Queens College, CUNY, Department of Computer Science

Computational Finance

CSCI 365 / 765

Fall 2017

Instructor: Dr. Sateesh Mane

October 4, 2017

1 Project Part 1

1.1 Abstract base class

- *Let us begin our grand adventure!*
- Let us design an abstract base class for the equity derivatives of this course!
- **Question: write a short essay to list the various ways you know how to design an abstract base class.**

  While the above question has nothing to do with finance (nor even computation), I would just like to read your knowledge of the topic, which might exceed mine.

- This is how I propose to design the abstract base class: *the class constructor will be protected.*
  1. **Question: what would happen if the constructor were private?**
  2. **Question: what about the destructor? Can that be protected? Private?**
  3. **Question: why must the destructor be virtual?**
- What will our abstract base class contain?
  1. There will be only one constructor (the default constructor). There is no point in being too clever with constructors: an abstract base class does not know *(and* **cannot** *know)* what the derived classes will look like.
     (a) **Question: what is a default constructor?**
  2. The compiler will supply the following two things by default: a copy constructor and assignment operator. For now, we shall not permit copies of our objects. I shall enforce this policy in the following way.
     (a) **Policy: the copy constructor and assignment operator will be private.**
     (b) **Question: how does this policy forbid copies of objects?**
  3. There are two other things the compiler will also supply by default, and we shall not overwrite the compiler defaults. Do you know what they are?

     This may be tricky. Not everyone knows about them. They are rarely overwritten.
  4. An abstract base class is an interface. It must have some virtual methods.
  5. It could also contain some data fields, but since the derived classes will contain their own specialized data, we must decide what is a "common set" of data for our purposes.

- Our abstract base class is intended for derivatives on equities (stocks or stock indices). We *could* attempt to design an abstract base class for a wider class of derivatives, but that is too complicated (for these lectures, anyway). There is no point in being too clever.

  1. All derivatives have a fair value.
  2. We can make the fair value a data member of the abstract base class, even though we cannot instantiate an object of the base class directly. *We shall revisit this decision.*

- In the case of a stock, we might say the fair value is trivial: it is the market price of the stock.

  1. **This is not so.**
  2. **We may wish to experiment *"what if the stock price drops by 10%?"* or other hypotheticals.**
  3. We shall have to learn to distinguish between the **market price** of a financial instrument and its (theoretical) **fair value**.
  4. The distinction is particularly clear in the case of options (to be studied later).
     (a) The market price of an option is the price at which it trades in the financial markets. It is an observable number, set by market supply and demand. **It does *not* depend on a mathematical model.**
     (b) The fair value of an option is a *theoretical number*. It is the result of our numerical calculations, based on various input data, and in general it *does* depend on a mathematical model. This is the "computational finance" content of this course.
  5. Traders can compare the market price to the fair value to make trading decisions.
     Is the financial instrument overpriced or underpriced? Is there a profit opportunity?
  6. We shall calculate the fair value, etc. to support the traders to make business decisions. That is computational finance.

- Understand that this homework assignment is more than a toy exercise: we are designing a set of equity derivatives classes which we shall employ throughout this course, which you may possibly take with you beyond this course. Designing the abstract base class is the first step.

1.2 Input parameters

- Our classes will require input data to perform their computations. There are two types of input parameters. The distinction between them is important. Do not confuse them.
- The **indicative** data parameters (or simply "indicatives") are those parameter values which specify the terms and conditions of a financial instrument.
    1. For example the expiration date is an indicative. The expiration date of a derivative does not change because of trading in the financial markets.
    2. Some types of inputs can be ambiguous to classify. The dividends (amounts and dates) paid by a stock are announced by the company. They do not change day to day because of trading in the financial markets. However, companies *do* announce changes in their dividend policies, but this happens maybe once a year and not every day. For this reason, we shall say the dividends paid by a stock are an indicative.
    3. **Indicative data values are typically stored in a database, and their values will typically be obtained from a database query.**
        (a) We shall not deal with databases in this course.
        (b) The calling application will set the values of the indicative data.
        (c) For example, we may type numbers into an input text file, which the calling application will read. The input text file is effectively our database.
- The **market data** are parameters whose values change frequently because of trading.
    1. The market price of a stock is the most obvious example of market data.
    2. Stock prices can change very rapidly in the stock market.
    3. The values of market data do not come from a database.
    4. The values of market data are input "by hand" by a customer (= calling application).
    5. The market data must reach us via a different channel from the indicative data.
- Here is a very typical scenario.
    1. The calling application calls the database only once, to obtain the values of the indicative data. The indicative data values are saved somewhere.
    2. The calling application obtains the market data many times, for example as stock prices change in the stock market.
    3. The calling application calls our class methods multiple times, for example every time a stock price changes. The indicative data values are the same for every function call.

