Introduction to C++ for Financial Engineers

An object-oriented approach

Daniel J. Duffy

November 21, 2019

Contents

Listings

List of Algorithms

0. Goals of this Book and Global Overview

0.1. What is this book?

0.2. Why has this book been written?

0.3. For whom is this book intended?

0.4. Why should I read this book?

0.5. The structure of this book

0.6. What this book does not cover

0.7. More information and support

Listing 1: Array class

```
// Array.cpp
//
// Array class. It uses an ArrayStructure for the actual storage.
// This class acts like a kind of adapter since it defines a common interface
// for different array structures like normal arrays and sparse arrays.
// The array structure to use is given as template argument.
//
// 29 januari 1999      RD      Started
// 2002-1-21 DD indexing starts at 1
// 2002-3-30 DD operator [] incorrectly implemented; corrected
// 2005-12-17 DD size_t ⟶ I
//
// (C) Datasim Component Technology 1999-2006

#ifndef DSArray_cpp
#define DSArray_cpp
```

```
#include "array.hpp"
#include <stddef.h>

// Constructors & destructor
template <class V, class I, class S>
Array<V, I, S>::Array()
{ // Default constructor

        m_structure=S();
        m_start=1;
}

template <class V, class I, class S>
Array<V, I, S>::Array(I size)
{ // Constructor with size. Start index=1.

        m_structure=S(size_t(size));
        m_start=1;
}

template <class V, class I, class S>
Array<V, I, S>::Array(I size, I start)
{ // Constructor with size & start index

        m_structure=S(size_t(size));
        m_start=start;
}

template <class V, class I, class S>
Array<V, I, S>::Array(I size, I start, const V& value)
{ // Constructor with size & start index

        m_structure=S(size_t(size));
        m_start=start;

        // Initialise array elements
        for (I i = MinIndex(); i <= MaxIndex(); i++) (*this)[i] = value;
}


template <class V, class I, class S>
Array<V, I, S>::Array(const Array<V, I, S>& source)
{ // Copy constructor
```

```
        m_structure=source.m_structure;
        m_start=source.m_start;
}

template <class V, class I, class S>
Array<V, I, S>::~Array()
{ // Destructor
}

// Selectors
template <class V, class I, class S>
I Array<V, I, S>::MinIndex() const
{ // Return the minimum index

        return m_start;
}

template <class V, class I, class S>
I Array<V, I, S>::MaxIndex() const
{ // Return the maximum index

        return m_start+Size()-1;
}

template <class V, class I, class S>
I Array<V, I, S>::Size() const
{ // The size of the array

        return I(m_structure.Size());
}

// Operators
template <class V, class I, class S>
inline V& Array<V, I, S>::operator [] (I index)
{ // Subscripting operator

        return m_structure[index - m_start + 1];
}

template <class V, class I, class S>
inline const V& Array<V, I, S>::operator [] (I index) const
```

```
{ // Subscripting operator

        return m_structure[index - m_start + 1];
}

template <class V, class I, class S>
Array<V, I, S>& Array<V, I, S>::operator = (const Array<V, I, S>& source)
{ // Assignment operator

        // Exit if same object
        if (this==&source) return *this;

        m_structure=source.m_structure;
        m_start=source.m_start;

        return *this;
}


#endif  // DSArray_cpp
```

Listing 2: CMakeLists

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
project(finance)
add_executable(finance
        "director.cpp"
        "array.cpp"
        "lattice.cpp"
        "arraymechanisms.cpp"
        "arraystructure.cpp"
        "binomiallatticestrategy.cpp"
        "binomialmethod.cpp"
        "europeanoptionfactory.hpp"
        "fullarray.cpp"
        "fullmatrix.cpp"
        "latticemechanisms.cpp"
        "matrix.cpp"
        "matrixmechanisms.cpp"
        "matrixstructure.cpp"
        "numericmatrix.cpp"
        "option.hpp"
        "property.cpp"
        "propertything.cpp"
        "range.cpp"
        "set.cpp"
        "simplepropertyset.cpp"
        "tensor.cpp"
        "vector.cpp"
)
```

Algorithm 1: IntervalRestriction

```
Data: $G = (X, U)$ such that $G^{tc}$ is an order.
Result: $G' = (X, V)$ with $V \subseteq U$ such that $G'^{tc}$ is aninterval order.
begin
    $V \longleftarrow U$
    $S \longleftarrow \emptyset$
    for $x \in X$ do
        NbSuccInS$(x) \longleftarrow 0$
        NbPredInMin$(x) \longleftarrow 0$
        NbPredNotInMin$(x) \longleftarrow |$ImPred$(x)|$
    for $x \in X$ do
        if NbPredInMin$(x) = 0$ and NbPredNotInMin$(x) = 0$ then
            AppendToMin$(x)$
1   while $S \neq \emptyset$ do
REM     remove $x$ from the list of $T$ of maximal index
2       while $|S \cap$ ImSucc$(x)| \neq |S|$ do
            for $y \in S - ImSucc(x)$ do
                { remove from $V$ all the arcs $zy$ : }
                for $z \in$ ImPred$(y) \cap$ Min do
                    remove the arc $zy$ from $V$
                    NbSuccInS$(z) \longleftarrow$ NbSuccInS$(z) - 1$
                    move $z$ in $T$ to the list preceding its present list
                    {i.e. If $z \in T[k]$, move $z$ from $T[k]$ to $T[k-1]$}
                NbPredInMin$(y) \longleftarrow 0$
                NbPredNotInMin$(y) \longleftarrow 0$
                $S \longleftarrow S - \{y\}$
                AppendToMin$(y)$
        RemoveFromMin$(x)$
```

Part I.

C++ Essential Skills

1. Introduction to C++ and Quantitative Finance

1.1. Introduction and objectives

1.2. A short history of C++

1.3. C++, a multi-paradigm language

1.3.1. Object-oriented paradigm

```
double PutPrince()
{
        double tmp = sig * sqrt(T);
        double d2 = d1 - tmp;
        return (K * exp(-r * T)* N(-d2)) - (U * exp((b - r)*T)* N(-d1));
}
```

1.3.2. Generic programming

```
template <ckass Numeric>
        Numeric Max(const Numeric& x, const Numeric& y)
{
        if (x > y)
                return x;
        return y;
}

long dA = 12334; long dB = 2;
std::cout << "\n\nMax and min of two numbers: " << std::endl;
std::cout << "Max value is: " << Max<long>(dA, dB) << std::endl;
```

1.3.3. Procedural, modular and functional programming

1.4. C++ and quantitative finance: what's the relationship?

1.5. What is software quality?

1.6. Summary and conclusions

1.7. Exercises

2. The Mechanics of C++: from Source Code to a Running Program

2.1. Introduction and objectives

2.2. The compilation process

2.3. Header files and source files

Listing 2.1: Header files containing declarations of functions

```
// Inequalities.hpp
//
// Header file containing declarations of functions
//
// (C) Datasim Education BV 2006
//

// Preprocessor directives; ensures that we do not
// include a file twice (gives compiler error)
#ifndef Inequalities_HPP
#define Inequalities_HPP

/////////// Useful functions //////////////////
```

```
// Max and Min of two numbers
double Max(double x, double y);
double Min(double x, double y);

// Max and Min of three numbers
double Max(double x, double y, double z);
double Min(double x, double y, double z);

////////////////////////////////////////////////

#endif
```

Listing 2.2: Code files containing bodies of functions

```
// Inequalities.cpp
//
// Code file containing bodies of functions
//
// Last Modification Dates:
//
//      2006-2-17 DD kick-off code
//
// (C) Datasim Education BV 2006
//

#include "Inequalities.hpp"

////////// Useful functions //////////////////

// Max and Min of two numbers
double Max(double x, double y)
{
        if (x > y)
                return x;

        return y;
}

double Min(double x, double y)
{
        if (x < y)
                return x;
        return y;
}

// Max and Min of three numbers
double Max(double x, double y, double z)
{
        return Max(Max(x,y), z);
}

double Min(double x, double y, double z)
{
        return Min(Min(x,y), z);
}
```

Listing 2.3: Math program (Console-based) to test Max and Min functions.

```
// TestInequalities.cpp
//
// Main program (Console-based) to test Max and
// Min functions.
//
// (C) Datasim Education BV 2006
//

#include <iostream>             // Console input and output

#include "Inequalities.hpp"

int main()
{
        // Prompt the user for input. Console output (cout)
        // and input (cin)
        double d1, d2;
        std::cout << "Give the first number: ";
        std::cin >> d1;
        std::cout << "Give the second number: ";
        std::cin >> d2;

        char c; // Character type
        std::cout << "Which function a) Max() or b) Min()? ";
        std::cin >> c;
        if (c == 'a')
        {
                std::cout << "Max value is: " << Max(d1, d2) <<
std::endl;
        }
        else
        {
                std::cout << "Min value is: " << Min(d1, d2) <<
std::endl;
        }

        double dA = 1.0; double dB = 2.0; double dC = 3.0;
        std::cout << "\n\nMax and min of three numbers: " << std::endl;
        std::cout << "Max value is: " << Max(dA, dB, dC) << std::endl;
        std::cout << "Min value is: " << Min(dA, dB, dC) << std::endl;

        return 0;
}
```

Listing 2.4: Output of `TestInequalities.cpp`.

```
Give the first number: 10
Give the second number: 20
Which function a) Max() or b) Min()? b
Min value is: 10


Max and min of three numbers:
Max value is: 3
Min value is: 1
```

2.4. Creating classes and using their objects

```
#include <string>       // Standard string class in C++
using namespace std;

#include "DatasimDate.hpp"      // Dates and other useful stuff
#include "Person.hpp"   // Interface functions for Person
```

Listing 2.5: Hello World class.

```
// Person.hpp
//
// 'Hello World' class. Function declarations.
//
// (C) Datasim Education BV 2005-2006
//
```

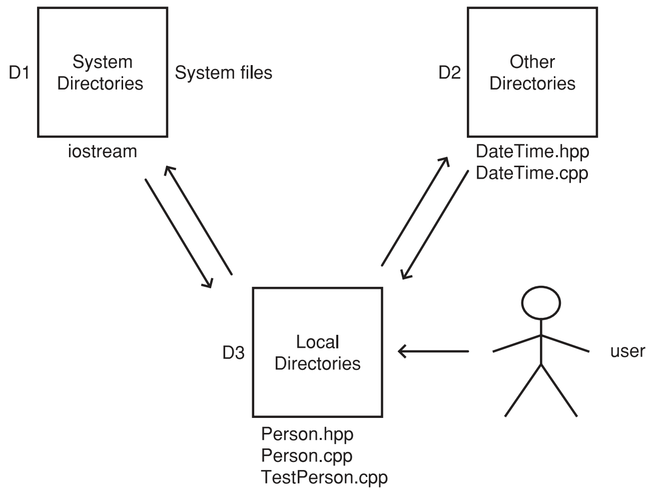

Figure 2.1.: Directory structure for project.

```
#ifndef Person_HPP
#define Person_HPP

#include "DatasimDate.hpp"      // My dates and other useful stuff
#include <string>                       // Standard string class in C++

class Person
{
public: // Everything public, for convenience only

                // Data
                std::string nam;                                        // Name o
                DatasimDate dob;                        // Date of birth

                DatasimDate createdD;           // Internal, when object was crea

public:
                Person (const std::string& name, const DatasimDate& DateofBirth);


                void print() const;

                int age() const;
};

#endif
```

Listing 2.6: Hello World class.

```
// Person.cpp
//
// 'Hello World' class
//
// Last Modification Dates
//
// 2006-2-17 DD Kick-off
//
// (C) Datasim Education BV 2005-2006
//


#include "Person.hpp"

Person::Person (const std::string& name, const DatasimDate& DateofBirth)
{
                        nam = name;
                        dob = DateofBirth;
                        createdD = DatasimDate();       // default, today REALLY!

}

void Person::print() const
{ // Who am I?

                        std::cout << "\n** Person Data **\n";

                        std::cout << "Name: " << nam << ", Date of birth: " <<
dob
                                                << ", Age: " <<
age() << std::endl;

}

int Person::age() const
{
                return int( double(DatasimDate() - dob) / 365.0);
}
```

Listing 2.7: Hello World Testing the first C++ class.

```
// TestPerson.cpp
//
// 'Hello World' Testing the first C++ class
//
// (C) Datasim Education BV 2005-2006
//


#include "DatasimDate.hpp"      // Dates and other useful stuff
#include "Person.hpp"           // Interface functions for Person
#include <string>                       // Standard string class in C++

int main()
{

        DatasimDate myBirthday(29, 8, 1952);
        std::string myName ("Daniel J. Duffy");
        Person dd(myName, myBirthday);
        dd.print();

        DatasimDate bBirthday(06, 8, 1994);
        std::string bName ("Brendan Duffy");
        Person bd(bName, bBirthday);
        bd.print();

        return 0;
}
```

Listing 2.8: Output of `TestPerson.cpp`.

```

** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 67

** Person Data **
Name: Brendan Duffy, Date of birth: 6/8/1994, Age: 25
```

2.5. Template classes and template functions

Listing 2.9: Header file containing declarations of functions.

```
// GenericInequalities.hpp
//
```

```
// Header file containing declarations of functions
//
// This is the template/generic version.
//
// (C) Datasim Education BV 2006
//

// Preprocessor directives; ensures that we do not
// include a file twice (gives compiler error)
#ifndef GenericInequalities_HPP
#define GenericInequalities_HPP

///////////// Useful functions //////////////////

// Max and Min of two numbers
template <class Numeric>
Numeric Max(const Numeric& x, const Numeric& y);
template <class Numeric>
Numeric Min(const Numeric& x, const Numeric& y);

// Max and Min of three numbers
template <class Numeric>
Numeric Max(const Numeric& x, const Numeric& y, const Numeric& z);
template <class Numeric>
Numeric Min(const Numeric& x, const Numeric& y, const Numeric& z);

#include "GenericInequalities.cpp"

#endif
```

Listing 2.10: Code file containing declarations of functions.

```
// Generic& valueInequalities.cpp
//
// Code file containing bodies of functions
//
// Last Modific& valueation Dates:
//
//      2006-2-17 DD kic& valuek-off code
//      2006-2-17 DD Copied from non-generic& value structure
//
// (C) Datasim Education BV 2006
//

#ifndef GenericInequalities_CPP
```

```
#define GenericInequalities_CPP

#include "GenericInequalities.hpp"

////////// Useful functions ///////////////////

// Max and Min of two numbers
template <class Numeric>
Numeric Max(const Numeric& x, const Numeric& y)
{
        if (x > y)
                return x;

        return y;
}

template <class Numeric>
Numeric Min(const Numeric& x, const Numeric& y)
{
        if (x < y)
                return x;

        return y;
}

// Max and Min of three numbers
template <class Numeric>
Numeric Max(const Numeric& x, const Numeric& y, const Numeric& z)
{
        return Max<Numeric>(Max<Numeric>(x,y), z);
}

template <class Numeric>
Numeric Min(const Numeric& x, const Numeric& y, const Numeric& z)
{
        return Min<Numeric>(Min<Numeric>(x,y), z);
}

#endif
```

Listing 2.11: Main program (Console-based) to test Max and Min functions.

```
// TestGenericInequalities.cpp
//
```

```
// Main program (Console-based) to test Max and
// Min functions.
//
// (C) Datasim Education BV 2006
//

#include <iostream>              // Console input and output

#include "GenericInequalities.hpp" // Needed because it is templated
int main()
{
        // Prompt the user for input. Console output (cout)
        // and input (cin)
        int d1, d2;
        std::cout << "Give the first number: ";
        std::cin >> d1;
        std::cout << "Give the second number: ";
        std::cin >> d2;

        char c; // Character type
        std::cout << "Which function a) Max() or b) Min()? ";
        std::cin >> c;
        if (c == 'a')
        {
                std::cout << "Max value is: " << Max<int>(d1, d2) <<
std::endl;
        }
        else
        {
                std::cout << "Min value is: " << Min<int>(d1, d2) <<
std::endl;
        }

        long dA = 12334; long dB = 2; long dC = -3;
        std::cout << "\n\nMax and min of three numbers: " << std::endl;
        std::cout << "Max value is: " << Max<long>(dA, dB, dC) <<
std::endl;
        std::cout << "Min value is: " << Min<long>(dA, dB, dC) <<
std::endl;

        return 0;
}
```

2.6. Kinds of errors

2.6.1. Compiler errors

2.6.2. Linker errors

```
int age(); // NO "const" in this declaration

int Person::age() const
{
        return int( double(DatasimDate() - dob) / 365.0);
}
```

Listing 2.12: Simple stuff for converting built-in types to strings.

