

Java to C++ (for NumCS)
3. Templates

Felix Friedrich
Malte Schwerhoff
D-INFK 2022

Generic Programming

Goal: Make code (functions, classes) as **widely usable/applicable** as possible

Familiar use-cases:

- `std::set<T>` — store objects of any type `T` (e.g. ints, strings, matrices, ...)
- `std::max(e1, e2)` — take the larger of any two comparable objects
- `Exp e = ...; e.eval()` — evaluate any expression (e.g. `Var`, `Add`, `Cos`, ...)

Generic Programming

How: Different languages implement different approaches (and typically more than one), but high-level ideas are often similar

Examples:

compile-time meta-programming



C++, Rust, Scala

run-time meta-programming



Java, Python, JavaScript

dynamic typing



Python, JavaScript

functional programming



Haskell, C++, JavaScript,
Python, Scala, Java

object-oriented programming



Java, Python, Scala, C++

...

Motivational Example

Goal: Generic class and functionality for matrices (and vectors), without duplicating code!

```
class Matrix { ... };  
  
auto m1 = Matrix<int>(5,3);  
auto m2 = Vector<std::string>("Zürich");  
auto m3 = Matrix<Complex>(...);  
  
m1(1,2) = 10;  
auto sum = m1 + m2 + m3;  
std::cout << m3.max(); // oops
```

Class Templates

1. Given an implementation, e.g. `class Matrix`, for a single, specific element type, e.g. `int`, replace each occurrence of the element type with a placeholder, e.g. `T`
2. Prepend the class with `template<typename T>`
(replace `T` with the placeholder you chose)

The syntax `template<typename T>` can be read as “for all types `T`”.

Example

```
// Matrix for integers
class Matrix {
    unsigned sizeR, sizeC;
    std::vector<int> data;
public:
    Matrix(unsigned R, unsigned C): sizeR(R), sizeC(C), data(R*C) {}

    int& operator() (unsigned r, unsigned c){
        assert ( r < sizeR && c < sizeC);
        return data[r*sizeC + c];
    }

    const int& operator() (unsigned r, unsigned c) const { .. }

    unsigned rows() const {
        return sizeR;
    }
    ...
};
```

Example

```
template <typename T>
class Matrix {
    unsigned sizeR, sizeC;
    std::vector<T> data;
public:
    Matrix(unsigned R, unsigned C): sizeR(R), sizeC(C), data(R*C) {}

    T& operator() (unsigned r, unsigned c){
        assert ( r < sizeR && c < sizeC);
        return data[r*sizeC + c];
    }

    const T& operator() (unsigned r, unsigned c) const { .. }

    unsigned rows() const {
        return sizeR;
    }
    ...
};
```

Template Instance

- `Matrix<type>` creates a **type-specific instance** of `Matrix`, using the substitution `T=type`. Therefore, `Matrix<T>` is sometimes called a *type constructor*.

```
Matrix<int> m1 = ...;  
Matrix<std::string> m2 = ...;  
Matrix<Complex> m3 = ...;
```

- The **compiler generates the code** of each instantiated class for us

m1

```
class Matrix_int {  
    ...  
    std::vector<int> data;  
    ...  
    int operator() ...  
};
```

m2

```
class Matrix_string {  
    ...  
    std::vector<std::string> data;  
    ...  
    std::string operator() ...  
};
```

m3

```
class Matrix_complex {  
    ...  
    std::vector<Complex> data;  
    ...  
    Complex operator() ...  
};
```

Function Templates: Analogously

1. Given an implementation, e.g. `add(matrix1, matrix2)`, for a single, specific element type, e.g. `int`, replace each occurrence of the element type with a placeholder, e.g. `T`
2. Prepend the function with `template<typename T>`
(replace `T` with the placeholder you chose)

The syntax `template<typename T>` can be read as “for all types `T`”.

```
template <typename T>
Matrix<T> add(Matrix<T> left, Matrix<T> right){
    ...
}
```

Function Templates: Example Stubs

- For free functions

```
template <typename T>
void swap(T& x, T& y) {
    T temp = x;
    x = y;
    y = temp;
}
```

```
template <typename Iter>
bool is_sorted(Iter begin, Iter end) {
    ...
}
```

- For operators

```
template <typename T>
ostream& operator<<(ostream& out,
                   const Matrix<T>& m) {
    ...
}
```

- For member functions

```
template <typename E>
class vector {
    ...
    template <typename C>
    void push_back_all(const C& other) {...}
};
```

Function Templates: Example

```
template <typename C>
void print(const C& container) {
    for (auto& elem : container)
        std::cout << elem << ' ';
}

// main
std::vector<int> data = {1, 2, 3};
print(data);
```

- Neat! A generic **print** function, applicable to all containers
- Can you see potential problems when invoking **print(...)**?
 - Requirements on container type (here **C**)?
 - Requirements on element type (here **auto**)?