1.3 Indicative data

- We shall store the indicative data in a class `IndicativeData`.
- Note that `IndicativeData` is a separate class.
  It is not derived from the abstract base class in Section 1.1.
- In principle, every derived class has a different set of indicative data. For example, options have an expiration date but stocks do not. Stocks pay dividends but options do not, etc.
- Logically, therefore, `IndicativeData` should also be an abstract base class, from which we derive subclasses for each financial derivative. This is rigorously the correct architecture, but it is too much work. We wish to focus on financial computations, not data entry.
  1. We shall declare `IndicativeData` to be a gigantic collection of **all** the indicative data for **all** the derived classes.
  2. Each derived class uses whatever it needs in `IndicativeData` and ignores the rest.
  3. If we write a new derived class which requires new indicative data fields, we add the fields to the class `IndicativeData`. This means we have to recompile all the previously defined classes. It is a weak point of this class architecture.
- Hence the class `IndicativeData` will look like this

```
class IndicativeData
{
public:
  // data for derived class 1
  .....

  // data for derived class 2
  .....

  IndicativeData() { ... } // constructor initializes all data to default values
};
```

- **Policy: all the class data members will be public.**
- If we want to get/fetch data values, we read them from the object.
- If we need to change/set data values, we go into the object and update them. The calling application will do this.
- Your CS professors will probably be aghast at this lack of data security.
- However, it is a waste of our time to write a long list of accessor/mutator class methods. It is merely clumsy to go through an input/output interface for data access, which teaches us nothing about computation or finance.

1.4 Market data

- The market data will follow the same paradigm as the indicative data.
- We shall store the market data in a class `MarketData`.
- We shall declare `MarketData` to be a gigantic collection of all the market data for all the derived classes.
- Each derived class uses whatever it needs in `MarketData` and ignores the rest.
- If we write a new derived class which requires new market data fields, we add the fields to the class `MarketData`. This means we have to recompile all the previously defined classes.
- Hence the class `MarketData` will look like this

```
class MarketData
{
public:
  // data for derived class 1
  .....

  // data for derived class 2
  .....

  MarketData() { ... } // constructor initializes all data to default values
};
```

- **Policy: all the class data members will be public.**
- As with the indicative data, if we want to get/set a data value, we read/write the data field in the `MarketData` object.

1.5 Client overwrites

- Actually, the above picture of input data is not complete. There is a third case.
- Consider a simple model of a stock with price S and it pays a continuous dividend yield q.
- The stock price S is **market data**. It changes rapidly because of trading.
- The continuous dividend yield q is **indicative data**. Its value is stored in a "database" and the value of q does not change because of trading.
- Now we shall learn that other things are also possible.
- It is possible that a client ***might wish to experiment and overwrite some data values.***
- The client might ask questions such as these:
  1. *"What would happen if I changed the value of the continuous dividend yield?"*
  2. *"What would happen if the value of the stock price were 10% lower?"*
- It is possible for clients to make such requests.
  1. The client (calling application) can set a new value for the continuous dividend yield q **even though there is already a value for q in the database (indicative data).**
  2. The client (calling application) can set a new value for the stock price S **even though there is already a value for S in the market data.**
- Logically, therefore, there are ***three*** sets of input data.
  1. Indicative data.
  2. Market data.
  3. **Client overwrites.**
- The client overwrites are not new data fields. Instead, they overwrite existing data values in the indicative data and/or market data.

