Advanced Programming
Topic 4: Classes and Objects

Nguyen, Thien Binh
Mechatronics and Sensor Systems Technology
Vietnamese-German University

Intake 2024

VGU
Vietnamese - German University

Outline

Outline

An Introductory Example

- Programs like the one we did in Coding 2 are called *structured programming* because they are a collection of functions in which each function performs a particular task.
- Now suppose that we want to define our own types called `classes` which contain not only their `data/properties` but also all associated `functions/methods`.
  For example, `class Matrix` in which each object of this class represents a matrix having its numbers or rows and columns, and its entry values as its properties, and all matrix operations as its member methods. Similarly, we could define `class Vector` to represent vectors.

An Introductory Example I

- Using our defined classes, Listing 9 can be rewritten as follows

```
1  #include <iostream>
2  #include "vector.h"
3  #include "matrix.h"
4  using namespace std;
5
6  int main()
7  {
8    double alpha(2.0);
9    Vector v(5), w(5), t(5);
10   Matrix A(5,5), B(5,5), C(5,5);
11
12   // initialize vectors and matrices
13   v.rand(); w.rand();
14   t.zeros();
15   A.rand(); B.rand();
16   C.zeros();
17
18   // set entries
19   v[3] = 10.0; w[5] = 1.0;
20   A(2,2) = 5.0;
```

An Introductory Example II

```
21
22    // print out the initialized vectors and matrices
23    v.print();
24    A.print();
25
26    // operations
27    t = v + w
28    t = alpha * v;
29
30    C = A + B;
31    w = A*v;
32
33    return 0;
34 }
```

Listing 1: Coding2Class.cpp

An Introductory Example

- `Coding2Class.cpp` is called an Object-Oriented Programming (OOP) code since it is based on objects not functions, e.g., `v`, `w`, or `A`, which are realizations of user-defined classes `Vector` or `Matrix`, respectively.
- Notice that:
  - Functions like `rand()`, `zeros()`, or `print()` are associated with classes `Vector` and `Matrix`. They are called `member methods` of these classes.
  - Variables like `size`, `numRows`, `numCols`, or vector/matrix `entries` are called class `properties/attributes/data members`. Data members are often hidden away, which in turn help increase the code readability.
  - Access to these class properties must be through their member methods which are the class `interfaces` used to interact with the outside program.

Recall: Key Concepts of OOP

- Key concepts of OOP:
  - Encapsulation: a mechanism that binds together the class data/attributes and member methods that manipulate the data into user-defined classes, and keeps these implementations hidden away from users in order to prevent data misuse or interception.
  - Abstraction: users work with objects at their abstract level, i.e., their important characteristics represented through their interfaces, and don't need to worry about the underlying details of how the objects are functioning.
  - Inheritance: represents an *"is-a"* relationship of objects. Inheritance allows the definition of derived classes which inherit the properties and functionality of a base class, i.e., a more general class.
  - Polymorphism: the ability to process objects differently depending on their data types or classes.

Recall: Benefits of OOP

- Benefits from OOP:
  - Modularity: thanks to encapsulation, all defined objects in a program are self-contained and interact with each other through their interfaces only. This makes code maintenance and debugging more feasible since errors could be narrowed down and traced back easier. Furthermore, modularity enables code fixing or upgrading to be done simultaneously since all classes are independent from one another. It also makes possible code packaging to provide users pre-compiled shared libraries.
  - Code re-usability: thanks to inheritance and templates, the same properties and methods in derived classes could be directly inherited from a base class. This not only reduces code redundancy (the same piece of code copied and pasted to many different functions/procedures), but also relieves code fixing and modifications, as well as speeds up implementation time (whatever modified in a base class automatically takes effects in it derived classes)
  - Flexibility: thanks to polymorphism, `virtual` methods of derived classes are able to *override* those of the base class. This reduces unnecessary complexity and enhances code readability.

VGU Vietnamese - German University

Outline

Declaration

- Class declaration is usually stored in a header file, e.g., `dmatrix.h`, which is included in any source file where it is used.
- A class is declared with keyword `class` followed by the class name.
- All the data members and member methods of the class are declared inside the class declaration. Non-member methods are declared outside the class.
- Each class declaration has special methods called class `constructors` and a `destructor`.

