Advanced Concepts in C++ Programming

Goals:

- Introduce object-oriented programming in C++
- Describe polymorphism and how we can use it to write general classes
- Review the use of templates and other advanced C++ features
- Discuss the use of design patterns in C++ programming

Course Projects

- **Reminder**: Proposals for your project are due October 15th.
- Your proposal should consist of:
  - A one or two paragraph summary of your topic and how you intend to apply it to quant finance.
  - A brief discussion of how you plan to design the project, including the programming language you intend to use.
  - An explanation of the data that you will need for your project and how you plan to obtain it.
  - At least two supporting references that you will leverage in your work.
- Make sure you have access to the data that you will need for your project before submitting your proposal.

Course Projects

- One approach that you can take with your project is to find a paper that is of interest and implement it.
- You could apply this paper to a different dataset or try to tweak the problem somehow.
- You should also attempt to find a few corroborating papers.
- My expectation is that your implementation will be robust.
- I also expect that you will analyze the strengths and weaknesses of the technique you are implementing and propose areas of further research.
- *A good rule of thumb is that you should be spending about the time you spend on 3-4 homeworks combined on your project.*

Course Projects: Sample Topics

- **Portfolio Optimization**: Mean-Variance Optimization, Risk Parity, Shrinkage Estimators of Covariance Matrices
- **Factor Modeling**: Statistical or Fundamental Factor Model Construction, Factor Portfolios
- **Risk Management**: VaR / CVaR analysis of different types of portfolios.
- **Fixed Income Modeling**: Yield Curve Construction, Bootstrapping a Credit Spread Curve
- **Stochastic Models/Exotic Options Pricing**: Stochastic Vol. Models, Local Vol. Models and their use in pricing exotics.
- **Quant Trading**: Back-testing mean-reversion, momentum or pairs trading strategies

Course Projects: Sample Topics

- **Risk Neutral Density Analysis**: Extract risk-neutral density via an implied vol. surface.
- **Index Replication**: Identify most efficient way of replicating an index
- **Comparison of Risk Neutral & Physical Densities**
- What else?

Review: Best practices for Coding and Proper Memory Management in C++

- In the last lecture we discussed how to pass parameters to functions in C++ and also discussed some of the most basic data structures and the use of pointers.
- To summarize, here is a list of suggestions for your coding in C++:
  - Whenever possible, parameters should be passed to functions by const reference. This ensures they are not changed involuntarily.
  - If the variable must be changed then pass the variable by reference instead of const reference.
  - You should never pass a parameter by value unless it is one of C++'s 5 basic types.

  - Use std::vectors instead of arrays, and always use the built-in safe functions to access an element and add elements. This will prevent us from *reading past the end of a container* or even more pernicious *writing past the end of a container*
  - Whenever you are creating a new pointer, use the **new** keyword to allocate memory dynamically on the heap.
  - Use boost::shared_ptr or another smart pointer class to make sure destruction of the pointer happens efficiently at the appropriate time. This will prevent **memory leaks** and **dangling pointers**
- Today we will cover designing classes in C++, as well as several other advanced concepts...

Object-Oriented Programming in C++

- Recall from our prior discussions of objected-oriented programming that it means writing code that is based on a set of objects, or classes, rather than a set of functions.
- C++ has robust support for object-oriented programming allowing us to define user-defined classes and define class hierarchies.
- Classes may interact with each other and may inherit behavior from each other (often dynamically as we will see)
- **Inheritance** and **Polymorphism** are two critical components of class design in C++, as we will see.

Object-Oriented Programming in C++

- The classes that we build may contain their own internal data (**member data**) and they may also contain their own functions (**member functions**).
- These functions may be accessible only from within the class, or may be called by users who instantiate the class.
- The member data may also be accessible only internally, or from any user.
- These classes may also define how the are instantiated, by creating a set of constructors.

