CH08-320143

# Programming in C++ II

C++ II

## Lecture 3 & 4

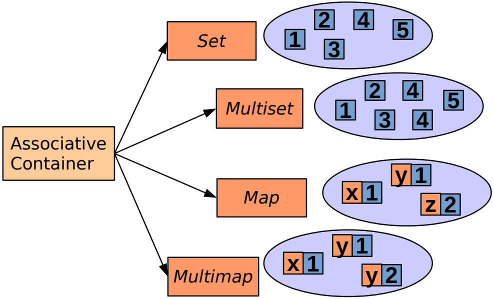
Dr. Kinga Lipskoch

Spring 2019

## Agenda Week 2

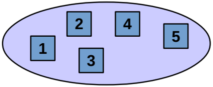
- STL: Associative Containers
- STL: Algorithms
- STL: More on Iterators
- C++11
- Exceptions

# Associative Containers



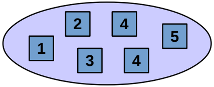
- Sorted collection (internally)
- Position of element depends on value (due to sorting criterion)
- Order of insertion irrelevant

## Sets



- Collection of elements, in which elements are sorted according to their values
- Duplicates are not allowed
- Interface:
  - `set`, `insert`, `erase`, `clear`, `empty`, `size`, `find`, `count`
- `sets.cpp`
- When are two elements equal?
  - It is possible to specify a functor, to be used when comparing objects
  - `set_functor.cpp`

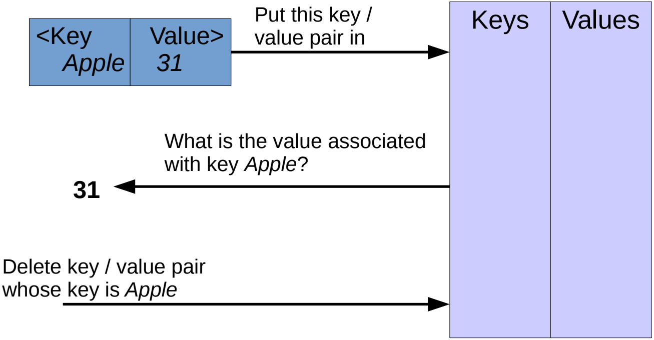
# Multisets



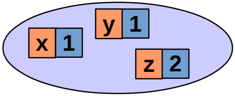
- A multiset is a container with an interface similar to set, but it accepts duplicate elements
- Both for sets and multisets, C++ STL provides algorithms for common (multiset) operations:
    - intersection, union, difference, symmetric difference

# Associations

Associations work with pairs of keys and values

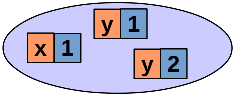

<Key Value>
*Apple* *31*
Put this key / value pair in
Keys
Values
What is the value associated with key *Apple*?
**31**
Delete key / value pair whose key is *Apple*

# Maps



- Collection of elements, which are key/value pairs; the key is basis for ordering
- Duplicate keys are not allowed
- Called “associative array”

## Multimaps



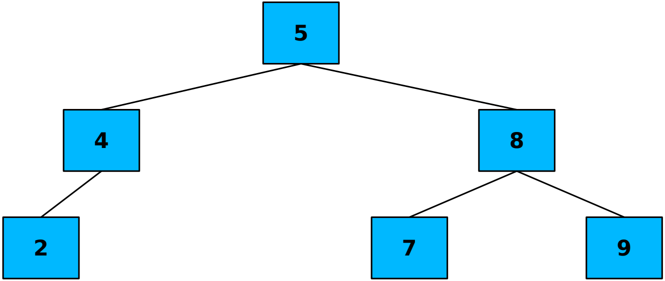
- Collection of elements, which are key/value pairs; the key is basis for ordering
- Duplicate keys are allowed
- Called "dictionary"

# Maps and Multimaps

- Basic interface: `find`, `clear`, `erase`, `insert`
- Map iterators return pairs: first element is the key and second element is the value
- `mapsexample.cpp`

# Internal Representation of Sets as Binary Tree

How do you iterate over the elements?



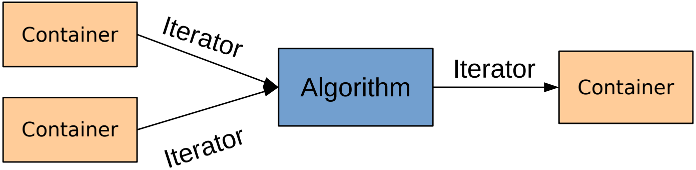
# Other Selected Member Functions

- Common to all containers:
    - `begin()`, `end()`, `erase(...)`, `size()`
- Optional member functions:
    - `pop_back()`, `pop_front()`, `push_back(const value_type& x)`, `push_front(const value_type& x)`
- Specific member functions:
    - sequences, (associative also possible, as hint)
        - `insert(iterator p, const value_type& x)`
    - associative
        - `insert(const value_type& x)`

