UFMG

Linguagens de Programação - Lista de Exercícios 3

Arthur Pontes Nader - 2019022294
Belo Horizonte - MG - Brasil

1) Os componentes de um programa que devem ser armazenados na memória são o programa em si e os estados que ele mantém, portanto:

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

2)
   (a) V
   (b) F
   (c) F
   (d) F
   (e) V
   (f) V
   (g) V
   (h) V

3)
   (a)

Variável	Tipo de memória
valor_inicial	Estática
valor_intermediario	Estática
valores	Dinâmica
taxa	Dinâmica

   (b) valores[0] = valor_inicial + valor_intermediario * taxa = 10 + 5 * 3 = 25
   valores[1] = valores[0] * 3 = 25 * 3 = 75
   valores[2] = valor_inicial + valor_intermediario = 10 + 5 = 15

4)

- Mark and sweep: nesse método, blocos relevantes são marcados e removem-se blocos que não estão no conjunto de interesse. Por ser necessário um tempo de parada longo para marcação dos blocos, esse método parece ser ideal para uso em sistemas que não operam continuamente e em que requisições e liberações de memórias não são muito frequentes, como é o caso de sistemas presentes em supermercados, farmácias, etc.

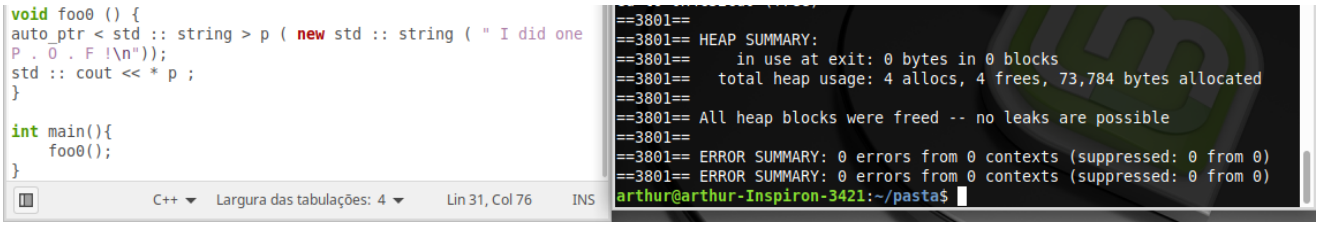
- Copying collector: nesse coletor, a memória consiste em duas partes, sendo que uma parte é usada de cada vez. Quando se completa uma, copia-se apenas blocos relevantes para a outra parte e se remove os blocos da primeira. Essa abordagem parece interessante para sistemas em que há grande volume de transferência de dados e em que é necessário alocar e liberar memória continuamente, como é o caso de servidores e provedores que oferecem serviços e produtos online.

- Reference counting: já nesse método, todo novo bloco armazenado é associado a um contador, sendo que esse contador determina o número de ponteiros para o bloco. Quando o contador chega a zero, o bloco é liberado da memória. Esse método parece ser muito relevante para sistemas em que não se pode ter um tempo de parada muito grande para realizar a coleta de lixo, como por exemplo, sistemas de tempo real de satélites e de equipamentos hospitalares.

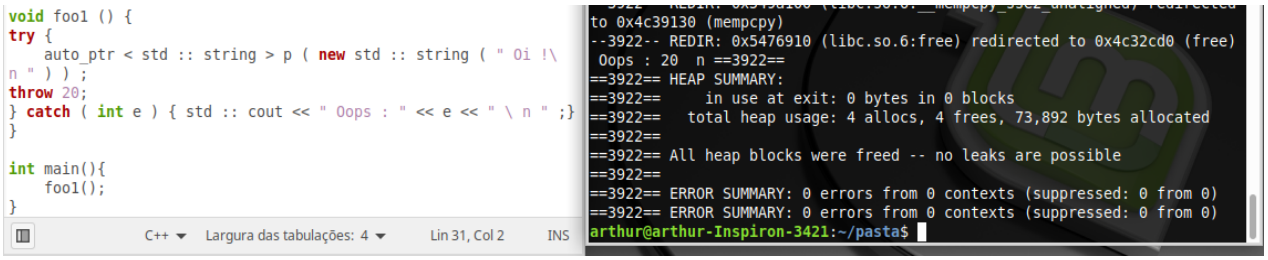
7)

(a) Os dois tipos de poliformismo usados são: sobrecarga de operadores e polimorfismo paramétrico.

(b) A função não contém problema de memória, porque a memória alocada é liberada pela chamada do destrutor ao final da execução.



(c) Essa função também não contém problema de memória, pois novamente, a memória alocada é liberada pela chamada do destrutor após a execução.

