Practical Session #04 - C++ Classes Cont.

Tom Mahler
In This Recitation We’ll Cover

- The Rule of 5
- Smart Pointers
- Inheritance
- Virtual Functions
- Namespace
The Rule of 5
The Rule of 5

- The rule of 3 suffers from inherent problems

- Therefore, starting from C++11 two addition functions were added, which aim to solve these problems:
  - The move constructor
  - The move assignment operator.
The Rule of 5 - Summary

- In total, the thumb rule states that the following methods need to be implemented:
  - Destructor
  - Copy constructor
  - Move constructor
  - Copy assignment operator
  - Move assignment operator
The Rule of 5 - Remarks

- Holding an object without holding a copy of it:
  - Sometimes we don't want to hold a copy of a certain object in the field of a class, but still hold the object
  - This can be done by declaring the field as a reference
  - However, in this case we must remember to initialize it in the constructor using initialization list
The Rule of 0

- The rule of 5 and the rule of 3 implies only for classes that manage dynamic resources, such as pointers.
- In general, we would usually try to avoid such classes and use classes that use only primitive or static fields.
- Such classes are not considered as classes that manage resources and therefore does not require an implementation based on the rule of 5 or the rule of 3.
- The rule of 0 states that classes that does not manage dynamic resources (pointers) should not implement any of the methods mentioned in the rule of 5.
The Rule of 5 Example - The Destructor

- The function clear():
  - A private function that releases the memory, similar to the destructor we built in PS3.
  - This is done as a preparation for the move and copy assignment operators that also need to perform a clearing operation.

- Note that the last line `cstringlen = 0;` is not actually required in the destructor, however it is done to maintain the correctness of our object:
  - If its cstring is null, not changing its length to 0 may lead to logical mistakes later on.
Examples

» See code...
The Rule of 5 Example - The Move Constructor

- The move constructor is typically called when an object is initialized from rvalue.
- This is done using rvalue references that we've seen in class.

- This usually happens in the following cases:
  - Initialization, `std::string a = "str"`
  - Function argument passing: `v.push_back("str")`
  - Function return by value
Examples

- See code...
The Rule of 5 Example - The Move Constructor

- Move constructors typically "steal" the resources held by the argument (e.g. pointers to dynamically-allocated objects, etc.), rather than make copies of them, and leave the argument in some valid but otherwise indeterminate state.

- For example:
  - Moving from a `std::string` or from a `std::vector` may result in the argument being left empty.
The Rule of 5 Example - The Move Constructor

- However, this behavior should not be relied upon
- For some types, such as std::unique_ptr, the moved-from state is fully specified

- For example, in the following line:
  - v.push_back(charArray("I will be moved, then destroyed"));
The Rule of 5 Example - The Move Constructor

The following actions will happen:

- A new object will be created with the string "I will be moved, then destroyed"
- The move constructor will create a new object inside the vector and will move the created object to it
- The pointer that was created in the first object will become nullptr and the object inside the vector will now hold the first pointer
- The object that was created outside of the vector will be destroyed
  - Since its cstring pointer was null, the destruction will not effect the object inside the vector
The Rule of 5 Example - The Move Assignment

- The move assignment operator is built using the same steps we've seen in the copy assignment operator.

- The only difference is that the move assignment operator doesn't copy the values, and instead only moves the ownership of its pointers from one object to another.
Examples

▶ See code...
Smart Pointers
Smart Pointers

- In modern C++ programming, the Standard Library includes smart pointers, which are used to help ensure that programs are free of memory and resource leaks and are exception-safe

- We will examine one of them, the `std::unique_ptr`
**std::unique_ptr**

- This smart pointers provides a limited garbage-collection facility, with little to no overhead over built-in pointers
  - Objects of this kind "steals" the ownership of a pointer
  - Once ownership is obtained, they manage the pointed object by becoming responsible for its deletion at some point

- `unique_ptr` objects automatically delete the object they manage, as soon as:
  - They themselves are destroyed
  - Their value changes either by:
    - An assignment operation
    - An explicit call to `unique_ptr::reset`
std::unique_ptr

- unique_ptr objects own their pointer uniquely:
  - No other facility shall take care of deleting the object
  - Thus no other managed pointer should point to its managed object

- unique_ptr objects delete their managed object without taking into account whether other pointers still point to the same object or not
  - Thus leaving any other pointers that point there as pointing to an invalid location
std::unique_ptr

- unique_ptr is implemented using two components:
  - **A stored pointer:** the pointer to the object it manages
  - **A stored deleter:** a callable object that takes an argument of the same type as the stored pointer and is called to delete the managed object
Examples

See code...
Inheritance

Inheritance is a relationship among classes, in which one class shares the structure or behavior defined in other class(es).

