Programming in C++

Goals:

- Review the basics of C++ programming
- Discuss the process of memory allocation and management
- Introduce object-oriented programming in C++
- Discuss best practices for writing efficient code in C++

Review: Testing for Normality

- There are many ways to test whether an underlying set of data comes from a normal distribution.
- This is a common task in financial modeling.
- When performing certain regressions and other econometric tasks you will assume certain variables are normally distributed. These tests will help you gauge whether these assumptions are realistic.
- When working with the Black-Scholes and Bachelier models, as in your homework, normality tests can help us to ensure our distribution is consistent with the dynamics of the model.
- Common tests include **QQ Plots** and the **Kolmogorov-Smirnov** test (although there are many others)

Review: Testing for Normality in Python

- The **statsmodels** module has a function for calculating the QQ-plot: **statsmodels.api.qqplot**.
- The QQ-plot will compare the CDF's of the empirical distribution and a normal distribution.
- The **statsmodels** module also has a function for the Kolmogorov-Smirnov test called **statsmodels.kstest**

Review: What have we learned so far?

- At this point in the course, you should be fairly comfortable with the Python language and its advanced features.
- You should also be familiar with parsing files, storing data in databases and writing SQL queries.
- You should also know how to write classes in Python and understand why using classes helps us write better code.
- The next main topic (and final programming topic) in this course is programming in C++.

Review: Object-Oriented Programming

- Recall that in object-oriented programming we design our code based on a set of classes or objects, rather than a set of functions.
- Object-oriented programming requires an upfront cost but leads to far more generic code that you are able to leverage later on.
- Depending on the project, you may decide to use either approach.
- In some situations (i.e. prototyping), functional programming is more appropriate as it allows us to get to a solution very quickly.
- In production systems, object-oriented programming is almost always more appropriate.
- This is because as we continue to add to or modify a production system, object-oriented design will allow us to maintain maximum flexibility.

Review: Principles of Object-Oriented Programming

- There are 4 main principles of object-oriented programming:
  - **Inheritance**: allows us to create class hierarchies and have classes inherit functionality from other classes.
  - **Encapsulation**: an object's internal representation is hidden from outside the object.
  - **Abstraction**: exposing necessary functionality from an object while abstracting other details from the user of a class.
  - **Polymorphism**: ability of a sub-class to take on different forms and either have their own behavior or inherit their behavior from a parent
- Abstraction and Encapsulation are clearly closely related.

Types of C++ Projects

- In C++ we can create many different types of projects, including:
  - **Console Application**: A piece of code that produces an executable (.exe) file that can be run from the command line.
  - **Library**: A piece of code that produces a static or dynamic link library (.dll) that can then be leveraged in console or other applications.
- Libraries are useful places to store common calculations that might be used by many console applications.
- In C++ we can also create an Excel Add-In which enables us to expose C++ functions to Excel. This is common practice at banks and hedge funds that use C++.

Projects and Solutions in C++ Visual Studio

- Each console application or library is a separate C++ proect.
- In many cases, it is helpful to link the projects if they are related, for example if we have a console application that depends on a library.
- In Visual Studio, we can create a solution, which is a group of projects.
- In a solution, we are able to, among other things, create build dependencies, for example always build the library then the dependent console application.

C++ Console Application Project

- A console application contains a file main.cpp whose main function is run when the program is launched.

```
1 int main(int argc, const char * argv[]) {
2     return 0;
3 }
```

- The main function must contain a certain signature.
- It must return an integer and it must take two arguments.
- Arguments passed to the program from the console will be available via the *argv* array.
- The first argument, *argc* tells us the number of arguments passed to the console application.

C++ Console Application Project

- For example, consider the following call to a console app named opt_model.exe:

```
opt_model.exe "Put" "2019-01-20" 4
```

- In this case the *argc* argument would be equal to 3 and the *argv* array would be an array of length 3, with strings representing each input.
- Because this is a string array, our application will need to convert the array elements to the correct data type as it reads them.
- The return value of a console application is generally interpreted as the status of the program. It is common to return 0 for success and 1 for failure.

