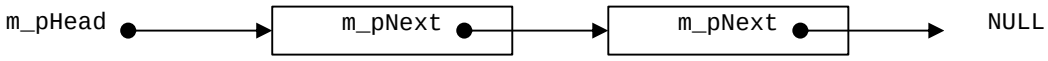
Aula 5

Objectivo: Construção de classes para estruturas de dados comuns.

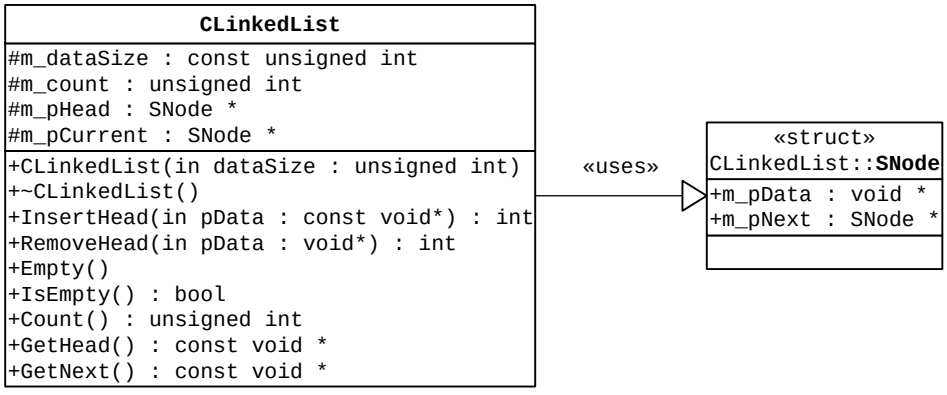
Exercício 1

Uma lista ligada é uma estrutura de dados dinâmica, que permite inserir e remover arbitrariamente elementos durante a execução do programa. O número máximo de elementos da lista não é conhecido à partida, sendo limitado apenas pela quantidade de memória disponível no sistema. Numa lista ligada, cada elemento possui um ponteiro para o elemento seguinte. Existe ainda um ponteiro adicional para o primeiro elemento da lista (o elemento que se encontra à cabeça da lista). Quando a lista está vazia este ponteiro possui o valor `NULL`. O ponteiro para o elemento seguinte do último elemento aponta sempre para `NULL`, marcando o final da lista.



Numa lista biligada cada elemento possui dois ponteiros, um para o elemento seguinte e outro para o elemento anterior. Possui ainda dois ponteiros adicionais, um para o elemento que se encontra à cabeça da lista e um para o elemento que se encontra na cauda da lista. Inicialmente, ambos possuem o valor `NULL`.

1.1 Construa uma classe chamada `CLinkedList` que implemente uma lista ligada genérica para armazenar objectos de qualquer tipo representados e manipulados por ponteiros do tipo `void*`. Para simplificar, a inserção e remoção de elementos da lista deve ser sempre efectuada numa das extremidades (por exemplo a cabeça).



Para tornar a utilização da lista mais segura, durante a inserção de um elemento deve ser feita uma cópia do respectivo objecto para uma zona de memória gerida internamente pela lista. Na remoção de um elemento deve-se copiar o conteúdo dessa zona interna para uma zona externa acessível através de um ponteiro fornecido pelo utilizador da lista. Por este motivo durante a construção de uma lista ligada deve-se indicar ao respectivo construtor o tamanho em *bytes* dos elementos que vão ser armazenados na lista.

O construtor da lista deve ter o seguinte protótipo:

`CLinkedList(unsigned int dataSize);`
O parâmetro de entrada `dataSize` indica o tamanho em *bytes* dos elementos a armazenar.

A classe `CLinkedList` deve disponibilizar os seguintes métodos:

`int InsertHead(const void* pData);`
Este método insere um elemento na cabeça da lista.
O parâmetro `pData` é um ponteiro para o elemento a inserir.
Deve devolver:

- `NULL_PTR` se o ponteiro `pData` possuir o valor `NULL`;
- `OUT_OF_MEMORY` se não for possível alocar memória para fazer a cópia do elemento a inserir ou para a sua estrutura de controlo;
- `OK` se a inserção for bem sucedida.

`int RemoveHead(void* pData);`
Este método remove o elemento que se encontra à cabeça da lista.
O parâmetro `pData` é um ponteiro fornecido pelo utilizador da lista para a zona de memória para onde deve ser copiado o elemento a remover.
Deve devolver:

- `LIST_EMPTY` se a lista estiver vazia;
- `NULL_PTR` se o ponteiro `pData` possuir o valor `NULL`;
- `OK` se a remoção for bem sucedida.

`void Empty();`
Este método apaga todos os elementos armazenados na lista.

`bool IsEmpty();`
Devolve `true` se a lista estiver vazia ou `false` em caso contrário.

`unsigned int Count();`
Devolve o número de elementos armazenados na lista.