Type Checking

- Templates: syntactic checks, some type checking (depends on compiler)
- Instances: checked as usual

```
template <typename T>
T abs(T v) {
    return 0 <= v ? v : -v;
}

abs(8); // OK
```

```
template <typename T>
T abs(T v) {
    return 0 <= v ? v : -v; // Error
}

abs("hi"); // Error
```

```
template <typename T>
void swap(T& x, T& y) {
    ...
}

double a = 1.0;
double b = 7;
swap(a, b);
```

```
template <typename T>
void swap(T& x, T& y) {
    ...
}

double a = 1.0;
std::string b = "seven";
swap(a, b);
```

```
template <typename T>
void swap(T& x, T& y) {
    ...
}

double a = 1.0;
int b = 7;
swap(a, b);
```

Type Checking

- Templates: syntactic checks, some type checking
- Instances: checked as usual

```
vector<int> vec = {1, 2, 3};  
print(vec); // OK
```

```
vector<vector<int>> matrix =  
    {{1,0,1}, {1,1,0}};  
print(matrix); // Error
```

```
Student winner = ...;  
print(winner); // Error
```

```
template <typename C>  
void print(const C& container) {  
    for (auto& elem : container)  
        std::cout << elem << ' ' ;  
}
```

`operator<<` not overloaded for `std::vector`

`class Student` not iterable (doesn't implement a suitable iterator)

Type Inference

- Upon instantiation, template parameters must be provided explicitly — except if the compiler can infer them from provided arguments
- Type inference improved with C++17

```
template <typename T>
T max(T v1, T v2) {
    return v1 < v2 ? v2 : v1;
}
```

```
auto v1 = max(-2, 8); // OK
```

```
template <typename T>
Pair<T> read() { ... }
```

```
auto q1 = read<int>(); // OK
Pair<int> q2 = read(); // Error
```

```
template <typename T>
struct Pair {
    T fst, snd;
    Pair(T f, T s): fst(f), snd(s) {}
};
```

```
auto p1 = Pair<int>(1,2); // OK
Pair<int> p2 = Pair(1,2); // C++14: error; C++17: OK
auto p3 = Pair(1,2); // C++14: error; C++17: OK
```

Functional Programming (next lecture)

- Common task: filter values satisfying a predicate
- Example: filter non-negative values
- Observation: **code duplication**, if predicate changes
- Idea: make filter function **parametric** in its predicate (boolean function)

```
std::vector<int> src_data = ...;

std::vector<int> data = {};
for (const auto& e : src_data)
    if (0 <= e) data.push_back(e);
```

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
    C data = {};

    for (const auto& e : src_data)
        if (pred(e)) data.push_back(e);

    return data;
};
```

```
bool is_nonneg(int i) { return 0 <= i; } // a predicate

std::vector<int> src_data = ...;
auto data = filter(src_data, is_nonneg);
```

```
// further predicates
bool is_even(int i) { return i % 2 == 0; }
bool is_prime(unsigned int i) { ... }
bool has_legi_format(std::string text) { ... }
bool has_sufficient_XP(Student student) { ... }
```

Specialisation

- **Basic idea:** **general** implementation for **arbitrary** types, but **specialised** (usually: more efficient) implementation for **specific** types
- Applicable to data structures (e.g. more space-efficient internal representation) and algorithms (e.g. more efficient sorting)
- Examples:
 - General **vector<T>**, but space-compressed **vector<bool>**
 - Quicksort for a random-access container, mergesort otherwise

Specialisation

- **Basic idea:** general implementation for arbitrary types, but specialised (usually: more efficient) implementation for specific types
- Syntax example (for classes; analogous for functions)

