Základy programovania (IZP)

Realokácia a debugging (8. cvičenie)

Ing. Pavol Dubovec

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

Čo to je debugging (ladenie)?

K čomu slúži a prečo sa používa?

- Ladenie (debugging):
  - je proces identifikácie a odstraňovania chýb v programe.
  - pomáha identifikovať a odstrániť chyby, ktoré by mohli spôsobiť zlyhanie programu, nesprávne výsledky alebo bezpečnostné hrozby.
  - umožňuje optimalizovať kód, aby bol efektívnejší a rýchlejší.
  - vplyv ladenia v posledných rokoch vzrástol.
- Ladenie vo VS code:
  - otvorte svoj projekt vo VS Code.
  - prejdite na Run and debug a kliknutím na ikonu ▷ v paneli aktivít alebo stlačením Ctrl+Shift+D.
  - nakonfigurujte svoje prostredie na ladenie vytvorením súboru launch.json a tasks.json, ak je to potrebné.
  - VS code obsahuje niekoľko panelov na ladenie:
    - **VARIABLES / Locals:** Tento panel zobrazuje lokálne premenné v aktuálnej oblasti pôsobnosti (scope). Pomáha kontrolovať hodnoty premenných v rôznych častiach kódu.
    - **WATCH:** slúži pre sledovanie výrazov počas prechádzania kódu.
    - **CALL STACK:** ukazuje postupnosť volaní funkcií, ktoré viedli k aktuálnemu bodu v programe.
    - **BREAKPOINTS:** umožňujú pozastaviť vykonávanie programu na konkrétnych riadkoch kódu. Je možné ich nastaviť kliknutím vľavo vedľa čísla riadku.

Teoretické opakovanie – typické kroky ladenia

1. Nastavenie zarážok (breakpoints) vo svojom kóde, na miestach kde je potrebné pozastavenie vykonávania.
2. Stlačením F5 alebo výberom možnosti `Run and debug` sa spustí ladenie.
3. Využitie panelov `Locals`, `WATCH` a `CALL STACK` na kontrolu premenných a toku vykonávania.
4. Úprava premenných a prechádzanie kódu, s cieľom identifikácie a opravy chýb.

Na čo používame program valgrind?

Teoretické opakovanie – Valgrind

- Valgrind je nástroj na ladenie a profilovanie programov, pre unix systémy ktorý pomáha identifikovať a opraviť problémy s pamäťou, ako sú úniky pamäte, neplatné prístupy do pamäte a neinicializované premenné.
- **Preklad s debug informáciami:** gcc -g -o nazov_programu nazov_programu.c
- **Spustenie programu s Valgrindom:** valgrind --leak-check=full --show-leak-kinds=all ./nazov_programu
- Pri korektnej práci s pamäťou vráti vráti:

  All heap blocks were freed -- no leaks are possible
  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
- Inak:

  ==20422== LEAK SUMMARY:
  ==20422== definitely lost: 448 bytes in 3 blocks
  ==20422== indirectly lost: 786,460 bytes in 1 blocks
  ==20422== possibly lost: 1,576,052 bytes in 46 blocks
  ==20422== still reachable: 1,077,107 bytes in 2,333 blocks
  ==20422== suppressed: 0 bytes in 0 blocks
  ==20422== Rerun with --leak-check=full to see details of leaked memory
  ==20422==
  ==20422== For counts of detected and suppressed errors, rerun with: -v
  ==20422== ERROR SUMMARY: 98307 errors from 5 contexts (suppressed: 2 from 2)
  Killed

Ako funguje realokácia pamäte?