User-defined classes

- User-defined classes allow us to create a user-defined data type which is some combination of member data and member functions, or actions that the class can perform.
- Each member data must have a type declared. This type can be a basic type, or another user-defined complex type.
- Member function signatures should look similar to the signatures for global functions we saw earlier. They should declare a return type and must include a return statement.
- Member data and functions must define their level of accessibility.
- Class definitions, like function declarations, should go in header files, whereas the implementation of each member function (or constructor) should go in a cpp file.

Class Example

```
    class Foo {
        public:
            int bar;

            Foo(int bar_){ bar = bar_;};

            int getFoo(){ return bar;}
    };
```

Notice that in this example we did not separate the declaration of the class into a header and implementation of its functions into a cpp file. This will become more important as classes get bigger.

Our simple class has a single **constructor**, a piece of **member data**, and a single **member function**.

Inheritance in C++

- Inheritance is one of the most powerful object-oriented programming concepts, and is an integral part of writing good C++ code.
- It involves creating a class hierarchy and creating classes that inherit certain behavior from their parent class, while providing greater levels of specialization in other areas.
- An inheritance hierarchy can be many levels.
- Derived classes can implement their own new functions and have their own new member data.
- Derived classes can also overload the behavior of their parent, that is they can override a member function.

Base & Derived Class Example

```
class Foo {
    public:
    Foo(int bar_){ bar = bar_;};

    int getBar(){ return this->bar;}

    int bar;
};

class childFoo : public Foo {
    public:

    childFoo(int bar_, int bar2_)
        : Foo(bar_) { bar2 = bar2_; } ;
    int bar2;
};
```

Public vs. Private Inheritance

- Notice that in the previous example of inheritance an access label had to be specified when specifying the base class.

```
        class childFoo : public Foo
```

- The public keyword in this context means that the member data in the base class will be visible, and modifiable in the derived class.
- Use of the private keyword, conversely, would have meant that we could not have accessed member data in the base class, even if they were declared as public.
- public inheritance is more commonly used for this reason.

User-defined structs

- We can also create **structs** in C++. Structs are fundamentally similar to classes, however, they have different default levels of accessibility of their member functions and member data.
- By default, structs have member data accessible to users.

```
struct Foo {
    public:
        int bar;
        Foo(int bar_){ bar = bar_;};
        int getFoo(){ return bar;}
};
```

- Notice how similar the definition of our struct is to the definition of our class. In fact they are quite similar (and almost interchangeable).

Difference between class and struct

- The main difference between a class and a struct is the accessibility of member data and functions.
- In particular, in a struct member data and member functions are public by default.
- Conversely, in a class, member data and member functions are private by default.
- Obviously, we can make these equivalent by always specifying the accessibility of member data and member functions.

Difference between class and struct

- In practice, I use classes for full objects and structs for simple data stucture / data container types.
- Using this distinction, which is common among developers, allows us to identify a struct or a class and immediately know what type of object to expect.
- Structs can then be used in std::vector or other similar containers and easily be iterated through, modified or transformed in any desired manner.
- Recall that it is best practice to make class data members private and to make getter/setter member functions. This is noticeably different than a struct, and therefore structs should be used for simple data structures where integrity checks are not needed.

Constructors: Initializing Objects

- Constructors help us define the code that is run when a user-defined object or class is initialized.
- We may often create many constructors for the same underlying class which provides us multiple ways to create the object.
- Constructors for the same class must have different signatures. That is they must take somewhat different parameters for the compiler to differentiate between the two.
- Constructors may also have default values for certain parameters. The parameters that have default values must be at the end of the argument list.

User-Defined Constructor Example

```
class Position {
    public:
        Position(double shares_,
                     bool long_){
        shares = shares_;
        is_long = long_;
    };

    double shares;
    bool is_long;
};
```

We can add default values to the arguments to the constructors by assigning the default values in the constructor definition.

User-defined Constructor with Default Values: Example

```
class Position {
    public:
        Position(double shares_ = 0.0,
                      bool long_ = true){
            shares = shares_;
            is_long = long_;
        };

    double shares;
    bool is_long;
};
```

Notice that the = **value** syntax after the argument was used to add a default value for the parameter.

