Universidade Federal de Minas Gerais
Ciência da Computação

Linguagens de Programação - Haniel Barbosa

Lista de Exercícios 3

Data de Entrega: 21/02/2021

1. Quais componentes de um programa devem ser armazenados na memória?

   (a) As variáveis, funções e seus resultados

   (b) O programa em si e os estados que ele mantém

   (c) O estado e os resultados de funções

   (d) As variáveis e suas atribuições

2. Marque V ou F para as alternativas que definem as características de cada tipo de memória em C:

   (a) Memória Estática tem gerenciamento automático

   (b) Memória dinâmica é representada como uma pilha com gerenciamento manual

   (c) Memória dinâmica é representada como uma heap com gerenciamento automático

   (d) Memória Dinâmica não é flexível

   (e) Memória dinâmica tem gerenciamento mais complexo do que memória estática

   (f) Memória estática armazena variáveis globais

   (g) Memória Dinâmica é representada como uma pilha com gerenciamento automático

   (h) Memória dinâmica é representada como uma heap com gerenciamento manual

3. Considerando os diferentes tipos de memória em C:

   (a) classifique qual tipo de memória está sendo utilizada para cada variável do programa em C apresentado a seguir; e

   (b) apresente quais valores serão gerados para a lista construída ao fim da execução.

```c
int valor_inicial = 10;

int valor_intermediario = 5;

void calcula(int* valores){
  int taxa = 3;
  valores[0] = valor_inicial + valor_intermediario * taxa;
  valores[1] = valores[0] * 3;
}
```

```
10
11 int main(){
12   int* valores = (int*)malloc(3 * sizeof(int));
13   calcula(valores);
14   valores[2] = valor_inicial + valor_intermediario;
15 }
```

4. Cite três exemplos de coletores de lixos que podem ser utilizados por uma linguagem de programação. Apresente um exemplo de aplicação para o qual um dos modelos, à sua escolha, é o mais adequado e justifique.

5. Considere uma heap e seu administrador abaixo, a classe *HeapManager*, em Python. Ela utiliza uma estratégia *first-fit* para encontrar o primeiro bloco de memória grande o suficiente para alocar uma requisição. Porém ela possuí apenas a opção para alocamento de espaço na memória. Você deverá implementar a função *deallocate()*, para assim finalizar as funcionalidades de gerenciamento desta heap.

```
1  NULL = -1 # The null link
2
3  class HeapManager:
4    """Implements a very simple heap manager."""
5
6    def __init__(self, initialMemory):
7      """Constructor. Parameter initialMemory is the array of
           data that we will
8        use to represent the memory."""
9      self.memory = initialMemory
10     self.memory[0] = self.memory.__len__()
11     self.memory[1] = NULL
12     self.freeStart = 0
13
14
15   def allocate(self, requestSize):
16     """Allocates a block of data, and return its address. The
           parameter
17       requestSize is the amount of space that must be allocaed
             ."""
18     size = requestSize + 1
19     # Do first-fit search: linear search of the free list for
           the first block
20     # of sufficient size.
21     p = self.freeStart
22     lag = NULL
23     while p != NULL and self.memory[p] < size:
24       lag = p
25       p = self.memory[p + 1]
26     if p == NULL:
27       raise MemoryError()
```

```
28      nextFree = self.memory[p + 1]
29      # Now p is the index of a block of sufficient size,
30      # lag is the index of p's predecessor in the
31      # free list, or NULL, and nextFree is the index of
32      # p's successor in the free list, or NULL.
33      # If the block has more space than we need, carve
34      # out what we need from the front and return the
35      # unused end part to the free list.
36      unused = self.memory[p] - size
37      if unused > 1:
38        nextFree = p + size
39        self.memory[nextFree] = unused
40        self.memory[nextFree + 1] = self.memory[p + 1]
41        self.memory[p] = size
42      if lag == NULL:
43        self.freeStart = nextFree
44      else:
45        self.memory[lag + 1] = nextFree
46      return p + 1
47
48
49  def test():
50    h = HeapManager([0 for x in range(0, 10)])
51    a = h.allocate(4)
52    print("a = ", a, ", Memory = ", h.memory)
53    b = h.allocate(2)
54    print("b = ", b, ", Memory = ", h.memory)
55
56  test()
```

6. Outra estratégia para gerenciamento de uma heap é o *best-fit*. Esta estratégia consiste em percorrer a lista de blocos livres em busca do pedaço de memória que seja o menor possível mas que seja grande o suficiente para comportar a área de memória requisitada. Se for encontrada uma área exatamente do tamanho da requisição, então pode-se interromper a busca, retornando a área encontrada. Do contrário, toda a lista deve ser percorrida, em busca do melhor pedaço de memória. A vantagem de *best-fit* é que esta estratégia não quebra áreas de memória muito grandes desnecessariamente. Se houver uma área de tamanho exato, *best-fit* a encontrará, não tendo, portanto, de quebrar nenhum bloco neste caso. Sendo assim, você deve implementar uma nova versão da classe *HeapManager* que utilize esta política de alocação de memória. Comece com uma cópia de *HeapManager* e então modifique o método *allocate* para implementar esta estratégia.

7. Muitas linguagens de programação não possuem qualquer mecanismo de coleta automática de lixo. Um exemplo típico é C++. Ainda assim, é possível programar de forma mais segura via bibliotecas. Uma estratégia comumente adotada em C++

é baseada no uso de ponteiros desalocados automaticamente. Uma possível implementação deste tipo de ponteiro é dada logo abaixo:

```
template <class T> class auto_ptr {
  private: T* ptr;
  public:
    explicit auto_ptr(T* p = 0) : ptr(p) { }
    ~auto_ptr() { delete ptr; }
    T& operator*() { return *ptr; }
    T* operator->() { return ptr; }
};
```

(a) A classe *auto_ptr* utiliza pelo menos dois tipos diferentes de polimorfismo. Que tipos de polimorfismos são estes?

(b) A função abaixo contém um problema de memória ou não? Em caso afirmativo, explique que falha é esta. Utilize a ferramenta *valgrind* para analisar este programa, por exemplo, tentando o comando valgrind -v ./a.out. Considere que uma falha de memória leva *valgrind* a fornecer algum aviso. Caso o erro não exista, justifique a sua resposta:

```
void foo0() {
  auto_ptr<std::string> p(new std::string("I did one P.O.F!\
     n"));
  std::cout << *p;
}
```

(c) Novamente: problema de memória ou não? Em caso afirmativo, explique que falha é esta. Em caso negativo, justifique. Note que exceções, neste caso, funcionam como em Java ou Python:

```
void foo1() {
  try {
    auto_ptr<std::string> p(new std::string("Oi!\n"));
    throw 20;
  } catch (int e) { std::cout << "Oops: " << e << "\n"; }
}
```

(d) Última pergunta: problema de memória ou não? Em caso afirmativo, explique que falha é esta. Em caso negativo, justifique a sua resposta:

```
void foo2() {
  try {
    std::string* p = new std::string("Oi!\n");
    throw 20;
    delete p;
  } catch (int e) { std::cout << "Oops: " << e << "\n"; }
}
```