Object-oriented scientific programming with C++

Matthias Möller, Jonas Thies, Cálin Georgescu, Jingya Li (Numerical Analysis, DIAM)
Lecture 5

Goal of this lecture

Enumerators
Type aliases
Variadic template parameters
C++ standard container classes and algorithms
Iterators
Range-based and for-each for loops

Enumerators

Enumerators make it possible to collect named values `enum Color { red, green, blue };`
Named values are mapped to, e.g., `red=0, green=1, blue=2`
Usage

```
Color col = Color::red; switch col {
      case Color::red:   // do something
          break;
      case Color::green: // do something else
          break;
}
```

Enumerators

Enumerators can be initialised explicitly

```
enum Color { red=2, green=4, blue=8 };
```

Enumerators can be derived from a particular integral type

```
enum Color : int { red=2, green=4, blue=8 };
```

Enumerators can make use of arithmetic operations

```
enum Color { red=2, green=4, blue=8,
             cyan = red + green   };
```

However, an enumerator must not occur more than once

`enum TrafficLight { red, yellow, green };`

Scoped Enumerators

C++11 introduces **scoped enumerators** which can occur more than once (since they have different scopes!)

```
enum class Color { red, green, blue };
enum class TrafficLight { red, yellow, green };
```

For the rest, scoped enumerators can be used exactly in the same way as non-scoped enumerators

```
enum class Color { red=2, green=4, blue=8 };
enum class Color : int { red=2, green=4, blue=8 };
enum class Color { red=2, green=4, blue=8,
                   cyan = red + green   };
```

Type aliases

Implementation I: type aliases via typedef

```
template<typename T, T v>
struct trait {
      typedef T type;           // type is a type
      static const T value = v; // value is a variable
  };
```

Implementation II (since C++11): type aliases via using

```
template<typename T, T v>
struct trait {
      using type = T;           // type is a type
      static const T value = v; // value is a variable
  };
```

Type aliases

https://www.online-ide.com/RLlfvTsGpK

In [1]:

```
#include <iostream>
#include <typeinfo>


template<typename T, T v>
struct trait {
    typedef T type;           // type is a type
    static const T value = v; // value is a variable
};

int main() {
    typedef trait<int, 10> mytrait; // before C++11
    // using mytrait = trait<int, 10>; // since C++11

    std::cout << mytrait::value << " "
              << typeid(mytrait::type).name()
              << std::endl;

    return 0;
}
```

Intermezzo: `using` vs. `typedef`

Remember the function pointers from session 3

In [9]:

```
#include <iostream>
double myfunc0(double x) { return x; }

using funcPtr = double(*) (double);
funcPtr f = myfunc0;
std::cout << f(2.3) << std::endl;
```

```
2.3
```

Intermezzo: `using` vs. `typedef`

This becomes much less intuitive with typedef

In [1]:

```
#include <iostream>

double myfunc1(double x) {
    return x;
}

typedef double (funcPtr)(double); // Function pointer type
funcPtr* f = myfunc1;               // Assign myfunc1 to pointer f
std::cout << f(2.3) << std::endl;  // Use the function pointer to call myfunc1
```

```
2.3
```

Out[1]:

```
@0x7f2a8206dde0
```

Variadic templates

Task: implement a function that takes an **arbitrary number** of possibly **different variables** and computes their sum

```
cout << sum(1.0)                                  << endl;
cout << sum(1.0, 1.0)                             << endl;
cout << sum(1.0, (int)1)                          << endl;
cout << sum(1.0, (int)1, (float)1.3, (double)1.3) << endl;
```

Variadic templates

None of the template meta programming techniques we know so far will solve this problem with satisfaction
New concept in C++11: **variadic template parameters**
Idea: reformulate the problem as “one + rest”:
`sum(x1,x2,x3,...,xn) = x1 + sum(x2,x3,...,xn)`
That is, we combine recursion and function overloading with the ability to accept an arbitrary parameter list

Variadic templates

Function overload for **one argument**

```
template<typename T>
double sum(T arg) { return arg; }
```

Function overload for **more than one argument**

```
template<typename T, typename ... Ts>
double sum(T arg, Ts ... args)
{ return arg + sum(args...); }
```

The **template parameter pack**

```
template<typename ... Ts>
```

accepts zero or more template arguments but there can only be one template parameter pack per function.