C++ Build Process

- Some languages are interpreted and, as a result, do not require that we build or compile them before we run them. Instead we can execute them line by line dynamically.
- C++, on the other hand, requires a substantial build process before we can run our code.
- This build process consists of two steps:
  - Compilation: Object files are created based on the output of a pre-processor.
  - Linking: The linker takes the object files, links to external libraries, and produces the desired executable or libary.
- In large-scale applications, the build process can take many minutes, making it more difficult to make changes quickly.

Declarations in C++

- In C++ the definition of a function or variable must appear before it is used, otherwise your code will not compile.
- This becomes somewhat complicated to keep track of for large applications.
- To avoid this, we can **forward declare** our functions or variables. That is, we declare them without yet defining them.

Forward Declaration Example

- To see this, notice that the following code does not compile:

```
1  #include <cstdlib>
2
3  int main(int argc, const char * argv[]) {
4  double arg_dbl = std::atof(argv[0]);
5  return FixMyDouble(arg_dbl);
6  }
7
8  double FixMyDouble(double dbl){
9  double retVal = dbl;
10 if (dbl < 0.0){
11     retVal *= -1;
12 }
13 return retVal;
14 };
```

Forward Declaration Example

- However if we add a **forward declaration**, then it does:

```
1  #include <cstdlib>
2  double FixMyDouble(double dbl);
3  int main(int argc, const char * argv[]) {
4  double arg_dbl = std::atof(argv[0]);
5  return FixMyDouble(arg_dbl);
6  }
7
8  double FixMyDouble(double dbl){
9  double retVal = dbl;
10 if (dbl < 0.0){
11 retVal *= -1;
12 }
13 return retVal;
14 };
```

File Structure in C++

- To help organize our code, we generally put all such forward declarations in a separate file.
- This ensures everything is declared first, eliminating the problem.
- Generally speaking, **header files**, or .h/.hpp filess are files that contain only declarations without any definitions.
- There is usually a corresponding **.cpp** file that defines each function declared in the header file.
- Note that if a function is declared in a header but is not subsequently defined in a .cpp file, then the code will compile but will fail to link, and the program will not build.

Including Header Files

- Once we have done this, we will then use the **#include** statement to include header files that have functions declarations in files that they are used.
- It is best practice to never include a **#include** statement in a .cpp file unless it is for its own header file.
- A **#include** statement in C++ is analogous to an import statement in Python in that it makes new code or functons available. The previous example used this to include the cstdlib library, which provides among other things the atof function.
- Continuing our previous example, let's say FixMyDouble was defined in a separate file. What would we do?

Header File Example

- Let's break our code into 3 files:
  - main.cpp
  - lib.hpp
  - lib.cpp
- The code in lib.hpp would be the function declaration:

```
1 double FixMyDouble(double dbl);
```

Header File Example

The lib.cpp would include the definition of the function:

```
#include "lib.hpp"

double FixMyDouble(double dbl){
double retVal = dbl;
if (dbl < 0.0){
    retVal *= -1;
}
    return retVal;
};
```

Header File Example

Lastly, main.cpp would contain:

```
1 #include <cstdlib>
2 #include "lib.hpp"
3
4 int main(int argc, const char * argv[]) {
5     double arg_dbl = std::atof(argv[0]);
6     return FixMyDouble(arg_dbl);
7 }
```

Multiple Declarations

- In C++ we can also only have each function declared once.
- If we have multiple header files that each include other headers, this can be difficult or impossible to avoid.
- To avoid this issue, we can use a **preprocessor directive** that tells us to only run the code (which declares the functions) if some variable hasn't been defined.

Preprocessor Directives

- Preprocessor directives are directives to the compiler on how to compile the code.
- **#include**, which we have used already, is preprocessor directive.
- The directive **#define** enables us to define a variable for the preprocessor.
- The directive **#ifndef** enables us to check if a preprocessor variable has been defined.
- These directives can be used in header files to make sure we don't have multiple declarations of the same functions.

