INF1301 - Programação Modular
Período: 2014-2, Prof. Alessandro Garcia
Trabalho T2

Data de divulgação: *8 de setembro (segunda)*
Data de entrega: *22 de outubro (quarta-feira)*

1. Descrição do trabalho inicial

O trabalho deste semestre visa implementar um juiz do jogo de Xadrez. Na versão do jogo a ser implementada, o computador funcionará somente como um juiz entre dois jogadores humanos. Ao final do trabalho 4, a aplicação permitirá que peças sejam movimentadas de forma alternada como em uma partida de xadrez. Ao movimentar será informado se o movimento é legal ou não. Após cada movimento, o programa avisará se esta peça movimentada pode ser atacada por alguma peça oponente, ou seja, se esta ocupa uma das casas ameaçadas. No caso do Rei, caso ele esteja em uma casa atacada e não possa se movimentar para qualquer outra casa que não esteja ameaçada, o programa emite a mensagem "Cheque Mate". Caso ele esteja em uma casa atacada e possa se movimentar para outra casa não ameaçada, emite a mensagem "Cheque".

O objetivo desta etapa do trabalho prático (T2 – entrega até 22 de outubro) é implementar e testar separadamente os módulos que implementam as estruturas a serem utilizadas no jogo de Xadrez a ser implementado.

2. Descrição do segundo trabalho

Módulo Lista

Este módulo encapsula a estrutura de lista genérica a ser utilizada na aplicação. Todas as listas utilizadas no jogo de Xadrez deverão ser instanciadas a partir deste módulo. Seguem abaixo os comandos do *script* para teste da lista (e que deverão ter funções correspondentes disponibilizadas na interface). Para o **teste** do módulo de lista, o seu conteúdo será do tipo char.

```
=criarLista <inx> <idLista> <condRetorno>
```

Cria a cabeça de uma lista cuja identificação é <idLista> (esta informação fica armazenada na cabeça). A função de criação de lista retornará um ponteiro para a lista criada. <idLista> é um string de até 4 caracteres. Não esqueça de controlar a corretude dos dados de entrada no módulo de teste. O módulo de teste armazena o ponteiro para a lista em um vetor auxiliar na inx-ésima posição. Todos os comandos identificam as listas através de <inx>. Deve ser controlado sempre se o elemento existe no caso de operações sobre listas criadas, ou não no caso de operações que criam listas novas. Todas as funções que operam sobre listas existentes devem verificar a existência da lista sendo manipulada (e indicada no parâmetro um comando similar a `assert( pLista != NULL ) ;` Estude a documentação de C para saber o que esta função faz.

Condições de Retorno: 0 = Criação de lista bem sucedida
6 = Faltou memória

```
=obterIdLista <inx> <idLista>
```

Obtém a identificação da lista contida na sua cabeça e a compara com <idLista>. Cabe observar que a comparação é realizada no módulo de teste e não no módulo lista.

```
=inserirNo <inx> <char> <condRetorno>
```

Insere na lista <inx> após o nó corrente um novo nó contendo <char>. Caso a lista esteja vazia, insere o primeiro nó. O conteúdo para teste é caracter, porém esta lista será utilizada para armazenar conteúdos de tipo diferente de char quando estiver sendo utilizada no tabuleiro.

Condições de Retorno: 0 = inserção OK
6 = Faltou memória

```
=obterNo <inx> <char> <condRetorno>
```

Obter o valor contido no nó corrente e o compara com <char>.

Condições de Retorno: 0 = obteve corretamente OK
2 = lista vazia

`=excluirNoCorrente <inx>`

Exclui o nó corrente da lista <inx>. Caso a lista esteja vazia, não faz nada e emite retorno com erro.

`=irProx <inx> <condRetorno>`

O ponteiro para corrente passa a apontar para o próximo nó do corrente. Caso o corrente seja o último, o ponteiro não muda.

Condições de Retorno: 0 = Movimentação bem sucedida
2 = Lista vazia
4 = Nó corrente é o último

`=irAnt <inx> <condRetorno>`

O ponteiro para corrente passa a apontar para o nó anterior ao corrente. Caso o corrente seja o primeiro, o ponteiro não muda e emite retorno com erro.

Condições de Retorno: 0 = Movimentação bem sucedida
2 = Lista vazia
5 = Nó corrente é o primeiro

`=alterarNoCorrente <inx> <char> <condRetorno>`

Altera o conteúdo do nó corrente da lista <inx> para <char>.