Variadic templates

The number of arguments in the parameter pack can be detected using the `sizeof...()` function

```
template<typename ... Ts>
  int length(Ts ... args)
  {
      return sizeof...(args);
  }
```

Task: Write a type trait that determines the number of arguments passed to a function as parameter pack. In other words, implement the `sizeof...()` function yourself.

Automatic return type deduction

Task: Implement the sum function for an arbitrary number of parameters using automatic return type deduction

Function overload for **one argument** (with C++11)

```
template<typename T>
auto sum(T arg) -> decltype(arg)
{ return arg; }
```

Function overload for **one argument** (with C++14)

```
template<typename T>
auto sum(T arg)
{ return arg; }
```

Automatic return type deduction

Function overload for **more than one argument** (C++11)

```
template<typename T, typename ... Ts>
auto sum(T arg, Ts ... args)
-> typename std::common_type<T, Ts...>::type
  { return arg + sum(args...); }
```

Function overload for **more than one argument** (C++14)

```
template<typename T, typename ... Ts>
  auto sum(T arg, Ts ... args)
  { return arg + sum(args...); }
```

C++ standard containers

Aim: provide a set of universal container classes that

- can **store arbitrary types** (in general, only objects of the same type (in each container; `std::tuple` for multi-type containers)
- provide a **uniform interface** to insert, delete, access, and manipulate, items and iterate over the items stored
- provide optimal implementations of **standard data structures**, e.g., double-linked lists, balanced trees (red-black tree)

C++ standard containers

- `std::array` :array with compile-time size (non-resizable)
- `std::vector` :array with run-time size (resizable)
- `std::list` :double-linked list
- `std::forward_list` :single-linked list
- `std::stack` :Last-In-First-Out stack
- `std::queue` :First-In-First-Out queue
- `std::set/std::multiset` :Set of unique elements
- `std::map/std::multimap` :Set of (key,value) elements

C++ standard containers

Container classes support the following base functionality

- `size()`: returns the size of the container
- `empty()`: returns true of the container is empty
- `swap(container& other)`: swaps contents of containers Many container classes provide so-called **iterators**
- `begin()`, `end()`: editable iterator
- `cbegin()`, `cend()`: constant, i.e., non-editable iterator

Simple array example

https://www.online-ide.com/VxJaDz7Avm

```
#include <array>
std::array<int, 5> a = {1, 2, 3, 4, 5};
std::cout << "empty: " << a.empty() << ”\n”;
std::cout << "size: " << (int) a.size() << ”\n”;
std::cout << "max_size:" << (int) a.max_size() << “\n”;
for (auto i = 0; i < a.size(); i++)
    std::cout << a[i] << “\n”;
```

Simple array example

In [7]:

```
#include <iostream>
#include <array>


std::array<int, 5> a = {1, 2, 3, 4, 5};
std::array<int, 4> b = {6, 7, 8, 9};

//a.swap(b); //Uncomment this line to see what will happen

std::cout << "size: " << (int) a.size() << "\n";
std::cout << "size: " << (int) b.size() << "\n";
```

```
size: 5
size: 4
```

Simple vector example

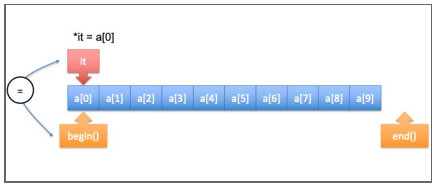
https://www.online-ide.com/OSemrgPAEK

```cpp
#include <iostream>
#include <vector>

  std::vector<int> v;
  v.reserve(20);
  v.push_back(42);
  v.push_back(11);
  v.push_back(1);
// ... additional push_back operations if needed
  for (auto i = 0; i < v.size(); ++i)
      std::cout << v[i] << "\n";
```

