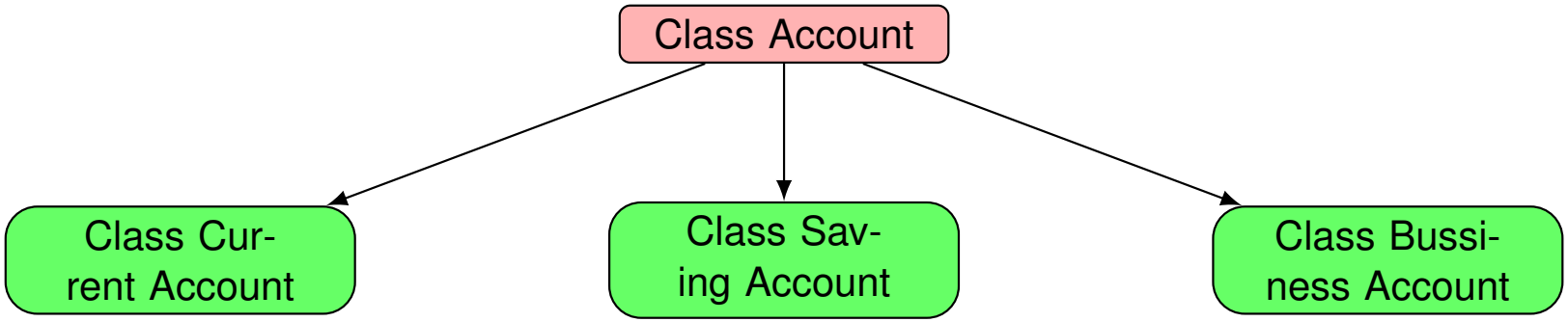
C++ fundamentals with use cases from finance

Lecture 4: Creating optimal code and Standard Library

Ivan Zhdankin

References and Inheritance



- Imagine we have different type of accounts (derived classes) inherited from the Account (base class)
- The reference can be of base class and can actually refer to the derived class instance

```
Account & acc = CurrentAccount;
```

- The class *CurrentAccount* inherits from *Account* has all the capabilities of *Account*
- Any base class function can be called through a base reference to a derived class instance

Virtual Function

- However if there are same functions implemented in both a base and a derived classes which function will be executed?:
  - If the function in a base class is marked as *virtual* - the derived class function will be executed
  - This is also known as *polymorphism* - as a function as several forms
  - If the function is NOT *virtual* - the base class function will be executed
- Virtual function - is *dynamic polymorphism* as which function to call is resolved in runtime (some times this is called run-time polymorphism)
- Virtual has impact on the run-time performance
- We can NOT call functions of the derived class by the base class reference which target is derived class instance:

```
acc.DerivedClassFunction();
```

- We can NOT create a derived class reference that refers to a base class instance:

```
CurrentAccount & acc = Account;
```

Pointers and Inheritance

- Situation with pointers and inheritance is similar to the reference and inheritance
- A pointer to a base class can actually point to a derived class instance:

```
Account * pt_acc = &CurrentAccount;
```

- Any base class function can be called using the pointer:
  - If the function in a base class is marked as *virtual* - the derived class function will be executed
  - If the function is NOT *virtual* - the base class function will be executed
- We can NOT call functions of the derived class by the pointer of base class which target is derived class instance:

```
pt_acc->DerivedClassFunction();
```

- We can NOT have pointer of the derived class pointing to the base class instance:

```
CurrentAccount * pt_acc = Account;
```

- Demo: Indirections and Inheritance

Slicing problem

- There are many advantages of using the inheritance and virtual functions as they enable to write generic code
- However we need to be careful as there is a *slicing* problem
- If we copy objects around a slicing problem can occur:
  - When copying a derived object into a base object extra member variable fall away
- For example, same rules applies when passing to a function by value

```
double report_fees(BankAccount acc)
{return acc.report_fees();}
```

- If function takes as parameter a bank account and we pass by value a saving account - a copy will be made and slicing will happen - we will have bank account inside the function, not a saving account
- To avoid the slicing we use pointers and references
- For that reason we pass by reference into functions

Casting

- Often when we have **polymorphism** we have a base class pointer but we know that in reality points to a derived class instance
- For example at some point of the program you know that the pointer points to the *business account* class which is derived from the *bank account*, however the pointer is of *bank account* class
- We would like the base class pointer to be a derived class pointer as this would allow us to access the derived class methods
- For these reasons we have *casting* in C++

Static Casting

- *Static_cast* $<$ *type* $>$
- The static cast happens in compile time
- Use only if the pointer is of derived class
- Does not require virtual functions
- Results in ”Compiler Error” if the casting fails
- Because of there are no any checks the *static casting* is faster than the *static casting*

