Programação Orientada a Objectos 2021/2022

Exercícios

Ficha Nº 5

Objetos dinâmicos.
Matrizes dinâmicas
Classes com construtores por cópia, operador de atribuição e destrutor

1. Analise o seguinte código. Trata-se de uma classe que não serve para nada em particular a não ser apresentar uma mensagem sempre que é criada explicitamente ou destruída. Passe o seu código para o computador e inclua as bibliotecas necessárias.

```
class MSG {
public:
   MSG(const char * p) {
      cout << "ola" << p << "\n";
   }
   ~MSG() {
      cout << "Adeus\n";
   }
};
```

a) Na função *main* crie dois objetos *a* e *b* de forma dinâmica mas sem usar *malloc*. Os objectos são criados independentemente um do outro. Execute o programa em modo "*without debugging*". Repare nas mensagens que aparecem no ecrã: a dos construtores aparecem, mas as dos destrutores não. Será que os objetos não são destruídos? Mas se o programa terminou, tudo o que pertence ao programa é libertado, objetos incluídos. O que se estará a passar? Explique.

b) Acrescente o código à função *main* para apagar os objetos e sem usar *free*. Experimente apagar os objetos por uma ordem (a e depois b) e depois por outra (b e depois a). Repare que tem total liberdade para escolher a ordem pela qual prefere apagar os objetos. Confirme pelas mensagens que agora os destrutores estão a ser invocados.

c) Acrescente ao seu programa uma função *void func(a, b)* e mova para dentro dela a criação dos dois objetos. Defina qual o tipo dos argumentos a e b. Repare que a vida dos objetos deixa de ser controlada pela duração da função onde os criou. Neste caso os objetos são criados na função *func* e apagados já depois de essa função ter terminado. Confirme com as mensagens dos construtores e destrutores, inserindo também mensagens nas funções *func* e *main* para o ajudar a identificar os pontos por onde a execução vai passando.

d) Agora crie uma matriz dinâmica de 3 objetos. Repare que a sintaxe não lhe permite a criação da matriz por causa do construtor. Modifique o parâmetro do construtor da classe MSG de forma a que possa ser invocado sem nenhum parâmetro e confirme que agora já é possível a criação da matriz dinâmica. Execute o programa e repare nas mensagens. O primeiro elemento da matriz a ser criado é o que está na posição com índice 0, e o último é o que está na posição com índice 2.

e) Acrescente o código necessário para destruir a matriz dinâmica. Corra novamente o programa e confirme pela ordem das mensagens que os objetos da matriz dinâmica estão a ser destruídos. Os objetos da matriz são destruídos pela ordem habitual: o primeiro a ser destruído é o que foi construído em último lugar (posição com índice 2).

f) Debata com o professor e verifique que entende todos os conceitos relativos aos seguintes tópicos:

- A sintaxe de criação/destruição de matrizes dinâmicas, nomeadamente, a correspondência que existe entre *new – delete* e *new[] – delete[]*.
- As razões pelas quais *new* e *delete* são superiores a *malloc* e *free*, nomeadamente quanto à questão da garantia de invocação dos construtores e destrutores e quanto ao *type-safety*.
- Uma matriz dinâmica é uma entidade indivisível: não pode "apagar" um elemento da matriz com o intuito de encolher a matriz, nem acrescentar um elemento novo. Se mudar o tamanho da matriz terá que criar uma nova matriz com o tamanho desejado.

Se tiver dúvidas em algum destes aspectos deve perguntar ao professor antes de prosseguir. Anote no seu caderno as suas conclusões. Anote também no caderno que <u>não pode usar as funções *free* e *malloc* nesta disciplina</u>.

Objetivos do exercício

- Introdução aos objetos dinâmicos.
- Entender as diferenças quanto à utilidade e quanto à sintaxe entre objetos automáticos e objetos dinâmicos.
- Entender as vantagens de new e delete sobre malloc e free.
- Entender a sintaxe da criação e destruição de matrizes dinâmicas e o relacionamento entre new - delete e new[] e delete[].