# Time Overhead of Operations Sequence Containers

| Operation | Vector | Deque | List |
|---|---|---|---|
| **access first element** | constant | constant | constant |
| **access last element** | constant | constant | constant |
| **access random element** | constant | constant | linear |
| **add/delete at beginning** | linear | constant | constant |
| **add/delete at end** | constant | constant | constant |
| **add/delete at random** | linear | linear | constant |

# Separation of Data and Algorithm



- Data is managed by container classes
- Operations are defined by configurable algorithms
- Iterators are the glue between these components
- Any algorithm may interact with any container

# Algorithms (1)

STL provides standard algorithms that may process elements in container

- Non-manipulating algorithms:
  - `find(...)` find value in range
  - `count(...)` count appearances of value in range
  - `for_each(...)` apply function to range
  - `equal(...)` test whether the elements in two ranges are equal
  - `...`
- Manipulating algorithms:
  - `copy(...)` copy range of elements
  - `swap(...)` exchange values of two objects
  - `replace(...)` replace value in range
  - `remove(...)` remove value from range
  - `...`

# Algorithms (2)

- Sorting algorithms:
  - `sort(...)` sort elements in range
  - `min(...)` return the smallest
  - `set_union(...)` union of two sorted ranges
  - `...`
- Numerical algorithms:
  - `accumulate(...)` accumulate values in range, use `#include <numeric>`
  - `...`
- They are not member functions of container classes
- Global functions that operate with iterators

# Iterator Categories

- Input iterator – can only be used to read a sequence of values
- Output iterator – can only be used to write a sequence of values
- Forward iterator – can be read, written to, and move forward
- Bidirectional iterator – are like forward iterators, but can also move backwards
- Random access iterator – can move freely any number of steps in one operation

# set_union() on Different Containers (1)

Used headers in both examples that follow

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
```

```
set_union(InpIterator first1, InpIterator last1,
          InpIterator first2, InpIterator last2,
          OutIterator result)

inserter(Container, InpIterator)
```

# set_union() on Different Containers (2)

```
int main() {
  typedef vector<int> IntVec;
  IntVec a, b, c;
  a.push_back(2);
  a.push_back(3);
  b.insert(b.end(), 2);
  b.insert(b.end(), 4);
  set_union(a.begin(),a.end(),
    b.begin(), b.end(),
    inserter(c, c.begin()));
  IntVec::const_iterator pos;
  for (pos=c.begin();pos!=c.
    end(); ++pos) {
    cout << *pos << ' ';
  }
  cout << endl;
  return 0;
}
```

```
2 3 4
```

```
int main() {
  typedef set<string> StrSet;
  StrSet a, b, c;
  a.insert("BAA");
  a.insert("CAA");
  b.insert("BAA");
  b.insert("DAA");
  set_union(a.begin(),a.end(),
    b.begin(), b.end(),
    inserter(c, c.begin()));
  StrSet::const_iterator pos;
  for (pos=c.begin();pos!=c.
    end(); ++pos) {
    cout << *pos << ' ';
  }
  cout << endl;
  return 0;
}
```

```
BAA CAA DAA
```

# Other Set Operations

| | |
|---|---|
| `set_intersection(...)` | $A \cap B$ |
| `set_difference(...)` | $A \setminus B$ |
| `set_symmetric_difference(...)` | $(A \setminus B) \cup (B \setminus A)$ |

# Pros and Cons: Algorithms

- Advantages
  - Implemented only once for any container type
  - Might operate on elements of different container types
  - Reduces the code size
- Disadvantages
  - Usage not intuitive (high learning curve)
  - Some combinations of containers and algorithms might not work
  - Or combination is possible but not useful (speed, needed size)

# Useful STL Resources

- The C++ Standard Library by Nicolai M. Josuttis, Addison Wesley, 2$^{nd}$ edition, 2012
- C++ Annotations (Version 10.7.2) by Frank B. Brokken http://www.icce.rug.nl/documents/cplusplus/cplusplus.html
- C++ Reference http://www.cppreference.com/
- The C++ Programming Language by Bjarne Stroustrup (3$^{rd}$ edition) Pub. Addison-Wesley, ISBN 0-201-88954-4
- STL Tutorial and Reference Guide C++ Programming with the Standard Template Library by David R. Musser and Atul Saini, Pub. Addison-Wesley, ISBN 0-201-63398-1

# C++ Evolution

- Until 1989 Annotated C++ Reference Manual (ARM C++)
- 1990 - 1998 C++98 with addition of STL in 1995
- C++0x development started in 2002
  - C99
  - Boost Library
  - Library Extension TR1

# C++11 (C++0x)

C++ is a general-purpose programming language with a bias towards systems' programming that

- Is a better C
- Supports data abstraction
- Supports object-oriented programming
- Supports generic programming

Compile with the option `-std=c++11` or `-std=c++0x`

Example: `g++ -std=c++11 -Wall -o test test.cpp`

# B. Stroustrup: Goals of C++11

- Make C++ a better language for systems' programming and library building
  - Build on C++'s contributions to programming
  - Not providing specialized facilities for a particular sub-community (e.g., numeric computation or Windows-style application development)
- Make C++ easier to teach and learn
  - Increased uniformity
  - Stronger guarantees
  - Facilities supportive of novices: there will always be more novices than experts

# C++11 Aims

- Maintain stability and compatibility
- Prefer libraries to language extensions
- Prefer generality to specialization
- Support both experts and novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Fit into the real world

# Maintain Stability and Compatibility

- Billions of lines of existing code, which should not be broken
- But new keywords such as:
  - `auto` – example later
  - `decltype` – `decltype.cpp`
  - `const_expr` – example later
  - `nullptr` – `nullptr.cpp`

  are included as needed
- But many new features via libraries

# auto vs. decltype