```
// TestConversions.cpp
//
// Simple stuff for converting built-in
// types to strings.
//
// (C) Datasim Education BV 2006
//

#include <sstream>
#include <string>
#include <iostream>

template <typename T>
        std::string getString(const T& value)
{
                std::stringstream s;
                s << value;

                return s.str();
}

int main()
{

        // Hard-coded example for starters
        double myDouble = 1.0;
        std::stringstream s;
        s << myDouble;
        std::string result = s.str();
        std::cout << "String value is: " << result << std::endl;

        int i = 10;
        long j = 1234567890;
        float f = 3.14f;
        double d = 2.712222222223;

        std::string myString = getString<int>(i);
        std::cout << myString << std::endl;

        myString = getString<long>(j);
        std::cout << myString << std::endl;

        myString = getString<float>(f);
        std::cout << myString << std::endl;

        myString = getString<double>(d);
        std::cout << myString << std::endl;

        return 0;
}

#include <string>
#include <cstddef>

stdd::string getString(long j)
{
        char str[200];
        sprintf(str, "%d", j);
        std::string result(str);

        return result;
}

stdd::string getString(int j)
{
        char str[200];
        sprintf(str, "%d", j);
        std::string result(str);

        return result;
}
```

```
stdd::string getString(size_t j)
{
        char str[200];
        sprintf(str, "%d", j);
        std::string result(str);

        return result;
}
```

2.6.3. Run-time errors

2.7. The struct concept

2.8. Useful data conversion routines

```
// Hard-coded example for starters
std::stringstream s;
s << myDouble;
std::string result = s.str();
std::cout << "String value is: " << result << std::endl;

template <typename T>
        std::string getString(const T& value)
{
        stringstream s;
        s << value;
        return s.str();
}
```

2.9. Summary and conclusions

2.10. Exercises and projects

3. C++ Fundamentals and My First Option Class

3.1. Introduction and objectives

3.2. Class == member data + member functions

```
ExactEuropeanOption myobject ("P", "Index Option");

// ...

double d = myObject.Price();
```

3.3. The header file (function prototypes)

```
// EuropeanOption.hpp
class EuropeanOption
{
private:

        void init();    // Initialise all default values
        void copy(const EuropeanOption& o2);
        // "Kernel" functions for option calculations
        double CallPrice() const;
        double PutPrice() const;
        double CallDelta() const;
        double PutDelta() const;

public:
        // Public member data for convenience only
        double r;       // Interest rate
        double sig;     // Volatility
        double K;       // Strike price
        double T;       // Expiry date
        double U;       // Current underlying price
        double b;       // Cost of carry

        string optType; // Option name (call, put)

public:
```

```
// Constructors
EuropeanOption();       // Default call option
EuropeanOption(const EuropeanOption& option2);  // Copy constructor
EuropeanOption (const string& optionType);      // Create option type

// Destructor
virtual ~EuropeanOption();

// Assignment operator
EuropeanOption& operator = (const EuropeanOption& option2);

// Functions that calculate option price and (some) sensitivities
double Price() const;
double Delta() const;

// Modifier functions
void toggle();  // Change option type (C/P, P/C)
};
```

3.4. The class body (code file)

```
#include "EuropeanOption.hpp"   // Declaration of functions
#include <cmath>        // For mathematical functions, e.g. std::exp()

double EuropeanOption::PutPrice() const
{
        double tmp = sig * std::sqrt(T);

        double d1 = ( std::log(U/K) + (b + (sig*sig)*0.5) * T) / tmp;
        double d2 = d1 - tmp;

        return (K * exp(-r * T)* N(-d2))  - (U * std::exp((b - r)*T) * N(-d1));
}
```

Listing 3.1: European options.

```
// EurpeanOption.cpp
//
//      Author: Daniel Duffy
//
// (C) Datasim Component Technology BV 2003
//
```

```
#ifndef EuropeanOption_cpp
#define EuropeanOption_cpp


#include "EuropeanOption.hpp"

//////////// Gaussian functions /////////////////////////////////

double EuropeanOption::n(double x) const
{

        double A = 1.0/std::sqrt(2.0 * 3.1415);
        return A * std::exp(-x*x*0.5);

}

double EuropeanOption::N(double x) const
{ // The approximation to the cumulative normal distribution


        double a1 = 0.4361836;
        double a2 = -0.1201676;
        double a3 = 0.9372980;

        double k = 1.0/(1.0 + (0.33267 * x));

        if (x >= 0.0)
        {
                return 1.0 - n(x)* (a1*k + (a2*k*k) + (a3*k*k*k));
        }
        else
        {
                return 1.0 - N(-x);
        }

}


// Kernel Functions (Haug)
double EuropeanOption::CallPrice() const
{
        double tmp = sig * std::sqrt(T);

        double d1 = ( std::log(U/K) + (b+ (sig*sig)*0.5 ) * T )/ tmp;
        double d2 = d1 - tmp;


        return (U * std::exp((b-r)*T) * N(d1)) - (K * std::exp(-r * T)* N(d2));

}

double EuropeanOption::PutPrice() const
{

        double tmp = sig * std::sqrt(T);

        double d1 = ( std::log(U/K) + (b+ (sig*sig)*0.5 ) * T )/ tmp;
        double d2 = d1 - tmp;

        return (K * std::exp(-r * T)* N(-d2)) - (U * std::exp((b-r)*T) * N(-d1));

}

double EuropeanOption::CallDelta() const
{
        double tmp = sig * sqrt(T);

        double d1 = ( std::log(U/K) + (b+ (sig*sig)*0.5 ) * T )/ tmp;


        return std::exp((b-r)*T) * N(d1);
}

double EuropeanOption::PutDelta() const
{
        double tmp = sig * std::sqrt(T);

        double d1 = ( std::log(U/K) + (b+ (sig*sig)*0.5 ) * T )/ tmp;

        return std::exp((b-r)*T) * (N(d1) - 1.0);
}
```

```
/////////////////////////////////////////////////////////////////////////

void EuropeanOption::init()
{       // Initialise all default values

        // Default values
        r = 0.08;
        sig= 0.30;
        K = 65.0;
        T = 0.25;
        U = 60.0;                // U == stock in this case
        b = r;                   // Black and Scholes stock option model (1973)

        optType = "C";           // European Call Option (this is the default type



}

void EuropeanOption::copy(const EuropeanOption& o2)
{

        r       = o2.r;
        sig = o2.sig;
        K       = o2.K;
        T       = o2.T;
        U       = o2.U;
        b       = o2.b;

        optType = o2.optType;



}

EuropeanOption::EuropeanOption()
{ // Default call option

        init();
}

EuropeanOption::EuropeanOption(const EuropeanOption& o2)
```

```
{ // Copy constructor

        copy(o2);
}

EuropeanOption::EuropeanOption (const std::string& optionType)
{       // Create option type

        init();
        optType = optionType;

        if (optType == "c")
                optType = "C";

}



EuropeanOption::~EuropeanOption()
{

}


EuropeanOption& EuropeanOption::operator = (const EuropeanOption& option2)
{

        if (this == &option2) return *this;

        copy (option2);

        return *this;
}

// Functions that calculate option price and sensitivities
double EuropeanOption::Price() const
{

        if (optType == "C")
        {
                return CallPrice();
        }
        else
                return PutPrice();

}

double EuropeanOption::Delta() const
{
        if (optType == "C")
                return CallDelta();
        else
                return PutDelta();

}



// Modifier functions
void EuropeanOption::toggle()
{ // Change option type (C/P, P/C)

        if (optType == "C")
                optType = "P";
        else
                optType = "C";
}

#endif
```

3.5. Using the class

Listing 3.2: TestEuropeanOption options.

```
// TestEuropeanOption.cpp
//
// Test program for the exact solutions of European options.
// Check answers with Haug 1998
//
// (C) Datasim Component Technology BV 2003-2006
//

#include "EuropeanOption.hpp"
#include <iostream>

int main()
```

```
{ // All options are European

        // Call option on a stock
        EuropeanOption callOption;
        std::cout << "Call option on a stock: " << callOption.Price() <<
std::endl;

        // Put option on a stock index
        EuropeanOption indexOption;
        indexOption.optType = "P";
        indexOption.U = 100.0;
        indexOption.K = 95.0;
        indexOption.T = 0.5;
        indexOption.r = 0.10;
        indexOption.sig = 0.20;

        double q = 0.05;                // Dividend yield
        indexOption.b = indexOption.r - q;

        std::cout << "Put option on an index: " << indexOption.Price() <<
std::endl;

        // Call and put options on a future
        EuropeanOption futureOption;
        futureOption.optType = "P";
        futureOption.U = 19.0;
        futureOption.K = 19.0;
        futureOption.T = 0.75;
        futureOption.r = 0.10;
        futureOption.sig = 0.28;

        futureOption.b = 0.0;

        std::cout << "Put option on a future: " << futureOption.Price() <<
std::endl;

        // Now change over to a call on the option
        futureOption.toggle();
        std::cout << "Call option on a future: " << futureOption.Price() <<
std::endl;


        // Call option on currency
        EuropeanOption currencyOption;
        currencyOption.optType = "C";
        currencyOption.U = 1.56;
        currencyOption.K = 1.60;
        currencyOption.T = 0.5;
        currencyOption.r = 0.06;
        currencyOption.sig = 0.12;

        double rf = 0.08;               // risk-free rate of foreign currency
        currencyOption.b = currencyOption.r - rf;

        std::cout << std::endl << "** Other pricing examples **" <<
std::endl << std::endl;

        std::cout << "Call option on a currency: " << currencyOption.Price() <<
std::endl;

        ////////   NOW CALCULATIONS OF SENSITIVITIES //////////////////////////////

        // Call and put options on a future: Delta and Elasticity
        EuropeanOption futureOption2;
        futureOption2.optType = "P";
        futureOption2.U = 105.0;
        futureOption2.K = 100.0;
        futureOption2.T = 0.5;
        futureOption2.r = 0.10;
        futureOption2.sig = 0.36;

        futureOption2.b = 0.0;

        std::cout << "Delta on a put future: " << futureOption2.Delta() <<
std::endl;

        // Now change over to a call on the option
        futureOption2.toggle();
        std::cout << "Delta on a call future: " << futureOption2.Delta() <<
std::endl;


        // Stock Option: Gamma
        EuropeanOption stockOption;
        stockOption.optType = "C";
        stockOption.U = 55.0;
        stockOption.K = 60.0;
        stockOption.T = 0.75;
        stockOption.r = 0.10;
        stockOption.sig = 0.30;

        stockOption.b = stockOption.r;


        stockOption.toggle();

        // Calculating theta of a European stock index
        EuropeanOption indexOption2;
        indexOption2.optType = "P";
        indexOption2.U = 430.0;
        indexOption2.K = 405.0;
        indexOption2.T = 0.0833;        // One month expiration
        indexOption2.r = 0.07;
        indexOption2.sig = 0.20;

        double divYield = 0.05;         // Dividend yield, 5% per annum
        indexOption2.b = indexOption2.r - divYield;


        // Stock Option: Rho
        EuropeanOption stockOption2;
        stockOption2.optType = "C";
        stockOption2.U = 72.0;
        stockOption2.K = 75.0;
        stockOption2.T = 1.0;
        stockOption2.r = 0.09;
        stockOption2.sig = 0.19;

        stockOption2.b = stockOption2.r;

        // Calculating Cost of Carry of a European stock index
        EuropeanOption indexOption3;
        indexOption3.optType = "P";
        indexOption3.U = 500.0;
        indexOption3.K = 490.0;
        indexOption3.T = 0.222225;
        indexOption3.r = 0.08;
        indexOption3.sig = 0.15;

        double divYield3 = 0.05;                // Dividend yield, 5% per annum
        indexOption3.b = indexOption3.r - divYield3 ;
```

```
        return 0;
}
```

Listing 3.3: Output of `TestEuropeanOption.cpp`.

```
Call option on a stock: 2.13293
Put option on an index: 2.4648
Put option on a future: 1.70118
Call option on a future: 1.70118

** Other pricing examples **

Call option on a currency: 0.0290937
Delta on a put future: -0.356609
Delta on a call future: 0.59462
```

3.6. Examining the class in detail

3.6.1. Accessibility issues

3.6.2. Using standard libraries

3.6.3. The scope resolution operator “ :: ”

```
double EuropeanOption::Price() const
{
        if (optType == "C")
        {
                return CallPrice();
        }
        else
                return PutPrice();
}
```

3.6.4. Virtual destructor: better safe than sorry

Declare all destructors to be virtual.

3.7. Other paradigms

Listing 3.4: Simple functions for interest rate calculations.

```
// SimpleBondPricing.hpp
//
// Simple functions for interest rate calcuations.
```

```
//
// (C) Datasim Education BV 2006
//

#ifndef SimpleBondPricing_HPP
#define SimpleBondPricing_HPP

#include <vector>
#include <cmath>
#include <cassert>

namespace Chapter3CPPBook // Logical grouping of functions and others
{

        // Handy shorthand synonyms
        typedef std::vector<double> Vector;

        // Recursive function to calculate power of a number. This
        // function calls itself, either directly or indirectly
        double power(double d, long n);

        // Future value of a sum of money invested today
        double FutureValue(double P0, long nPeriods, double r);

        // Future value of a sum of money invested today, m periods
        // per year. r is annual interest rate
        double FutureValue(double P0, long nPeriods, double r, long m);

        // Continuous compounding, i.e. limit as m -> INFINITY
        double FutureValueContinuous(double P0, long nPeriods, double r);

        // Future value of an ordinary annuity
        double OrdinaryAnnuity(double A, long nPeriods, double r);

        // Present Value
        double PresentValue(double Pn, long nPeriods, double r);

        // Present Value of a series of future values
        double PresentValue(const Vector& prices, long nPeriods, double r);

        // Present Value of an ordinary annuity
        double PresentValueOrdinaryAnnuity(double A, long nPeriods, double r);
```

```
}

#endif

double fv2 = Chapter3CPPBook::FutureValue(P, nPeriods, r, freq);

using namespace Chapter3CPPBook;
std::cout << "**Future with " << m << " periods: " << FutureValue(P0, nPeriods, r
std::endl;

// Future value of a sum of money invested today
double FutureValue(double P0, long nPeriods, double r)
{
        double factor = 1.0 + r;
        return P0 * power(factor, nPeriods);
}

// Non-recursive function to calculate power of a number.
double power(double d, long n)
{
        if (n == 0) return 1.0;
        if (n == 1) return d;
        double result = d;
        for (long j = 1; j < n; j++)
        {
                result *= d;
        }
        return result;
}

// Handy shorthand synonyms
typdef vector<double> Vector;

// Present Value of a series of future values
double PresentValue(const Vector& Prices, long nPeriods, double r)
{
        // Number of periods MUST == size of the vector
        assert (nPeriods == prices.size());

        double factor = 1.0 + r;

        double PV = 0.0;
        for (long t = 0; t < nPeriods; t++)
        {
                PV += prices[t] / power(factor, t + 1);
        }

        return PV;
}

// Present Value of a series of future values
Vector futureValues(5); // For five years, calls constructor
for (long j = 0; j < 4; j++)
{       // The first 4 years
                futureValues[j] = 100.0;        // Vector has indexing []
}
futureValues[4] = 1100.0;

int nPeriods = 5;
double r = 0.0625;
std::cout << "**Present value, series: " << PresentValue(futureValues, nPeriods,
std::endl;
```

3.8. Summary and conclusions

3.9. Questions, exercises and projects

Consider the future value of a sum of money

- $P_n := P_0(1+r)^n$.
- n is the number of periods.
- P_n is the future value n periods from now.
- P_0 is the original principal.
- r is the interest rate per period (decimal form)

Future value of an ordinary annuity

- $P_n := A\left[\frac{(1+r)^n-1}{r}\right]$.
- A is the annuity amount.
- r is the interest rate.
- n is the number of periods.

Simple present values calculations

- $PV := P_0 = P_n \left[\frac{1}{(1+r)^n}\right]$
- P_n is the future value n periods from now.
- r is the interest rate.
- PV is the present value.

Present value of a series of future values

- $PV := \sum_{t=1}^{n} \frac{P_t}{(1+r)^t}$
- P_t is the value at period t from now.
- r is the interest rate.

Present value of an ordinary annuity

- $PV = A \left\{ \frac{1-\frac{1}{(1+r)^n}}{r} \right\}$
- A is the amount of the annuity

Continuous Compounding

- $P_n := P_0 e^{rn}$.
- r is the interest rate.
- P_0 is the original principal.
- n is the number of years.

m–Period Compounding

$$P_n = P_0\left(1 + \frac{r}{m}\right)^{mn}.$$

4. Creating Robust Classes

4.1. Introduction and objectives

4.2. Call by reference and call by value