Default Constructors

- A default constructor is a *special constructor* that requires *no input arguments* in order to initialize an instance of the class.
- Default constructors are sometimes called automatically, and if a default constructor is not defined then your code may not compile if one of these automatic calls occurs.
- For example, the following code calls an object's default constructor:

```
Position pos;
```

Default Constructor Example

```
class Position {
    public:
        Position(){
            shares = 0; is_long = true;
        };
        Position(double shares_,
                     bool long_){
            shares = shares_;
            is_long = long_;
        };
        double shares;
        bool is_long;
};
```

Notice that there are now two constructors defined. Without the first constructor the line Position pos above would not compile.

Copy Constructors

- A copy constructor is another special type of constructor that defines how objects should be copied.
- Like a default constructor, sometimes these are called implicitly.
- Classes have default copy constructors that can be overriden if the desired behavior doesn't match that of the default copy constructor.
- In particular, by default a **shallow copy** of the object is done when the object is copied.
- We may want to override this constructor and create a copy constructor that performs a **deep copy**

Copy Constructor Example

```
Position pos;
//copy constructor is called
Position pos2(pos);

Position pos3(5, true);
//assignment operator is called
pos3 = pos;
```

Here is an example of an example that performs a deep copy:

```
Position(const Position& other){
    shares = other.shares;
    is_long = other.is_long;
    vec_prices = std::auto_ptr<std::vector>
            (new std::vector())
    for (unsigned int i = 0;
        i < other.vec_prices->size();
        i++){
        vec_prices->push_back(other.vec_prices->at(i));
    }
};
```

Constructors in Inheritance Hierarchies

- When creating an instance of a derived class multiple calls to a constructor are made.
- Notice that in our inheritance example earlier the constructor for the childFoo class had a call to the Foo constructor. This creates an explicit call to the base class constructor where you can pass in the parameters you choose.
- If this piece of code is omitted, then the default constructor of the base class will be called. If a default constructor is not defined for the base class, then your code will not compile.
- The order of the calls to the constructors is from the highest level in the inheritance hierarchy (i.e. the base class) to the lowest (i.e. the derived class).

Destructors

- Destructors are functions that define the code that is executed as an object is destroyed.
- Destructors are responsible for cleaning up associated memory, as well as cleaning up all loose database/file connections.
- Destructors are declared as normal functions and must have a certain signature:

```
~Position(){
    std::cout << "Position class being destroyed"
              << std::endl;
};
```

- Notice that the character denotes a destructor and that no arguments are passed to the destructor.

Destructors in Inheritance Hierarchies

- When classes are defined within a class hierarchy, multiple destructors are called when an object is deleted.
- The order of calls to a destructor is from lowest level in the class hierarchy (i.e., derived class) to highest level in the class hierarchy (i.e., base class).
- Notice that this is the opposite order of constructor calls in a class hierarchy.

Member Data

- Classes often contain one or more elements of member data.
- Use of member data helps us with **data abstraction** as each class can be responsible for its own internal data.
- To declare an element of member data, we must define a type and a name for the member data within the class definition, for example:.

```
class Position {
    Position(double shares_,
                        bool long_){
        shares = shares_; is_long = long_; };
    double shares;
    bool is_long;
};
```

- In this example, **shares** and **is_long** are member data for the Position class.
- Each element of member data must have an access label of public, private or protected.
- The default access label for member data in classes is private.
- The order that the member data appears within the class definition does not matter.

Member Functions

- In addition to member data, most classes that you create will have at least one member function that defines some sort of action for the class.
- This action might define how the class acts on itself, or may define how the class interacts with another class that is passed in as a parameter.
- Member functions have access to all member data for the class.
- They can also take a set of input parameters just as we saw with global functions.
- These parameters can be passed by value, by reference, by const reference or by pointer. The same best practices apply for passing parameters to member functions.

Member Functions

Consider the following simple Position class:

```cpp
class Position {
    public:
        Position(double shares_,
                bool long_){
            shares = shares_; is_long = long_;
        };
        double calcMktValue(double price){
            return price * this->shares;
        };
        double shares;
        bool is_long;
};
```

The function calcMktValue is a member function of the Position class, returns a double and has access to the shares member data.