```
// General implementation of template class  
// (analogous syntax for template functions)  
template<typename T>  
class my_class {  
    ...  
};  
  
// A specialised implementation for  
// characters  
template<>  
class my_class<char> {  
    ...  
};  
  
// ... further specialisations ...
```

```
auto mc1 =  
    my_class<int>(...) // General implementation  
  
auto mc2 =  
    my_class<char>(...) // Specialised impl.
```

Specialisation

vector<T>

```
// Default implementation of a very basic
// vector (no bounds checks, doesn't grow,
// little functionality)
template<typename E>
class vector {
    unsigned size;
    E* elements;

public:
    vector(): size(128), elements(new E[size]) {}

    void set(unsigned idx, const E& element) {
        elements[idx] = element;
    }

    const E& get(unsigned idx) {
        return elements[idx];
    }
};
```

pointer used for dynamic allocation, please ignore the details

vector<bool> (typically 1/8 memory)

```
template<>
class vector<bool> {
    static const auto bits_per_uint = 8 * sizeof(unsigned);
    unsigned size;
    unsigned* blocks;

public:
    vector():
        size(128),
        blocks(new unsigned[std::ceil((float)size / bits_per_uint)]{})
    {}

    void set(unsigned i, bool element) {
        unsigned& block = *(blocks + i/bits_per_uint);
        unsigned k = i % bits_per_uint;
        block = (block & ~(1u << k)) | element << k;
    }

    bool get(unsigned i) {
        return 1u & (blocks[i / bits_per_uint] >> (i % bits_per_uint));
    }
};
```

Specialisation

Specialisation has many different use cases, and can come in different forms:

- **Generic, with specialisations** (just seen):
e.g. generic implementation `vector<T>`, and specialisation for `vector<bool>`
- **Enabling, via specialisations**:
e.g. `std::hash<T>`, used e.g. by `std::unordered_set`, has no generic implementation, must be specialised for custom objects.
- **Specialisation for statically known values**:
e.g. `std::array<T, n>` where `n` is the statically known size of the array. This is basically a fully-statically allocated vector: it does not entail dynamic memory allocation, but it also can't grow/shrink.

Java Generics vs. C++ Templates

Syntactically *quite* similar

C++

```
template <class T>
class Pair {
private:
    T left;
    T right;
public:
    Pair(T L, T R): left(L), right(R) {}
    ...
}

Pair<int> pair(3,5);
```

Java

```
public class Pair<T>{
    private T left;
    private T right;

    public Pair(T L, T r){
        left = l;
        right = r;
    }
    ...
}

Pair<Integer> pair = new Pair<>(3,5);
```

Java Generics vs. C++ Templates

Semantically *quite* different

C++

- Templates, based on type replacements and pattern-matching, can only be rudimentary checked in isolation.
- All type checking is delayed to when instances are created.
- Template sources must be included.
- Template specialisation is possible. Code can be statically optimized for concrete types.

Java

- Java Generics can be checked in isolation. Extensibility and type checking is based on OOP (subtyping).
- Generics are checked at compile-time for correctness.
- Cannot be used for non-objects.
- Generic Type-Information is removed for instances (type-erasure): generic type information not available at runtime.

Templates Parameterized with Values

- Eigen supports matrices with statically or dynamically allocated sizes
 - for optimisation of operations (e.g. 3x3 matrix multiply vs. matrix multiplication with arbitrary size)
 - for representation of special shapes (e.g. vectors stored as matrices)

```
typedef Matrix<double, Dynamic, Dynamic> MatrixXd;  
typedef Matrix<float, 3, 1> Vector3f  
typedef Matrix<int, 1, 2> RowVector2i;
```

Templates Parameterized with Values

- Eigen can check some things during compilation time:

```
Matrix3d m;
```

```
m[1] = 3;
```

```
/usr/include/eigen3/Eigen/src/Core/util/StaticAssert.h:33:40: error: static assertion failed:  
THE_BRACKET_OPERATOR_IS_ONLY_FOR_VECTORS__USE_THE_PARENTHESIS_OPERATOR_INSTEAD  
#define EIGEN_STATIC_ASSERT(X,MSG) static_assert(X,#MSG);  
                                   ^
```