```
double Max(double x, double y)
{
        if (x > y)
                return x;
        return y;
}

dobule d1 = 1.0;
double d2 = - 34536.00;

// Copies of d1 and d2 offered to the function Max()
double result = Max(d1, d2);
std::cout << "Maxvalue is " << result << std::endl;

class SampleClass
{
public: // For convenience only
        // This data created at compile time
        double contents[100];

public:
        SampleClass(double d)
        {
                for (int i = 0; i < 1000; i++)
                {
                        contents[i] = d;
                }
        }
        virtual ~SampleClass()
        {
                std::cout << "SampleClass instancce being deleted\n";
        }
}'

double Sum(SampleClass myClass)
{
        double result = myClass.contents[0];
        for (int i = 1; i < 1000; i ++)
        {
                result += myClass.contents[i];
        }
        return result;
}
```

```
SampleClass sc(1.0);
double sum = Sum(sc);

double Sum2(SampleClass & myClass)
{
        double result = myClass.contents[0];
        for (int i = 1; i < 1000; i++)
        {
                result += myClass.contents[i];
        }
        return result;
};

double Sum2(SampleClass* myClass);
```

4.3. Constant objects everywhere

```
double Sum3(SampleClass* myClass)
{
        // N.B. not possible to modify myClass
};
```

4.3.1. Read-only (const) member functions

```
class Point
{
private:
        void init(double xs, double ys);

        // Properties for x- and y-coordinates
        double x;
        double y;

public:

        // Constructors and destructor
        Point();        // Default constructor
        Point(double xs, double ys);    // Construct with coordinates
        Point(const Point& source);     // Copy constructor
        virtual ~Point();

        // Selectors
        double X() const;       // Return x
        double Y() const;       // Return y
        //  ...
};

double Point::X() const
{// Return x
                return x;
}

double Point::Y() const
{// Return y
                return y;
}

// Modifiers
void X(double NewX);    // Set x
void Y(double NewY);    // Set y

// Modifiers
void Point::X(double NewX)
{// Set x
                x = NewX;
}

void Point::Y(double NewY)
{// Set y
y = NewY;
}

Point p1(1.0, 3.14);

// Read the coordinate onto the Console
std::cout << "First coordinate: " << p1.X() << std::endl;
std::cout << "Second coordinate: " << p1.Y() << std::endl;

// Modify coordinates
p1.X(2.0);
p1.Y(5.0);

// Read the coordinate onto the Console
std::cout << "First coordinate: " << p1.X() << std::endl;
std::cout << "Second coordinate: " << p1.Y() << std::endl;
```

4.4. Constructors in detail

```
Point::Point()
{// Default constructor

        init(0.0, 0.0);
}

Point::Point(double xs, double ys)
{       // Normal constructor with coordinates

                init(xs, ys);
}

Point::Point(const Point &source)
{       // Copy constructor

                init(source.x, source.y);
}

void Point::init(double xs, double ys)
{       /// Initialize the point

                x = xs;
                y = ys;
}

Point();        // Default constructor
Point(const Point& source);     // Copy constructor

Point p1(1.0, 3.14);
Point p2(p1);
Point p3;
```

4.4.1. Member initialisation

```
Point::Point(double newx, double newy)
{       // Initialize using newx and newy
                init(newx, newy);
}

Point::Point(double newx, double newy) : x(newx), y(newy)
{       // Initialize using newx and newy
```

```
                // init(newx, newy); NOT NEEDED
}
```

4.5. Static member data and static member functions

```
class Point: public Shape
{
private:

        // Properties for x- and y-coordinates
        double x;
        double y;

        static Point OriginPoint;

public:

        // Other members
};

Point Point::OriginPoint = Point(0.0, 0.0);

// Accessing the "global" object
statics Point& GetOriginPoint();

Point& Point::GetOriginPoint()
{
        return OriginPoint;
}

// Work with unique Origin Point
std::cout << "Origin point: " << Point::GetOriginPoint() << std::endl;

// Now choose new coordinates for the new origin
Point::GetOriginPoint() = Point(1.0, 2.0);
std::cout << "Origin point: " << Point::GetOriginPoint() << std::endl;
```

4.6. Function overloading

```
Point();        // Default constructor
Point(double xs, double ys);     // Construct with coordinates
Point(const Point& source);      // Copy constructor
```

4.7. Non-member functions

4.8. Performance tips and guidelines

4.8.1. The “inline” keyword

```
double X() const{return x;}
double Y() const{return y;}
```

```
inline double X() const{return x;}
inline double Y() const{return y;}
```

4.8.2. Anonymous objects in function code

```
Pount MidPoint(const Point& p2) const;
```

```
Point Point::MidPoint(const Point& p2) const
{       // Calculate the point between the two points

                Point result((x + p2.x)*0.5, (y + p2.y)*0.5);
                return result;
}
```

```
Point Point::MidPoint(const Point& p2) const
{       // Calculate the point between the two points

                // Create "any old" point
                return Point( (x + p2.x)*0.5, (y + p2.y)*0.5);
}
```

```
Point pL(0.0, 0.0);
Point pU(1.0, 1.0);

Point pM = pL.MidPoint(pU);
std::cout << "Midpoint: " << pM << std::endl;

Point pM2 = pL.MidPoint(Point(1.0, 1.0));
std::cout << "Midpoint: " << pM2 << std::endl;
```

4.8.3. Loop optimization

4.9. Summary and conclusions

4.10. Questions, exercises and projects

```
// Incorrect example
int& FlunkyFunc()
{
        int result = 1;
        std::cout << "Funny";
        return result;
}

FlunkyFunc() = 12;
std::cout << FlunkyFunc() << "eee";

double& WrongFunction()
{
        double d = 3.1415; return d;
}

Point pt; std::cout << "wrong" << pt.WrongFunction() << std::endl;
```

Listing 4.1: Line segments in two dimensions.

```
// LineSegment.hpp
//
// (Finite) line segments in 2 dimensions. This class represents an undirected
// line segment.
// The functionality is basic. If you wish to get more functions then convert the
// line segment to a vector or to a line and use their respective member function
//
// This is a good example of Composition (a line segment consists of two points()
// the Delegation principle. For example, the member fucntion that calculates the
// length of a line is implemented as the distance function between the line's en
// points.
//
// (C) Datasim BV 1995-2006
//

#ifndef LineSegment_HPP
#define LineSegment_HPP

#include "Point.hpp"


class LineSegment
{
private:
        Point e1;       // End Point of line
        Point e2;       // End Point of line

public:
        // Constructors
        LineSegment();
        LineSegment(const Point& p1, const Point& p2);  // Line with end Points [
        LineSegment(const LineSegment& l);                              // Copy c
        virtual ~LineSegment();                                             /

        // Accesssing functions
        Point start() const;                                                /
        Point end() const;

        // Modifiers
        void start(const Point& pt);                                    // Set Po
        void end(const Point& pt);                                          /

        // Arithemetic
        double length() const;                                              /

        // Interaction with Points
        Point midPoint() const;                                             /
};

#endif
```

5. Operator Overloading in C++

5.1. Introduction and objectives

```
Complex z1(-23.0, 5.3);
Complex z2(2.0, 3.0);
Complex z3 = z1 * z2;
Complex z4 = 2.0 * z4;
Complex z5 = - z3;
```

5.2. What is operator overloading and what are the possibilities?

```
+       -       *       /       %       ^       &       |       ~
~       =       <       >       <=      >=      ==      +=      -=
/=      ++      --      !=      &&      >>      <<      &=      <<=
>>=     ^=      ()      []      new     delete
```

```
x + y
os << "help";   // Using the operator <<
a % b

Complex z6 = z12* 2.0;
Complex z7 = 2.0 * z2;
Complex z8 = z6 * z7;;

Matrix m1(100, 50);     // A matrix with 100 rows and 50 columns
Matrix m2 = m1; // A new matrix of same size as m1

b = -a; // b is the negative of a
b++;    // Postfix incremenet
++a;    // Prefix incremeent
```

5.3. Why use operator overloading? The advantages

```
Matrix m3 = m1 * m2;
Matrix m4 = m1.Add(m2);

DatasimDate fixed(1, 1, 94);
DatasimDate current(1, 1, 94);
int interval = 30;

for (int j = 0; i < 12; j++)
{
        current = fixed - (j*interval);
        std::cout << current << std::endl;
}

1/1/94, 2/12/93, 2/11/93, 3/10/93, 3/9/93, 4/8/93
5/7/93, 5/6/93, 6/5/93, 6/4/93, 7/3/93, 5/2/93
```

```
// Ofsets; quarters + half years
std::cout << "Offset stuff\n";
DatasimDate today;
DatasimDate d3 = today.add_quarter();
DatasimDate d4 = today.add_halfyear();

std::cout << d3 << std::endl;
std::cout << d4 << std::endl;

d3 = d3.sub_quarter();
d4 = d4.sub_quarter();

std::cout << d3 << std::endl;
std::cout << d4 << std::endl;

Vector cross(const Vector& vec) const;  // Cross product
Vector operator ^ (const Vector& vec) const;    // Cross product

double dot(cont Vector& vec) const;     // Dot product
double operator % (const Vector& vec) const;    // Dot product

Vector Vector::vtproduct(const Vector& B, const Vector& C) const
{       // Vector triple product A X (B X C)

        // Schaum Vectors page 17

        Vector tmp = B ^ C;
        return (*this) ^ tmp;
}

double Vector::stprodcut(const Vector& B, const Vector& C) const
{// Scalar triple prodcut A . (B X C)

                // Schaum Vectors, page 17
                Vector tmp = B ^ C;     // Cross
                return (*this) % tmp;   // Dot
}

class Plane
{       // A class for a plane in three dimensions
private:

        Vector n;       // Normal unit vector to plane
        Point p;        // Point on plane
};

Point Plane::closest(const Point& pt) const
{       // The point on plane closest to the point pt
                // GEMS IV page 154
                double d = n % (p - pt);
                Point result = pt + (n.componenets() * d);
                return result;
}
```

5.4. Operator overloading: the steps

```
Complex add (const Complex& c2) const;

Complex Complex:: add(const Complex& c2) const
{       // Add two complex numbers

        Complex result;
        result.x = x + c2.x;
        result.y = y + c2.y;

        return result;
}

Complex z3 = z1.add(z2);

Complex operator + (const Complex& c2) const;

Complex Complex::operator + (const Complex& c2) const
{       // Add two complex numbers
        Complex result;
        result.x = x + c2.x;
        result.y = y + c2.y;

        return result;
}

Complex Complex::operator + (const Complex& c2) const
{       // Add two complex numbers

                return Complex(x + c2.x, y + c2.y);
}
```

```
Complex operator - (const Complex& c2) const;
Complex operator * (const Complex& c2) const;
Complex operator / (const Complex& c2) const;
```

```
Complex z2(2.0, 3.0);

Complex z7 = 2.0 * z2;
Complex z8 = z2 * 2.0;
```

```
friend Complex operator * (const Complex& c, double d);
friend Complex operator * (double d, const Complex& c);
```

```
Complex operator * (const Complex & c, double d)
{       // Scaling by a double

                return Complex(c.x * d, c.y * d);
}
```

```
Complex operator * (double d, const Complex& c)
{       // Scaling by a double
                // Reuse already made operator
                return c * d;
}
```

5.4.1. A special case: the assignment operator

```
Complex z7 = z2;
```

```
Complex& operator = (const Complex& c);
```

```
Complex& Complex::operator = (const Complex& c)
{
                // Avoid doing assign to myself
                if (this == &c)
                        return *this;

                x = p.x;
                y = p.y;

                return *this;
}
```

```
Complex z0(1.0, 2.0);
Complex z1, z2, z3, z4;
z4 = z3 = z1 = z0;
std::cout << "Chain: " << z0 << z1 << z3 << z4;

+=, *=, -=, /=

z4 += z1;       // Multiply z4 by z1 and modify it

Complex& operator += (const Complex& c);

Complex& Complex::operator += (const Complex & c)
{
        x += c.x;
        y += c.y;

        return *this;
}
```

5.5. Using operator overloading for simple I/O

```
#include <iostream>
using namespace std;

double real;
double imaginary;
cout << "Creating a complex number" << endl;
cout << "Give real part: ";
cin >> real;
cout << "Give imaginary part: ";
cin >> imaginary;

// User-defined class and putput for its objects
Complex c(real, imaginary);
cout << c;

friend ostream& operator << (ostream& os, const Complex& cmp);

osteam& operator << (ostream& os, const Complex& cmp)
{       // Print the complex number

                os << "(" << cmp.x << ", " << cmp.y << ")\n";
                return os;
}
```

```
cout << "First: " << z1 << "Second: " << z2 << endl;
```

5.6. Friend functions in general

```
friend Complex exp(const Complex& c);   // Exponential
friend Complex cos(const Complex& c);   // Cosine function
friend Complex sin(const Complex& c);   // Sine function
friend Complex cosh(const Complex& c);  // Hyperbolic  cosine
friend Complex sinh(const Complex& c);  // Hyperbolic sine

Complex exp(const Complex& c)
{       // Exponential function
                double ex = std::exp(c.x);
                return Complex(ex * cos(c.y), ex * sin(c.y));
}

Complex cosh(const Complex& z)
{       // Hyperbolic cosine function
                return (exp(z) + exp(- (z))) * 0.5;
}

Complex cotanh(const Complex& z)
{       // Hyperbolc cotangent
                return cosh(z) / sinh(z);
}

Complex za = exp(Complex(0.0, 0.0));
cout << za;

Complex zs = sinh(za);
Complex zc = cosh(za);

cout << zs << zc;
Complex c2(0.0, 0.0);
cout << sinh(c2) << cosh(c2);
```

5.6.1. Friend classes

```
class A
{
private: // 'Dont tell others who my friends are
friend class B; // Hi B class, you my friend
// ...
};
```

5.7. Summary and conclusions

5.8. Exercise

Listing 5.1: Testing complex numbers.