Accessibility of Member Data / Functions

- C++ has three levels of accessibility for member data & functions:
  - **public**:
    - member data is visible from outside the class.
    - member data can be modified from outside the class.
    - member function can be called from outside the call.
  - **private**
    - member data can't be read or modified outside the class.
    - member functions can't be called outside the class.
  - **protected**
    - member data can't be read or modified outside the class, unless it is in a parent or base class.
    - member functions can't be called from outside the class, unless they are called from a child or derived class.

Defining Accessor Functions

- It is best practive to define all member data private to ensure that only the class modifies the object.
- This type of data abstraction, or data hiding is an important principle in C++.
- This may seem like a small and inconsequential difference, but for large systems it can make a critical difference.
- If this principle is followed, then all changes are centralized and easier to validate.
- I try to name all getter functions using the convention **get***** and all setter functions using the convention **set*****

Defining Accessor Functions: Example

```
class Foo {
public:
Foo(int bar_){ bar = bar_;};

int getBar(){ return this->bar;}
void setBar(int bar_){ this->bar = bar_; }

private:
int bar;
};
```

Notice that the member variable is declared as private, meaning it is not accessible outside the class.

Also notice that the setBar function has **void** as a return type. This means that the function does not return anything.

const Member Functions

- A const member function is a function that doesn't change the underlying member data of the class in its execution.
- We can declare a member function const by adding the **const** keyword to the end of the function declaration:

```
double calcMktValue(double price) const{
    return price * this->shares;
};
```

This calcMktValue function will not be allowed to change the shares variable, and attempting to do this will cause a compile time error.

const Member Functions

For example, the following code will not compile:

```
double calcMktValue(double price) const{
// this line will not compile for const function
this->shares = 0.0;
return price * this->shares;
};
```

It is best practice to declare member functions const whenever possible. This avoids subtle bugs associated with member data being changed inadvertantly.

static Member Functions

- If we want to define a function that is part of a class, but does not depend on an instance of the class, we can do so using the **static** keyword.
- Note that static functions have no access to data from instances of the class.
- Doing this is analogous to creating a global function that is unaffiliated with any class.
- We might still prefer to include a static function as part of a class, rather than just as part of the global namespace in order to help group our functions and signal what they do.

static Member Function Example

```
class Foo {
public:
Foo(int bar_){ bar = bar_;};
static int calcAverage(int bar1, int bar2){
return (bar1 + bar2) / 2; }
int getBar(){ return this->bar;}
void setBar(int bar_){ this->bar = bar_; }
private:
int bar;
};
int main(int argc, const char * argv[]) {
Foo foo1(5);
Foo foo2(5);
int avg = Foo::calcAverage(foo1.getBar(), foo2.getBar());
return avg;
}
```

Equality Comparisons of Classes

- By default, user-defined classes do not implement a default version of the **== operator**.
- This means that by default, we cannot check two user-defined classes for equality.
- For example, the following code will not compile:

```
Position pos1(100.0, true);
Position pos2(100.0, true);

bool are_equal = (pos1 == pos2);
```

- In many cases we will want to accomplish this, and we can do so by **overloading the == operator**.

Equality Comparison Overload Example

```
class Position {
    public:
        Position(double shares_,bool long_){
            shares = shares_; is_long = long_;
        };
        bool operator==(const Position & rhs) const{
            return (this->shares == rhs.shares &&
                this->is_long == rhs.is_long);
        }
        int shares;
        bool is_long;
};
```

Now that we have implemented the $==$ operator, the previous code will compile and will perform a check of the member data to check for equality of two Position class instances.

Operators and Operator overloading

- There are many operators that we may choose to overload to make our coding more efficient and more readable.
- We saw an example of overloading the **== operator**.
- We also talked about overloading a copy constructor's behavior.
- It would also be natural to want to over the **assignment operator**, or $=$ operator.
- We may also want to overload the $+$ operator, or other mathematical operators.
- The $==$ operator is just one example of a relational operator, or a comparator. We may also want to overload the $<$, $<=$, $>$ and $>=$ operators in some cases.

