TW3720TU: Object Oriented Scientific Programming with C++ (11/14/17)

Matthias Möller

Numerical Analysis

Goal of this lecture

- Clarification on enumerators and type traits
- 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
    case Color::green: // do something else
}
```

Enumerators, cont'd

- 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 traits

- Generic structure of a type-value trait

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

- Alternative implementation since C++11

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

Type traits, cont'd

- Usage

```
#include <iostream>
#include <typeinfo>
int main()
{
    typedef trait<int, 10> mytrait;
    // or
    using mytrait = trait<int, 10>;

    std::cout << mytrait::value
              << typeinfo(mytrait::type).name()
              << std::endl;
}
```

Intermezzo: *using vs. typedef*

- Remember the function pointers from week 3

```
const double myfunc1(double x) { return x; }

int main()
{
    using funcPtr = double(*) (double);
    funcPtr f = myfunc1;
    std::cout << f(2.3) << std::endl;
}
```

Intermezzo: *using vs. typedef*

- This becomes much less intuitive with `typedef`

```
const double myfunc1(double x) { return x; }

int main()
{
    typedef double (funcPtr)(double);
    funcPtr* f = myfunc1;
    std::cout << f(2.3) << std::endl;
}
```

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 follows
  `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, cont'd

- Function overload for one argument

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

- Function overload for more than one argument

```
template<typename T, typename ... Ts>
static 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, cont'd

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

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

- **Task:** Write a 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>
static auto sum(T arg) -> decltype(arg)
{ return arg; }
```

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

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

Automatic return type deduction, cont'd

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

```
template<typename T, typename ... Ts>
static 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>
static 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 access, manipulate, insert, delete 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, cont'd

- `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, cont'd

- 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

```
#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, cont'd

```
#include <array>

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

A.swap(B);

std::cout << "size: " << (int) A.size() << "\n";
std::cout << "size: " << (int) B.size() << "\n";
```

Simple vector example

```
#include <vector>

std::vector<int> V;
V.reserve(20);
V.push_back(42);
V.push_back(11);
V.push_back(1);
...

for (auto i=0; i<V.size() ; i++)
    std::cout << V[i] << “\n”;
```

Simple vector example, cont'd

- **Constant iterator** over all entries

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

- **Non-constant iterator** over all entries

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

- More elegant screen output using ternary operator

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

Notice the difference between `*it` and `it+1`

Intermezzo: Ternary operator

- The **ternary operator** `?:` implements an **inline if**
  `(condition ? true case : false case)`
- Usage:
  - `std::cout << (a>b ? a : b) << "\n";`
  - `(a>b ? a : b) = 1;`
  -
```
auto myfunc(int a, double b) // in C++14
{
    return (a>b ? a : b);
}
```

Simple vector example, cont'd

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

```
// access by constant reference (cannot modify i at all)
for ( const auto& i : V )
    std::cout << i << "\n";

// access by value (modify the local copy)
for ( auto i : V )
    std::cout << i++ << "\n";

// access by reference (modify the original data)
for ( auto&& i : V )
    std::cout << i++ << "\n";
```

Range-based for loops

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

```
for ( int n : {0, 1, 2, 3, 4} )
    std::cout << n << “\n”;

for ( double h : {0.1, 0.05, 0.025, 0.0125} )
    auto sol = solve_poisson(h);

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

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

```
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, cont'd

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

```
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

```
#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, cont'd

- 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 a, int b) { return b<a; } );
```

Simple vector example, cont'd

- Provide user-defined comparison as function object

```
struct
{
    bool operator()(int a, int b) { return b>a; }
} 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

```
#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";
```

Maps

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

```
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, cont'd

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

```
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;
std::cout << std::get<1>(t) << std::endl;
std::cout << std::get<2>(t) << std::endl;
```

- Create tuple from parameter pack

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

Tuples, cont'd

- 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, cont'd

```
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, cont'd

- Usage:

```
int main() {
    auto t = std::make_tuple(3.8, 'A', "String");

    printer<std::tuple_size<decltype(t)>::value-1,
            decltype(t)>::print(t);
}
```