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

Matthias Möller

Numerical Analysis

Goal of this lecture

- Structuring code into libraries and applications
- Introduction to the concept of expression templates
- Practical examples of expression template libraries

Structuring of source code

- Until now, we have written the complete source code (classes, structs, functions, main function) in a single file.
- For larger project, it makes sense to structure the code into
  - Commonly used core functionality (library)
  - Individual user programs (application)
- Inside both groups it, again, makes sense to split the source code into multiple files to keep overview (for yourself) and improve readability and maintainability (for others)

Header-only libraries

- Libraries can be implemented in two ways:
  - **Header-only:** common functionality is implemented in header files that are included and compiled together with application
  - **Compiled:** common functionality is compiled into a stand-alone library that needs to be linked to the executable binary

Header-only libraries, cont'd

- Header file `LAlib.hpp`

  ```
  #pragma once
  class vector {
      ...
  };

  class matrix {
      ...
  };
  ```

- The `#pragma once` directive ensures that the file is included only once per compilation

- Application code `demo.cxx`

  ```
  #include <LAlib.hpp>
  int main() {
    vector v;
  }
  ```

- Instruct CMake to consider the header file during compilation

  ```
  include_directories(src)
  add_executable(demo,
                 demo.cxx,
                 src/LAlib.hpp)
  ```

Header-only libraries, cont'd

- The #pragma once directive is non-standard and not supported by all compilers. Alternative: include guards

```
#ifndef LALIB_HPP
#define LALIB_HPP

// Implementation

#endif // LALIB_HPP
```

Header-only libraries, cont'd

- PROS
  - Compiler can inline functions and perform aggressive optimisation
  - Only parts of the library used by the application are compiled
  - No problems with binary incompatibility
  - No problems with templates(!!!)
- CONS
  - You give away the source code
  - Longer compilation times since each application compiles ('its part' of) the library source again
  - Larger size of executable binaries since common functionality is not collected into a single library

Compiled libraries

- How to separate declaration from implementation?
- General strategy
  - Create `file.h`, `file.cpp` with declaration and implementation
  - Compile all cpp-files into a single library
  - Include h-files in application and compile it
  - Link the library to the application to produce the final executable

Compiled libraries, cont’d

- **Declaration** in file `vector.h`

```
#ifndef VECTOR_H
#define VECTOR_H

class vector {
public:
    vector(std::size_t n);
private:
    double * data;
    std::size_t n;
};

#endif // VECTOR_H
```

- **Implementation** in file `vector.cpp`

```
#include “vector.h”


vector::vector(std::size_t n)
: data(new double[n], n(n)
{}
```

Compiled libraries, cont’d

- Instruct CMake to compile all cpp-files into a single library

```
add_library(LAlib vector.cpp, vector.h,
                  matrix.cpp, matrix.h, ...)
```

- Include h-files in application and instruct CMake to link the library to the application’s executable

```
include_directories(src)
add_executable(demo main.cxx, vector.h, matrix.h)
target_link_libraries(demo LINK_PUBLIC LAlib)
```

Compiled libraries, cont'd

- How to use templates in pre-compiled libraries?
- **Answer 1:** Explicit template instantiation (in CPP-file) of all variants you want to use in your applications

```
#include "vector.h"

template<typename T>
vector<T>::vector(std::size_t n)
: data(new T[n]), n(n)
{}

template class vector<double>;
template class vector<float>;
```

Compiled libraries, cont’d

- How to use templates in pre-compiled libraries?
- **Answer 2:** If your code makes heavily use of template metaprogramming then simply don’t use compiled libraries
- Remember, we introduced templates to let the compiler do smart copy-paste for you. Don’t fool yourself by explicitly instantiating all possible combinations of template parameters just to compile your code into a library

The ultimate code

- We would like to be able to write mathematical expressions in the same comfortable way as it is possible in MATLAB

```
x = (A+2*B)\(f+sin(g));
```

  and at the same time exploit the full speed of C++
- Here, we assume that A and B are dense nxn matrices and f and g are vectors of size n. In practice, A and B can also be sparse matrices with or without the same sparsity pattern given in either the same or a different matrix format.