Operators and Operator overloading

- In some cases, we may need to overload certain operators in order to use our user-defined classes in certain ways.
- For example, we need to overload certain comparison operators in order to have our user-defined class be stored in a sorted list.
- Overloading an operator simply requires creating a function with the proper signature, as we saw with the $==$ operator.
- Here is another example, with the $+$ operator:

```
Position operator+(const Position rhs){
    int newShares = this->shares + rhs.shares;
    return Position(newShares, this->is_long);
};
```

Internal this pointer

- In a C++ object, the **this** variable refers to a pointer to the class that's being referenced.
- If we **dereference** the **this** pointer, we can then us the **this** variable to access a class's member data and functions.

```cpp
class Foo {
    public:
        int bar;
        Foo(int bar_){ bar = bar_;};

        int getFoo(){ return this->bar;}
};
```

- Using the **this** pointer is a good, explicit way to show that you are referencing a class member variable or function.

Abstract Base Classes

- An abstract class is a class that we never intend to initialize, but we want to inherit from.
- An abstract class is created by defining the class with at least one **pure virtual function**, which can be done by adding **=0** after the declaration and not defining the function.
- A pure virtual function is a function that must be overriden by a derived class.
- Abstract classes are used to define a class structure and force derived classes to define certain methods.

Abstract Base Class Example

```
class BasePosition{
public:
    BasePosition(){;};
    virtual double calcMktValue(double price) const = 0;
    double shares;
    bool is_long;
};
class Position : public BasePosition {
    public:
        Position(double shares_,
                     bool long_){
            shares = shares_;
            is_long = long_;
        };

        virtual double calcMktValue(double price) const{
            return price * this->shares;
        };
};
```

Note that if I were to try to instantiate an instance of the BasePosition class directly, then I would receive a compilation error because the calcMktValue function is not defined in the base class.

Polymorphism

- Polymorphism is a fundamental concept in object-oriented programming.
- It describes the ability to inherit functionality from a parent class, but also to overload functions from the parent class and define different behaviors for them.
- Operator overloading is an example of polymorphism at work.
- We must define our classes and class hierarchies such that the program knows which functions in the class hierarchy to call.
- Consider an example where the same function with the same signature is defined in both the base and the derived class. *Which function should the program call when we call that method?*

Polymorphism

- Let's consider two examples in our Position class hierarchy:

```
class BasePosition {
    public:
        int shares;
        bool is_long;
        void my_func(){
            std::cout << "calling from the base class"
                    << std::endl;
            }
};
class Position : public BasePosition {
    public:
        Position(double shares_,
                bool long_){
            shares = shares_;
            is_long = long_;
    };

    void my_func(){
        std::cout << "calling from the derived class"
                  << std::endl;
    }
};
```

- First, let's consider creating an instance of the derived class. Which version of my_func() should it call?

```
Position pos(100.0, true);
pos.my_func();
```

- Second, let's consider creating a pointer to the base class and initializing an instance of the derived class. What version of my_func() should it call now?

```
std::auto_ptr<BasePosition> basePtr(new Position(100, true));
basePosPtr->my_func();
```

Polymorphism

- In our previous example, in the first case the derived classes my_func() was called and in the latter case the base classes my_func() was called.
- This may not always be desired. Notice that the pointer was initialized as a Position object and not a BasePosition object. Therefore we most likely want it to behave as a Position object even if we store it in a pointer for the base class.

virtual Functions

- It turns out that the way to accomplish this is through the use of **virtual functions**.
- A virtual function is a function with the keyword virtual declared in front of its return type.
- virtual functions tell the compiler to apply dynamic binding to the member function.
- virtual functions notify the compiler to replace the body of the base class function with the body of the derived class.

