

Linguagens de Programação

Professor Haniel Barbosa

Lista de exercícios 4

Luís Felipe Ramos Ferreira
2019022553

June 20, 2023

1 Passagem de Parâmetros

1. (a) O valor impresso pelo programa é 5
- (b) O valor impresso nesse caso é 7. Como *C++* possui escopo estático, o valor da variável *x* utilizado dentro da função *p* será o definido globalmente, isto é, onde *x* é igual a zero. O variável *x* dentro do escopo da função *main* terá sua referência passada para os dois parâmetros da função *p*. Dentro do escopo da função, o valor da variável global *x* será incrementada para 1, e as variáveis *y* e *x* serão incrementadas. Como elas são referências para a variável *x* com valor igual a 1 no escopo da função *main*, esse mesmo valor será incrementado 2 vezes. Podemos entender *x* e *y* como duas variáveis que apontam para a mesma posição da memória. Dessa maneira, o valor nessa posição da memória será incrementado para 3, e ele será somado duas vezes no comando de impressão. Dessa forma, teremos uma saída com valor 1 + 3 + 3, ou então 7.

2. (a)

```
1      #define SUM(X, Y) (X) + (Y)
2      int main(int argc, char** argv){
3          printf("sum = %d\n", {(argc) + (argv[0][0])});
4      }
5
```

- (b) A captura de variáveis é um problema que ocorre na expansão de macros quando dentro da definição da macro há a definição de uma variável cujo nome está sendo utilizado por outra variável passada como parâmetro para a macro, causando um conflito entre a qual variável cada operação se refere. Um exemplo da situação apresentado em sala foi o do uso da macro *SWAP*, que define uma função para a troca de valores entre duas variáveis, e que dentro do escopo da macro define uma variável auxiliar denominada *tmp*. Abaixo, o código disponibilizado nas notas de aula pode ser visto:

```
1      #include "stdio.h"
2      #define SWAP(X, Y) \
3          { \
4              int tmp = X; \
5              X = Y; \
6              Y = tmp; \
7          }
8      int main()
9      {
10         int a = 2;
11         int tmp = 15;
12         printf("Before: %d, %d\n", a, tmp);
13         SWAP(a, tmp);
14         printf("After: %d, %d\n", a, tmp);
15     }
16
```

Neste cenário, após o pré processamento, o código da maneira a seguir. Nela, podemos ver que há a captura da variável `tmp`, e o comportamento não é o desejado, uma vez que a variável auxiliar `tmp` que é definida dentro da macro encobre e faz com que a variável `tmp` original, passada como parâmetro, se perca, e assim o *swap* de variáveis não ocorre como esperado.

```

1      #include "stdio.h"
2      #define SWAP(X, Y) \
3      { \
4          int tmp = X; \
5          X = Y; \
6          Y = tmp; \
7      }
8      int main()
9      {
10         int a = 2;
11         int tmp = 15;
12         printf("Before: %d, %d\n", a, tmp);
13         {
14             int tmp = a;
15             a = tmp;
16             tmp = tmp;
17         }
18         printf("After: %d, %d\n", a, tmp);
19     }
20

```

- (c) O código em *C* abaixo contém o problema da múltipla avaliação de parâmetros. Ele foi apresentado e discutido durante as aulas da disciplina.

```

1      #include "stdio.h"
2      int x = 0;
3      int foo()
4      {
5          x++;
6          return 1;
7      }
8      #define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
9      int main()
10     {
11         int y = MAX(0, foo());
12         printf("Max: %d, global x: %d\n", y, x);
13     }
14

```

Após a expansão da macro, como os argumentos passados irão ser substituídos no corpo da macro, a função `foo` será chamada duas vezes, o que irá impactar no incremento da variável global `x`. A saída do código a seguir seria então:

Max: 1, global x: 2

3. (a) O valor de `z` na chamada é 30
- (b) O valor
4. (a) `m1.i = 4` e `m2.i = 4`
- (b) `m1.i = 3` e `m2.i = 3`
- (c) `m1.i = 4` e `m2.i = 4`
- (d) Java adota o tipo de passagem por valor para tipos primitivos
- (e) Java adota o tipo de passagem por valor para objetos

2 Programação Lógica

1. As implementações para os predicados *firstCousin* e *descendant* estão abaixo dos comentários que indicam a letra respectiva à cada um.

```
1      parent(kim,holly).
2      parent(margaret,kim).
3      parent(margaret,kent).
4      parent(esther,margaret).
5      parent(herbert,margaret).
6      parent(herbert,jean).
7      parent(kent, bruce).
8      parent(jean, sam).
9      parent(sam, max).
10     greatGrandParent(GGP,GGC) :- parent(GGP,GP), parent(GP,P),
        parent(P,GGC).
11     sibling(X,Y) :- parent(P,X), parent(P, Y), not(X=Y).
12
13     % a
14     firstCousin(X, Y) :- sibling(M, N), parent(M, X), parent(N, Y
        ), not(X=Y), not(sibling(X, Y)).
15
16     % b
17     descendant(X,Y) :- parent(Y, X); (parent(Y, K), descendant(X,
        K)).
```

2. Implementação do predicado *third*.

```
1      % fact
2      thirdFact([_, _, X|_], X).
3
4      % predicate
5      third([_, _, X|_], Y) :- X=Y.
```

3. Implementação do predicado *dupList*.

```
1      dupList([], []).
2      dupList([H|T], [G|[G|K]]) :- dupList(T, K), H=G.
```

4. Implementação do predicado *isEqual*, com o uso do predicado da biblioteca padrão de manipulação de listas em *Prolog*.

```
1      isEqual([], []).
2      isEqual(X, Y) :- subset(X, Y), subset(Y, X).
```

5. Implementação do predicado *isDifference*.

```
1      isEqual([], []).
2      isEqual(X, Y) :- subset(X, Y), subset(Y, X).
```