The BLAS approach

- Basic Linear Algebra Subprograms define a list of routines
  - for vector-vector (BLAS1) operations, e.g., $y=ax+y$, $y=ay$
  - matrix-vector (BLAS2) operations, e.g., $y=aAx+by$
  - matrix-matrix (BLAS3) operations, e.g., $C=aAB+bC$
- Optimised BLAS libraries available: ACML, ATLAS, OpenBLAS, Intel MKL, uBLAS, CUBLAS, …

The BLAS approach, cont’d

- CMake offers a simple mechanism to let you find installed BLAS libraries and link your code to them

```
# Search for installed BLAS library
find_package(BLAS REQUIRED)

# Link to BLAS library
target_link_libraries (demo ${BLAS_LIBRARIES})
```

The BLAS approach, cont’d

- If you do not make use of the CBLAS header files (not default with all BLAS libraries) you need to explicitly declare the list of BLAS routines that you want to use

```
extern “C” {
void daxpy_(int * n,
            double * alpha, double * x, int * ldx,
            double * y, int * ldy);

void dscal_(int * n,
            double * alpha, double * x, int * ldx);
}
```

The BLAS approach, cont’d

- Use BLAS routines in the main program

```
double x[10], y[10], alpha=1.5, beta=2.0;
int    n=10, incx=1, incy=1;

// Compute y = 1.5*x + 2.0*y
daxpy_(&n, &alpha, x, &incx, y, &incy);
dscal_(&n, &beta,  y, &incy);
```

The BLAS approach, cont'd

- Shortcomings:
  - You can only call the available routines, e.g., if you want to compute *y=ax+by* then you must split it up into *y=ax+y* and *y=by*
  - BLAS routines have their origin in Fortran 77 (long before template metaprogramming etcetera has been developed)
  - BLAS routines are strongly typed (no support for mixing different types) and have an old-fashioned calling convection

Our own LA library

- **Task:** Let us design our own linear algebra library that supports the following operations:
  - Element-wise Add, Sub, Div, Mul of vectors
  - Add, Sub, Div, Mul of a vector with a scalar
  - Copy assignment (other functionality can be added later)
- Additional requirements:
  - Support for arbitrary types: template metaprogramming
  - Mathematical notation: operator overloading

LA library

- **Vector class:** Attributes

```
template<typename T>
class vector {

private:
   // Attributes
   T* data;
   std::size_t n;

...
```

LA library, cont'd

- **Vector class:** Constructors

```
template<typename T>
class vector {

public:
   // Default constructor
   vector()
   : n(0), data(nullptr) {}

   // Size constructor
   vector(std::size_t n)
   : n(n), data(new T[n]) {}
...
```

LA library, cont’d

- **Vector class:** Constructors, cont’d

```
template<typename T>
class vector {

  // Copy constructor
  vector(const vector<T>& other)
  : vector(other.size())
  {
    for (std::size_t i=0; i<other.size(); i++)
        data[i] = other.data[i];
  }
...
```

LA library, cont'd

- **Vector class:** Constructors, cont'd

```
template<typename T>
class vector {

  // Move constructor
  vector(vector<T>&& other)
  {
    data = other.data; other.data = nullptr;
    n = other.n;       other.n    = 0;
  }
...
```

LA library, cont'd

- **Vector class:** Destructor and helper functions

```
template<typename T>
class vector {

  // Destructor
  ~vector() { delete[] data; data=nullptr; n=0; }

  // Return size
  inline const std::size_t size() const { return n; }
```

LA library, cont’d

- **Vector class:** Helper functions

```
template<typename T>
class vector {

  // Get data pointer (constant reference)
  inline const T& get(const std::size_t& i) const {
    return data[i];
  }

  // Get data pointer
  inline T& get(const std::size_t& i) {
    return data[i];
  }
```

LA library, cont’d

- **Vector class:** Assignment operator

```
template<typename T>
class vector {

  // Scalar assignment operator
  vector<T>& operator=(const T& value) {
    for (std::size_t i = 0; i < size(); i++)
      data[i] = value;
    return *this;
  }
```

LA library, cont'd

- **Vector class:** Assignment operator

```
template<typename T>
class vector {

  // Copy assignment operator
  const vector<T>& operator=(const vector<T>& other) const {
    if (this != &other) {
      delete[] data; n = other.size(); data = new T[n];
      for (std::size_t i = 0; i < other.size(); i++)
        data[i] = other.data[i];
    }
    return *this;
  }
```