virtual Function Example

```cpp
class BasePosition {
    public:
        int shares;
        bool is_long;
        virtual void my_func(){
            std::cout << "calling from the base class"
                    << std::endl;
        }
};
class Position : public BasePosition {
    public:
        Position(double shares_,
                bool long_){
            shares = shares_;
            is_long = long_;
        };
        virtual void my_func(){
            std::cout << "calling from the derived class"
                                << std::endl;
        }
};
```

- Notice that now having declared the function my_func() virtual, the derived class is called in both of our above examples.
- This is quite a powerful concept. We can create a pointer to the base class and store within it one of many derived classes.
- You will find that you use polymorphism a lot in writing good general code and it is worth taking a step back and trying to understand its power.

virtual Destructors

- Destructors should always be declared as virtual functions.
- If they are not, we might see undefined behavior when we store a derived class object in a pointer to the base class.
- For example, the following code will result in undefined behavior if the destructor is not declared as virtual:

```
std::auto_ptr<BasePosition>
    basePosPtr (new Position(100.0, true));
basePosPtr->my_func();
```

The undefined behavior will be caused by basePosPtr being deleted.

Recurvise Functions

- C++, just like Python supports recursive functions.
- Recall that recursive functions are when we call a function from within itself, with the idea being that we are progressively calling simpler problems as we iterate through recursive function calls.
- Recursive functions must have some end condition where the value is computed internally in the function, otherwise we will receive an infinite recursion error.

Recursive Functions

To see a simple example, let's consider computing a factorial function:

```cpp
int Factorial(int fact){
    if (fact == 1){
        return fact;
    } else{
        return fact * Factorial(fact-1);
    }
}
```

Notice that the terminal condition, combined with the fact that we are continually decreasing the integer passed to the function, enables us to solve this via recursion.

What other examples of recursive functions can you think of?

Exception Handling

- Exception handling is a critical component of writing good code.
- Even the best written code will encounter unexpected exceptions.
- Exceptions are themselves objects.
- There are many different types of exception classes that we might see.
- Exception objects have certain common properties (i.e. a member function **message**)

Catching Exceptions

- Generally speaking we want to catch exceptions and handle them rather than allow them to derail our application.
- In some cases, we may want to catch any exception thrown.
- In other cases, we may want to handle different exception types differently.
- C++ supports try {} catch {} finally {} statements to handle exceptions.
- More than one catch {} block is supported to catch different types of exceptions.

Try Catch Example

Consider the augmented Position class:

```
class Position {
    public:
        Position(double shares_,
                bool long_){
            shares = shares_;
            is_long = long_;
        };

        double calcMktValue(double price){
            return price * this->shares;
        };
        double shares;
        bool is_long;
};
```

Try Catch Example

We might write the following piece of code to compute market value of a position and catch exceptions that occur:

```
try {
Position pos(5, true);
pos.calcMktValue(100.0);
} catch (std::exception){
std::cout << "caught std::exception" << std::endl;
} catch (...){
std::cout << "caught non std::exception" << std::endl;
}
```

Notice that we are initially trying to catch all exceptions of type std::exception and then have a second catch block to catch any other types of exceptions.

Throwing Exceptions

- There are some cases that we may want to throw our own exceptions. (Why?)
- Earlier in the course, we talked about an example in the Black-Scholes formula where we might want to throw our own exception if a negative volatility was passed to the function.
- We may also want to throw an exception to prevent a user from reading to or writing from beyond the end of an array or other container.
- An exception can be thrown by first creating a new instance of an exception object and then using the **throw** keyword.

Throwing Exceptions: Example

Here's a simple example that we could add to our Position class code that ensures that negative values for shares don't correspond with is_long set to true:

```cpp
if (pos.shares < 0 && pos.is_long == true){
     std::string message = "Throwing an exception";
     std::runtime_error ex(message);
     throw ex;
}
```

Notice that we threw an exception of type **runtime_error**. We could have thrown any type of exception here, including a custom exception class that we created, however, the base class, std::exception is an abstract base class and can't be initialized directly.

