Ficha 3

Programação (L.EIC009)

Objectivos

- Depuração de programas usando o gdb.
- Manipulação de memória alocada dinamicamente.
- Uso de Address Sanitizer (ASan), e Undefined Behavior Sanitizer (UBSan).

Recursos

Slides das aulas teóricas: HTML PDF

1

Siga as instruções do guião "Uma pequena introdução ao gdb" para se familiarizar com o gdb.

Uso de sanitizadores e ficheiro `Makefile`

Para maior conveniência na compilação dos restantes exercícios desta aula, edite um ficheiro chamado `Makefile` com o seguinte conteúdo inicial.

```
CC=gcc
CFLAGS=-Wall -g -fsanitize=address -fsanitize=undefined -fno-omit-frame-poi
```

Pode depois compilar um programa em `prog.c` usando o comando `make`, por ex.

```
$ make prog
gcc -Wall -g --std=c99 -fsanitize=address -fsanitize=undefined    prog.c
```

A intenção é habilitar o uso dos sanitizadores ASan e UBSan. Inspecione as mensagens de erro que obtiver para perceber os erros nos seus programas. Será bom também empregar o gdb para debugging.

2

Considere o problema de calcular a mediana de números dados. Escreva um programa para efeito que:

1. leia um número inteiro `n` positivo;
2. aloque um array `a` de tipo `int` de tamanho `n` usando `malloc`;
3. leia `n` valores de tipo `int` preenchendo o array `a`;
4. calcule de seguida a mediana dos valores lidos:
   - ordene primeiro `a` usando a função `qsort` como exemplificado abaixo;
   - se `n` é impar então a mediana é dada por `a[n / 2]` (após a ordenação);
   - se `n` é par, a mediana é dada pela média dos valores `a[n / 2 - 1]` e `a[n / 2]`.
5. liberte no final do programa o array `a` alocado usando `free`.

Esqueleto

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

int compare_int(const void* v1, const void* v2) {
  return (* (int *) v1) - (* (int*) v2);
}

void sort_int_array(int a[], int n) {
  qsort(a, n, sizeof(int), &compare_int );
}

int main(void) {
  int n;
  printf("How many numbers: ");
  scanf("%d", &n);

  int *a = ...;
  ...
  return 0;
}
```

Exemplos de execução

(`n` é impar)

```
How many numbers: 5
Enter values: 10 11 10 14 16
Median: 11.000000
```

(`n` é par)

```
How many numbers: 6
Enter values: 10 11 10 14 16 12
Median: 11.500000
```

3

Reformule o programa anterior de forma a que *_não seja dado* o valor de `n` inicialmente e que a sequência de inteiros seja implicitamente terminada com o valor `0`.

O programa deve manter um array do tipo `int`, inicialmente alocado com `malloc` e de tamanho 4, e que cresce dinamicamente com `realloc` para o dobro do tamanho quando necessário.

Exemplos de execução

(`n` é impar)

```
Enter values: 10 11 10 14 16 0
Median: 11.000000
```

(`n` é par)

```
Enter values:
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
```

```
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
10 11 10 14 16 12
0
Median: 11.500000
```

4

Considere a implementação de um vector de inteiros, com esqueleto dado abaixo para o tipo `ivector` e seguintes funções associadas:

Função	Descrição
`v_create(n)`	Aloca um `ivector` com capacidade inicial `n`
`v_size(v)`	Devolve número de elementos no vector (apenas posições realmente usadas).
`v_data(v)`	Devolve apontador para o array interno onde estão guardados os elementos de `v`.
`v_destroy(v)`	Liberta memória associada a `v` e ao seu array interno.
`v_add(v, x)`	Adiciona `x` ao vector `v` na última posição, realocando se necessário o array interno usado por `v`.

O código já é dado para as primeiras 3 operações. Defina convenientemente `v_destroy` e `v_add`. De seguida adapte o programa do exercício anterior para usar o tipo `ivector` em vez de um array alocado directamente.

```c
typedef struct {
  int* data; // Elementos
  int capacity; // Capacidade do array
  int size; // Nº de elementos (posições ocupadas)
} ivector;
```

```
ivector* v_create(int initial_capacity) {
  ivector* v = malloc(sizeof(ivector));
  v -> data = malloc(initial_capacity * sizeof(int));
  v -> capacity = initial_capacity;
  v -> size = 0;
  return v;
}

int v_size(ivector* v) {
  return v -> size;
}

int* v_data(ivector* v) {
  return v -> data;
}

void v_destroy(ivector* v) {
  ...
}

void v_add(ivector* v, int x) {
  ...
}
```

5

Considere o seguinte esqueleto para um programa C com funções auxiliares `copy` e `print` como se segue:

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

void copy(char dst[], const char *src) {
  char* p = dst;
  const char* q = src;
  while (*q != '\0') {
    *p = *q;
    p++;
    q++;
  }
  *p = 0;
}
void print(const char s[]) {
  const char* p = s;
  while (*p != '\0') {
```

```
    putchar(*p);
    p++;
  }
  putchar('\n');
}
int main(void) {
  // ... CORPO de main ...
}
```

Considere a inclusão de cada um dos seguintes fragmentos no corpo de `main`. Para cada versão, compile o programa habilitando o uso de "sanitizers" como descrito acima, e de seguida execute-o. Interprete os erros obtidos que podem ser de vários tipos:

- "memory leaks"
- "dangling references"
- "buffer overflows"
- ...