```
// TestComplex.cpp
//
// Testing complex numbers
//

#include "Complex.hpp"

int main()
{

        Complex z1(-23.0, 5.3);
        Complex z2(2.0, 3.0);

        Complex z3 = z1 * z2;
        Complex z4 = 2.0 * z4;
        Complex z5 = - z3;

        std::cout << z1;
        std::cout << z3;
        std::cout << z5; std::cout << "****\n";

        Complex z6 = z2 * 2.0;
        Complex z7 = 2.0 * z2;
        Complex z8 = z2 * 2.0;
        Complex z9= z6* z7;

        std::cout << z6;
        std::cout << z7;
        std::cout << z8; std::cout << "****\n";

        Complex z10 = z1.add(z2);
        std::cout << z10;

        Complex z0(1.0, 2.0);
        z4 = z3 = z1 = z0;
        std::cout << "Chain: " << z0 << z1 << z3 << z4;

        z4 += z1;       // Multiply z4 by z1 and modify it
        std::cout << z4;

        double real;
        double imaginary;
        std::cout << "Creating a complex number" << std::endl;
        std::cout << "Give real part: ";
        std::cin >> real;
        std::cout << "Give imaginary part: ";
        std::cin >> imaginary;

        // User-defined class and output for its objects
        Complex c(real, imaginary);
        std::cout << c;

        std::cout << "First: " << z1 << "Second: " << z2 << std::endl;

        Complex za = exp(Complex(0.0, 0.0));
        std::cout << za;

        Complex zs = sinh(za);
        Complex zc = cosh(za);

        std::cout << zs << zc;

        Complex c2(0.0, 0.0);
        std::cout << sinh(c2) << cosh(c2);

        return 0;
}
```

We concentrate on one-dimensional and two-dimensional data structures. To this end, we introduce basic *foundation* classes, namely:

- `Array`: sequential, indexible container containing arbitrary data types.
- `Vector`: array class that contains numeric data.
- `Matrix`: sequential, indexible container containing arbitrary data types.
- `NumericMatrix`: matrix class that contains numeric data.

The code for these classes is on the accompanying CD. The classes `Array` and `Vector` are one-dimensional containers whose elements we access using a single index while

`Matrix` and `NumericMatrix` are two-dimensional containers whose elements we access using two indices.

We now discuss each of these classes in more detail.

We start with the class `Array`. This is the most fundamental class in the library and it represents a sequential collection of values. This template class that we denote by `Array<V, I, S>` has three generic parameters:

- V: the data type of the underlying values in the array.
- I: the data type used for indexing the values in the array (integral).
- S: the so-called storage class for the array.

The *storage class* is in fact an encapsulation of the STL vector class and it is here that the data in the array is actually initialised. At the moment there are specific storage classes, namely `FullArray<V>` and `BandArray<V>` that store a full array and a banded array of values, respectively.

Please note that it is **not** possible to change the size of an `Array` instance once it has been constructed. This is in contrast to the STL vector class where it is possible to let it grow.

The declaration of the class `Array` is given by:

The declaration of the class Array is given by:

```
template <class V, class I = int, class S = FullArray<V> >
class Array
{
private:
        S m_structure; // The array structure
        I m_start; // The start index
};
```

We see that `Array` has an embedded storage object of type *S* and a start index. The default storage is `FullArray<V>` and the default index type is `int`. This means that if we work with these types on a regular basis that we do not have to include them in the template declaration.

Thus, the following three declarations are the same:

```
Array<double, int, FullArray<double> > arr1;
Array<double, int> arr1;
Array<double> arr1;
```

You may choose whichever data types that are most suitable for your needs.

The constructors in `Array` allow us to create instances based on size of the array, start index and so on. The constructors are:

```
Array();         // Default constructor
Array(size t size);     // Give length start index ==1
Array(size t size, I start);    // Length and start index
Array(size_t size, I start, const V& value);    // Size, start, value
Array(const Array<V, I, S>& source);    // Copy constructor
```

Once we have created an array, we may wish to navigate in the array, access the elements in the array and to modify these elements. The member functions to help you in this case are:

```
// Selectors
I MinIndex() const;     // Return the minimum index
I MaxIndex() const;     // Return the maximum index
size_t Size() const;    // The size of the array
const V& Element(I index) const;        // Element at position

// Modifiers
void Element(I index, const V& val);    // Change element at position
void StartIndex(I index);       // Change the start index

// Operators
virtual V& operator [] (I index);       // Subscripting operator
virtual const V& operator [] (I index) const;
```

This completes the description of the `Array` class. We do not describe the class that actually stores the data in the array. The reader can find the source code on the accompanying media kit. We now discuss the `Vector` and `NumericMatrix` classes in detail. These classes are derived from `Array` and `Matrix`, respectively. Furthermore, we have created constructors for `Vector` and `NumericMatrix` classes as well. So what have these classes got that their base classes do not have? The general answer is that `Vector` and `NumericMatrix` assume that their underlying types are numeric. We thus model these classes as implementations of the corresponding mathematical structures, namely *vector space* and, *inner product* spaces.

We have implemented `Vector` and `NumericMatrix` as approximations to a vector space. In some cases we have added functionality to suit our needs. However, we have simplified things a little because we assume that the data types in a vector space are of the same types as the underlying field. This is for convenience only and it satisfies our needs for most applications in financial engineering.

Class `Vector` is derived from `Array`. Its definition in C++ is:

```
template <class V, class I=int, class S=FullArray<V> >
class Vector: public Array<V, I, S>
{
private:
        // No member data
};
```

We give the prototypes for some of the mathematical operations in `Vector`. The first is a straight implementation of a vector space; notice that we have applied operator overloading in C++:

```
Vector<V, I, S> operator - () const;
Vector<V, I, S> operator + (const Vector<V, I, S>& v) const;
Vector<V, I, S> operator - (const Vector<V, I, S>& v) const;
```

The second group of functions is useful because it provides functionality for *offsetting* the values in a vector:

```
Vector<V, I, S> operator + (const V& v) const;
Vector<V, I, S> operator - (const V& v) const;
Vector<V, I, S> operator * (const V& v) const;
```

The first function adds an element to each element in the vector and returns a new vector. The second and third functions are similar except that we apply subtraction and multiplication operators.

Class `NumericMatrix` is derived from `Matrix`. Its definition in C++ is:

```
template <class V, class I=int, class S=FullMatrix<V> >
class NumericMatrix: public Matrix<V, I, S>
{
private:
        // No member data
};
```

The constructors in `NumericMatrix` are the same as for `Matrix`. We may also wish to manipulate the rows and columns of matrices to this end to provide "set/get" functionality. Notice that we return vectors for selectors but that modifiers accept `Array` instances (and instances of any derived class!):

```
// Selectors
Vector<V, I> Row(I row) const;
Vector<V, I> Column(I column) const;

// Modifiers
void Row(I row, const Array<V, I>& val);
void Column(I column, const Array<V, I>& val);
```

Since we will be solving linear systems of equations in later chapters we must provide functionality for multiplying matrices with vectors and with other matrices:

- Multiply a matrix and a vector.
- Multiply a (transpose of a) vector and a matrix.
- Multiply two matrices

Notice that the last two functions are not member of `NumericMatrix` but are non-member friends. This ploy allows us to multiply a matrix by a vector or vice versa.

We give some simple examples showing how to create vectors and how to perform some mathematical operations on the vectors.

```
// Create some vectors
Vector<double, int> vec1(10, 1, 2.0);   // Start = 1, value 2.0
Vector<double, int> vec2(10, 1, 3.0);   // Start = 1, value 3.0

Vector<double, int> vec3 = vec1 + vec2;
Vector<double, int> vec4 = vec1 - vec2;

Vector<double, int> vec5 = vec1 - 3.14;
```

We give an example to show how to use numeric matrices. The code is:

```
int rowstart = 1;
int colstart = 1;
NumericMatrix<double, int> m3(3, 3, rowstart, colstart);
for (int i = m3.MinRowIndex(); i ⩽ m3.MaxRowIndex(); i++)
{
}
        for (int j = m3.MinColumnIndex(); j ⩽ m3.MaxColumnIndex(); j++)
        {
                m3(i, j) = 1.0 /(i + j -1.0);
        }
print (m3);

MinRowIndex: 1 , MaxRowIndex: 3
MinColumnIndex: 1 , MaxColumnIndex: 3
MAT:[
Row 1 (1,0.5,0.333333,)
Row 2 (0.5,0.333333,0.25,)
Row 3 (0.333333,0.25,0.2,)]
```

6. Memory Management in C++

6.1. Introduction and objectives

6.2. Single objects and arrays of objects on the stack

```
int main()
{
        {       // Define a scope
                int j = 2;
                cout << j << endl;
        }

        cout << j;

        return 0;
}

int myArr[10];

// Initialise the array
for (int j = 0; j < 10; j++)
{
                myArr[j] = j + 1;
                cout << myArr[j] << ",";
}

cout << endl << myArr[-1];
cout << endl << myArr[1000];

class SimpleOption
{
public:
                double T;
                double K;
                // ...

                SimpleOption () { T = 1.0; K = 100.0; }

        void print() const
        {       // Read contents of option

        cout << "Expiry: "<< T << ", " << "Strike: " << K;
        }
};

{       // Define a scope

                SimpleOption opt1;
                opt1.print();
}
```

```
SimpleOption myPortfolio[10];

// Initialise the array
for (int j = 0; j < 10; j++)
{
        myPortfolio[j].print();
}

myPortfolio[-1].print();
myPortfolio[1000].print();
```

6.3. Special operators: “new” and “delete”

6.3.1. Single objects

```
SimpleOption* opt;
SimpleOption* opt2;

// Call default constructor
opt = new SimpleOption;
opt -> print();

(* opt).print();
// Call constructor with 2 parameters
opt2 = new SimpleOption(0.25, 90.0);
opt2 -> print();

delete opt;
delete opt2;

opt = new SimpleOption;
(* opt).print();
delete opt;
```

6.3.2. Arrays of objects

```
// Now create an array of options
SimpleOption* optArray;

const int N = 10;
optArray = new SimpleOption[N]; // Default constructor called
```

```
for (int j = 0; j < N; j++)
{ // Member data public for convenience only
        optArray[j].T = 1.0; // 1 year expiry
        optArray[j].K = 100.0; // Strike price
        optArray[j].print();
}

delete [] optArray;
```

6.4. Small application: working with complex numbers

```
Complex myFunc(const Complex& z)
{ // Single valued function of a complex variable
}
return z * z;

Complex z1(1.0, 1.0);
Complex z2(2.0, 3.0);

Complex z3 = z1 * z2;
Complex z4 = 2.0 * z3;
Complex z5 = - z3;

// Create a dynamic list of Complex numbers
int Size = 5;
Complex* cpArray = new Complex[Size];
cpArray[0] = z1;
cpArray[1] = z2;
cpArray[2] = z3;
cpArray[3] = z4;
cpArray[4] = z5;

// Call function and print values for each z
for (int j = 0; j < Size; j++)
{
                cout << myFunc(cpArray[j]) << ", ";
}

delete [] cpArray;

Complex ComplexProduct(Complex* carr, int n)
{ // Complex function of several complex variables
        Complex product = carr[0];
        for (int j = 1; j < n; j++)
        {
                product * = carr[j];
        }

        return product;

}

Complex ComplexSum(Complex* carr, int n)
{ // Complex function of several complex variables

        Complex sum = carr[0];

        for (int j = 1; j < n; j++)
        {
                sum += carr[j];
        }

        return sum;

}

const int N = 5;
Complex fixedArray[5]; // The constant "5" is mandatory
for (int i = 0; i < Size; i++)
{
                fixedArray[i] = Complex ((double) i, 0.0);
}

Complex product = ComplexProduct(fixedArray, Size);
cout << "Product: " << product << endl;

Complex sum = ComplexSum(fixedArray, Size);
cout << "Sum: " << sum << endl;
```

6.5. Creating an array class

6.5.1. Memory allocation and deallocation

```
private:
        Complex* arr;
        int size;
```

```
ComplexArray(int size);
ComplexArray(const ComplexArray& source);

// Constructor with size
ComplexArray::ComplexArray(int Size)
{
        arr=new Complex[size];
        size=Size;
}

// Copy constructor
ComplexArray::ComplexArray(const ComplexArray& source)
{
        // Deep copy source
        size=source.size;
        arr=new Complex[size];
        for (int i=0; i<size; i++) arr[i]=source.arr[i];
}

virtual ~ComplexArray();

ComplexArray::~ComplexArray()
{
        delete[] arr;
}
```

6.5.2. Accessing functions

```
int MinIndex() const; // Smallest index in array
int MaxIndex() const; // Largest index in array

const Complex& operator[](int index) const;
Complex& operator[](int index);

Complex ComplexSum(const ComplexArray& carr, int n)
{       // Complex function of several complex variables

                Complex sum = carr[carr.MinIndex()];

                for(int j = carr.MinIndex() + 1; j <= carr.MaxIndex(); j++)
                {
                        sum += carr[j];
                }
                return sum;
}
```

6.5.3. Examples

```
ComplexArray fixedArray(5);
for (int i=fixedArray.MinIndex();i<=fixedArray.MaxIndex(); i++)
{
        fixedArray[i] = Complex ((double) i, 0.0);
}
```

6.5.4. The full header file

Listing 6.1: Simple Complex Array class.

```
// ComplexArray.hpp
//
// Simple Complex Array class.
//
// (C) Datasim Education BV  1995-2003

#ifndef ComplexArray_hpp
#define ComplexArray_hpp

#include "Complex.hpp"

class ComplexArray
{
private:
        Complex* arr;
        int size;

        ComplexArray();

public:

        // Constructors & destructor
        ComplexArray(int size);
        ComplexArray(const ComplexArray& source);
        virtual ~ComplexArray();

        // Selectors
        int Size() const;
        int MinIndex() const;   // Smallest index in array
        int MaxIndex() const;   // Largest index in array
        // Operators
        const Complex& operator[](int index) const;     // Index operator for con
        Complex& operator[](int index);                         // Index operator

        ComplexArray& operator = (const ComplexArray& source);
};

#endif // ComplexArray_hpp
```

6.6. Summary and conclusions

6.7. Exercises

```
// Compile-time
Complex matrix [10][20];

// Run-time (pointers to pointers)
Complex** matrix2;

// "Hybrid": pointers to fixed arrays
Complex* matrix3[10];

void MySwap(Complex c1, Complex c2); // Call by value
void MySwap2(Complex& c1, Complex& c2); // Call by reference
void MySwap3(const Complex& c1, const Complex& c2); // const references
void MySwap4(Complex* c1, Complex* c2); // Using pointers
void MySwap5(const Complex* c1, const Complex* c2); // const pointers

struct Point
{
        double x;       // x coordinate
        double y;       // y coordinate
}

class Stack
{
private:

                // EXX What is the member data in this case?

                Stack(const Stack& source);     // Copy constructor
                // Operators
                Stack& operator = (const Stack& source);

public:
                // Constructors & destructor
                Stack();        // Default constructor
                Stack(int NewSize);     // Initial size of the stack

                void Push(double NewItem);      // Push element onto stack
                double Pop();   // Pop last pushed element
};

#endif // STACK HPP

#include <iostream>
using namespace std;

int main()
{
        double m[10][20];

        m[0][1] = 2.0;

        // etc.

        cout << m[0][1] << endl;

        return 0;
}
```

6.8. Review questions and comments

```
double mat[10][20];
```

7. Functions, Namespaces and Introduction to Inheritance

7.1. Introduction and objectives

7.2. Functions and function pointers

7.2.1. Functions in financial engineering

7.2.2. Function categories

A common notation for a function f from D to R is

$$f : D \to R.$$

Functions can be composed. For example, suppose f is a mapping from D to $R1$ and g is a mapping from R_1 and g is a mapping from $R1$ to $R2$ then the composition of g and f is defined by

$$(gf)(x) = g(f(x)) \text{ for all } x \text{ in } D.$$

Notice that the range of the composed mapping gf is $R2$.

```
void genericFunction (double myX, double myY, double (* f) (double x, double y))
{

        // Call the function f with arguments myX and myY
        double result = (* f)(myX, myY);

        cout << "Result is: " << result << endl;
}

double add(double x, double y)
{
        cout << "** Adding two numbers: " << x << ", " << y <<
endl;
        return x + y;
}
double multiply(double x, double y)
{
        cout << "** Multiplying two numbers: " << x << ", " <<
y << endl;
        return x * y;
}
```

```
double subtract(double x, double y)
{
        cout << "** Subtracting two numbers: " << x << ", " <<
y << endl;
        return x - y;
}
```

7.2.3. Modelling functions in C++

```
int main()
{
        double x = 3.0;
        double y = 2.0;
        genericFunction(x, y, add);
        genericFunction(x, y, multiply);
        genericFunction(x, y, subtract);

        return 0;
}
```

7.2.4. Application areas for function pointers, part I

7.3. An introduction to namespaces in C++

```
namespace MyFunctions
{
        double diffusion (double x) { return x; }
        double convection (double x) { return x* x; }
}
namespace YourFunctions
{
        double diffusion (double x) { return 2.0; }
        double convection (double x) { return 1.0; }
}

cout << YourFunctions::convection (10.0) << endl;

using YourFunctions::convection;
cout << convection (10.0) << endl;

using namespace MyFunctions;
cout << "Directive: \ n";
cout << convection (10.0) << endl;
cout << diffusion (2.0) << endl;

using namespace YourFunctions;
cout << convection (10.0) << endl;
cout << diffusion (2.0) << endl;
```

7.3.1. Some extra functionality

```
namespace YA = YourFunctions; // Define alias NS called YA
cout << YA::diffusion (2.0) << endl;

namespace StandardInterface
{
        // Namespace consisting of function pointers
        double (* func1) (double x);
        double (* func2) (double x, double y);
}

namespace Implementation1
{
        double F1 (double x) { return x; }
        double F2 (double x, double y) { return x* y; }
}

namespace Implementation2
{
        double G1 (double x) { return -x; }
        double G2 (double x, double y) { return -x* y; }
}

// Assign the function pointers from NS
StandardInterface::func1 = Implementation1::F1;
StandardInterface::func2 = Implementation1::F2;

using namespace StandardInterface;
cout << func1(2.0) << ", " << func2(3.0, -4.0) << endl;

func1 = Implementation2::G1;
func2 = Implementation2::G2;
cout << func1(2.0) << ", " << func2(3.0, -4.0) << endl;
```

7.4. An introduction to the inheritance mechanism in C++

```
class Person
{
public: // Everything public, for convenience only
        // Data
        string nam;     // Name of person
        DatasimDate dob;        // Date of birth
        DatasimDate createdD;   // Internal, date created
        DatasimDateTime createdT;       // Internal, time created
public:
        // Public functions
};

Person (const string& name, const DatasimDate& DateofBirth)
{
        nam = name;
        dob = DateofBirth;
        createdD = DatasimDate();       // default, today REALLY!
        createdT = DatasimDateTime();   // default, now REALLY!
}

void print() const
{
// Who am I?
        cout << "\ n** Person Data ** \ n";
        cout << "Name: " << nam << "Date of birth: " << dob
                << ", Age: " << age() << endl;
        cout << "Object created: " << createdD
                << " "; createdT.print(); cout << endl;
}
int age() const
{
        return int( double(DatasimDate() - dob) / 365.0);
}

DatasimDate myBirthday(29, 8, 1952);
string myName ("Daniel J. Duffy");
Person dd(myName, myBirthday);
dd.print();

DatasimDate bBirthday(06, 8, 1994);
string bName ("Brendan Duffy");
Person bd(bName, bBirthday);
bd.print();
```

7.4.1. Basic inheritance structure

Listing 7.1: Output of `Person.cpp`.