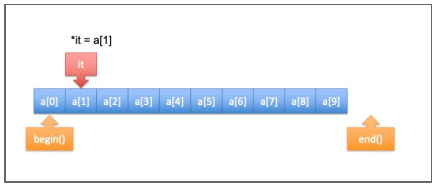
Iterators

Fixed size arrays



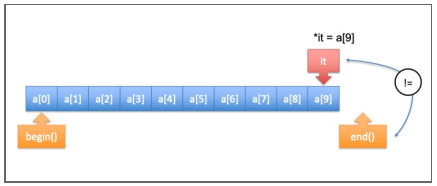
Iterators

Fixed size arrays



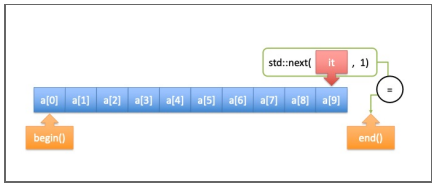
Iterators

Fixed size arrays



Iterators

Fixed size arrays



Simple vector example

Constant iterator over all entries

```
for (auto it = a.cbegin(); it != a.cend(); ++it)
      std::cout << *it << “\n”;
```

Non-constant iterator over all entries

```
for (auto it = a.begin(); it != a.end(); ++it)
    *it++;
```

More elegant screen output using **ternary operator**

```
for (auto it = a.cbegin(); it != a.cend(); ++it)
      std::cout << *it
                << (std::next(it,1) != a.cend() ? "," : "\n");
```

Simple vector example

The **ternary operator** `?:` implements an **inline if**

```
(condition ? true case : false case)
```

Usage:

```
– std::cout << (x>y ? x : y) << "\n";
– (x>y ? x : y) = 1;
– auto myfunc(int x, double y) // in C++14 {
        return (x>y ? x : y);
    }
```

Simple vector example

Range-based for loop (since C++11)

```
// access by constant reference (cannot modify i at all) for ( const auto& i : a )
      std::cout << i << "\n";
  // access by value (modify the local copy)
  for ( auto i : a )
      std::cout << i++ << "\n";
// access by reference (modify the original data) for ( auto&& i : a )
      std::cout << i++ << "\n";
```

Range-based for loops

Range-based for loops can be used with nearly all types

In [4]:

```
#include <iostream>
#include <array>
for ( int n : {0, 1, 2, 3, 4} )
      std::cout << n << " ";
for ( double h : {0.1, 0.05, 0.025, 0.0125} )
      //auto sol = solve_poisson(h); //Here you can iteratively call the function
```

```
0 1 2 3 4
```

Why **nearly**?

```
for ( auto c : {std::array<int,5>(); std::array<int,1>()})
    std::cout << c.size() << "\n";
```

In [2]:

```
auto array1 = std::array<int, 5>();
std::cout << typeid(array1).name() << "\n";
auto array2 = std::array<int, 1>();
std::cout << typeid(array2).name() << "\n";
```

```
St5arrayIiLm5EE
St5arrayIiLm1EE
```

C++ standard algorithms

Header file `algorithm` provides many standard algorithms

- `for_each(begin, end, function)`
- `position = find(begin, end, x)`
- `position = find_if(begin, end, function)`
- `number = count(begin, end, x)`
- `number = count_if(begin, end, function)`
- `sort(begin, end)`
- `sort(begin, end, function)`
- `position = merge(begin1, end1, begin2, end2, out)`

For-each loops

For-each loop iterates over all items and applies a user- defined unary function realized as lambda expression

https://www.online-ide.com/VrE05oFfZe

```
std::vector<int> v = {1, 2, 3, 4, 5};
std::for_each(v.begin(), v.end(),
              [](const int& n) { std::cout << " " << n; });
std::for_each(v.begin(), v.end(), [](int &n){ n++; });
```

For-each loops

For-each loop iterates over all items and applies a user- defined unary function realized as function object

https://www.online-ide.com/PNHB7kuGvX

```
struct Sum {
    Sum() : sum(0) {}
    void operator()(int n) { sum += n; }
    int sum;
};

Sum s = std::for_each(v.begin(), v.end(), Sum());
std::cout << “sum: “ << s.sum << “\n”;
```