```
/usr/include/eigen3/Eigen/src/Core/DenseCoeffsBase.h:406:7: note: in expansion of macro 'EIGEN_STATIC_ASSERT'  
EIGEN_STATIC_ASSERT(Derived::IsVectorAtCompileTime,  
  ^~~~~~
```

```
Compilation failed
```

To get some Intuition ...

- Let's try to understand how things might be implemented in Eigen
- Eigen source code is written to be efficient and generic, using templates and macros, which makes it not easy to read.
- The documentation, however, is very useful.

Fixed vs. Dynamic size

When should one use fixed sizes (e.g. `Matrix4f`), and when should one prefer dynamic sizes (e.g. `MatrixXf`)? The simple answer is: use fixed sizes for very small sizes where you can, and use dynamic sizes for larger sizes or where you have to. For small sizes, especially for sizes smaller than (roughly) 16, using fixed sizes is hugely beneficial to performance, as it allows **Eigen** to avoid dynamic memory allocation and to unroll loops. Internally, a fixed-size **Eigen** matrix is just a plain array, i.e. doing

```
Matrix4f mymatrix;
```

really amounts to just doing

```
float mymatrix[16];
```

so this really has zero runtime cost. By contrast, the array of a dynamic-size matrix is always allocated on the heap, so doing

```
MatrixXf mymatrix(rows, columns);
```

amounts to doing

Static and Dynamic Sizes

```
const unsigned Dynamic = -1;

template <typename T, unsigned SizeR, unsigned SizeC>
class Matrix {
public:
    Matrix(unsigned R, unsigned C): rows(R), cols(C), data(R*C) {
        assert(SizeR == Dynamic || SizeR == R); assert(SizeC == Dynamic || SizeC == C);
    }
    ...

    T& operator() (unsigned pos){
        static_assert(SizeR == 1 || SizeC == 1 , "must be a vector");
        ---
    }

    T& operator() (unsigned r, unsigned c){
        assert ( r < rows && c < cols && "out of bounds");
        return data[r*cols + c];
    }

private:
    unsigned rows, cols;
    std::vector<T> data;
};
```

static
check

dynamic
check

Operators

- Eigen supports a special comma-initializer for matrices

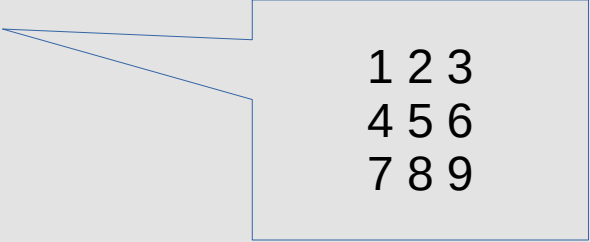
```
Eigen::Matrix3d m3;
```

```
m3 << 1, 2, 3,
```

```
      4, 5, 6,
```

```
      7, 8, 9;
```

```
std::cout << m3;
```



```
1 2 3  
4 5 6  
7 8 9
```

Operators, iterators and all that

```
template <typename Iterator>
class Initializer {
using ValueT = typename std::iterator_traits<Iterator>::value_type;
public:
    Initializer(Iterator Begin, Iterator End): it(Begin), end(End){ }

    void push_back(ValueT v){
        assert(it != end);
        *it = v;
        ++it;
    }

    Initializer& operator=(ValueT v){
        push_back(v);
        return *this;
    }

private:
    Iterator it;
    Iterator end;
};
```

```
m << 1 , 2 , 3 ..
<==>
operator,(operator,(operator<<(m,1),2),3)
```

```
template <typename T>
class Matrix {
    ....

    using IteratorT = typename std::vector<T>::iterator;

    Initializer<IteratorT> operator<<(T scalar){
        Initializer<IteratorT> init(data.begin(), data.end());
        init.push_back(scalar);
        return init;
    }
};
```

Advanced Optimisations

- Eigen uses *Expression Templates* for the evaluation of complicated expressions.
- From the documentation:

```
VectorXf a(50), b(50), c(50), d(50);  
...  
a = 3*b + 4*c + 5*d;
```

Eigen compiles it to just one for loop, so that the arrays are traversed only once. Simplifying (e.g. ignoring SIMD optimizations), this loop looks like this:

```
for(int i = 0; i < 50; ++i)  
    a[i] = 3*b[i] + 4*c[i] + 5*d[i];
```

Example

Using operator overloading, an expression like this

