CHAPTER 1

Potential Profit as a Measure of Market Performance

The goal of trading is to make money, and for many, profits are the best way to measure that success. It is one of the most important performance characteristics of trading. In this chapter, I would like to emphasize that in contrast with ordinary profit, potential or maximum profit—the central subject of this book—does not deal at all with the activity of an individual trader. Potential profit and the strategy producing it are market properties. Along with this, I will write a C++ program computing Pardo's potential profit.

PROFIT AND POTENTIAL PROFIT

What does a profit tell us? Is it a characteristic of the trader's skills? To some extent yes, but that is not all. The profit is a result of interaction of the human with the market. It characterizes the trader as well as the existing market conditions.

If we apply a mechanical trading system to several historical intervals of market data and get the average annualized return on investment, what does this value mean? Is it a system characteristic? In many respects yes, but not exactly. This value is a measure of both system and market performances.

If a developer says that his system produced a 60 percent return on investment, does it mean that the system is good? To make the hidden sense obvious, I will reformulate the question. Can we expect a 60 percent return on investment if we apply the same system to a flat market? One can argue that such markets luckily do not exist and that prices always fluctuate. This is not the point. We can find historical periods of very low price volatility and trend, where it is unreasonable to believe that 60 percent could be achieved. However, if one says that he made a 100 percent return on margin trading a *soybean futures contract* in the first quarter of 2005 (see Figure 1.1), should we conclude that the return is good?

FIGURE 1.1 Open, high, low, close prices for soybean contract SK05 expired in May 2005 and traded on the Chicago Board of Trade (CBOT) during January–March of 2005.

Source: Courtesy of XPRESSTRADE, www.xpresstrade.com.

One way to judge traders' performance is to compare it with results achieved by others. For our purposes the best example is Larry Williams, a well-known trader and writer of several best-selling books (Williams 1979, 1999, 2000, 2005), who has been documented as having demonstrated extraordinary performance participating in the Robbins Trading Company World Championship in 1987. Starting with $10,000, he increased the account value up to over $1 million in one year. This result remains the competition's record at the time of this writing, and it is certainly an extraordinary return for one year. Ten years later, in 1997, his daughter ended the year with more than $100,000, beginning with the same $10,000. It is interesting to compare the impressive results (see Table 1.1) shown in different years by other winners of this championship.

You would think that a return of 100 percent on margin in three months of trading soybeans would be considered a good return. But how much did the market offer during those three months? Although the returns in Table 1.1 show only the best of all participants, everyone should agree that potentially bigger or substantially bigger profits were possible in the markets during the time of the competition. Moreover, we understand that Larry Williams, in achieving his 1987 result, had his account equity exceed $2 millions before giving back part of

TABLE 1.1 Robbins World Cup Championships of Futures Trading—Top Overall Performances for All Divisions

Year	Winner	Return (%)
2004	Kurt Sakaeda	929
2003	Int'l. Capital Mngt.	88
2002	John Holsinger	608
2001	David Cash	53
2000	Kurt Sakaeda	595
1999	Chuck Hughes	315
1998	Jason Park	99
1997	Michelle Williams	1,000
1996	Reinhart Rentsch	95
1995	Dennis Minogue	219
1994	Frank Suler	85
1993	Richard Hedreen	173
1992	Mike Lundgren	212
1991	Thomas Kobara	200
1990	Mike Lundgren	244
1989	Mike Lundgren	176
1988	David Kline	148
1987	Larry Williams	11,376
1986	Henry Thayer	231
1985	Ralph Casazzone	1,283
1984	Ralph Casazzone	264

Source: Robbins Trading Company (http://robbinstrading.com/worldcup/standings.asp)

those profits to the market, of course, in the hope of getting even more. How can we know what the potential would have been? While I have no exact records of which futures contracts were traded by participants of the World Cup, we shall get an idea about potential profits by analyzing *daily prices* and *intraday tick prices* in later chapters. Meanwhile, these observations and formulated questions distinguish the actual profit obtained by a trader or a system from the potential profit that could be realized in the same market during the same time. The former deals with both trading activity and market behavior, and the last is a property of a market during any given time interval. This market property can be referred to as *potential profit*, *maximum profit*, *market profit*, or *market offer*. Therefore, we can conclude the following:

- If a market does not offer a profit, then there is no trader or system that can create profits in that market.
- If a market does offer a profit, then there is no trader or system that can create a bigger profit than one offered by the market. From this point of view, the market never can be beaten. In the best case, a trader can play a draw game with the market!

