



Financované  
Európskou úniou  
NextGenerationEU

**PLÁN [OBNOVY]**

---

Nitrianske kompetenčné centrum kybernetickej bezpečnosti  
Fakulta prírodných vied a informatiky  
Univerzita Konštantína Filozofa v Nitre  
Tr. A. Hlinku 1, 949 01 Nitra

---

# Bezpečnostné aspekty programovania v jazyku C

Predmet: KI/PJC/22 – Programovanie v jazyku C

Téma **Bezpečnostné aspekty programovania v jazyku C** sa venuje rozšíreniu základného programovania o identifikáciu a prevenciu častých bezpečnostných chýb pri práci s pamäťou, vstupom a procesmi. Študenti sa oboznámia s praktickými útokmi aj obrannými mechanizmami, ktoré zvyšujú odolnosť programov voči zneužitiu.

Mgr. František Forgáč, Phd.  
fforgac@ukf.sk

Tento dokument bol vypracovaný pre project Nitrianske kompetenčné centrum pre kybernetickú bezpečnosť (17R05-04-V01-00003).



Nitrianske kompetenčné  
centrum kybernetickej  
bezpečnosti



**PLÁN [OBNOVY]**

# Obsah

|          |                                                                   |           |
|----------|-------------------------------------------------------------------|-----------|
| <b>1</b> | <b>Úvod</b>                                                       | <b>1</b>  |
| <b>2</b> | <b>Skriptá</b>                                                    | <b>3</b>  |
| 2.1      | Úvod do bezpečnosti v jazyku C . . . . .                          | 3         |
| 2.1.1    | Historický kontext jazyka C . . . . .                             | 3         |
| 2.1.2    | Vlastnosti jazyka C . . . . .                                     | 3         |
| 2.1.3    | Bezpečnostné výzvy pri programovaní v jazyku C . . . . .          | 3         |
| 2.1.4    | Porovnanie s modernými jazykmi . . . . .                          | 4         |
| 2.1.5    | Prečo je C stále aktuálny? . . . . .                              | 4         |
| 2.2      | Najčastejšie chyby a ich dopady . . . . .                         | 4         |
| 2.2.1    | Prístup mimo hraníc poľa . . . . .                                | 4         |
| 2.2.2    | Použitie neinicializovaných premenných . . . . .                  | 5         |
| 2.2.3    | Nesprávne spracovanie reťazcov a formátovacích reťazcov . . . . . | 5         |
| 2.2.4    | Chyby v práci s dynamickou pamäťou . . . . .                      | 5         |
| 2.2.5    | Chyby pri overovaní vstupov od používateľa . . . . .              | 5         |
| 2.3      | Príklady zneužitia . . . . .                                      | 5         |
| 2.3.1    | Buffer overflow . . . . .                                         | 6         |
| 2.3.2    | Format string attack . . . . .                                    | 6         |
| 2.3.3    | Význam pre výučbu . . . . .                                       | 7         |
| 2.4      | Detekcia a prevencia . . . . .                                    | 7         |
| 2.4.1    | AddressSanitizer . . . . .                                        | 7         |
| 2.4.2    | Nástroje na statickú analýzu . . . . .                            | 8         |
| 2.4.3    | Bezpečné knižnice a náhrady za rizikové funkcie . . . . .         | 8         |
| 2.4.4    | Pracovná metodológia . . . . .                                    | 8         |
| 2.5      | Simulácia útoku a jeho mitigácie . . . . .                        | 9         |
| 2.5.1    | Východiskový scenár . . . . .                                     | 9         |
| 2.5.2    | Simulácia buffer overflow útoku . . . . .                         | 9         |
| 2.5.3    | Mitigácia . . . . .                                               | 9         |
| 2.5.4    | Reflexia . . . . .                                                | 9         |
| 2.6      | Záver . . . . .                                                   | 10        |
| <b>3</b> | <b>Projekty</b>                                                   | <b>11</b> |
| 3.1      | Textová adventúra . . . . .                                       | 11        |
| 3.1.1    | Požiadavky . . . . .                                              | 11        |
| 3.1.2    | Príklady použitia . . . . .                                       | 11        |
| 3.1.3    | Príklad priebehu . . . . .                                        | 12        |
| 3.2      | Simulátor Braillovej tlačiarne . . . . .                          | 12        |
| 3.2.1    | Požiadavky . . . . .                                              | 12        |

|          |                                                               |           |
|----------|---------------------------------------------------------------|-----------|
| 3.2.2    | Príklady použitia . . . . .                                   | 13        |
| 3.2.3    | Príklad výstupu . . . . .                                     | 13        |
| 3.2.4    | Pomôcky . . . . .                                             | 13        |
| 3.3      | Simulátor MHD . . . . .                                       | 14        |
| 3.3.1    | Požiadavky . . . . .                                          | 14        |
| 3.3.2    | Príklady použitia . . . . .                                   | 14        |
| 3.3.3    | Ukážka priebehu . . . . .                                     | 15        |
| 3.4      | Lode . . . . .                                                | 15        |
| 3.4.1    | Požiadavky . . . . .                                          | 15        |
| 3.4.2    | Príklady použitia . . . . .                                   | 15        |
| 3.4.3    | Ukážka priebehu . . . . .                                     | 16        |
| 3.5      | Huffmanovo kódovanie . . . . .                                | 16        |
| 3.5.1    | Požiadavky . . . . .                                          | 16        |
| 3.5.2    | Formát binárneho súboru . . . . .                             | 16        |
| 3.5.3    | Príklady použitia . . . . .                                   | 17        |
| 3.5.4    | Ukážka štatistiky . . . . .                                   | 17        |
| <b>4</b> | <b>Skúškové úlohy</b>                                         | <b>18</b> |
| 4.1      | Chybná alokácia pamäte . . . . .                              | 19        |
| 4.1.1    | Zadanie . . . . .                                             | 19        |
| 4.1.2    | Vstupný kód . . . . .                                         | 19        |
| 4.1.3    | Požadované riešenie . . . . .                                 | 19        |
| 4.2      | Pretečenie zásobníka - konštantný index . . . . .             | 20        |
| 4.2.1    | Zadanie . . . . .                                             | 20        |
| 4.2.2    | Vstupný kód . . . . .                                         | 20        |
| 4.2.3    | Požadované riešenie . . . . .                                 | 20        |
| 4.3      | Pretečenie zásobníka - index zo vstupu . . . . .              | 21        |
| 4.3.1    | Zadanie . . . . .                                             | 21        |
| 4.3.2    | Vstupný kód . . . . .                                         | 21        |
| 4.3.3    | Požadované riešenie . . . . .                                 | 21        |
| 4.4      | Únik pamäte pri calloc . . . . .                              | 22        |
| 4.4.1    | Zadanie . . . . .                                             | 22        |
| 4.4.2    | Vstupný kód . . . . .                                         | 22        |
| 4.4.3    | Požadované riešenie . . . . .                                 | 22        |
| 4.5      | Únik pamäte pri malloc . . . . .                              | 23        |
| 4.5.1    | Zadanie . . . . .                                             | 23        |
| 4.5.2    | Vstupný kód . . . . .                                         | 23        |
| 4.5.3    | Požadované riešenie . . . . .                                 | 23        |
| 4.6      | Únik pamäte pri realloc . . . . .                             | 24        |
| 4.6.1    | Zadanie . . . . .                                             | 24        |
| 4.6.2    | Vstupný kód . . . . .                                         | 24        |
| 4.6.3    | Požadované riešenie . . . . .                                 | 24        |
| 4.7      | Vyčerpanie systémových zdrojov . . . . .                      | 25        |
| 4.7.1    | Zadanie . . . . .                                             | 25        |
| 4.7.2    | Vstupný kód . . . . .                                         | 25        |
| 4.7.3    | Požadované riešenie . . . . .                                 | 25        |
| 4.8      | Vyčerpanie systémových zdrojov pri zápise do súboru . . . . . | 26        |
| 4.8.1    | Zadanie . . . . .                                             | 26        |

|        |                                                 |    |
|--------|-------------------------------------------------|----|
| 4.8.2  | Vstupný kód                                     | 26 |
| 4.8.3  | Požadované riešenie                             | 26 |
| 4.9    | Nesprávna kontrola návratovej hodnoty fgets     | 27 |
| 4.9.1  | Zadanie                                         | 27 |
| 4.9.2  | Vstupný kód                                     | 27 |
| 4.9.3  | Požadované riešenie                             | 27 |
| 4.10   | Nesprávna kontrola návratovej hodnoty remove    | 28 |
| 4.10.1 | Zadanie                                         | 28 |
| 4.10.2 | Vstupný kód                                     | 28 |
| 4.10.3 | Požadované riešenie                             | 28 |
| 4.11   | Deklarácia viacerých premenných so smerníkom    | 29 |
| 4.11.1 | Zadanie                                         | 29 |
| 4.11.2 | Vstupný kód                                     | 29 |
| 4.11.3 | Požadované riešenie                             | 29 |
| 4.12   | Alokácia cez neinicializovaný smerník           | 30 |
| 4.12.1 | Zadanie                                         | 30 |
| 4.12.2 | Vstupný kód                                     | 30 |
| 4.12.3 | Požadované riešenie                             | 30 |
| 4.13   | Const a typedef smerníka                        | 31 |
| 4.13.1 | Zadanie                                         | 31 |
| 4.13.2 | Vstupný kód                                     | 31 |
| 4.13.3 | Požadované riešenie                             | 31 |
| 4.14   | Zápis do reťazcového literálu                   | 32 |
| 4.14.1 | Zadanie                                         | 32 |
| 4.14.2 | Vstupný kód                                     | 32 |
| 4.14.3 | Požadované riešenie                             | 32 |
| 4.15   | Veľkosť externého poľa                          | 33 |
| 4.15.1 | Zadanie                                         | 33 |
| 4.15.2 | Vstupný kód                                     | 33 |
| 4.15.3 | Požadované riešenie                             | 33 |
| 4.16   | Nesprávna aritmetika smerníkov                  | 34 |
| 4.16.1 | Zadanie                                         | 34 |
| 4.16.2 | Vstupný kód                                     | 34 |
| 4.16.3 | Požadované riešenie                             | 34 |
| 4.17   | Zmena smerníka vo funkcii                       | 35 |
| 4.17.1 | Zadanie                                         | 35 |
| 4.17.2 | Vstupný kód                                     | 35 |
| 4.17.3 | Požadované riešenie                             | 35 |
| 4.18   | Priradenie do poľa                              | 36 |
| 4.18.1 | Zadanie                                         | 36 |
| 4.18.2 | Vstupný kód                                     | 36 |
| 4.18.3 | Požadované riešenie                             | 36 |
| 4.19   | Veľkosť poľa ako parameter funkcie              | 37 |
| 4.19.1 | Zadanie                                         | 37 |
| 4.19.2 | Vstupný kód                                     | 37 |
| 4.19.3 | Požadované riešenie                             | 37 |
| 4.20   | Zámena dvojrozmerného poľa a dvojitého smerníka | 38 |
| 4.20.1 | Zadanie                                         | 38 |

|        |                                                |    |
|--------|------------------------------------------------|----|
| 4.20.2 | Vstupný kód                                    | 38 |
| 4.20.3 | Požadované riešenie                            | 38 |
| 4.21   | Čítanie do nealokovaného reťazca               | 39 |
| 4.21.1 | Zadanie                                        | 39 |
| 4.21.2 | Vstupný kód                                    | 39 |
| 4.21.3 | Požadované riešenie                            | 39 |
| 4.22   | Spájanie reťazcov bez cieľového buffera        | 40 |
| 4.22.1 | Zadanie                                        | 40 |
| 4.22.2 | Vstupný kód                                    | 40 |
| 4.22.3 | Požadované riešenie                            | 40 |
| 4.23   | Uloženie viacerých riadkov do jedného buffera  | 41 |
| 4.23.1 | Zadanie                                        | 41 |
| 4.23.2 | Vstupný kód                                    | 41 |
| 4.23.3 | Požadované riešenie                            | 41 |
| 4.24   | Návrat lokálneho poľa z funkcie                | 42 |
| 4.24.1 | Zadanie                                        | 42 |
| 4.24.2 | Vstupný kód                                    | 42 |
| 4.24.3 | Požadované riešenie                            | 42 |
| 4.25   | Alokácia poľa bez sizeof prvku                 | 43 |
| 4.25.1 | Zadanie                                        | 43 |
| 4.25.2 | Vstupný kód                                    | 43 |
| 4.25.3 | Požadované riešenie                            | 43 |
| 4.26   | Použitie pamäte po free                        | 44 |
| 4.26.1 | Zadanie                                        | 44 |
| 4.26.2 | Vstupný kód                                    | 44 |
| 4.26.3 | Požadované riešenie                            | 44 |
| 4.27   | Smerník po free nie je automaticky NULL        | 45 |
| 4.27.1 | Zadanie                                        | 45 |
| 4.27.2 | Vstupný kód                                    | 45 |
| 4.27.3 | Požadované riešenie                            | 45 |
| 4.28   | Zistenie veľkosti alokovaného bloku cez sizeof | 46 |
| 4.28.1 | Zadanie                                        | 46 |
| 4.28.2 | Vstupný kód                                    | 46 |
| 4.28.3 | Požadované riešenie                            | 46 |
| 4.29   | Porovnávanie reťazcov operátorom ==            | 47 |
| 4.29.1 | Zadanie                                        | 47 |
| 4.29.2 | Vstupný kód                                    | 47 |
| 4.29.3 | Požadované riešenie                            | 47 |
| 4.30   | Pridanie znaku pomocou strcat                  | 48 |
| 4.30.1 | Zadanie                                        | 48 |
| 4.30.2 | Vstupný kód                                    | 48 |
| 4.30.3 | Požadované riešenie                            | 48 |
| 4.31   | Bonus: Úprava reťazcového literálu             | 49 |
| 4.31.1 | Zadanie                                        | 49 |
| 4.31.2 | Vstupný kód                                    | 49 |
| 4.31.3 | Požadované riešenie                            | 49 |
| 4.32   | Bonus: Nezarovnaný prístup do pamäte           | 50 |
| 4.32.1 | Zadanie                                        | 50 |