Declaration

- Note that a class declaration must be ended with a semi-colon (;). Objects of the class can be directly declared after the bracket and before the semi-colon (not recommended.)

```
1  #ifndef _MATRIXDOUBLECLASS_
2  #define _MATRIXDOUBLECLASS_
3
4  // for matrices with entries of type double
5  class MatrixDouble
6  {
7  private:
8    // data members are declared here
9
10 public:
11   // member methods are declared here
12 };
13
14 #endif
```

Listing 2: dmatrix.h

Class Declaration

- Question: What are the data members and member methods of class `Matrix` in which each object is an $m \times n$ matrix, e.g.,?

$$A = \begin{bmatrix} 2 & 8 & 5 & 7 & 0 \\ 5 & 9 & 9 & 3 & 5 \\ 6 & 6 & 2 & 8 & 2 \\ 2 & 6 & 3 & 8 & 7 \\ 2 & 5 & 3 & 4 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} 3 & 2 & 7 & 9 & 6 \\ 8 & 7 & 2 & 9 & 10 \\ 3 & 8 & 10 & 6 & 5 \\ 4 & 2 & 3 & 4 & 4 \\ 5 & 2 & 2 & 4 & 9 \end{bmatrix} \tag{1}$$

  - Entries $A(i, j)$, number of rows, number of columns, arithmetic operators $A + B, A - B, A * B$, etc. What else?

Declaration

- Question: What are the data members and member methods of class `Matrix`?
  - Data members:
    1. `numRows`: number of rows
    2. `numCols`: number of columns
    3. `entries`: entry values $A(i,j)$
  - Member methods:
    1. Memory allocation, de-allocation for storing objects
    2. Copy method: to copy a matrix from another existing one
    3. Matrix initialization: `zeros()` (set all entries to zero), `ones()` (set all entries to one), `random()` (set each entry to a random value)
    4. Class data access: `setEntries(i,j)`, `getEntries(i,j)`, `getRows()`, `getCols()`
    5. Entry access operators: `A(i,j)`
    6. Output methods: `print()` (print all entries to the console)
    7. Assignment operator: $A = B$
    8. Unary operators: $+A$, -A, $++A$, $A++$, $--A$, $A--$
    9. Binary operators: $A+ = B$, $A- = B$, $A* = \alpha$, $A* = \mathbf{v}$, $A* = B$, $A + B$, $A - B$, αA, $A\mathbf{v}$, AB, A^{α}

VGU Vietnamese - German University

Encapsulation

- Encapsulation: to combine into a class both its data members and member methods
- Data hiding: the data members of a class should be hidden away from class users, and the class only communicates with the outside program through its interfaces which are the member methods
- Data hiding not only prevents class data from unwanted modification, but also conceives all unnecessary information from users so that they approach the use of classes at their abstraction level. This is one of the key ideas of OOP.

Encapsulation

- Encapsulation is achieved by granting access privileges to the data and methods of a class using access specifiers:
  - `private:` no communications with the outside program, can only be accessed from other class data and methods, as well as `friend` class or methods
  - `protected:` similar to `private`, except that methods from *derived* classes can have their access to protected data/methods
  - `public:` the class interfaces to interact with the outside program, can be accessed from outside functions
- If an access specifier is omitted, by default that part is private.
- Access specifiers can be placed anywhere and used more than once in a class declaration.

Encapsulation

- In class `Matrix`, which parts should be `private`, and which should be made `public`?

Declaration

- Question: Which parts should be `private`, and which should be made `public`?
  - Data members: (`private`, no access from the outside)
    1. `numRows`: number of rows
    2. `numCols`: number of columns
    3. `entries`: entry values $A(i,j)$
    4. `type`: matrix type
  - Member methods: (`public`, class interfaces)
    1. Memory allocation, de-allocation for storing objects
    2. Copy method: to copy a matrix from another existing one
    3. Matrix initialization: `zeros()` (set all entries to zero), `ones()` (set all entries to one), `random()` (set each entry to a random value)
    4. Class data access: `setEntries(i,j,value)`, `getEntries(i,j)`, `getRows()`, `getCols()`
    5. Entry access operators: `A(i,j)`
    6. Output methods: `print()` (print all entries to the console)
    7. Assignment operator: $A = B$
    8. Unary operators: $+A$, -A, $++A$, $A++$, $--A$, $A--$
    9. Binary operators: $A+ = B$, $A- = B$, $A* = \alpha$, $A* = \mathbf{v}$, $A* = B$, $A + B$, $A - B$, αA, $A\mathbf{v}$, AB, A^{α}