Condições de Retorno: 0 = Alteração bem sucedida
2 = Lista vazia
3 = Lista não existe

`=destroiLista <inx> <condRetorno>`

Destrói a cabeça da lista e todos os nós existentes na lista <inx>. É importante que todos os espaços alocados sejam liberados com free.

Condições de Retorno: 0 = exclusão OK
3 = Lista não existe

Módulo Tabuleiro

Este módulo armazenará a estrutura identificadora de cada peça em uma matriz de 8 por 8 casas. Cada casa armazena uma identificação da peça nela contida e mais duas listas, a saber:
- lista `Ameacantes` contendo as casas que contém peças que legalmente ameaçam a presente casa
- lista `Ameacados` contendo as casas legalmente ameaçadas pela peça contida na presente casa

A estrutura identificadora da peça é composta pelo nome da peça e pela cor. Os nomes das peças são:

V = casa do tabuleiro não contém peça
T = Torre – pode mover n casas na horizontal ou vertical
C = Cavalo – em “L” 2 na vertical ou horizontal e 1 respectivamente na horizontal ou vertical
B = Bispo – n casas na diagonal
R = Rei – 1 casa em qualquer direção
D = Dama – n casas em qualquer direção
P = Peão – 1 casa em frente para mover e 1 casa na diagonal para capturar

As cores são:
V = casa do tabuleiro não contém peça
B = Branco
P = Preto

Nas coordenadas linha-coluna, as colunas são referenciadas por letras de A a H e as linhas por números de 1 a 8.

Para este trabalho, deve ser possível manipular esta estrutura através das seguintes funções (que deverão ter seus respectivos comandos de script de teste).

Função `InserirPeca` – Receberá a coordenada linha-coluna, o identificador da peça a ser inserida e a sua cor. Crie os retornos necessários inclusive prevendo a colocação da peça em uma coordenada inexistente.

Função `MoverPeca` – Receberá a coordenada de origem e a coordenada de destino. Esta função deverá verificar se a peça poderá executar este movimento e se capturará uma peça de outra cor. Caso isso aconteça, a peça oponente será retirada do tabuleiro. Crie os retornos necessários.

Função `RetirarPeca` – Receberá uma coordenada linha-coluna e a peça contida nesta casa será retirada. Crie os retornos necessários.

Função `ObterPeca` – Receberá uma coordenada linha-coluna e retornará a identificação da peça <Nome , Cor>.

Função `ObterListaAmeacantes` – Receberá uma coordenada linha-coluna e retornará o ponteiro para a correspondente cabeça de lista. Não se esqueça de projetar uma forma de operar sobre esta lista no módulo de teste.

Função `ObterListaAmeacados` – Receberá uma coordenada linha-coluna e retornará o ponteiro para a correspondente cabeça de lista.

Função `DestruirTabuleiro` – destrói o conteúdo de cada casa do tabuleiro e o próprio tabuleiro, caso este tenha sido alocado em memória dinâmica. Obs. no 4º. trabalho será examinado se ocorre vazamento de memória.

Projete a linguagem script de teste e desenvolva o correspondente interpretador. O script gerado para testar o tabuleiro ainda não simulará uma partida. Basta ter situações criadas com as funções acima. Tampouco cria os nós das listas. Estas são meramente criadas e permanecem vazias durante a execução do programa do trabalho 2.

3. Entrega do Trabalho

O trabalho deve ser feito em grupos de dois ou três alunos. Os programas devem ser redigidos em "C". Não será aceita nenhuma outra linguagem de programação. Todos os programas devem estar em conformidade com os padrões dos apêndices de 1 a 10 do livro-texto da disciplina. Em particular, os módulos e funções devem estar devidamente especificados. Recomenda-se fortemente a leitura do exemplo e do texto explanatório do arcabouço. Além de mostrar como implementar um teste automatizado dirigido por *script*, ilustra também as características de um programa desenvolvido conforme os padrões do livro.

O trabalho deve ser enviado por e-mail em um único arquivo `.zip` (codificação do *attachment*: MIME). Veja os *Critérios de Correção de Trabalhos* contidos na página da disciplina para uma explicação de como entregar.

