

CH08-320143

Object-oriented Programming II

OOP II

Lecture 1 & 2

Dr. Kinga Lipskoch

Spring 2018

Who am I?

- ▶ PhD in Computer Science at the Carl von Ossietzky University of Oldenburg
- ▶ University lecturer at the Department of Computer Science
- ▶ Joined Jacobs University in January 2013
- ▶ Office: Research I, Room 94
- ▶ Telephone: +49 421 200-3148
- ▶ E-Mail: k.lipskoch@jacobs-university.de
- ▶ Office hours: Mondays 10:00 – 12:00

Course Goals

- ▶ Learn more advanced aspects of object-oriented programming
- ▶ Learn advanced details of the C++ programming language
- ▶ Write, test, debug programs
- ▶ “Hands on”: not just theory, but practice lab sessions to apply what you have learned in the lectures

Course Details

- ▶ 3 weeks = 24 hours
- ▶ Every week will consist of
 - ▶ 2 lectures: Thu/Fri afternoon, 14:15 – 16:15
 - ▶ 2 lab sessions: Thu/Fri afternoon, 16:15 – 18:30
- ▶ During each lab session you will have to solve a programming assignment sheet (consisting of multiple exercises) related to the corresponding lecture

Course Resources

Textbooks:

- ▶ Frank B. Brokken, C++ Annotations Version 10.7.2
<http://www.icce.rug.nl/documents/cplusplus/>
- ▶ Bruce Eckel, Thinking in C++: Introduction to Standard C++, second edition, volume 1, Prentice Hall, 2000
- ▶ Bruce Eckel, Chuck Allison: Thinking in C++: Practical Programming, volume 2, Prentice Hall, 2004

Slides, program assignments, and code will be posted on Grader

- ▶ https://grader.eecs.jacobs-university.de/courses/320143/2018_1r2/

Grading Policy

- ▶ 35% – average grade of the assignments
- ▶ 65% – grade of the final exam
- ▶ In the (written) final exam you will be asked to solve exercises similar to ones in the assignments.
- ▶ The final exam will take place at the end of the semester

Programming Assignments

- ▶ There will be presence assignments that need to be solved in the lab
- ▶ Other assignments are due on the following Tuesday and Wednesday morning at 10:00
- ▶ Solutions have to be submitted via web interface to <https://grader.eecs.jacobs-university.de>
- ▶ Assignments are graded by the TAs
- ▶ Grading criteria
https://grader.eecs.jacobs-university.de/courses/320143/2018_1r2/Grading-Criteria-C++.pdf

Lab Sessions

- ▶ TAs will be available to help you in case of problems
- ▶ Solve the assignments during lab sessions
- ▶ Do not copy the solutions for the assignments, you will certainly fail the written final exam without practice in programming

Syllabus of the Course

- ▶ More on Overloading
- ▶ Streams
- ▶ Multiple Inheritance
- ▶ Templates: Functions, Classes
- ▶ Standard Template Library (STL): Containers, Algorithms, Iterators
- ▶ Features of C++11
- ▶ Exception Handling
- ▶ Debugging Techniques
- ▶ Memory Leaks
- ▶ Unit Testing and Makefiles

Programming Environment (1)

- ▶ C++ is available on practically every operating system
- ▶ Commercial
 - ▶ Microsoft C++, Intel C++, Portland
- ▶ As well as free compilers available
 - ▶ g++
- ▶ g++ available on many platforms
- ▶ g++ 6.3.0 is used on Grader

Programming Environment (2)

- ▶ We will refer to the Unix operating system and related GNU tools (g++, gdb, some editor, IDE, etc.)
- ▶ Install a C++ compiler on your notebook
- ▶ IDE (Integrated Development Environment) may be helpful
- ▶ CodeLite <http://www.codelite.org/>
 - ▶ powerful IDE
 - ▶ runs on Linux, Mac, Windows

Programming Environment (3)

▶ Linux

- ▶ preferred environment
- ▶ prepare to get to know Unix/Linux
- ▶ g++ should be already on your machine
- ▶ you can install CodeLite from
<http://codelite.org/LiteEditor/Repositories>
- ▶ any other IDE or editor is also fine

Programming Environment (4)

- ▶ Mac OS
 - ▶ it is a Unix system as well
 - ▶ XCode (is on MacOS DVD)
 - ▶ if the DVD is not available you might need to register as Apple Developer to be able to download XCode
 - ▶ the rest you will need to figure out yourself
 - ▶ CodeLite <http://downloads.codelite.org/>

Programming Environment (5)