Preprocessor Directive Example

We can use the preprocessor directives **#ifndef**, **#endif** and **#define** to make sure each header is only included once.

```
1 #ifndef lib_hpp
2 #define lib_hpp
3
4 #include <stdio.h>
5
6 double FixMyDouble(double dbl);
7
8 #endif /* lib_hpp */
```

Namespaces

- In C++ we can define our functions, classes or variables in different namespaces.
- By default, all user-defined objects are defined as part of the default, global namespace.
- This can be a helpful way of differentiating code.
- If you are trying to call a function in a different namespace you will need to add a **namespace::** prefix to your function call.
- To avoid having to do this, we can add a **using** statement to add the namespace to the current scope.
- A common pre-defined namespace is **std**, which helps us read from and write to the console, among other things.

std Namespace

- The Standard Template Library (STL) has many useful functions and classes that exist with the std namespace.
- It consists of a set of:
  - Algorithms (i.e. sorting algorithms)
  - Containers (i.e. maps)
  - Iterators
  - Functors
- To use the functionality in this library you will often need a **#include** statement and will also either need to specify the prefix std:: or add a using std statement to the beginning of your file.

Programming in C++

- Now that we are familiar with the structure of C++ we are ready to learn how to code in C++. Let's start with a quick review of the basics of C++ and then we will dive into object-oriented programming and other advanced concepts.
- A lot of overhead goes into writing a C++, however, it is a powerful language and is used widely throughout the quant world.
- Because of this, efficient design and coding best practices become even more important when using C++.
- Building/Debugging are more difficult in C++ so we should be careful designing code and experiments to help us code efficiently.
- Errors in C++ can also be more mysterious, as we will see, so proper exception handling is of critical importance.

Interacting with the console

- The std namespace has functions for writing to and reading from the console:
  - std::cout
  - std::cin
  - std::cerr
  - std::clog
- **std::cout** is by far the most useful command and will print a string to the console.
- We can concatenate multiple strings in a std::cout statements using $<<$
- To add a newline we can use the **std::endl** statement.

Interacting with the console: Example

```
1     std::string str1 = "SPY";
2     std::string str2 = "QQQ";
3     std::cout << str1 << ", " << str2 << std::endl;
4
5     std::string name;
6     std::cin >> name;
```

Data Types

- C++ comes with a set of standard, built-in data types.
- The five basice C++ types are:
  - bool
  - int
  - char
  - float
  - double
- Data types can be **signed** or **unsigned**.
- As an example an unsigned int cannot store negative numbers.

Data Types

- Unsigned data types require less storage and are more efficient.
- However unsigned ints cause problems if they are inadvertently assigned negative numbers.
- For example this program returns:

```
1          unsigned int x = 5;
2          std::cout << x << std::endl;
3          x = -5;
4          std::cout << x << std::endl;
5
6          5
7          4294967291
```

- Unsigned ints are of particular use in for loops, if we are enumerating for 0 to the length of an array / vector.

Data Conversion Functions

- C++ has built-in functions within the std namespace for converting between these basic data types
- We saw an example earlier in a sample program where we converted from an item in a character array to a double, **std::atof**
- A few other examples of data type conversion functions are:
  - **std::itoa**: convert integer to a string
  - **std::atoi**: convert string to integer
  - **std::atof** convert string to double

Typedefs

- Typedefs provide us a way to name a datatype with a convenient pseudonym.

```
1 typedef unsigned int Counter;
2
3 int main(int argc, const char * argv[]) {
4     Counter x = 5;
5     std::cout << x << std::endl;
6     return x;
7 }
```

Enums

- Enums provide us with a way to provide a list of possible values and assign them an intuitive name, and have them stored internally as an integer.

```
1 enum PositionType { Equity = 1 ,
2                     Bond = 2,
3                     Option = 3 };
4
5 int main(int argc, const char * argv[]) {
6     PositionType posType = Equity;
7     std::cout << posType << std::endl;
8     return posType;
9 }
```

Working with Strings

- Notice that string is not one of the basic types in C++.
- A single character is a built-in basic data type, however. An array of characters, often referred to as a C-string, is one way to represent a string in C++.
- The std namespace also has a **std::string** class for storing strings.
- The std::string class has a c_str function to convert back to a C-style string.
- The std::string class also contains some other useful functionality, such as appending strings using the + operator.

std::string class

- The string class lets us append strings:

```
1 std::string str1 = "string1";
2 std::string str2 = "string2";
3 std::string str_all = str1 + str2;
```

- It also lets us iterate through the characters in a string:

```
1     std::string str1 = "string1";
2     for (std::string::const_iterator cit = str1.begin();
3                                 cit != str1.end();
4                                 cit++){
5         std::cout << *cit << std::endl;
6     }
```

std::string class

- It also lets us search for one string within another string:

```
1          std::string str1 = "string1";
2          std::string str2 = "str";
3          std::size_t found = str1.find(str2);
```

- We can also check a string's length using the **length()** function, return a substring of another string using the **substr()** function, access a specific character using the **at()** function.
- C++ also has a useful **stringstream** class that allows you to work with strings the same way you'd work with the console using cout statements.