```
m = a + b + c;
```

will be translated into this

```
operator=(m,operator+(operator+(a,b),c));
```

Example

Assume operator+() looks like this

```
Matrix operator+(const Matrix& left, const Matrix& right){  
    Matrix result(left.rows(), left.columns());  
    for (unsigned r = 0; r<result.rows(); ++r){  
        for (unsigned c = 0; c< result.columns(); ++c){  
            result(r,c) = left(r,c) + right(r,c);  
        }  
    }  
    return result;  
}
```

Then

```
operator=(m,operator+(operator+(a,b),c);
```

will create at least **two temporary results** and **execute the nested loop twice**.

Example

Would this not be preferable?

```
operator=(m, add3(a, b, c));
```

```
Matrix add3(const Matrix& A, const Matrix& B, const Matrix& C){  
    Matrix result(a.rows(), b.columns());  
    for (unsigned r = 0; r < result.rows(); ++r){  
        for (unsigned c = 0; c < result.columns(); ++c){  
            result(r, c) = A(r, c) + B(r, c) + C(r, c);  
        }  
    }  
    return result;  
}
```

(but how to **automatically** infer this from the expression $m = a + b + c$?)

C++ Templates to the Rescue

This is a case for **expression templates** (aka loop-fusion).

The trick is to use lazy evaluation, i.e. to delay the computation of a large expression to the point where the result is required, but statically -- during compile-time.

The following is a working but simplified approach, intended to illustrate the basic idea.

Lazy Evaluation

The evaluation of the expression happens when the result is assigned:

```
template <typename T>
class Matrix {
public:
    template <typename Expression>
    Matrix& operator=(const Expression& e) {
        assert(e.rows() == rows() && e.cols() == cols());
        for (unsigned r = 0; r < rows(); ++r){
            for (unsigned c = 0; c < cols(); ++c){
                (*this)(r,c) = e(r,c); // element-wise evaluation of e here
            }
        }
        return *this;
    }
    ...
};
```

The Expression for Addition

```
template <typename lExpr, typename rExpr>
class PlusExpression {
public:
    PlusExpression(const lExpr &lxpr, const rExpr &rxpr) : left(lxpr), right(rxpr) {};

    unsigned rows() const { return left.rows(); }
    unsigned cols() const { return left.cols(); }

    auto operator()(unsigned r, unsigned c) const {
        return left(r,c) + right(r,c);
    };
private:
    lExpr left;
    rExpr right;
};
```

interface shared with Matrix

element-wise computation

Operators

```
// Plus operator overload on expressions
template<typename lExpr, typename rExpr>
auto operator+(const lExpr &lhs, const rExpr &rhs) {
    return PlusExpression<lExpr,rExpr>(lhs,rhs);
}
```

The expression $A + B + C$ is of type

`PlusExpression<PlusExpression<Matrix<int>,Matrix<int>>,Matrix<int>>`

The evaluation of Expression e (in the overloaded assignment operator) thus expands as

```
e(r,c)
→ e.left(r,c) + e.right(r,c)
→ (e.left.left(r,c) + e.left.right(r,c)) + e.right(r,c)
= (A(r,c) + B(r,c)) + C(r,c)
```

Why is this a simplification?

The overloading of plus above is overly generic. The compiler will try to apply it to any object. So, it needs to be limited to our expression objects (like Matrix and PlusExpression).

But then Matrix and PlusExpression need to be of same base type. This can be solved using (static) polymorphism with the so called Curiously Recurring Template Pattern.

An interesting article on expression templates (beyond the scope of this course) can be found here:

<https://devtut.github.io/cpp/expression-templates.html#basic-expression-templates-on-element-wise-algebraic-expressions>

Templates Conclusion I

- Templates **improve code reusability**, by making classes and functions parametric w.r.t. types
- Implemented via **static code-generation (compile-time meta-programming)**
 - Advantages
 - **No runtime overhead** (compared to dynamic solutions, e.g. inheritance or dynamic typing)
 - Compiler can optimise generated code as usual
 - Type-dependent static specialisation possible (illustrated soon)
 - Disadvantages
 - Separate compilation (**.h** vs. **.cpp**) no longer possible
 - Resulting binary (machine code) larger
 - Delayed type checking, more complex error messages (mitigatable via concepts, see next)