2. A classe ABC armazena alguns caracteres no ecrã. Analise e copie para o computador o código abaixo. Repare que não se usam em lado nenhum as antigas funções de C *malloc* e *free*, mas sim as versões de C++ *new* e *delete*. Este exercício exige o conhecimento de *new* e *delete*, tanto na criação/libertação de um objecto de cada vez, como de vários de cada vez. Se tiver dúvidas na sintaxe ou significado de *new* e *delete* deve perguntar imediatamente ao professor antes de prosseguir.

```
class ABC {
   char * p;
public:
   ABC(char * s) {
      p = new char[strlen(s)+1];
      strcpy(p,s);
   }
};
```

a) Analise a lógica do construtor. O que pode concluir à forma como os objetos da classe ABC vêm os caracteres apontados por *p*? Nomeadamente, será que os objetos de ABC podem assumir que os caracteres lhe pertencem? Mais especificamente, estar-se-á perante um caso de composição ou será um caso de agregação? Discuta com o professor a sua opinião.

b) Analise o código abaixo. O que acha que acontece à memória livre no final do ciclo? Claramente falta à classe o mecanismo que permite devolver ao computador os recursos detidos pelos seus objetos quando estes são destruídos. Acrescente esse mecanismo devolvendo o espaço ocupado pelos caracteres apontados por *p*.

```
void gastaMem() {
   ABC temp("Texto temporário que ocupa espaço");
   // Etc. Tanto faz.
}

int main() {
   for (unsigned int i=0; i<100000000; i++)
      gastaMem();
   cout << "o programa ja deve terminado por falta de memória ";
   return 0;
}
```

c) Verifique que na alínea anterior fez o destrutor tal como está no código abaixo. Torna-se agora mais claro que se está perante um caso de composição e que os caracteres apontados por *p* pertencem ao objeto de ABC e que devem ser apagados quando o objeto a que pertencem desaparece. Confronte a sua classe com a que se encontra abaixo. Transcreva o resto do programa para o computador e experimente-o. Confirme que ele rebenta antes de chegar ao primeiro *cout* da função *main*. Explique porquê, identificando o problema.

```
class ABC {                                  // usar copy & paste se for preciso
   char * p;                                 // (não é para perder tempo a
public:                                      // escrever de raiz no computador).
   ABC(char * s) {
      p = new char[strlen(s)+1];
      strcpy(p,s);
   }
   ~ABC() {
      delete []p;
   }
   const char * getConteudo() const { return p; }
};                                                    // continua na prox. pag.
```

```
void func(ABC x) {
  // tanto faz
}

void func() {
   ABC a("aaa");
   ABC b("bbb");
   a = b;
}

int main() {
   ABC y("ola");
   func(y);
   cout << y.getConteudo();  // se não rebentar experimentar outra func(y)
   cout << "o programa já deve ter rebentado";
   func();
   cout << "o programa já deve ter rebentado novamente";
   return 0;
}
```

d) Resolva o problema que identificou na alínea anterior. Experimente novamente e verifique que o programa agora já avança um pouco mais mas agora rebenta antes de chegar ao segundo *cout* da função *main*. Explique porquê, identificando o problema.

e) Resolva o problema que ainda faltava resolver e corra novamente o programa em modo "*without debugging*". Confirme que o programa termina corretamente sem erro nenhum. Esta classe não era robusta nas operações de cópia nem de atribuição, mas ao chegar a esta alínea já é.

f) No decorrer deste exercício deve ter feito um construtor por cópia e um operador de atribuição. Estas duas funções membro partilham boa parte do código e é muito habitual fazer-se uma à custa da outra. Se não o fez ainda, faça:

I. O construtor por cópia à custa do (invocando o) operador de atribuição.

II. O operador de atribuição à custa do (invocando o) construtor por cópia (idioma *swap*)

g) Verifique que entendeu os seguintes conceitos:

- Classes que precisam que o mecanismo de construção por cópia seja ajustado pelo programador. Construtor por cópia.
- Classes que precisam que o mecanismo de atribuição seja ajustado pelo programador. Operador de atribuição.
- Formas de fazer o construtor por cópia à custa do operador de atribuição. Aspetos a ter em atenção para evitar a autoatribuição.
- Significado da expressão "a classe deve ser robusta para as operações habituais".

Anote as suas conclusões no caderno.