- Demo: Casting

Dynamic Casting

- *Dynamic_cast* $<$ *type* $>$
- Happens in run time
- Because of the run-time checks it is safer
- Works when there is at least one virtual function
- Returns ”null” if the casting fails
- Because of the run-time checks the *dynamic casting* is slower but safer then the *static casting*

Map

- A collection that is organised in pairs
- Set of pairs: key-value, where we can lookup a values by the corresponding key

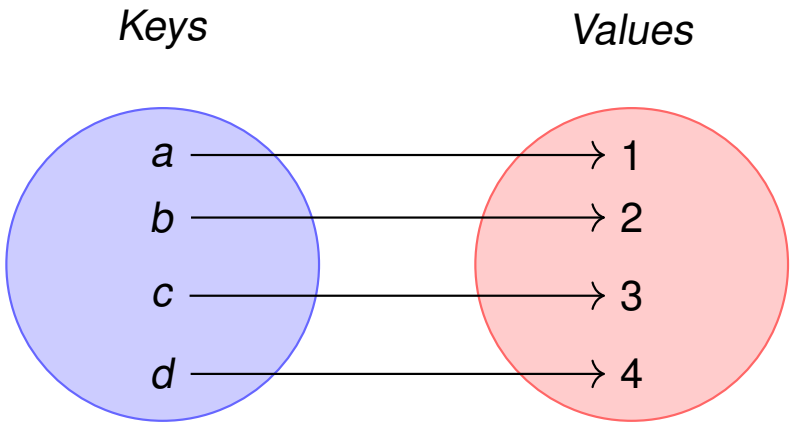
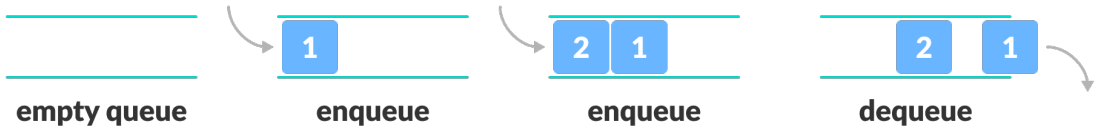


Figure: Map with $< Key, Value >$ pairs

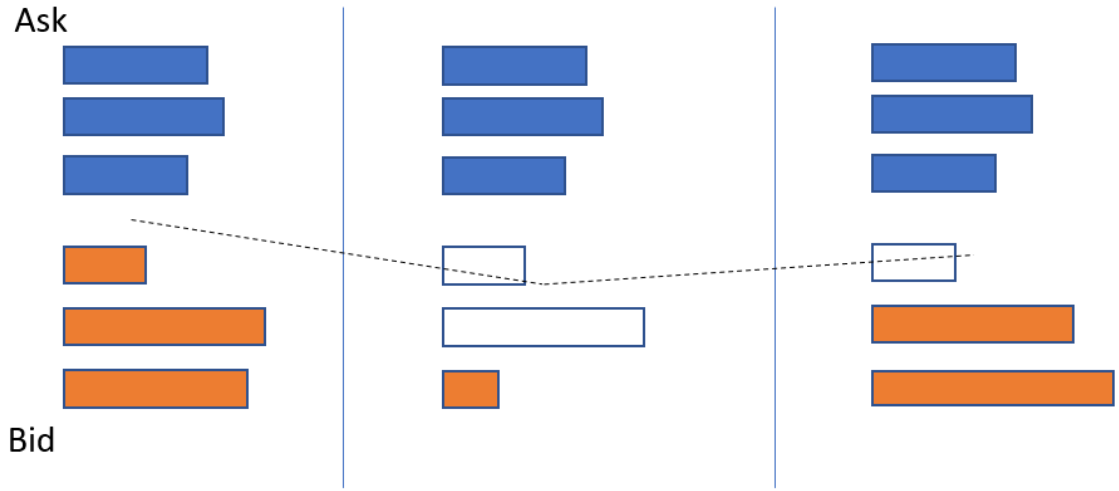
- There are can be one value for each key
- It is dynamically adjusted if we need to add/remove a pair
- A map keeps the pairs sorted internally to speed searching
- To add or access items $[]$ can be used
- There are different methods implemented in map to help developers including $find()$
- To use maps we need to $\#include < map >$
- Demo: maps

Queues

- Another fundamental data structure is **queue** - a container of objects that are inserted and removed according to first-in-first-out principle (FIFO)
- We say that elements are inserted in the **rear** and removed from the **front**
- To **enqueue** means inserting an element in the **rear**; to **dequeue** means removing an element from the **front**