|          |                                                               |           |
|----------|---------------------------------------------------------------|-----------|
| 4.32.2   | Vstupný kód . . . . .                                         | 50        |
| 4.32.3   | Požadované riešenie . . . . .                                 | 50        |
| <b>5</b> | <b>Riešenia</b>                                               | <b>51</b> |
| 5.1      | Chybná alokácia pamäte . . . . .                              | 52        |
| 5.1.1    | Očakávaná odpoveď . . . . .                                   | 52        |
| 5.1.2    | Vysvetlenie chyby . . . . .                                   | 52        |
| 5.1.3    | Opravený kód . . . . .                                        | 52        |
| 5.2      | Pretečenie zásobníka - konštantný index . . . . .             | 53        |
| 5.2.1    | Očakávaná odpoveď . . . . .                                   | 53        |
| 5.2.2    | Vysvetlenie chyby . . . . .                                   | 53        |
| 5.2.3    | Opravený kód . . . . .                                        | 53        |
| 5.3      | Pretečenie zásobníka - index zo vstupu . . . . .              | 54        |
| 5.3.1    | Očakávaná odpoveď . . . . .                                   | 54        |
| 5.3.2    | Vysvetlenie chyby . . . . .                                   | 54        |
| 5.3.3    | Opravený kód . . . . .                                        | 54        |
| 5.4      | Únik pamäte pri calloc . . . . .                              | 55        |
| 5.4.1    | Očakávaná odpoveď . . . . .                                   | 55        |
| 5.4.2    | Vysvetlenie chyby . . . . .                                   | 55        |
| 5.4.3    | Opravený kód . . . . .                                        | 55        |
| 5.5      | Únik pamäte pri malloc . . . . .                              | 56        |
| 5.5.1    | Očakávaná odpoveď . . . . .                                   | 56        |
| 5.5.2    | Vysvetlenie chyby . . . . .                                   | 56        |
| 5.5.3    | Opravený kód . . . . .                                        | 56        |
| 5.6      | Únik pamäte pri realloc . . . . .                             | 57        |
| 5.6.1    | Očakávaná odpoveď . . . . .                                   | 57        |
| 5.6.2    | Vysvetlenie chyby . . . . .                                   | 57        |
| 5.6.3    | Opravený kód . . . . .                                        | 57        |
| 5.7      | Vyčerpanie systémových zdrojov . . . . .                      | 59        |
| 5.7.1    | Očakávaná odpoveď . . . . .                                   | 59        |
| 5.7.2    | Vysvetlenie chyby . . . . .                                   | 59        |
| 5.7.3    | Opravený kód . . . . .                                        | 59        |
| 5.8      | Vyčerpanie systémových zdrojov pri zápise do súboru . . . . . | 60        |
| 5.8.1    | Očakávaná odpoveď . . . . .                                   | 60        |
| 5.8.2    | Vysvetlenie chyby . . . . .                                   | 60        |
| 5.8.3    | Opravený kód . . . . .                                        | 60        |
| 5.9      | Nesprávna kontrola návratovej hodnoty fgets . . . . .         | 62        |
| 5.9.1    | Očakávaná odpoveď . . . . .                                   | 62        |
| 5.9.2    | Vysvetlenie chyby . . . . .                                   | 62        |
| 5.9.3    | Opravený kód . . . . .                                        | 62        |
| 5.10     | Nesprávna kontrola návratovej hodnoty remove . . . . .        | 63        |
| 5.10.1   | Očakávaná odpoveď . . . . .                                   | 63        |
| 5.10.2   | Vysvetlenie chyby . . . . .                                   | 63        |
| 5.10.3   | Opravený kód . . . . .                                        | 63        |
| 5.11     | Deklarácia viacerých premenných so smerníkom . . . . .        | 64        |
| 5.11.1   | Očakávaná odpoveď . . . . .                                   | 64        |
| 5.11.2   | Vysvetlenie chyby . . . . .                                   | 64        |
| 5.11.3   | Opravený kód . . . . .                                        | 64        |

|        |                                                           |    |
|--------|-----------------------------------------------------------|----|
| 5.12   | Alokácia cez neinicializovaný smerník . . . . .           | 65 |
| 5.12.1 | Očakávaná odpoveď . . . . .                               | 65 |
| 5.12.2 | Vysvetlenie chyby . . . . .                               | 65 |
| 5.12.3 | Opravený kód . . . . .                                    | 65 |
| 5.13   | Const a typedef smerníka . . . . .                        | 66 |
| 5.13.1 | Očakávaná odpoveď . . . . .                               | 66 |
| 5.13.2 | Vysvetlenie chyby . . . . .                               | 66 |
| 5.13.3 | Opravený kód . . . . .                                    | 66 |
| 5.14   | Zápis do reťazcového literálu . . . . .                   | 67 |
| 5.14.1 | Očakávaná odpoveď . . . . .                               | 67 |
| 5.14.2 | Vysvetlenie chyby . . . . .                               | 67 |
| 5.14.3 | Opravený kód . . . . .                                    | 67 |
| 5.15   | Veľkosť externého poľa . . . . .                          | 68 |
| 5.15.1 | Očakávaná odpoveď . . . . .                               | 68 |
| 5.15.2 | Vysvetlenie chyby . . . . .                               | 68 |
| 5.15.3 | Opravený kód . . . . .                                    | 68 |
| 5.16   | Nesprávna aritmetika smerníkov . . . . .                  | 69 |
| 5.16.1 | Očakávaná odpoveď . . . . .                               | 69 |
| 5.16.2 | Vysvetlenie chyby . . . . .                               | 69 |
| 5.16.3 | Opravený kód . . . . .                                    | 69 |
| 5.17   | Zmena smerníka vo funkcii . . . . .                       | 70 |
| 5.17.1 | Očakávaná odpoveď . . . . .                               | 70 |
| 5.17.2 | Vysvetlenie chyby . . . . .                               | 70 |
| 5.17.3 | Opravený kód . . . . .                                    | 70 |
| 5.18   | Priradenie do poľa . . . . .                              | 71 |
| 5.18.1 | Očakávaná odpoveď . . . . .                               | 71 |
| 5.18.2 | Vysvetlenie chyby . . . . .                               | 71 |
| 5.18.3 | Opravený kód . . . . .                                    | 71 |
| 5.19   | Veľkosť poľa ako parameter funkcie . . . . .              | 72 |
| 5.19.1 | Očakávaná odpoveď . . . . .                               | 72 |
| 5.19.2 | Vysvetlenie chyby . . . . .                               | 72 |
| 5.19.3 | Opravený kód . . . . .                                    | 72 |
| 5.20   | Zámena dvojrozmerného poľa a dvojitého smerníka . . . . . | 73 |
| 5.20.1 | Očakávaná odpoveď . . . . .                               | 73 |
| 5.20.2 | Vysvetlenie chyby . . . . .                               | 73 |
| 5.20.3 | Opravený kód . . . . .                                    | 73 |
| 5.21   | Čítanie do nealokovaného reťazca . . . . .                | 74 |
| 5.21.1 | Očakávaná odpoveď . . . . .                               | 74 |
| 5.21.2 | Vysvetlenie chyby . . . . .                               | 74 |
| 5.21.3 | Opravený kód . . . . .                                    | 74 |
| 5.22   | Spájanie reťazcov bez cieľového buffera . . . . .         | 75 |
| 5.22.1 | Očakávaná odpoveď . . . . .                               | 75 |
| 5.22.2 | Vysvetlenie chyby . . . . .                               | 75 |
| 5.22.3 | Opravený kód . . . . .                                    | 75 |
| 5.23   | Uloženie viacerých riadkov do jedného buffera . . . . .   | 76 |
| 5.23.1 | Očakávaná odpoveď . . . . .                               | 76 |
| 5.23.2 | Vysvetlenie chyby . . . . .                               | 76 |
| 5.23.3 | Opravený kód . . . . .                                    | 76 |



|        |                                                          |    |
|--------|----------------------------------------------------------|----|
| 5.24   | Návrat lokálneho poľa z funkcie . . . . .                | 77 |
| 5.24.1 | Očakávaná odpoveď . . . . .                              | 77 |
| 5.24.2 | Vysvetlenie chyby . . . . .                              | 77 |
| 5.24.3 | Opravený kód . . . . .                                   | 77 |
| 5.25   | Alokácia poľa bez sizeof prvku . . . . .                 | 78 |
| 5.25.1 | Očakávaná odpoveď . . . . .                              | 78 |
| 5.25.2 | Vysvetlenie chyby . . . . .                              | 78 |
| 5.25.3 | Opravený kód . . . . .                                   | 78 |
| 5.26   | Použitie pamäte po free . . . . .                        | 79 |
| 5.26.1 | Očakávaná odpoveď . . . . .                              | 79 |
| 5.26.2 | Vysvetlenie chyby . . . . .                              | 79 |
| 5.26.3 | Opravený kód . . . . .                                   | 79 |
| 5.27   | Smerník po free nie je automaticky NULL . . . . .        | 80 |
| 5.27.1 | Očakávaná odpoveď . . . . .                              | 80 |
| 5.27.2 | Vysvetlenie chyby . . . . .                              | 80 |
| 5.27.3 | Opravený kód . . . . .                                   | 80 |
| 5.28   | Zistenie veľkosti alokovaného bloku cez sizeof . . . . . | 81 |
| 5.28.1 | Očakávaná odpoveď . . . . .                              | 81 |
| 5.28.2 | Vysvetlenie chyby . . . . .                              | 81 |
| 5.28.3 | Opravený kód . . . . .                                   | 81 |
| 5.29   | Porovnávanie refazcov operátorom == . . . . .            | 82 |
| 5.29.1 | Očakávaná odpoveď . . . . .                              | 82 |
| 5.29.2 | Vysvetlenie chyby . . . . .                              | 82 |
| 5.29.3 | Opravený kód . . . . .                                   | 82 |
| 5.30   | Pridanie znaku pomocou strcat . . . . .                  | 83 |
| 5.30.1 | Očakávaná odpoveď . . . . .                              | 83 |
| 5.30.2 | Vysvetlenie chyby . . . . .                              | 83 |
| 5.30.3 | Opravený kód . . . . .                                   | 83 |
| 5.31   | Bonus: Úprava refazcového literálu . . . . .             | 84 |
| 5.31.1 | Očakávaná odpoveď . . . . .                              | 84 |
| 5.31.2 | Vysvetlenie chyby . . . . .                              | 84 |
| 5.31.3 | Opravený kód . . . . .                                   | 84 |
| 5.32   | Bonus: Nezarovnaný prístup do pamäte . . . . .           | 85 |
| 5.32.1 | Očakávaná odpoveď . . . . .                              | 85 |
| 5.32.2 | Vysvetlenie chyby . . . . .                              | 85 |
| 5.32.3 | Opravený kód . . . . .                                   | 85 |

# 1 Úvod

Programovanie v jazyku C patrí medzi základné stavebné kamene systémového programovania. Jazyk C je prítomný v operačných systémoch, sieťových službách, knižniciach, embedded zariadeniach aj v nástrojoch, ktoré denne používajú vývojári. Jeho sila spočíva v jednoduchom modeli jazyka, efektívnom preklade a priamej kontrole nad pamäťou. Práve tieto vlastnosti však znamenajú, že zodpovednosť za bezpečné správanie programu nesie vo veľkej miere programátor.

Tento materiál je pripravený ako učiteľská verzia k predmetu Programovanie v jazyku C so zameraním na bezpečnostné aspekty práce v jazyku C. Nejde iba o doplnkový prehľad chýb, ale o súbor výkladových častí, semestrálnych projektov, skúškových úloh a riešení, ktoré majú študentom ukázať, ako sa bežné programátorské rozhodnutia premietajú do stability a bezpečnosti softvéru. Dôraz je kladený najmä na smerníky, polia, dynamickú alokáciu, reťazce, inicializáciu premenných, návratové hodnoty knižničných funkcií a prácu s binárnymi dátami.

Jazyk C neposkytuje automatickú ochranu pred mnohými chybami, ktoré moderné vyššie jazyky zachytávajú priamo v behovom prostredí alebo v typovom systéme. Prístup mimo hraníc poľa, použitie neinicializovaného smerníka, práca s pamäťou po jej uvoľnení, nesprávna veľkosť alokácie alebo zápis do reťazcového literálu môžu viesť k nedefinovanému správaniu. Takéto správanie sa nemusí prejaviť okamžite. Program môže na jednom počítači fungovať, na inom spadnúť a v produkčnom prostredí vytvoriť zraniteľnosť, ktorú je možné zneužiť.

Pre výučbu je preto dôležité, aby študenti nebrali chyby v C iba ako syntaktické alebo technické omyly. Veľká časť bezpečnostných incidentov začína presne pri drobných predpokladoch: že pole pozná svoju veľkosť aj vo funkcii, že `sizeof` na smerníku vráti veľkosť alokovaného bloku, že `free` nastaví smerník na NULL, že `strcat` vytvorí nový reťazec alebo že návratová hodnota funkcie sa dá kontrolovať odhadom. Cieľom materiálu je tieto predpoklady rozobrať a nahradiť ich presným mentálnym modelom jazyka.

Skriptá najprv vysvetľujú, prečo je C stále aktuálny a prečo je zároveň bezpečnostne náročný. Nasledujú kapitoly o častých chybách a ich dopadoch, príkladoch zneužitia, detekcii problémov a praktickej simulácii útoku a mitigácie. Projekty rozširujú predmet o väčšie samostatné zadania, v ktorých študenti pracujú s dynamickými dátovými štruktúrami, binárnymi súbormi, parsermi, simuláciou a vlastnými algoritmami. Skúškové úlohy sú formulované ako krátke analýzy kódu alebo konceptuálne otázky, pri ktorých má študent identifikovať problém, vysvetliť jeho dôsledok a navrhnúť opravu.

Učiteľská verzia obsahuje aj riešenia. Tie nie sú určené iba ako kontrola správnosti, ale aj ako pomôcka pri hodnotení odpovedí. Dobrá odpoveď by nemala skončiť pri tvrdení, že program „môže spadnúť“. Mala by pomenovať konkrétnu príčinu, napríklad zápis mimo rozsahu, zámenu poľa a smerníka, nesprávne použitie návratovej hodnoty alebo prácu s pamäťou po skončení jej životnosti. Rovnako dôležité je navrhnúť opravu, ktorá je primeraná pôvodnému zámeru programu.