Conditional Statements

- C++ supports **if / else if / else** statements and also **switch** statements.
- In practice I almost always use if statements and never use switch statements.
- An if or switch statement can include one or more conditions to check.

If Statement Example

```
1  enum PositionType {Equity = 1 ,Bond = 2, Option = 3};
2  PositionType posType = Equity;
3  if (posType == Equity){
4      cout << "processing equity position" << endl;
5  } else if (posType == Bond){
6      cout << "processing bond position" << endl;
7  } else if (posType == Option){
8      cout << "processing option position" << endl;
9  } else {
10     cout << "unexpected position type" << endl;
11 }
```

Switch Statement Example

```
1  enum PositionType {Equity = 1 ,Bond = 2, Option = 3};
2  PositionType posType = Equity;
3  switch (posType){
4      case Equity:
5          cout << "processing equity position" << endl;
6      case Bond:
7          cout << "processing bond position" << endl;
8      case Option:
9          cout << "processing option position" << endl;
10     default:
11         cout << "unexpected position type" << endl;
12     }
```

Loops

- C++ supports both definite (**for**) and indefinite loops (**while**).
- When writing indefinite loops, we must be sure to avoid an infinite loop, that is, we must be sure that the loop's break condition is met at some point in all scenarios.
- Because of this, for loops are safer and more common.
- It is common to have a loop that uses an integer index and loops until it reaches a certain value (i.e. the size or length of an array).
- It is also common to have a loop that uses an **iterator**, or pointers to positions in underlying objects / containers.

For Loop Example

```
1     for (unsigned int i = 1; i < 10; i++){
2         std::cout << i << std::endl;
3     }
```

Notice that we can declare the variable i within the for loop. If we do so, it will go out of scope when the for loop is exited.

Also notice that we used an unsigned int in the loop because we are only expecting positive numbers.

While Loop Example

```
1    unsigned int i = 1;
2    while (i < 10){
3        std::cout << i << std::endl;
4        i++;
5    }
```

Notice that in this case we needed to declare i separately above the while loop. As a result, it will not go out of scope when the while loope exits.

Also notice the $i++$ statement within the while loop. If this line is omitted, then it will result in an infinite loop.

Assignment vs. Equality Comparisons

- Recall that in C++ the assignment operator is the = operator and the == operator is used to test for equality.

```
1 // checks whether x and y are equal
2 // returns true if they are equal
3 // false otherwise
4 y == x;
5
6 // changes the value of y to x
7 y = x;
```

Assignment vs. Equality Comparisons

- What will be the result of the following piece of code?

```
1 while (x = 0){
2     run_some_code();
3
4     if (exit_condition){
5         x = 1;
6     }
7 }
```

Additional Operators

- C++ supports several operators in addition to the assignment operator and equality operator.
- As in Python or any other language, the + operator is generally defined to mean either add or concatenate, depending on the class.
- The [] operator is usually defined to allow accessing an element.
- C++ also has += and *= operators that allow us to add and multiply in place. For example, the following is equivalent:

```
1 int x = 5;
2 int y = 6;
3
4 x = x + y;
5 x += y;
```

File Manipulation in C++

- C++ provides a set of stream-based classes that allow us read from and write to text files.
- Because they are stream-based, they work similarly to writing to the console.
- The **ofstream** class can be used to write to files.
- The **ifstream** class can be used to read in files.
- There is also **fstream** class that supports both reading and writing.
- These classes work by creating a connection to the file, so we should always close that connection when we are done reading or modifying the file using the close() method of the class.

Reading Files

- To read from a file using the ifstream class we start by creating a new **ifstream** object. We can then read it line by line using the **getline** function:

```
1 std::string fname = "test.txt";
2 std::ifstream file ( fname );
3 std::string line;
4 while (getline(file, line)){
5     std::cout << line;
6 }
7 file.close();
```

- Notice that we close the file immediately after we are done reading it to ensure we don't have any loose connections.