Objetivos do exercício

- Entender o conceito e significado de "classe robusta para as operações habituais"
- Entender as situações em que é necessário um operador de atribuição e um construtor por cópia.
- Relacionar a necessidade de construtores por cópia e operador de atribuição com os conceitos de composição, de destrutor, e de robustez da classe.
- Conseguir fazer o construtor por cópia à custa do operador de atribuição e vice-versa a diferença na semântica entre uma opção e outra.

3. A classe Ponto representa as coordenadas de um ponto no plano. O código inicial é o seguinte:

```
class Ponto {
   int x,y;
public:
   Ponto(int a, int b) : x(a), y(b) {}
   int getX() { return x; }
   int getY() const { return y; }
   void setX(int i) { x = I; }
   void setY(int i) { y = I; }
};
```

a) Pretende-se que esta classe seja robusta para todas as operações habituais em que os seus objetos possam participar (cópia de objetos por cópia, por referência, por ponteiro, atribuição, inicialização, destruição, etc.). Identifique o que está em falta na classe. Confirme com o professor e depois (e só depois) faça aquilo que está em falta e **apenas isso**.

Objetivos do exercício

- Treinar o conceito de classe "robusta para as operações habituais"
- Treinar o entendimento que "classe robusta" não implica necessariamente construtores / destrutores / operadores de atribuição.
- Treinar a capacidade de identificação das situações onde é necessária a definição de construtor por cópia / operador de atribuição / destrutor e as situações onde tal não é necessário.

4. A classe *EquipaFutebol* representa uma equipa de futebol. Cada equipa de futebol “tem” 11 pessoas, que são os seus jogadores. Os jogadores da equipa são pessoas que têm a sua própria vida e por acaso são referidas numa entidade que é a equipa. As pessoas podem “pertencer” a várias equipas e outras coisas e se a equipa for extinta, as pessoas continuam na sua vida habitual.

a) Dada a descrição da natureza das pessoas, equipa e da relação entre ambos, parece evidente que os objetos de Pessoa têm uma existência independente do objeto equipa, e que este vai apenas armazenar ponteiros para objetos pessoas que já existem previamente. Os objetos *Pessoa* não estão armazenados dentro do objeto *Equipa* e está-se perante um caso de agregação e não de composição. Nesta altura este aspeto deve ser óbvio. Confirme que a sua análise do problema coincide e pergunte ao professor caso não concorde. Nesta alínea é só para analisar a situação e não para escrever código.

b) Da descrição da situação e do exposto na alínea anterior, fez-se o código apresentado abaixo. Analise-o. Pretende-se que a classe *EquipaFutebol* seja robusta para todas as operações habituais em que os seus objetos possam participar (cópia de objetos por cópia, por referência, por ponteiro, atribuição, inicialização, destruição, etc.), independentemente se existir no código-exemplo abaixo alguma situação onde eventuais problemas se notem ou não (os eventuais problemas são intrínsecos às classes e não a eventual código exemplo). Identifique o que está em falta na classe. Confirme com o professor e depois (e só depois) faça aquilo que está em falta e apenas isso.

```
class Pessoa {                                  // Isto é para fazer copy & paste
   // tanto faz o que aqui está                 // para o programa no computador.
};                                              // Mas pode nem ser sequer preciso
                                                // -> Ver primeiro as perguntas.
class EquipaFutebol {
   Pessoa * jogadores[11];   // 11 máximo
public:
   EquipaFutebol() {
      for (unsigned int i=0; i<11; i++)         // inicialmente equipa vazia
         jogadores[i]= NULL;                    // NULL significa sem jogador
   }
   setJogador(Pessoa * p, int pos) {
      jogadores[pos] = p;         // Notar que o obj. Pessoa é visto pela equipa
   }                              // mas a pessoa não passa para "dentro" da
};                                // equipa (é uma agregação de Pessoas)

int main() { // exemplo de utilização. Não serve para ver se a classe é robusta
   Pessoa eus(...), ebio(...); // de acordo com o que estiver na classe Pessoa
   EquipaFutebol fcpoo;
   fcpoo.setJogador(& eus,0);  // eus passa a ~~pertencer~~ ser jogador da equipa
   fcpoo.setJogador(& ebio,1); // Idem para ebio
   return 0;
}
```

Objetivos do exercício

- Treinar o conceito de classe "robusta para as operações habituais"
- Treinar o entendimento que "classe robusta" não implica necessariamente construtores / destrutores / operadores de atribuição.
- Treinar a capacidade de identificação das situações onde é necessária a definição de construtor por cópia / operador de atribuição / destrutor e as situações onde tal não é necessário.