Robert Pardo (Pardo 1992) suggested dividing profit by potential profit and using the ratio as a new measure of *model performance*:

> *An excellent measure of model performance is the efficiency with which the trading model converts potential profits offered by the market into trading profits. This measure is simple to calculate: Divide the net trading profit by the potential market profit. . . . The model efficiency measure makes it easy to compare market-to-market performance and to evaluate model performance on a year-to-year basis.*

We shall see that it is easy to calculate potential profit under conditions where transaction costs are not involved. However, introducing even simple commissions makes things substantially more complicated. This and other transaction costs lead to algorithms and indicators described in the following chapters.

PRICE FLOW AND C++

Why C++?

The purpose of this book is not only to introduce solutions for the calculation of potential profit under different conditions but also to compute those values from real prices. To accomplish this, I need a programming language for writing corresponding programs, and a good candidate seems to be C++ (Stroustrup 2000). It has excellent capabilities to express concepts in terms of *classes* and supports several programming paradigms, including *procedural programming*, *programming with abstract data types*, *generic programming*, and *object-oriented programming* (Stroustrup 2000, Booch 1994). Conveniently, there are several different C++ compilers commercially and publicly available. The modern C++ Standard Library (International Standard ISO/IEC 14882 2003) and Standard Template Library (STL), which is a part of it (Musser 1996) contain rich data collections and algorithms that can serve our purpose very well and simplify design and coding. Over the next few chapters, I shall gradually introduce the necessary notions and related C++ representations. In this chapter, we need to work with a sequence of prices that we will call *price flow*.

Why Skip Date and Time Classes?

Market prices come sequentially in time and are referred to as a *time series*. The most detailed information is called *tick data*. Every new transaction on an exchange is a discovery process that identifies the traded price and makes it known to the public. The time of each transaction is registered. Each time-price pair becomes a single point on an intraday price chart. In active or liquid markets, the time interval between two transactions can be just a fraction of a second. This is why in order to keep accurate records of this information and write corresponding software, one needs a class representing and measuring time with a precision of at least one second.

If we work with intraday prices and combine them across consecutive days into a single stream, a developer would need a date class. Alternatively, one could develop a class that combined both time and date computations. Such an aggregate would serve the use of intraday as well as daily price flows. A most common example of *daily price information* is a set of open, high, low, closing, and/or settlement prices. For trading either futures or equities, a daily record may also contain trading volume. For futures contracts, most analysts also include open interest. The calculation of annualized profit or return, the times and dates of the investment activity, and a sequence of realized profits and losses are crucial. In order to satisfy the complex requirements and conventions of the variety of fixed income and other investment instruments, and to compute the present values of cash flows and discount factors, a software library must have date, calendar, day fraction, and time classes. By contrast, for only profit computations, based wholly on prices, the knowledge of time and date intervals is not critical. Then, for our purposes, we can simplify the program by skipping date and time classes in this book.

Vector for Price Flow

Although date and time classes will be omitted, we still need to pay attention to the fact that a price flow is a sequence of prices. Ignoring time difference between elements of the sequence does not eliminate the need to reference and access each of them. This sequence can be expressed and implemented as a *sequence container*. In STL such sequence containers are *deque*, *list*, *queue*, *stack*, and *vector* (International Standard ISO/IEC 14882 2003). The last, *vector*, is very useful for our application. Because it is a template, the class `vector` may contain elements of different built-in types or classes. It automatically and efficiently handles memory management when objects are added to the collection or removed from it. The class `vector` gives multiple advantages compared to the C++ built-in arrays. It helps the writing of programs without *memory leaks* caused by a failure explicitly to release a dynamically allocated memory consumed by objects and makes development pleasant.

Classes for Prices

One way to program prices is to use the C++ built-in type `double`. If this low-order level type is used to express the notion of price, then it is up to us to make sure that wrong values do not find their way into our program. Such wrong values can be zero and/or negative numbers and those that are not whole multiples of minimal price increments—*ticks*. It is easier to organize these two verification tasks in a class or a *framework* (a collection of classes providing a set of services for a particular domain [Booch 1994]) encapsulating the built-in type `double`. In describing the evolution of prices by differential stochastic equations, one of the goals of modern theoretical approaches is to move from discrete cases to continuous functions or processes as soon as possible (Hunt 2000). In contrast with this tendency, I will emphasize the discrete properties of prices and transactions.

