The Adaptiv Framework

Nuno Alves de Sousa

Instituto Superior Técnico
Área Científica de Mecânica Aplicada e Aeroespacial

March 8, 2019

Outline

Outline

Attributes of good software

"Software and cathedrals are much the same – first we build them, then we pray."

(Anonymous)

Attributes of good software

Not *what* the program does, but *how well* it does it:

Maintainability	reduce/reverse "code entropy" cheaper/safer to change than to rewrite
Dependability	availability, reliability, safety, integrity
Efficiency	algorithmic efficiency storage efficiency
Usability	"consumer" effectiveness and efficiency elegance and clarity perceived by the user

Outline

Where to start?

"What happens before one gets to the coding stage is often of crucial importance to the success of the project."
(Meek & Heath - Guide to Good Programming Practice)

Higher-level prerequisites to provide a solid foundation for coding:

- Coding standards
- Choice of programming language
- Life cycle, architecture, design
- Requirements

Coding standards

Coding conventions are particularly important in collaborative projects:

- Much easier to read someone else's code
- Uniform style (*e.g.* naming conventions for filenames, variables, etc)
- Deal with undereducated programmers
- Avoid insufficient library use
- Portability
- Commenting conventions:
  - Speed up knowledge transfer
  - Comment only what code expresses poorly (intent)
  - Comments lie, code never lies
  - Do not comment code modifications (use a version control system)

Version control

Source code is the most valuable asset of any software project

Version control systems (VCS)

- Management of changes to all non-binary files
- Complete retrace of all versions of each file
- History of the authors of such changes

Critical advantages

- Rollback of all tracked changes
- Work in an isolated fashion
- Seamless team collaboration
- Efficient and flexible scaleability

git - the world's leading version control system

TÉCNICO LISBOA

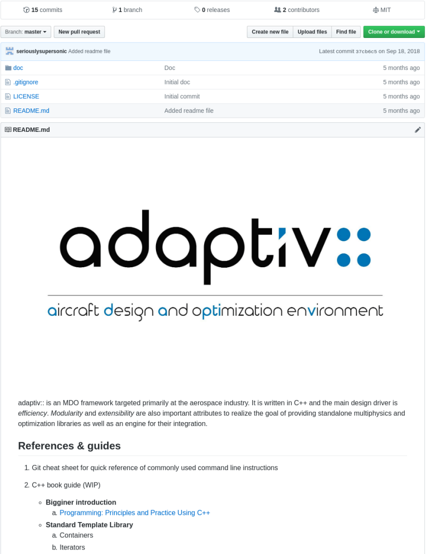
git

Why git?

- Free and open-source
- Small and fast
- Encourages branching
- Distributed
- Built-in IDE support

As a service:

- Source code hosting
- Code sharing platform
- GitHub, GitLab, etc.



Outline

Build system

- Open-source, cross-platform set of tools to build, test and package software.
- Controls compilation process using platform and compiler independent config. files

The defacto standard for building C++ projects

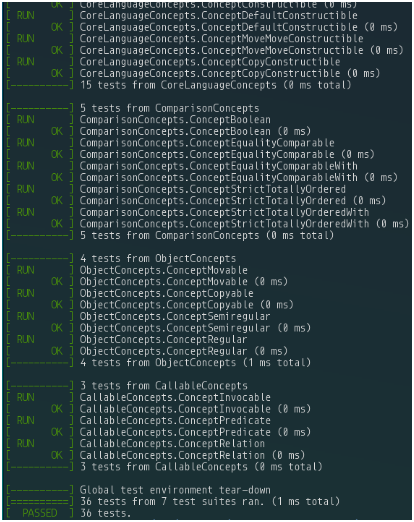
Advantages

- More time for coding
- Supported by most popular IDEs (*e.g.* VS, JetBrains, QtCreator)
- Support for multiple compilers (*e.g.* MSVC, GCC, Clang, Intel)
- Easy integration of 3rd party libraries

Testing

"Beware of bugs in the above code; I have only proved it correct, not tried it."
(Donald Knuth)

- Testing is essential when collaborating
- Can prevent reviewing bad code
- Should keep up with development
- Write once, test many
- Popular frameworks:
  - Google Test (on the right)
  - catch2 (header only)



Outline

C++ templates

What are templates?

- Foundation of generic programming
- Blueprint for creating a generic class or function

What are their uses?

- Avoid repeating code
- Generate code at compile-time
- Perform compile-time computations

But really... why bother?

- C++ template magic!
  - Static polymorphism (no overhead)
  - Higher chances for compiler optimizations (*e.g.* inlining)
  - Create elegant interfaces with highly optimized implementations

  and more...

C++ template metaprogramming (TMP)

Object-oriented programming and TMP techniques allow OpenFOAM users to represent

$$\frac{\partial}{\partial t}(\rho \mathbf{U}) + \nabla \cdot (\phi \mathbf{U}) - \mu \nabla^2 \mathbf{U} = -\nabla p, \tag{1}$$

with a syntax that closely resembles the mathematical formulation:

```
solve
(
     fvm::ddt(rho,U)
   + fvm::div(phi,U)
   - fvm::laplacian(mu,U)
 ==
   - fvc::grad(p)
);
```

Note: what if **U** is not actually a vector field?