LA library, cont'd

- **Vector class:** Assignment operator

```
template<typename T>
class vector {

  // Move assignment operator
  vector<T>& operator=(vector<T>&& other) {
    if (this != &other) {
      std::swap(this->data, other.data);
      std::swap(this->n,    other.n);
      other.n = 0; delete[] other.data; other.data = nullptr;
    }
    return *this;
  }
```

LA library, cont'd

- **Vector class:** binary vector-vector operators (**outside class!**)

```
template<typename T1, typename T2>
auto operator+(const vector<T1>& v1,
               const vector<T2>& v2) {
  vector<typename std::common_type<T1,T2>::type>
    v(v1.size());
  for (auto i=0; i<v1.size(); i++)
    v.get[i] = v1.get[i] + v2.get[i];
  return v;
}
```

- Similar for operator-, operator* and operator/

LA library, cont'd

- **Vector class:** binary scalar-vector operators (**outside class!**)

```
template<typename T1, typename T2>
auto operator+(const T1&         s1,
               const vector<T2>& v2) {
  vector<typename std::common_type<T1,T2>::type>
    v(v2.size());
  for (auto i=0; i<v1.size(); i++)
    v.get[i] = s1 + v2.get[i];
  return v;
}
```

- Similar for operator-, operator* and operator/

LA library, cont'd

- Vector class: binary vector-scalar operators (**outside class!**)

```
template<typename T1, typename T2>
auto operator+(const vector<T1>& v1,
               const T2&         s2) {
  vector<typename std::common_type<T1,T2>::type>
    v(v1.size());
  for (auto i=0; i<v1.size(); i++)
    v.get[i] = v1.get[i] + s2;
  return v;
}
```

- Similar for operator-, operator* and operator/

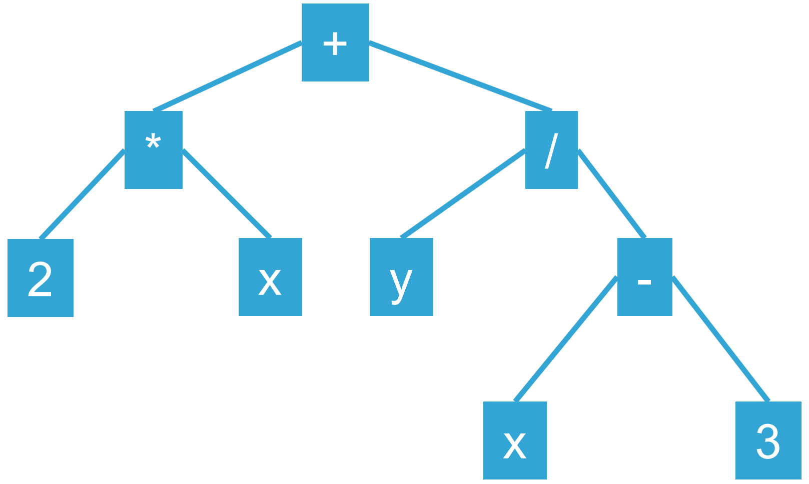
Quiz: LA library

- What is the **theoretical complexity** (#memory transfers and #floating-point operations) of the expression `z=2*x+y/(x-3)`?
- What is the **practical complexity** of the following code?

```
int main() {
  vector<double> x(10), y(10), z(10); x=1.0; y=2.0;
  z=2*x+y/(x-3);
}
```

LA expression template library

- **Aim:** We want that the expression `z=2*x+y/(x-3)` is evaluated in a single for loop without temporaries.
- **Strategy:** Build up an **expression tree** that holds the steps to be performed to evaluate the expression but delay the actual evaluation until the assignment operator is used.



- The expression tree is a **symbolic** and **lightweight** representation of the expression and does not carry data.

LA expression template library, cont'd

- Implement templated vector class (derived from base class `vectorBase`) as before but remove all operators (+,-,*,/)
- Implement an additional **assignment operator**, which loops through the expression `e` and assigns the value to `data[i]`

```
template <typename E>
vector<T>& operator=(const E& e) {
    for (std::size_t i = 0; i < size(); i++)
        data[i] = e.get(i);
    return *this;
}
```

LA expression template library, cont'd

- For each **binary operator** we need to implement a helper class that represents the operation in the expression tree

```
template<typename E1, typename E2>
class VectorAdd : vectorBase {
private:
  const E1& e1;
  const E2& e2;

public:
  VectorAdd(const E1& e1, const E2& e2)
  : e1(e1), e2(e2)
  {}
...
```