Prices either do not change or change by increments of minimum ticks. Each market has its conventions. For instance, the minimal nonzero price fluctuation of a soybean futures contract, traded on the Chicago Board of Trade (CBOT), is one quarter of a cent per bushel. When one sees soybean prices in Figure 1.1 or in an issue of the *Wall Street Journal* as 661.75, it is

read as 6 dollars 61 cents and three quarters of a cent per bushel. For accounting matters one does not need to know that a soybean contract assumes that a trading unit is 5,000 bushels. The only important information is that when the price changes by one tick up or down, the value of a bushel changes by +0.25 or –0.25 cents, the futures contract value changes by $12.50, higher or lower, respectively, and the value of the account gains or loses, depending on whether a single contract was bought or sold. If you bought and the price increased or sold short and the price decreased, then you've gained; otherwise, you've lost. Clearly, all these numbers relate each to other: the value 12.5 [dollars] is equal to 0.25 [cents per bushel] $\times$ 5000 [bushels]/100 [cents per dollar]. There are days when a soybean contract, which is traded on the CBOT from 9:30 A.M. to 1:15 P.M., can range from up 20 points to down 20 points (each point is 1 cent per bushel). A net rise or fall of 20 cents per bushel results in a profit or loss of $1,000 per contract without commissions and other fees.

C++ provides convenient and helpful tools to encapsulate details of price checking while hiding the internal state of an object of a price class. In the *object model* (a collective name of elements of a sound engineering foundation), which serves as a basis for modern programming, the main *object characteristics* are *identity*, *state*, and *behavior* (Booch 1994). In C++, the *state of an object* as a concept denotes collectively all values of the class members and other objects referred to by class members. It is a software design goal that an object is maintained in a well-defined state. The property making the state of an object well defined is known as *invariant* (Stroustrup 2000). A useful invariant for our price object means maintaining positive price values represented by a whole number of minimal price ticks. In particular, a *constructor* of a price class can create and initialize an object, where invariant holds. This can be achieved by checking that the input price is positive and consistent with market conventions. Once an invalid price is detected, a constructor may *throw* (`throw`) a C++ *exception*. All other *class operations* should maintain this invariant.

Sometimes certain operations have to violate an invariant. In such situations, it is assumed that several operations must be called in sequence, with the final effect recovering the invariant. If such operations are called outside of the sequence, then this can violate the invariant and bring an object into an inconsistent state. The best practice is to prohibit the use of everything that may violate an invariant. In C++ operations, violating an invariant, can be encapsulated as `private` or `protected`. Preferably, a `public` interface of a class should consist of operations maintaining the invariant of the objects of the class.

The design of a price class that solves the two price-checking tasks can be more complicated than the simple wrapper of the C++ built-in type `double`. While the constructor of such a wrapping class would be able to check that the input price is positive for any market data, the second task, checking that the price is in the correct increment of minimal ticks, depends on a concrete market convention. Literally, part of the code that validates soybean and gold prices cannot be the same. This is because a one-tick move for the *gold futures contract*, traded on the Commodity Exchange (COMEX), is 0.1 dollars per ounce and one contract is 100 troy ounces. This makes the *dollar value of one tick* equal to $10 per contract. This convention differentiates gold futures contract specifications from those of soybeans. Object-oriented or generic programming techniques are valuable for solving both of these verification tasks.

Object-orientation implies developing a hierarchy of price classes based on *inheritance* (Rumbaugh et al. 1999). The Unified Modeling Language (UML) (Rumbaugh et al. 1999) defines this term as:

> *The mechanism by which more specific elements incorporate structure and behavior defined by more general elements.*

With my choice of the programming language, the inheritance relationship between classes is expressed using the syntax and mechanism of C++ inheritance, which distinguishes base and derived classes. The *interface inheritance* can be useful in order to access objects of different price classes in run time conveniently by means of a single *interface* (a named set of operations that characterize the behavior of an element) (Rumbaugh et al. 1999). It is defined as:

> *The inheritance of the interface of a parent element but not its implementation or data structure.*

In C++, this can be achieved by defining a base price class without any data members and with `public`, `virtual`, *pure operations* (Stroustrup 2000) and deriving from it the classes supplying definitions of the pure operations. I will use object-orientation but not for the class `Price`.

For the development of the major class `Price` I have chosen generic programming—programming with types efficiently supported in C++ by the *class* and *function templates*. In our case, this means a parameterization of the class template Price by classes, checking prices in accordance with a particular contract specification.

Procedural Programming

Let me begin with the classes to be used for *default* (an *abstract contract* with the minimum tick 0.0001 and the tick dollar value 0.0001), gold and soybean futures contracts specifications. They are defined in the header file `Spec.h`.

```
#ifndef __Spec_h__
#define __Spec_h__

namespace PPBOOK {

    class SpecDefault {
    public:
        static  const char* name(){return "default";}
        static  double      tick(){return 0.0001;}
        static  double      tickValue(){return 0.0001;}
    };

    // Gold
    class SpecGC {
    public:
        static  const char* name(){return "GC";}
        static  double      tick(){return 0.1;}
```

```
        static  double      tickValue(){return 10.0;}
    };

    // Soybean
    class SpecS {
    public:
        static  const char* name(){return "S";}
        static  double      tick(){return 0.25;}
        static  double      tickValue(){return 12.5;}
    };
    //…

} // PPBOOK

#endif /* __Spec_h__ */
```

The notation … means that more lines in the file are possible. In the code above, this notation is shown as a C++ comment so that one can copy and paste this text as it is and use it in a real program. I have verified that all code extracted in this manner will compile without errors. In order to avoid collision of a class name with the same name used in other libraries, I place the definition of the class inside the `namespace PPBOOK`, which means "potential profit book." Similar `namespace` syntax will be used for other identifiers.