VGU Vietnamese - German University

Encapsulation

```
1  #ifndef _MATRIXDOUBLECLASS_
2  #define _MATRIXDOUBLECLASS_
3  class MatrixDouble
4  {
5  private:
6     // data members are declared here
7     double **entries;
8     int numRows;
9     int numCols;
10    int type;
11
12 public:
13    // all member methods are declared here
14
15 };
16 #endif
```

Listing 3: dmatrix.h

VGU Vietnamese - German University

Method Definitions

- It is common that all methods of a declared class are defined in a separate source file having the same name as that of the header file, e.g., `dmatrix.cpp`, in which the header file is included.
- When defining a class member method, its name must be prefixed by the name of the class followed by a double colon (`::`), e.g., `void Matrix::print() const`
- The definition of non-member methods need not a class name prefixed.

Outline

Constructors

- Question: What happens when we define an object?

```
1    Matrix A(5,5), B(5,5), C(5,5);
```

  $\Rightarrow$ *An appropriate constructor method defined in the class will be called and executed.*

- A `constructor`:
  - has exactly the same name as of the class
  - does not have a return type, even `void`
  - is called right after an object is created before any other methods
  - is used for memory allocation and object initialization
- A class can have multiple overloading constructors.
- `Matrix(const Matrix& mat_)` is called a `copy constructor` which copies a matrix from an existing one.
- If there is not any constructor declared in a class, a default constructor without input arguments will be automatically created.

VGU Vietnamese - German University

Constructors I

- Constructor declarations:

```
1  #ifndef _MATRIXDOUBLECLASS_
2  #define _MATRIXDOUBLECLASS_
3
4  class MatrixDouble
5  {
6  private:
7    // data members are declared here
8    double **entries;
9    int numRows;
10   int numCols;
11   int type;
12
13 public:
14   // memory allocation utilities
15   void allocate();
16   void deallocate();
17
```

Vietnamese - German University

Constructors II

```
18    // constructors
19    MatrixDouble();
20    Matrix(const int& numRows_, const int& numCols_);
21    Matrix(const int& numRows_, const int& numCols_,
22           double const& val_);
23    Matrix(const Matrix& mat_);
24
25 };
26
27 #endif
```

Listing 4: dmatrix.h

Constructors I

- Constructor definitions:

```
1  #include <iostream>
2  #include <cassert>              // for assert
3  #include "constants.h"           // user-defined constants
4  #include "matrix.h"
5  using namespace std;
6
7  //memory allocation
8  void MatrixDouble::allocate()
9  {
10   entries = new double* [numRows];
11   for (int i = 0; i < numRows; ++i)
12     entries[i] = new double [numCols];
13 }
14
15 void Matrix::deallocate()
16 {
17   for (int i = 0; i < numRows; ++i)
```

Constructors II

```
18      delete[] entries[i];
19    delete[] entries;
20    entries = NULL;
21  }
22
23  //constructors
24  Matrix::Matrix()
25  {
26    entries = NULL;
27    numRows = 0;
28    numCols = 0;
29    type = DMAT;
30  }
31
32  Matrix::Matrix(const int& numRows_, const int& numCols_)
33  {
34    assert (numRows_ > 0 && numCols_ > 0);
35    numRows = numRows_;
36    numCols = numCols_;
```

Vietnamese - German University

Constructors III

```
37    type = DMAT;
38    allocate();
39  }
40
41  // set all entries to val_
42  Matrix::Matrix(const int& numRows_, const int& numCols_,
43                 const double& val_)
44  {
45    assert (numRows_ > 0 && numCols_ > 0);
46    numRows = numRows_;
47    numCols = numCols_;
48    type = DMAT;
49    allocate();
50
51    for (int i = 0; i < numRows; ++i)
52      for (int j = 0; j < numCols; ++j)
53        entries[i][j] = val_;
54  }
55
```

Vietnamese - German University

Constructors IV

```
56 // copy constructor
57 Matrix::Matrix(const Matrix& mat_)
58 {
59   numRows = mat_.getRows();
60   numCols = mat_.getCols();
61   type = DMAT;
62   allocate();
63
64   for (int i = 0; i < numRows; ++i)
65     for (int j = 0; j < numCols; ++j)
66       entries[i][j] = mat_(i,j);
67 }
```