Writing Files

- To write to a file using the ofstream class we first need to call the constructor with a single argument specifying the file name.
- We can then write to it the same way we would the console using cout statements.

```
1 std::string fname = "test.txt";
2 std::ofstream file (fname);
3 file << "SPY" << ", " << "QQQ" << std::endl;
4 file.close();
```

- Notice that we close the file at the end of the code.
- Also notice that we can write multiple strings to the same file on the same line using the $<<$ statements as we could with cout statements.

User-Defined Functions

- In C++ we can create user-defined functions either as parts of the objects we create or as separate standalone functions.
- We have already seen examples of standalone functions in some of our previous examples, but let's have another look:

```
1  double calcAverage(double x, double y){
2      return 0.5 * (x + y);
3  }
4
5  int main(int argc, const char * argv[]) {
6      double x = 5.5;
7      double y = 8.2;
8      double avg = calcAverage(x, y);
9      return avg;
10 }
```

User-Defined Functions

- Here the simple function calcAverage is a user-defined function that takes two doubles, and returns a double.
- The arguments to the function, x and y, are **passed by value**.
- Note that if we change x or y in the calculation of calcAverage, the values of x and y in the main function will not be affected.

Passing By Reference vs. By Value

- In C++ we can pass parameters to functions either **by value** or **by reference**.
- In the previous slide we saw an example of passing by value. If we add a & to the argument type in the function call, then the function will pass by reference.
- When we pass by value, we make a deep copy of the object that we pass into the function.
- Because we are working with a copy, if we modify the object in the function it does not affect the object being passed in.
- The copy operation contains some overhead as we need to create a new object to pass to the function, so this method is less efficient for large objects.

Passing By Reference vs. By Value

- If we pass by reference, then we are passing the memory location of the underlying variable.
- In this case if we modify the object it will impact the variable outside the function call, as they both reference the same object.
- We will also not have to deal with overhead of creating a new object, because they both reference the same underlying object.
- It is best practice to avoid passing arguments by value in most circumstances.

Passing By Reference vs. By Value: Example

```
1  double calcAverageVal(double x, double y){
2      x = 4.0;
3      return 0.5 * (x + y);
4  }
5  double calcAverageRef(double& x, double& y){
6      x = 4.0;
7      return 0.5 * (x + y);
8  }
9  int main(int argc, const char * argv[]) {
10     double x = 5.5;
11     double y = 8.2;
12     double avgRef = calcAverageRef(x, y);
13     double avgVal = calcAverageVal(x, y);
14     return 0.0;
15 }
```

const Function Parameters

- In some cases, we may want to ensure that the contents of an object or variable are unchanged but may not want the overhead and lost efficiency of passing by value.
- To accomplish this, we can use the **const** keyword in front of our parameters:

```
1 double calcAverageConstRef(const double& x,
2                             const double& y){
3     //x = 4.0;
4     return 0.5 * (x + y);
5 }
```

- If we use the const keyword, then the code will not compile if try to modify a parameter marked as const (for example if we uncomment line 3)

const Function Parameters

- This method for passing a parameter is called **passing by const reference**.
- It is best practice to pass parameters by const reference whenever possible.
- This lets the user of the function know that its arguments aren't going to be modified.
- It also lets the creator of the function know if it is unintentionally modifying an argument.
- It also leads to high performance code because we don't need to copy any objects unnecessarily.
- The const keyword can also be applied to variables, in which case the variable cannpt be modified after being initialized.

Arrays

- Array's are the simplest data structure in C++.
- Array's store strongly typed data.
- An array is a series of elements stored next to each other in memory that be accessed individually using the [] operator.
- Array's are relatively raw structures with some limitations and need to be used with caution.
- In particular, as we will see, if we try to access an element beyond the length of an array it leads to unstable results and unpredictable behavior.

Arrays

- In C++ we can create an array by adding a [] to the end of the variable declaration, for example

```
1 int intArr[] = {};
```

  would create an empty but initialized array of integers, of length 0.

- We can also create an array with a specified length:

```
1 int intArr[10];
```

  This would still create an empty but uninitialized array, but this time of length 10.

- Note that this is not the same as creating an array of length 10 with all values set to 0 or some other default.