The gold contract with the ticker symbol GC is traded on the COMEX division of the New York Mercantile Exchange (NYMEX). The soybean contract with the ticker symbol S is traded on the CBOT. The symbol, minimum tick value, and dollar value of the tick (the tick value times the contract size) are only a part of what can constitute the contract specifications. I have selected only those specifications that are needed for profit computation and can improve diagnostics.

The three classes contain only `static` functions. No objects are required in order to call them. Calling these functions is applying C++ for procedural programming: the interfaces use only functions and no objects of classes. However, compared to other procedural programming languages, such as C, C++ still gives technical advantages by using namespace, class scope, and stronger type checking (Stroustrup 2000).

While the three classes that we have defined, `SpecDefault`, `SpecGC`, and `SpecS`, are different they have the same number and type of `static` functions—the same interfaces. Combining these functions into a class scope creates a new quality. This quality already distinguishes the obtained result from pure procedural programming, where nine stand-alone functions would need to be introduced to handle the three contracts. It is quite common that several programming paradigms can be mixed together in a software project.

Once the number of selected futures contracts or stocks increases, more specification classes can be defined and added in different header files in the same manner. Next, I am going to introduce the class `Price` parameterized by a specification class.

Object-Based and Generic Programming

The definition of the class `Price` is in the header file `Price.h`:

```
#ifndef __Price_h__
#define __Price_h__

#include <cmath>
#include <sstream>
#include <stdexcept>
using namespace std;

namespace PPBOOK {

    template<class S>
    class Price {
    public:
        Price(double p) : p_(p){check(p);}
        Price(const Price<S>& p) : p_(p.p_){}
        double          price() const {return p_;}
        Price<S>&       operator=(double p)
                        {check(p); p_ = p; return *this;}
        Price<S>&       operator=(const Price<S>& p)
                        {p_ = p.p_; return *this;}
    private:
        double          p_;
        static  void    check(double p)
        {
            if(p <= 0.0) {
                ostringstream   s;
                s   << S::name() << " price " << p
                    << " must be positive.";
                throw   invalid_argument(s.str());
            }
            double  nt = p / S::tick();
            if(fabs(floor(nt) * S::tick() - p) > 1.0e-8 &&
                fabs(ceil(nt) * S::tick() - p) > 1.0e-8) {
                ostringstream   s;
                s   << S::name() << " price " << p
                    << " must be a whole number of ticks " << S::tick();
                throw   invalid_argument(s.str());
            }
        }
    };

    // Only for illustration
    //template<class S>
    //double
    //operator-(const Price<S>& lhs, const Price<S>& rhs)
    //{
```

```
    //    return  lhs.price() - rhs.price();
    //}

} // PPBOOK

#endif /* __Price_h__ */
```

Programming with objects, which are instances of classes not related through an inheritance relationship (see previous sections), is known as object-based programming or programming with abstract data types (Booch 1994). However, in this design a specification class parameterizes the class template `Price`. The last can be successfully instantiated, if the public interface of a specification class contains the `static` functions `name()` and `tick()`. All three classes—`SpecDefault`, `SpecGC`, and `SpecS`–satisfy this requirement. Using them as parametric types to change the behavior of an object of the class `Price` is known as *generic programming*—programming with types. This small example shows a hybrid of object-based and generic programming paradigms.

The `private static` function `check()` throws an exception if a price is not positive or not equal to whole multiples of ticks. For making error messages more descriptive I use the C++ Standard class `ostringstream`. The normal work of the function `check()` is based on the two assumptions that `S::name()` may not return a zero pointer and `S::tick()` may not return a zero value. Of course, I could make the function longer and check both conditions; however, returning zeros in both cases is out of the problem's domain. Even if this is done by mistake, it must be corrected in the trivial implementation of the specification classes. Hence, instead of writing additional checking code, which should never be used under the normal conditions existing at compile time, I am omitting it.

The function `check()` is called explicitly by the constructor creating a price object from raw `double` data and by the overloaded `operator=()` assigning the new `double` value price to the existing object. This design closes all gates and prevents the input of inconsistent prices into an object of the class `Price`. For better invariant protection, I excluded the *default constructor*. There is no reasonable value for a price created by a constructor without arguments. This is because zero prices have been excluded from our domain. Sometimes the lack of a default constructor may cause a technical inconvenience because a built-in array cannot be created from a class without it (Koenig 1996). However, I am going to reuse the Standard C++ class `vector` not requiring a constructor without arguments for a class of an element.

