Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
IMD0030 – Linguagem de Programação I

Docente: Umberto S. Costa

Problema: desenvolvimento de habilidades de programação na linguagem C++.

Subproblema 2: recursividade, passagem de parâmetros, sobrecarga e templates de funções.

Produto do subproblema: (*i*) resumo das principais características e recursos C++ identificados durante a exploração das questões deste subproblema (até duas páginas, podendo haver apêndices); (*ii*) respostas às questões abaixo; e (*iii*) código-fonte dos programas implementados.

Data de entrega via SIGAA: 17 de agosto de 2017.

Instruções: neste problema o aluno deve consultar as referências indicadas pelo docente para se familiarizar com os recursos necessários à criação de programas simples em C++, sem prejuízo à consulta de outras fontes como manuais e tutoriais. Usar as questões e programas mostrados a seguir como guia para as discussões em grupo e para orientar a exploração da linguagem C++. Para facilitar o aprendizado, recomenda-se que o aluno compare os recursos e conceitos de C++ com seu conhecimento prévio acerca de outras linguagens de programação. Leia e modifique os códigos mostrados e utilize os conceitos e recursos explorados para a criar os programas solicitados. Recursos exclusivos da linguagem C devem ser ignorados e substituídos por seus correspondentes em C++.

Questões[1]:

1. O que é um subprograma recursivo?
2. Em um programa recursivo devemos considerar o **caso base** e o **passo indutivo**.
   (a) Explique esses conceitos e como eles promovem a solução recursiva.
   (b) Explique o que é uma *recursão infinita* e como evitá-la.
   (c) Considere a relação de recorrência da função fatorial:
   $$fatorial(n) = \begin{cases} 1 & se\ n = 0 \\ n \times fatorial(n-1) & se\ n > 0 \end{cases}$$
   Mostre como esta relação de recorrência pode ser aplicada para calcularmos $fatorial(3)$ e identifique o caso base e seus passos indutivos aplicados.

[1] Em parte inspiradas em *Exploring C++ 11*, Ray Lischner. Alguns programas foram retirados desta mesma fonte.

3. Crie um programa C++ para cálculo do fatorial, assumindo que:
   - O cálculo do fatorial deve ser realizado por meio de uma função recursiva `int fatorial(int)`;
   - O programa deve ser nomeado `fatorial.cpp`;
   - O valor do parâmetro da função fatorial poderá ser informado pelo teclado ou fornecido por linha de comando, quando da invocação do programa. Abaixo, veja exemplos destas duas formas de execução (assumimos que o executável gerado tem o nome `fatorial`):

```
$ ./fatorial
Informe o valor de n: 5
Fatorial(5) = 120
$ ./fatorial 5
Fatorial(5) = 120
```

4. No programa criado no item anterior, adicione uma nova função `int fatorial_verboso(string, int)` para permitir comportamento verboso, conforme os exemplos de execução a seguir:

```
$ ./fatorial
Informe o valor de n: 3
Modo verboso (s/n)? n
Fatorial(3) = 6
$ ./fatorial
Informe o valor de n: 3
Modo verboso (s/n)? s
Fatorial(3) = 3 * fatorial(2)
 = 3 * 2 * fatorial(1)
 = 3 * 2 * 1 * fatorial(0)
 = 3 * 2 * 1 * 1
 = 6
$ ./fatorial 3
Modo verboso (s/n)? n
Fatorial(3) = 6
$ ./fatorial 3
Modo verboso (s/n)? s
Fatorial(3) = 3 * fatorial(2)
 = 3 * 2 * fatorial(1)
 = 3 * 2 * 1 * fatorial(0)
 = 3 * 2 * 1 * 1
 = 6
$ ./fatorial 3 -v
Fatorial(3) = 3 * fatorial(2)
 = 3 * 2 * fatorial(1)
 = 3 * 2 * 1 * fatorial(0)
 = 3 * 2 * 1 * 1
 = 6
```

5. Considere o programa do item anterior. O que acontece quando renomeamos a função `int fatorial_verboso(string, int)` como `int fatorial(string, int)`, mantendo a função `int fatorial(int)` original? Explique o comportamento do programa e o conceito subjacente.

6. Ainda considerando o programa do item 4, combine `int fatorial_verboso(string, int)` e `int fatorial(int)` em uma nova função `int fatorial(bool, string, int)`, eliminando as funções anteriores. O comportamento do global do programa deve permanecer o mesmo.