► Windows

- CodeLite includes a C++ compiler
<http://downloads.codelite.org/>
- choose: CodeLite Installer 10.0 for Windows
- download and install it
- includes gdb and g++
- Alternatives: Visual C++ Express (free of charge), Eclipse, NetBeans

Programming Environment (6)

- ▶ You will upload your solutions to Grader, where your source code will be automatically compiled
- ▶ Your programs must compile without any warning with `g++`
- ▶ On your local machine turn on all warnings:
`g++ -Wall -o executable file.cpp`

Agenda Week 1

- ▶ More on Overloading
- ▶ Streams
- ▶ Multiple Inheritance
- ▶ Templates: Functions, Classes
- ▶ Standard Template Library (STL): Containers, Iterators

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](#)

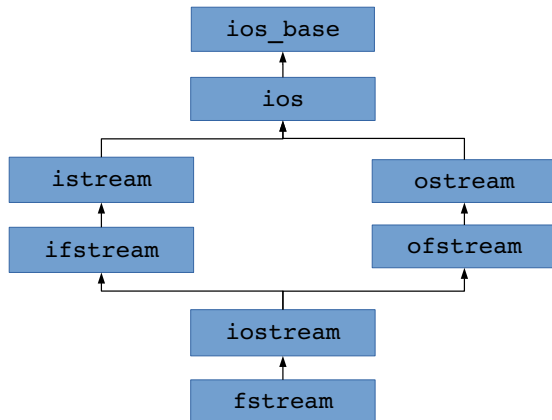
Streams

- ▶ 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`

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

Open Mode Flags

Flag	Function
<code>ios::in</code>	Open as input
<code>ios::out</code>	Open as output
<code>ios::binary</code>	Open in binary mode
<code>ios::app</code>	Open for appending
<code>ios::ate</code>	Open and go at end
<code>ios::nocreate</code>	Open only if exists
<code>ios::noreplace</code>	Open only if does not exists
<code>ios::trunc</code>	Open and delete the old if present

Moving Within a Stream

- ▶ The next byte you will put/get to/from a stream has a position
- ▶ `tell` functions return the current absolute position in a stream
- ▶ `seek` functions move the specified positions
- ▶ Positions can be either absolute or relative:
 - ▶ `ios::beg` relative to beginning (absolute)
 - ▶ `ios::cur` relative to current position
 - ▶ `ios::end` relative to end of file
- ▶ `p` (put) suffix for `ostreams`
- ▶ `g` (get) suffix for `istreams`

How to Move in a Stream?

- ▶ `ios::pos_type tellp()` returns current position
- ▶ `ios::pos_type tellg()` returns current position
- ▶ `seekp(pos_type)` moves to given absolute position
- ▶ `seekg(pos_type)` moves to given absolute position
- ▶ `seekp(pos_type, off_type)` moves to relative position, as given by `off_type`
- ▶ `seekg(pos_type, off_type)` moves to relative position, as given by `off_type`
- ▶ `pos_type` can be converted to `int` or `long`
- ▶ `off_type` is either `ios::beg`, `ios::cur` or `ios::end`

Some Examples

- ▶ `seek.cpp`
- ▶ The same stream can be used multiple times for reading from different files
 - ▶ Just close it and reopen it to bind to a different file
- ▶ `filestream.cpp`

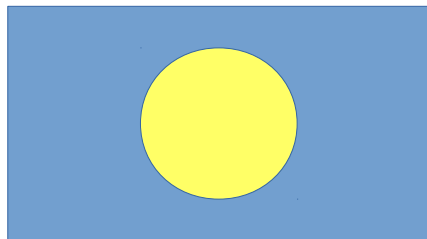
Overloading Extractors and Inserters for your Types

- ▶ 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`

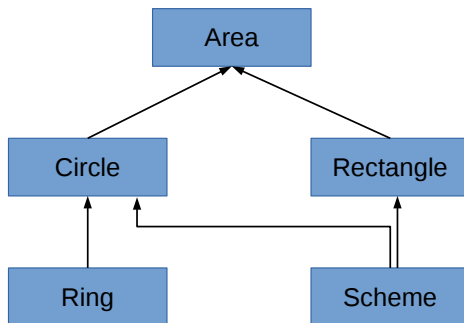
Beware of the Namespace

- ▶ `istream` and `ostream` are in the `std` namespace
- ▶ So if you try to overload inside a header file, where you are not supposed to use the namespace directive you must specify the namespace explicitly
- ▶ `friend std::ostream& operator<<(std::ostream&, const worker&);`