Para tal inspecione os erros reportados pelos "sanitizers". Para perceber melhor os erros pode também empregar o gdb.

(a)

```
char s[5];
copy(s, "ABCDE");
print(s);
```

(b)

```
char* s = malloc(6);
copy(s, "ABCDE");
print(s);
```

(c)

```
char* s = malloc(6);
copy(s, "ABCDE");
free(s);
print(s);
```

(d)

```
char* s = malloc(6);
copy(s, "ABCDE");
print(s);
free(s);
free(s);
```

(e)

```
char* s = malloc(6);
copy(s, "ABCDE");
s = realloc(s, 10);
copy(s + 5, "12345");
print(s);
free(s);
```

(f)

```
char c = '\0';
char* s = malloc(6);
copy(s, "ABCDE");
s[0] = s[0] / c;
print(s);
free(s);
```

(g)

```
char* s = malloc(6);
char *ps[] = { s, "ABCDE", 0};
copy(ps[0], ps[2]);
print(s);
free(s);
```

(h)

```
char* s = malloc(6);
char *ps[] = { s, "ABCDE", 0};
ps[2]++;
copy(ps[0], ps[2]);
print(s);
free(s);
```

6

Considere uma lista duplamente ligada de valores inteiros com o esqueleto de implementação dado abaixo para o tipo `ilist` e as seguintes funções associadas.

Função	Descrição
`l_create()`	Aloca uma nova lista.
`l_destroy(l)`	Liberta memória associada a `l` e seus elementos
`l_size(l)`	Devolve número de elementos em `l`.
`l_print(l)`	Imprime conteúdo de `l` (útil para debugging).
`l_add_first(l, v)`	Adiciona `v` ao início de `l`.
`l_add_last(l, v)`	Adiciona `v` ao fim de `l`.
`l_remove_first(l, pv)`	Se `l` não vazia: remove primeiro elemento de `l`, devolvendo resultado em `pv`, e retorna `true`. Caso contrário retorna `false`.
`l_remove_last(l, pv)`	Se `l` não vazia: remove último elemento de `l`, devolvendo resultado em `pv`, e retorna `true`. Caso contrário retorna `false`.

O código abaixo implementa todas as funções com excepção de `l_add_last`, `l_remove_first` e `l_remove_last`. Implemente as restantes. É codificado em `main` um pequeno teste que poderá adaptar.

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

struct _node {
  int value;
  struct _node *prev;
  struct _node *next;
};
typedef struct _node node;
```

```c
typedef struct {
  int size;
  node* first;
  node* last;
} ilist;

ilist* l_create(void) {
  ilist* l = malloc(sizeof(ilist));
  l -> size = 0;
  l -> first = NULL;
  l -> last = NULL;
  return l;
}

void l_destroy(ilist* l) {
  node *n = l -> first, *n2;
  while (n != NULL) {
    n2 = n -> next;
    free(n2);
    n = n2;
  }
  free(l);
}

int l_size(ilist *l) {
  return l -> size;
}

void l_print(const ilist* l) {
  node* n = l -> first;
  printf("-- (%d elements) --\n", l -> size);
  while (n != NULL) {
    printf("[%p] value: %d prev: %p next: %p\n",
      n,
      n -> value,
      n -> prev,
      n -> next);
    n = n -> next;
  }
  printf("-\n");
}

void l_add_first(ilist* l, int value) {
  node* new_node = calloc(1, sizeof(node));
  new_node -> value = value;
  new_node -> prev = l -> last;
  if (l -> size == 0) {
```

```
    l -> first = l -> last = new_node;
  } else {
    new_node -> next = l -> first;
    l -> first -> prev = new_node;
    l -> first = new_node;
  }
  l -> size ++;
}

void l_add_last(ilist* l, int value) {
  // ...
}

bool l_remove_first(ilist* l, int* pvalue) {
  // ...
}

bool l_remove_last(ilist* l, int* pvalue) {
  // ...
}

int main(void) {
  const int n = 5;
  ilist* l = l_create();
  l_print(l);
  for (int i = 0; i < n; i++) {
    int v = i * 100;
    l_add_last(l, v);
    // ilist_print(l);
    printf("Value added: %d\n", v);
  }
  for (int i = 0; i < n; i++) {
    int v;
    l_remove_first(l, &v);
    // ilist_print(l);
    printf("Value removed: %d\n", v);
  }
  l_destroy(l);
}
```