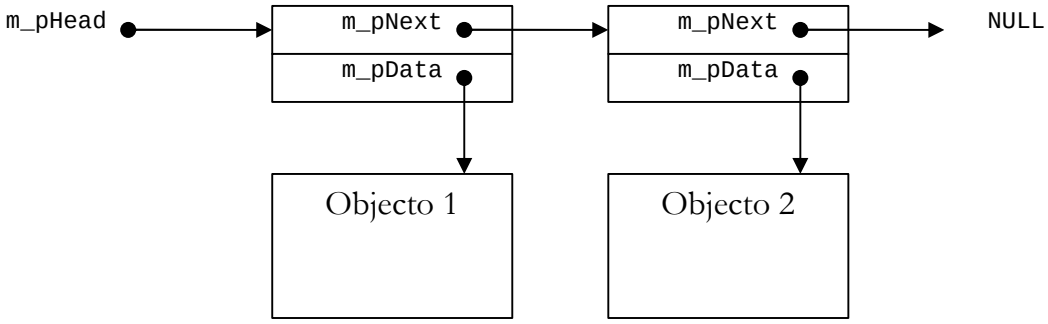
Para percorrer uma lista ligada, isto é, aceder a cada elemento, do primeiro até ao último elemento devem ser usados os métodos `GetHead` e `GetNext`. O primeiro elemento obtêm-se usando o método `GetHead`. Os elementos seguintes são obtidos através de invocações sucessivas do método `GetNext`. Isto significa que a classe deve manter internamente um ponteiro para o "elemento actual" da lista, isto é, o último elemento devolvido pelos métodos `GetHead`/`GetNext`. É importante referir que qualquer operação que afecte o estado interno da lista pode invalidar esse ponteiro.

`const void* GetHead();`
Este método devolve o elemento que se encontra à cabeça da lista ou `NULL` no caso da lista se encontrar vazia. Actualiza também o ponteiro para o "elemento actual" de forma a que este aponte para o elemento devolvido. Os elementos seguintes são obtidos através de invocações sucessivas do método `GetNext`.

`const void* GetNext();`
Este método devolve o próximo elemento da lista ou `NULL` no caso de ter sido atingido o fim da lista. Deve também actualizar o ponteiro para o "elemento actual".

De forma a separar os elementos (dados), da estrutura de controlo da lista ligada, a lista é constituída internamente por um conjunto de nodos, um por cada elemento de informação, definidos pela estrutura `SNode`:

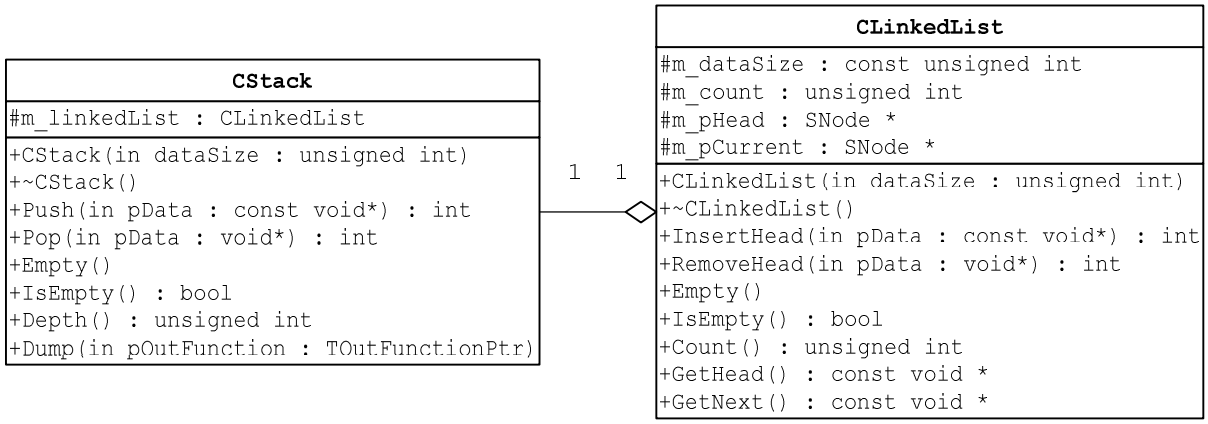
```
struct SNode
{
    void* m_pData;  //ponteiro para os dados do elemento
    SNode* m_pNext; //ponteiro para o próximo nodo da lista
};
```



Exercício 2

Como aplicação da lista ligada desenvolvida no exercício anterior, pretende-se agora implementar uma memória do tipo *stack* (pilha) usando uma solução dinâmica. Num *stack*, a retirada de elementos faz-se por ordem inversa da inserção usando primitivas do tipo *push*/*pop*. A utilização da lista ligada do exercício anterior permite a implementação de uma solução dinâmica em que, quer o tamanho de cada elemento, quer o número máximo de elementos a armazenar no *stack* não são conhecidos à partida.

2.1 Implemente uma classe chamada `CStack`, que possa armazenar elementos de um tipo de dados abstracto manipulados através de ponteiros do tipo `void*`.



