Aula 10

Objectivos: *Templates* (polimorfismo paramétrico).

Problema A1

Construa uma classe `ARRAY` que implemente um `array` dinâmico genérico (ou seja cujos elementos podem ser de um qualquer tipo). Esta classe deve utilizar adequadamente "templates" e deve ter os operadores normais na utilização de um `array`, e deve estar devidamente protegida contra maus usos (uso de um índice inválido por exemplo).

```
template <class T>
class ARRAY
{
public:
  ARRAY(int indice_minimo,int indice_maximo);
  ARRAY(int num_elementos); // índices de 0 a num_elementos-1
  virtual ~ARRAY();

  ARRAY(ARRAY&outro);
  ARRAY &operator =(ARRAY&outro);

  virtual T &operator [](int indice);

  virtual int indice_minimo(void);
  virtual int indice_maximo(void);
  virtual int tamanho(void);

// (...)
};
```

A1.1. Teste a classe instanciando-a, por exemplo, para um array de números inteiros.

A1.2. Faça procedimentos globais (com "templates") que permitam a ordenação de "arrays" um com o algoritmo de selecção e outro com o de bolha.

(**Nota:** Assuma que os tipos de dados ordenáveis têm implementadas os operadores de relação de ordem `<` e `>`.)

A1.3. Crie uma nova classe genérica: `ARRAY_ORDENAVEL`, que acrescenta a

possibilidade de ordenação do `array` (logo só é aplicável a elementos que implícita ou explicitamente estabeleçam uma relação de ordem).
Esta nova classe deve ser descendente da classe genérica `ARRAY` e para além da interface desta deve incluir os seguintes serviços:

```
template <class T>
class ARRAY_ORDENAVEL: public ARRAY
{
public:
  void define_proc_ordenacao(void (*proc_ord)(ARRAY_ORDENAVEL &arr));
  void ordena(void);
  T maximo(void);
};
```

Assim sem prejuízo de a classe estar à partida ligada a um algoritmo de ordenação especifico (por exemplo o de selecção), esta permite que este seja redefinido fazendo uso da possibilidade que existe em *C++* de utilizar variáveis (e atributos) do tipo ponteiro para funções.

(**Nota:** A declaração de uma variável ponteiro para uma função `void` com uma argumento inteiro é feita da seguinte forma:

```
void (*p_func)(int arg)); )
```

A1.4. Experimente a classe para vários tipos de elementos (`int` e `DATA&`), e para vários algoritmos de ordenação.

A1.5. Implemente e experimente o método `maximo`, que deve devolver o valor máximo existente no "array".

Problema B1

B1.1. Implemente um `array` de ponteiros parametrizável (`TArray`). Este deve incluir o construtor, o destrutor, a função `Insert` e a função `Max` (que devolve o "maior" elemento existente no `array`). O destrutor deve apagar todos os ponteiros bem como todos os objectos apontados pelos ponteiros.

B1.2. Construa uma classe aninhada (*nested*) chamada `iterator` que vai implementar um iterador constante. Esta inclui uma referência para `TArray`, um índice (para guardar a posição actual dentro do `array`). Implemente o construtor, o operador `++` postfix/prefix (para passar para a posição seguinte), operador `*` (para devolver um elemento do `array` cuja posição coincide com o índice), os operadores `==`, `!=` (para comparar dois iteradores) e `<<` (para visualizar o elemento do `array` cuja posição coincide com o índice).

B1.3. Adicione à classe `TArray` as funções `begin` e `end` que geram dois iteradores o primeiro dos quais aponta para o início do `array` e o segundo para o fim. A interface das classes é a seguinte:

```
template <class T> class TArray
{
  T** m_array;
  unsigned m_nSize;

public:
  class iterator;
  friend class iterator;
  class iterator
  {
    const TArray& t;
    unsigned index;
  public:
    iterator(const TArray<T>& a, unsigned s = 0)
      : t(a), index(s) {}
    const T* const operator++ () { /* */ }
    const T* const operator++ (int) { /* */}
    const T* const operator* () const { /* */ };

    bool operator== (const iterator& r) const { /* */}
    bool operator!= (const iterator& r) const { /* */}
    friend std::ostream operator << (std::ostream& os,
      const iterator& it);
  };
```

```
  TArray();
  virtual ~TArray();

  void Insert (T* t);
  const T* const Max() const;

  const iterator begin() const;
  const iterator end() const;
};
```

B1.4. Implemente a função global `template <class IT> void Visualizar (const std::string& text, IT in, const IT& f)`. Esta aceite um texto que deve imprimir no écran e dois iteradores (para imprimir tudo para que estes apontam).

B1.5. Teste tudo com a função `main` seguinte:

```
int main()
{
  using std::endl;  using std::cout;

  TArray<int> ar_int;
  for (unsigned i = 0; i < 10; i++)
    ar_int.Insert(new int(i));

  Visualizar("Template de inteiros: ", ar_int.begin(), ar_int.end());
  cout << "Maximo: " << *ar_int.Max() << endl << endl;

  TArray<double> ar_dob;
  for (i = 0; i < 10; i++)
    ar_dob.Insert(new double (i*2.3));

  Visualizar("Template de doubles: ", ar_dob.begin(), ar_dob.end());
  cout << "Maximo: " << *ar_dob.Max() << endl << endl;

  TArray<CFigura> ar_fig;
  ar_fig.Insert(new CCirculo (4));
  ar_fig.Insert(new CQuadrado (6));
  ar_fig.Insert(new CRectangulo (2,7));

  Visualizar("Template de figuras: ", ar_fig.begin(), ar_fig.end());
  cout << "Maximum: " << *ar_fig.Max() << endl << endl;

  return 0;
}
```