1.6 Client overwrites: implementation

- Client overwrites are not really our responsibility.
  1. It is the *calling application* (not us) who wishes to overwrite various items in the indicative and/or market data.
  2. Hence it is the responsibility of the calling application to manage the input data values it asks us to use for calculations.
- Nevertheless, we must do *something*.
- **Question: why? Answer: because we must provide a user-friendly interface.**
- Consider the following scenario (very typical).
  1. A client wishes to plot a graph of the fair value of an option as function of the stock price, from $S = 1$ to $S = 100$ in steps of 0.1.
  2. The actual market price of the stock is not relevant here. The client wishes to overwrite it in a loop.
  3. The loop has a large number of steps. It is tiresome for the client to edit the `MarkeData` object so many times.
  4. It is also time-consuming to reinitialize our class objects on every pass through the loop.
  5. We must make it easy for the client to change the value of S and recalculate the option fair value in a loop.
- **We must provide an easy way for the client to overwrite the input data.**

  (Although the client still has to take the responsibility for the changes.)
- In fact we have already solved the problem of an easy interface for client overwrites.
- Recall that all of the data fields in our class objects are public.
- Hence to overwrite data, the calling application simply goes into our class object and updates whatever value it wishes.
- The calling application does not need to edit the `IndicativeData` and/or `MarketData` objects and reinitialize our class objects.
- **Question: do you understand that this creates a problem?**
  1. **Question: do you understand that this means that even after the indicative and market data have been input into our class objects, we do not know if the data values will be valid when we use them for mathematical computations?**
  2. The input data values might have been changed (overwritten) by the calling application.
  3. In the above example, what will happen if the client begins the loop starting with a negative value for S? Who will validate that?
  4. This is a problem and we shall have to deal with it (later).

1.7 Client overwrites: accessor/mutator methods

- In a more rigorous architecture, we would supply a set of accessor/mutator methods for the calling application to update the values in our class objects.
    1. We could choose to allow access only to selected data fields, i.e. the calling application would only be able to overwrite the values of some data but not others.
    2. For example, the calling application might not be allowed to overwrite the expiration date of an option.
    3. There could be validation: the calling application would not be allowed to overwrite the stock price to a negative number.
    4. Our architecture has no restrictions on what the calling application can overwrite and no validation checks on the updated values.
    5. Furthermore, our architecture requires the calling application to know the names of the data members in our class objects.
    6. Accessor/mutator methods could have more descriptive names which would be more meaningful to a client.
- However, none of this is computation or finance. We shall live with public data members in our class objects.
- The CS professors may scoff, but it gets the job done.

1.8 Input data: architecture (Part 1)

- The foregoing discussion about client overwrites reveals important lessons about the processing of input data. It teaches important lessons about the architecture of our class objects.
- When we receive an input `IndicativeData` object, our class object must copy the data out of the `IndicativeData` object into the class object itself.
- The same procedure when we receive an input `MarketData` object: our class object must copy the data out of the `MarketData` object into the class object itself.
- We cannot keep pointers or references to the `IndicativeData` and `MarketData` objects.
- **Question: do you understand why?**
- Suppose we hold our input data using pointers or references to the input `IndicativeData` and `MarketData` objects.
  1. Now suppose we overwrite the data, for any reason.
  2. It will **overwrite the original `IndicativeData` and `MarketData` objects.**
  3. This is **very bad**.
- If we maintain pointers to the `IndicativeData` and `MarketData` objects and the calling application deletes them (deallocates memory), it will be bad news for us.
- **Question: what will go wrong if the calling application deletes the input `IndicativeData` and `MarketData` objects?**
- This is our design:
  1. **Our library specifies an interface for the classes for the indicative and market data.**
  2. Effectively, both `IndicativeData` and `MarketData` become abstract base classes.
  3. Calling applications create classes which implement that interface and populate the class objects with data. For example via a stored procedure call to a database, to obtain the indicative data. *We do not need to know how the objects are populated.*
  4. The calling applications pass the populated objects to our class methods, to initialize our class objects. The class methods are virtual functions, and know how to read the data in each case.
  5. Each virtual function reads the input data objects and copies their data into its own self, as explained above.
- Our library design should be formulated so that it would work even if both `IndicativeData` and `MarketData` are changed to abstract base classes. We shall do so.