Dangers of Arrays: Example

- As mentioned, if we try to access a value outside the bounds of an array it will lead to undefined behavior that differs depending on OS, etc.

```
1 int intArr [] = { 1, 2, 3};
2 int val = intArr[3];
```

- Despite the fact that the behavior may vary, it is safe to say that the result will be useless.
- Potentially even more problematic is when we write beyond the bounds of an array:

```
1 int intArr [] = { 1, 2, 3};
2 intArr[3] = 0;
```

- This will write to a memory allocation beyond the array (and referencing who knows what)
- This is clearly a very dangerous proposition and must be safeguarded against.
- Making this more challenging is that often times these errors won't be detected at compile time, meaning you must handle them at run time.

Using Arrays with Functions

- In C++ an array can be passed as an argument to a function.
- We cannot however create a function that returns an array.
- If we want to return an array from a function, we would have to return a pointer to the array.
- C++ has other data structures that we will discuss, such as std::vectors, which do not suffer from these problems and as a result are generally preferable.

Working with Pointers

- A pointer is a variable whose value is a reference or memory location of another object or variable.
- It is analogous to an argument that is passed by reference in that both refer to a memory location or address of an object.
- A pointer is created using a * after the type in a variable declaration, for example:

```
1 int val = 5;
2 int* ptr = &val;
```

- The value within ptr updates anytime the variable val changes.

Working with Pointers

- In complex systems, we can have many pointers referencing the same object which will all update concurrently.
- We need to make sure that pointers are removed once we are done with them to avoid dangling pointers.
- For example, if an underlying object is deleted, then we should make sure all pointers that reference its location in memory are deleted as well.

Passing Pointers to Functions

- Previously we talked about passing parameters to a function either by reference (or const reference) or by value. Pointers provide us with another option.

```
1  double calcAverageVal(double x, double y){
2      x = 4.0;
3      return 0.5 * (x + y);
4  }
5  double calcAverageRef(double& x, double& y){
6      x = 4.0;
7      return 0.5 * (x + y);
8  }
9  double calcAveragePtr(double* x, double* y){
10     *x = 4.0;
11     return 0.5 * (*x + *y);
12 }
```

```
13
15 int main(int argc, const char * argv[]) {
16     double x = 5.5;
17     double y = 8.2;
18     double avgRef = calcAverageRef(x, y);
19     double avgVal = calcAverageVal(x, y);
20     double avgPtr = calcAveragePtr(&x, &y);
21     return 0.0;
22 }
```

- Notice that in the call to the pointer based function, we needed to pass the memory address of the variable, which we did via &.
- Also notice that in the body of the pointer based function we need to **dereference** the pointer in access its value.
- Lastly, notice that this code behaves similarly to the pass by reference function (but not the const reference example) in that we can modify the contents of the underlying variable.

Passing Pointers vs. Passing by Reference

- Passing a pointer to a function is fundamentally similar to passing by reference (without the const keyword). (Why?)
- There are a few notable differences to keep in mind:
  - Pointers can be set to **NULL** whereas references cannot.
  - Pointers can be **reassigned** to a different object, whereas a reference cannot.
- Just like with passing by reference, we can use the const keyword to make sure that either the pointer is unchanged, or the underlying data is unchanged.

Memory Allocation

- In C++ memory is stored on the **stack** and the **heap**.
- The stack stores temporary variables and is very fast. All variables are local and when they go out of scope they are eliminated from the stack automatically.
- The heap is a separate region where we can allocate memory that has global scope. In the heap we need to allocate and release the memory ourselves, which we can do using the **new** and **delete** commands
- This memory allocation paradigm is also true in other programming languages, however, in other languages the allocation of memory is almost always abstracted from the user.

Freeing unused Memory

- If we allocate memory on the heap using the **new** keyword we need to make sure and free the memory when the object is done being used.
- If we don't do this, our code will have a **memory leak** and will be extremely inefficient.
- That is, we will have memory allocated that we are not using, which is wasteful.