In object-oriented programming (OOP), inheritance is a way to reuse code by creating collections of attributes and behaviors called objects which can be based on previously created objects.
Inheritance

- In C++ a class can be defined by means of an older, pre-existing class, which leads to a situation in which a new class has all the functionality of the older class, and additionally introduces its own specific functionality.
  - The new class inherits the functionality of an existing class, while the existing class does not appear as a data member in the definition of the new class.

- When speaking of inheritance the existing class is called the base class, while the new class is called the derived class.
Examples

See code...
Inheritance

- In C++ class can be defined by means of an older, pre-existing class, which leads to a situation in which a new class has all the functionality of the older class, and additionally introduces its own specific functionality.
  - The new class inherits the functionality of an existing class, while the existing class does not appear as a data member in the definition of the new class.

- When speaking of inheritance the existing class is called the base class, while the new class is called the derived class.
Examples

See code...
Inheritance

- The relationship is better achieved with inheritance:
  - `intSortedArray` is derived from `intArray`, in which `intArray` is the base class of the derivation
  - By post-fixing the class name `intSortedArray` in its definition by public `intArray` the derivation is defined:
    - The class `intSortedArray` now contains all the functionality of its base class `intArray` plus its own specific information

- The extra functionality consists here of a CTORs and interface functions to access the biggest item in array
Public Inheritance Example

- In C++ there are 3 types of inheritance

- In the example previous, we used public inheritance

- In C++ we can also use a protected/private inheritance using: private or : protected instead of : public, however this is generally not recommended (we will not elaborate on that)
Examples

- See code...
This example shows three features of derivation:

- `empty()` - is no direct member of a `intSortedArray`
  - This member function is an implicit part of the class, inherited from its "parent" class
- `biggest()` - derived class `intSortedArray` now adds to functionality of `IntArray`
- `insert()` - is overloaded in the class `intSortedArray`
In C++ we can declare a variable or a function that seems private to outside users of a class but seems public to classes that inherit the class.

To do so, we use the keyword protected, just like we used the keyword public above to indicate this behavior.
Protected Inheritance Example

For example

Say that the `intSortedArray` class needed direct access to the `arrSize` variable in `intArray` in order to improve its performance, but we still want to keep normal instances of `intArray` and `intSortedArray` from accessing the `arrSize` directly.

By declaring `arrSize` as a `protected` member in `intArray`, it would be accessible by the derived class `intSortedArray` but not by normal instances of `intArray` or `intSortedArray`
Examples

See code...
Private Inheritance Example

- Private inheritance makes all of the public functions of the parent class private in the child class.

- This means that they can be used in order to implement the child class without being accessible to the outside world.

- Unlike a public inheritance, a private inheritance defines a "has-a" relation with its parent, and not an "is-a" relation like public inheritance.
Examples

► See code...
The CTOR Of A Derived Class

- A derived class inherits the functionality of its base class

- How effects the inheritance on the CTOR of a derived class?
The CTOR Of A Derived Class

As can be seen from the definition of the class intSortedArray, a CTOR exists to set the biggestElem of an object

The implementations of this CTORs could be:

```cpp
intSortedArray::intSortedArray() :
biggestElem(std::numeric_limits<int>::min()) { }
```
The CTOR Of A Derived Class: Memory Layout Of Derived Class

- Memory layout of derived class:
  
  | ***************** |
  | Base part       |
  | ***************** |
  | Derived part    |
  | ***************** |

- The C++ compiler will generate code to call the default CTOR of a base class from each CTOR in the derived class, unless explicitly instructed otherwise
The CTOR Of A Derived Class: Calling CTOR Of The Base Class

- The better solution is of course to directly call the CTOR of `intArray` which expects an `int` argument.