5. Analise o código abaixo. A classe *Clube* representa uma coleção de pessoas com um interesse em comum. É um caso muito semelhante ao da *EquipaFutebol* do exercício anterior, mas neste caso em vez de equipa é um clube, e o número de ~~jogad...~~ sócios é um número que tanto pode ser 11 como 11000: cada clube pode ter um número de sócios que não se conhece durante a compilação do programa – só durante a execução é que se sabe, e clubes diferentes poderão ter números de sócios muito diferentes.

```
class Pessoa {                                      // Isto é para fazer copy & paste
   // tanto faz o que aqui está                     // para o programa no computador.
};                                                  // Mas pode nem ser sequer preciso
                                                    // -> Ver primeiro as perguntas.
class Clube {
   Pessoa * * socios;
   int tam;
public:
   Clube(int t) {
      tam = t;                                  // por questões de espaço assume-se
      socios = new Pessoa * [tam];              // que há memória e não dá erro, mas
      for (unsigned int i=0; i<tam; i++)        // isto deve ser devidamente validado
         socios[i]= NULL;                       //    (  if (socios != NULL) ...  )
   }
   ~Clube() { delete []p; }
   setMembroDoClube(Pessoa * p, int pos) {
      socios[pos] = p;          // Notar que o obj. Pessoa é visto pelo Clube
   }                            // mas o clube não toma posse desse objecto
};                              // (o clube é uma agregação de Pessoas)

int main() {
   Pessoa a(...), b(...);  // de acordo com o que estiver na classe Pessoa
   Clube c(50);
   c.setMembroDoClube(&a, 0);  // Pessoa a passa a ~~pertencer~~ ser membro do clube
   c.setMembrodoClube(&b, 1);  // Idem Pessoa b
   return 0;
}
```

a) Analise a forma de representar os sócios. Tal como no caso da *EquipaFutebol*, vai-se guardar apenas ponteiros para pessoas. Mas neste caso, devido à questão do número de sócios, não se pode usar uma matriz de tamanho fixo. Tem que ser uma matriz de tamanho dinâmico. Verifique que compreende este aspeto, e se tiver dúvidas esclareça já com o professor do laboratório. Também poderia usar um vector (de ponteiros) em vez de uma matriz dinâmica, mas neste caso pretende-se que não use vetores (imposição do enunciado). Nesta alínea pretende-se analisar a situação e não fazer código.

b) Analisando o código, verifique se está perante um caso de agregação ou perante um caso de composição. Confirme com o professor. Para o ajudar a decidir, repare no destrutor:

- O destrutor apaga a matriz de ponteiros, mas não as pessoas. Estas residem "fora" do âmbito do objeto *Clube*. O destrutor podia apagar cada um dos objetos *Pessoa* cujos ponteiros se encontram na matriz, mas opta por não o fazer (talvez o *Clube* entenda que as pessoas continuam a existir mesmo que o objeto *Clube* desapareça).

c) Repare que esta classe usa memória dinâmica. Talvez isso indique que seja preciso ter cuidados especiais, ou talvez não. Pretende-se que esta classe seja robusta para todas as operações habituais em que os seus objetos possam participar (passagem de objetos por cópia, por referência, por ponteiro, atribuição, inicialização, destruição, etc.). Identifique o que está em falta na classe. Confirme com o professor e depois (e só depois) faça aquilo que está em falta e apenas isso.

d) Mude a implementação de matriz dinâmica para vector. Os algoritmos de diversas funções ficam, naturalmente, diferentes.

I. Depois de adaptar para vector, refaça a alínea c) (faça mesmo alterações ao código se forem necessárias). Aquilo que teve que fazer num caso e noutro é diferente. Porquê?

II. Para além das diferenças do que tem que fazer na alínea c) em função de ter usado matriz ou vector, existe algum código/função que já existia (desde o início do exercício) que também tenha que ser alterado? O quê e porquê?