Simple vector example

https://www.online-ide.com/fgqks71DQc

```cpp
#include <vector>
#include <algorithm>
  std::vector<int> v;
  v.reserve(20);
  v.push_back(42);
  v.push_back(11);
  v.push_back(1);
  std::sort(v.begin(), v.end());
  for (const auto& i : v)
      std::cout << i << std::endl;
```

Simple vector example

Provide standard library **compare function object**

```
#include <functional>
std::sort(v.begin(), v.end(), std::greater<int>() );
```

Provide user-defined **comparison as lambda expression**

```
std::sort(v.begin(), v.end(),
            [](int x, int y) { return y<x; } );
```

Simple vector example

Provide user-defined **comparison as function object**

```
struct {
      bool operator()(int x, int y) { return y>x; }
      } customGreater;
std::sort(v.begin(), v.end(), customGreater );
```

The power of iterators

C++ standard library functionality is largely based on iterators rather than absolute access via `operator[]`
For data structures like `std::list` the `operator[]` is not even defined since items can be inserted arbitrarily

In [2]:

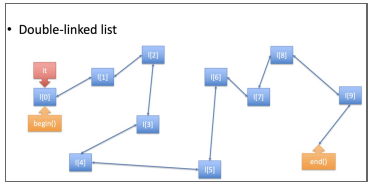
```cpp
#include <iostream>
#include <list>

std::list<int> L;
L.push_back(13);
L.push_front(3);
L.insert(++L.begin(), 4);

for (const auto& i : L) {
    std::cout << i << "\n";
}
```

```
3
4
13
```

Iterators



Maps

`std::map` handles key-value pairs efficiently

https://www.online-ide.com/jbpHoZer2X

```
enum class Color { red, green, blue };

std::map<Color, std::string> ColorMap = {
      {Color::red,   “red”},
      {Color::green, “green”},
      {Color::blue,  ”blue”}
  };
std::cout << ColorMap[Color::green] << "\n";
```

Maps

`std::map` handles key-value pairs efficiently
https://www.online-ide.com/Qekz72cqHF

```
enum class Color { red, green, blue };

std::map<std::string, Color> ColorMapReverse = {
        {"red", Color::red},
      {"green", Color::green},
      {"blue",  Color::blue}
  };

auto it = ColorMapReverse.find("green");

std::cout << it->first        // key
          << (int)it->second // value
          << "\n";
```

Tuples

Container `std::tuples` stores **heterogeneous types**

```
#include <tuple>
auto t = std::make_tuple(3.8, 'A', "String");
```

Access to individual elements

```
std::cout << std::get<0>(t) << std::endl; //Output: 3.8
std::cout << std::get<1>(t) << std::endl; //Output: A
std::cout << std::get<2>(t) << std::endl; //Output: String
```

Create tuple from parameter pack

```
auto t = std::tuple<Ts...>(args...);
```

Tuples

Get the size of the tuple

```
std::cout << std::tuple_size<decltype(t)>::value;
```

However, it is impossible to iterate over the elements of a tuple using any of the run-time techniques seen before

- No operator[]
- No iterators
- No range-based or for-each loops

Task: Find a way to iterate over the elements of a tuple using compile-time techniques

Tuples

https://www.online-ide.com/cygjF7U2vI

```
template<int N, typename Tuple> struct printer {
      static void print(Tuple t) {
          printer<N-1,Tuple>::print(t);
          std::cout << std::get<N>(t) << "\n";
}};
  // Specialization for first entry
  template<typename Tuple>
  struct printer<0,Tuple> {
      static void print(Tuple t) {
          std::cout << std::get<0>(t) << "\n";
}};
```

Tuples

Usage

```
int main() {
    auto t = std::make_tuple(3.8, 'A', "String");
    printer<std::tuple_size<decltype(t)>::value-1,
            decltype(t)>::print(t);
}
```

In [ ]:

```

```