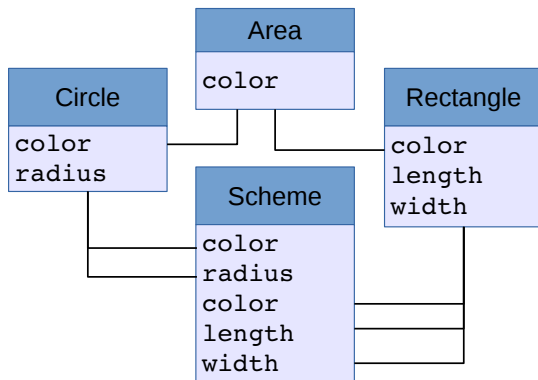
Multiple Inheritance: A Scheme



Class Hierarchy



Class Hierarchy (non-virtual Base Class)



Non-virtual Base Classes

- ▶ With polymorphism it is possible that a class can act as **indirect base class multiple times**
- ▶ Here Scheme will include members of the indirect base class Area multiple times
- ▶ One possibility to solve this problem is to use `static_cast<Circle *>` to create unambiguousness

Virtual Base Class

- ▶ Create unambiguousness by declaring a base class as virtual (i.e., inherit with : public virtual)
- ▶ Then only one subobject will be created
- ▶ In our example we need to explicitly call the Area constructor (with parameters) since no default constructor exists
- ▶ `nonvirtualinheritance.cpp`
- ▶ `virtualinheritance.cpp`
- ▶ `nonvirtualinheritance2.cpp`
- ▶ `virtualinheritance2.cpp`

Templates

- ▶ Templates allow to write generic code, i.e., code which will work with different types
 - ▶ Again those types could be unknown at code time
- ▶ A template tells the compiler that “what is following” will deal with an unknown type
- ▶ Later a specific type will be provided and the compiler will substitute it and generate ad-hoc code

Templates: Motivation

- ▶ Many times it is required to write different snippets of code which differ only in the types dealt with, but not in the underlying logic
 - ▶ Imagine the code to check for the existence of an element in an array of floats, or an array of pointers to a class, or an array of images
 - ▶ The logic is always the same
- ▶ So, why do not we write code which is parametric with respect to the possible types?

Searching in a Vector

- ▶ Assuming that a comparison operator is defined, the following code captures the logic to locate an element in a vector

```
1 int seek(sometype A[], int n, sometype toseek) {  
2     for (int i = 0; i < n; i++)  
3         if (A[i] == toseek)  
4             return i;  
5     return -1;  
6 }
```

- ▶ Should write different versions if sometype is `int`, or `float`, or `Complex`, or ...?

Templates: Functions and Classes

Type parameterization can be introduced for:

- ▶ **Functions:** like in the previous example; this helps in developing “algorithms”; you can concentrate on the logic, rather than on type details
 - ▶ Also, this decreases your coding time
- ▶ **Classes:** helps in developing “generic” classes; think about an array: the underlying logic is the same, whether it holds elements of type `int`, `Car`, `Student`, `double`, etc.
 - ▶ Again: concentrate on developing a working generic version

Templates: Basic Syntax (1)

- ▶ Two keyphrases are involved: `template` `class` and `template typename`
- ▶ They are functionally equivalent
- ▶ Template function: `template_function.cpp`

```
1 template <class T>
2 class Something {
3     T *p;
4     public: Something() { p = new T[100]; }
5 };
```

- ▶ Here the type `T` is not known, it will (and must) be specified when declaring instances of the class `Something`

Templates: Basic Syntax (2)

- ▶ When declaring an instance, the type is provided between angular brackets

```
1 int main(int argc, char** argv) {  
2     Something<int> ints;  
3     Something<char*> chars;  
4     Something<student> studentsome;  
5 }
```

- ▶ The compiler will generate the code necessary for the three different types
- ▶ `templatesone.cpp`

Some Remarks

- ▶ The code is generated by the compiler, which substitutes the generic `T` with the provided type
- ▶ Templates can be used both for methods and for “algorithms”, i.e., functions which work with different data types
- ▶ It is common (and necessary ...) to put both declaration and definition of the templated classes in the same header file
 - ▶ Does not break anything
 - ▶ The compiler will not allocate space and will not run into duplicate definition problems
- ▶ `stemplate.h`
- ▶ `stemplate.cpp`
- ▶ `stemplate_main.cpp`

Additional Remarks

- ▶ Multiple definitions are merged together
- ▶ If you specify `template<class T>`, the type parameter can be either a class or a basic data type
- ▶ You can have more than one parameter (generic or not):
`template<class T, int, double>`

One More Example

- ▶ Templates are very useful when developing general purpose containers, i.e., classes whose task is to store objects
- ▶ Most of containers use the same business logic to access data; the only difference is the data type
- ▶ A good container library can dramatically cut down your developing time
- ▶ `templatestack.cpp`

The Ownership Problem

What should a container hold? Objects or pointers to objects?