```
int& foo() {
  ...
}

decltype(foo()) a = foo();   // int&
auto b = foo();                         // int
auto& c= foo();                         // int&
```

- auto determines value types
- decltype needs expression

## Support both Experts and Novices

- Nested containers are allowed
  - `vector_list.cpp`
- New keyword `auto` creates easier to read code
  - `list_old.cpp`
  - `list_auto.cpp`
  - `list_range_for.cpp`

# Improvements in the Standard Library

- New initializers – `initializer.cpp`
- Lambda-functions – `auto-lambda.cpp`
  - Anonymous functions
  - Allows to specify comparison function where it is needed
  - `[] () ->`
  - capture, parameter list, return type, function body
  - `lambda.cpp`

# Variadic Functions

- To access the variadic arguments from the function body, library facilities are provided (`<cstdarg>`):
  - `va_start` – enables access to variadic function arguments
  - `va_arg` – accesses the next variadic function argument
  - `va_copy` – (C++11) makes a copy of the variadic function arguments
  - `va_end` – ends traversal of the variadic function arguments
  - `va_list` – holds the information needed by `va_start`, `va_arg`, `va_end`, and `va_copy`
- `variadic_function.cpp`

# Variadic Templates

Allow to handle arbitrary number of template parameters

- `variadic_templates.cpp`
- `f()` takes arbitrary number of parameters and returns its number
- `printCommaSeparatedList()` expects one or more parameters and returns them in a comma separated list
- new operator `sizeof...`
- recursive call to `printCommaSeparatedList()`

## Tuples

- `pair` can be expanded to `tuple` now
- It is more general
- `tuple.cpp`

## Constant Expressions

- Sometimes compiler needs constant to e.g., create an array
  - `int vals[4];`
  - `Array<SZ> arr;`
- But not
  - `int val[getsize()];`
  - `Array<std::max(3, 4)>`
- New keyword
  - `constexpr`

# constexpr

- Determine expression's value at compile time
- Otherwise throw error
- May be declared as `constexpr`:
  - variables
  - functions
  - constructors
  - static methods
- `const_expr.cpp`

# static_assert

- Allows to use assertions at compile time
  - possible before by using the Boost library or preprocessor
- static_assert.cpp

# Exceptions

Errors happen because of:

- Hardware
- Changed environments
- Wrong usage or operation
- Bugs

# Conventional Error Handling

- Already available in C
  - Check whether pointer is `NULL`
  - Check `errno`
- `conventional_error_handling.cpp`

# New Keywords (1)

```
try
{
  // code, where exception
  // might occur
}
catch (char* text)
{
  // statements to be executed if
  // char* exception  occurs
}
```

# New Keywords (2)

- Statement that explicitly triggers a `char *` exception
  - `throw "No memory available";`
- Statement that explicitly triggers an `int` exception
  - `throw 12345;`

# try and catch (1)

- No exception in try-block
  - No exception handler is called
  - Program continues after catch-block
- throw within try creates exception
  - No further code in try-block is executed
  - Destructor for locally defined objects is called, before code in exception handler is run

# try and catch (2)

- Exception in try-block
  - First matching catch-block is executed
  - All other handlers are ignored
  - At most one handler is being called
- Exception in try-block, but no matching handler
  - Default action for uncaught exceptions
  - Usually it ends the program

# Exception Handling

- Blocks of code are specially marked
- If error occurs than control goes to special error routines
- `exception_handler.cpp`

# exception Class

- Class defines error class that receives objects via throw on exception
- Provides methods to give information about the error
- class_exception.h
- class_test.h
- class_test.cpp
- test_exception.cpp
- test_exception2.cpp

# All-round Handler

terminate.cpp