One may say that it would be convenient to have an operator that converts a price object to a `double`. Then class does not need the operation `price()` and can behave in many situations in the same way as the built-in type `double`. However, it has been pointed out (Stroustrup 2000) that the presence of both a nonexplicit constructor from a type and a conversion operator to the type can lead to ambiguity or surprises when conversion is unexpected. For instance, the C++ Standard class `string` has a constructor from `const char*`; however, it supplies the explicit operation `c_str()` in order to convert it to `const char*` and does not allow automatic user-defined conversion by an `operator const char*()`.

You may notice that if I do not supply an automatic conversion operator, then at least I may need to overload global arithmetic and input/output operators so that they could accept objects of the class `Price` as well as values of the type `double`. The way this is done is shown

right after the definition of the class `Price` in a comment. However, I will not use this approach because, in my opinion, it compromises the safety that has already been achieved in the introduced classes.

Example Test1.cpp

From time to time I shall apply the C++ `typedef` specifier. It helps to create a new identifier for naming already existing types. This does not introduce new classes but alias names making the code shorter and more readable. The program `test1.cpp` containing a few `typedef` and C++ `main()` function is:

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

#include "Spec.h"
#include "Price.h"
using namespace PPBOOK;

typedef Price<SpecGC>           GoldPrice;
typedef vector<GoldPrice>       GoldPrices;
typedef Price<SpecS>            SoybeanPrice;
typedef vector<SoybeanPrice>    SoybeanPrices;

int main(int, char*[])
{
    try {
        GoldPrices      gp;
        gp.push_back(449.10);

        SoybeanPrices  sp;
        sp.push_back(661.74);
    }
    catch(const exception& e) {
        cerr    << e.what() << endl;
    }
    catch(...) {
        cerr    << "Unknown exception" << endl;
    }
    return  0;
}
```

If we are going to use new identifiers such as `GoldPrices` and `SoybeanPrices` in multiple source and header files, then the `typedef` statements should be placed in a separate C++ header file. Notice how prices can be appended to the collections using `push_back()`. In this

case, an implicit user-defined conversion of `double` to object of the class `Price` works because I added the constructor from `double` and did not declare it using the C++ keyword `explicit`. The number of currently available prices in the collection is returned by the operation `vector::size()`. A price can be extracted given an index by `operator[]`. An object of the class `GoldPrices` accepts only positive numbers, which are whole multiples of 0.1 (the minimum price move). An object of the class `SoybeanPrices` accepts only positive numbers, which are whole multiples of 0.25. This program generates the following output:

```
S price 661.74 must be a whole number of ticks 0.25
```

The wrong soybean price has been rejected!

This simple framework consisting of the specification classes, price class, and sequence vector collection illustrates another important principle of software design known as the *Open-Closed Principle* (Meyer 1988, Martin 1996). It is opened for extensions assuming adding new specification classes and closed for modifications. "Closed for modifications" means that in extending this framework one does not need to change existing code, which might introduce bugs into a program that is already working. Of course, "closed for modifications" assumes that we do not extend our problem domain by changing the number of requirements. For instance, if a common default price value is known for each price specification, then specification classes might get the additional `static` function `S::defaultPrice()`. It would then be reused for defining the default constructor in the class `Price`. Adding those operations would be a modification of existing classes and a violation of the principle. The need to make these changes would indicate that our original design was not adequate to the solving task.

Object-Oriented Programming

Working with a class template `Price` and corresponding `vector` means that template parameters must be known in compile time. Consequently, I would need to write a template version of each algorithm calling the *vector of prices*. However, in order to apply such template algorithms, the price specification template parameters again must be known in compile time. This can easily fulfill our application working with prices of different specifications by `if-else` or `switch` statements. The introduction of new specification classes would require the changing of these places in the code, which is very likely prone to errors. I would like to simplify the writing of these applications so that they select the correct algorithms in run time based on the contract price specifications. To accomplish this, we will need a class in run time that manages either the algorithms or the vectors of prices. I have chosen the last option.

In order to reach the goal, I apply object-oriented programming. This means that the fundamental, logical building blocks should be objects. The objects must be instances of some classes. The classes are related via inheritance relationships (Booch 1994). I build a hierarchy of the classes available through a common interface, where each concrete class implements a sequential collection of prices with given contract price specifications. This hierarchy is encapsulated within a concrete class managing collections of different price types. This managing class aggregates an object of an appropriate concrete class from the hierarchy and delegates to this object a subset of its own collection responsibilities. The *aggregation* and

delegation techniques and also multiple *design patterns* based on object-oriented programming are described in Gamma et al. (1994). The following code shows the interface class `IPrices` from the header file `IPrice.h` (the leading character "I" stands for "interface"):

