Aula 11

Objectivos: Excepções e problemas finais.

Problema A1

Em geral, a técnica utilizada quando se pretende o desenvolvimento de programas tolerantes a falhas (ou seja, programas que mesmo na presença de [certos] erros, continuam a dar resultados correctos), consiste no uso (criterioso) de redundância funcional de código (implementações diferentes para as mesmas funcionalidades).

(**Nota:** Não confundir este tipo de redundância com a indesejável repetição de código algoritmicamente similar pelo programa.)
Neste problema pretende-se a implementação de uma estrutura simples de tolerância a erros no cálculo da função `seno(x)`, fazendo uso (na ordem apresentada) dos seguintes algoritmos:

1. $seno\_1(x) = x\ , \qquad erro \leq \left|x^3/3!\right|$
2. $seno\_2(x) = x - x^3/3!\ , \qquad erro \leq \left|x^5/5!\right|$
3. $seno\_3(x) = \Sigma_{i=0}^{3} \frac{(-1)^i * x^{2*i+1}}{(2*i+1)!}\ , \qquad erro \leq \left|x^9/9!\right|$

(Estes algoritmos assentam no desenvolvimento em série de Maclaurin para a função $seno(x)$)

A1.1. Construa um módulo de cálculo numérico de funções matemáticas, que permita o cálculo numérico da função $seno(x)$. Este módulo deve permitir que se defina um erro (absoluto) máximo aceitável no cálculo, e o cálculo do *seno* deve ser feito por tentativas (fazendo uso do mecanismo de excepções do *C++*), desde o algoritmo 1, até que o erro seja aceitável ou então até ao algoritmo 3. Caso nenhum algoritmos dê um resultado aceitável o módulo deve propagar a excepção para os seus clientes (mas apenas nesta situação).

A1.2. Faça com que as excepções geradas levem o valor o erro máximo da estimativa de cada algoritmo.

Problema A2

Uma das metodologias que nas últimas décadas mais tem contribuído para o melhoramento da correcção (capacidade de um programa funcionar de acordo com a sua especificação) do software, é a chamada programação por contrato. Esta técnica consiste, essencialmente, na associação de asserções (condições) às várias entidades dos programas (asserções são condições que têm sempre que se verificar no sitio onde são aplicadas).
Há vários tipos possíveis de asserções: pré-condições – condições a verificar na chamada de rotinas (funções, procedimentos, ou métodos de classes), pós-condições (a verificar no fim da execução de rotinas), invariantes (propriedades das classes), e outras asserções de verificação algoritmica.
Um exemplo simples de pré-condições e pós-condições são as que se aplicam à função `sqrt`. Esta função só pode ser utilizada sse o seu argumento for um número não negativo:

```
double sqrt(double n)
{
  requer(n >= 0);

  double result;

  (...)

  garante(result * result == n);

  return result;
}
```

(**Nota:** Até agora temos usado o `assert` para este fim.)
Um dos "problemas" levantados com o uso de asserções é a forma como deve o programa reagir quando elas não se verificam. Sendo verdade que na grande maioria das situações o comportamento adequado será o de terminar de imediato o programa relatando e identificando o ponto onde a asserção falhou (especialmente durante o desenvolvimento do programa), há certas situações (tolerância a falhas) onde tal não pode ocorrer (por exemplo num sistema de piloto automático de um avião). As excepções do *C++* dão-nos a possibilidade de adaptar o comportamento das asserções às nossas necessidades (garantindo, por defeito, que o comportamento é o de terminar o programa).

Neste problema pretende-se implementar as seguintes asserções, dando a possibilidade de se detectar a ocorrência de uma qualquer destas falhas de asserções (ou uma qualquer delas), fazendo uso das excepções do *C++*:

- pré-condições: `requer`;
- pós-condições: `garante`;
- verificações: `verifica`.

(Lamentavelmente, os invariantes não são facilmente implementáveis em *C++*, pelo que serão omitidos)

(**Nota:** Para dar a possibilidade de detecção de uma qualquer destas asserções, podem-se criar hierarquias de classes de excepções.)

A2.1. Aplique estas asserções nas classes `ARRAY` desenvolvidas na aula anterior.

Problema A3

Projecte e construa um conjunto de módulos que permitam a codificação e descodificação de mensagens (de texto), de forma a que estes sejam facilmente extensíveis para novos tipos de codificadores.
Numa primeira versão implemente apenas um codificador que vire a mensagem ao contrário (`"batata"` passa a `"atatab"`).
Sugere-se que este problema seja abordado da seguinte forma:

1. Identifique as classes (módulos) que fazem sentido criar, tentando que estas sejam tão abstractas (genéricas) quanto possível;
2. Construa (testando) essas classes;
3. Faça um programa que permita a codificação, envio, recepção e descodificação de mensagens (atenção que o emissor e o receptor apenas interagem através do **texto** da mensagem codificada).

A3.1. Acrescente os seguintes codificadores:

- XOR (operador `^` do *C++*);
- Converta as letras minúsculas para maiúsculas, rodando duas posições para a frente (`"a"` passa a `"C"`, `"x"` a `"Z"`, `"y"` a `"A"`, etc.).