O construtor deve ter o seguinte protótipo:

`CStack(unsigned int dataSize);`
O parâmetro de entrada `dataSize` indica o tamanho em *bytes* dos elementos a armazenar.

A classe `CStack` deve disponibilizar os seguintes métodos:

`int Push(const void* pData);`
Insere no topo do *stack*, isto é, na cabeça da lista, o elemento apontado por `pData`.
Devolve o código de erro reportado pela função de inserção na lista ligada.

`int Pop(void* pData);`
Retira um elemento do topo do *stack*, isto é, da cabeça da lista, e copia-o para a zona de memória apontada por `pData`.
Devolve o código de erro reportado pela função de remoção da lista ligada.

`void Empty();`
Apaga todos os elementos armazenados no *stack*.

`bool IsEmpty();`
Verifica se o *stack* está vazio.
Devolve `true` se não existirem objectos armazenados no *stack* ou `false` em caso contrário.

`unsigned int Depth();`
Devolve o número de elementos armazenados no *stack*.

`void Dump(TOutFunctionPtr pOutFunction);`
Esta função escreve na *stream* standard de saída (`cout`) o conteúdo do *stack*, do topo para a base. Deve utilizar as funções `GetHead` e `GetNext` da classe `CLinkedList`. No caso do *stack* se encontrar vazio, deve escrever a mensagem "*Stack vazio*".
O parâmetro `outFunction` é um ponteiro para uma função, fornecida pelo utilizador, apropriada para escrever no ecrã os elementos armazenados no *stack*.

Exercício 3

Considere a seguinte expressão:

```
(2 + 4.1) * (3.3 + 5) – 1.2
```

Para determinar o resultado desta expressão fazem-se os seguintes cálculos: soma-se o 2 com o 4.1 e guarda-se o resultado; soma-se o 3.3 com o 5 e guarda-se o resultado; multiplicam-se os dois resultados anteriores e guarda-se o resultado; finalmente subtrai-se 1.2 ao último resultado para se obter o resultado final.
Uma notação alternativa a esta, designada notação polaca, permite representar a mesma expressão sem recurso a quaisquer parêntesis – na realidade permite representar qualquer expressão sem recurso a parêntesis. Eis a representação em notação polaca da expressão anterior:

```
2 4.1 + 3.3 5 + * 1.2 -
```

O cálculo deste tipo de expressões é feito da seguinte maneira:

- Sempre que aparece um número, guarda-se;
- Sempre que aparece um operador binário (que utilize dois operandos), sacam-se os dois últimos números guardados, aplica-se-lhes o operador e guarda-se o resultado;
- Sempre que aparece um operador unário (que utilize apenas um operando), saca-se o último número guardado, aplica-se-lhe o operando e guarda-se o resultado.

A manipulação dos operandos pode ser facilmente implementada usando um *stack*, resultando o cálculo da expressão anterior nos seguintes passos:

- Entra a *string* "2" e guarda-se o 2;
- Entra a *string* "4.1" e guarda-se o 4.1;
- Entra a *string* "+", sacam-se o 4.1 e o 2 e guarda-se o 6.1 (de 2+4.1);
- Entra a *string* "3.3" e guarda-se o 3.3;
- Entra a *string* "5" e guarda-se o 5;
- Entra a *string* "+", sacam-se o 5 e o 3.3 e guarda-se o 8.3 (de 5+3.3);
- Entra a *string* "*", sacam-se o 8.3 e o 6.1 e guarda-se o 50.63 (de 8.3*6.1);
- Entra a *string* "1.2" e guarda-se o 1.2;
- Entra a *string* "-", sacam-se o 1.2 e o 50.63 e guarda-se o 49.43 (de 50.63-1.2).

Se considerar que dispõe de um visor onde o último número guardado é afixado, o procedimento anterior acaba com o 49.43 afixado nesse visor.

3.1 Faça um programa que usando a classe `CStack` implemente uma máquina de calcular em notação polaca. A máquina de calcular deve aceitar operandos do tipo real, os operadores binários "+", "-", "*" e "/" e os operadores unários "sim" e "inv" que calculam o valor simétrico e o inverso do seu operando, respectivamente. Para o auxiliar na construção deste programa é fornecido o módulo *Token.h/Token.cpp* com a função *GetToken* que analisa uma *string* dada e identifica o operando ou operador que ela representa. Esta função devolve um *token* (número inteiro) que representa o operador ou operando. No caso de um operando fornece também o respectivo valor numérico real.
A tabela seguinte mostra o valor do *token* para diferentes *strings*.

String	Significado	Token
ABC123	*Token* inválido	-1
55.5	Operando	0
+	Adição (operador)	1
-	Subtracção (operador)	2
*	Multiplicação (operador)	3
/	Divisão (operador)	4
sim	Simétrico (operador)	5
inv	Inverso (operador)	6

3.2 Utilizando o método `Dump` da classe `CStack` faça com que o conteúdo do *stack* seja escrito no ecrã após cada operação de `Push`.