Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
IMD0030 – Linguagem de Programação I - T3

Docente: Umberto S. Costa

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

Subproblema 7: classes base virtuais, funções e classes amigas, containers e iteradores.

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: 10 de novembro de 2016.

Instruções: neste problema o aluno deve consultar as referências indicadas pelo docente para se familiarizar com os recursos necessários à criação de programas 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. Considere, novamente, a listagem da questão 8 do Problema 6 (listagem `list6601.cpp`). Nessa listagem, cada uma das classes `base1`, `base2` e `base3` tinha sua própria cópia da classe `visible` e precisamos deixar claro ao compilador qual versão do método `msg()` (herdado da classe `visible` por `base1`, `base2` e `base3`) desejávamos executar na classe `derived`, por meio de um qualificador do nome do método a ser utilizado (linha 40). Considere, agora, a listagem `list6602.cpp`:

```
#include <iostream>                                                    1
#include <string>                                                      2
                                                                       3
using namespace std;                                                   4
```

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

```
5
6  class visible {
7  public:
8    visible(string&& msg) : msg_{move(msg)} { cout << msg_ << '\n'; }
9    string const& msg() const { return msg_; }
10 private:
11   string msg_;
12 };
13
14 class base1 : virtual public visible {
15 public:
16   base1(int x) : visible{"base1 constructed"}, value_{x} {}
17   int value() const { return value_; }
18 private:
19   int value_;
20 };
21
22 class base2 : virtual public visible {
23 public:
24   base2(string const& str) : visible{"base2{" + str + "} constructed"} {}
25 };
26
27 class base3 : virtual public visible {
28 public:
29   base3() : visible{"base3 constructed"} {}
30   int value() const { return 42; }
31 };
32
33 class derived : public base1, public base2, public base3 {
34 public:
35   derived(int i, string const& str)
36   : base3{}, base2{str}, base1{i}, visible{"derived"}
37   {}
38   int value() const { return base1::value() + base3::value(); }
39
40   string msg() const{
41     return base1::msg() + "\n" + base2::msg() + "\n" + base3::msg();
42   }
43 };
44
45 int main(){
46   derived d{42, "example"};
47   cout << d.value() << '\n' << d.msg() << '\n';
48 }
```

lists/list6602.cpp

Nesta listagem, a classe base `visible` é virtual nas classes derivadas `base1`, `base2` e `base3` (linhas 14, 22 e 27), fazendo com que uma única instância da classe base comum exista, sendo esta instância comum compartilhada entre as classes derivadas.

(a) Tomando por base a diagrama UML apresentado para a listagem `list6601.cpp`, construa um diagrama UML equivalente para a listagem `list6602.cpp`.

(b) Note que as classes `base1`, `base2` e `base3` passam diferentes valores para o construtor de `visible`. Neste caso, como existirá uma única instância compartilhada de `visible`, apenas o construtor da classe derivada mais profunda será executado. Identifique a mudança que precisou ser implementada na classe `derived` para acomodar tal mudança.

(c) Qual a saída esperada deste programa?

Observação: a maioria das classes base virtuais definem apenas um construtor default, liberando o criador das classes derivadas de passar argumentos para o construtor da classe base virtual. Esta listagem fugiu ao padrão para poder ilustrar como classes base virtuais funcionam.

2. Considere a listagem `bat.cpp` a seguir:

```
#include <iostream>

using namespace std;

class Animal {
 public:
  Animal() {cout << "Animal constructor... Done!" << endl;}
  virtual void eat() {cout << "It eats..." << endl;};
};

class Mammal : public Animal {
 public:
  Mammal() {cout << "Mamal constructor... Done!" << endl;}
  virtual void breathe() {cout << "It breathes..." << endl;};
};

class WingedAnimal : public Animal {
 public:
  WingedAnimal() {cout << "WingedAnimal constructor... Done!" << endl;}
  virtual void flap(){cout << "It flaps..." << endl;};
};

class Bat : public Mammal, public WingedAnimal {
};

int main(){
  Bat bat;
  //Animal &mammal= bat;
 }
```

lists/bat.cpp

(a) Qual o resultado esperado da execução deste programa?

(b) Descomente a linha 28. Explique o erro apontado pelo compilador.

(c) Corrija o problema da linha 28 da listagem `bat.cpp` utilizando cast estático. Salve a nova versão como `bat02.cpp`. Qual o resultado da execução deste programa?

(d) Corrija o problema da linha 28 da listagem `bat.cpp` utilizando a noção de classe base virtual. Salve a nova versão como `bat03.cpp`. Qual o resultado da execução deste programa?

(e) Podemos, na listagem `bat03.cpp`, invocar o método `mammal.flap()`? Por que?