Metaprogramming pitfalls

Becoming a template wizard takes time (and a great deal of insanity):

- Many TMP techniques require knowledge of specific C++ idioms
- Frequently, error messages are cryptic:
  - Most errors are triggered only on template instantiation
  - Stack trace might be very deep
  - Type names can be extremely long (*e.g.* templates instantiations as template arguments)
  - Overload resolution failure can produce a long list of candidates

Inexperienced programmers can easily get stuck (and frustrated) but...

Often, TMP errors are related to instantiation with an invalid type

C++ concepts

Concepts are constraints that limit the set of arguments accepted as template parameters:

- Type-checking
- Simplified compiler diagnostics
- Select overloads/specializations based on type properties (introspection)

Concepts allows us to enforce an interface on a type without the overhead of inheritance.

Example...

Custom concept using the concepts library

```cpp
#include <conceptslib/concepts.hpp>

struct MeshType { };

REQUIREMENT VectorFieldReq {
    template<class T>
    auto REQUIRES(T&& t) -> decltype(concepts::valid_expr(
        t.mesh,
        concepts::valid_if<concepts::Same<decltype(t.mesh), MeshType>>()
    ));
};

template<class T>
CONCEPT IsVectorField = concepts::requires_<VectorFieldReq, T>;

struct NotVectorField { double mesh; };
struct VectorField { MeshType mesh; };

int main(){ // No hard errors
    static_assert(!IsVectorField<int>);                // has no mesh member
    static_assert(!IsVectorField<NotVectorField>); // mesh is not of MeshType
    static_assert(IsVectorField<VectorField>);         // mesh is of MeshType
}
```

Library summary

- The concepts library is based on C++17
- Models all the future C++20 concepts in header `<concepts>`
  - Core language concepts (*e.g.* `Same`, `DerivedFrom`, `ConvertibleTo`, ...)
  - Comparison concepts (*e.g.* `Boolean`, `EqualityComparable`, ...)
  - Object concepts (*e.g.* `Movable`, `Copyable`, ...)
  - Callable concepts (*e.g.* `Invocable`, `Predicate`, ...)
- Allows users to easily define new concepts
- Uses TMP techniques (SFINAE & detection idiom)
- Introduces C++20 type traits (`traits::common_reference`)

https://github.com/seriouslyhypersonic/experimental_concepts

Outline

CMatrix library (MDO GUI)

Updates:

- Build system changed to CMake
- Created FindMKL cmake module
- Works on Linux and Windows
- Does not work on macOS (library bug)

Issues:

- Probably pre-C++11
- Inefficient:
  - No support for sparse matrices (?)
  - Does not use rvalue references (unnecessary temporaries)
  - Does not meet MKL memory alignment requirements (SSE, AVX)
  - Eager evaluation generates unoptimized code
- Interface is complex and lacks uniformity

https://github.com/seriouslyhypersonic/CMatrix

Interface elegance *vs* code efficiency

Level 3 BLAS operations $T(n) = O(n^3)$
Example:

$$C \leftarrow \alpha A^T B^T + \beta C \tag{2}$$

Desired interface:

```cpp
#include <matrix.hpp>

using DMatrix = Matrix<double>;

const double alfa = 42;
const double beta = 1.618;

void example() {
    const std::size_t dim = 100;
    auto a = DMatrix::random(dim); // Same for b and c...

    c = alfa * a.transpose() * b.transpose() + beta * c;
}
```

Interface elegance *vs* code efficiency

TÉCNICO LISBOA

For an efficient implementation, the statement

```
    c = alfa * a.transpose() * b.transpose() + beta * c;
```

should be translated into a call to the specialized CBLAS function:

```
    cblas_dgemm(CblasColMajor, CblasTrans, CblasTrans
               ,dim ,dim, dim
               ,alpha
               ,a.data(), dim
               ,b.data(), dim
               ,beta
               ,c.data(), dim);
```

Overhead:

1 function call
0 temporaries

Conventional operator overloading

Due to the normal order of evaluation of the C++ language,

```
    c = alfa * a.transpose() * b.transpose() + beta * c;
```

leads to the following execution context:

```
void example() { // Assume proper initialization of a, b and, c
    DMatrix temp1 = beta * c;           // call (A): 1 copy, cblas_dscal()
    DMatrix temp2 = b.transpose();      // call (B)
    DMatrix temp3 = a.transpose();      // call (B)
    DMatrix temp4 = alfa * temp3;       // call (A): 1 copy, cblas_dscal()
    DMatrix temp5 = temp4 * temp2;      // call (C): cblas_dgemm()
    DMatrix temp6 = temp5 + temp1;      // call (D): vdAdd()
    c = temp6;                          // call (E): memcpy()
}

DMatrix operator*(double d, const DMatrix& mat);          // (A)
void DMatrix::transpose();                                 // (B)
DMatrix operator*(const DMatrix& m1, const DMatrix& m2); // (C)
DMatrix operator+(const DMatrix& m1, const DMatrix& m2); // (D)
DMatrix operator=(const DMatrix& m1, const DMatrix& m2); // (E)
```

