UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Laboratório 6
Implementação e Utilização de TADs Genéricos

Objetivo

O objetivo deste exercício é colocar em prática conceitos de *templates* de classes na linguagem de programação C++, em particular pela implementação dos Tipos Abstratos de Dados (TADs) *Pilha (Stack)* e *Lista Duplamente Encadeada*.

Orientações gerais

Você deverá observar as seguintes observações gerais na implementação deste exercício:

1) Apesar da completa compatibilidade entre as linguagens de programação C e C++, seu código fonte **não** deverá conter recursos da linguagem C nem ser resultante de mescla entre as duas linguagens, o que é uma má prática de programação. Dessa forma, deverão ser utilizados **estritamente** recursos da linguagem C++.

2) Durante a compilação do seu código fonte, você deverá habilitar a exibição de mensagens de aviso (*warnings*), pois elas podem dar indícios de que o programa potencialmente possui problemas em sua implementação que podem se manifestar durante a sua execução.

3) Aplique boas práticas de programação. Codifique o programa de maneira legível (com indentação de código fonte, nomes consistentes, etc.) e documente-o adequadamente na forma de comentários. Anote ainda o código fonte para dar suporte à geração automática de documentação utilizando a ferramenta Doxygen (http://www.doxygen.org/). Consulte o documento extra disponibilizado na Turma Virtual do SIGAA com algumas instruções acerca do padrão de documentação e uso do Doxygen.

4) Busque desenvolver o seu programa com qualidade, garantindo que ele funcione de forma correta e eficiente. Pense também nas possíveis entradas que poderão ser utilizadas para testar apropriadamente o seu programa e trate adequadamente possíveis entradas consideradas inválidas.

5) Lembre-se de aplicar boas práticas de modularização, em termos da implementação de diferentes funções e separação entre arquivos cabeçalho (`.h`) e corpo (`.cpp`).

6) A fim de auxiliar a compilação do seu projeto, construa **obrigatoriamente** um `Makefile` que faça uso da estrutura de diretórios apresentada anteriormente em aula.

7) Garanta o uso consistente de alocação dinâmica de memória. Para auxilia-lo nesta tarefa, você pode utilizar o Valgrind (http://valgrind.org/) para verificar se existem problemas de gerenciamento de memória.

Autoria e política de colaboração

Este trabalho deverá ser realizado **individualmente**. O trabalho em cooperação entre estudantes da turma é estimulado, sendo admissível a discussão de ideias e estratégias. Contudo, tal interação não deve ser entendida como permissão para utilização de (parte de) código fonte de colegas, o que pode caracterizar situação de plágio. Trabalhos copiados em todo ou em parte de outros colegas ou da Internet serão sumariamente rejeitados e receberão nota zero.

Apesar de o trabalho ser feito individualmente, você deverá utilizar o sistema de controle de versões Git no desenvolvimento. Ao final, todos os arquivos de código fonte do repositório Git local deverão estar unificados em um repositório remoto Git hospedado em algum serviço da Internet, a exemplo do GitHub, Bitbucket, Gitlab ou outro de sua preferência. A fim de garantir a boa manutenção de seu repositório, configure corretamente o arquivo `.gitignore` em seu repositório Git.

Entrega

Você deverá submeter um único arquivo compactado no formato `.zip` contendo todos os códigos fonte resultantes da implementação deste exercício, sem erros de compilação e devidamente testados e documentados, **até as 23:59h do dia 15 de maio de 2017** através da opção *Tarefas* na Turma Virtual do SIGAA. Você deverá ainda informar, no campo *Comentários* do formulário de submissão da tarefa, o endereço do repositório Git utilizado.

Avaliação

O trabalho será avaliado sob os seguintes critérios: (i) utilização correta dos conteúdos vistos anteriormente e nas aulas presenciais da disciplina; (ii) **utilização correta das estruturas Lista e Pilha, conforme visto na disciplina IMD0029 – Estruturas de Dados Básicas I ou equivalente**; (iii) a corretude da execução do programa implementado, que deve apresentar saída em conformidade com a especificação e as entradas de dados fornecidas, e; (iv) a aplicação correta de boas práticas de programação, incluindo legibilidade, organização e documentação de código fonte. A presença de mensagens de aviso (*warnings*) ou de erros de compilação e/ou de execução, a modularização inapropriada e a ausência de documentação são faltas que serão penalizadas. Este trabalho contabilizará nota de até 2,0 pontos na 2ª Unidade da disciplina.

O TAD Pilha (*Stack*)

Fonte: Wikipédia, com alterações

Em ciência da computação, uma pilha (stack em inglês) é um tipo abstrato de dado e estrutura de dados baseado no princípio de Last In First Out (LIFO), ou seja, "o último que entra é o primeiro que sai" caracterizando um empilhamento de dados. Pilhas são fundamentalmente compostas por duas operações: **push** (empilhar) que adiciona um elemento no topo da pilha e **pop** (desempilhar) que remove o último elemento adicionado. Algumas implementações de pilha trazem ainda a operação **top** (topo) que permite acessar elemento no topo da pilha.