3. Considere a listagem `friend_function.cpp`[2] a seguir:

```
// friend functions
#include <iostream>

using namespace std;

class Rectangle {
private:
  int width, height;
public:
  Rectangle() {}
  Rectangle (int x, int y) : width(x), height(y) {}
  int area() {return width * height;}
friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

lists/friend_function.cpp

Esta listagem ilustra o uso de funções amigas (*friend functions*). Pede-se:

(a) Explique o conceito de funções e classes amigas e sua relação com os modos de acesso de uma classe. Utilize a bibliografia indicada e/ou sites sobre a linguagem C++.

(b) Note que função `duplicate` não é um membro da classe `Rectangle`. Qual o uso típico de funções amigas? Utilize a bibliografia indicada e/ou sites sobre a linguagem C++.

(c) Se comentássemos a linha 13, o compilador acusaria erros. Qual princípio seria quebrado?

4. Considere a listagem `class_function.cpp`2 a seguir:

```
// friend class
#include <iostream>

using namespace std;

class Square;

class Rectangle {
```

[2]Código adaptado de `http://www.cplusplus.com/`.

```
private:
  int width, height;
public:
  int area () {return (width * height);}
  void convert (Square a);
};

class Square {
friend class Rectangle;
private:
  int side;
public:
  Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

lists/friend_class.cpp

(a) Para que serva a linha 6 desta listagem?

(b) Qual a relação entre as linhas 6 e 13?

(c) A relação de amizade definida por C++ é simétrica ou transitiva?

5. Um algoritmo genérico define operações sobre tipos de dados variados, incluindo operações como ordenação, busca, cópia, comparação, entre outras. Muitos algoritmos genéricos são declarados no cabeçalho `<algorithm>`, embora o cabeçalho `<numeric>` contenha algoritmos definidos sobre números. A maioria dos algoritmos genéricos operam com base em iteradores. Iteradores fornecem acesso a uma sequência de valores, elemento a elemento. Um *iterador de leitura* faz referência a uma posição da sequência para permitir a leitura de valores. Neste caso, podemos especificar uma faixa com um par de iteradores, um para o início e outro para o fim de uma sequência. Um *iterador de escrita* faz referência a uma posição onde deseja-se começar a escrever valores. Neste caso, normalmente especifica-se apenas o ponto inicial da sequência, ficando a cargo do programador a identificação de situações de overflow, ou seja, o programador deve verificar se o destino dos valores tem capacidade suficiente. A listagem `list1001.cpp` ilustra o uso de iteradores:

```
#include <algorithm>
#include <cassert>
#include <vector>

```

```
int main(){                                                                    5
  std::vector<int> input{ 10, 20, 30 };                                        6
  std::vector<int> output{};                                                   7
                                                                               8
  output.resize(input.size());                                                 9
  std::copy(input.begin(), input.end(), output.begin());                       10
                                                                               11
  assert(input == output);                                                     12
}                                                                              13
```

lists/list1001.cpp

A função `std::copy` toma dois iteradores de leitura para especificar uma faixa de entrada e um iterador de saída para especificar o começo de uma faixa de saída. Esta função retorna um iterador de saída, a saber, o valor do iterador de saída após a realização da cópia:

```
WriteIterator copy(ReadIterator start, ReadIterator end, WriteIterator result);
```

Certifique-se de entender toda a listagem e observe o uso dos iteradores na linha 10. Pede-se:

(a) O que faz a linha 12?

(b) Comente a linha 10 e recompile o programa. Qual o resultado da execução?

6. Considere a listagem `list0901.cpp`:

```
#include <algorithm>                                                           1
#include <iostream>                                                            2
#include <vector>                                                              3
                                                                               4
int main(){                                                                    5
  std::vector<int> data{};                                                     6
  int x{};                                                                     7
  // initialized to be empty                                                   8
  // Read integers one at a time.                                              9
  while (std::cin >> x)                                                        10
  // Store each integer in the vector.                                         11
  data.push_back(x);                                                           12
  // Sort the vector.                                                          13
  std::sort(data.begin(), data.end());                                         14
  // Print the vector, one number per line.                                    15
  for (std::vector<int>::size_type i{0}, end{data.size()}; i != end; ++i)      16
    std::cout << data.at(i) << '\n';                                           17
}                                                                              18
```

lists/list0901.cpp

(a) Reescreva as linhas 16 e 17 para usar iteradores. Salve o novo código como `list0901v2.cpp`.

(b) Substitua as linhas 16 e 17 da listagem `list0901.cpp` pelas seguintes instruções:

```
for (int element : data)
  std::cout << element << '\n';
```

Este estilo de laço é chamado de *range-based* ou *for-each*. Salve o novo código como `list0901v3.cpp` e teste-o.