Objetivos do exercício

- Treinar o conceito de classe "robusta para as operações habituais"
- Treinar o entendimento que "classe robusta" não implica necessariamente construtores / destrutores / operadores de atribuição.
- Treinar a capacidade de identificação das situações onde é necessária a definição de construtor por cópia / operador de atribuição / destrutor e as situações onde tal não é necessário.
- Entender a diferença na responsabilidade de gestão entre uma matriz fixa, uma matriz dinâmica e um vector.

6. Analise o código abaixo. A classe representa *Agenda* representa uma coleção de contactos telefónicos com capacidade para 30 contactos. O programador escolheu, e muito bem, representar os contactos por uma classe à parte. Uma vez que só se vai ter 30 contactos, a opção por uma matriz na classe *Agenda* é razoável e fácil, mas existe o problema que, quando se tem uma matriz de objetos, esses objetos são logo criados, o que iria exigir dados concretos para a inicialização desses objetos todos (neste caso para 30 contactos logo na criação da agenda), ou então que faça sentido que haja (e que haja mesmo) um construtor por omissão para os objetos que ficam na matriz (neste caso, contactos).

No cenário deste exercício, pretende-se manter a matriz na *Agenda*, mas não existe (nem se vai adicionar) o construtor por omissão na classe *Contacto*. O programador optou, então, e muito bem, por armazenar os objetos fora da matriz. Esta fica apenas uma matriz de ponteiros para objetos de *Contacto*, e os contactos serão criados um de cada vez, usando o construtor que existe na classe *Contacto*. **Nota:** esta explicação é uma revisão da matéria sobre classes e encapsulamento. Certifique-se que compreende o que foi dito aqui quanto à razão pela qual se tem uma matriz de ponteiros e coloque as duvidas que achar pertinentes.

A classe *Contacto* está terminada, mas a classe *Agenda* talvez esteja incompleta. O foco deste exercício é a classe Agenda.

```
class Contacto {                                  // Isto é para fazer copy & paste
   string nome;                                   // para o programa no computador.
   unsigned int tel;                              // Mas pode nem ser sequer preciso
public:                                           // -> Ver primeiro as perguntas.
   Contacto(string n, unsigned int t) : nome(n), tel(t) {}
   string getNome() const { return nome; }
   int getTel() const { return tel; }
   void setNome(string s) { string = s; }
   void setTel(unsigned int t) { tel = t; }
};

class Agenda {
   Contactos * lista[30];  // os elementos não usados ficarão a NULL
public:
   Agenda() {
      for (unsigned int i=0; i<30; i++)
         lista[i]= NULL;
   }
   void adicionaContacto(string nome, unsigned int tel, int pos) {
      // os algoritmos da agenda foram simplificados para manter o foco nos
      // aspectos relevantes do exercício.
      if (lista[pos] == NULL)   // se a entrada na lista está vazia prossegue
         lista[pos] = new Contacto(nome, tel);   // reparar no new
   }
   // funções para procurar, remover, atualizar, listar. Pode faze-las como TPC
};
```

a) Observe que os objetos de *Contacto* são criados dentro do código da classe *Agenda*. O resto do programa não tem noção que tais objetos existem. Se o objeto *Agenda* for destruído, o que deve acontecer aos objetos *Contacto* apontados pelos ponteiros na matriz *lista*? Será que devem ser destruídos também? Será que esse processo ocorre automaticamente? Por outras palavras, deverá existir um destrutor na classe Agenda? Se achar que sim, faça-o.

b) Agora que tem um destrutor que apaga os objetos apontados pelos ponteiros na matriz lista (sim, era mesmo necessário), verifique se a sua classe tem um comportamento aceitável nas operações de cópia e de atribuição dos seus objetos. Por esta altura já se tornou claro que os contactos são propriedade exclusiva da agenda, e a análise/testes que devem ser feitos são do mesmo tipo que na classe ABC (por exemplo). Debata com o professor as suas conclusões.

c) Nesta alínea já terá percebido que é necessário ter um construtor por cópia que duplica os objetos contacto da agenda original, e um operador de atribuição que apaga os contactos antigos e duplica os contactos da agenda original. No entanto, a classe Agenda, não tem nenhuma matriz dinâmica. Os construtores por cópia/operadores de atribuição não estão necessariamente associados a matrizes dinâmicas. Escreva no seu caderno as suas conclusões e faça o operador de atribuição e o construtor por cópia da seguinte maneira:

i) Operador de atribuição e construtor por cópia independentemente um do outro

ii) O construtor por cópia à custa do operador de atribuição.

d) A segunda das opções anteriores é a preferível, tal como visto nas aulas. Não se esqueça de:

- Sempre que faz um construtor por cópia à custa de operador de atribuição deverá "preparar" o objeto de forma a torna-lo compatível com o operador.
- Sempre que faz um operador de atribuição deve verificar o caso da autoatribuição.

e) Faça as funções para consultar, remover, listar, atualizar, etc. Esta alínea destina-se a treinar algoritmos.