Templates

- Templated programming is an alternative way to write generic and reusable code.
- We have already seen examples of templates, such as std::vector.
- Templated classes enable us to specify the type when creating the object.
- Templates are very good for generic operations that apply to all data types with few restrictions (i.e. storing a container of objects)
- They are less useful for performing specialized actions.
- Functions, or classes can be templated.
- In a templated class, all class functionality must adhere to the template paradigm.

Template Function Example

```
template <class U>
U add(U u1, U u2){
     return u1 + u2;
};
int main(int argc, const char * argv[]) {
     int x = 1;
     int y = 4;
     int z = add(x, y);
     std::string a = "test1";
     std::string b = "test2";
     std::string c = add(a, b);
     return z;
}
```

Notice that I was able to call my simple template function with both int's and string's as inputs.

Template Class Example

```
template <class T>
class Foo
{
    public:
        Foo(T val_){
        val = val_;
    };

    T addToVal(T newval){
        return this->val + newval;
    };

    T val;
};
int main(int argc, const char * argv[]) {
    int x = 4;
    int y = 6;
    Foo<int> foo1(x);
    int z = foo1.addToVal(y);
    Foo<std::string> foo2("test1");
    std::string b = "Test2";
    std::string c = foo2.addToVal(b);

    return z;
}
```

boost::shared_ptr class

- boost::shared_ptr is a class defined within the boost library intended to make working with pointers easier and safer.
- It enforces certain checks, such as reference count checks to prevent memory leaks and dangling pointers.
- In particular, the use_count() function returns the number of pointers currently watching the object. When the count reaches zero, the object is safely destroyed.
- Note: if you do not want to bother installing the boost library, but want to use a smart pointer, then you can use std::auto_ptr. It has much of the same functionality as the boost::shared_ptr class.

Smart Pointer Class Example

Let's look at an example of a smart pointer using the position class that we defined previously:

```cpp
int shares = 5;
bool is_long = true;
std::auto_ptr<Position> pos(new Position(shares, is_long));
```

Notice that when we use a smart pointer, in this case an auto_ptr, we do not need to use the delete command as we would in a raw pointer.

Notice also that we did use the new command to allocate new memory dynamically on the heap.

Casting between Types

- Type casting refers to the process of transforming a variable from one datatype to another.
- Sometimes, we may call a function that helps us with this transformation, for example, we saw an example using the **std::atof** function.
- In other cases, we will on implicit or explicit casts. There are many of types of casts, including:
  - C-style type casts
  - Implicit, or Automatic Casts
  - Explicit Casts
  - Pointer Casts

Implicit Casts

- In many cases, casts from one type to another may happen automatically without you noticing.
- Some casts can happen automatically, whereas other classes don't have defined conversions that enable this automatic conversion.
- Consider the following code:

```cpp
int x = 5.2;
```

  The right-hand side is a double, and it is implicitly converted to match the type of the left-hand side, an int.
- If we try to perform the same conversion with a string instead of an int, then we will get a compilation error:

```cpp
std::string x = 5.2;
```

C-style Casts

- One way to explicitly cast a variable is using c-style casts.
- C-style casts can be performed by adding a **(type)** statement in front of a variable.
- For example, to explictly convert from a double to an int in our previous example, we could add a c-style cast:

```
int x = (int)5.2;
```

Explicit Casts

- An explicit cast is when when explicitly call a function that performs either a static or dynamic cast between two data types.
- Several types of casts exist, including:
  - static_cast
  - const_cast
  - dynamic_cast
- static_casts perform no run-time checks.
- dynamic_casts do perform run-time checks and can therefore handle more complex conversions.
- Error handling should be added to casts in case conversion fails.
- Each of these cast keywords are templated.

Explicit Casts

- The basic syntax for a cast statement is:

```
TYPE static_cast<TYPE> (object);
```

- Casts can convert between basic types (i.e. double to int)

```
double x = 5.5;
int a = static_cast<int>(x);
```

- Casts can also be used to cast variables to different levels in a class hierarchy:

```
Position pos(100, true);
BasePosition basePos = static_cast<BasePosition>(pos);
```

  This type of cast can be extremely useful when working with class hierarchies.