1.9 Input data: architecture (Part 2)

- Consider the following **bad idea:**

  1. **Our objects hold the input data in copies of the `IndicativeData` and `MarketData` objects.**
  2. This will work. A client overwrite will update the data *in our copy* but not the original objects from the calling application.
  3. Nevertheless, this is a bad idea.
  4. The above idea works because we have defined the `IndicativeData` and `MarketData` objects to be effectively large dumpsters which hold every input data field for every derived class.
  5. In a better quality library design, both `IndicativeData` and `MarketData` would be abstract base classes. We could not instantiate copies of them and we would not know what the derived class objects would look like.

1.10 Input data: architecture (Part 3)

- Do not make the mistake of thinking that, if the `IndicativeData` and `MarketData` classes are someday implemented as abstract base classes (in a future design), there will be one derived class (of the input data) for every one of our classes (of equity derivatives).
- In other words, the new design will have a one to one match of the input data classes to our equity derivative classes.
- **No.** There could be **many** input data classes to just **one** equity derivative class.
    1. Even for something as simple as a stock, for certain applications we might require historical data of the stock prices, maybe for the previous month, or one year.
    2. Admittedly, "stock prices" are "market data" but *historical data* does not change due to trading, and it is stored in a database.
    3. A stored procedure database query will return the historical stock prices to us in an `IndicativeData` object (derived class object).
    4. Hence, depending on what calculaton the calling application wants to do, there could be different indicative data classes, some might have historical data, some might not.
    5. In our current design, where the `IndicativeData` object "contains everything" you see that "everything" is a messy concept. "Everything" is a ***very big thing.***
- **Question: do you see that a true design for the input data classes is a vast design in its own right?**
- **Question: do you see that we want an interface that will accept that future design, whatever it is?**
- We do not know what that future design of the input classes is, but if we architect our interface for the input data so that the `IndicativeData` and `MarketData` classes are treated as abstract base classes, then our library architecture will work.
- Hence our class objects should not hold the input data in local copies of the `IndicativeData` and `MarketData` objects. We do not know what the new input classes will really be. Moreover there may be many input classes for just *one* of our equity derivative classes.
- We want our class objects to have their own data members, to support all the calculations they need to do. For each calculation, there will be some database query (and market data) and some input objects will bring in that data. For each calculation, our class objects will copy what they want from the input data objects.
- ***Question: do you understand?***

1.11 Input data: validation

- The issue of client overwrites also teaches us a lesson about validation of the input data.
- Naturally, our class methods to read the input from the `IndicativeData` and `MarketData` objects will validate the input data. We will flag an error if the input data fails for any reason.
    1. Example: indicative data, stock dividend amount < 0: fail.
    2. Example: market data, stock price ≤ 0: fail.
- ***However, a client could overwrite the inputs with invalid data values.***
- Since our architecture for "client overwrite" is to simply allow the calling application to enter our class objects and do anything it likes, our architecture has no safeguard to validate a client overwrite.
- There is also another important scenario: the indicative and market data may be inconsistent, and we must check for this. Consider this scenario:
    1. Indicative data: stock dividend D, check $D > 0$, validation check passes.
    2. Market data: stock price S, check $S > 0$, validation check passes.
    3. There is no client overwrite.
    4. **However, when we try to do a calculation, we find $S < D$, the stock price is less than the dividend, which is invalid.**
- Hence the indicative and market data may both pass their validation checks, but the two sets of data may be inconsistent, for mathematical reasons.
- **Before we begin a mathematical calculation, we must validate all the data and we must do this inside the calculation function itself.**
- Hence even if we wrote accessor/mutator methods to control data access, ***we would have to perform the above validation check anyway.***
- It is simpler to make all the data members public.