Listing 5: dmatrix.cpp

Constructors I

- Data initialization: member data can be directly initialized in constructor's initializer list when the constructor is defined.
- A constructor can call another constructor in its initializer list (from C++11).
- dmatrix.cpp can be improved to reduce the redundancy.

```
1  #include <iostream>
2  #include <cassert>              // for assert
3  #include "constants.h"            // user-defined constants
4  #include "matrix.h"
5  using namespace std;
6
7  //constructors
8  Matrix::Matrix()
9  {
10   entries = NULL;
11   numRows = 0;
12   numCols = 0;
```

Constructors II

```
13    type = DMAT;
14  }
15
16  Matrix::Matrix(const int& numRows_, const int& numCols_) :
17    numRows(numRows_), numCols(numCols_), type(DMAT)
18  {
19    assert (numRows > 0 && numCols > 0);
20    allocate();
21  }
22
23  // set all entries to val_
24  Matrix::Matrix(const int& numRows_, const int& numCols_,
25                 const double& val_) :
26    Matrix(numRows_, numCols_)
27  {
28    for (int i = 0; i < numRows; ++i)
29      for (int j = 0; j < numCols; ++j)
30        entries[i][j] = val_;
31  }
```

Constructors III

```
32
33 // copy constructor
34 Matrix::Matrix(const Matrix& mat_) :
35   Matrix(mat_.getRows(), mat_.getCols())
36 {
37   for (int i = 0; i < numRows; ++i)
38     for (int j = 0; j < numCols; ++j)
39       entries[i][j] = mat_(i,j);
40 }
```

Listing 6: dmatrix.cpp

Destructor

- There is only ONE destructor per class.
- A destructor does not receive any input arguments.
- Syntax: `~Matrix()`
- A destructor must be declared in the `public` part
- A destructor is particularly important to manually de-allocate the memories allocated with command `new` in the constructors. Otherwise, there will be memory leaks.

```
1  Matrix::~Matrix()
2  {
3     deallocate();
4  }
```

VGU Vietnamese - German University

Outline

Initialization I

- Declaration:

```
1  public:
2    // zero matrix
3    void zeros() const;
4    // one matrix
5    void ones() const;
6    // identity matrix
7    void eye() const;
8    // random matrix with
9    // values range from lower_ to upper_, Gaussian law
10   void random(const int& lower_,
11               const int& upper_) const;
```

Listing 7: dmatrix.h

- Definition:

Initialization II

```
1  void Matrix::zeros() const
2  {
3    for (int i = 0; i < numRows; ++i)
4      for (int j = 0; j < numCols; ++j)
5        entries[i][j] = ZERO;
6  }
7
8  void Matrix::ones() const
9  {
10   for (int i = 0; i < numRows; ++i)
11     for (int j = 0; j < numCols; ++j)
12       entries[i][j] = ONE;
13 }
14
15 void Matrix::eye() const
16 {
17   for (int i = 0; i < numRows; ++i)
18     for (int j = 0; j < numCols; ++j)
19       entries[i][j] = ZERO;
20
21   for (int i = 0; i < numRows; ++i)
```

Vietnamese - German University

Initialization III

```
22      entries[i][i] = ONE;
23  }
24
25  void Matrix::random(const int& lower_, const int& upper_) const        /
26  {
27    assert(lower_ <= upper_);
28    for (int i = 0; i < numRows; ++i)
29      for (int j = 0; j < numCols; ++j)
30        entries[i][j] = (double)( lower_ + rand() % upper_ );
31  }
```

Listing 8: dmatrix.cpp

`const` Methods

- A member method with keyword `const` followed at the end is called a constant method.
- A `const` member method does not allow modification of the object on which they are called $\Rightarrow$ it cannot return by reference or pointer, if that reference or pointer is not `const`
- A `const` member method can be called on any type of objects, const and non-const. A non-const member method cannot accept a `const` object as its argument.

