) //pokud nasledujici prvek neni vetsi
10             rostouci = false;       //posloupnost neni rostouci
11     }
12     if (rostouci)                   //pokud jsme nenasli problem
13         printf("JE rostouci\n");    //posloupnost JE rostouci
14     else                            //jinak
15         printf("NENI rostouci\n");//posloupnost NENI rostouci
16     return 0;                       //konec programu
17 }
```

Realokace

Realokace paměti

Velikost alokované paměti můžeme změnit voláním funkce realloc z knihovny stdlib.h:

- Vstupem funkce je současná adresa realokovaného pole a jeho nová velikost.
- Výstupem funkce je nová adresa pole.

Proměnné alokované na hromadě **NEJSOU** jeden souvislý blok dat:

- Pokud je na současné pozici dostatek místa, relokace pole rozšíří na novou velikost.
- Pokud na současné pozici dostatek místa není, realokace všechny položky překopíruje na novou adresu, a starou adresu uvolní.

Alokovanou paměť můžeme realokací i zmenšit:

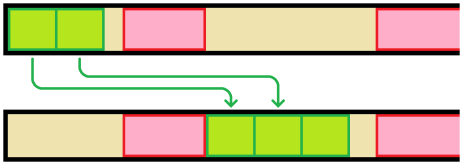
- Při zmenšování může dojít ke změně adresy.

```
1 //alokujeme pole peti cisel
2 int* x = malloc(sizeof(int)*5);
```

Alokovaná, Nedostupná a Volná Paměť

```
3 //realokujeme na deset cisel
4 x = realloc(x, sizeof(int)*10);
```

```
5 //realokujeme na patnact cisel
6 x = realloc(x, sizeof(int)*15);
```



Ošetření realokace

Pokud realokace selže, funkce vrátí NULL, ale původní adresu **NEUVOLNÍ**:

- Pokud jsme ukazatel přepsali, adresu již nelze uvolnit a nastává únik paměti.

```
1  int* x = malloc(sizeof(int)*5);          //tuto pamet nepujde uvolnit
2  x = realloc(x, sizeof(int)*10);          //pokud tato realokace selze
```

Uniklá paměť zůstane nepoužitelná dokud nedojde k ukončení celého programu:

- Realokaci paměti ošetříme použitím pomocného ukazatele.

```
3  int* x = malloc(sizeof(int)*5);          //alokujeme pole peti cisel
4  if (x != NULL)                           //pokud alokace uspela
5  {
6      /* ??? */       //z nejakeho duvodu budeme potrebovat vice mista
7
8      int* y = realloc(x, sizeof(int)*10); //realokujeme pomocny ukazatel
9      if (y != NULL) //pokud realokace uspela
10         x = y;      //aktualizujeme puvodni ukazatel
11     else            //jinak (pokud realokace selhala)
12         free(x);    //uvolnujeme starou adresu
13 }
```

Vyzkoušejte si:

- Sledujte jak realokace mění adresu pole.

```
1  int main()
2  {
3      int velikost = 1;                //velikost alokovane pameti
4      void* pole = malloc(velikost); //alokujeme pole (obecneho typu)
5
6      while (pole != NULL)             //dokud alokace neselhala
7      {
8          printf("Pole zabira %9i B na adrese %9i\n", velikost, pole);
9          velikost = velikost * 2;              //zdvojnasobujeme velikost
10         void* temp = realloc(pole, velikost); //realokujeme pole
11
12         if (temp != NULL)       //pokud realokace uspela
13             pole = temp;        //aktualizujeme adresu pole
14         else                    //jinak (pokud realokace selhala)
15             free(pole);         //starou adresu pole uvolnujeme
16     }
17     return 0;                   //program konci bez chyby
18 }
```

Na procvičení

Vyzkoušejte si:

- Implementujte následující funkce pro práci s vektory:

```
1 void vypisVektor(vektor* A);
2 void pridejPosledni(vektor* A, int cislo);
3 void odeberVsechny(vektor* A);
4 void pridejPrvni(vektor* A, int cislo);
5 void odeberJeden(vektor* A, int index);
```

- vypisVektor vypíše hodnotu všech prvků pole vektoru.
- pridejPosledni provede realokaci a nový prvek přidá na konec.
- odeberVsechny smaže všechny prvky a uvolní paměť.
- pridejPrvni provede realokaci a nový prvek přidá na začátek.
- odeberJeden smaže prvek z daného indexu a provede realokaci (pokud index neexistuje tak funkce neprovede nic).
- Kód funkce main si zkopírujte z následujícího slajdu.
- Na serveru merlin.fit.vutbr.cz ověřte že v programu nedochází k únikům ani k neplatnému přístupu do paměti.

Například:

```
1  Vektor:
2  Vektor: 10
3  Vektor: 10 20
4  Vektor:
5  Vektor: 30
6  Vektor: 40 30
7  Vektor: 50 40 30
8  Vektor: 50 40
9  Vektor: 40
10 Vektor: 40
11 Vektor:
```

Na procvičení

```
1  typedef struct Svektor                  //deklarujeme datovy typ pro strukturu
2  {                                       //jmenem "Svektor" se dvema polozkami
3      int delka;                          //pocet polozek
4      int* pole;                          //dynamicky alokovane pole
5  } vektor;                               //jmeno tohoto typu je "vektor"
6
7  int main()                                          //zacatek programu
8  {
9      vektor A = {0, NULL};    vypisVektor(&A); //vytvarime novy vektor
10     pridejPosledni(&A, 10); vypisVektor(&A); //pridavame posledni prvek
11     pridejPosledni(&A, 20); vypisVektor(&A); //pridavame posledni prvek
12     odeberVsechny(&A);       vypisVektor(&A); //mazeme vsechny prvky
13     pridejPrvni(&A, 30);     vypisVektor(&A); //pridavame prvni prvek
14     pridejPrvni(&A, 40);     vypisVektor(&A); //pridavame prvni prvek
15     pridejPrvni(&A, 50);     vypisVektor(&A); //pridavame prvni prvek
16     odeberJeden(&A, 2);      vypisVektor(&A); //mazeme posledni prvek
17     odeberJeden(&A, 0);      vypisVektor(&A); //mazeme prvni prvek
18     odeberJeden(&A, 1);      vypisVektor(&A); //mazeme neexistujici prvek
19     odeberJeden(&A, 0);      vypisVektor(&A); //mazeme prvni prvek
20     return 0;                                 //konec programu
21 }
```