Materiál je koncipovaný prakticky. Študenti majú čítať kód, spúšťať ho v kontrolovanom prostredí, používať prekladačové varovania, sanitizéry a statickú analýzu. Pri riešeníach sa očakáva dôsledná práca s návratovými hodnotami, kontrolou hraníc, veľkosťami bufferov a životnosťou objektov. Takýto prístup vedie k návyku, že bezpečnosť nie je samostatná fáza na konci vývoja, ale súčasť každého rozhodnutia pri návrhu a implementácii programu.

Výsledkom predmetu má byť schopnosť písať čitateľný, robustný a bezpečnejší C kód. Rovnako dôležitá je schopnosť rozpoznať rizikový kód v existujúcich programoch. Študent, ktorý rozumie týmto princípom, dokáže lepšie používať aj moderné jazyky a nástroje, pretože vie, aké problémy sa pod ich abstrakciami skrývajú. C tak v tomto predmete neslúži iba ako programovací jazyk, ale aj ako nástroj na pochopenie pamäťového modelu, systémového správania a zodpovednosti vývojára za dôsledky svojho kódu.

## 2 Skriptá

### 2.1 Úvod do bezpečnosti v jazyku C

#### 2.1.1 Historický kontext jazyka C

Programovací jazyk C vznikol na začiatku 70. rokov 20. storočia ako nástroj na vývoj operačného systému Unix. Jeho autormi sú Dennis Ritchie a Brian Kernighan, ktorí sa zamerali na vytvorenie univerzálneho jazyka s efektívnym prístupom k systémovým zdrojom. Jazyk C umožnil prepisovať pôvodne assemblerové programy do čitateľnejšej, no stále veľmi efektívnej podoby.

Vďaka svojej výkonnosti, jednoduchej syntaxi a blízkosti k hardvéru sa C stal štandardom v oblasti systémového programovania. Mnohé kritické systémy, ako sú operačné systémy, sieťové zariadenia, ovládače alebo firmvér mikrokontrolérov, sú napísané v jazyku C alebo obsahujú jeho časti.

#### 2.1.2 Vlastnosti jazyka C

Jazyk C poskytuje priamu manipuláciu s pamäťou. Tá je výhodná pri vývoji výkonovo kritických aplikácií, ale zároveň predstavuje významné bezpečnostné riziko. Na rozdiel od viacerých moderných jazykov C:

- neposkytuje automatickú správu pamäti,
- neobsahuje zabudovanú kontrolu rozsahu polí,
- umožňuje prácu s ukazovateľmi vrátane aritmetiky ukazovateľov.

C sa často označuje ako nízkoúrovňový jazyk s vysokou úrovňovou syntaxou. Kombinuje čitateľnosť, štruktúry a funkcie s priamym prístupom k hardvéru. Táto flexibilita si však vyžaduje zodpovedného programátora, ktorý rozumie možným chybám a vie im predchádzať.

#### 2.1.3 Bezpečnostné výzvy pri programovaní v jazyku C

Jazyk C kladie dôraz na efektivitu a výkon, nie na predvolenú bezpečnosť. Vela ochranných mechanizmov, ktoré sú bežné v moderných jazykoch, sú v C voliteľné alebo vôbec neexistujú. Medzi najčastejšie bezpečnostné problémy patria:

- dereferencovanie NULL ukazovateľov, keď sa program pokúsi pristupovať k neplatnej adrese,
- prístup mimo hraníc poľa, ktorý môže prepísať iné premenné alebo návratovú adresu,
- nesprávne spravovanie dynamickej pamäti, napríklad pamäťové úniky, použitie uvoľnenej pamäte alebo opakované uvoľnenie tej istej oblasti,

- zlá práca s reťazcami, napríklad použitie `strcpy` bez overenia dĺžky vstupu.

Tieto chyby často nevznikajú pre zlý úmysel, ale pre nevedomosť alebo nedbanlivosť. Preto je dôležité, aby sa študenti od začiatku učili bezpečnému štýlu programovania, najmä v jazyku C, kde neexistujú predvolené ochrany.

### 2.1.4 Porovnanie s modernými jazykmi

Moderné jazyky ako Rust, Go, Python alebo Java ponúkajú vstavané bezpečnostné mechanizmy. Rust napríklad striktne zabráňuje použitiu neplatných ukazovateľov cez mechanizmy vlastníctva a požičiavania. Python zase skrýva prácu s pamäťou, čím eliminuje veľkú časť pamäťových chýb, hoci za cenu výkonu.

Jazyk C je preto dobrým nástrojom na výučbu princípov správy pamäti, efektivity a bezpečnostných nástrah. Študenti sa učia myslieť ako systémoví vývojári a chápať, čo môže zlyhať, keď program nemá automatické ochranné mechanizmy.

### 2.1.5 Prečo je C stále aktuálny?

Napriek svojmu veku zostáva jazyk C dôležitý v mnohých oblastiach:

- vstavané zariadenia,
- vývoj operačných systémov,
- kritické aplikácie v leteectve alebo zdravotníctve,
- kompilačné nástroje a interpretery,
- systémové knižnice a nízkoúrovňové API.

Znalosť C je preto dôležitým základom pre vývojárov, ktorí sa chcú venovať systémovému programovaniu alebo bezpečnosti softvéru. Študenti sa v úvodnej časti učia, čo robí jazyk C výnimočným a aké riziká predstavuje z hľadiska kybernetickej bezpečnosti.

Programátor má v C takmer úplnú kontrolu nad správaním programu. Táto kontrola je výhodou aj rizikom. Bezpečný kód preto nevzniká automaticky, ale ako výsledok dôsledného návrhu, kontroly vstupov a zodpovednej práce s pamäťou.

## 2.2 Najčastejšie chyby a ich dopady

Pri programovaní v jazyku C je nevyhnutné pochopiť, že veľká časť bezpečnostných incidentov nevzniká kvôli úmyselnému útoku zvonka, ale kvôli chybnému návrhu alebo implementácii softvéru. Jazyk C neposkytuje automatickú ochranu pred typickými chybami, ktoré môže programátor urobiť. Preto je dôležité poznať najčastejšie typy chýb a ich dopad na bezpečnosť a spoľahlivosť aplikácií.

### 2.2.1 Prístup mimo hraníc poľa

Jednou z najčastejších chýb v jazyku C je prístup k pamäti mimo vyhradeného rozsahu poľa. Jazyk C nekontroluje, či sa pri čítaní alebo zápise do poľa neprekračujú jeho hranice.

Táto chyba môže mať rôzne následky: od narušenia internej logiky programu až po prepísanie kritických častí pamäte, napríklad návratových adries funkcií. V závažných prípadoch môže viesť až k vykonaniu útočnického kódu.

### 2.2.2 Použitie neinicializovaných premenných

C neobsahuje ochranu proti používaniu premenných, ktorým nebola pred použitím priradená hodnota. Použitie takejto premennej vedie k nedefinovanému správaniu programu, ktoré môže byť veľmi ťažko odhaliteľné pri testovaní.

Neinicializované premenné môžu ovplyvniť rozhodovanie programu, logiku výpočtov alebo obsahovať zvyšky starých dát z pamäte. V horších prípadoch môžu viesť k úniku citlivých údajov.

### 2.2.3 Nesprávne spracovanie reťazcov a formátovacích reťazcov

V jazyku C sú reťazce reprezentované ako polia znakov ukončené nulovým bajtom. Väčšina funkcií na spracovanie reťazcov predpokladá, že vstupy sú korektne ukončené a majú správnu dĺžku. V praxi to však nemusí platiť.

Nesprávne spracovanie reťazcov môže spôsobiť čítanie za koniec reťazca, prepísanie pamäte alebo vykonanie škodlivého kódu. Zvlášť nebezpečné je používanie formátovacích reťazcov, kde neoverený vstup od používateľa môže meniť správanie výstupných funkcií, čítať obsah pamäte alebo prepísať dáta na kritických miestach.

### 2.2.4 Chyby v práci s dynamickou pamäťou

Efektívna práca s dynamickou pamäťou je jednou z najväčších výhod aj nevýhod jazyka C. Vývojár má úplnú kontrolu nad alokovaním a uvoľňovaním pamäťových blokov, no zároveň nesie plnú zodpovednosť za ich správne použitie.

Medzi časté chyby patria:

- neuvoľnenie pamäte po skončení jej použitia,
- opakované uvoľnenie toho istého bloku,
- prístup k pamäti po jej uvoľnení.

Tieto chyby môžu postupne degradovať výkon aplikácie, viesť k jej zlyhaniu alebo byť zneužitú na vzdialené vykonanie škodlivého kódu.

### 2.2.5 Chyby pri overovaní vstupov od používateľa

Jednou z najzásadnejších oblastí, kde dochádza k bezpečnostným zlyháním, je nedostatočné alebo žiadne overovanie vstupov od používateľa. Ak program slepo dôveruje vstupu z vonkajšieho prostredia, môže dôjsť k zahlteniu systému, pretečeniu vstupných bufferov alebo logickým chybám.

Ak je neoverený vstup použitý ako súčasť systémového volania alebo databázového dopytu, hrozia aj injekčné útoky. Tie môžu obísť bezpečnostnú logiku aplikácie a viesť k vážnemu narušeniu systému.

## 2.3 Príklady zneužitia

Zraniteľnosti v softvéri napísanom v jazyku C často nie sú spôsobené neznámymi technikami, ale opakovaným zneužívaním známych chýb v spracovaní pamäte a vstupov. Medzi

najznámejšie patria buffer overflow a format string attack. Obe kategórie útokov vyplývajú z flexibility jazyka C a z nedostatku vstavaných bezpečnostných mechanizmov.

Cieľom tejto časti je predstaviť mechanizmus fungovania týchto útokov, spôsob, akým narúšajú normálne správanie aplikácií, a ich možné následky v praxi.

### 2.3.1 Buffer overflow

Buffer overflow predstavuje zraniteľnosť, pri ktorej program umožní zapísanie dát do poľa bez overenia, či sa vstup zmestí do prideleného pamäťového priestoru. Ak vstup presiahne kapacitu poľa, začne prepisovať príslušné oblasti pamäte.

Ak sa buffer nachádza na zásobníku, môžu byť prepísané lokálne premenné, štruktúry riadiaceho toku alebo návratová adresa funkcie.

#### Mechanizmus zneužitia

Útočník dokáže cielene manipulovať obsahom pretečeného vstupu tak, aby prepísal návratovú adresu funkcie. Tá sa pri ukončení funkcie použije na skok do inej pamätej oblasti, napríklad do oblasti, kde sa nachádza útočníkov kód. Tým môže dôjsť k prevzatiu kontroly nad vykonávaním programu.

#### Historický význam

Buffer overflow patrí medzi najstaršie dokumentované softvérové zraniteľnosti a stal sa základom mnohých známych útokov. Zraniteľnosti tohto typu sú dodnes relevantné, hoci moderné systémy ich často zmierňujú technikami ako stack canary, DEP alebo ASLR.

#### Riziká a dopady

- spustenie ľubovoľného kódu útočníka,
- pád alebo nestabilita systému,
- krádež údajov,
- prevzatie systému,
- šírenie škodlivého kódu.

### 2.3.2 Format string attack

Útok formátovacím reťazcom je menej známy, no veľmi nebezpečný spôsob zneužitia vstupov v jazyku C. Zraniteľnosť vzniká vtedy, keď programátor vloží používateľský vstup priamo ako formátovací reťazec do funkcií určených na formátovaný výstup, napríklad `printf`, `fprintf` alebo `snprintf`.

Namiesto toho, aby sa vstup spracoval ako obyčajný text, môže byť interpretovaný ako sekvencia formátovacích inštrukcií, napríklad `%x` alebo `%n`.

#### Mechanizmus zneužitia

Ak útočník kontroluje formátovací reťazec, môže čítať obsah zásobníka, meniť premenné v pamäti alebo zapisovať konkrétne hodnoty na určené adresy. Ide o nepriamy spôsob

manipulácie s pamäťou, ktorý nevyžaduje pretečenie poľa, ale zneužíva dôveru aplikácie k vstupu používateľa.

### Kde je útok rizikový

Tento typ útoku je zvlášť nebezpečný v aplikáciách, ktoré prijímajú vstupy od používateľa zo štandardného vstupu, siete alebo logovacích rozhraní. Riziko vzniká vtedy, keď sa používateľské dáta pred spracovaním formátovacou funkciou nezabezpečia.

### Riziká a dopady

- čítanie alebo modifikácia citlivých údajov v pamäti,
- prepísanie hodnoty v pamäti,
- získanie informácií o štruktúre zásobníka,
- potenciál na vzdialené vykonanie kódu.

### 2.3.3 Význam pre výučbu

Buffer overflow a format string attack sú základné kategórie zneužitia zraniteľností v jazyku C. Ich spoločným znakom je nesprávne zaobchádzanie s pamäťou alebo vstupom, ktoré jazyk C automaticky nechráni.

V praxi boli podobné útoky zneužitie napríklad v serverových službách, starších systémových knižniciach alebo embedded zariadeniach. V rámci výučby sa analyzujú a simulujú iba v izolovanom prostredí, aby si študenti osvojili schopnosť identifikovať riziká v kóde a navrhovať obranné opatrenia.

## 2.4 Detekcia a prevencia

Bezpečné programovanie nie je len otázkou správneho návrhu algoritmov, ale aj systematického odhaľovania chýb počas vývoja. Jazyk C neobsahuje vstavané ochranné mechanizmy, preto je nevyhnutné používať externé nástroje a metodiky, ktoré umožňujú odhaliť zraniteľnosti ešte pred nasadením softvéru.

Táto časť sa venuje nástrojom a prístupom, ktoré patria medzi štandardné súčasti profesionálneho vývoja softvéru, najmä v oblastiach so zvýšenými požiadavkami na bezpečnosť.

### 2.4.1 AddressSanitizer

AddressSanitizer je nástroj určený na dynamickú detekciu pamäťových chýb počas behu programu. Je dostupný v hlavných kompilátoroch ako `clang` a `gcc` a aktivuje sa kompilátorovým prepínačom.

Počas spúšťania programu s AddressSanitizerom sa sleduje každý pokus o prístup k pamäti a vyhodnocuje sa, či je tento prístup platný.