LA expression template library, cont'd

- For each **binary operator** we need to implement a helper class that represents the operation in the expression tree

```
template<typename E1, typename E2>
class VectorAdd : vectorBase {
...
  inline const
  typename std::common_type<typename E1::value_type,
                            typename E2::value_type>::type
  get(std::size_t index) const {
    return e1.get(index)+e2.get(index);
  }
};
```

LA expression template library, cont’d

- For each **binary operator** we need to implement a helper class that represents the operation in the expression tree

```
template<typename E1, typename E2>
class VectorAdd : vectorBase { ... };
```

- Base class `vectorBase` has no functionality and is used to check if a template argument belongs to the expression tree

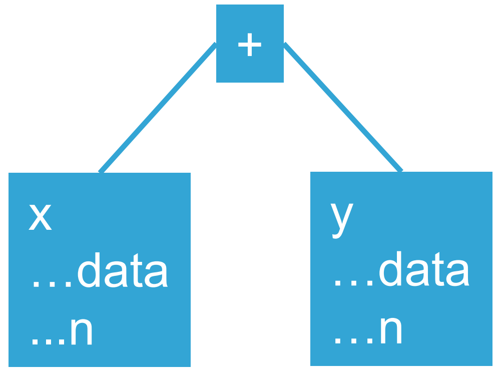
```
typename std::enable_if<
           std::is_base_of<vectorBase, E1>::value
         >::type
```

LA expression template library, cont’d

- Implement the corresponding **binary operator** overload

```
template<typename E1, typename E2>
auto operator+(const E1& e1, const E2& e2) {
    return VectorAdd<E1,E2>(e1,e2);
};
```

- Now, expression `auto e=x+y` creates a new item in the expression tree and keeps constant references to `x` and `y`



- Expression evaluation for a single index is triggered by `e.get(index)`

LA expression template library, cont'd

- The expression `auto a=x+y+z` has the type

```
vectorAdd<vectorAdd<vector<TX>,
                    vector<TY>>,
          vector<TZ> >
```

- During the assignment of this expression to a vector object `vector<double> v=a;` the `get(index)` function is called for each single index. The overall computation takes place in a **single for loop** as if we had written the following code

```
for (std::size_t i=0; i<x.size(); i++)
  v.get(i) = x.get(i)+y.get(i)+z.get(i);
```

LA expression template library, cont'd

- **A word of caution:** Expression templates (ET) are very powerful (for you as a user) but writing an **efficient** ET linear algebra library takes much time and is not trivial
- ET eliminate multiple for-loops but they do by no means imply parallelism, use of SIMD intrinsics, etcetera
- A recent trend is to use the ET concept as high-level user-interface and implement all the dirty tricks (inlined assembler code, vector intrinsics, …) in helper classes

ET linear algebra libraries

- **Armadillo:** MATLAB-like notation
- **ArrayFire:** broad support on ARM/x86 CPUs and GPUs
- **Blaze:** aims for ultimate performance
- **Eigen:** easy usage and broad user community
- **IT++:** grandfather of all ET LA libraries
- **MTL4:** distributed HPC + fast linear solvers
- **VexCL:** broad support on CPUs and GPUs
- **ViennaCL:** fast linear solvers