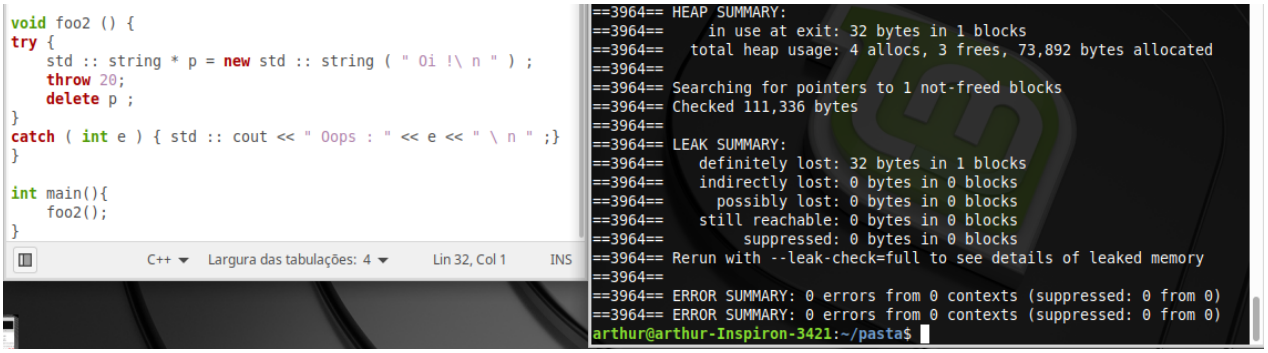


```
void foo1 () {
try {
    auto_ptr < std :: string > p ( new std :: string ( " Oi !\
n " ) ) ;
throw 20;
} catch ( int e ) { std :: cout << " Oops : " << e << " \ n " ;}
}

int main(){
    foo1();
}
```

```
to 0x4c39130 (mempcpy)
--3922-- REDIR: 0x5476910 (libc.so.6:free) redirected to 0x4c32cd0 (free)
 Oops : 20  n ==3922==
==3922== HEAP SUMMARY:
==3922==     in use at exit: 0 bytes in 0 blocks
==3922==   total heap usage: 4 allocs, 4 frees, 73,892 bytes allocated
==3922==
==3922== All heap blocks were freed -- no leaks are possible
==3922==
==3922== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==3922== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
arthur@arthur-Inspiron-3421:~/pasta$
```

(d) Já nessa função ocorre problema de memória, pois durante a execução, perde-se a referência para a memória alocada. Assim ocorre um vazamento de memória, pois não será possível liberá-la posteriormente.



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

int main(){
    foo2();
}
```

```
==3964== HEAP SUMMARY:
==3964==     in use at exit: 32 bytes in 1 blocks
==3964==   total heap usage: 4 allocs, 3 frees, 73,892 bytes allocated
==3964==
==3964== Searching for pointers to 1 not-freed blocks
==3964== Checked 111,336 bytes
==3964==
==3964== LEAK SUMMARY:
==3964==    definitely lost: 32 bytes in 1 blocks
==3964==    indirectly lost: 0 bytes in 0 blocks
==3964==      possibly lost: 0 bytes in 0 blocks
==3964==    still reachable: 0 bytes in 0 blocks
==3964==         suppressed: 0 bytes in 0 blocks
==3964== Rerun with --leak-check=full to see details of leaked memory
==3964==
==3964== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==3964== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
arthur@arthur-Inspiron-3421:~/pasta$
```

11)

(a) Deve ser garantido que os objetos passados para o método possuam definições para os métodos “isNotEmpty” e “remove” que são acionados dentro da função.

(b) A expressão *duck typing* significa que, se um objeto age como um pato, então ele é um pato. Mais especificamente, é um estilo de tipagem em que, ao invés de ser o tipo do objeto, são os atributos e métodos do objeto que definem se a semântica do programa é válida. A relação de *duck typing* com esse método é que o objeto passado como argumento para o método pode ser de qualquer tipo, desde de que possua um comportamento bem definido para os métodos “isNotEmpty” e “remove”.

12)

Linha 1: será impresso “Pavao is an animal”
Linha 2: será impresso “Tigre is a mammal”
Linha 3: será impresso “Krypto is a dog”
Linha 4: será impresso “Pavao, which is an animal, is eating."
Linha 5: será impresso “Tigre, which is a mammal, is sucking milk."
Linha 6: será impresso “Tigre, which is an animal, is eating."

Linha 7: será impresso “Krypto is barking rather loudly."
Linha 8: será impresso “Krypto, which is a mammal, is sucking milk."
Linha 9: será impresso “Krypto barks when it eats.”
Linha 10: um erro será produzido, uma vez que objetos da classe Animal não possuem uma definição para o método “bark”
Linha 11: será impresso “Krypto is barking rather loudly ."

13) O problema do diamante é um tipo de ambiguidade que ocorre quando se usa herança múltipla em um determinado contexto de programação orientada a objetos. O problema ocorre quando se tem a seguinte situação: suponha que duas classes C2 e C3 herdam de uma classe C1 que possui um determinado método, sendo que tanto C2 e C3 tem suas próprias implementações para esse mesmo método. Dessa forma, se tivermos uma classe C4 que herda de C2 e C3 e não tem uma implementação para o método em questão, fica ambíguo se C4 irá herdar o método de C2 ou C3.

15)

```
class FormulaError(Exception):

    def __init__(self, alerta):
        self.mensagem = alerta

    def __str__(self):
        return repr(self.mensagem)

conta = input("Digite a operacao: ")
termos = conta.split(" ")
value_erro = False

if len(termos) != 3:
    raise FormulaError("A entrada não consiste de 3 elementos")
else:
    try:
        operando1 = float(termos[0])
        operando2 = float(termos[2])
    except ValueError:
        value_erro = True

    if value_erro:
            raise FormulaError("O primeiro e o terceiro valor de entrada devem ser numeros")
```

```
if termos[1] == "+":
    print(operando1 + operando2)
elif termos[1] == "-":
    print(operando1 - operando2)
elif termos[1] == "*":
    print(operando1 * operando2)
elif termos[1] == "/":
    print(operando1 / operando2)
else:
    raise FormulaError("%s não é um operador válido" %termos[1])
```