1.12 Output data

- After our class methods have performed their computations using the input data, they will return output data to the calling application. It should come as no surprise that the output data will follow the same paradigm as the input data.
- We shall store the output data in a class `OutputData`.
- We shall declare `OutputData` to be a gigantic collection of all the output data for all the derived classes. Each calling application reads whatever it needs out of `OutputData` and ignores the rest.
- If we write a new derived class which requires new output data fields, we add the fields to the class `OutputData`. This means we have to recompile all the previously defined classes.
- Hence the class `OutputData` will look like this

```
class OutputData
{
public:
  // data for derived class 1
  .....

  // data for derived class 2
  .....

  // etc
};
```

- **Policy: all the class data members will be public.**
- As with the input data, if we want to get/set a data value, we read/write the data field in the `OutputData` object.
- Our class objects will write data into the object. The calling application will read it.
- After we return an `OutputData` object, the calling application will own the object. It does not belong to us.
- We do not care what happens after we return an `OutputData` object to the calling application.
- It does not matter if the calling application overwrites data in the `OutputData` object, etc.

1.13 Output data: clear()

- We have not said anything about the input and output data classes except that (at present) they are large dumpsters which will hold "everything" and in a future design they will be abstract base classes.
- Nevertheless, we need something from the `OutputData` class that we are not demanding from the `IndicativeData` and `MarketData` classes.
- We require a `clear()` method.
- It is ***we*** who must populate the `OutputData` object at the end of a calculation, and return it to the calling application.
- We cannot allow the `OutputData` object to contain stale data, for example from a previous function call. That would be very bad, professionally.
    1. **It could also be dangerous.**
    2. The calling application might make a decision, based on our output, and it might make a wrong decision because of stale/bad/wrong data that ***we did not put into the object.***
- **Question: do you understand the difference of "responsibility" for input and output data?**
- **Question: do you understand that if the input data is wrong, it is the fault of the calling application for giving us bad data?**
- **Question: do you understand that if the *output data* is wrong, it is *our fault* for giving outside users bad data?**
- Our class methods cannot go into an `OutputData` object and clear it item by item. Every time the `OutputData` class was modified, we would have to rewrite all our relevant class methods. That would be impractical.
- The `OutputData` class must expose a public method `clear()` to clear its data members.
- Note that `clear()` may not set all the data members in the `OutputData` class to zero. It would set them to default values (which might be 1, not zero, for `int` and `double` variables, or `true` for Booleans, etc.)

1.14 "Get" methods for input data objects

- There are two input data classes: `IndicativeData` and `MarketData`.
- The `IndicativeData` and `MarketData` objects will be populated by the calling application, and given to us. We can write function prototypes to process them.

```
virtual int getIndicativeData(const IndicativeData & indData) {return 0;}
virtual int getMarketData(const MarketData & mktData)         {return 0;}
```

  1. Both functions will perform validation checks on their input data.
  2. Both functions have return type `int` because they can fail if the input data is not valid.
  3. The functions will change the values of data members in our class objects. Hence they cannot be declared `const`.
  4. However, they will *not* change the values of the *input objects.* Hence the *inputs* are declared `const`.
  5. We can give function bodies for the base class implementation. Do nothing and just return 0.

- **Question: do you see that the above simple interface treats both `IndicativeData` and `MarketData` as abstract base classes?**
- The functions are virtual and the derived classes will know what to do with them.

1.15 "Set" methods for data objects

- The `OutputData` object will be populated by us, after the computations have finished, and returned to the calling application.
- We shall call the function `setOutputs()`. Note the get/set naming convention.
- We can write a function prototype.

```
virtual void setOutputs(OutputData & outData) const {}
```

  1. The function does not change the values of the data members in our class object: it only populates the output object.
  2. Hence the function return type is `void` and the function is declared `const`.
  3. Notice that this is the opposite to the "get" methods: here the class method is `const` but the output object is not `const`.
  4. We can give a function body for the base class: it does nothing.

- We shall always ask the calling application to instantiate the `OutputData` object and pass a reference to us. We populate the object and return it to the calling application.
- As explained earlier, it is the calling application which owns the `OutputData` object. It does not belong to us.
- If we instantiated an `OutputData` object and gave it to the calling application, we would lose ownership of it and there would be a memory leak.