Pointer Example: Using the new and delete commands

```
1 int main(int argc, const char * argv[]) {
2 std::vector<int>* vecIntPtr(new std::vector<int>());
3 vecIntPtr->push_back(0.0);
4 vecIntPtr->push_back(5.0);
5
6 delete vecIntPtr;
7
8 return 0.0;
9 }
```

Smart Pointers

- In practice I always use some version of a **smart pointer** instead of raw pointers.
- These types of pointers combine the benefits of using pointers, without the potential dangers and performance issues associated with using raw pointers incorrectly.
- One example of this is the **shared_ptr** class defined in the **boost** library.
- Smart pointer classes provide similar functionality, but do the appropriate cleanup for us as things go out of scope and get deleted.

std::vectors

- std::vector provides another data structure in C++ that is functionally similar.
- The vector class is strongly typed, like the array class.
- The following code creates a variable that is a vector of integers:

```
1 std::vector<int> vecInt;
```

- Notice the different syntax for creating a vector. We could replace int with any other type of our choice, user-defined or built-in.
- std::vector is an example of a **template** class. We will discuss templates in more detail next week.

std::vectors

- std::vectors provide some additional functionality and safety checks that are not present in arrays.
- Because of this, in practice, I would recommend using vectors rather than arrays in your coding.
- Of particular use are the **push_back()** and **at()** functions.
- std::vectors has a set of constructors that enables us to create an vector of a given length with all elements initialized to a set value.
- They also provide **begin()** and **end()** functions which enable us to iterate through vectors in a safe manner, using an **iterator**
- std::vectors can be multi-dimensional, that is, they can be nested.

Safety of Vectors

- As we saw in the case of an array, using the [] operator to access an element can be dangerous if our code tries to look or write beyond the length of the array.
- std::vectors handle this by implementing the **at()** function which behaves the same way as the [] operator but has additional checks to ensure this case is handled safely (it throw an exception rather than produce undefined results)
- Similarly, the **push_back()** gives us a safe method for adding a value to a vector at the end of the vector and dynamically increasing the length of the vector.

Safety of Vectors: Example

```
1 std::vector<int> vecInt(5, 100);
2 int tmp = vecInt.at(10);
3
4 std::cout << vecInt.size() << std::endl;
5
6 vecInt.push_back(200);
7
8 std::cout << vecInt.size() << std::endl;
```

Line 2 of this code will produce an std::out_of_range: vector exception rather than return a useless value.

Most of the time this still likely has to be caught at run-time (rather than compile-time), but it is preferable to an undefined result that is difficult for users to handle.

std::maps

- std::maps are another data structure within the STL library (and std namespace).
- They are most similar to Dictionaries in Python in that the contain a series of key-value **pairs**
- Pair is another useful class within the STL library.
- std::maps are strongly typed template classes.

  The key in a map can be any basic type. It can also be any complex type or useful defined type as long as certain operators are implemented.

  The value class in a map can be any type.
- maps can also be nested, and the key of a map can be a **pair**.

std::maps

- A new pair can be added to a map using the **insert()** function
- As with std::vector, the **at()** function is defined to access an element by key.
- The behavior of the [] operator is a bit unintuitive. It searches for an element by key as the at() function does, and if it is not found a new item is inserted into the map.
- Just like with std::vector, the **begin()** and **end()** functions help us iterate through our map via an **iterator**
- std::maps are useful to store mapping tables, as we saw with Dictionaries in Python.

Iterators

- An iterator is an object that points to an element in a range of elements.
- Iterators are themselves classes, and there are different types of iterators.
- Some are const, meaning they don't allow the user to modify the underlying data elements.
- Some iterators iterate from the beginning of an object or container to the end and others iterate in reverse.
- Iterators define how to safely traverse a container or data structure without trying to continue past the end.

Iterator Example: std::vector

```
1 std::vector<int> vecInt(5, 100);
2
3 for (std::vector<int>::const_iterator vit = vecInt.begin();
4              vit != vecInt.end();
5              vit++){
6     std::cout << *vit << std::endl;
7 }
```

Notice that because iterators are pointers we needed to dereference it in order to obtain the value to print to the console.

Useful External Libraries

- **Boost**: An opensource library with some basic math libraries (such as Linear Algebra calculations) and also some helpful classes. Most production system rely on boost.
- Of particular use is the shared_ptr class in boost that provides the functionality of pointers that is conveniently wrapped to ensure that pointers are destroyed at the appropriate time.

  More info on boost can be found here Boost Docs
- **QuantLib**: An opensource Quant Finance library available in many languages including C++. Many firms use this as a basis for their own proprietary analytics libraries so they don't have to recreate

  More info on QuantLib can be found here QuantLib Docs