Objetivos do exercício

- Lidar com situações em que uma classe é responsável por recursos (objetos dinâmicos ou outras coisas) que residem no seu exterior e exigem a criação de operador de atribuição / construtor por cópia / destrutor, mas em que não estão necessariamente envolvidas matrizes dinâmicas.
- Treinar algoritmos menos-que-triviais (última alínea).
- Consolidar os aspetos relacionados com robustez de classes.

7. Pretende-se expandir o potencial da classe do exercício anterior. Mais concretamente, passa a poder existir qualquer número de contactos. Vai então ter que se usar uma matriz dinâmica em vez de uma matriz estática. Os objetos de *Contacto* continuam a ter que ser armazenados no exterior dessa matriz porque continua a não se ter um construtor por omissão na classe *Contacto*. Ou seja, a matriz é dinâmica, mas continua a ser de ponteiros. Deixa de fazer sentido estar a dizer em que posição vai ficar um novo contacto: acrescentar um contacto implica ter que se criar uma nova matriz com tamanho "mais um", e se se apagar um contacto, vai ser necessário copiar todos os contactos menos esse para uma nova matriz de tamanho "menos um". Os algoritmos para lidar com isto são semelhantes ao da classe *MyString* feita atrás. Este exercício não implica modificar a classe Contacto.

a) Modifique a classe *Agenda* do exercício anterior de acordo com o pretendido aqui. Adapte os dados e as funções de consulta, eliminação, listagem, atualização, etc.

b) Verifique que nas operações de destruição, já não basta apagar os contactos (faça um diagrama para o ajudar a ver este aspeto). Agora existe uma matriz dinâmica (a que guarda os ponteiros) que também é um recurso da exclusiva responsabilidade da classe *Agenda*. Adapte o destrutor em conformidade com esta situação.

c) As razões que tornaram necessário adaptar o destrutor também existem no caso da atribuição e de construção por cópia: existe uma matriz dinâmica cuja cópia não ocorre automaticamente. Adapte o operador de atribuição e o construtor por cópia em conformidade.

Objetivos do exercício

- Lidar com situações em que uma classe é responsável por recursos (objetos dinâmicos ou outras coisas) que residem no seu exterior e exigem a criação de operador de atribuição / construtor por cópia / destrutor, mas em que não estão necessariamente envolvidas matrizes dinâmicas.
- Treinar a análise das situações que envolvem objetos que pertencem ou não ao objeto e são armazenados de forma dinâmica e as consequências sobre o construtor por cópia / operador de atribuição e destrutor.
- Treinar algoritmos de manipulação de matrizes dinâmicas.
- Consolidar os aspetos relacionados com robustez de classes.

8. Pretende-se simplificar o código da classe Agenda resultante do exercício imediatamente anterior. Neste caso, vai-se substituir a matriz dinâmica por um vector (continua a ser de ponteiros para *Contacto*, embora agora já pudessem ser armazenados objeto diretamente no vector). O vector simplifica as operações de inserção e remoção: as funções membro do vector auxiliam a inserção, procura, eliminação de elementos (ex: função push_back).

a) Modifique a classe *Agenda* do exercício anterior de acordo com o pretendido aqui. Adapte os dados e as funções de consulta, eliminação, listagem, atualização, etc.

b) Verifique que agora, o armazenamento dos ponteiros para Contacto é da responsabilidade do vector. Este é autocontido na sua funcionalidade, e aquilo que for preciso fazer no que toca à gestão desse armazenamento já é feito pelo vector. Deixa de ser necessário ter a manipulação de uma matriz dinâmica no destrutor, construtor por cópia e operador de atribuição na classe *Agenda* porque tal matriz já nem existe. Adapte o destrutor, operador de atribuição e construtor por cópia da classe Agenda em conformidade.

c) Confirme que voltou a ter uma solução semelhante à situação da classe *Agenda* original como a que obteve no final do exercício 6. Se não for semelhante chame o professor e peça esclarecimentos.