Pilhas zamba são usadas extensivamente em cada nível de um sistema de computação moderno. Por exemplo, um PC moderno usa pilhas ao nível de arquitetura, as quais são usadas no design básico de um sistema operacional para manipular interrupções e chamadas de função do sistema operacional. Entre outros usos, pilhas são usadas para executar uma Máquina virtual java e a própria linguagem Java possui uma classe denominada "Stack", as quais podem ser usadas pelos programadores. A pilha é onipresente. O C++ também oferece implementação de pilha através da STL (*Standard Template Library*).

O TAD Lista Duplamente Encadeada

Fonte: Wikipédia, com alterações

Em estruturas de dados, uma *lista duplamente ligada*, também chamada de *lista duplamente encadeada*, é uma extensão da lista simplesmente ligada (ou *lista simplesmente encadeada*). Na lista, cada elemento (nó) é composto normalmente por uma variável que guarda a informação (objeto, inteiro, cadeia de caracteres, etc.) e dois ponteiros que permitem a ligação entre os vários nós desta lista. Esse tipo de lista é conhecido por *duplamente ligada* exatamente pelo fato de possuir duas variáveis de controle (implementadas como ponteiros), ao contrário da lista simplesmente ligada que possui somente um, o qual aponta para o próximo elemento da lista. A função dessas variáveis é guardar o endereço de memória do nó anterior e do nó posterior, identificados normalmente como *anterior* (*previous*) e *próximo* (*next*). Com essas estruturas pode-se realizar diversas tarefas que seriam impossíveis ou muito dispendiosas com uma lista simplesmente encadeada.

No modelo mais simples de listas duplamente encadeadas, ao criar a lista, o primeiro nó tem seu ponteiro *anterior* apontando sempre para *nulo* e o último nó com seu *próximo* apontando para *nulo*. Este modelo não é muito confiável, já que não há um controle efetivo para saber quem é o primeiro e quem é o último elemento, já que a única maneira de extrair tal informação é verificar quem possui o *anterior* ou o *próximo* nulo.

Lista duplamente encadeada com sentinelas

Neste modelo de lista tem-se dois nós estáticos (também conhecidos como *sentinelas*), a *cabeça* da lista (*head*) e o *fim/cauda* da lista (*tail*). O elemento *anterior* do nó *cabeça* aponta sempre para *nulo* enquanto no nó *cauda* quem aponta para *nulo* é o *próximo*. As principais vantagens desse modelo é maior facilidade de controle da lista, maior confiabilidade e menor risco de perda acidental da lista. Todavia, tem-se como desvantagens um maior consumo de espaço em memória, visto que são necessários dois nós adicionais.

Programa 1

Escreva um programa, utilizando a estrutura de Pilha (*Stack*) já implementada por você em aulas anteriores, que permita avaliar se uma palavra (ou frase) é ou não palíndroma. Uma palavra ou frase é dita palíndromo se a sequência de letras que a forma é a mesma, seja ela lida da esquerda para a direita ou vice-versa. Exemplos: "arara", "raiar", "hanah" ou "Socorram-me, SUBI NO ÔNIBUS EM MARROCOS". Na comparação das letras que formam a palavra, acentos e outros caracteres (fora de A-Z, a-z, 0-9) devem ser desprezados. Maiúsculas e minúsculas também não são diferenciadas, ou seja, (A == a).

Dica: Utilize as características da estrutura de dados Pilha para facilitar a resolução do problema.

Nota: Sua implementação de Pilha (código fonte) deve ser incorporada ao seu projeto.

Programa 2

Implemente em C++ as respectivas classes, atributos e métodos (incluindo construtores e destrutor) necessários para atender às especificações de uma **lista duplamente encadeada ordenada com sentinelas genérica**, ou seja, capaz de operar com qualquer tipo de dado. Note que o fato da lista ser ordenada implica em alterações apenas na inserção e eliminação dos itens.

Programa 3

Implemente em C++ as classes, atributos e métodos (incluindo construtores e destrutor) e programa principal necessários para controlar o cadastro de turmas, atendendo às seguintes especificações:

a) Cada turma deve permitir informar a listagem dos alunos na turma, a quantidade de alunos e a média das notas dos alunos;
b) Sobre cada aluno são guardadas algumas informações básicas, tais como matricula, como nome completo, total de faltas e nota (uma apenas por turma);
c) Cada turma concentra um conjunto de alunos que devem ser armazenados em uma lista (utilize a implementação do Programa 2);
d) Não deve ser permitida a inclusão duplicada de alunos na mesma turma;
e) Ao finalizar, o programa deve sempre salvar os dados atuais para um conjunto de arquivos (você deverá definir a estrutura necessária);

f) Ao iniciar o programa, caso os arquivos existam, os dados deverão ser lidos e instanciados nas classes correspondentes.

Dica: Faça uso de sobrecarga de operadores para realizar as operações necessárias sobre as listas.