Outline

Operator Overloading

- Operator methods: are needed to
  - set and get the value of class attributes, e.g., `A(i,j) = val`, `A[i][j] = val`, α = `A(i,j)`
  - assign objects, e.g., `A = B`, `A = -B` ,where `A`, `B` are objects of class `MatrixDouble`
  - carry out arithmetic operations, e.g., `A+B`, `++A`, `AB`, `Av`, where **v** is a vector of class

Operator Overloading

- An operator method can be declared with keyword `operator` followed by the symbol of the operator and list of the arguments, e.g., `operator+(const Matrix& A, const Matrix& B)`
- An operator method requires a return type and input argument list, just like ordinary functions.
- The number of arguments depends on the operator and cannot be changed in redefinitions.
- A number can be defined as
  - a member method
  - a non-member method
- Operators can be overloaded.

Entry Access Operators

- Operators [] and () are used to set and get entry values through the expression A[i][j] = 10, A(i,j) = 10, x = A(i,j)
- They can be overloaded multiple times.

Entry Access Operators I

- Declaration:

```
1 public:
2    // for non-const objects
3    // allow both to set and to get entry values
4    Matrix& operator() (const int& i, const int& j);
5    // for const objects
6    // allow only to get entry values
7    Matrix operator() (const int& i, const int& j) const;
```

Listing 9: dmatrix.h

- Definition:

Entry Access Operators II

```
1  double& Matrix::operator() (const int& i, const int& j)
2  {
3    assert(i>-1 && i < numRows);
4    assert(j > -1 && j < numCols);
5    return entries[i][j];
6  }
7
8  double  Matrix::operator() (const int& i, const int& j) const
9  {
10   assert(i>-1 && i < numRows);
11   assert(j > -1 && j < numCols);
12   return entries[i][j];
13 }
```

Listing 10: dmatrix.cpp

Unary Operators

- An unary operator has only one operand. Examples include `+A`, `-A`, `++A`, `A++`, `--A`, `A--`, `!A`
- Unary operators can be declared either as member methods (with no arguments, e.g., `Matrix& Matrix::operator++()`) or non-member methods (with one argument, e.g, `Matrix& operator++(Matrix A)`)

Unary Operators I

- Declaration: as member methods (without input arguments)

```
1  public:
2     // A = -B, A = -A OK
3     Matrix  operator- () const;
4
5     // prefix ++A
6     Matrix& operator++ ();
7
8     // postfix A++
9     // int here is just a dummy argument
10    // used to distinguish with ++A
11    Matrix  operator++ (int);
```

Listing 11: dmatrix.h

- Definition:

Unary Operators II

```
1  // A = -B
2  // cannot return by reference here
3  // since, e.g., A = -B, B is changed to -B
4  Matrix Matrix::operator- () const
5  {
6    Matrix T0(numRows, numCols);
7    T0.zeros();
8    for (int i = 0; i < numRows; ++i)
9      for (int j = 0; j < numCols; ++j)
10       T0(i,j) = -entries[i][j];
11   return T0;
12 }
13
14 // prefix ++A
15 Matrix& Matrix::operator-- ()
16 {
17   for (int i = 0; i < numRows; ++i)
18     for (int j = 0; j < numCols; ++j)
```

Vietnamese - German University

Unary Operators III

```
19          ++entries[i][j];
20      return *this;
21  }
22
23  // postfix A++
24  Matrix Matrix::operator++ (int)
25  {
26      Matrix T0(numRows, numCols);
27      T0.zeros();
28      for (int i = 0; i < numRows; ++i)
29          for (int j = 0; j < numCols; ++j)
30              T0(i,j) = ++entries[i][j];
31      return T0;
32  }
```

Listing 12: dmatrix.cpp

Self-Reference with `this` Pointer

- The object which is called by a member method can be self-referred as a pointer `this`
- Self-reference is needed
  - when the object is modified through an operator
  - grant access to the member data of the base class when templates are used

Unary Operators I

- Declaration: as non-member methods (with one input argument)

```
1  class Matrix
2  {
3  private:
4     // data members
5  public:
6     // member method
7
8     // friend operator methods
9     friend Matrix& operator++ (const Matrix& mat_);
10 };
11 // non-member operator methods
12 Matrix& operator++ (const Matrix& mat_);
```

Listing 13: dmatrix.h

- Definition:

Unary Operators II

```
1 Matrix& operator++ (const Matrix& mat_)
2 {
3   for (int i = 0; i < numRows; ++i)
4     for(int j = 0; j < numCols; ++j)
5       entries[i][j] += mat_(i,j);
6 }
```

Listing 14: dmatrix.cpp

- If a function is declared as a `friend` method of a class, it can get access to the `private` or `protected` member data.
- Friendship can also be applied to classes.
- Note that `class A` is a `friend` of `class B` does not mean that `class B` is a `friend` of `class A`.