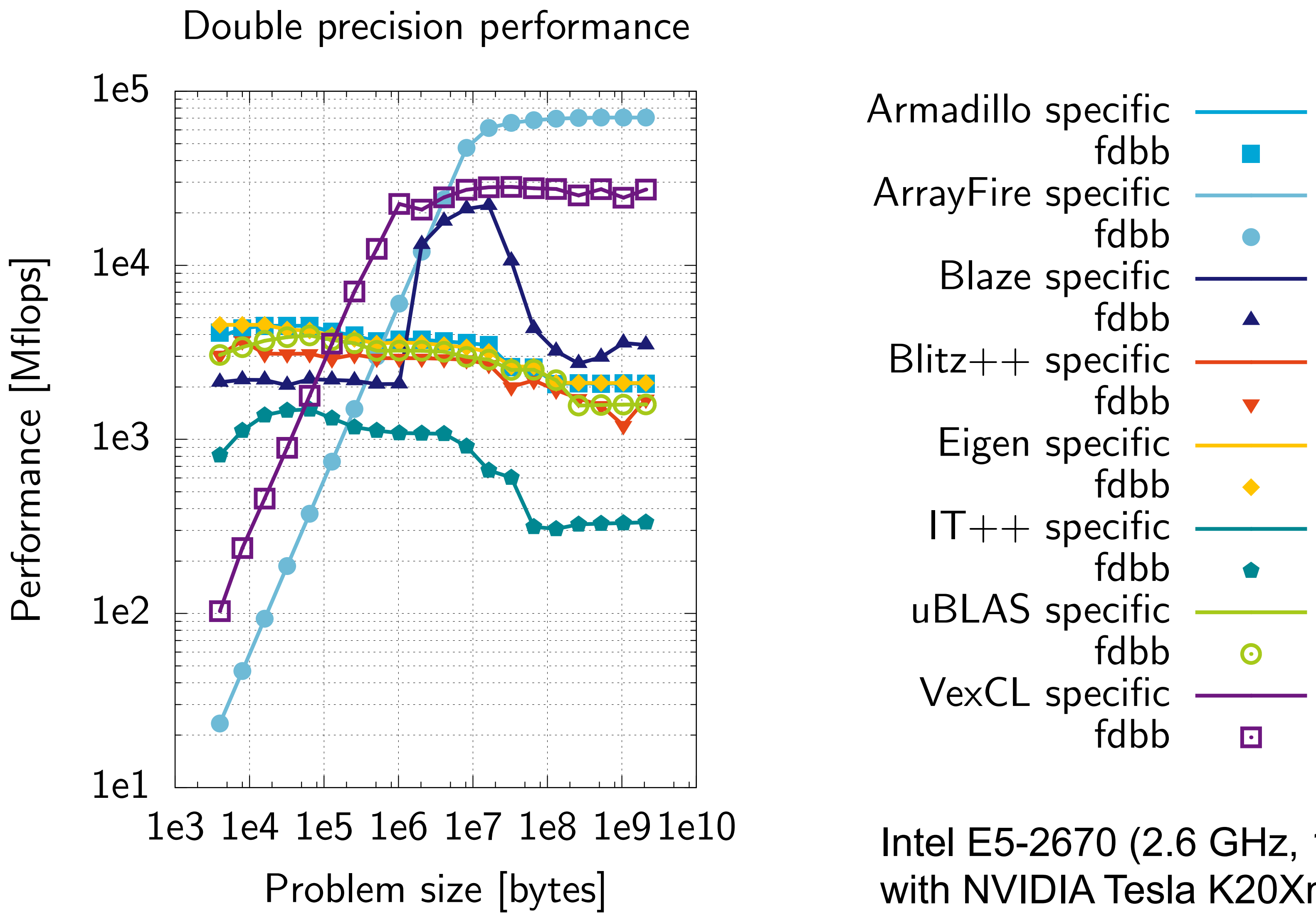
Example: Eigen library

-
  ```
  #include <Eigen/Dense>
  using namespace Eigen;
  int main() {
    MatrixXd A(3,3), B(3,3);
    VectorXd x(3), y(3), z(3);
    A << 1,2,3,4,5,6,7,8,9;
    B << 1,0,0,0,1,0,0,0,1;
    x << 1,2,3;
    y << 4,5,6;

    z = A.transpose() * x.sqrt() + B * y.abs();

    std::cout << “z=“ << z << std::endl;
  }
  ```

Performance test: $(u^2+v^2+w^2)/r^2$



Intel E5-2670 (2.6 GHz, 16 cores)
with NVIDIA Tesla K20Xm GPU

Summary

- In this course, you have learned
  - Concepts of modern object-oriented scientific programming
  - State-of-the-art C++11/14/17 programming techniques
  - Professional software development workflows (Git, CMake, …)
- You will learn how to use these techniques efficiently by applying them in practice for, e.g., master projects
- My rule of thumb: spend 80% of the time on the design (and re-re-redesign) and only 20% on implementation

Outlook

- If you got interested in this topic and want to know more about high-performance computing contact me or have a look at my website and/or some of my recent research projects:
  - Fluid Dynamics Building Blocks: gitlab.com/mmoelle1/FDBB
  - Universal ET library interface: gitlab.com/mmoelle1/Uetli
  - Compressible flow solver in G+Smo: gs.jku.at/gismo
  - Multi-objective design optimization of fluid energy machines: motor-project.eu