AddressSanitizer dokáže detegovať najmä:

- buffer overflow na zásobníku alebo halde,
- použitie pamäte po jej uvoľnení,
- čítanie alebo zápis do neplatnej pamäte,



- pretečenie globálnych premenných a štruktúr.

Počas výučby sa študenti učia, ako tento nástroj aktivovať, spúšťať a interpretovať jeho výstupy. AddressSanitizer je vhodný na praktické cvičenia, pretože poskytuje okamžitú spätnú väzbu o stave pamäte programu.

## 2.4.2 Nástroje na statickú analýzu

Statická analýza je proces, pri ktorom sa zdrojový kód analyzuje bez jeho spúšťania. Cieľom je identifikovať potenciálne chyby a nekonzistencie v logike programu, ktoré môžu viesť k bezpečnostným rizikám, nepredvídateľnému správaniu alebo porušeniu pravidiel dobrého štýlu programovania.

Využívajú sa napríklad:

- **Cppcheck**, ktorý sa zameriava na detekciu logických chýb, zbytočných operácií, únikov pamäte a neplatných prístupov,
- **Clang Static Analyzer**, ktorý analyzuje kód na úrovni alokácie pamäte, vetvenia logiky a manipulácie s ukazovateľmi.

Statické analyzátory nevyžadujú spustenie programu, preto sa dajú začleniť priamo do vývojového cyklu, napríklad ako súčasť automatizovaných testov alebo CI/CD procesov. V rámci kurzu sa študenti učia, aké typy chýb tieto nástroje nachádzajú, aké majú obmedzenia a ako ich používať pri každodennej práci.

## 2.4.3 Bezpečné knižnice a náhrady za rizikové funkcie

Veľká časť historických zraniteľností v programoch napísaných v jazyku C vznikla kvôli používaniu nebezpečných štandardných funkcií, ktoré neobsahujú kontrolné mechanizmy. Príkladmi sú **gets**, **strcpy** alebo **sprintf**, ktoré pracujú s reťazcami bez spoľahlivej kontroly veľkosti vstupu alebo cieľového bufferu.

Moderný prístup odporúča tieto funkcie nahradiť:

- bezpečnejšími variantmi dostupnými v štandardných knižniciach, napríklad **fgets** namiesto **gets** alebo **snprintf** namiesto **sprintf**,
- knižnicami typu Safe C Libraries, ktoré poskytujú funkcie so vstavanou ochranou pred pretečením, neplatnými ukazovateľmi alebo inými typickými chybami.

Tieto knižnice podporujú obranné programovanie a umožňujú písať kód, ktorý je robustnejší voči neštandardnému správaniu používateľa alebo externých systémov.

## 2.4.4 Pracovná metodológia

Nástroje na detekciu chýb majú zásadný význam, no ich prínos závisí najmä od pracovných návykov. Medzi takéto návyky patrí pravidelné spúšťanie analýz pred nasadením softvéru, vyhýbanie sa nebezpečným konštrukciám už vo fáze návrhu a písanie testovacích prípadov pre hraničné aj neštandardné situácie.

Bezpečnosť sa má vnímať ako súčasť celkovej kvality softvérového riešenia, nie ako dodatočný doplnok. Cieľom tejto kapitoly je ukázať, že bezpečné programovanie v jazyku C je kombináciou technických vedomostí, správneho pracovného postupu, dôslednosti a disciplíny.

## 2.5 Simulácia útoku a jeho mitigácie

Táto praktická časť výučby ponúka študentom možnosť prejsť si celý cyklus práce so zraniteľným softvérom: od identifikácie problému, cez simuláciu útoku až po aplikáciu obranných riešení.

Dôležitým aspektom modulu je práca v bezpečnom a izolovanom prostredí. To umožňuje testovať a analyzovať nebezpečné scenáre bez ohrozenia reálneho systému.

### 2.5.1 Výhodiskový scenár

Na začiatku aktivity sa študenti zoznámia so zraniteľným programom, ktorý obsahuje bezpečnostné nedostatky v práci s pamäťou a vstupmi. Typickým scenárom je aplikácia, ktorá číta používateľský vstup do pevne alokovaného poľa bez overenia dĺžky.

Na tomto základe prebehnú:

- úvodná analýza možných dôsledkov,
- identifikácia zraniteľného miesta,
- návrh hypotézy útoku.

### 2.5.2 Simulácia buffer overflow útoku

Konkrétnym príkladom aktivity je simulácia buffer overflow útoku na jednoduchý program, ktorý spracúva textový vstup používateľa. Študenti sa pokúsia spôsobiť nepredvídateľné správanie aplikácie, napríklad pád programu alebo narušenie vnútornej logiky, pomocou vstupov, ktoré presahujú kapacitu dostupnej pamäte.

V následnom kroku použijú AddressSanitizer na analýzu a detekciu konkrétneho miesta, kde k chybe dochádza.

### 2.5.3 Mitigácia

Po ukončení simulácie útoku nasleduje fáza mitigácie. Študenti upravia kód tak, aby sa riziko zneužitia eliminovalo.

Súčasťou riešenia je:

- použitie bezpečnejších knižničných funkcií,
- pridanie kontrol dĺžky vstupu,
- revízia spôsobu alokácie pamäte,
- opätovné overenie správania programu nástrojmi na detekciu chýb.

Cieľom nie je len opraviť konkrétnu chybu, ale navrhnúť robustnejšie riešenie, ktoré je dlhodobo odolné voči podobným typom zraniteľností.

### 2.5.4 Reflexia

Záverečnou fázou aktivity je spoločná diskusia. Študenti reflektujú, ako odhalili chybu, aké boli dôsledky zanedbania bezpečnostného overenia a ktoré opatrenia viedli k jej odstráneniu.

Táto časť výučby rozvíja praktické schopnosti v oblasti analýzy rizík, implementácie ochranných opatrení a vnímania bezpečnosti ako prirodzenej súčasti vývoja softvéru. Zároveň poskytuje vhlad do uvažovania útočníka, čo je dôležitý predpoklad pre rozvoj defenzívneho myslenia.

## 2.6 Záver

Výučbová jednotka zameraná na bezpečnostné aspekty programovania v jazyku C rozširuje tradičné chápanie programovania o bezpečnostné uvedomenie a obranné myslenie. Prepája technické zručnosti s bezpečnostnou gramotnosťou a pripravuje študentov na vývoj softvéru, kde bezpečnosť nie je voliteľným doplnkom, ale základnou požiadavkou.

Počas výučby študenti prechádzajú od pochopenia zraniteľnosti jazyka C, cez analýzu najčastejších chýb a ich dopadov, až po praktické osvojenie techník prevencie a detekcie. Učia sa navrhovať softvér tak, aby sa minimalizovalo riziko vzniku slabín ešte pred tým, než dôjde k ich zneužitiu.

Kľúčové výstupy modulu sú:

- teoretická znalosť bezpečnostných princípov viazaných na nízkoúrovňové programovanie v jazyku C,
- praktická skúsenosť so zraniteľným kódom,
- schopnosť odhaliť chyby a odstrániť ich pomocou overených nástrojov,
- zručnosti v používaní AddressSanitizeru a statických analyzátorov,
- pochopenie významu preventívnych opatrení, kontroly vstupov a obranných programovacích techník.

Študenti zároveň získavajú schopnosť vnímať vývoj softvéru z pohľadu potenciálneho útočníka. Táto perspektíva je dôležitá pri budovaní obranných mechanizmov. Bezpečný kód nie je výsledkom náhody, ale systematického a dôsledného prístupu, ktorý kombinuje teóriu s praxou.

Pre predmet Programovanie v jazyku C tento blok prináša dôležité doplnenie. Zatiaľ čo základné časti predmetu kladú dôraz na syntax, štruktúry a logiku programovania, bezpečnostný modul posilňuje kvalitatívny rozmer výučby tým, že upozorňuje na zodpovednosť vývojára za dopady jeho kódu.

Výsledkom je, že študent odchádza z predmetu nielen s technickými znalosťami, ale aj s uvedomením si rizík, ktoré nesie jeho práca. Vie, že aj malá chyba v kóde môže viesť k vážnym dôsledkom a že úlohou zodpovedného vývojára je týmto chybám predchádzať.

## 3 Projekty

### 3.1 Textová adventúra

Naprogramujte v jazyku C adventúru s textovým rozhraním, v ktorej sa hráč pohybuje po mape s miestnosťami.

Niektoré miestnosti sú prázdne, v iných môže byť NPC postava alebo užitočný predmet, napríklad zbraň, kľúč alebo indícia. Predmet sa môže dať zobrať bez boja, po vyhratom boji alebo po správnej odpovedi na otázku.

Aby hráč splnil cieľ hry, musí na mape nájsť príslušnú miestnosť a použiť predmety alebo indície, ktoré cestou získal. Hra môže skončiť aj neúspechom hráča.

Hráč s hrou komunikuje výhradne textovo. Zadáva príkazy a číta informácie o tom, kde sa nachádza, čo vidí okolo seba, koho stretol, či vyhral súboj a aké predmety má v inventári. Hra mapu sveta nezobrazuje; hráč ju postupne skúma.

#### 3.1.1 Požiadavky

- Pamäť pre údajovú štruktúru reprezentujúcu mapu sveta musí byť alokovaná dynamicky.
- Na mape musí existovať aspoň 5 rôznych miestností, ktoré treba navštíviť.
- Aspoň v jednej miestnosti musí úspech hráča závisieť od náhody.
- Hra sa musí spúšťať z príkazového riadku.
- Stav hry sa má pri ukončení ukladať do binárneho súboru s vhodnou hlavičkou a dátovou časťou.
- K rozohriatej hre sa má dať vrátiť obnovením stavu podľa mena hráča.
- V príbehu hry má byť viditeľný tvorivý prístup.

Všetky časti riešenia implementujte samostatne, bez použitia externých knižníc. Dbajte na efektívne uloženie a spracovanie dát, vhodné rozdelenie kódu do modulov a funkcií a riadne ošetrovanie chybových stavov.

#### 3.1.2 Príklady použitia

```
adventura -n Miso
```

Program vytvorí hru pre nového hráča **Miso**. Pri skončení hry sa stav uloží do súboru **Miso.dat**.

```
adventura -l Miso
```

Program načíta uložený stav zo súboru `Miso.dat` a pokračuje od posledného uloženia.

### 3.1.3 Príklad priebehu

```
Vitaj v hre. Nachadzas sa na zamku s mnohymi komnatami.
V kazdej je 4 dveri orientovanych na jednotlivé svetové strany.
V jednej z nich je ukryty carovny prsten.
Na odomknutie komnaty s prstenom potrebujes kluc.
Zoznam prikazov: sever, juh, vychod, zapad, info, bojovat, zobrat <predmet>, pouzit <predmet>
> info
Si v prazdnej komnate s obrazmi na stenach.
Pocet zivotov: 3
Inventar: prazdny
> sever
Si v komnate s velkym stolom uprostred. Lezi na nom mec.
> zobrat mec
Zobral si predmet: mec
> vychod
Si v komnate s obrom, ktory na Teba utoci.
> bojovat
Smola, stracas zivot.
> bojovat
Si uspesny, obra si zneskodnil, vypadol mu z vrecka kluc.
> zobrat kluc
Zobral si predmet: kluc
> koniec
Koncis hru predcasne, ukladam jej stav na disk.
```

## 3.2 Simulátor Braillovej tlačiarne

Naprogramujte v jazyku C program, ktorý zadaný text prevedie do Braillovho zápisu a výsledok uloží do binárneho súboru v predpísanom formáte.

Z binárneho súboru bude možné obsah bezpečne načítať a simulovať jeho vytlačenie. Program má na obrazovke z ASCII znakov vytvoriť mriežky znázorňujúce reliéfne body.

Text určený na preklad môže program načítať z parametra príkazového riadku alebo zo vstupného textového súboru.

### 3.2.1 Požiadavky

- Vstupný text môže obsahovať malé a veľké písmená, číslice a aspoň vybrané znaky interpunkcie.
- Binárny súbor musí obsahovať 32 B hlavičku a dátovú časť.

- Hlavička má obsahovať identifikátor formátu, verziu, mód, počet Braillových buniek, voliteľný kontrolný súčet a rezervované nulové bajty.
- Dátová časť nasleduje za hlavičkou a obsahuje sekvenciu kódov Braillových buniek.
- Všetky časti riešenia implementujte samostatne, bez použitia externých knižníc.
- Dbajte na efektívne uloženie a spracovanie dát, modulárny návrh a ošetrenie chybových stavov.

### 3.2.2 Príklady použitia

```
braille -t vstup.txt preklad.brl
```

Program otvorí textový súbor, spracuje ho a do výstupného binárneho súboru zapíše hlavičku vrátane kontrolného súčtu a binárnu reprezentáciu Braillovho zápisu.

```
braille -p sprava.brl
```

Program otvorí a prečíta binárny súbor, overí jeho integritu pomocou kontrolného súčtu a na obrazovke, prípadne aj do textového súboru, simuluje tlač.

### 3.2.3 Príklad výstupu

Ak je v súbore uložený Braillov zápis textu **nitra**, na obrazovke sa môže objaviť:

```
00 .0 .0 0. 0.  
.0 0. 00 00 ..  
0. .. 0. 0. ..
```

Znak 0 reprezentuje reliéfny bod a znak . prázdne miesto.

Po vytlačení Braillovho zápisu vypíše krátku štatistiku:

```
pocet znakov: 5, pocet reliefnych bodov: 15
```

### 3.2.4 Pomôcky

S pravidlami pre zápis slovenského textu v 6-bodovom Braillovom písme sa môžete oboznámiť napríklad tu:

- <https://beliana.sav.sk/heslo/braillovo-pismo>
- [https://sk.wikipedia.org/wiki/Braillovo\TU\textbackslash{}\pismo](https://sk.wikipedia.org/wiki/Braillovo%5C%5Ctextbackslash%5C%5C_pismo)
- [https://www.skn.sk/swift\TU\textbackslash{}\\\_data/source/2021/txt/sabp/pravidla-bp01.pdf](https://www.skn.sk/swift%5C%5Ctextbackslash%5C%5C_data/source/2021/txt/sabp/pravidla-bp01.pdf)
- <https://www.skn.sk/prekladac/>

## 3.3 Simulátor MHD

Naprogramujte textovú aplikáciu pre malé mesto s mestskou hromadnou dopravou. Program načíta popis siete, teda zastávky, linky a jazdné časy, a umožní plánovať trasu aj simulovať zmeny v doprave.