Binary Operators

- A binary operator involves two operands. Examples include
  - Assignment operators: `A = B`
  - Relational operators: `A > B`, `A < B`, `A == A`, `A != B`, `A >= B`, `A <= B`
  - Shortcut operators: `A += B`, `A -= B`, `A *= B`, `A /=` α
  - Arithmetic operators: `A + B`, `A - B`, `A * B`, `A /` α, A^{α}
  - Input - Output operators: `>>`, `<<`
  - etc.
- A binary operator can be declared as either a member operator (with one argument) or non-member operator (with two arguments)
- Operators which modify an argument can only be declared as member methods, e.g., `+=`, `-=`, `*=`, `/=`

Binary Operators

- Assignment operator: declaration and definition

```
1 public:
2   Matrix& operator= (const Matrix& mat_);
```

Listing 15: dmatrix.h

```
1 Matrix& Matrix::operator= (const Matrix mat_)
2 {
3   assert(numRows == mat_.getRows()
4       && numCols == mat_.getCols());
5   for (int i = 0; i < numRows; ++i)
6     for (int j = 0; j < numCols; ++j)
7       entries[i][j] = mat_(i,j);
8   return *this;
9 }
```

Listing 16: dmatrix.cpp

VGU Vietnamese - German University

Binary Operators

- Shortcut operators: (must be member methods)

```
1 public:
2   Matrix& operator+= (const Matrix& mat_);
3   Matrix& operator-= (const Matrix& mat_);
4   Matrix& operator*= (const double& alp_);
5   Matrix& operator*= (const Matrix& mat_);
```

Listing 17: dmatrix.h

Binary Operators I

- Arithmetic operators: member methods (take one input arguments)

```
1 public:
2   Matrix   operator+ (const Matrix& mat_);
3   Matrix   operator- (const Matrix& mat_);
4   Matrix   operator* (const double& alp_);
5   Matrix   operator* (const Matrix& mat_);
```

Listing 18: dmatrix.h

- Arithmetic operators: non-member methods (take two input arguments)

VGU Vietnamese - German University

Binary Operators II

```
1  class Matrix
2  {
3  private:
4    // data members
5  public:
6    // member methods
7    // friend binary operators
8    friend Matrix operator+(const Matrix& mat1_,const Matrix& mat2_);
9    friend Matrix operator-(const Matrix& mat1_,const Matrix& mat2_);
10   friend Matrix operator*(const double& alp_,const Matrix& mat_);
11   friend Matrix operator*(const Matrix& mat1_,const Matrix& mat2_);
12 };
13 // non-member binary operators
14 Matrix operator+ (const Matrix& mat1_, const Matrix& mat2_);
15 Matrix operator- (const Matrix& mat1_, const Matrix& mat2_);
16 Matrix operator* (const double& alp_, const Matrix& mat_);
17 Matrix operator* (const Matrix& mat1_, const Matrix& mat2_);
```

Listing 19: dmatrix.h

VGU Vietnamese - German University

Binary Operators

- Shortcut operators: (must be member methods)

```
1 public:
2 Matrix& operator+= (const Matrix& mat_);
3 Matrix& operator-= (const Matrix& mat_);
4 Matrix& operator*= (const double& alp_);
5 Matrix& operator*= (const Matrix& mat_);
```

Listing 20: dmatrix.h

Notes for Operator Overloading

- Return by reference for operators that modify the objects
- Return by value for operators that do not modify the objects
- Not any operators can be declared as non-member methods. These include the assignment (`A = B`), entry access (`A(i,j)`, `A[i][j]`), member selection (e.g., `A->getEntry(i,j)`), shortcut operators.
- Not any operators can be declared as member methods, e.g, `<<`

Reading

1. Capper, *Introducing C++ for Scientists, Engineers, and Mathematicians*, **Chapters 8 - 10**
2. Pitt-Francis, and Whiteley, *Guide to Scientific Computing in C++*, **Chapter 6**