CH-230-A

Programming in C and C++

C/C++

Lecture 13

Dr. Kinga Lipskoch

Fall 2020

Abstract Classes (1)

▶ It should be evident that classes near to the root of the hierarchy are seldom instantiated
  ▶ Very general but also very unspecialized
▶ Some classes are introduced just to define common behaviors, but are not self sufficient
  ▶ Think of the class shape in one of the former examples
▶ Those classes are useful only for abstraction

Abstract Classes (2)

- ▶ Abstract classes define a set of methods to be shared by a derived class but are not yet implemented
  - ▶ Implementation will be defined in a derived class
    - ▶ Virtual mechanism plays a fundamental role
- ▶ A pure virtual method is a method declared as:
  `virtual void something() = 0;`
- ▶ A class having one or more pure virtual methods is abstract

Abstract Classes (3)

- ▶ Abstract classes cannot be instantiated
- ▶ Abstract classes can also include non-pure virtual methods
- ▶ Methods and functions can accept pointers to abstract classes
  - ▶ This is their main use: through virtual calls generic code is developed

Shapes Example Revised

- ▶ In the shape example the shape class has not actually represented a shape (instance), but rather collected some data common to all shapes
- ▶ Therefore, Shape is a good candidate to be an abstract class
    - ▶ shapesrevised.h
    - ▶ shapesrevised.cpp
    - ▶ testshapesrevised.cpp

Virtual Destructors?

Destructors are almost always `virtual`

- ▶ If you are manipulating objects via pointers to the base class, then the base class should define its destructor as `virtual`
- ▶ Otherwise just the base class destructor is called
- ▶ Recall that destructors are called from bottom to up
- ▶ Destructors can be pure `virtual`
  - ▶ There are some subtle details concerning this aspect (see Eckel's book, chapter 15)

Virtual Constructors?

- ▶ You cannot have virtual constructors
    - ▶ Remember that constructors are called from the base to the leaves of the derivation tree
- ▶ Inside a constructor you can call a virtual method, but this will execute the local version
    - ▶ No downsearch is performed, as the assembly of the object is still being performed and elements belonging to derived classes are not guaranteed to be properly initialized

Overloading Operators for Casting 💬

- ▶ It is possible to create operators for converting a type to another, thus performing a sort of casting
  operatorconversion.cpp
- ▶ This can also be done by implementing an ad-hoc constructor taking the type we want convert from
  constructorconversion.cpp

The explicit Keyword

▶ If a constructor is declared with the explicit modifier, it will be used for type conversion only if the typename is explicitly inserted

▶ Then it is possible to choose which kind of conversion will take place: constructor driven or operator driven
explicitconversion.cpp

Streams

- ▶ A stream is a flow of data from a source to a destination
    - ▶ Widely used concept in Unix
    - ▶ Think to water flowing in a pipe
- ▶ Standard C++ provides classes for handling streams of data connected to the console or to files
    - ▶ Common interface: learn once use everywhere

Iostreams

- ▶ You already used them
- ▶ The instances cin, cout and cerr are declared in the header files included in <iostream>
- ▶ Exceptional use due to their wide use
  - ▶ Preprocessor directives for conditional compiling avoid multi-declaration problems
- ▶ Extractors and inserters are overloaded operators designed to work with different data types
  - ▶ Consider to overload them to work with your own developed classes

Class Hierarchy
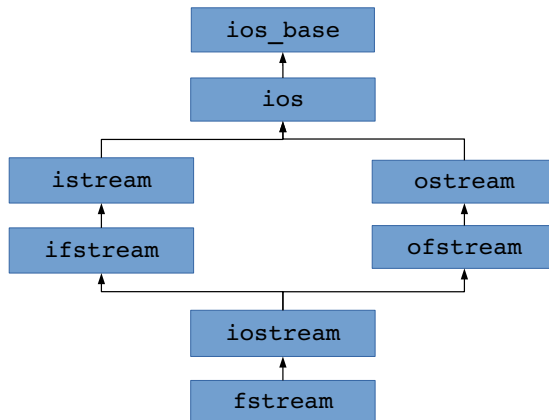
Output Streams and the Inserter Operator <<

▶ Operator << has been overloaded to work with all language data types and many classes
   ▶ It sends data to an output stream (ostream)
▶ Inserters can be concatenated
▶ Additionally, manipulators can modify the output
   ▶ endl, flush, hex, oct, dec
     Example: cout << hex << "0x" << 34 << flush;

The << Operator

- ▶ Converts internal data type into sequence of ASCII characters
  ostream& operator<<(const char *)
  ostream& operator<<(char)
  ostream& operator<<(int)
  ostream& operator<<(float)
  ostream& operator<<(double)
- ▶ Returns reference to ostream

Input Streams and the Extractor Operator >>

- ▶ The operator >> has been overloaded to work with predefined language data types
  - ▶ It gets data from an input stream (istream)
- ▶ Extractor stops reading when it finds a whitespace
- ▶ The manipulator `ws` removes leading and trailing white space from an istream

```
cout << "First Name: ";
cin >> ws;
cin.getline(FirstName, 20);
```

Line Oriented Input

▶ Istreams provide two methods to get a whole line of text:
  ▶ get() get the text but do not remove the delimiter
  ▶ getline() get the text and remove the delimiter
  ▶ Both accept three parameters: char buffer to store data, buffersize and terminator character
  ▶ Default value of terminator is '\n'

▶ It can be useful to grab input as a char sequence and then convert it using C functions

Raw I/O

- ▶ Binary files: images, audio, self-defined formats, etc.
- ▶ Raw I/O member functions are used to write/read binary data to/from streams
  - ▶ Istreams:
    - ▶ `read(char *, int)`
    - ▶ `gcount()` returns the number of characters extracted
  - ▶ Ostreams:
    - ▶ `write(char *, int)`

The State of a Stream

The following member functions can be used to investigate on the state of a stream:

- ▶ `good()`   true if goodbit is the current state
- ▶ `eof()`   true if endoffile
- ▶ `fail()`   true if failbit or badbit set
- ▶ `bad()`   true if badbit set
- ▶ `clear()`   set state to goodbit

File Streams

`ifstream` and `ofstream` classes can be used to connect a stream to a file

- ▶ Just provide the name of the file as a parameter to the constructor
- ▶ You do not need to open or close the file (up to constructor and destructor)
- ▶ Classes are declared in the `fstream` header file
- ▶ `filestream.cpp`

Open Mode Flags

Flag	Function
`ios::in`	Open as input
`ios::out`	Open as output
`ios::binary`	Open in binary mode
`ios::app`	Open for appending
`ios::ate`	Open and go at end
`ios::trunc`	Open and delete the old if present

Overloading Extractors and Inserters for your Types

▶ It can be useful to overload << and >> to dump and/or read classes instances to streams
  ▶ For example to save/retrieve the state of the application to/from a file
▶ Add an overloaded operator << or >> definition to the class
  ▶ Should be friend
  ▶ Returns an ostream/istream reference
  ▶ Should take an istream/ostream reference and a (const) reference to the class as parameters
  ▶ overloadedstream.cpp