Aplikácia má vedieť:

- nájsť najrýchlejšiu trasu medzi dvoma zastávkami,
- zohľadniť penalizáciu prestupov,
- simulovať meškania a poruchy,
- posúvať čas simulácie,
- odpovedať na otázku, kde sa práve nachádzajú vozidlá.

### 3.3.1 Požiadavky

- Sieť reprezentujte ako orientovaný vážený graf.
- Použite dynamické štruktúry, napríklad zoznamy susedov, pole zastávok a zoznam liniek.
- Implementujte parser textového formátu siete, napríklad CSV so zastávkami, linkami, trasami a časmi.
- Plánovač trasy implementujte pomocou Dijkstrovho algoritmu.
- Min-heap môžete implementovať ako vlastný binárny heap.
- Simulácia má podporovať náhodné meškania, poruchu linky, uzavretie úseku a následné preplánovanie.
- Náhodou urobte seedovateľnú.
- Stav simulácie ukladajte do binárneho súboru a umožnite jeho obnovenie.
- Ukladaný stav má obsahovať aktuálny čas, polohy vozidiel, meškania a RNG seed.
- Implementujte CLI rozhranie a logovanie dotazov a výsledkov.

Informácie o linkách, časoch a trasách načítavajte zo súborov, napríklad z CSV.

### 3.3.2 Príklady použitia

```
mhd -n Miso -i data/ -s 20251109T073000
mhd -l Miso
```

Prvý príkaz vytvorí novú simuláciu od zadaného času. Druhý príkaz obnoví posledný uložený stav hráča alebo používateľa Miso.

### 3.3.3 Ukážka priebehu

```
> route "Hlavna stanica" "Nakupne centrum"
Najdena trasa (23 min, 1 prestup):
  07:34 Zast. Hlavna stanica -> Linka 4 -> 07:41 Park
  07:43 Linka 12 -> 07:57 Nakupne centrum
> tick 5
Cas posunutý o 5 min. Linka 4 meska +2 min (nahodna udalost).
> route "Hlavna stanica" "Nakupne centrum"
Upravený odhad: 25 min (meskanie 2 min).
> save
Ukladam Miso.dat
```

## 3.4 Lode

Naprogramujte klasickú hru Lode na dvoch poliach veľkosti 10 x 10 alebo väčších. Hráč hrá proti AI, ktorá používa stratégiu **hunt & target**: po zásahu cieľi okolité políčka.

Rozmiestnenie lodí môže byť náhodné alebo načítané zo súboru. Rozhranie hry je čisto textové.

### 3.4.1 Požiadavky

- Dynamicky alokujte polia a zoznam lodí.
- Použite aspoň 5 lodí rozličných dĺžok.
- Implementujte validáciu vstupov vrátane kontroly prekrývania lodí a okrajov hracieho poľa.
- AI musí minimálne používať náhodné hľadanie a po zásahu sa prepnúť do režimu **target**.
- AI si musí pamätať, kam už strieľala.
- Náhodné rozmiestnenie a ťahy AI musia byť seedovateľné.
- Hra sa musí spúšťať z príkazového riadku.
- Rozohru ukladajte do binárneho súboru.
- Ukladaný stav má obsahovať stav oboch polí, zásahy, zostávajúce lode a informáciu, kto je na ťahu.
- Priebeh hry exportujte do textového logu na testovanie.

### 3.4.2 Príklady použitia

```
lode -n Miso -s 9876
lode -l Miso
lode -i rozlozenie.txt
```



Prvý príkaz spustí novú hru so seedom 9876. Druhý načíta posledný uložený stav. Tretí voliteľne načíta manuálne zadané rozloženie lodí.

### 3.4.3 Ukážka priebehu

```
> vystrel B7
Zasah! (Nepriateľska lod poskodena)
AI striela na C3 ... vedla.
> vystrel B8
Potopena! (Kriznik, dlzka 3)
> uloz
Stav ulozeny do Miso.dat
```

## 3.5 Huffmanovo kódovanie

Implementujte bezstratový kompresor používajúci Huffmanovo kódovanie. Program zo vstupného textu spočíta frekvencie, postaví kódový strom a zapíše binárny komprimovaný súbor vrátane hlavičky a popisu stromu.

Druhý režim má súbor načítať, overiť a dekomprimovať. Program má zobrazovať štatistiky, napríklad veľkosť pred kompresiou a po kompresii, kompresný pomer, entropiu a priemernú dĺžku kódu.

### 3.5.1 Požiadavky

- Podporujte minimálne rozšírenú 8-bitovú sadu vstupov.
- Diakritika v UTF-8 je voliteľná; vstup môžete spracovať bajtovo.
- Implementujte vlastné počítanie frekvencií.
- Implementujte vlastný min-heap.
- Implementujte stavbu Huffmanovho stromu.
- Implementujte bitový zapisovač a bitový čítač.
- Binárny formát musí obsahovať strom uložený v pre-order s označením listov a vnútorných uzlov.
- Implementujte overenie integrity pomocou kontrolného súčtu dátovej časti.
- Program musí mať CLI rozhranie.
- Nepoužívajte externé knižnice.

### 3.5.2 Formát binárneho súboru

Výstupný súbor má príponu .huf.

Hlavička má veľkosť 32 B:

- 4 B: identifikátor HUF1,

- 1 B: verzia,
- 1 B: flagy, napríklad 0 pre raw bytes alebo 1 pre experimentálne UTF-8,
- 2 B: vyhradené bajty s hodnotou 0,
- 8 B: pôvodná veľkosť v bajtoch,
- 8 B: veľkosť komprimovaných dát v bajtoch,
- 4 B: kontrolný súčet dátovej časti,
- 4 B: rezerva.

Za hlavičkou nasleduje popis stromu. Strom je uložený binárne v pre-order poradí: 1 bit označuje list alebo vnútorný uzol, pri liste nasleduje 1 bajt symbolu.

Dátová časť obsahuje súvislý bitový stream kódov. Na konci sa stream zarovná nulami do celého bajtu.

### 3.5.3 Príklady použitia

```
huf -c roman.txt roman.huf
huf -d roman.huf roman.out.txt
huf -i roman.huf
```

Prvý príkaz komprimuje vstupný súbor. Druhý dekomprimuje .huf súbor. Tretí vypíše informácie o súbore a skontroluje jeho integritu.

### 3.5.4 Ukážka štatistiky

```
vstup: 125034 B, vystup: 75988 B
kompresny_pomer: 1.645
entropia: 4.12 b/symbol, avg_kod: 4.28 b/symbol
integrita: OK
```

## 4 Skúškové úlohy

## 4.1 Chybná alokácia pamäte

### 4.1.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.1.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    char *pw = malloc(10);
    if (!pw) {
        exit(EXIT_FAILURE);
    }

    int source[10] = {0};
    for (size_t i = 0; i < 10; i++) {
        pw[i] = source[i];
    }
    printf("%d\n", pw[0]);
    free(pw);
}

int main() {
    f();
    return EXIT_SUCCESS;
}
```

### 4.1.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.2 Pretečenie zásobníka - konštantný index

### 4.2.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.2.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    int index = 10;
    int buffer[10] = {0};

    if (index >= 0) {
        buffer[index] = 1;
        for (int i = 0; i < 10; i++) {
            printf("%d\n", buffer[i]);
        }
    } else {
        printf("ERROR: Array index is negative.\n");
    }
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.2.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.3 Pretečenie zásobníka - index zo vstupu

### 4.3.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.3.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    char inputBuffer[32] = "";
    int data = -1;
    int buffer[10] = {0};

    if (fgets(inputBuffer, sizeof(inputBuffer), stdin) != NULL) {
        data = atoi(inputBuffer);
    } else {
        fprintf(stderr, "fgets() failed.\n");
        exit(EXIT_FAILURE);
    }

    if (data >= 0) {
        buffer[data] = 1;
        for (int i = 0; i < 10; i++) {
            printf("%d\n", buffer[i]);
        }
    } else {
        printf("ERROR: Array index is negative.\n");
    }
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.3.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.4 Únik pamäte pri calloc

### 4.4.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.4.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void f(void) {
    char *data = NULL;

    data = (char *)calloc(100, sizeof(char));
    if (!data) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }

    strcpy(data, "A String");
    puts(data);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.4.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.5 Únik pamäte pri malloc

### 4.5.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.5.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

struct twoInts {
    int intOne;
    int intTwo;
};

void f(void) {
    struct twoInts *data = NULL;

    data = malloc(100 * sizeof(*data));
    if (!data) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    data[0].intOne = 0;
    data[0].intTwo = 0;
    printf("%d %d\n", data[0].intOne, data[0].intTwo);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.5.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.



## 4.6 Únik pamäte pri realloc

### 4.6.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.6.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

void f(void) {
    wchar_t *data = NULL;

    data = realloc(data, 100 * sizeof *data);
    if (!data) {
        perror("realloc");
        exit(EXIT_FAILURE);
    }

    wcsncpy(data, L"A String");
    wprintf(L"%ls\n", data);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.6.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.7 Vyčerpanie systémových zdrojov

### 4.7.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.7.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

void f() {
    char input[32];
    int count = 0;

    if (fgets(input, sizeof(input), stdin) != NULL) {
        count = atoi(input);
    } else {
        fprintf(stderr, "fgets() failed.\n");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < count; i++) {
        printf("Hello\n");
    }
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.7.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.8 Vyčerpanie systémových zdrojov pri zápise do súboru

### 4.8.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.8.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define SENTENCE "This is the sentence we are printing to the file. "

void f() {
    srand((unsigned)time(NULL));
    int count = rand();
    FILE *f = fopen("output.txt", "w+");
    if (!f) {
        exit(EXIT_FAILURE);
    }

    for (size_t i = 0; i < (size_t)count; i++) {
        if (fwrite(SENTENCE, 1, strlen(SENTENCE), f) != strlen(SENTENCE)) {
            fclose(f);
            exit(EXIT_FAILURE);
        }
    }

    fclose(f);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.8.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.9 Nesprávna kontrola návratovej hodnoty fgets

### 4.9.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.9.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

void f() {
    char buf[100] = "";

    printf("Please enter a string:\n");

    if (fgets(buf, sizeof(buf), stdin) < 0) {
        fprintf(stderr, "fgets failed!\n");
        exit(EXIT_FAILURE);
    }

    printf("%s", buf);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.9.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.10 Nesprávna kontrola návratovej hodnoty remove

### 4.10.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.10.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

void f() {
    if (remove("removeme.txt") == 0) {
        fprintf(stderr, "remove failed!\n");
    }
}

int main() {
    f();
    return EXIT_FAILURE;
}
```

### 4.10.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.11 Deklarácia viacerých premenných so smerníkom

### 4.11.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.11.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void f(void) {
    char* p1, p2;

    p1 = malloc(16);
    p2 = malloc(16);
    if (!p1 || !p2) {
        exit(EXIT_FAILURE);
    }

    strcpy(p1, "hello");
    strcpy(p2, "world");
    printf("%s %s\n", p1, p2);

    free(p1);
    free(p2);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.11.3 Požadované riešenie

1. Správne prečítajte deklaráciu premenných.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.12 Alokácia cez neinicializovaný smerník

### 4.12.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.12.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void f(void) {
    char *p;

    *p = malloc(10);
    strcpy(p, "abc");
    printf("%s\n", p);

    free(p);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.12.3 Požadované riešenie

1. Správne prečítajte kód.
2. Nájdite chybu, ak existuje, a vysvetlite, prečo je to chyba.
3. Opravte program.

## 4.13 Const a typedef smerníka

### 4.13.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program robí to, čo autor pravdepodobne zamýšľal. Ak nie, problém vysvetlite a opravte.

### 4.13.2 Vstupný kód

```
#include <stdio.h>

typedef char *charp;

void f(void) {
    char text[] = "abc";
    const charp p = text;

    p[0] = 'A';
    printf("%s\n", p);
}

int main(void) {
    f();
    return 0;
}
```

### 4.13.3 Požadované riešenie

1. Vysvetlite význam deklarácie `const charp p`.
2. Rozhodnite, či sú chránené znaky alebo samotný smerník.
3. Opravte deklaráciu podľa zámeru, aby sa znaky nedali meniť.



## 4.14 Zápis do reťazcového literálu

### 4.14.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či program obsahuje chybu. Ak áno, chybu nájdite, vysvetlite a opravte.

### 4.14.2 Vstupný kód

```
#include <stdio.h>

void f(void) {
    char *p = "hello";

    p[0] = 'H';
    printf("%s\n", p);
}

int main(void) {
    f();
    return 0;
}
```

### 4.14.3 Požadované riešenie

1. Vysvetlite rozdiel medzi poľom znakov a smerníkom na reťazcový literál.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.15 Veľkosť externého poľa

### 4.15.1 Zadanie

Prečítajte si nasledujúci program rozdelený do dvoch súborov. Vašou úlohou je zistiť, či je možné v druhom súbore určiť počet prvkov poľa pomocou `sizeof`.

### 4.15.2 Vstupný kód

```
/* data.c */  
int values[] = {1, 2, 3, 4};
```

```
/* main.c */  
#include <stdio.h>  
  
extern int values[];  
  
int main(void) {  
    size_t count = sizeof(values) / sizeof(values[0]);  
    printf("%zu\n", count);  
    return 0;  
}
```

### 4.15.3 Požadované riešenie

1. Vysvetlite, prečo je deklarácia `extern int values[]` neúplná.
2. Rozhodnite, či `sizeof(values)` v `main.c` funguje.
3. Navrhnite opravu.

## 4.16 Nesprávna aritmetika smerníkov

### 4.16.1 Zadanie

Prečítajte si nasledujúci program. Autor očakáva, že program vypíše hodnotu 3. Vašou úlohou je zistiť, či je tento predpoklad správny.

### 4.16.2 Vstupný kód

```
#include <stdio.h>

int main(void) {
    int array[5];
    int *ip;

    for (int i = 0; i < 5; i++) {
        array[i] = i;
    }

    ip = array;
    printf("%d\n", *(ip + 3 * sizeof(int)));
    return 0;
}
```

### 4.16.3 Požadované riešenie

1. Vysvetlite, ako funguje aritmetika smerníkov.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.17 Zmena smerníka vo funkcii

### 4.17.1 Zadanie

Prečítajte si nasledujúci program. Autor očakáva, že funkcia `init` nastaví smerník v `main`. Vašou úlohou je zistiť, či sa to naozaj stane.