O arquivo ZIP deverá conter:

- os arquivos-fonte dos diversos módulos que compõem os programas.
- os arquivos de *script* de teste desenvolvidos pelo grupo.
- um arquivo **MODELO.DOC** ou **MODELO.PDF** contendo:
    - a descrição de requisitos,
    - o modelo da *arquitetura* do programa,
    - os modelos *físicos* de todas as estruturas criadas para este trabalho. Para cada modelo deve ser mostrado também um *exemplo físico* e as *assertivas estruturais.*
- o programa executável: **TRAB2-1.EXE**. Caso queiram entregar mais de um executável, estes deverão ter o nome no formato: **TRAB2-2.EXE**, **TRAB2-3.EXE**, e assim por diante. Cada módulo deverá ser testado separadamente e cada um terá o seu programa executável. Planeje a seqüência de desenvolvimento que permita o teste minimizando o desenvolvimento de módulos de apoio ao teste (*stubs* ou enchimentos).

- um arquivo **LEIAME.TXT** contendo a explicação de como utilizar o(s) programa(s).
- O *script* de composição dos construtos (**.comp**), os arquivos *script* **.make** e os arquivos *script* **.build** para o *linker* gerados pelo utilitário **GMAKE** disponível no arcabouço de teste. Deverão ser desenvolvidos construtos para testar cada um dos módulos.
- Os arquivos *batch* (**.bat**) que coordenam a execução dos testes. (veja o exemplo **testatudo.bat** no arcabouço).
- Tantos arquivos **RELATORIO-nome.TXT** quantos forem os membros do grupo. O tema **nome** deve identificar o membro do grupo ao qual se refere o relatório. Estes arquivos devem conter uma tabela de registro de trabalho organizada como a seguir:

Data, Horas Trabalhadas, Tipo Tarefa, Descrição da Tarefa Realizada

Na descrição da tarefa redija uma explicação breve sobre o que o componente do grupo fez. Esta descrição deve estar de acordo com o Tipo Tarefa. Cada Tipo Tarefa identifica uma natureza de atividade que deverá ser discriminada explicitamente, mesmo que, durante uma mesma sessão de trabalho tenham sido realizadas diversas tarefas. Os tipos de tarefa são:

- estudar
- especificar os módulos
- especificar as funções
- revisar especificações
- projetar
- revisar projetos
- codificar
- revisar código
- planejar teste
- revisar plano de teste
- testar
- corrigir

Dica 1: Preencha esta tabela de atividades ao longo do processo. **NÃO DEIXE PARA ÚLTIMA HORA, POIS VOCÊ NÃO SE LEMBRARÁ DO QUE FEZ TAL DIA, TAL HORA**. Com relatórios similares a esse você aprende a planejar o seu trabalho.

Importante: Ao entregar o arquivo ZIP, este **SOMENTE DEVERÁ CONTER OS ARQUIVOS RELACIONADOS A ESTE TRABALHO**. Caso existam outros arquivos incluídos ao compactar toda pasta do arcabouço e enviar (por exemplo: os arquivos **.bak**, os de trabalho criados pelo ambiente de desenvolvimento usado, etc.) o grupo **perderá 2 pontos**. GASTE UM TEMPO SELECIONANDO TODOS OS ARQUIVOS QUE VOCÊ EFETIVAMENTE PRECISA ENVIAR PARA PODER INSPECIONAR, MODIFICAR, USAR E RECOMPILAR OS PROGRAMAS ENTREGUES.

Observações:

- A mensagem de encaminhamento deve ter o assunto (*subject*) **INF1301-Trab02-*idGrupo***. O ***idGrupo*** deve ser formado pelas iniciais dos nomes dos membros do grupo. O texto da mensagem deve conter somente a lista de alunos que compõem o grupo (formato: número de matrícula, nome e endereço do e-mail). Perde-se **2 pontos** caso não seja encaminhado desta forma. Mais detalhes podem ser encontrados no documento *Critérios de Correção dos Trabalhos* disponível na página da disciplina.
- O programa será testado utilizando o programa compilado fornecido. Deve rodar sem requerer bibliotecas ou programas complementares. O sistema operacional utilizado durante os testes será o Windows 7.

4. Critérios de correção básicos

Leia atentamente o documento *Critérios de Correção dos Trabalhos* disponível na página da disciplina. Muitas das causas para a perda substancial de pontos decorrem meramente da falta de cuidado ao entregar o trabalho. Este documento também pode ser encontrado no anexo à descrição da disciplina distribuída na primeira aula.

Não deixem para a última hora.
Este trabalho dá trabalho!