- Examples of using queues data structure include:
  - Storing orders in LOB (limit order book):



  - When a resource is shared among multiple consumers
  - When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes

Collections in Standard Library

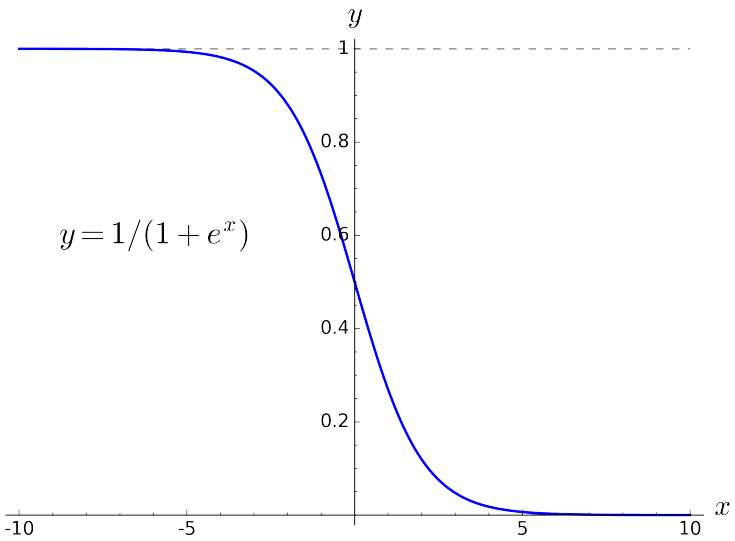
- So far we have seen different collections in SL including stings, vectors and maps
- There are other collections in C++ SL
- List: implemented as linked list
  - Faster adding the element
  - Slower than vector for accessing
- However all the collections are similar in use: all of them contain similar methods and iterators
- Other examples of collections:
  - Queues, dequeue: push and pop from one and both end, FIFO
  - Priority_queue - has a priority inside a queue
  - Set - contains unique element
  - Multimap - a map which can have more then one value for one key

Sorting and Searching in SL

- So far we have done iteration through different elements of the collections
- There also sorting and searching algorithms in C++
- These functions are implemented as free functions that take a collection as argument rather than method of the collection class
- We use same functions for any collections: vector, map, list
- To use the functions in need to include $\#include < algorithm >$
- Demo: Algorithms

Finding the price given Probability To Trade

- In RFQ based protocol dealers produce quotes replying ot the clients requests
- The dealers' pricing logic may be based on the "willingness" to trade which is measured as probability to trade
- When dealers quote price to the client the following relationship hold between Spread and Probability To trade:



- Binary Search is used to to find a spread and produce the price given probability to trade

More on SL

- Date and time:
  - $\#include < chrono >$
  - $\#include < ctime >$
- Complex numbers:
  - $\#include < complex >$
- Matrix math:
  - $\#include < numeric >$
- Random number generator:
  - $\#include < random >$
- Math: abs, rounds, sqrt, pow, sin
  - $\#include < cmath >$

Lambda Functions

- Lambda is an expression that represents doing something, performing some operation and calculation
- When using lambdas we handle the function to some other operation or function
- Lambda enables:
  - generic programming
  - functional programming
  - readability of the program while eliminating tiny functions
- In the tiny function implementation of which can be hidden far away in the code we can use Lambda function

```
void print(int i)
{
cout << i << endl;
};
```

- At the same time for_each operator will be more readable with lambda function

```
for_each(v.begin(), v.end(), print);
```

- Lambdas do not just do operations, they can return values like normal functions:
  - The compiler can specify the return type of Lambda
  - Normally the developer should specify it as with normal functions
  - However the syntax for specifying the return type is different
- Demo: Lambda

Exceptions

- Errors and failures happen
- Probably the largest part of programming is reacting to the errors and understanding bugs
- Some errors are predictable:
  - Non-integer entered in the field
  - The withdrawal amount is more than available
- Some are not:
  - Out of memory
  - Result too big for the integer
  - File is not found
  - Access denied
  - Division by zero
- There should be balance between what should be checked and what is not because the check slow down your application
- When you expect the errors or *exceptions* happen you can transfer the flow of execution from the problem location to the place where the problem can be handled
- The rule is that to handle the exceptions as close to it is source as possible
- Demo: Exceptions

Try and Catch rules

- Any block of code that might throw an exception should be wrapped by *try* block
- Catch blocks should follow the try block
- Try and catch block should be as close to the problem as possible
- Do not use try and catch when you do not expect potential issues
- Catch more specific exceptions first
- Catch exceptions by reference