- The syntax to achieve this, is to place the CTOR to be called (supplied with an argument) following the argument list of the CTOR of the derived class:

```cpp
intSortedArray::intSortedArray(const int &size) : intArray(size), biggestElem(std::numeric_limits<int>::min()) {}
```
Examples

See code...
Virtual Functions
Virtual Functions

- Virtual functions are member functions whose behavior can be overridden in derived classes.

- As opposed to non-virtual functions, the overridden behavior is preserved even if there is no compile-time information about the actual type of the class.

- The access of virtual member functions is determined by its declaration in the base class can’t be changed in derived class.
Examples

See code...
Virtual Functions

- This means that overridden functions can be put in any place in an interface of derived class.

- The only reason that we put them in the same access block is to be consistence with base class which shows us the real access of member function.
Examples

See code...
Virtual Functions: Virtual Destructor

- Without a garbage collector, polymorphism can be problematic when attempting to release memory

- This is best explained with an example

- We first look at what happens when not using virtual destructor
Examples

- See code...
Virtual Functions: Virtual Destructor

Constructing Base
Constructing Derive
Destroying Base

- Since we told the compiler that `basePtr` is a pointer of type `Base`, the destructor of the `Base` class was used.

- This means that if `Derive` allocated memory, it would not have been deleted (since the destructor of `Derive` was not called) and we would have a memory leak.
Virtual Functions: Virtual Destructor

Now lets take a look at what happens when we use a virtual destructor

We now change the destructor's definition in the Base class to be virtual:

```cpp
virtual ~Base() { cout << "Destroying Base"; }
```

And repeat the same function
Examples

See code...
Virtual Functions: Virtual Destructor

- The output will now be:

  Constructing Base
  Constructing Derive
  Destroying Derive
  Destroying Base
Virtual Functions: Virtual Destructor

► Now, when the compiler see that we assigned a Derive class to a Base class with a virtual destructor, it knows that in order to delete this pointer, we need to call the destructor of Derive before calling the destructor of Base.

► The best way to avoid such cases is to use virtual destructor whenever possible.
Virtual Functions: Pure Virtual Functions & Abstract Base Classes

- Let’s say we want to create a set of classes that implement different kinds of shapes (circle, rectangle ...)

- The functionality of them is similar

- We’ll not want to rewrite the same code many times and we would like to create some class that will contain common code but would be impossible to make instantiation of such class
Virtual Functions: Pure Virtual Functions & Abstract Base Classes

- Example is class "shape" which will be a base class of all kinds of real shapes as circle, ellipse, rectangle etc.

- In C++ this is possible to do by declaration of one or more member functions as **pure virtual**
Examples

See code...
Virtual Functions: Pure Virtual Functions & Abstract Base Classes

- Until now the base class shape contained its own, concrete, implementations of the virtual functions `draw()`

- In C++ it is also possible only to mention virtual functions in a base class, and not define them.

- The functions are concretely implemented in a derived class.
Virtual Functions: Pure Virtual Functions & Abstract Base Classes

- The special feature of only declaring functions in a base class, and not defining them, is that derived classes must take care of the actual definition:
  - C++ compiler will not allow the definition of an object of a class which doesn't concretely define the function
Virtual Functions: Pure Virtual Functions & Abstract Base Classes

- The base class enforces a protocol by declaring a function prototype, but the derived classes must care of the actual implementation.

- The base class itself is therefore only a model, to be used for the derivation of other classes.

- Such base classes are also called **abstract classes**.
Virtual Functions: Pure Virtual Functions & Abstract Base Classes

- A function is made pure virtual by preceding its declaration with the keyword virtual and by post-fixing it with =0

- The functions are called pure virtual functions
Examples

- See code...
Namespace
Namespace

- C++ namespace is similar to Java packages only in the sense that it helps identifying names

- The C++ compiler will not create a folder hierarchy of the namespace
Examples

See code...
Splitting to H and CPP files

See code...