Problema B1

B1.1. Crie uma classe `Exception` que inclui uma variável membro que um objecto da classe `string`. Crie a classe `ERangeError` que deve incluir mais dois membros do tipo `unsigned`: `m_Max` (para indicar o máximo permitido) e `m_Used` (para indicar o índice usado) inicializados no construtor. As interfaces das classes devem ser as seguintes:

Exception.h //a implementação está já incluída! Não precisam de fazer mais nada!

```
class Exception
{
  const std::string m_sDescr;
public:
  const std::string& What() const { return m_sDescr; }
  Exception(const std::string& descr) : m_sDescr(descr)  { }
  Exception(const Exception& ma) : m_sDescr(ma.m_sDescr) { }
  virtual ~Exception() { }
};
```

ERangeError.h //falta implementar o construtor e o construtor de cópia

```
class ERangeError : public Exception
{
  const unsigned m_Max;
  const unsigned m_Used;
public:
  ERangeError(const std::string& descr, unsigned max,unsigned used);
  ERangeError(const ERangeError& ma);
  virtual ~ERangeError() {}

  unsigned Max() const { return m_Max; }
  unsigned Used() const { return m_Used; }

};
```

B1.2. Construa a classe `CBooleanVector` que representa um vector booleano. A classe `CBooleanVector` deve incluir uma classe aninhada ("nested") amiga chamada `CProxy`. Esta classe inclui um membro que é uma referência para um elemento do `CBooleanVector`. Implemente todas as funções que são declaradas na definição das classes:

BooleanVector.h

```
class CBooleanVector
{
  unsigned m_nElements;
  unsigned* m_arElements;

  class CProxy;
  friend class CProxy;
  class CProxy
  {
    unsigned& m_VectorElement;
  public:
    CProxy(unsigned& VectorElement);
    CProxy& operator= (unsigned r);   //lvalue
    CProxy& operator|= (unsigned r);  //lvalue
    operator unsigned() const;        //rvalue
  };

public:

  CBooleanVector(unsigned el = 0);
  CBooleanVector(const CBooleanVector& v);

  virtual ~CBooleanVector();

  const CProxy operator[](unsigned pos) const;
  CProxy operator[](unsigned pos);

  CBooleanVector& operator=(const CBooleanVector& rv);
};
```

B1.3. Implemente a classe `CBooleanMatrix` que contém um `array` de vectores booleanos. Implemente todas as funções que são declaradas na definição da classe:

```
class CBooleanMatrix
{
  unsigned m_nRows;
  unsigned m_nColumns;
  CBooleanVector* m_arVectors;

public:
  CBooleanMatrix(unsigned r = 0, unsigned c = 0);
  CBooleanMatrix(const CBooleanMatrix& rm);
```

```
  virtual ~CBooleanMatrix();

  friend const CBooleanMatrix operator* (const CBooleanMatrix& lm,
                                         const CBooleanMatrix& rm);
  CBooleanMatrix& operator= (const CBooleanMatrix& rm);

  const CBooleanVector& operator[](unsigned row) const;
  CBooleanVector& operator[](unsigned row);

  friend std::ostream& operator << (std::ostream& os,
                                    const CBooleanMatrix& rm);
};
```

B1.4. Incorpore o mecanismo de controlo de excepções. Todas as funções que trabalham com índices, devem gerar excepções do tipo `ERangeError`. A excepção `Exception` é gerada quando se tenta atribuir a algum elemento da matriz um valor que não é booleano.

B1.5. Implemente uma função global que é `void terminator()`. Faça com que esta função seja chamada quando existe alguma excepção não processada. A função `terminator` deve imprimir no écran o texto que não se sabe o que fazer e concluir a execução.

B1.6. Teste o código com a função main seguinte:

```
class CBooleanMatrix
int main()
{
  using namespace std;

  try
  {
    CBooleanMatrix mb1(3, 2);
    CBooleanMatrix mb2(2, 4);
    CBooleanMatrix mb3(3, 4);

    //mb1 * mb3;  //descomentar esta linha e testar

    //mb1[0][0] = 5;  //descomentar esta linha e testar
    //mb1[10][0] = 1; //descomentar esta linha e testar
    //mb1[0][4] = 1;  //descomentar esta linha e testar

    mb1[0][0] = 1;   mb1[1][1] = 1;
    mb1[2][0] = 1;   mb1[2][1] = 1;
```

```
    mb2[0][2] = 1;   mb2[0][3] = 1;
    mb2[1][1] = 1;   mb2[1][3] = 1;

    cout << mb1 << endl;
    CBooleanVector v(2);
    v[1] = 1;
    cout << mb1 << endl;
    cout << mb2 << endl;
    cout << mb1 * mb2 << endl;

    cout << (mb1 = mb2) << endl;
  }
  //aqui apanhe todas as excepções na ordem correcta e
  //visualize no écran toda a informação acerca destas.

  return 0;
}
```

B1.7. Tente retirar o comentário cada uma das linhas numeradas (uma a uma) e verifique como as excepções são geradas e processadas.