1.16 Get/set mini quiz

- **Question: do you understand why all three get/set methods must be virtual?**
- **Question: do you understand why `getIndicativeData()` and `getMarketData()` must be public?**
  - *Well, not necessarily. This can be discussed.*
- **Question: do you understand why `setOutputs()` must be protected?**
  - *We do not want a calling application to call* `setOutputs()` *prematurely.*
- **Question: do you understand why `getIndicativeData()` and `getMarketData()` are not `const` methods but the input objects are `const`?**
- **Question: do you understand why `setOutputs()` is a `const` method but the output object is not `const`?**

1.17 Virtual functions

- Our list of virtual functions will include at least the following.

```
virtual int getIndicativeData(const IndicativeData & indData) {return 0;}
virtual int getMarketData(const MarketData & mktData)         {return 0;}
virtual void setOutputs(OutputData & outData) const {}

virtual int calc() {return 0;}
```

- The first three functions have been described previously but `calc()` is new.
- The method `calc()` will contain the computational core of our class.
  1. The values of inputs will be set prior to calling it, and the values of output parameters will be set after it has finished (if the computation is successful).
  2. The function `calc()` cannot be declared `const`.
  3. The computations in `calc()` can fail, hence it must have return type `int`, return 0 (success) and nonzero (fail).
  4. We can give `calc()` a function body in the base class: it does nothing and returns 0.
  5. Note that `calc()` will not call `setOutputs()` directly. Other class methods (not yet specified) will do that. If the computation fails, those other class methods will call `setOutputs()` and set the relevant output fields to appropriate values and exit gracefully.
- The function `calc()` is the "finance heart" of our library. Other class methods serve to set up data for it to operate, and to process the output that it produces. It lies deep in our library and contains proprietary software that we do not reveal to the world.
- Note that `calc()` must run fast, hence the data must already be validated before calling it. Other class methods (not specified yet) will validate the inputs so it is safe to call `calc()`.
- We do not want outside applications to be able to call `calc()` directly without proper data validation. We may also have to allocate memory, etc. prior to calling `calc()`.
- Hence the function `calc()` must be **protected.**
- **Question: what would happen if `calc()` were private?**

1.18 Non-virtual functions

- We can give one example of a non-virtual method of the abstract base class.
- Recall in Sec. 1.1 it was pointed out that every derivative has a fair value. (It was also stated in Sec. 1.1 that we can make the fair value a data member of the abstract base class, but we see now that it will be a member of the `OutputData` class.) The class method is

```
int FairValue(OutputData & outData);
```

- The method `FairValue()` must be public. Also it cannot be `const`.
- Note that `FairValue()` is **not virtual**.
- The method `FairValue()` will operate as follows.
  1. It will internally validate the input data (by calling a virtual function).
  2. If the validation check passes, it may perform internal processing (it depends on the derived class).
  3. If the processing is successful, it will call `calc()`.
  4. If the calculation is successful, it will call `setOutputs()` to populate the output data.
  5. Then it will exit with return value 0 (success).
  6. If any of the steps fail, `FairValue()` will exit with a nonzero return value.
  7. Note also that if any of the steps fail, `FairValue()` will be responsible for managing allocated memory, etc.
  8. It is a "high level" public method, which will deal with all the momeory management and input/output issues.
- The method `calc()` is a low level method, whose task is to do mathematical computations. Data validation, input/output and memory management are not its responsibility.
- We have a high level public function to wrap around the low level computation function.
- Our software architecture is taking shape.

1.19 Why non-virtual?

- Why is `FairValue()` non-virtual?
- Why not make it (and all the base class methods) virtual?
- **It is a matter of quality control.**
  1. We want to be sure that `FairValue()` performs all of the steps described above, and in the correct order.
  2. If `FairValue()` were virtual, the derived classes could override it and who knows what they might do. They might forget to validate the data properly, etc.
  3. By declaring `FairValue()` to be non-virtual, we can guarantee that all of the steps described above are performed, and in the correct order.
- Of course `FairValue()` will call virtual functions internally, for example for data validation, and there is no guarantee that the derived classes will do their jobs correctly. That is a fact of life of polymorphism and inheritance. We have to delegate some responsibility to the derived classes.