```
#ifndef __IPrices_h__
#define __IPrices_h__

namespace PPBOOK {

    class IPrices {
    public:
        virtual ~IPrices(){}
        virtual IPrices*    clone() const = 0;
        virtual const char* name() const = 0;
        virtual double      tick() const = 0;
        virtual double      tickValue() const = 0;
        virtual size_t      size() const = 0;
        virtual double      operator[](size_t n) const = 0;
        virtual void        assign(size_t n, double price) = 0;
        virtual void        append(double price) = 0;
        virtual void        clear() = 0;
    };

} // PPBOOK

#endif /* __IPrices_h__ */
```

It is always necessary to decide if a `virtual` operation (method) should be declared constant. This issue is discussed in Lippman (1996). In this case, distinguishing between operations accessing and modifying an object's state seems straightforward. The operation `clone()` plays a role of so-called "virtual copy constructor" (Stroustrup 2000). The template class `CPrices` from the header file `CPrices.h` implements the interface (the leading character "C" means "concrete"):

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

#include "Price.h"
#include "IPrices.h"
using namespace PPBOOK;

class Prices;

namespace PPBOOK {
```

```
    template<class S>
    class CPrices : public IPrices {
        friend  class Prices;
    public:
        virtual CPrices*    clone() const {return new CPrices(*this);}
        virtual const char* name() const {return S::name();}
        virtual double      tick() const {return S::tick();}
        virtual double      tickValue() const {return S::tickValue();}
        virtual size_t      size() const {return p_.size();}
        virtual double      operator[](size_t n) const
                            {return p_.at(n).price();}
        virtual void        append(double price){p_.push_back(price);}
        virtual void        assign(size_t n, double price)
                            {p_.at(n) = price;}
        virtual void        clear(){p_.clear();}
    private:
        vector<Price<S> >   p_;
        CPrices(){}
    };

} // PPBOOK

#endif /* __CPrices_h__ */
```

In defining this class template, I introduce the entire hierarchy of concrete classes implementing the interface `IPrices`. The creator of C++ Bjarne Stroustrup (2000) discussed this very powerful technique, where a template class is derived from a nontemplate *abstract class*. The default constructor is made `private` and the class `Prices` is declared as `friend`. Default copy and assignment semantics are suitable in this case. Let me introduce the last item of this triad: the managing class defined in the header file `Prices.h`:

```
#ifndef __Prices_h__
#define __Prices_h__

#include <string>
using namespace std;

#include "IPrices.h"
using namespace PPBOOK;

namespace PPBOOK {

    class Prices {
    public:
        Prices(const string& s) : p_(create(s)){}
```

```
        Prices(const Prices& src) : p_(src.p_->clone()){}
        ~Prices(){delete p_;}

        const char* name() const {return p_->name();}
        double      tick() const {return p_->tick();}
        double      tickValue() const {return p_->tickValue();}
        size_t      size() const {return p_->size();}
        double      operator[](size_t n) const {return (*p_)[n];}
        void        assign(size_t n, double price){p_->assign(n, price);}
        void        append(double price){p_->append(price);}
        void        clear(){p_->clear();}
        Prices&     operator=(const Prices& rhs)
        {
            if(this != &rhs) {
                IPrices*    tmp = rhs.p_->clone();
                delete  p_;
                p_ = tmp;
            }
            return  *this;
        }
    private:
        IPrices*    p_;
        static IPrices* create(const string& s);
    };

} // PPBOOK

#endif /* __Prices_h__ */
```

An object of the class `Prices` is a sequential collection of prices similar to a vector. It does not provide for a reference to an element in the collection because such a reference would depend on a price specification class. Nor does it have `operator[]` returning a reference to the built-in type `double` allowing left-hand side assignment. This class is concrete and has no `virtual` operations. The default constructor is not available. The copy constructor as well as assignment operator is defined. An object of this class can be an element of a value-based data container such as the class `vector`. The `private static` function `create()` is a factory producing objects of different types from our hierarchy. If either this function or the operations `clone()` return a valid nonzero pointer or throw an exception, then constructors either create an object with invariant hold or an object will not be created at all. Hence, it is an implementation task to ensure that `create()` and `clone()` possess this property. This simplifies implementation of other operations, because it is no longer necessary to check that pointer `p_` is zero. In this situation using a zero pointer without checking would be a software disaster. This will not happen if the `operator new` on failure throws `bad_alloc` exception instead of returning 0. If this is not the case, then it is clear that modification of `create()` and `clone()` would be straightforward. Basically, it is easy to write implementations where the `create()` and `clone()`

operations act independently on the behavior of the `operator new` and never return 0 but throw an exception if something is wrong.