- Realokácia pamäte umožňuje zmeniť veľkosť dynamicky alokovaného bloku pamäte (alokovať nový blok pamäte po dealokácii predtým alokovanej pamäte) pomocou funkcie `realloc` z knižnice `stdlib.h`.
- Použitie
  - **Vstup:** Aktuálna adresa alokovaného bloku a nová požadovaná veľkosť.
  - **Výstup:** Nová adresa alokovaného bloku.

```c
#include <stdlib.h>

int *ptr = (int *)malloc(10 * sizeof(int));
ptr = (int *)realloc(ptr, 20 * sizeof(int));
```

- Ak je na aktuálnej pozícii dostatok miesta, pole sa rozšíri na novú veľkosť.
- Ak nie je dostatok miesta, `realloc` skopíruje obsah na novú adresu a uvoľní starú adresu.

```c
int *ptr = (int *)malloc(10 * sizeof(int));
if (ptr == NULL) {
// Chyba pri alokácii
}
ptr = (int *)realloc(ptr, 20 * sizeof(int));
if (ptr == NULL) {
// Chyba pri realokácii
}
```

Teoretické opakovanie – realokácia pamäte 2

- Pri zmenšení veľkosti sa adresa môže zmeniť.
- Ak realloc zlyhá, vráti NULL, ale pôvodná adresa zostane nezmenená.
- Ak prepíšete pôvodný ukazovateľ bez kontroly návratovej hodnoty realloc, môže dôjsť k úniku pamäte.
- Na bezpečné prerozdelenie pamäte použite pomocný ukazovateľ.

```
int *temp = (int *)realloc(ptr, 20 * sizeof(int));
if (temp != NULL) {
ptr = temp;
} else {
// Chyba pri realokácii
}
```

Praktické cvičenie – dynamická pamäť a realokácia

1. **Úloha:** Implementujte funkcie, ktoré pridajú prvok na začiatok / koniec dynamického poľa. (1 čiarka)
2. **Úloha:** Implementujte funkcie, ktorá odstráni prvok na zadanom indexe z dynamického poľa / všetky prvky v dynamickom poli. (1 čiarky)
3. **Úloha:** Implementujte funkciu, ktorá k danému dynamickému poľu vytvorí pole unikátnych prvkov, pričom zachová konkrétne prvý výskyt tohto prvku. (2 čiarky)
4. **Úloha:** Implementujte funkciu, ktorá z dynamického poľa odstráni náhodný prvok menší ako hodnota priemeru prvkov a vráti výsledok. (2 čiarka)
5. **Úloha:** Implementujte funkciu, ktorá dynamicky alokuje a pracuje s 2D poľom (matica) s možnosťou pridávania riadkov a stĺpcov. (3 čiarky)
6. **Úloha:** Implementujte funkciu, ktorá nahradí dynamické pole čísel za dynamické pole štruktúr obsahujúcich pôvodné číslo v poli a jeho binárnu reprezentáciu. (3 čiarky)

```c
void append_d_array(dynamic_array_t* array, int item);// Vstup: [2, 3, 4], 1 → Výstup: [1, 2, 3, 4]
void prepend_d_array(dynamic_array_t* array, int item); // Vstup: [1, 2, 3], 4 → Výstup: [1, 2, 3, 4]
void remove_at_idx(dynamic_array_t* array, int index); // Vstup: [1, 2, 3, 4], 2 → Výstup: [1, 2, 4]
void remove_all(dynamic_array_t* array); // Vstup: [1, 2, 3, 4] → Výstup: []
dynamic_array_t* unique_elements(dynamic_array_t* array); // Vstup: [1, 2, 2, 3, 1] → Výstup: [1, 2, 3]
int remove_random_above_average(dynamic_array_t* array); // Vstup: [1, 2, 3, 4, 5] → Výstup: 4 (náhodný prvok väčší ako priemer), array = [1, 2, 3, 5]
dynamic_array_2D_t* create_2d_array(int initial_rows, int initial_cols); // Vstup: 2, 2 → Výstup: 2D pole s 2 riadkami a 2 stĺpcami
void add_row(dynamic_array_2D_t* array); // Vstup: 2D pole s 2 riadkami → Výstup: 2D pole s 3 riadkami
void add_column(dynamic_array_2D_t* array); // Vstup: 2D pole s 2 stĺpcami → Výstup: 2D pole s 3 stĺpcami
void free_2d_array(dynamic_array_2D_t* array); // Vstup: 2D pole → Výstup: uvoľnená pamäť
typedef struct {
    int number;
    char binary[32];
} int_with_binary_struct_t;
int_with_binary_struct_t* replace_int_with_struct(dynamic_array_2D_t* array); // Vstup: [5, 10, 15] → Výstup: [{5, "101"}, {10, "1010"}, {15, "1111"}]
```