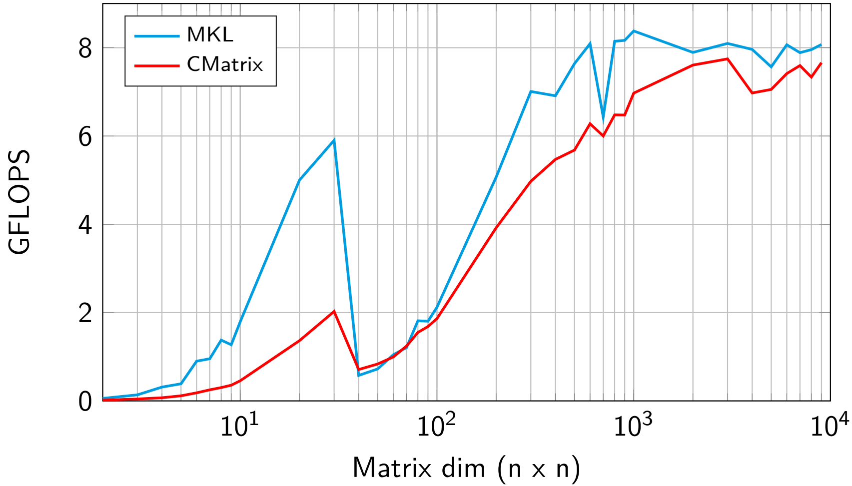
Conventional operator overloading

Overhead:

12 function calls
6 temporaries

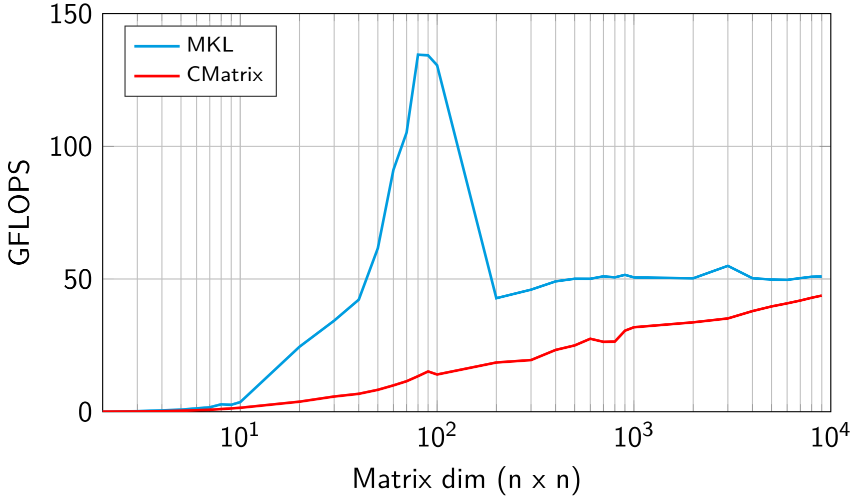
Performance comparison on a E5620 (76.8 GFLOPS, SSE4.2)

$$C \leftarrow \alpha A^T B^T + \beta C$$



Performance comparison on a i7-4770k(224 GFLOPS, AVX2)

$$C \leftarrow \alpha A^T B^T + \beta C$$



Is there a solution?

The problem is that the compiler is too eager when evaluating the rhs of

```
    c = alfa * a.transpose() * b.transpose() + beta * c;
```

What do we need?

- Bypass the normal order of evaluation of the C++ language
- An expression now represents a computation at compile time
- The result is a structure representing that computation
- Expressions are evaluated only as needed (lazy evaluation)

What will the structure look like?

```
    MatrixAdd<MatrixScale<double
                        ,MatrixMult<MatrixTranspose<DMatrix>
                                   ,MatrixTranspose<DMatrix>
                                   >
                        >
            ,MatrixScale<double, DMatrix>>
```

How is it done?

We need logic to instruct the compiler how to build the structure

$\Rightarrow$ we need TMP!
(Because we're using TMP, the structure is really just a type)

This technique is know as expression templates (ET):

- We'll abuse C++'s type system
- Use expressions to build a structure (type) during compile time
- Use that type to perform optimizations
- Only evaluate when needed (assignment triggers evaluation)

Expression templates

The machinery behind expression templates is complex:

- Curiously recurring template pattern
- Recursing down the expression tree must be done efficiently
- Different type of optimizations:
    - Recursive nature (*e.g.* loop fusion)
    - Specializations (*e.g.* calls to MKL functions)

Once implemented:

- Elegant interface generates efficient code
- ET can be used to accelerate automatic differentiation
- Will be possible to embed DSLs (*à la* OpenFOAM)
- Optimizations and parsing happens at compile time