```
** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 67
Object created: 19/11/2019 09:11:19

** Person Data **
Name: Brendan Duffy, Date of birth: 6/8/1994, Age: 25
Object created: 19/11/2019 09:11:19

** Employee Data **

** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 67
Object created: 19/11/2019 09:11:19

Function: Cuchulainn Chief, Salary: 0.01, Retires at: 65
-2 years to retirement.
Working with pointers I

** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 67
Object created: 19/11/2019 09:11:19
Working with pointers II

** Employee Data **

** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 67
Object created: 19/11/2019 09:11:19

Function: Cuchulainn Chief, Salary: 0.01, Retires at: 65
-2 years to retirement.
```

```cpp
class Employee : public Person
{
public: // For convenience only
        std::string fun; // Function
        double sal; // Salary
        int rAge; // Retirement age
        // other functions
};
```

7.4.2. Adding new functionality

```
void print() const
{ // Who am I and what do I do?

// Print Base class data
cout << "\ n** Employee Data ** \ n";
Person::print(); // Call print() in the base class

// Now print data from current derived class
cout << "\ nFunction: " << fun << ", Salary: " << sal
        << ", Retires at: " << rAge << endl
        << YearsToRetirement() << " years to retirement.\ n";
}

int YearsToRetirement() const
{ // How many more years slogging away at C++?
        return rAge - age();
}

Employee dde (myName, myBirthday, string("Cuchulainn Chief"), 0.01, 65);
dde.print();
```

Listing 7.2: Output of `TestPersonAndEmployee.cpp`.

```
** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 67
Object created: 19/11/2019 09:24:06

** Person Data **
Name: Brendan Duffy, Date of birth: 6/8/1994, Age: 25
Object created: 19/11/2019 09:24:06

** Employee Data **

** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 67
Object created: 19/11/2019 09:24:06

Function: Cuchulainn Chief, Salary: 0.01, Retires at: 65
-2 years to retirement.
Working with pointers I
```

```
** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 67
Object created: 19/11/2019 09:24:06
Working with pointers II

** Employee Data **

** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 67
Object created: 19/11/2019 09:24:06

Function: Cuchulainn Chief, Salary: 0.01, Retires at: 65
-2 years to retirement.
```

7.4.3. Overriding functionality: polymorphic and non-polymorphic behaviour

```
Employee dde (myName, myBirthday, string("Cuchulainn"), 0.01, 65);
Person* p = &dde;
p → print();
```

```
** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 52
Object created: 15/4/2005 19:11:32
Working with pointers II
```

```
virtual void DeepPrint() const
{
        print(); // Calls Person::print()
}

void DeepPrint() const
{
        print(); // Calls Employee::print()
}
```

```
Employee dde (myName, myBirthday, std::string("Cuchulainn Chief"), 0.01, 65);
Person* p = &dde;
p → DeepPrint();
```

```
** Employee Data **
** Person Data **
Name: Daniel J. Duffy, Date of birth: 29/8/1952, Age: 52
Object created: 15/4/2005 19:11:32
```

```
Function: Cuchulainn, Salary: 0.01, Retires at: 65
13 years to retirement.

Person* parr[3]; // Array of pointers
parr[0] = new Employee( ... );
parr[1] = new TaxPayer( ... );
parr[2] = new ShareHolder( ... );
// Now print the array
for (int j = 0; j < 3; j++)
{
        parr[j] -> DeepPrint();
}
```

7.5. Multiple inheritance

```
class D: public Base1, public Base2
{
        // Members of D here
};

class D : public Base1
{
private:
        Base2* base2;
public:
        // Members here
};
```

7.6. Solution of nonlinear equations

In this section we discus the problem of finding real values x that satisft the equation

$$f(x) = 0$$

where f is a real-valued function. The methods to be discussed are:

- Bisection method.
- Newton's method.
- Secant method.
- Steffensen iteration

The *bisection method* assume that the function $f(x)$ has a zero in the interval (a,b) and we assume that the signs are opposite at the end points, that is $f(a)\,f(b) < 0$. The essence of the method is to divide the interval into equals parts until we arrive at an interval that is so small that contains the zero of the function and is small enough to satisfy the given tolerance. The basic algorithm is defined using a sequence of intervals of ever-diminishing size:

$$(a,b) \supset (a_1,b_1) \supset (a_2,b_2) \supset (a_3,b_3) \supset \cdots$$

where

$$(a_k,b_k) = \begin{cases} (m_k, b_{k-1}) & \text{if } f(m_k) < 0 \\ (a_{k-1}, m_k) & \text{if } f(m_k) > 0 \end{cases}$$

and

$$m_k = \frac{1}{2}\left(a_{k-1} + b_{k-1}\right).$$

After n steps the root is in an interval having a length given by

$$b_n - a_n = 2^{-1}\left(b_{n-1} - a_{n-1}\right) = 2^{-n}\left(b-a\right).$$

Thus the deviation from the exact root α is given by

$$\alpha = m_{n+1} \pm d_n, \quad d_n = 2^{-n-1}\left(b-a\right).$$

In general we are interested in locating the zero of the function to within a given tolerance `TOL`. This means that we wish to calculate the number of subdivisions of the original interval (a,b). To this end, some arithmetic based on the equation below give us the following estimate:

$$n > \frac{\log\left(\frac{b-a}{\mathtt{TOL}}\right)}{\log 2} - 1.$$

The advantage of the Bisection method is that we always can define an interval of arbitrary size in which the zero is located. The disadvantage is that convergence is slow. In fact, at each step we gain one binary digit in accuracy. We note also that the rate of convergence is independent of the given function $f(x)$. The method may be used to give us good initial approximations to more sophisticated nonlinear solvers.

Newton's method (or the *Newton-Raphson* method as it is also called) is probably one of the most famous iterative schemes in Numerical Analysis.

The main advantage of the Newton-Raphson method is that it converges quickly. We say thaat its order of convergence is two by which we mean that the error at each iteration decreases quadratically:

$$x_{n+1} = x_n + h_n, \quad h_n = -\frac{f(x_n)}{f'(x_n)}.$$

The disadvantage is that the choice of the initial approximation is vitally important. If this is not chosen carefully the method may not converge at all or it may even converge to the wrong solution. Furthermore, we must have an analytical expression for the derivative of f and this may not always be available.

The *secant method* can be derived from the Newton-Raphson methody by approximating the derivate by a divided difference. The resulting iterative scheme now becomes:

$$x_{n+1} = x_n + h_n, \quad h_n = -f_n \frac{x_n - x_{n-1}}{f_n - f_{n-1}}, \quad f_n \neq f_{n-1}.$$

We note that the secant needs two initial approximations in contrast to Newton-Raphson, that only needs one initial approximation. In general, we must be careful when programming the secant method when the function values are close and/or the solutions at levels n and $n+1$ are close; in such cases we calculate terms that effectively $0/0$!

A disadvantage of the secant method is that it is only first-order accurate. In order to achieve second-order accuracy without having to evaluate derivatives we propose *Steffensen's method*, given by the scheme:

$$x_{n+1} = x_n - \frac{f(x_n)}{g(x_n)}$$

$$g(x_n) = \frac{f(x_n + f(x_n)) - f(x_n)}{f(x_n)}.$$

This scheme requires two function evaluations but no computation of a derivative. With the exception of the bisection method, the choice of the initial approximation is vital and it must be “close” to the exact solution, otherwise the iterative scheme may not converge. There are iterative methods based on the continuation (or homotopy) methods that converge to the true solution een when the initial guess is not good, but a discussion of these techniques it outside the scope of this book.

7.7. Nonlinear solvers in C++: design and implementation

```
class NonlinearSolver
{
public:
        double (*myF)(double x);        // Function whose root we want
        double tol;     // Desired accuracy
public:
        NonlinearSolver(double (* function)(double)) { }
        virtual double solve() = 0;
};

class SteffensensSolver : public NonlinearSolver
{ // One-step nonlinear solver
private:
        double x0;      // Initial guess
        double xPrevious, xCurrent;
        long n; // Number of iterations taken
public:
        SteffensensSolver(double guess, double (* myFunc)(double x))
        {
                x0 = guess; xPrevious = x0;
                myF = myFunc;
        }

double solve()
        {
                double tmp; double hn; n = 1; xPrevious = x0;
L1:
                tmp = myF(xPrevious);
                hn = (myF(xPrevious + tmp) - tmp)/ tmp;
                hn = tmp/hn;
                xCurrent = xPrevious - hn;

                xPrevious = xCurrent;
                n++;
                if (::fabs(hn) ≤ tol)
                {
                        return xCurrent;

                }
                gotoL1;
        }
        void printStatistics() const
        {
                cout << "\ nData pertaining to Steffensen's method\ n";
                }
                cout << "Value: " << xCurrent << endl;
                cout << "Number of iterations (actual): "
                << n << endl;
        }
};
```

7.8. Applying nonlinear solvers: calculating volatility

```
double CallPrice(double sig)
{
        // Test case Haug p. 172; student exercise to extend it
        double S = 59.0;
        double K = 60.0;
        double r = 0.067;
        double marketPrice = 2.82;      // The call price
        double b = r;
        double T = 0.25;        // Three months

        double tmp = sig * sqrt(T);
        double d1 = ( log(S/K) + (b+ (sig* sig)* 0.5 ) * T )/ tmp;
        double d2 = d1 - tmp;

        double calculatedValue = (S * exp((b-r)* T) * N(d1)) - (K * exp(-r * T)*

        // Function in the form f(x) = 0
        return marketPrice - calculatedValue;
}

// Steffensen's method
double guess = 0.2;

SteffensensSolver steff(guess, CallPrice);
steff.tol = 0.0001;

double resultST= steff.solve();
cout << "Steffensen's Method: " << resultST << endl;

steff.printStatistics();
```

7.9. Summary and conclusions

7.10. Exercises and projects

$$y = f(a) + \frac{f(a) - f(b)}{a - b}(x - a)$$

giving the estimate for c:

$$c = a - \frac{(a - b) f(a)}{f(a) - f(b)}.$$

This all translates to the algorithm:

$$x_n = x_{n-1} - \frac{(x_{n-1} - x_{n-2}) f(x_{n-1})}{f(x_{n-1}) - f(x_n - 2)}.$$

8. Advanced Inheritance and Payoff Class Hierarchies

8.1. Introduction and objectives

8.2. The **virtual** specifier and memory deallocation

```
class B
{ // Class with non-virtual destructor
private:
        double* d;
public:
        B() { d = new double (1.0); }�
        B() { cout << "Base destructor\ n"; delete d; }
};

class D : public B
{ // Derived class
private:
        int* iarr;
public:
        D(int N) { iarr = new int[N]; }�
        D() { cout << "Derived destructor\ n"; delete [] iarr; }
};

int main()
{
        {
                B* b = new D(10);
                delete b;
        }

        return 0;
}
```

```
virtual �B() { cout << "Base destructor\ n"; delete d; }
```

```
Derived destructor
Base destructor
```

8.3. Abstract and concrete classes

```
class Payoff
{
public:
        // Constructors and destructor
        Payoff();       // Default constructor
        Payoff(const Payoff& source);   // Copy constructor
        virtual ⍰Payoff();// Destructor

        // Operator overloading
        Payoff& operator = (const Payoff& source);

        // Pure virtual payoff function
        virtual double payoff(double S) const = 0; // Spot price S
};

class CallPayoff: public Payoff
{
private:

        double K;       // Strike price

public:
        // Constructors and destructor
        CallPayoff();
        CallPayoff(double strike);
        CallPayoff(const CallPayoff& source);
        virtual ⍰CallPayoff();

        // Selectors
        double Strike() const;  // Return strike price

        // Modifiers
        void Strike(double NewStrike);  // Set strike price

        CallPayoff& operator = (const CallPayoff& source);

        // Implement the pure virtual payoff function from base class
        double payoff(double S) const; // For a given spot price
};
```

We see that this class has private number data representing the strike price of the call option as well as public set/get member for this data. Furthermore, we have coded all essential functions in this class:

- Default constructor.
- Copy constructor.
- Virtual destructor.
- Assigment operator.

We call this *canonical header file*. Finally, we must implement the `payoff()` function, otherwise `CallPayoff` will itself be an abstract class. We now examine the bodies of the member functions of `CallPayoff`. In general, a constructor in a derived class must initialise the local as well as the data in the class that it is derived from. For the former case we use normal assignment but we can also use the so-called colon syntax to initialise the data in a base class. For example, the copy constructor in `CallPayoff` is given by:

```
CallPayoff::CallPayoff(const CallPayoff& source): Payoff(source)
{ // Copy constructor
        K = source.K;
}

CallPayoff& CallPayoff::operator = (const CallPayoff &source)
{ // Assignment operator

                // Exit if same object
                if (this==&source) return * this;

                // Call base class assignment
                Payoff::operator = (source);

                // Copy state
                K = source.K;

                return * this;
}

double CallPayoff::Strike() const
{
// Return K
        return K;
}

void CallPayoff::Strike(double NewStrike)
```

```
{// Set K
        K = NewStrike;
}

double CallPayoff::payoff(double S) const
{ // For a given spot price

        if (S > K)
                return (S - K);
        return 0.0;
        // remark; possible to say max (S - K, 0)if you prefer
}

double BullSpreadPayoff::payoff(double S) const
{ // Based on Hull's book

        if (S ⩾ K2)
                return K2 - K1;
        if (S ⩽ K1)
                return 0.0;

        // In the interval [K1, K2]
        return S - K1;
}
```

8.3.1. Using payoff classes

```
CallPayoff call(20.0);

cout << "Give a stock price (plain Call): ";
double S;
cin >> S;

cout << "Call Payoff is: " << call.payoff(S) << endl;

double K1 = 30.0;       // Strike price of bought call
double K2 = 35.0;       // Strike price of sell call
double costBuy = 3.0;   // Cost to buy a call
double sellPrice = 1.0; // Sell price for call
BullSpreadPayoff bs(K1, K2, costBuy, sellPrice);
cout << "Give a stock price (BullSpread): ";
cin >> S;
```

```
cout << "Bull Spread Payoff is: " << bs.payoff(S) << endl;
cout << "Bull Spread Profit is: " << bs.profit(S) << endl;

double BullSpreadPayoff::profit(double S) const
{ // Profit
        return payoff(S) - (buyValue - sellValue);
}
```

The techniques developed in this section can be used in other applications in which we need to create derived classes in C++.

8.4. Lightweight payoff classes

```
class Payoff
{
private:
        PayoffStrategy* ps;
public:
        // Constructors and destructor
        Payoff(PayoffStrategy& pstrat);

// Other member functions
};

class PayoffStrategy
{
public:
        virtual double payoff(double S) const = 0;
};

class CallStrategy : public PayoffStrategy
{
private:
        double K;
public:
        CallStrategy(double strike) { K = strike;}
        double payoff(double S) const
        {
                if (S > K)
                        return (S - K);

                return 0.0;
        }
};

