Aula 6

Objectivo: Redefinição de operadores;
Utilização de construtores de cópia;
Funções e atributos static;
Alocação/Realocação e libertação de memória.

Exercício 1

1.1 O objectivo deste exercício é a construção de uma classe chamada `CDate` para manipular datas. Esta classe deve possuir atributos para armazenar:

- o dia (`m_day`),
- o mês (`m_month`),
- o ano (`m_year`) e
- os nomes dos meses, como por exemplo "Jan", "Fev", etc. (`static char* m_monthNames[12]`). Note que sendo *static* esta variável é única, independentemente do número de instâncias de `CDate`. Deve ser inicializada no início do módulo *date.cpp*, antes de qualquer outra operação.

Os métodos da classe `CDate` que deverá implementar são:

- um construtor por defeito:
```
CDate();
```
- um construtor de cópia:
```
CDate(const CDate& date);
```
- um destrutor:
```
~CDate();
```
- os operadores == e != para comparar datas:
```
bool operator==(const CDate& date);
bool operator!=(const CDate& date);
```
- o operador = para efectuar atribuições de objectos do tipo `CDate`:
```
CDate& operator=(const CDate& date);
```
- os operadores + e - para adicionar ou retirar um determinado número de dias à data actual de uma dada instância de `CDate`:
```
CDate operator+(unsigned int nDays);
CDate operator-(unsigned int nDays);
```
- os operadores >> e << para efectuar leituras e escritas do/para o ecrã de objectos do tipo `CDate`:
```
friend istream& operator>>(istream& is, CDate& date);
friend ostream& operator<<(ostream& os, const CDate& date);
```
- uma função que calcule o número de dias de um dado mês:
```
static unsigned int DaysOfMonth(unsigned int month);
```

1.2 Considere as seguintes instruções em C++:
```
CDate date1, date2;
date2 = date1 + 30;  // linha 1
date1 = date2;  // linha 2
```

Diga quais os métodos da classe `CDate` que irão ser invocados e a respectiva ordem de invocação quando a linha 1 e linha 2 do código C++ apresentado acima forem executadas.

Exercício 2

2.1 Construa um projecto chamado *ArrayDinamico* e acrescente uma classe chamada `CArrayInt`, que possa armazenar um número ilimitado de elementos do tipo inteiro. Tal como o nome da classe sugere, irá construir um array de elementos do tipo inteiro com a particularidade de este ser capaz de crescer em tamanho de forma a possibilitar o armazenamento de mais elementos quando o array estiver cheio e de decrescer em tamanho quando o número de posições livres do array exceder um determinado valor.
Assuma que inicialmente o array se encontra vazio. Cada vez que se pretende inserir um elemento e seja necessário alocar memória, faça com que o array cresça N posições, sendo N um valor constante. Se o número de posições livres exceder o valor de N então faça com que o array diminua de tal forma que as posições livres não excedam N.

Utilize as funções `malloc`, `realloc` e `free` para gerir a memória utilizada pelo array. Para poder invocar estas funções deve incluir o ficheiro `memory.h`.

Atributos

`int *m_pArray;`
Ponteiro para o primeiro elemento do array.

`int m_count;`
Número de elementos armazenados no array.

`int m_size;`
Tamanho real do array alocado em número de elementos inteiros. Inclui posições ocupadas por elementos e posições que foram alocadas mas que ainda não foram preenchidas.

`const int m_allocSize;`
Número de elementos inteiros que deverá ser utilizado para efectuar realocações de memória.

Métodos

O construtor por defeito e o construtor de cópia devem ter os seguintes protótipos:

```
CArrayInt(int allocSize = N);
CArrayInt(const CArrayInt& arInt);
```

O parâmetro de entrada `allocSize` indica o número de inteiros que o array deve crescer para que possa armazenar novos elementos.

Para gerir o armazenamento de inteiros no seu array dinâmico deve construir os seguintes métodos:

`int Append(int data);`
Adiciona o elemento `data` à última posição do array.
Devolve `OUT_OF_MEMORY` se não foi possível alocar memória para esse elemento ou `OK` se a operação tiver sido bem sucedida.

`int Delete(int index);`
Retira do array o elemento armazenado na posição `index`. Para que todos os elementos do array ocupem posições contíguas, após a remoção de um elemento deve-se deslocar para a esquerda os elementos que se encontram à sua direita.
Devolve `INVALID_INDEX` se o índice indicado no argumento não for válido, `INDEX_EMPTY` se o índice aponta para uma posição do array onde não foi armazenado qualquer elemento, ou `OK` caso seja possível removê-lo.

`int Insert(int index, int data);`
Insere o elemento `data` na posição `index`. Caso o array esteja vazio, sem posições livres, ou se index especificar uma posição vazia, o elemento deverá ser adicionado ao array utilizando o método `Append`. Por outro lado, se o array tiver espaço para inserir o elemento mas a posição indicada por `index` já estiver ocupada, então os elementos de índice superior devem ser deslocados para a direita de forma a que o novo elemento seja guardado na posição pretendida.
Devolve `OUT_OF_MEMORY` se não foi possível alocar memória para esse elemento ou `OK` se a operação tiver sido bem sucedida.

`int DeleteAll(int data);`
Remove do array todas as ocorrências do elemento `data`. Para simplificar copie para um novo array, chamado `arrayAux` por exemplo, todos os elementos diferentes de `data`. Liberte a memória ocupada pelo atributo da classe `m_pArray` e utilize o operador = de forma a que `m_pArray` tome o valor de `arrayAux`.
Devolve `OUT_OF_MEMORY` se a alocação de memória não foi conseguida ou `OK` em caso contrário.

`int Get(int index, int& data);`
A função deverá colocar em `data` o valor armazenado no array na posição `index`. Se o índice especificado não for válido a função deve devolver `INVALID_INDEX`. Se a operação for bem sucedida a função devolve `OK`.

`int Set(int index, int data);`
A função coloca o valor `data` na posição `index` do array. O valor do índice especificado deve ser validado de forma a que a função possa devolver `INVALID_INDEX` em caso de erro. Se a operação for bem sucedida a função deve devolver `OK`.

`CArrayInt operator=(CArrayInt& arInt);`
O operador de atribuição deve ser construído de forma a que o array especificado no primeiro operando seja uma cópia do segundo operando, isto é, os seus atributos devem tomar os mesmos valores.

`int& operator[](int index);`

O operador [ ] deve devolver o valor armazenado em `m_pArray` na posição `index`. Assim, dadas as variáveis:

```
int a;
CArrayInt b;
```

A intrução `a = b[0];` é válida. Como o valor de retorno deste operador é do tipo `int&`, a instrução `b[0] = a;` é igualmente válida.

Ao contrário dos métodos `Get` e `Set`, este operador não tem a possibilidade de efectuar a validação de `index`.

`ostream& operator<<(ostream& os, const CArrayInt& arInt);`

A função deste operador é apresentar no ecrã o valor de todos os elementos armazenados no array. De notar que o número de elementos armazenados pode ser inferior ao tamanho do array.

`int GetCount();`

Este método devolve o número de elementos efectivamente armazenados no array.