### 4.17.2 Vstupný kód

```
#include <stdio.h>

void init(int *p) {
    static int value = 5;
    p = &value;
}

int main(void) {
    int *p = NULL;

    init(p);
    printf("%d\n", *p);
    return 0;
}
```

### 4.17.3 Požadované riešenie

1. Vysvetlite, ako sa v C odovzdávajú argumenty funkciám.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.18 Priradenie do poľa

### 4.18.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či je možné priradiť reťazec priamo do poľa znakov po jeho deklarácii.

### 4.18.2 Vstupný kód

```
#include <stdio.h>

int main(void) {
    char name[16];

    name = "student";
    printf("%s\n", name);
    return 0;
}
```

### 4.18.3 Požadované riešenie

1. Vysvetlite rozdiel medzi inicializáciou poľa a priradením do poľa.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.19 Veľkosť poľa ako parameter funkcie

### 4.19.1 Zadanie

Prečítajte si nasledujúci program. Autor očakáva, že funkcia `print_size` vypíše veľkosť celého poľa. Vašou úlohou je zistiť, či je tento predpoklad správny.

### 4.19.2 Vstupný kód

```
#include <stdio.h>

void print_size(char text[10]) {
    printf("%zu\n", sizeof(text));
}

int main(void) {
    char text[10] = "abc";

    print_size(text);
    return 0;
}
```

### 4.19.3 Požadované riešenie

1. Vysvetlite, čo sa stane s parametrom `char text[10]`.
2. Nájdite chybu v očakávaní autora.
3. Navrhните správny spôsob práce s veľkosťou poľa.

## 4.20 Záměna dvojrozměrného pole a dvojitého smerníka

### 4.20.1 Zadanie

Prečítajte si nasledujúci program. Autor sa pokúša odovzdať dvojrozmerné pole funkcii, ktorá očakáva `int **`. Vašou úlohou je zistiť, či sú tieto typy zameniteľné.

### 4.20.2 Vstupný kód

```
#include <stdio.h>

void print_matrix(int **m, size_t rows, size_t cols) {
    for (size_t r = 0; r < rows; r++) {
        for (size_t c = 0; c < cols; c++) {
            printf("%d ", m[r][c]);
        }
        putchar('\n');
    }
}

int main(void) {
    int matrix[2][3] = {
        {1, 2, 3},
        {4, 5, 6},
    };

    print_matrix((int **)matrix, 2, 3);
    return 0;
}
```

### 4.20.3 Požadované riešenie

1. Vysvetlite rozdiel medzi `int [2][3]` a `int **`.
2. Nájdite chybu, ak existuje.
3. Opravte deklaráciu funkcie.

## 4.21 Čítanie do nealokovaného reťazca

### 4.21.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či je cieľový reťazec pripravený na čítanie vstupu.

### 4.21.2 Vstupný kód

```
#include <stdio.h>

int main(void) {
    char *answer;

    printf("Zadajte text:\n");
    scanf("%31s", answer);
    printf("Zadali ste: %s\n", answer);

    return 0;
}
```

### 4.21.3 Požadované riešenie

1. Vysvetlite, čo obsahuje premenná `answer`.
2. Nájdite chybu, ak existuje.
3. Opravte program.



## 4.22 Spájanie reťazcov bez cieľového buffera

### 4.22.1 Zadanie

Prečítajte si nasledujúci program. Autor chce spojiť dva reťazce do výsledku `Hello, world!`. Vašou úlohou je zistiť, či je cieľový buffer pripravený správne.

### 4.22.2 Vstupný kód

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char *s1 = "Hello, ";
    char *s2 = "world!";
    char *s3 = strcat(s1, s2);

    printf("%s\n", s3);
    return 0;
}
```

### 4.22.3 Požadované riešenie

1. Vysvetlite, kam zapisuje `strcat`.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.23 Uloženie viacerých riadkov do jedného buffera

### 4.23.1 Zadanie

Prečítajte si nasledujúci program. Autor chce načítať viac riadkov a neskôr ich vypísať. Vašou úlohou je zistiť, či sa jednotlivé riadky ukladajú samostatne.

### 4.23.2 Vstupný kód

```
#include <stdio.h>

void f(FILE *fp) {
    char linebuf[80];
    char *lines[100];
    int count = 0;

    for (int i = 0; i < 100; i++) {
        char *p = fgets(linebuf, sizeof(linebuf), fp);
        if (p == NULL) {
            break;
        }
        lines[count++] = p;
    }

    for (int i = 0; i < count; i++) {
        printf("%s", lines[i]);
    }
}
```

### 4.23.3 Požadované riešenie

1. Vysvetlite, čo sa ukladá do poľa `lines`.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.24 Návrat lokálneho poľa z funkcie

### 4.24.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či funkcia bezpečne vracia reťazec volajúcemu.

### 4.24.2 Vstupný kód

```
#include <stdio.h>
#include <string.h>

char *make_name(void) {
    char name[32];

    strcpy(name, "student");
    return name;
}

int main(void) {
    char *name = make_name();

    printf("%s\n", name);
    return 0;
}
```

### 4.24.3 Požadované riešenie

1. Vysvetlite životnosť lokálneho poľa `name`.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.25 Alokácia poľa bez sizeof prvku

### 4.25.1 Zadanie

Prečítajte si nasledujúci program. Autor chce dynamicky alokovať pole `nints` hodnôt typu `int`. Vašou úlohou je zistiť, či je veľkosť alokácie správna.

### 4.25.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

void f(size_t nints) {
    int *iarray = malloc(nints);
    if (!iarray) {
        exit(EXIT_FAILURE);
    }

    for (size_t i = 0; i < nints; i++) {
        iarray[i] = (int)i;
    }

    printf("%d\n", iarray[0]);
    free(iarray);
}
```

### 4.25.3 Požadované riešenie

1. Vysvetlite, akú jednotku očakáva `malloc`.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.26 Použitie pamäte po free

### 4.26.1 Zadanie

Prečítajte si nasledujúci program. Vašou úlohou je zistiť, či sa s dynamicky alokovanou pamäťou pracuje po celý čas bezpečne.

### 4.26.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    int *p = malloc(sizeof(*p));
    if (!p) {
        exit(EXIT_FAILURE);
    }

    *p = 42;
    free(p);

    printf("%d\n", *p);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

### 4.26.3 Požadované riešenie

1. Vysvetlite, čo znamená uvoľnenie pamäte.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.27 Smerník po free nie je automaticky NULL

### 4.27.1 Zadanie

Prečítajte si nasledujúci program. Autor očakáva, že po `free` bude smerník automaticky nastavený na `NULL`. Vašou úlohou je zistiť, či je tento predpoklad správny.

### 4.27.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    int *p = malloc(sizeof(*p));
    if (!p) {
        exit(EXIT_FAILURE);
    }

    *p = 7;
    free(p);

    if (p != NULL) {
        printf("p stale nie je NULL\n");
    }
}
```

### 4.27.3 Požadované riešenie

1. Vysvetlite, čo `free` mení a čo nemení.
2. Rozhodnite, či je kontrola `p != NULL` po `free` spoľahlivý test platnosti pamäte.
3. Navrhnite bezpečnejší postup.

## 4.28 Zistenie veľkosti alokovaného bloku cez sizeof

### 4.28.1 Zadanie

Prečítajte si nasledujúci program. Autor chce pomocou `sizeof` zistiť počet prvkov dynamicky alokovaného poľa. Vašou úlohou je zistiť, či je tento výpočet správny.

### 4.28.2 Vstupný kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    int *values = malloc(10 * sizeof(*values));
    if (!values) {
        exit(EXIT_FAILURE);
    }

    size_t count = sizeof(values) / sizeof(values[0]);
    printf("%zu\n", count);

    free(values);
}
```

### 4.28.3 Požadované riešenie

1. Vysvetlite, čo vracia `sizeof(values)`.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.29 Porovnávanie reťazcov operátorom ==

### 4.29.1 Zadanie

Prečítajte si nasledujúci program. Autor chce zistiť, či používateľ zadal slovo `admin`. Vašou úlohou je zistiť, či porovnanie funguje správne.

### 4.29.2 Vstupný kód

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char input[32];

    if (fgets(input, sizeof(input), stdin) == NULL) {
        return 1;
    }
    input[strcspn(input, "\n")] = '\0';

    if (input == "admin") {
        printf("OK\n");
    } else {
        printf("DENIED\n");
    }

    return 0;
}
```

### 4.29.3 Požadované riešenie

1. Vysvetlite, čo porovnáva operátor `==` pri poliach a smerníkoch.
2. Nájdite chybu, ak existuje.
3. Opravte program.



## 4.30 Pridanie znaku pomocou strcat

### 4.30.1 Zadanie

Prečítajte si nasledujúci program. Autor chce pridať znak ! na koniec reťazca. Vašou úlohou je zistiť, či je volanie `strcat` správne.

### 4.30.2 Vstupný kód

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char text[16] = "hello";

    strcat(text, '!');
    printf("%s\n", text);
    return 0;
}
```

### 4.30.3 Požadované riešenie

1. Vysvetlite, aký typ očakáva druhý parameter `strcat`.
2. Nájdite chybu, ak existuje.
3. Opravte program.

## 4.31 Bonus: Úprava reťazcového literálu

### 4.31.1 Zadanie

Toto je bonusová náročnejšia úloha. Prečítajte si nasledujúci program. Program môže na niektorých systémoch okamžite spadnúť, inde sa môže správať inak. Vašou úlohou je vysvetliť prečo.

### 4.31.2 Vstupný kód

```
#include <stdio.h>

int main(void) {
    char *message = "hello, world";

    message[0] = 'H';
    puts(message);
    return 0;
}
```

### 4.31.3 Požadované riešenie

1. Vysvetlite, kde je reťazcový literál uložený z pohľadu programu.
2. Vysvetlite, prečo môže program spadnúť až pri behu.
3. Opravte program tak, aby bol reťazec modifikovateľný.

## 4.32 Bonus: Nezarovnaný prístup do pamäte

### 4.32.1 Zadanie

Toto je bonusová náročnejšia úloha. Program číta binárne dáta zo súboru do poľa bajtov a následne ich pretypuje na väčšie číselné typy. Vašou úlohou je zistiť, prečo môže program na niektorých architektúrach spadnúť alebo čítať nesprávne hodnoty.

### 4.32.2 Vstupný kód

```
#include <stdio.h>
#include <stdint.h>

struct record {
    char tag;
    uint32_t id;
    uint16_t flags;
};

int read_record(FILE *fp, struct record *out) {
    unsigned char buf[7];
    unsigned char *p = buf;

    if (fread(buf, sizeof(buf), 1, fp) != 1) {
        return 0;
    }

    out->tag = *p++;
    out->id = *(uint32_t *)p;
    p += 4;
    out->flags = *(uint16_t *)p;
    return 1;
}
```

### 4.32.3 Požadované riešenie

1. Vysvetlite problém so zarovnaním pamäte.
2. Vysvetlite, prečo je pretypovanie bajtového buffera na `uint32_t *` alebo `uint16_t *` rizikové.
3. Opravte program bezpečným čítaním bajtov.

## 5 Riešenia

## 5.1 Chybná alokácia pamäte

### 5.1.1 Očakávaná odpoveď

Program obsahuje chybu pri práci s alokovanou pamäťou. Premenná `pw` je typu `char *` a alokuje sa pre ňu 10 bajtov, ale program do nej kopíruje hodnoty z poľa `int source[10]`.

### 5.1.2 Vysvetlenie chyby

Volanie `malloc(10)` alokuje 10 bajtov. Cyklus však pracuje s desiatimi hodnotami typu `int`. Ak chceme ukladať hodnoty typu `int`, cieľový ukazovateľ má mať typ `int *` a alokácia má zohľadniť veľkosť jedného prvku.

Pôvodný kód navyše pri priradení `pw[i] = source[i]` implicitne konvertuje hodnotu typu `int` na `char`, čo nie je vhodné, ak zámerom programu je kopírovať celé čísla.

### 5.1.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    int *pw = malloc(10 * sizeof(*pw));
    if (!pw) {
        exit(EXIT_FAILURE);
    }

    int source[10] = {0};
    for (size_t i = 0; i < 10; i++) {
        pw[i] = source[i];
    }
    printf("%d\n", pw[0]);
    free(pw);
}