- ▶ If it contains object instances, we say it owns the objects
- ▶ If it contains pointers, we say it does not own objects
 - ▶ If it holds objects, they can be safely removed from memory during destructor execution
 - ▶ If it holds pointers to objects, other code is in charge of the destruction
 - ▶ Both seem to have their own advantages

Ownership: General Guidelines

Containers should not own objects; thus their destruction should be managed in places other than container destructors

- ▶ In general it is better to create objects on the heap and to access them via pointers
 - ▶ This opens the doors to a consistent use of polymorphism
- ▶ Management of object instances created on the heap is the source of many many bugs
 - ▶ Always double check your code involving `new` and `delete`

Are Templates Against OOP?

- ▶ Some OOP languages do not provide templates
- ▶ To write generic code, they just play with inheritance
 - ▶ For example, inherit everything from a single class and write code dealing with that class
 - ▶ Java and Smalltalk use this approach, but newer versions of Java have introduced something like templates
 - ▶ By offering both, C++ allows you to use both, at your choice

The Standard Template Library (STL)

- ▶ C++ standardization began in 1989 (until 1998)
- ▶ STL was later added in 1994
- ▶ STL is part of the Standard C++ Library
- ▶ Extends the core language by some general components
- ▶ May be reused for different purposes
- ▶ Programmers do not need to reinvent the wheel again and again
- ▶ Eases development of applications
- ▶ Makes software more maintainable

Brief Definitions

- ▶ Containers

- ▶ Manage collections of objects

- ▶ Iterators

- ▶ Navigate (step) through the elements of a container

- ▶ Algorithms

- ▶ Process elements of collections
 - ▶ E.g., search, sort, modify

Standard Template Library (1)

- ▶ Data and algorithm separated rather than combined
- ▶ Every kind of container can be combined with every kind of algorithm
- ▶ All components work with arbitrary types
- ▶ Components are **templates** for (almost) any type
- ▶ STL good example for “generic programming”

Standard Template Library (2)

- ▶ Containers are objects used to store other objects
- ▶ Containers size changes dynamically
- ▶ Very useful when objects are created on the heap
 - ▶ In that case containers hold their pointers
- ▶ Based on templates, containers can be used to store any data type

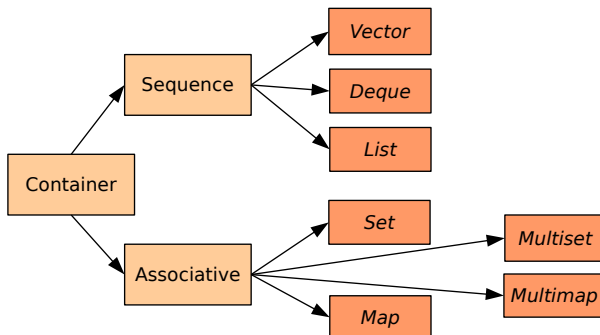
The Standard C++ Containers Library

- ▶ Derived from the STL, the two terms are often used as synonyms, but they are two different things (although pretty similar)
- ▶ Before reinventing the wheel check the standard library
 - ▶ In most of the situations, it is unlikely that you will need to develop yet another linked list class, or vector, or other widely used containers
 - ▶ Rely on widely used code developed by specialists
- ▶ Good documentation at <http://www.sgi.com/tech/stl/>

Containers Library

- ▶ Built over a restricted set of simple concepts, it can be used to quickly develop your software
- ▶ The two main concepts are **containers** and **iterators**
 - ▶ Containers hold objects, while iterators are used to move through containers to get/set objects
 - ▶ The iterator's mechanism is independent from the underlying container implementation, so you can use the same approach in many different situations
 - ▶ And of course without knowing how containers work
 - ▶ Containers dynamically grow or shrink to accommodate your storage needs

Fundamental Container Classes

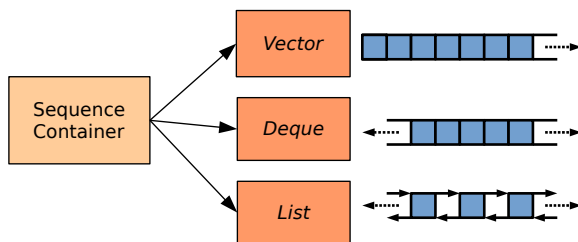


Predefined classes have different characteristics regarding
insert/access speed, size, usability