7. Considere a listagem `list1003.cpp`:

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main(){
  std::vector<int> data{};

  // Read integers one at a time.
  std::copy(std::istream_iterator<int>(std::cin),
            std::istream_iterator<int>(),
            std::back_inserter(data));

  // Sort the vector.
  std::sort(data.begin(), data.end());

  // Print the vector, one number per line.
  std::copy(data.begin(),
            data.end(),
            std::ostream_iterator<int>(std::cout, "\n"));
}
```

lists/list1003.cpp

   (a) Certifique-se de entender o código em detalhes. Consulte o site http://www.cplusplus.com.

   (b) Usando um laço *range-based*, escreva um programa para ler três inteiros da entrada padrão e armazená-los em um vector. Depois, imprima cada valor seguido de seu dobro e seu quadrado, uma linha por valor de entrada, três valores por linha. Alinhe as colunas usando preenchimento e largura de campo (`std::cout.fill(' ')` e `std::setw(4)`). Salve seu programa como `list1004.cpp`. Veja o exemplo de execução:

```
$ ./list1004
1
2
3
fim
1   2   1
2   4   4
3   6   9
```

   (c) Usando iteradores, refaça o item anterior. Salve seu programa como `list1005.cpp`.

8. Considere a listagem `list4401.cpp`:

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
```

```
  std::vector<int> data{};
  int x{};

  while (std::cin >> x)
    data.push_back(x);

  for (auto start(data.begin()), end(data.end()); start < end; ++start){
    --end;
    std::iter_swap(start, end); // swap contents of two iterators
  }

  std::copy(data.begin(),
            data.end(),
            std::ostream_iterator<int>(std::cout, "\n"));
}
```

lists/list4401.cpp

O que este programa faz? Certifique-se de entender todos os detalhes da listagem.

9. O tipo *map* (também conhecido como *dictionary* ou *association*) define uma estrutura de dados simples que armazena um par (chave, valor) indexado pela chave. Um map mapeia uma chave em um valor e, para tanto, cada chave deve ser única. Considerando este tipo de dado, crie um programa que leia palavras, conte suas frequências e imprima os pares de chaves e valores utilizando um laço *range-based*. Por simplicidade, considere palavras como cadeias caracteres não-vazias, separadas por espaços em branco. Utilize a listagem `list1501.cpp` como base:

```
#include <iostream>
#include <map>
#include <string>

int main(){
  std::map<std::string, int> counts{};
  std::string word{};

  // Read words from the standard input and count the number of times
  // each word occurs.
  while (std::cin >> word)
    ++counts[word];

  // TODO: Print the results.
}
```

lists/list1501.cpp

Nesta listagem, o map armazena strings como chaves e o número de ocorrências como o valores. Como acontece com o tipo `vector`, um map é inicializado vazio por padrão. Utilize colchetes para procurar por valores de um map. Por exemplo, `counts["palavra"]` retorna o valor associado à chave `"palavra"`. Se a chave não estiver no map, ela é adicionada com um valor inicial zero (pois o valor associado à chave foi definido como inteiro nesta listagem). Dado o elemento `e` de um map, `e.first` retorna sua chave, enquanto `e.second` retorna o valor associado. As chaves de um map são armazenadas em ordem ascendente. Salve seu programa com o nome `list1501v2.cpp`.

10. Melhore a solução proposta na listagem `list1501v2.cpp` para que seu programa se comporte conforme mostrado a seguir:

```
$ ./list1503
Oi, tudo bem com voce?
Oi,           1
bem           1
com           1
tudo          1
voce?         1
```

Nesta versão, utilizamos o tamanho `n` da maior chave para ajustar a largura de campo para a escrita da chave (`std::setw(n)`) e escrevemos o número de ocorrências de cada chave com largura de campo 10. Note que precisamos percorrer todo o map antes da escrita, para determinar o valor de `n`. Para isso, utilize um laço *range-based*. Salve seu programa como `list1501v3.cpp`.

11. Utilizando a listagem `list1501.cpp` como ponto de partida, escreva um programa para imprimir o número de ocorrências de uma palavra indicada pelo usuário. Seu programa deve ter o seguinte comportamento:

```
$ more input.txt
List
Lists are sequence containers that allow constant time insert and erase
operations anywhere within the sequence, and iteration in both directions.
List containers are implemented as doubly-linked lists; Doubly linked lists
can store each of the elements they contain in different and unrelated storage
locations. The ordering is kept internally by the association to each element
of a link to the element preceding it and a link to the element following it.
$ ./list1504 input.txt "the"
the: occurs 5 times
```

Dica: utilize o método `std::map::find` para retornar o iterador para uma dada chave.

12. Consulte a seguinte referência para conhecer mais containers da C++ 11 STL (Standard Template Library): http://www.cplusplus.com/reference/stl/. Defina um problema de sua preferência para ser solucionado com base em um desses containers. Você deve utilizar ao menos um container não explorado por este problema para solucionar o problema proposto.