Exception Safety

It is important to note the *exception safety* properties of the class `Prices`. To do this, it is useful to follow the classification of levels of exception safety discussed in Stroustrup (2000). The level *no guarantee* means that if an exception is thrown by an operation working on an object, then the object is left corrupted. The invariant is not hold. The level *basic guarantee* means that after an exception is thrown, basic invariants are hold and no memory or other resources leak. The level *strong guarantee* means that after an exception is thrown the object remains in the same state as it was before, calling the operation throwing the exception. The level *no throw guarantee* means that an operation never throws an exception. Careful examination shows that operations in the class `Prices` belong to the strong guarantee level and the *destructor* belongs to the no throw guarantee level. The definition of the assignment operator shows how this is reached. If `clone()` possess the properties required in the previous section, then it either throws an exception and nothing changes in the object, or it returns a valid pointer. After the last is returned, the `operator delete` does not throw exception (no throw guarantee for destructor). Assignment of one pointer to another pointer may not throw an exception either. Here the order of lines is important. For instance, if one deleted `p_` and after that called `clone()`, then an exception thrown by `clone()` would leave the object in a corrupted state because the pointer would contain an address of a destroyed object. A suitable order of lines here is a very cheap way to reach the strong guarantee.

Production of Concrete Objects

The `static` function `create()` plays the role of a *factory* producing price objects of a given type dependent on a specification passed as a string. In order to maximally restrict access to this function, it is declared `private` and defined in the source file `Prices.cpp`:

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

#include "Spec.h"
#include "Prices.h"
#include "CPrices.h"
using namespace PPBOOK;

namespace PPBOOK {

    IPrices* Prices::create(const string& s)
    {
        if(s == SpecDefault::name())
            return  new CPrices<SpecDefault>;
        if(s == SpecGC::name())
```

```
            return  new CPrices<SpecGC>;
        if(s == SpecS::name())
            return  new CPrices<SpecS>;
        throw   invalid_argument(
                    "Cannot create object of class Prices for " + s);
    }

} // PPBOOK
```

This function parses the string parameter and decides the object of which concrete type should be created. If the specification is not in the list, then an exception is thrown. The parsing is character case sensitive. If one needs a new type of specification in the system, then a new class is written similar to `SpecDefault`, `SpecGC`, and `SpecS`. This is added in new header files. Until this point, the Open-Closed Principle discussed earlier is obeyed: there are to be no existing code changes while a new price specification is introduced. However, the function `create()` violates this principle. For a class `Prices` based on a new contract specification, an additional `#include` statement, `if`, and `new` operators must be added in the source file `Prices.cpp`. True, these are only three lines, but the file and function `create()` must be changed within this design every time a new contract specification is added.

You may recognize in the design based on the functions `clone()` a variation of the design pattern *Factory Method* also known as the design pattern *Virtual Constructor* (Gamma et al. 1994). The factory method `clone()` was used in order to implement the copy constructor and assignment operator in the class `Prices`. In the context of factories, it makes sense to mention interesting alternative variations of the prototype-based *abstract factory* (Gamma et al. 1994), (Vlissides 1998, 1999).

We are now fully equipped to write a program computing Pardo's potential profit.

PARDO'S POTENTIAL PROFIT

Simple Algorithm for a True Reverse System

Robert Pardo (Pardo 1992) formulated a definition of *potential profit* and an algorithm of its computation in this statement:

> *Potential profit is the profit that could be realized by buying every bottom and selling every top. More precisely, it is the sum of every price change where each change is taken as a positive number.*

Are the first and second sentences in agreement? Yes, they are if we assume that selling every top means short selling so that the result of this transaction is a net short position in the market. In other words, if we were long one *unit* (a contract or share) before the transaction, then the transaction liquidates the long position and enters a new short position at the same price. The same process occurs when buying every bottom. We are always

in the market and our trading strategy is a true reversal system, switching our position from long to short and back again. Only under these conditions will the sum of every price change, where each change is taken as a positive number, give us the right result. This number is substantially greater than a long-only strategy where we buy a bottom, then sell the next top in order to exit the market, then wait until the price drops to the next bottom before buying again. The C++ implementation of this algorithm is placed in the header file `PardoPotentialProfitAlg.h`:

```
#include "Prices.h"
using namespace PPBOOK;

namespace PPBOOK {

    inline  double
    pardo_potential_profit(const Prices& prices)
    {
        double  ppp = 0.0;
        for(size_t j = 1; j < prices.size(); j++)
            ppp += fabs(prices[j] - prices[j - 1]);
        return  ppp * prices.tickValue() / prices.tick();
    }