Containers

- ▶ Different containers for different needs
 - ▶ Sequences:
 - ▶ vector, deque, list
 - ▶ Associations:
 - ▶ set, multiset, map, multimap
- ▶ Common operations:
 - ▶ Insert an object into it
 - ▶ Remove an object from it
 - ▶ Iterate over all the elements (using an iterator)

Sequence Containers



- ▶ Ordered collection where every element has certain position
- ▶ Position depends on time and place of insertion, but independent of value of element
- ▶ Predefined containers differ in speed of insertion of elements and access to elements

Vector



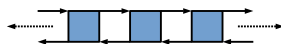
- ▶ Mimics an array
 - ▶ Provides random access
 - ▶ Fast insertion at the end, and fast indexing through overloaded `[]` operator
 - ▶ Needs more time if element is added at the middle of array
 - ▶ Not very efficient while resizing, and for what concerns memory allocation
- ▶ Constructors: `vector()`, `vector(int)`
- ▶ Basic methods: `push_back`, `pop_back`, `back`, `clear`, `size`, `max_size`, `empty`
- ▶ `vectorexample.cpp`

Deque



- ▶ Double ended queue
 - ▶ Elements are managed in dynamic array, which can grow in both directions
 - ▶ Appending and removing elements at beginning / end very fast
 - ▶ Needs more time if element is added at the middle of array
- ▶ Basic interface: very similar to vector; in addition `push_front`, `pop_front`, `front`
- ▶ Preferred to vector, unless you know exactly how many elements you will store
- ▶ `dequeexample.cpp`

List



- ▶ A double linked list
 - ▶ Element consists of
 - ▶ value
 - ▶ link to predecessor
 - ▶ link to successor
 - ▶ No random access
 - ▶ Fast insertion at both ends, slow access to intermediate elements
- ▶ Basic interface: similar to deque, but missing the `[]` operator; in addition
 - ▶ reverse, sort
- ▶ `listexample.cpp`

A Few Comments

- ▶ There are many more methods in every class
- ▶ If you use just the common methods of the containers, you will be able to change container by just changing their declaration and not the client code
- ▶ As with all containers, you can use them as black boxes
 - ▶ You do not need to care about their internal implementation

And More

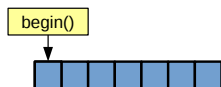
Stacks, queues, priority queues

- ▶ All implemented over the basic containers seen before
- ▶ Sometimes called adapters
- ▶ They do not provide additional capabilities, but rather reshape underlying containers

Iterators (1)

- ▶ Object that iterates over elements, which are all or part of the elements of an STL container
- ▶ Represents a certain position in a container
- ▶ Operations
 - ▶ * returns element at current position
 - ▶ ++ step forward to next element
 - ▶ == equals same position
 - ▶ != not equals same position
- ▶ May iterate over complicated data structures of containers (such as binary trees)
- ▶ Internal implementation depends on container

Member functions `begin()` and `end()` return Iterators



- ▶ Returns an iterator, that points to beginning of the elements in container



- ▶ Returns an iterator, that points to end of the elements in container; this is the position **behind** the last element (past-the-end-iterator)
- ▶ Both functions define a half-open range (includes first, but excludes last element)

Iterators (2)

- ▶ Iterators can be dereferenced, to gain access to the element they point to
 - ▶ Think to iterators as “very smart pointers”
 - ▶ But do not push this similarity too far
- ▶ Iterators are declared as follows:

```
vector<int> vint;  
vector<int>::iterator viterator;
```

 - ▶ This is because iterators are declared as container inner classes
- ▶ `iteratorsexample.cpp`

Using Iterators on Vectors

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     vector<int> v; // vector container for integers
7     v.push_back(2);
8     v.push_back(5);
9
10    vector<int>::const_iterator pos;
11
12    for (pos = v.begin(); pos != v.end(); ++pos) {
13        cout << *pos << ' ';
14    }
15    cout << endl;
16    return 0;
17 }
18
19 // if using C++11, you can use cbegin() and cend() instead
20 // of begin() and end()
```

Nested Templates

- ▶ Templates can be nested
- ▶ Keyword `typename` can be needed if not the current instantiation of a type but a dependent type is used
- ▶ `templ_in_templ1.cpp`
- ▶ `templ_in_templ2.cpp`