Objetivos do exercício

- Treinar situações em que uma classe é responsável por recursos (objetos dinâmicos ou outras coisas) que residem no seu exterior e exigem a criação de operador de atribuição / construtor por cópia / destrutor, mas em que não estão necessariamente envolvidas matrizes dinâmicas.
- Rever e treinar vetores.
- Consolidar os aspetos relacionados com robustez de classes.

9. Pretende-se construir uma classe chamada *MyString* que tenha um objetivo e comportamento semelhante à classe da STL *string*. A classe deve ser robusta para as operações habituais em que os seus objetos possam ser usados. Os objetos da classe têm as seguintes características:

- Armazena um número indeterminado de caracteres: podem ser poucos ou muitos.
- Os caracteres armazenados num objecto *MyString* pertecem-lhe exclusivamente.
- É possível construir objetos *MyString* da seguinte forma:
  - *MyString a;* → *a* fica sem caracteres nenhuns.
  - *MyString b("ola");* → *b* fica com "ola".
  - *MyString c(123);* → *c* fica com "123" (caracteres – não é o número).
- Deve ser possível imnprimir objectos *MyString* usando a expressão
  - *cout << a << b;* → *a* e *b* são objetos de *MyString*

A classe deve ser robusta para as operações habituais em que os seus objetos possam participar.

a) Faça a classe *MyString* com as características pedidas.

b) Elabore uma função *main* e eventuais funções globais auxiliares para testar a robustez da classe. Inclua situações onde os objectos de *MyString* sejam atribuídos, passados por cópia em parâmetro e em retorno. Pode testar também com o código que se encontra abaixo.

```
void func(MyString x) {
  // tanto faz
}

void func() {
   MyString a("aaa");
   MyString b("bbb");
   a = b;
}

MyString func2() {
   MyString c("aaa");
   Return x;
}

int main() {
   MyString y("ola");
   func(y);
   cout << y;
   cout << "será que o programa já rebentou? (1)\n";
   func2();
   cout << "será que o programa já rebentou? (2)\n";
   func();
   cout << "será que o programa já rebentou? (3)\n";
   return 0;
}
```

c) Acrescente à sua classe a possibilidade de acrescentar um caracter numa determinada posição com a função *insertCharAt(char c, int pos)*. Teste essa funcionalidade.

d) Acrescente à sua classe a possibilidade de eliminar um caracter numa determinada posição com a função *removeCharAt(char c, int pos)*. Teste essa funcionalidade.

e) Acrescente à sua classe a possibilidade de concatenar os caracteres de uma *MyString* a outra. Veja o código abaixo.

```
int main() {
   MyStringABC x("ola");
   MyStringABC y("mundo");
   x.concatenaCom(" ");
   x.concatenaCom(y);
   cout << x << endl;  // aparece "ola mundo"
   cout << y << endl;  // aparece "mundo"
   return 0;
}
```

Objetivos do exercício

- Treinar a elaboração de classes com memória dinâmica em que o conteúdo dessa memória dinâmica pertence aos objetos dessa classe
- Treinar os algoritmos de manipulação de memória dinâmica
- Consolidar os aspetos de robustez de classes que envolvem operador de atribuição, construtor por cópia e destrutor.

10. Refaça (modifique) o exercício anterior usando desta vez operadores como alternativa à função que já tinha feito para a concatenação e acrescentando a funcionalidade de obter um caracter numa determinada posição, com a possibilidade de a usar à esquerda numa atribuição, e de eliminação da primeira ocorrência de uma string noutra. Veja o exemplo e use operadores para a funcionalidade pedida.

```
int main() {
   MyStringABC x("ola");
   MyStringABC y("mundo");
   MyStringABC w("azul");
   MyStringABC z;
   x += y;        // FAZER: concatenar strings passa a ser por operador +=
   cout << x;     // aparece "olamundo"
   z = x + y;     // FAZER: suportar a obtenção de uma string nova por junção
                  // de outras duas
   cout << z;     //    aparece "olamundoazul"
   z -= y;        // FAZER: remover a 1ª ocorrência de uma string com o op. -=
   cout << z;     // aparece "olaazul"    ("mundo" foi removido)
   cout << z[4];  // FAZER: obter caracter numa dada posição. Aparece 'z'
   z[3] = 'e';    //    Pode-se usar do lado esquerdo de uma atribuição.
   cout << z;     //    aparece "olaezul"    (z[3] <- passou a ser 'e')
   z[300] = 'x';  //    Não acontece nada. Os limites são validados
   cin >> x;      // FAZER: preencher x com dados obtidos do teclado
   return 0;}
```