int main() {
    f();
    return EXIT_SUCCESS;
}
```

## 5.2 Pretečenie zásobníka - konštantný index

### 5.2.1 Očakávaná odpoveď

Program obsahuje zápis mimo rozsahu poľa. Premenná `index` má hodnotu 10, ale pole `buffer` má 10 prvkov s platnými indexmi 0 až 9.

### 5.2.2 Vysvetlenie chyby

Podmienka `index >= 0` kontroluje iba spodnú hranicu indexu. Horná hranica sa nekontroluje, preto sa vykoná zápis `buffer[10] = 1`, ktorý zapisuje za koniec poľa na zásobníku.

Takýto zápis môže prepísať inú lokálnu premennú, riadiace údaje funkcie alebo spôsobiť pád programu. Ide o nedefinované správanie a potenciálnu bezpečnostnú zraniteľnosť.

### 5.2.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    int index = 10;
    int buffer[10] = {0};
    const size_t buffer_size = sizeof(buffer) / sizeof(buffer[0]);

    if (index >= 0 && (size_t)index < buffer_size) {
        buffer[index] = 1;
        for (size_t i = 0; i < buffer_size; i++) {
            printf("%d\n", buffer[i]);
        }
    } else {
        printf("ERROR: Array index is out of bounds.\n");
    }
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

## 5.3 Pretečenie zásobníka - index zo vstupu

### 5.3.1 Očakávaná odpoveď

Program obsahuje zápis mimo rozsahu poľa. Index sa číta zo vstupu a pred použitím sa kontroluje iba to, či nie je záporný. Horná hranica poľa sa nekontroluje.

### 5.3.2 Vysvetlenie chyby

Pole `buffer` má platné indexy 0 až 9. Ak používateľ zadá napríklad hodnotu 10, podmienka `data >= 0` je splnená a program vykoná zápis `buffer[10] = 1`.

Použitie `atoi` navyše neposkytuje spoľahlivú kontrolu nečíselného vstupu ani pretečenia. V bezpečnejšom riešení je vhodné použiť `strtol` a následne overiť celý rozsah.

### 5.3.3 Opravený kód

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    char inputBuffer[32] = "";
    int buffer[10] = {0};
    const size_t buffer_size = sizeof(buffer) / sizeof(buffer[0]);

    if (fgets(inputBuffer, sizeof(inputBuffer), stdin) == NULL) {
        fprintf(stderr, "fgets() failed.\n");
        exit(EXIT_FAILURE);
    }

    errno = 0;
    char *end = NULL;
    long data = strtol(inputBuffer, &end, 10);
    if (errno != 0 || end == inputBuffer || data < 0 || (size_t)data >= buffer_size) {
        fprintf(stderr, "ERROR: Array index is out of bounds.\n");
        exit(EXIT_FAILURE);
    }

    buffer[data] = 1;
    for (size_t i = 0; i < buffer_size; i++) {
        printf("%d\n", buffer[i]);
    }
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

## 5.4 Únik pamäte pri calloc

### 5.4.1 Očakávaná odpoveď

Program obsahuje únik pamäte. Pamäť alokovaná pomocou `calloc` sa po použití neuvolní volaním `free`.

### 5.4.2 Vysvetlenie chyby

Premenná `data` ukazuje na blok pamäte na halde. Po skončení funkcie lokálny ukazovateľ zanikne, ale alokovaný blok zostane pridelený. Pri opakovanom volaní alebo v dlho bežiacom programe to vedie k rastúcej spotrebe pamäte.

### 5.4.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void f(void) {
    char *data = calloc(100, sizeof(*data));
    if (!data) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }

    strcpy(data, "A String");
    puts(data);
    free(data);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```



## 5.5 Únik pamäte pri malloc

### 5.5.1 Očakávaná odpoveď

Program obsahuje únik pamäte. Blok alokovaný volaním `malloc(100 * sizeof(*data))` sa nikdy neuvoľní.

### 5.5.2 Vysvetlenie chyby

Po návrate z funkcie `f` lokálny ukazovateľ `data` zanikne, ale pamäť na halde zostane alokovaná. Program tak stratí možnosť blok korektne uvoľniť.

### 5.5.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>

struct twoInts {
    int intOne;
    int intTwo;
};

void f(void) {
    struct twoInts *data = malloc(100 * sizeof(*data));
    if (!data) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    data[0].intOne = 0;
    data[0].intTwo = 0;
    printf("%d %d\n", data[0].intOne, data[0].intTwo);
    free(data);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

## 5.6 Únik pamäte pri realloc

### 5.6.1 Očakávaná odpoveď

Program obsahuje únik pamäte. Volanie `realloc` s počiatočným ukazovateľom `NULL` funguje ako `malloc`, ale výsledný blok sa po použití neuvoľní.

### 5.6.2 Vysvetlenie chyby

Premenná `data` ukazuje na dynamicky alokovaný blok pre 100 prvkov typu `wchar_t`. Funkcia do bloku skopíruje reťazec a vypíše ho, ale pred návratom nezavolá `free(data)`.

### 5.6.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

void f(void) {
    wchar_t *data = realloc(NULL, 100 * sizeof(*data));
    if (!data) {
        perror("realloc");
        exit(EXIT_FAILURE);
    }

    wcscpy(data, L"A String");
    wprintf(L"%ls\n", data);
    free(data);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

Ak dynamická alokácia nie je potrebná, alternatívou je lokálne pole:

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

void f(void) {
    wchar_t data[100];
    wcscpy(data, L"A String");
    wprintf(L"%ls\n", data);
}

int main(void) {
    f();
}
```

```
    return EXIT_SUCCESS;  
}
```

## 5.7 Vyčerpanie systémových zdrojov

### 5.7.1 Očakávaná odpoveď

Program používa hodnotu zo vstupu priamo ako počet iterácií cyklu. Ak používateľ zadá veľmi veľké číslo, program môže extrémne dlho vypisovať text a vyčerpať CPU, čas alebo logovací výstup.

### 5.7.2 Vysvetlenie chyby

Vstup sa konvertuje pomocou `atoi`, bez kontroly rozsahu a bez overenia, či bol vstup skutočne platné číslo. Hodnota sa následne použije v cykle `for`.

Bezpečnejšie riešenie má vstup validovať a nastaviť rozumný limit.

### 5.7.3 Opravený kód

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    char input[32];

    if (fgets(input, sizeof(input), stdin) == NULL) {
        fprintf(stderr, "fgets() failed.\n");
        exit(EXIT_FAILURE);
    }

    errno = 0;
    char *end = NULL;
    long count = strtol(input, &end, 10);
    if (errno != 0 || end == input || count < 1 || count > 20) {
        fprintf(stderr, "ERROR: count out of range (1..20).\n");
        exit(EXIT_FAILURE);
    }

    for (long i = 0; i < count; i++) {
        printf("Hello\n");
    }
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

## 5.8 Vyčerpanie systémových zdrojov pri zápise do súboru

### 5.8.1 Očakávaná odpoveď

Program používa hodnotu z `rand()` ako počet zápisov do súboru bez obmedzenia. Keďže `rand()` môže vrátiť veľkú hodnotu, program môže vytvoriť veľmi veľký súbor a vyčerpať diskový priestor alebo I/O kapacitu.

### 5.8.2 Vysvetlenie chyby

Premenná `count` nie je validovaná. Cyklus zapisuje reťazec do súboru `count`-krát. Pri veľkej hodnote ide o typické vyčerpanie systémových zdrojov.

Pri práci so súborami je zároveň vhodné overovať návratové hodnoty `fwrite` aj `fclose`.

### 5.8.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define SENTENCE "This is the sentence we are printing to the file. "

void f(void) {
    srand((unsigned)time(NULL));
    int count = rand();
    const int min = 1;
    const int max = 20;

    if (count < min || count > max) {
        fprintf(stderr, "ERROR: invalid count (%d), allowed range is %d..%d\n",
            count, min, max);
        exit(EXIT_FAILURE);
    }

    FILE *file = fopen("output.txt", "w");
    if (!file) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    const size_t expected = strlen(SENTENCE);
    for (int i = 0; i < count; i++) {
        if (fwrite(SENTENCE, 1, expected, file) != expected) {
            perror("fwrite");
            fclose(file);
            exit(EXIT_FAILURE);
        }
    }
}
```

```
    if (fclose(file) != 0) {  
        perror("fclose");  
        exit(EXIT_FAILURE);  
    }  
}  
  
int main(void) {  
    f();  
    return EXIT_SUCCESS;  
}
```

## 5.9 Nesprávna kontrola návratovej hodnoty fgets

### 5.9.1 Očakávaná odpoveď

Program nesprávne kontroluje návratovú hodnotu funkcie `fgets`. Funkcia `fgets` pri chybe alebo pri EOF bez načítaných dát vracia `NULL`, nie záporné číslo.

### 5.9.2 Vysvetlenie chyby

Výraz `fgets(...) < 0` porovnáva ukazovateľ so zápornou hodnotou, čo nedáva zmysel. Správna kontrola je porovnanie návratovej hodnoty s `NULL`.

Ak sa chyba neodhalí, program môže pracovať s neplatným alebo neaktuálnym obsahom buffera.

### 5.9.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    char buf[100] = "";

    printf("Please enter a string:\n");

    if (fgets(buf, sizeof(buf), stdin) == NULL) {
        fprintf(stderr, "fgets failed!\n");
        exit(EXIT_FAILURE);
    }

    printf("%s", buf);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

## 5.10 Nesprávna kontrola návratovej hodnoty remove

### 5.10.1 Očakávaná odpoveď

Program nesprávne kontroluje návratovú hodnotu funkcie `remove`. Funkcia `remove` vracia 0 pri úspechu a nenulovú hodnotu pri chybe.

### 5.10.2 Vysvetlenie chyby

Pôvodný kód vypíše hlásenie o chybe práve vtedy, keď sa súbor úspešne odstráni. Ak odstránenie zlyhá, program chybu ignoruje. To vedie k mätúcim logom a k nesprávnym predpokladom o stave súborového systému.

Funkciu je vhodné kontrolovať na nenulovú návratovú hodnotu a pri chybe vypísať dôvod pomocou `perror`.

### 5.10.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    if (remove("removeme.txt") != 0) {
        perror("remove failed");
        exit(EXIT_FAILURE);
    }

    puts("File removed successfully.");
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```



## 5.11 Deklarácia viacerých premenných so smerníkom

### 5.11.1 Očakávaná odpoveď

Program obsahuje chybu v deklarácii `char* p1, p2;`. Smerníkom je iba premenná `p1`; premenná `p2` je obyčajný `char`.

### 5.11.2 Vysvetlenie chyby

Znak `*` sa viaže na konkrétny deklarátor, nie na celý zoznam premenných. Výraz `p2 = malloc(16)` sa preto pokúša priradiť adresu do premennej typu `char`. Následné použitie `p2` ako reťazca a volanie `free(p2)` sú chybné.

### 5.11.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void f(void) {
    char *p1, *p2;

    p1 = malloc(16);
    p2 = malloc(16);
    if (!p1 || !p2) {
        free(p1);
        free(p2);
        exit(EXIT_FAILURE);
    }

    strcpy(p1, "hello");
    strcpy(p2, "world");
    printf("%s %s\n", p1, p2);

    free(p1);
    free(p2);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

## 5.12 Alokácia cez neinicializovaný smerník

### 5.12.1 Očakávaná odpoveď

Program používa neinicializovaný smerník `p`. Volanie `*p = malloc(10);` nedáva alokovanú adresu do `p`, ale dereferencuje neplatnú hodnotu smerníka a pokúša sa zapisovať na neznáme miesto.

### 5.12.2 Vysvetlenie chyby

Ak chceme uložiť adresu prideleného bloku do smerníka, priradujeme priamo do smerníka: `p = malloc(...)`. Dereferencovanie `*p` je správne až vtedy, keď `p` ukazuje na platný objekt.

### 5.12.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void f(void) {
    char *p = malloc(10);
    if (!p) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    strcpy(p, "abc");
    printf("%s\n", p);

    free(p);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```

## 5.13 Const a typedef smerníka

### 5.13.1 Očakávaná odpoveď

`const charp` p neznamená `const char *p`. Keďže `charp` je `typedef` pre `char *`, deklarácia znamená konštantný smerník na modifikovateľný `char`.

### 5.13.2 Vysvetlenie chyby

`typedef` nevykonáva textovú náhradu ako makro. Typ `charp` je už typ smerníka. Pridané `const` sa preto viaže na celý typ smerníka, nie na znak, na ktorý smerník ukazuje. Premennú `p` nemožno presmerovať, ale znaky cez `p` meniť možno.

Ak má byť chránený obsah reťazca, treba deklarovať smerník na konštantné znaky.

### 5.13.3 Opravený kód

```
#include <stdio.h>

void f(void) {
    char text[] = "abc";
    const char *p = text;

    /* p[0] = 'A'; // chyba: cez p sa obsah meniť nesmie */
    printf("%s\n", p);
}

int main(void) {
    f();
    return 0;
}
```

## 5.14 Zápis do reťazcového literálu

### 5.14.1 Očakávaná odpoveď

Program sa pokúša meniť reťazcový literál. Reťazcový literál nie je určený na zápis a pokus o jeho modifikáciu má nedefinované správanie.

### 5.14.2 Vysvetlenie chyby

Deklarácia `char *p = "hello";` nastaví `p` na adresu literálu. Aj keď historicky má literál typ poľa `char`, jeho úprava nie je dovolená. Program môže spadnúť alebo sa správať nepredvídateľne.

Ak chceme reťazec meniť, potrebujeme vlastné pole.

### 5.14.3 Opravený kód

```
#include <stdio.h>

void f(void) {
    char p[] = "hello";

    p[0] = 'H';
    printf("%s\n", p);
}

int main(void) {
    f();
    return 0;
}
```

## 5.15 Veľkosť externého poľa

### 5.15.1 Očakávaná odpoveď

V súbore `main.c` je `values` deklarované ako externé pole neznámej veľkosti. Typ poľa je neúplný, preto z neho nemožno pomocou `sizeof(values)` vypočítať počet prvkov.

### 5.15.2 Vysvetlenie chyby

Veľkosť poľa je známa v súbore, kde je pole definované inicializátorom. V inom prekladovom module deklarácia `extern int values[]` hovorí iba to, že niekde existuje pole typu `int`, ale neuvádza jeho počet prvkov.

Jedno riešenie je zverejniť aj samostatnú premennú s počtom prvkov.

### 5.15.3 Opravený kód

```
/* data.c */
#include <stddef.h>

int values[] = {1, 2, 3, 4};
const size_t values_count = sizeof(values) / sizeof(values[0]);
```

```
/* main.c */
#include <stdio.h>
#include <stddef.h>

extern int values[];
extern const size_t values_count;

int main(void) {
    printf("%zu\n", values_count);
    return 0;
}
```

## 5.16 Nesprávna aritmetika smerníkov

### 5.16.1 Očakávaná odpoveď

Program používa nesprávnu aritmetiku smerníkov. Pri výraze `ip + 3` sa adresa posunie o tri prvky typu `int`, nie o tri bajty. Násobenie hodnotou `sizeof(int)` preto spôsobí príliš veľký posun.

### 5.16.2 Vysvetlenie chyby

Ak je `ip` typu `int *`, výraz `ip + n` ukazuje na `n`-tý ďalší prvok typu `int`. Výraz `ip + 3 * sizeof(int)` pri bežnom `sizeof(int) == 4` posunie smerník o 12 prvkov, čo je mimo poľa.

### 5.16.3 Opravený kód

```
#include <stdio.h>

int main(void) {
    int array[5];
    int *ip;

    for (int i = 0; i < 5; i++) {
        array[i] = i;
    }

    ip = array;
    printf("%d\n", *(ip + 3));
    return 0;
}
```

## 5.17 Zmena smerníka vo funkcii

### 5.17.1 Očakávaná odpoveď

Program obsahuje chybu. Funkcia `init` mení iba lokálnu kópiu smerníka `p`, nie smerník v `main`.

### 5.17.2 Vysvetlenie chyby

C odovzdáva argumenty hodnotou. Aj smerník je hodnota. Ak má funkcia zmeniť smerník volajúceho, musí dostať adresu tohto smerníka, teda parameter typu `int **`, alebo musí nový smerník vrátiť ako návratovú hodnotu.

### 5.17.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>

void init(int **p) {
    static int value = 5;
    *p = &value;
}

int main(void) {
    int *p = NULL;

    init(&p);
    if (!p) {
        return EXIT_FAILURE;
    }

    printf("%d\n", *p);
    return EXIT_SUCCESS;
}
```

## 5.18 Priradenie do poľa

### 5.18.1 Očakávaná odpoveď

Program je chybný. Pole v C nie je priraditeľný objekt, takže po deklarácii nemožno napísať `name = "student";`.