1.20 FairValue() schematic

- We can give a body to `FairValue()`, at least symbolically. This will show what we need, to set up the abstract base class to do a complete fair value calculation.

```
int ABC::FairValue(OutputData & outData)
{
  int rc1 = validate();       // obvious
  if (rc1) {
    clear(outData);
    return rc1;
  }
  int rc2 = preprocess();     // this may allocate memory
  if (rc2) {
    clear(outData);
    return rc2;
  }
  int rc3 = calc();           // do the calculation --- the heart of the system
  int rc4 = postprocess();    // cleanup ... do this before exit
  if (rc3 || rc4) {
    clear(outData);
    return (rc3 ? rc3 : rc4);
  }
  setOutputs(outData);        // obvious
  return 0;
}
```

- We require three additional class methods `validate()`, `preprocess()` and `postprocess()`.
- All three are virtual functions, because they must operate on the derived class objects.
- All three have return type `int`, because they can fail, hence must return an error status.

1.21 Preprocess, postprocess and validate

- It was explained in Sec. 1.11 why we must validate the input data inside the mathematical calculation function itself: this is the purpose of `validate()`. It is `const` because it does not alter the values of any data members in the class. It is a public class method.
- The virtual function `preprocess()` performs any tasks that must be done before calling `calc()`. It may allocate memory internally. Many things are possible, and `preprocess()` is a place to do them.
- The virtual function `postprocess()` is a partner to `preprocess()`. It performs a cleanup job, after `calc()` has finished.
- Both `preprocess()` and `postprocess()` must be protected class methods. We cannot allow outside calling applications to call them. They cannot be declared `const`.
- For a simple derived class such as `Stock` for stocks, it is possible that both `preprocess()` and `postprocess()` are empty. Not every derived class needs to override the base class implementation.
- We can give function bodies for all three in the base class: they do nothing and return 0.

```
virtual int validate() const {return 0;}
virtual int preprocess()     {return 0;}
virtual int postprocess()    {return 0;}
```

1.22 Abstract base class: schematic

Do you think you could write it?

- **Name the class `ABC` (for "abstract base class").**
- Our class architecture avoids pure virtual functions.
- A pure virtual function is a nuisance because every (non-abstract) derived class must override it, even if the derived class does nothing with it.

```
class ABC
{
public:
  ...

protected:
  ABC();                                    // protected constructor
  ...

private:
  ABC(const ABC&) {}                        // do not allow copy
  ABC& operator=(const ABC&) {return *this; } // do not allow copy
};
```

1.23 Indicatives data

This is a beginning. We shall add to it later. But this is enough to get started.

```
class IndicativeData
{
public:

  // many things
  double _expirationDate;

  // stock
  double _divYield;

  void clear()
  {
    _expirationDate = 0.0;
    _divYield = 0.0;
  }

  IndicativeData()
  {
    clear();
  }
};
```

1.24 Market data

This is a beginning. We shall add to it later. But this is enough to get started.

```
class MarketData
{
public:
  double _interestRate;
  double _marketPrice;
  double _t0;
  double _volatility;

  void clear()
  {
    _interestRate = 0.0;
    _marketPrice = 0.0;
    _t0 = 0.0;
    _volatility = 0.0;
  }

  MarketData()
  {
    clear();
  }
};
```

1.25 Output data

This is a beginning. We shall add to it later. But this is enough to get started.

```
class OutputData
{
public:
  double _FairValue;
  double _Delta;
  double _Gamma;
  double _Theta;
  double _Vega;
  double _Rho;
  double _impliedVolatility;

  void clear()
  {
    _FairValue = 0.0;
    _Delta = 0.0;
    _Gamma = 0.0;
    _Theta = 0.0;
    _Vega = 0.0;
    _Rho = 0.0;
    _impliedVolatility = 0.0;
  }

  OutputData()
  {
    clear();
  }
};
```