Objetivos do exercício

- Rever a matéria de operadores.
- Consolidar a elaboração de classes com memória dinâmica em que o conteúdo dessa memória dinâmica pertence aos objetos dessa classe.
- Consolidar os algoritmos de manipulação de memória dinâmica.
- Consolidar os aspetos de robustez de classes que envolvem operador de atribuição, construtor por cópia e destrutor.

11. Pretende-se construir uma classe que represente imagens num ecrã. As imagens são formadas por pontos de cor. Assim uma imagem não é mais do que matriz de pontos organizada em linhas e colunas. Cada ponto é uma associação de três cores: vermelho, verde e azul (cores primárias aditivas), sendo que a combinação destas três cores em diversas intensidades resulta em todas as cores que se conseguem ver. Já existe o seguinte código para a classe de pontos de cor (classe *PixelRGB*).

```
class PixelRGB {
   char r,g,b;         // RED GREEN BLUE, valores de 0 a 255
public:
   PixelRGB(char cr = 0, int cg = 0, int cb = 0) : r(cr), g(cg), b(cb) {}
   int getR() const { return r; }
   int getG() const { return g; }
   int getB() const { return b; }
   void setR(int c) { r = c; }
   void setG(int c) { g = c; }
   void setB(int c) { b = c; }
};
```

a) Usando a classe *PixelRGB*, escreva a classe *Imagem* para representar imagens retangulares constituídas por pontos de cor. A classe Imagem deve ter as seguintes características:

- Armazena um número indeterminado de pontos de cor representados por objetos de *PixelRGB*. A classe armazena internamente os pontos de cor (objetos de *PixelRGB*) como for adequado, mas transmite para o exterior a ideia que se encontram organizados por linhas e colunas. Sugestão: pretende-se que seja usada memória dinâmica.
- A construção de objetos de Imagem pressupõe a indicação do número de linhas e de colunas. A *Imagem* fica logo com *número-de-linhas x número-de-colunas* pontos de cor. Consegue prever qual será a cor inicial que esses pontos de cor vão ter?
- Deve ser possível obter/chegar ao ponto de cor que se encontra numa determinada posição *linha* e *coluna*. Faça uma função para este propósito devolvendo o ponto de cor:
  III. Por referência para objetos *PixelRGB*.
  IV. Por ponteiro para objetos *PixelRGB*.
  V. De forma a que as seguintes expressões funcionem:

```
int main() {
   Imagem img(30,40);
   PixelRGB aux = img[15][16]); // obtém ponto de cor em 15,16
   PixelRGB temp(30,40,50);
   img[13][15] = temp;          // coloca pixel de cor diferente
   return 0;
}
```

Não esqueça que a classe deve ser robusta para todas as operações habituais em que os seus objetos possam participar (passagem de objetos por cópia, por referência, por ponteiro, atribuição, inicialização, destruição, etc.).

b) Existem duas formas de armazenar os pontos de cor através de memória dinâmica:

- Através de uma matriz unidimensional de objetos de pontos de cor de dimensão *número-de-linhas x número-de-colunas*, e depois convertendo posições *(linha, coluna)* em *(linha x numero-de-colunas + coluna)*.
- Através de uma matriz de *linhas*, em que cada *linha* é uma matriz de pontos de cor. A utilização de coordenadas *(linha, coluna)* é mais direta, mas tem-se mais matrizes dinâmicas para gerir.

Se fez a alínea a) de uma forma, agora faça da outra.

Objetivos do exercício

- Treinar a construção de classes que gerem os seus recursos dinâmicos
- Treinar reconhecer situações em que é necessário construtor por cópia, operador de atribuição e destrutor
- Treinar a utilização de memória dinâmica