### 5.18.2 Vysvetlenie chyby

Retazec možno použiť pri inicializácii poľa:

```
char name[] = "student";
```

Po deklarácii už treba obsah do poľa kopírovať, napríklad pomocou `strcpy` alebo bezpečnejšie s kontrolou veľkosti cieľového poľa.

### 5.18.3 Opravený kód

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char name[16];

    strcpy(name, "student");
    printf("%s\n", name);
    return 0;
}
```



## 5.19 Veľkosť poľa ako parameter funkcie

### 5.19.1 Očakávaná odpoveď

Vo funkčnom parametri sa `char text[10]` upraví na `char *text`. Výraz `sizeof(text)` preto nevracia veľkosť pôvodného poľa, ale veľkosť smerníka.

### 5.19.2 Vysvetlenie chyby

Pole odovzdané funkcii sa v parametri správa ako smerník na prvý prvok. Funkcia z parametra nevie zistiť počet prvkov pôvodného poľa. Veľkosť treba odovzdať samostatne.

### 5.19.3 Opravený kód

```
#include <stdio.h>

void print_size(char text[], size_t size) {
    (void)text;
    printf("%zu\n", size);
}

int main(void) {
    char text[10] = "abc";

    print_size(text, sizeof(text));
    return 0;
}
```

## 5.20 Zámena dvojrozmerného poľa a dvojitého smerníka

### 5.20.1 Očakávaná odpoveď

Dvojrozmerné pole `int matrix[2][3]` nie je to isté ako `int` . Pretypovanie na `int` je chybné a následné indexovanie má nedefinované správanie.

### 5.20.2 Vysvetlenie chyby

Pole `int [2][3]` je súvislý blok šiestich hodnôt `int`, usporiadaný po riadkoch. Typ `int **` je smerník na smerník, typicky používaný pri poli smerníkov na samostatne alokované riadky. Tieto reprezentácie nie sú kompatibilné.

Pri staticky známom počte stĺpcov má funkcia prijať smerník na pole troch prvkov.

### 5.20.3 Opravený kód

```
#include <stdio.h>

void print_matrix(size_t rows, size_t cols, int m[rows][cols]) {
    for (size_t r = 0; r < rows; r++) {
        for (size_t c = 0; c < cols; c++) {
            printf("%d ", m[r][c]);
        }
        putchar('\n');
    }
}

int main(void) {
    int matrix[2][3] = {
        {1, 2, 3},
        {4, 5, 6},
    };

    print_matrix(2, 3, matrix);
    return 0;
}
```

## 5.21 Čítanie do nealokovaného reťazca

### 5.21.1 Očakávaná odpoveď

Program obsahuje chybu. Premenná `answer` je neinicializovaný smerník a neukazuje na žiadny platný buffer, do ktorého by bolo možné čítať.

### 5.21.2 Vysvetlenie chyby

Smerník sám o sebe nealokuje priestor pre znaky. Funkcia `scanf` potrebuje adresu platného poľa znakov alebo dynamicky alokovaného bloku. Zápis cez neinicializovaný smerník má nedefinované správanie.

### 5.21.3 Opravený kód

```
#include <stdio.h>

int main(void) {
    char answer[32];

    printf("Zadajte text:\n");
    if (scanf("%31s", answer) != 1) {
        fprintf(stderr, "Nepodarilo sa nacitat vstup.\n");
        return 1;
    }

    printf("Zadali ste: %s\n", answer);
    return 0;
}
```

## 5.22 Spájanie reťazcov bez cieľového buffera

### 5.22.1 Očakávaná odpoveď

Program obsahuje chybu. Funkcia `strcat` nepridelí nový reťazec. Pripája druhý reťazec na koniec prvého a zapisuje do cieľového buffera.

### 5.22.2 Vysvetlenie chyby

Premenná `s1` ukazuje na reťazcový literál, ktorý sa nesmie meniť a navyše nemá rezervované miesto na pripojenie ďalších znakov. Volanie `strcat(s1, s2)` má nedefinované správanie.

### 5.22.3 Opravený kód

```
#include <stdio.h>
#include <string.h>

int main(void) {
    const char *s1 = "Hello, ";
    const char *s2 = "world!";
    char result[32];

    strcpy(result, s1);
    strcat(result, s2);

    printf("%s\n", result);
    return 0;
}
```

## 5.23 Uloženie viacerých riadkov do jedného buffera

### 5.23.1 Očakávaná odpoveď

Program neukladá samostatné riadky. Do poľa `lines` ukladá vždy rovnakú adresu lokálneho buffera `linebuf`.

### 5.23.2 Vysvetlenie chyby

Funkcia `fgets` vracia ukazovateľ na buffer, ktorý jej bol odovzdaný. V každej iterácii sa teda prepíše rovnaké pole `linebuf` a všetky prvky `lines` ukazujú na rovnaké miesto. Na konci obsahujú kópie posledného načítaného riadku.

Každý riadok treba skopírovať do vlastnej pamäte.

### 5.23.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void f(FILE *fp) {
    char linebuf[80];
    char *lines[100];
    int count = 0;

    while (count < 100 && fgets(linebuf, sizeof(linebuf), fp) != NULL) {
        lines[count] = malloc(strlen(linebuf) + 1);
        if (!lines[count]) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }
        strcpy(lines[count], linebuf);
        count++;
    }

    for (int i = 0; i < count; i++) {
        printf("%s", lines[i]);
        free(lines[i]);
    }
}
```

## 5.24 Návrat lokálneho poľa z funkcie

### 5.24.1 Očakávaná odpoveď

Program vracia adresu lokálneho poľa, ktoré zanikne pri návrate z funkcie `make_name`. Použitie vráteného smerníka má nedefinované správanie.

### 5.24.2 Vysvetlenie chyby

Pole `name` má automatickú životnosť. Po skončení funkcie už neexistuje. Smerník vrátený z funkcie preto ukazuje na neplatné miesto.

Jedno bezpečné riešenie je odovzdať cieľový buffer do funkcie.

### 5.24.3 Opravený kód

```
#include <stdio.h>
#include <string.h>

void make_name(char *name, size_t size) {
    if (size > 0) {
        snprintf(name, size, "%s", "student");
    }
}

int main(void) {
    char name[32];

    make_name(name, sizeof(name));
    printf("%s\n", name);
    return 0;
}
```

## 5.25 Alokácia poľa bez sizeof prvku

### 5.25.1 Očakávaná odpoveď

Program alokuje iba `nints` bajtov, nie miesto pre `nints` prvkov typu `int`.

### 5.25.2 Vysvetlenie chyby

Argument funkcie `malloc` je počet bajtov. Ak chceme alokovať pole `nints` hodnôt typu `int`, treba násobiť počtom bajtov jedného prvku.

### 5.25.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>

void f(size_t nints) {
    int *iarray = malloc(nints * sizeof(*iarray));
    if (!iarray) {
        exit(EXIT_FAILURE);
    }

    for (size_t i = 0; i < nints; i++) {
        iarray[i] = (int)i;
    }

    printf("%d\n", iarray[0]);
    free(iarray);
}
```

## 5.26 Použitie pamäte po free

### 5.26.1 Očakávaná odpoveď

Program používa pamäť po jej uvoľnení. Výraz `*p` po volaní `free(p)` má nedefinované správanie.

### 5.26.2 Vysvetlenie chyby

Po `free(p)` už program nesmie čítať ani zapisovať cez `p`. Blok môže byť znovu použitý alokátorom alebo môže byť interný stav alokátora poškodený ďalším zápisom.

### 5.26.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    int *p = malloc(sizeof(*p));
    if (!p) {
        exit(EXIT_FAILURE);
    }

    *p = 42;
    printf("%d\n", *p);

    free(p);
}

int main(void) {
    f();
    return EXIT_SUCCESS;
}
```



## 5.27 Smerník po free nie je automaticky NULL

### 5.27.1 Očakávaná odpoveď

Volanie `free(p)` uvoľní blok pamäte, ale nezmení hodnotu lokálnej premennej `p`. Smerník po `free` typicky stále obsahuje pôvodnú adresu, ktorá už však nie je platná na dereferencovanie.

### 5.27.2 Vysvetlenie chyby

Kontrola `p != NULL` po `free` nehovorí, či je blok stále platný. Ukazuje iba to, že premenná `p` neobsahuje nulový smerník. Ak chceme znížiť riziko náhodného opätovného použitia, môžeme po `free` nastaviť smerník na `NULL`.

### 5.27.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    int *p = malloc(sizeof(*p));
    if (!p) {
        exit(EXIT_FAILURE);
    }

    *p = 7;
    free(p);
    p = NULL;

    if (p == NULL) {
        printf("p je nastaveny na NULL\n");
    }
}
```

## 5.28 Zistenie veľkosti alokovaného bloku cez sizeof

### 5.28.1 Očakávaná odpoveď

Výraz `sizeof(values)` vracia veľkosť smerníka, nie veľkosť dynamicky alokovaného bloku.

### 5.28.2 Vysvetlenie chyby

Po alokácii cez `malloc` si program musí počet prvkov pamätať samostatne. Jazyk C neposkytuje prenosný spôsob, ako zo samotného smerníka zistiť veľkosť prideleného bloku.

### 5.28.3 Opravený kód

```
#include <stdio.h>
#include <stdlib.h>

void f(void) {
    size_t count = 10;
    int *values = malloc(count * sizeof(*values));
    if (!values) {
        exit(EXIT_FAILURE);
    }

    printf("%zu\n", count);

    free(values);
}
```

## 5.29 Porovnávanie reťazcov operátorom ==

### 5.29.1 Očakávaná odpoveď

Program porovnáva adresy, nie obsah reťazcov. Výraz `input == "admin"` netestuje, či majú reťazce rovnaké znaky.

### 5.29.2 Vysvetlenie chyby

V porovnaní sa pole `input` konvertuje na smerník na svoj prvý prvok. Reťazcový literál `"admin"` je iný objekt v pamäti. Operátor `==` preto porovnáva adresy dvoch objektov.

Na porovnanie obsahu reťazcov treba použiť `strcmp`.

### 5.29.3 Opravený kód

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char input[32];

    if (fgets(input, sizeof(input), stdin) == NULL) {
        return 1;
    }
    input[strcspn(input, "\n")] = '\0';

    if (strcmp(input, "admin") == 0) {
        printf("OK\n");
    } else {
        printf("DENIED\n");
    }

    return 0;
}
```

## 5.30 Pridanie znaku pomocou strcat

### 5.30.1 Očakávaná odpoveď

Program volá `strcat` s nesprávnym druhým argumentom. Funkcia `strcat` očakáva ukazovateľ na nulou ukončený reťazec, nie jeden znak typu `int` alebo `char`.

### 5.30.2 Vysvetlenie chyby

Znak `'!'` nie je reťazec. Ak chceme použiť `strcat`, druhý argument musí byť napríklad `!"`. Zároveň musí byť v cieľovom poli dostatok miesta.

### 5.30.3 Opravený kód

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char text[16] = "hello";

    strcat(text, "!");
    printf("%s\n", text);
    return 0;
}
```

Alternatívne možno znak pridať ručne:

```
size_t len = strlen(text);
if (len + 1 < sizeof(text)) {
    text[len] = '!';
    text[len + 1] = '\0';
}
```

## 5.31 Bonus: Úprava reťazcového literálu

### 5.31.1 Očakávaná odpoveď

Program sa pokúša zapisovať do reťazcového literálu. Úprava literálu má nedefinované správanie. Na mnohých systémoch sú literály uložené v pamäti určenej iba na čítanie, takže zápis spôsobí pád programu.

### 5.31.2 Vysvetlenie chyby

Deklarácia `char *message = "hello, world";` vytvorí smerník na literál. Samotný smerník je možné presmerovať, ale obsah literálu sa nesmie meniť. To, že typ historicky nie je `const char *`, neznamená, že zápis je bezpečný.

### 5.31.3 Opravený kód

```
#include <stdio.h>

int main(void) {
    char message[] = "hello, world";

    message[0] = 'H';
    puts(message);
    return 0;
}
```

## 5.32 Bonus: Nezarovnaný prístup do pamäte

### 5.32.1 Očakávaná odpoveď

Program robí potenciálne nezarovnaný prístup do pamäte. Po prečítaní prvého bajtu `tag` ukazuje `p` na adresu `buf + 1`. Tá nemusí byť správne zarovnaná pre typ `uint32_t`.

### 5.32.2 Vysvetlenie chyby

Niektoré architektúry vyžadujú, aby viacbajtové typy boli uložené na adresách zarovnaných podľa ich veľkosti. Výraz `*(uint32_t *)p` môže preto spadnúť alebo mať nedefinované správanie. Navyše takýto kód ignoruje endianitu formátu súboru.

Bezpečnejší postup je čítať hodnoty po bajtoch podľa definovaného formátu.

### 5.32.3 Opravený kód

```
#include <stdio.h>
#include <stdint.h>

struct record {
    char tag;
    uint32_t id;
    uint16_t flags;
};

int read_record(FILE *fp, struct record *out) {
    unsigned char buf[7];

    if (fread(buf, sizeof(buf), 1, fp) != 1) {
        return 0;
    }

    out->tag = (char)buf[0];
    out->id = ((uint32_t)buf[1] << 24) |
              ((uint32_t)buf[2] << 16) |
              ((uint32_t)buf[3] << 8) |
              (uint32_t)buf[4];
    out->flags = ((uint16_t)buf[5] << 8) |
                (uint16_t)buf[6];
    return 1;
}
```

Tento príklad predpokladá, že binárny formát ukladá čísla v poradí big-endian. Ak má formát použiť little-endian, poskladanie bajtov treba obrátiť.