7. Observe que, até agora, em todas as versões que foram implementadas para o cálculo do fatorial, a efetuação das multiplicações fica suspensa até que a execução chegue ao final da recursão, quando elas podem ser realizadas a partir do valor retornado pelo caso base. Pede-se:

   (a) Utilize o KDbg para verificar a afirmação acerca do cálculo das multiplicações.

   (b) Onde ficam armazenados os operandos enquanto as multiplicações não se realizam?

   (c) Crie uma versão alternativa da função fatorial utilizando *recursão em cauda*, de forma a evitar o comportamento discutido acima. Utilize duas funções `int fatorial(int)` e `int fatorialIter(int, int)` que produzam o seguinte comportamento:

```
fatorial(3) = fatorialIter(3, 1)
  = fatorialIter(2, 3 * 1)
  = fatorialIter(1, 2 * 3)
  = fatorialIter(0, 1 * 6)
  = 6
```

8. Seja a Função de Ackermann:

$$A(m,n) = \begin{cases} n+1 & se\ m = 0 \\ A(m-1,1) & se\ m > 0\ e\ n = 0 \\ A(m-1, A(m,n-1)) & se\ m > 0\ e\ n > 0 \end{cases}$$

   (a) Crie um subprograma para a Função de Ackerman e um programa principal que solicita os dados, executa esta função e imprime seu resultado. Veja o comportamento esperado:

```
$ ./ackermann
Função de Ackermann A(m,n)
Informe o valor de m: 2
Informe o valor de n: 3
A(2, 3) = 9
Continuar (s/n)? x
Continuar (s/n)? s
Função de Ackermann A(m,n)
Informe o valor de m: 3
Informe o valor de n: 1
A(3, 1) = 13
Continuar (s/n)? N
```

   (b) Para testar sua implementação, compare os resultados obtidos com a seguinte tabela:

m\n	0	1	2	3	4
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125

9. Para determinar se um número é par ou ímpar, podemos utilizar as seguintes definições:

$$par(n) = \begin{cases} true & se\ n = 0 \\ impar(n-1) & se\ n > 0 \\ par(-n) & se\ n < 0 \end{cases} \qquad impar(n) = \begin{cases} false & se\ n = 0 \\ par(n-1) & se\ n > 0 \\ impar(-n) & se\ n < 0 \end{cases}$$

   Estas definições nos dizem que:

   - Zero é par, mas não é ímpar;
   - Um número positivo é par (ou ímpar) se seu antecessor é ímpar (ou par);
   - Um número negativo é par (ou ímpar) se o seu oposto for par (ou ímpar).

   Com base nestas definições, crie um programa que informa se um dado número é par ou ímpar:

```
$ ./par
Informe o valor de n: 5
5 eh um numero impar.
Continuar (s/n)? x
Continuar (s/n)? s
Informe o valor de n: 20
20 eh um numero par.
Continuar (s/n)? s
Informe o valor de n: -50
-50 eh um numero par.
Continuar (s/n)? n
```

10. A seqüência 0, 1, 1, 2, 3, 5, 8, 13, 21, ... é conhecida como Seqüência de Fibonacci. Os dois primeiros números são 0 e 1, e os demais são sempre a soma dos dois anteriores. Construa uma função `int fib(int)` que calcule um número de Fibonacci, dada sua posição na seqüência. Uma solução ingênua poderia ser obtida diretamente a partir da definição desta sequência:

$$fib(n) = \begin{cases} 0 & se\ n = 0 \\ 1 & se\ n = 1 \\ fib(n-2) + fib(n-1) & se\ n > 1 \end{cases}$$

    Essa solução é muito ineficiente, pois pode executar várias chamadas repetidas para um mesmo valor de entrada. Por exemplo, `fib(5)` executa `fib(2)` três vezes. Como sugestão para uma solução mais eficiente, crie uma função `void fibDupla(int n, int& a, int& b)` que, dado um número `n > 0`, altera os parâmetros formais `a` e `b`, onde `b` é o número de Fibonacci da posição `n` e `a` é o número de Fibonacci da posição `n-1`. Em sua solução, lembre que:

    - `fib(0) = 0`

- `fib(n)` = valor de `b` resultante da chamada a `fibDupla(n, a, b)`
- `fibDupla(1, a, b)` = resulta em a = `0` e b = `1`
- `fibDupla(n, a, b)` = aplicar recursividade sobre `fibDupla (n-1, a, b)`

11. Considere o programa a seguir:

```cpp
#include <iostream>      // std::cout, std::endl, std::fixed
#include <limits>        // std::numeric_limits
#include <iomanip>       // std::setprecision

using namespace std;

int absval(int x) // calcula o valor absoluto de um inteiro
{
  if (x < 0)
    return -x;
  else
    return x;
}

float absval(float x) // calcula o valor absoluto de um flutuante
{
  if (x < 0)
    return -x;
  else
    return x;
}

long int absval(long int x) // calcula o valor absoluto de um inteiro longo
{
  if (x < 0)
    return -x;
  else
    return x;
}

int main(int argc, char *argv[])
{
  int i{std::numeric_limits<int>::lowest() + 1};
  float f{std::numeric_limits<float>::lowest()};
  long int l{std::numeric_limits<long int>::lowest() + 1};

  std::cout << "abs(" << i << ") = " << absval(i) << endl;
  std::cout << "abs(" << fixed << setprecision(0) << f << ") = "
                      << fixed << setprecision(0) << absval(f) << endl;
  std::cout << "abs(" << l << ") = " << absval(l) << endl;

  return EXIT_SUCCESS;
}
```

lists/absval.cpp

(a) O que há de novo em relação aos espaços de nomes? Você recomendaria esta prática?

(b) O que mudou no formato da inicialização das variáveis (linhas 33 a 35)? Explique.

(c) Por que o programa incrementa os inicializadores das linhas 33 e 35 em uma unidade mas não faz o mesmo na linha 34? O que acontece se retirarmos tais incrementos?

(d) Qual a diferença entre `std::numeric_limits<T>::min()`, visto no Problema 1, e `std::numeric_limits<T>::lowest()`?

(e) O que fazem as funções `fixed` e `setprecision` (linhas 38 e 39)?

(f) Podemos substituir estas três versões da função `absval()` pelo código seguinte?

```
template<class T>
T absval(T x)
{
  if (x < 0)
    return -x;
  else return x;
}
```

(g) Explique o uso e as vantagens do template mostrado no item anterior.

(h) O que acontece quando um código contendo templates é compilado?

(i) O que acontecerá se incluirmos as instruções a seguir no subprograma principal? Explique.

```
std::string s{"texto"};
std::cout << absval(s) << std::endl;
```

12. Considere o programa a seguir:

```
1  template <typename T>
2  T add(T lhs, T rhs)
3  {
4    return lhs(rhs);
5  }
6
7  int main() {}
```

lists/mystery.cpp

(a) Qual é o erro deste programa? Qual o erro relatado pelo compilador?

(b) O que o compilador relata ao incluirmos `return add(0,0);` no subprograma principal?

(c) Preste atenção à linha 1 desta listagem. O que mudou na criação do template da função?

13. Considere o programa a seguir:

```
1  #include <iostream>
2
3  template<class T>
4  T my_min(T a, T b)
5  {
6    if (a < b)
7      return a;
8    else if (b < a)
9      return b;
10   else
```

```
    return a; // a and b are equivalent, so return a
}

int main()
{
        int x{10};
        long y{20};
        std::cout << my_min(x, y) << std::endl;
}
```

lists/my_min.cpp

(a) Este programa poderá ser compilado com sucesso? Explique.

(b) Altera a linha 18 de forma a corrigir o programa da forma mais segura.

14. Considere o programa a seguir:

```
#include <iostream>

template<class T, class U>
U input_sum(std::istream& in)
{
  T x{};
  U sum{0};
  while (in >> x)
    sum += x;
  return sum;
}

int main()
{
  long sum{input_sum<int, long> (std::cin)};
  std::cout << sum << std::endl;
}
```

lists/list4808.cpp

(a) Você pode dizer o que este programa faz sem compilá-lo?

(b) Podemos substituir a linha 6 por `T x();` sem alterar o programa? Explique.

(c) Na linha 15, podemos omitir os tipos do template sem alterar o programa? Explique.

(d) Defina cinco conjuntos de dados (entradas e saídas) para testar este programa.