} // PPBOOK

#endif /* __PardoPotentialProfitAlg_h__ */
```

This algorithm returns zero if the collection of prices is empty. The code of the function `pardo_potential_profit` is my interpretation of Robert Pardo's algorithm description. It does not mean that the same or similar code or formulas are used by Robert Pardo. In order to emphasize his contribution, I am applying the prefit "pardo" in function name `pardo_potential_profit` and file names `pardo.cpp` and `PardoPotentialProfitAlg.h`.

The Program Computing Pardo's Potential Profit

It is convenient to have a program that works as a filter with the following interface on Microsoft Windows: type prices.txt | pardo or on UNIX it might look like cat prices.txt | pardo (in both cases the same effect is reached using the syntax pardo < prices.txt). Both programs, "type" on Windows and "cat" on UNIX, send the contents of text files to the standard output. A *filter program* takes information from *standard input*, processes it, and sends the result to the *standard output*. The *pipe syntax* (|) is a mechanism that passes the output of one program to the input of another, and is supported by the operating system (Stevens 1999). This program may determine the type of a price by reading a conventional descriptor from input. The final program from the file `pardo.cpp` is:

```
#include <iostream>
```

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

#include "PardoPotentialProfitAlg.h"
using namespace PPBOOK;

int main(int, char*[])
{
    try {
        string  market;
        cin >> market;
        Prices  p(market);
        double  price;
        while(cin >> price)
            p.append(price);
        cout    << market << " " << pardo_potential_profit(p) << endl;
    }
    catch(const exception& e) {
        cerr    << e.what() << endl;
    }
    catch(...) {
        cerr    << "Unknown exception" << endl;
    }
    return  0;
}
```

This program assumes that the input file has a descriptor of the market (default, GC, S) as the first token. Because this is a filter program it can operate in all possible ways, where the input is obtained from standard input object `cin`. The "echo" command can be applied as follows:

```
echo S 661.50 662.75 659.25 | pardo
S 237.5

echo S 661.50 662.75 659.21 | pardo
S price 659.21 must be a whole number of ticks 0.25

echo HG 661.50 662.75 659.25 | pardo
Cannot create object of class Prices for HG
```

The program rejects the soybean price 659.21 because it is not a multiple of 0.25, the minimum move. It also knows nothing about HG. The last entry could be a copper contract but has no identification. Adding a *copper futures contract* specification class should extend the program. However, even if this is not yet done, you still can get Pardo's potential profit for such a contract. To accomplish this, you can apply the extended capabilities based on the *descriptor default* and the class `SpecDefault`. This class has the minimum tick 0.0001 and the dollar tick

value 0.0001. This makes the dollar value of one point (*tick value/tick*) equal to one dollar. If you know the dollar value of one point for the contract, which is not yet programmed, then you can multiply the "default" Pardo's profit by this value and get the correct final result.

For instance, the soybean contract S is already in the list. However, if it would not be yet included, then the task could be solved as:

```
echo default 661.50 662.75 659.25 | pardo
default 4.75
```

The command line contains the descriptor *default* and the same set of soybean prices. These prices are certainly whole multiples of 0.0001 and will be accepted. The *default* Pardo's profit value is equal to 4.75. The value of one point in the soybean contract is $50. Multiplying 4.75 by 50 we get 237.5. This is the right profit observed earlier. Of course, using *default* means that you need now to take care about correct input prices. The program will accept everything that is a multiple of 0.0001. However, once a specification for a new contract is added permanently, the input prices will be verified automatically.

In practice, most prices will be available as a long list of records in a text file. Such a text file must begin with the character descriptor of the market or stock, which is followed by prices. The descriptor and individual prices must be separated by a delimiter, which can be space, tab, or new line characters. The program allows arbitrary combinations of these delimiters, collectively known as *white spaces*. The program reuses the magic C++ Standard `operator>>` to read the input token by token.

This program performs many operations: reading and checking prices, creating appropriate objects in run time, filling them with prices and managing memory, and applying a simple mathematical algorithm. It consists of exception safe building blocks. At the same time, it is compact. Implementation of many operations can be done in just one line of code. A combination of generic and object-oriented programming, reusing variations of sound design patterns and the C++ Standard Library classes, makes the system stable and open for extensions. A minor drawback—that the existing code is not completely closed to modifications—is well compensated because new changes inside `create()` can be done in a simple and safe manner, localizing the place of changes. Another benefit of this design is that modifications that extend the specifications will not require recompilation of other modules but only the file `Prices.cpp`.

CONCLUSIONS

- Potential or maximum profit is a market property.
- Pardo's algorithm computes potential profit with all transaction costs taken as zero.
- Pardo's algorithm implies a true reversal trading strategy.
- A simple program illustrating major C++ design principles and programming paradigms is written to calculate Pardo's potential profit.