// Create a strategy and couple it with a payoff
CallStrategy call(20.0);
Payoff pay1(call);
```

8.5. Super lightweight payoff functions

```
class OneFactorPayoff
{
private:
        double K;
        double (* payoffFN)(double K, double S);
public:
        // Constructors and destructor
        OneFactorPayoff(double strike, double(* payoff)(double K,double S));
        // More ...

        double payoff(double S) const; // For a given spot price

};
```

The bodies of these member functions are given by:

```
OneFactorPayoff::OneFactorPayoff(double strike, double (* pay)(double K, double S
{
        K = strike;
        payoffFN = pay;
}
double OneFactorPayoff::payoff(double S) const
{ // For a given spot price

        return payoffFN(K, S); // Call function
}
```

An example of specific payoff functions is:

```
double CallPayoffFN(double K, double S)
{
        if (S > K)
        return (S - K);
}
return 0.0;

double PutPayoffFN(double K, double S)
```

```
{
        // max (K-S, 0)
        if (K > S)
                return (K - S);

        return (K -return 0.0;
}

int main()
{
        OneFactorPayoff pay1(20.0, CallPayoffFN);
        cout << "Give a stock price (plain Call): ";
        double S;
        cin >> S;

        cout << "Call Payoff is: " << pay1.payoff(S) << endl;

        OneFactorPayoff pay2(20.0, PutPayoffFN);

        cout << "Give a stock price (plain Put): ";
        cin >> S;

        cout << "Put Payoff is: " << pay2.payoff(S) << endl;

        return 0;
}
```

8.6. The dangers of inheritance: a counterexample

We now discuss some of the issues to look out for when creating a C++ class hierarchy. In general, deriving a class D from a class B introduces dependencies between B and D; in other words changes to B may result to changes in D.

Even though the inheritance mechanism is very powerful and useful it can be misused and its capabilities stretched to unacceptable limits. We now give an example to show how subtle errors can enter your code by the incorrect use of the inheritance mechanism. We shall then show how to resolve the corresponding problems induced by using the mechanism. We shall then show how to resolve the corresponding problems induced by using the mechanism. The example that we take is unambiguous and there is little chance of it being misinterpreted. To this end, we create two-dimensional shapes, namely rectangles and squares. Of course, we would like to create code that is as resuable as possible. The first solution that might spring to mind is to say that a square is a specilisation of a rectangle. This sounds reasonable. We investigate the consequences of this decision. The interface for the rectangle class is:

```
class Rectangle : public Shape
{
protected: // Derived classes can access this data directly
        Point bp; // Base point, Point class given
        double h; // height
        double w; // width

public:
        Rectangle()
        {
                bp = Point();
                h = 1.0;
                w = 2.0;
        }
        Rectangle(const Point& basePoint, double height, double width)
        {
                bp = basePoint;
                h = height;
                w = width;
        }

        void setHeight(double newHeight)
        {
                h = newHeight;
        }

        void setWidth(double newWidth)
        {
                w = newWidth;
        }

        void print() const
        {
                cout << bp;
                cout << "Dimensions (H, W): " << h << ", "
                << w <<endl;
        }
};
```

Thus, we can create rectangles and change its dimensions. We now derive a square class from it as follows

```
class BadSquare : public Rectangle
{ // Version 1, a Square is a Rectangle
private:
        // No member data, inherited from Rectangle
public:
        BadSquare() : Rectangle (Point(), 1.0, 1.0)
        {
                // We must implement this, otherwise this
                        // default constructor inherits a default rectangle
        }

        BadSquare (const Point& basePoint, double size)
        {
                bp = basePoint;
                h = w = size;   // Starts off well
        }
};
```

This class has no member data because it is derived from `Rectangle`. In this case we defined the data in `Rectangle` as being protected (this means that it is directly available to derived classes) for convenience only. Furthermore, we need to say something about default constructors. Some rules are:

- A default constructor will be generated if you do not provide one
- The default constructor in the derived class will automatically call the default constructor in the base class

In general, we prefer to write default constructors in all classes because we control what default behaviour will be. Secondly, a default constructor in class `BadSquare` is not necessarily a default `Rectangle`! For this reason we must do things differently as can be seen in the above code.

We now come to the next attention point. Since squares are publicly derived from rectangles we can call all the latter's member function for instances of `BadSquare`. Let us take an example:

```
BadSquare bs;
bs.print();
```

This code creates a square at the origin with a side of 1. Now we call a member function that is inherited from the base class:

```
// Now change the size
bs.setHeight(2.0);
bs.print();
```

When we print the square we see that the height and width are different. So we do not have a square anymore!

What is the problem? At a superficial level the square inherits functionality that destroys its consistency. At a deeper level, a square is not a specialisation of a rectangle.

We resolve this problem by using a slightly different design technique that is well documented in the literature but is not as well known as inheritance.

In general terms, we implement a new class that uses a rectangle as a private member data and we develop the member functions in the new square while we supress those functions that are not relevant. That UML diagram is shown in Figure and here we use the *Composition* technique: a square is composed from a rectangle. Furthermore, square delegates needed functionality to rectangle. The source code is given by:

```
class GoodSquare : public Shape
{ // Version 2, adapts a Rectangle's interface
        private:
        Rectangle r;
public:
        GoodSquare ()
        {
                double size = 1.0;
                r = Rectangle (Point(), size, size); // Unit square
        }
        GoodSquare (const Point& basePoint, double size)
        {
                r = Rectangle (basePoint, size, size);
        }

        void setSize(double newSize)
        { // An adaptor function, this ensures that constraints
                // on the square are always satisfied

                        r.setHeight(newSize);
                        r.setWidth(newSize);
        }
        void print() const
        {
                // Delegate to the embedded 'rectangles output

                r.print();
        }
};
```

Not much can go wrong now; the square publishes its member functions to client code that have no idea that the square is implemented as a rectangle. The following source code shows how to use a new functionality:

```
GoodSquare gs;
```

```
gs.print();

// Now change the size
gs.setSize(2.0);
gs.print();
```

Concluding, we have given an example to show that inheritance is not always the best way to extend the functionality of a class. The above discussion has relevance to many situations in object-orient design. In particular, the same warnings apply when developing C++ class hierarchies in financial engineering applications.

8.7. Implementation inheritance and fragile base class problem

We know that a derived class `D` can use the public and protected members in a class `B` that is is derived from. In general, when a member (data or functions) is changed in `B` then it may be necessary to change the code in `D`. Of course, a change in `D` may trigger other changes in *its* derived classes. This particular situation is called the *Fragile Base Class Problem*.

There are some possible solutions to this problemm, depending on the context:

- Instead of using inheritance we could consider using Composition and adapting the interface of `B`.
- Try to create base classes with as few member data as possible.
- Avoid deep inheritance hierarchies.

8.7.1. Deep hierarchies

This is a problem that we, as C++ community have inherited (no pun intended) from the 1990's. In the past we had seen inheritance hierarchies with the "most deep" derived class having up to six indirect base classes. Some of the disadvantages are:

- It has been proven that the human brain can hold approximately seven pieces of information in short-term memory at any one moment in time; this number is much less than the amount of information that we need to understand when working with deep C++ class hierarchies.
- The chances that the structural relationships between the classes are correct decreases as the hierarchy grows.

8.7.2. Multiple inheritance problems

We have discussed multiple inheritance in Chapter 7. I do not use it for three main reasons:

- Its use leads to unmaintainable software systems.
- It is a flawed way of thinking about relationships between entities in general and C++ classes in particular (in the author's opinion)
- In some cases we can use single inheritance for base class `B1` and composition with class `B2` instead of deriving class `D` from both`B1` and `B2`.

In the last case we see that `D` inherits all of the functionality from `B1` and it *adapts* the interface functions in `B2` to suit its own needs.

8.8. Two-factor payoff functions and classes

In this section we give an overview of some kinds of options that depend on two or more underlying assets. These are called *correlation options* in general. Our interest in these options is to cast them in PDE form. In particular, we must define the payoff function, boundary conditions and the coefficients of the PDE and we focus on the following specific types:

- Exchange options
- Rainbow options
- Basket options
- Best/worst options
- Quotient options
- Foreign exchange options
- Spread options
- Dual-strike options
- Out-performance options

We have discussed the finnancial and mathematical background to these option types in detail in Duffy (2006). In particular we solved the partial differential equations associated with these option types by the finite difference method. In this book we concentrate on the C++ implementation for the payoff functions for these types and to this end we create a C++ class hierarchy consisting of a base class and derived classes, with one derived class for each payoff function. The abstract base class (note, it has no member data) is defined as:

```
class MultiAssetPayoffStrategy
{ // Interface specification
public:
        virtual double payoff(double S1, double S2) const = 0;
};
```

There is no default structure or behaviour defined here, only pure virtual function specification describing the pricing function based on two underlyings. Each derived class must implement this function. For example, for a basket option class we have the specification:

```
class BasketStrategy : public MultiAssetPayoffStrategy
{ // 2-asset basket option payoff
private:
        double K;       // Strike
        double w;       // +1 call, -1 put
        double w1, w2; // w1 + w2 = 1
public:
        // All classes need default constructor
        BasketStrategy(): MultiAssetPayoffStrategy()
        { K = 95.0; w = +1; w1 = 0.5; w2 = 0.5;
        }
        BasketStrategy(double strike, double cp,double weight1, double weight2) :
        { K = strike; w = cp; w1 = weight1; w2 = weight2;
        }
        double payoff(double S1, double S2) const
        {
                double sum = w1* S1 + w2* S2;
                return DMax(w* (sum - K), 0.0);
        }
};
```

Please not that we use the colon syntax ":" to call the constructor of the base class from the derived class. This ensures that the member data in the base class will be initialised. Of course, the base class in this case has no member data (at the moment) but it might have in a future version of the software.

We have used two-factor payoff classes in conjunction with the finite difference method (see Duffy, 2006). The actual data and the relationship with the payoff function can be seen in the following classes (note that we have taken a basket option payoff for readability but it is easy to extend to be more general case):

```
class TwoFactorInstrument
{ // Empty class
public:

};
class TwoFactorOptionData : public TwoFactorInstrument
{
//private:
public: // For convenience only
        // An option uses a polymorphic payoff object
        BasketStrategy pay;
public:
        // PUBLIC, PURE FOR PEDAGOGICAL REASONS, 13 parameters
        double r;       // Interest rate
        double D1, D2;  // Dividends
        double sig1, sig2;      // Volatility
        double rho;     // Cross correlation
        double K;       // Strike price, now place in IC
        double T;       // Expiry date
        double SMax1, SMax2;    // Far field condition
        double w1, w2;
        int type;       // Call +1 or put -1

        TwoFactorOptionData()
        {
                // Initialise all data and the payoff
                // Use Topper's data as the default, Table 7.1
                r = 0.01;
                D1 = D2 = 0.0;
                sig1 = 0.1; sig2 = 0.2;
                rho = 0.5;
                K = 40.0;
                T = 0.5;
                w1 = 1.0;
                w2 = 20.0;

                type = -1;
                // Now create the payoff Strategy object
                BasketStrategy(double strike, double cp, double weight1, double w
                { K = strike; w = cp; w1 = weight1; w2 = weight2;}

                pay = BasketStrategy(K, type, w1, w2);

                // In general 1) we need a Factory object as we have done
                // in the one-dimensional case and 2) work with pointers to
                // base classes
        }
        double payoff(double x, double y) const
        {
                return pay.payoff(x, y);
        }
};
```

We hope that this example gives an idea of using the inheritance mechanism in quantitative finance, To be honest, inheritance has its uses but the generic programming model is also useful. We discuss this topic later, beginning with Chapter 10 where we introduce template programming in C++.

The source code for two-factor payoff classes is to be found on the CD.

8.9. Conclusions and summary

We have given a detailed discussion of using inheritance in applications. In particular, we showed how to derive a class from another, more general base class. We applied this knowledge to creating a hierarchy of option payoff functions. Finally, we have included several sections on the potential problems when using inheritance, why they occur and how they can be mitigated.

8.10. Exercises and projects

9. Run-Time Behaviour in C++

9.1. Introduction and objectives

In this chapter we introduce C++ functionality that allows us to query objects at run-time, for example:

- Determining the class than an object pointer "points to" (dynamic type checking)
- Casting an object pointer to an object pointer of another class (dynamic casting)
- Navigating in class hierarchies
- *Static casting*
- Modelling run-time errors with the exception handling mechanism

We have included the above functionality in this chapter for three main reasons. First, it is an essential part of the language and we would like to be as complete as possible. Second, we need to use this functionality in specific (and sometimes very esoteric) applications. In fact, you may need to use it because of a design flaw that you wish to have a workaround for. Thus, it is for this reason good to know the essentials of run-time behaviour in C++.

9.2. An introduction to reflection and self-aware objects

Most of the examples up until now dealt with creation of classes and the instantiation of these classes to form objects. Having created an object we can then call its member functions.

But we now want to view classes themselves as objects in the sense that we wish to find out certain things about their structural and behavioural properties, for example:

- What are the member data of a given class?
- What are the names (as strings) of the member functions of a class?
- Execute a member functions by “invoking” it.

It should be obvious that these questions have to do with a high level of abstraction. This feature is called *Reflection* and its added value is that it allows the programmer to query the characteristics of classes, objects and other software entities at run-time. In a sense we can create *self-aware objects* that know about their data an methods. We speak of *metadata*, or data that describes other data. A number of programming environments have extensive support for Reflection (for example “’Microsoft’s .NET framework’) and support for the following feautures is not unusual:

- Reading metadata of types, classes, interfaces, methods and other software entities.
- Dynamic object creation.
- Dynamic method invocation.
- Code generation.

C++ has some support for these kinds of features. In the following sections we discuss these specific features in detail.

In order to show how run-time functionality works we examine two model examples. The first example is generic and minimal and the second example discusses run-time behaviour associated with classes in a class hierarchy.

9.3. Run-time type information (RTTI)

In this section we begin our discussion of the run-time behaviour of objects. To this end, we introduce two features in C++:

- The `typeid()` operator.
- The `type_info` class.

First, by using `type_info` we can get the name of any object or built-in type in C++. Second, we use the `typeid()` operator on some object or built-in type to give a `const` reference to `type_info`. The interface functions in `type_info` are:

```
class type info {

public:
        virtual ~type info();
        int operator==(const type info& rhs) const;
        int operator≠(const type info& rhs) const;
        int before(const type info& rhs) const;
        const char* name() const;
        const char* raw name() const;
};
```

We see from this interface that we can:

- Compare two objects (operators == and ≠).
- Find the human-readable name of an object.

We do not discuss the other interface functions in this chapter as we think that they are too esoteric for most applications.

We must take note of the fact that this RTTI feature only works for polymorphic classes, that is classes with at least one virtual function. Furthermore, some compilers (for example, Microsoft C++ compiler) demand that you define certain compiler settings, otherwise a run-time error will occur. In the Microsoft enviroment, for example the RTTI option must be enabled. See the relevant on-line help for that enviroment.

We now take a simple generic example. To this end, we create a class hierarchy with two derived classes. The base class is defined as:

```
class Base
{
        public:
        Base() {}
        virtual ~Base() {}

        virtual void print() const { cout << "I'm base\ n"; }
        virtual double calculate(double d) const = 0;

};
```

and the derived classes are:

```
class D1: public Base
{

public:
        D1() {}
        virtual ~D1() {}
        virtual void print() const { cout << "I'm a D1\ n"; }
        virtual double calculate(double d) const { return 1.0 * d; }
};

class D2: public Base
{
public:
        D2() {}
        virtual ~D2() {}

        virtual void print() const { cout << "I'm a D2\ n"; }
        virtual double calculate(double d) const { return 2.0 * d; }
};
```

The first example creates instances of the derived classes and examines their run-time properties:

```
D1 d1, d11;
D2 d2;
// Define a reference to type
const type info& myRef = typeid(d1);

cout << "Human-readable name: " << myRef.name() << endl;

// Test if two objects have same type or not
if (typeid(d1) == typeid(d11))
{
        cout << "Types are the same\ n";
}

if (typeid(d1) != typeid(d2))
{
        cout << "Types are NOT the same\ n";
}
```

In this code we get the answers that we expect; first, the human-readable form of the class name is displayed on the console and second the classes corresponding to objects `d1`, `d11` and `d2` are compared. You can run this code to see how it works.

We now note that RTTI can be applied to base classes pointers. Let us take a simple example. As already discussed in this book, we can assign base class pointers to addresses of derived classes. What is of interest now is that RTTI knows that the dereferenced pointer is actually a derived class object:

```
Base* b = &d1;
```

```
const type info& myRef2 = typeid(* b);
cout << "Human-readable name: " << myRef2.name() << endl;
```

This is quite useful feature because in some applications we might have an array of base class pointers and we may wish to know what the "real" underlying types are. Here is an example in which we create an array to pointers to `D1` and `D2` instances. Then we iterate in the array to determine what the actual type is that is "hiding" behind the pointer:

```
// Create an array of Base class pointers
int size = 10;
Base* myArr[10]; // An array of pointers!

for (int j = 0; j < 6; j++)
{
        myArr[j] = &d1;
}

for (int k = 6; k < size; k++)
{
        myArr[k] = &d2;
}

// Now "filter" the real types. We have D1 and D2 types!
int counterD1 = 0;
int counterD2 = 0;

for (int i = 0; i < size; i++)
{
        if (typeid(* myArr[i]) == typeid(D1))
        {
                cout << "We have a D1\ n"; counterD1++;
        }

        if (typeid(* myArr[i]) == typeid(D2))
        {
                cout << "We have a D2\ n"; counterD2++;
        }
}
// Print final counts
cout << "Number of D1s: " << counterD1 << endl;
cout << "Number of D2s: " << counterD2 << endl;
```

We thus keep a count of the different types in the collection. This example could be modified to work in a financial engineering application. For example, let us suppose that we can have a class hierarchy of financial instruments with derived classes for options, bonds, futures and other specific derivatives products. We could then use RTTI to perform the following kinds of operations:

- Select all options in the portfolio.
- Select all derivatives except futures in the portfolio.
- Remove all futures from the portfolio.
- Update some member data in options in the portfolio.

In fact, we could create a simple query and manipulation language that allows us to access specific information in a portfolio. We could then use it in a pricing engine.

One final remark; when working with pointers to base classes, it is in general not possible to determine what the "real" object (of a derived class) without using the RTTI functionality.

9.4. Casting between types

Casting is the process of converting an object pointer of one class to an object pointer of another class. In general we must distinguish between *static casting* (for non-polymorphic classes) and *dynamic casting* (for polymorphic classes, that is classes with at least one virtual function). To this end, C++ has several operator that allow us to perform casting:

- `dynamic_cast operator`.
- `static_cast operator`.

The first casting operator takes two operands:

```
dynamic_cast<T*> (p)
```

In this case we attempt to cast the pointer `p` to one of class `T`, that is a pointer `T*`. We can use this operator when casting between base and derived class. An example is:

```
D1 d1A;
Base* base2 = &d1A;

D1* d1Cast = dynamic cast<D1* > (base2);

if (d1Cast == 0)
{
        cout << "Cast not possible:\ n";
        // Should ideally throw an exception here
}
else
```

```
{ // This function gets called
        cout << "Cast is possible: ";
        d1Cast -> print();
}
```

In this case we have a pointer that points to a `D1` instance and we can convert it to a pointer of the same class because the return type is not zero. We now give an example in which we attempt to convert `D1` pointer to a `D2` pointer and of course this is not possible because they are completely different classes:

```
// Now cast a D1 to a D2 (not possible)
D2* d2Cast = dynamic cast<D2* > (base2);

if (d2Cast == 0)
{ // This function gets called
        cout << "Cast not possible:\ n";
}
else
{
        cout << "Cast is possible:\ n";
        d2Cast -> print();
}
```

The above examples use `downcasting` because we are moving the base class pointer to one of a derived class. It does not always give desired results as we can see. *Upcasting*, on the other hand allows us to cast a derived class pointer to a base class pointer (this is a consequence of the fact that we are implementing generalization/specialization relationships and in this case the operation will be successful). The casting to be done can be to a direct or indirect base class:

```
D1* dd = new D1;
Base* b3 = dynamic cast<Base* > (dd);

if (b3 == 0)
{ // This function gets called

        cout << "Cast not possible:\ n";

}
else
{
        cout << "Cast is possible:\ n";
        b3 -> print();
        b3 -> doIt();
}
```

There is a small run-time cost associated with the use of the `dynamic_cast` operator. We now discuss static casting. We find it to be very esoteric and we do not use it at all. However, we include it for completeness. An example is:

```
// Static casting
Base* bA = &d1;
Base* bB = &d2;
D1* dA = static cast<D1*> (bA);

// Unsafe static cast
cout << "Unsafe cast ...\ n";
D1* dB = static cast<D1* > (bB);
dB → print();
```

In the last case we are casting a `D2` pointer to a `D1` pointer! We advise against the use of this specific feature. A possible workaround is to redesign the problem or implement it in such a way that it does not have to use casting.

9.4.1. More esoteric casting

We discuss two more operators for casting between objects:

- The `reinterpret_cast`.
- The `const_cast`.

The `reinterpret_cast` is the crudest and potentially nastiest of the type conversion operations. It delivers a value with the same bit pattern as its argument with the type required. It is used for implementation-dependent problems such as:

- Converting integer values to pointers.
- Converting pointers to integer values.

We give an example of this mechanism. Here we create an object and we assign it to a fixed memory location. Then we can convert the memory location by using `reinterpret_cast`:

```
D2 d2Any;
Base* bb = reinterpret cast<Base* >(&d2Any);
bb → print();
```

In this case we create a new object by giving it the same bit pattern as its argument. Finally, we now discuss a mechanism that allows us to convert a const pointer to a non-const pointer. Here is an example:

```
D1 dAny;
const Base* bConst = &dAny;
bConst → print();
```

```
// Base* bNonConst = bConst; DOES NOT WORK
Base* bNonConst = const cast<Base* > (bConst);
bNonConst -> print();
```

Please note that the following conversions on the `const` pointer will not work:

```
Base* bNonConst1 = static cast<Base* > (bConst);
Base* bNonConst2 = dynamic cast<Base* > (bConst);
Base* bNonConst3 = reinterpret cast<Base* > (bConst);
```

The compiler error message you get is something like:

```
d:\ users\ daniel\ books\ cppfeintro\ code\ chap14\ example1.cpp(193) :
error C2440: "static cast" : cannot convert from "const class Base * "
to "class Base * "
```

9.5. Client-server programming and exception handling

In the following sections we introduce the reader to client-server programming in C++ and how this relates to the problem of run-time exceptions and errors that arise in a running program.

We first discuss some things that can go wrong when clients call functions. In order to reduce the scope, let us examine a simple example of find the sum of reciprocals of a vector:

```
template <class V> V sumReciprocals(const vector<V>& array)
{ // Sum of reciprocals

        V ans = V(0.0);
        for (int j = 0; j < array.size(); j++)
        {
                ans += 1.0/array[j];
        }
        return ans;
}
```

We see that if any one of the elements in the vector is zero we shall get a run-time error and your program will crash. To circumvent this problem we use the exception handling mechanism in C++ to provide the client of the software with a “net” as it were which will save the client if it encounters an exception. For example, consider the following code:

```
int size = 10;
double elementsValue = 3.1415;
vector<double> myArray (size, elementsValue);
myArray[5] = 0.0;
```

```
double answer = sumReciprocals(myArray);
```

In this case only one element in the vector has a zero value but the end-result will not be correct. In fact, we get a value that is not a number, for example *overflow*. Another example is when we change a value in a vector based on a given index. A problem is when you enter an index value is outside the range of the vector. The consequence in this case is that the value will not be changed at best and that some other part of the memory will be overwritten at worst. Consider the following code:

```
cout << "Which index change value to new value? ";
int index;
cin >> index;
myArray[index] = 2.0;
print(myArray);
```

In both of the above cases we could argue that the client should provide good input values but this line of thought is too naive. Instead, we provide both client and server with tools to resolve the problem themselves. We now discuss the fine details of this mechanism.

9.6. `try`, `throw` and `catch`: ingredients of the C++ exception mechanism

9.7. C++ implementation

```
class MathErr
{ // Base class for my current exceptions
private:
        string mess;    // The error message
        string meth;    // The method that threw the exception
public:
        MathErr()
        {
                mess = meth = string();
        }
        MathErr (const string& message, const string& method)
        {
                mess = message;
                meth = method;
        }
        string Message() const { return mess; }
        string Method() const { return meth; }
        virtual vector<string> MessageDump() const = 0;

        virtual void print() const
        {
                // This uses a Template method pattern

                // Variant part
                vector<string> r = MessageDump();

                // Invariant part
                for (int j = 0; j < r.size(); j++)
                {
                        cout << r[j] << endl;
                }
        }
};
```

and the interface for zero divide exceptions is given by:

```
class ZeroDivide : public MathErr
{
private:
        string mess;    // Extra information
public:
        ZeroDivide() : MathErr()
        {
                mess = string();
        }
        ZeroDivide(const string& message,const string& method, const string& anno
        {
                mess = annotation;
        }

        vector<string> MessageDump() const
        { // Full message
                vector<string> result(3);
                result[0] = Message();
                result[1] = Method();
                result[2] = mess;
                return result;
        }
};
```

We wish to use these exceptions in server code. In this case we extend the function `simReciprocals` so that it checks for zero divide:

```
template <class V> V sumReciprocals(const vector<V>& array)
{ // Sum of reciprocals

        V ans = V(0.0);
        for (int j = 0; j < array.size(); j++)
        {
                if (fabs(array[j] < 0.001)) // Magic number!
                {
                        throw ZeroDivide("divide by zero", "sumReciprocals", stri
                }
                ans += 1.0/array[j];
        }

        return ans;
}
```

Here we see that the code checks for zero divide conditions and if such an even occurs it will create an exception object of the appropiate type and throw it back to the client. It is then the client's responsibility to catch the exception and then determine what to do with it. Our `try/catch` block is:

```
int size = 10;
double elementsValue = 3.1415
vector<double> myArray (size, elementsValue);
myArray[5] = 0.0;
print(myArray);

// Now creates a try/catch block
try
{
        double answer = sumReciprocals(myArray);
        cout << "Sum of reciprocals: " << answer << endl;
}
catch (ZeroDivide& exception)
{
        exception.print();
}
```

In this case the exception will be "catched" in the *catch block* and the program continues. Incidentally, if a server code throws an exception and no try/catch block has been defined in the cliente clode a run-time error will occur and the program will terminate abnormally.

We now investigate the case where the client can decide to repair the situation by giving itself the chance to give a new value for the offending value (namely `myArray[5] = 0.0`) or allowing it to exit the try block. To this end, the following code allows the client to give a new value and infinitum or it can exit giving the magic number `999`:

```
Lab1: try
{
        cout << "\ nGive a new value for index number 5:";
        double val;
        cin >> val;
        myArray[5] = val;
        if (val == 999.0)
        {
                return 1; // Exit the program
        }

        double answer = sumReciprocals(myArray);
        cout << "Sum of reciprocals: " << answer << endl;
}
catch (ZeroDivide& exception)
{
        exception.print();
        goto Lab1;
}
```

This completes our first example of the exception handling mechanism in C++. In later chapters we shall use the mechanism in financial engineering applications and test cases. Summarising, we have introduced the essentials of exception handling in C++ and we have shown how to use it. The knowledge can be applied in your applications.

We have not discussed the following topics:

- The throw specification (a function can declare the exceptions that it throws to the outside world).
- Nested try blocks and rethrow of exceptions.
- Alternative ways of implementing exception handling.
- Exceptions with the constructor and destructors.
- Exceptions and efficiency.

9.8. Pragmatic exception mechanisms

Instead of creating a hierarchy of exceptions classes (which involves having to create and maintain these) we take a somewhat more pragmatic approach by creating a single class that we use in all applications. Typical information could be:

- The error message.
- The method that threw the exception
- More informational message

All this information is implemented as strings. The complete interface is given by:

```
#ifndef DatasimExceptions HPP
#define DatasimExceptions HPP
#include <string>
#include <vector>
using namespace std;
#include <iostream>
class DatasimException
{ // Base class for my current exceptions
private:
        string mess;    // The error message
        string meth;    // The method that threw the exception
        string why;     // More info on message

        // Redundant data
        vector<string> result;

public:
        DatasimException();

        DatasimException (const string& message, const string& method, const stri

        string Message() const; //      The message itself
        string rationale() const;       // Extra information
        string Method() const;  // In which method?

        // All information in one packet
        vector<string> MessageDump() const;

        // Print the full packet
        virtual void print() const;
};
#endif
```

The code file is:

```
#include "DatasimException.hpp"
```

```
DatasimException::DatasimException()
{
        mess = meth = why = string();
        result = vector<string>(3);
        result[0] = mess;
        result[1] = meth;
        result[2] = why;
}

DatasimException::DatasimException (const string& message, const string& method,
{
        mess = message;
        meth = method;
        why = extraInfo;
        result = vector<string>(3);
        result[0] = mess;
        result[1] = meth;
        result[2] = why;
}
string DatasimException::Message() const
{
        return mess;
}
string DatasimException::rationale() const
{
        return why;
}

string DatasimException::Method() const
{
        return meth;
}

vector<string> DatasimException::MessageDump() const
{ // Full message
        return result;
}

void DatasimException::print() const
{
        // Variant part
        vector<string> r = MessageDump();
        cout << endl << r[0] << endl;
        // Invariant part
        for (int j = 1; j < r.size(); j++)
        {
                cout << "\ t" << r[j] << endl;
        }
}
```

Please not the use of the Standard Template Library (STL) vector data container. This is forward reference and we shall discuss it in a later chapter.

At some stage we could employ a single instance of the class `DatasimException` so that performance is improved because we create this object at program start and it is removed from memory when the program ends. In this sense, we can apply the *Singleton* pattern to this problem. In fact, the Singleton is reminiscent of the `Err` object in early versions of the Visual Basic programming language.

9.8.1. An example of use

We give an example. In this case we take the trivial problem of dividing a number `x` by another number `y`. The answer is undefined if `y` is zero and in this case a run-time error will occur if no provisions are taken for dealing with the problem. Thus, we use exception handling as follows:

Listing 9.1: A

```
double Divide(double x, double y)
{
                // Precondition: y is not zero
                if (y == 0.0)
                {

                        throw DatasimException(std::string("\tDivide by zero"), s
                                std::string("Try with non-zero value"));
                }

                return x/y;
}
```

This is the server code. The client code resides in a main program:

Listing 9.2: B

```

int main()
{
```

```
/*
        try
        {
```

If we run this program and enter `0.0` for `y` we shall get the following output:

```
Give a number to divide by: 0.0
Divide by zero
In function Divide
Try with non-zero value
```

In later chapters we shall encounter more applications of this exception class when we discuss a visualization package in Excel.

9.9. Conclusions and summary

We have given an introduction to a number of advanced and somewhat esoteric techniques that allow the programmer to investigate run-time behaviour in C++ programs. In particular, we discussed:

- Determining class of an object at run-time.
- Casting pointers.
- Designing and implementing exceptions in C++.

We have included these topics for completeness. We shall use them here and there in future chapters.

9.10. Exercises and research

10. An Introduction to C++ Templates

10.1. Introduction and objectives

C++ was in the first mainstream object-oriented language to support the *generic programming paradigm*. In general terms this means that we can create classes that do not use a concrete underlying data type for their representation but rather use an unspecified data type that will later be replaced by a concrete type. The advantage of *generic* *classes* is that we only have to write them once and they can subsequently be used again and again in many contexts. For example, a common data structure in financial engineering is one representing the data in the binomial method. We know that the binomial method has two dimensions, namely time t and asset price S. When investigating this lattice we may wish to keep the following components as flexible as possible:

- The index set (call it `I`) that describes the discrete time values.
- The data type (call it `V`) that represents the values at the vertex points of the binomial mesh structure.

In a sense, we would like to write classes in a kind of macro language, using the data structures `I` and `V` and then replacing these by concrete types in our application. This feature is possible in C++ and is called C++ *template mechanism*. Going back to above example we would declare the lattice class as depending on two generic parameters as follows:

```
template <class I, class V> class Lattice
{

// private and public code here
// Constructors, building a lattice, navigating
};
```

Different users will have their own specific data types for the data structures `I` and `V`. For example, a common example in the literature is when we work with integers for the values in the time direction and `double` to hold the values at the nodes:

```
Lattice<int, double> myBinomial( ... );
```

In chapter 15, we shall use this generic data container as a storage medium for the binomial method. We shall also use it in chapter 19 when we discuss the trinomial method.

Some applications may demand a more complex structure, for example where the index set is a date and the node values is an array (containing various parameters such as jump probabilities, prices and sensitivities:)

```
Lattice<DatasimDate, Array> myCallableBond( ... );
```

Thus, we only have to write the generic code once and then instantiate it for many different kinds of concrete data types, as the two examples above show.

In this chapter we pay attention to the process of creating simple *template classes* and *template functions* as we think that this is one of the best ways to learn generic programming in C++. We learn by doing and by example. It is important to understand the corresponding syntax well because the compiler errors can be quite cryptic and difficult to understand. In most cases these errors are caused by typing errors or other silly mistakes. This may be the reason why generic programming techniques are not as popular as the more established object-oriented paradigm.

After having read and understood this chapter you will have crossed one hurdle, namely mastering the essential syntax of C++ templates.

We focus on syntax in this chapter. A good understanding of templates is a precondition for further progress.

10.2. My first template class

In general it is our goal to deal with each topic in this chapter as concisely as possible. To this end, we introduce the template mechanism in C++ by first giving a representative example.

Templates in C++ are classes or functions that contain members or have parameters whose types are not yet specified but are generic. For example, we could create a class that represents complex numbers are `double` precision numbers we now would like to create a more generic complex number class in which the real and imaginary parts can take on any types.

We now discuss the actual process of creating a template class. Our first remark is that creating template classes is not much more difficult than creating non-template classes.

The main difference is that the template class depends on one or more generic parameters (generic type) and this has to be made known to the compiler. To this end, we need to introduce the keyword `template` in combination with the names of the generic types. We shall create a generic class that represents a one-dimensional interval or range. The header file for this class declares the class, its member data and member function prototypes:

```
template <class Type = double> class Range
{       // Default underlyng type is double

private:
        Type lo;
        Type hi;
        Range();        // Default constructor

public:
        // Constructos
        Range(const Type& low, const Type& high);       // Low and high value
        Range(const Range<Type>& ran2); // Copy constructor

        // Destructor
        virtual ~Range();

        // Modifier functions
        void low(const Type& t1);       // Sets low value of current range
        void high(const Type& t2);      // Sets high value of current range
        // Accessing functions
        Type low() const;       Lowest value in range
        Type High() const;      // Highest value in the range

        Type length() const;    High - Low value

        // Boolean functions
        bool left(const Type& value) const;     // Is value to the left?
        bool right(const Type& value) const;    // Is value to the right?
        bool contains(const Type& value) const; // Range contains value?

        // Operator overloading
        Range<Type>& operator = (const Range<Type>& ran2);
};
```

Thus, this class models one-dimensional intervals and it has the appropiate functionality for such entities, for example constructors, set-like operations and set/get functions. In this declaration we see that `Type` is generic (and not real class). Furthermore, the word `Range` is not a class because, being a template we must use it in combination with the underlying type `Type`. For example, the copy constructor must be declared as:

```
Range(const Range<Type>& ran2); // Copy constructor
```

We now must know how to implement the above member functions. The syntax is similar to the non-template-case; however, there are two issues that we must pay attention to:

- The `template` keyword must be used with each function.
- The class name is `Range<Type>`

The compiler needs this information because the generic type will be replaced by a real type in our applications, as we shall see.

We give some examples. The following code represents some typical examples:

```
template <class Type> Range <Type>::Range()
{
        // Default constructor.

                // Not defined since privare
}

template<class Type> Range<Type>::Range(const Type& l, const Type& h)
{//
        if (l < h)
        {
                lo = l;
                hi = h;
        }
        else
        {
                hi = l;
                lo = h;
        }
}

template <class Type> bool Range<Type>::contains (const Type& t) const
{// Does range contain t

        if((lo <= t) && (hi >=t))
                return true;
        return false;
}
```

Please note that constructors are called `Range` and not `Range<Type>`. This can be confusing but the constructors are just member functions.

10.3. Using template classes

A template class is a *type*. In contrast to non-templated classes, it has no instances as such. Instead, when you wish to use template class in a program you need to *instantiate the template class*, that is we replace the generic underlying type (or types) by some concrete type. The result is a "normal" C++ class. For example, we take the example of a range whose underlying data type is a date. This class is useful in many financial engeering aplications, for example when defining cash flow dates in fixed-income modeling applications:

```
DatasimDate now;
DatasimDate nextYear = now.add_years(1);
Range<DatasimDate> dataSchedule(now, nextYear);
```

This is very useful because we can now reuse all the generic code to work with dates. Some examples are:

```
DatasimDate datL = now - 1;     // yesterday
DatasimDate datM = now.add_halfyear();
DatasimDate datR = nextYear + 1;        // One year and a day from now

if (dataSchedule.left(datL) == true)
        cout << datL << "to left, OK\n";
```

```
if (dataSchedule.left(datM) == true)
        cout << datM << "in interval, OK\n";

if (dataSchedule.right(datR) == true)
        cout << datR << "to right, OK\n";
```

To take another example, we can instantiate the template class so that it works with integers, as the following code shows:

```
Range<int> range1 (0, 10);

int valL = -1;
int valM = 5;
int valR = 20;

if (range1.left(valL) == true)
        cout << valL << " to left, OK\ n";

if (range1.contains(valM) == true)
        cout << valM << " in interval, OK\ n";

if (range1.right(valR) == true)
        cout << valR << " to right, OK\ n";
```

Again, we did not have write any extra code because we instantiated the template class again. On the other hand, the template class assumes that the underlying data type implements a number of functions and operators (for example, <, >, ⩽). You must realise this fact. We have used the following syntax in the declaration of the range class:

```
template <class Type = double> class Range
```

This means that when you instantiate the template class without a parameter the underlying type will be `double`! This save you typing the data type each time (which can become quite tedious, especially then working with multiple data types). To this end, we can define synonyms by using `typedef`:

```
ttypedef Range<> DoubleRange;
DoubleRange d1(-1.0, 1.0);
print(d1);
```

When working with dates we can proceed as follows:

```
typedef Range<DatasimDate> DateRange;
DatasimDate anotherDate = today + 365;
DateRange whatsAnotherYear(today, anotherDate);
print(whatsAnotherYear);
```

It is advisable to define all your synonyms in one place so that you can easily locate them later.

10.3.1. (Massive) reusability of the range template class

Let us pause for a moment to think about what we have done. We have created a template class that can be used in many kinds of applications. In the current book, there are many situations where we can use and reuse the range template class:

- Binomial, trinomial and finite difference methods
- Interval arithmetic and solution of linear and non-linear equations: interval arithmetic is based on achieving two-sided estimates for some (unknown) quantity.
- Range as a building block in larger data structures as we shall see in Chapters 16 and 17.

On the down side, if you are not satisfied with the functionality that a template class delivers you can always extend the functionality using any one of the three mechanisms:

- Inheritance
- Composition and delegation
- Defining non-member function that accept an instance of a template class

10.4. Template functions

In the previous section we have discussed how to create and instantiate a template class. It is perhaps worth mentioning that is possible to create non-member (or C-style) template functions. This is useful feature for several reasons. It is sometimes cumbersome or even wrong to be forced to embed functionality as a member function of some artificial class and second we wish to extend the functionality of a class without having to inherit from it. Let us take a simple example to show how to define a tempate function. To this end, we wish to write a function to swap two instances of the same class (incidentally , the Standard Template Library (STL) has a function to swap any two objects). The code for the swap function is given by:

```
template <class V> void mySwap(V& v1, V& v2)
{
        V tmp = v2;
        v2 = v1;
        v1 = tmp;
}
```

The code is very easy to understand. Of course, we see that the underlying type V should have a public assignment operator defined. If it is private, for example the code will not compile when we instantiate V with some concrete data type. Some examples of using the swap functions are:

```
int j = 10;
int k = 20;
mySwap(j, k);

Range<int> range1 (0, 10);
Range<int> range2 (20, 30);

mySwap (range1, range2);
```

We can write many kinds of related template functions and then group them into a namespace. This is the topic of the next section.

10.5. Consolidation: understanding templates

It takes some time to master templates in C++ and most compiler errors are caused by invalid syntax, typing errors and forgetting to supply all the syntax that is needed. To make matters worse, the compiler can become confused and it then produces error messages that can be very difficult to understand.

In this and the following sections we give a complete example of a template class that you should read and understand. It models a point in two dimensions whose coordinates may belong to different data types. There are a number of issues to take care of:

- Declare that the class is a template class.
- Define the class' member data.
- Define the class constructors and destructor.
- Define the other member and non-member functions

The full interface is now given. Please study it:

Listing 10.1: Generic point class

```
// Point.cpp
//
// Generic point class.
//
// Last modification dates:
//
// DD 2006-3-5 Kick off and << for print
```

```
//
// (C) Datasim Education BV 2006

#ifndef Point_cpp
#define Point_cpp

#include "Point.hpp"
#include <cmath>


// Default constructor
template <class TFirst, class TSecond>
Point<TFirst, TSecond>::Point()
{
        m_first=TFirst();
        m_second=TSecond();
}


// Constructor with coordinates
template <class TFirst, class TSecond>
Point<TFirst, TSecond>::Point(TFirst first, TSecond second)
{
        m_first=first;
        m_second=second;
}

// Copy constructor
template <class TFirst, class TSecond>
Point<TFirst, TSecond>::Point(const Point<TFirst, TSecond>& source)
{
        m_first=source.m_first;
        m_second=source.m_second;
}

// Destructortemplate <class TFirst, class TSecond>
template <class TFirst, class TSecond>
Point<TFirst, TSecond>::~Point()
{
}

// Get first coordinates
template <class TFirst, class TSecond>
```

```
TFirst Point<TFirst, TSecond>::First() const
{
        return m_first;
}

// Get second coordinate
template <class TFirst, class TSecond>
TSecond Point<TFirst, TSecond>::Second() const
{
        return m_second;
}

// Set first coordinate
template <class TFirst, class TSecond>
void Point<TFirst, TSecond>::First(const TFirst& val)
{
        m_first=val;
}

// Set second coordinates
template <class TFirst, class TSecond>
void Point<TFirst, TSecond>::Second(const TSecond& val)
{
        m_second=val;
}

// Calculate distance
template <class TFirst, class TSecond>
double Point<TFirst, TSecond>::Distance(const Point<TFirst, TSecond>& p) const
{
        // Get the length of the sides
        TFirst a=p.m_first-m_first;
        TSecond b=p.m_second-m_second;

        // Use Pythagoras to calculate distance
        return std::sqrt(a*a + b*b);
}

// Assignment operator
template <class TFirst, class TSecond>
Point<TFirst, TSecond>& Point<TFirst, TSecond>::operator = (const Point<TFirst, T
{
        if (this == &source)
```

```
                return *this;

        m_first=source.m_first;
        m_second=source.m_second;

        return *this;
}

// Print
template <class TFirst, class TSecond>
        std::ostream& operator << (std::ostream& os, const Point<TFirst, TSecond>
{
        os << "(" << p.m_first << "," << p.m_second << ")\n";
        return os;
}

#endif
```

10.5.1. A test program

We now show how to use the above template class in an application. First, it is mandatory to include the file that contains the source code of the member functions (int this case, `Point.cpp`) include the file that contains the source code of the member functions (in this case, `Point.cpp`) in client code. Any other construction will not work. Then we replace the generic data types by specific data types (this is called the *instantiation process*). To this end, consider the following code:

Listing 10.2: Testing generic points

```
// TestGenericPoint.cpp
//
// Testing generic points
//
// (C) Datasim Education BV 2006
//

// The file that actually contains the code must be included
#include "Point.cpp"

int main()
{

        Point<double, double> p1(1.0, 1.0);
        Point<double, double> p2;                       // Default point
        std::cout << "p1: " << p1 << std::endl;
        std::cout << "p2: " << p2 << std::endl;

        std::cout << "Distance between points: " << p1.Distance(p2) <<
std::endl;

        return 0;
}
```

In this case we create points whose underlying types are double.

10.6. Summary and conclusions

This chapter introduced the notion of generic programming as a programming paradigm and in particular we showed – by examples – how it is realised in C++ by the template mechanism.

We focused on the syntax that we must learn if we are going to be successful in using template classes and functions in Quantitative Finance applications.

A template class is not much different from a non-template class except that we explicitly state that certain members are generic. Second, all functions that use the generic data types must that it is using these types, for example:

```
template <class TFirst, class TSecond> Point<TFirst, TSecond>::Point()
{
        m_first = TFirst();
        m_second = TSecond();
}
```

This chapter lays the foundation for the rest of this book in the sense that many of the specific instantiated classes and resulting applications make use of template classes and functions.

A good way to learn template programming is to take a working non-template class and then port it to a templated equivalent. A good example is the C++ class that models complex numbers. We have already discussed this class in Chapter 5.

10.7. Exercises and projects

Part II.

Data structures, templates and patterns

11. Introduction to Generic Data Structures and Standard Template Library (STL)

11.1. Introduction and objectives

11.2. Complexity analysis

11.2.1. Examples of complexities

11.3. An introduction to data structures

11.3.1. Lists

11.3.2. Stacks and queues

11.3.3. Sets and multisets

11.3.4. Maps and multimaps

11.4. Algorithms

11.5. Navigation in data structures: iterators in STL

11.6. STL by example: my first example

11.6.1. Constructors and memory allocation

11.6.2. Iterators

11.6.3. Algorithms

11.7. Conclusions and summary

11.8. Exercises and projects

12. Creating Simpler Interfaces to STL for QF Applications

12.1. Introduction and objectives

12.2. Maps and dictionaries

12.2.1. Iterating in maps

12.2.2. Erasing records in a map

12.3. Applications of maps

12.4. User-friendly sets

12.4.1. STL sets

12.4.2. User-defined and wrapper classes for STL sets

12.5. Associative arrays and associative matrices

12.6. Applications of associative data structures

12.7. Conclusions and summary

12.8. Exercises and projects

13. Data Structures for Financial Engineering Applications

13.1. Introduction and objectives

13.2. The property pattern

13.3. Property sets

13.3.1. Basic functionality

13.3.2. Addition and removal of properties

13.3.3. Navigating in a property set: creating your own iterators

13.3.4. Property sets with heterogeneous data types

13.4. Property sets and data modelling for quantitative finance

13.4.1. Fixed assemblies (assemblies-parts relationship)

13.4.2. Collection-members relationship

13.4.3. Container-contents relationship

13.4.4. Recursive and nested property sets

13.5. Lattice structures

13.5.1. Some simple examples

13.5.2. 13.5.2 A simple binomial method solver

13.6. Conclusions and summary

13.7. Exercises and projects

14. An Introduction to Design Patterns

14.1. Introduction and objectives

14.2. The software lifecycle

14.3. Documentation issues

14.3.1. Generalisation and specialisation

14.3.2. Aggregation and composition

14.3.3. Associations

14.3.4. Other diagrams

14.4. An Introduction to design patterns

14.4.1. Creational patterns

14.4.2. Structural patterns

14.4.3. Behavioural patterns

14.5. Are we using the wrong design? Choosing the appropriate pattern

14.6. CADObject, a C++ library for computer graphics

14.7. Using patterns in CADObject

14.8. Conclusions and summary

14.9. Exercises and projects

Part III.

QF Applications

15. Programming the Binomial Method in C++

15.1. Introduction and objectives

15.2. Scoping the problem

15.3. A short overview of the binomial method

15.4. Software requirements for a binomial solver

15.5. Class design and class structure

15.5.1. UML class structure

15.5.2. Information flow

15.6. Applying design patterns

15.7. The builder and director classes

15.8. The process and the steps

15.9. Test cases and examples

15.10. Conclusions and summary

15.11. Exercises and questions

16. Implementing One-Factor Black Scholes in C++

16.1. Introduction and objectives

16.2. Scope and assumptions

16.3. Assembling the C++ building blocks

16.4. Modelling the black scholes PDE

16.4.1. Creating your own one-factor financial models

16.5. Finite difference schemes

16.5.1. Explicit schemes

16.5.2. Implicit schemes

16.5.3. A note on exception handling

16.5.4. Other schemes

16.6. Test cases and presentation in excel

16.6.1. Creating the user interface dialogue

16.6.2. Vector and matrix output

16.6.3. Presentation

16.7. Summary

16.8. Exercises and projects