C++ Training

Concepts

Olivier Parcollet

Center for Computational Quantum Physics (CCQ)

Flatiron Institute, Simons Foundation

New York

FLATIRON INSTITUTE
Division of Simons Foundation

Outline

- Motivation : expression template for arrays
- Basic of generic programming
- The issue with generic programming : the need for concepts
- A case study : trace (A + B)
- Type erasure : stop template proliferation !

Motivation

Zero cost abstraction

- What is *simple* should be coded *simply*
- High level and yet fast.

- Common wrong idea : compact, simple, readable code is slow.
- We want simplicity (abstraction), without any performance penalty (at zero cost).

Motivation

- A, B, C, Z: arrays of rank 5 e.g. We want to say

```
Z = A + B + C / 2;
```

- Naive object oriented way :
  - Each addition makes a new array
  - Slow : a lot of temporaries and loops !

$$Z = \underbrace{A + \underbrace{B + \underbrace{C/2}}}$$

Motivation

- A, B, C, Z: arrays of rank 5 e.g. We want to say

```
Z = A + B + C / 2;
```

- A basic answer is : write all the loops !

```
for (int i = 0; i < b1; ++i)
 for (int j = 0; j < b2; ++j)
  for (int k = 0; k < b4; ++k)
   for (int l = 0; l < b3; ++l)
    for (int m = 0; m < b5; ++m) {
     Z(i, j, k, l, m) =
         A(i, j, k, l, m) + B(i, j, k, l, m) + C(i, j, k, l, m) / 2;
    }
```

- Error prone, hard to read and code review. Memory traversal ?
- The compiler should do this for us !

Other example

- With our multidimensional array class

```
auto a = array<double, 3>(5, 2, 2);   // Declare a 5x2x2 array of double

sum(a * a);           // Sum all the square elements

max_element(abs(a)); // maximum of the absolute value of the array
```

- Rewriting it manually requires the code of *sum*

Laziness is good !

- Naive object oriented way :
  - The addition, etc returns a new array (e.g. simple object oriented)

$$Z = \underbrace{A + \underbrace{B + \underbrace{C/2}_{}}_{}}_{}$$

  - A lot of temporaries and loops !
  - It hurts performance a lot.

- "Lazy" way :
  - We can make such code work without a performance penalty.
  - Mainstream and natural technique in modern C++.
  - Generic programming plays an essential role.

Generic programming

Generic programming : template function

- Question : write a (trivial) function sqr that computes the square for int, double, complex
- Solution 1 (C): several functions. Cf std::abs, std::fabs

```
int sqr_i(int x) {
 return x*x;
}

double sqr_f(double x) {
 return x*x;
}

double sqr_c(std::complex<double> x) {
 return x*x;
}
```

Generic programming : template function

- Question : write a (trivial) function sqr that computes the square for int, double, complex
- Solution 2 (C++): overload. Easier to call : sqr(x)

```
int sqr(int x) {
 return x*x;
}

double sqr(double x) {
 return x*x;
}

double sqr(std::complex<double> x) {
 return x*x;
}
```

Generic programming : template function

- Question : write a (trivial) function sqr that computes the square for int, double, complex

- Solution 3 (C++): template.

```
template<typename T>
T sqr (T x) {
 return x*x;
}
```

- Let the compiler do the job !

Generic programming : template class/struct

- std::vector<T>
- Valid for any type T (with basic requirement, in particular regular types).
- Example : a template class of matrix

```
template <typename T> class square_matrix {
 int n;                             // the dimension
 std::vector<T> data; // a place to store the data

 public:
 // ... the rest of the class
};
```

STL

- Standard C++ library is made of
  - Containers (data structure)
  - Generic algorithms
- Separation of data and algorithms

Main idea of generic programming

What matters is how an object behaves, not what it is.

```
template<typename T>
T sqr (T x) {
 return x*x;
}
```

The problem with generic programming …

When an error occurs …

```
1  template<typename T>
2  T sqr( T const & x)  {
3    return x*x;
4  }
5
6  struct A {
7  };
8
9  int main()  {
10
11  A a;
12  auto s = sqr(a);
13 }
```

A object can not be multiplied !!

```
Mac:~/ clang++ -std=c++17 sqr.cpp
sqr.cpp:4:11: error: invalid operands to binary expression ('const A' and 'const A')
  return x*x;
         ~^~
sqr.cpp:15:11: note: in instantiation of function template specialization 'sqr<A>' requested here
 auto s = sqr(a);
          ^

1 error generated.
```

When an error occurs …

- Error messages can be horribly long

```cpp
#include <vector>
#include <algorithm>

struct A  {
  int n;
};

int main() {
  std::vector<A> v(10);
  std::sort(begin(v), end(v));
}
```

```
clang++ -std=c++17 long_error.cpp
```

!?

```
Mac:~/Dropbox (Simons Foundation)/TALKS/TrainingC++ clang++ -std=c++17 long_error.cpp
In file included from long_error.cpp:1:
In file included from /usr/local/Cellar/llvm/7.0.1/include/c++/v1/vector:270:
In file included from /usr/local/Cellar/llvm/7.0.1/include/c++/v1/__bit_reference:15:
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:721:71: error: invalid operands to binary expression
      ('const A' and 'const A')
    bool operator()(const _T1& __x, const _T1& __y) const {return __x < __y;}
                                                                  ~~~ ^ ~~~
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:3931:17: note: in instantiation of member function
      'std::__1::__less<A, A>::operator()' requested here
            if (__comp(*--__last, *__first))
                ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:4117:5: note: in instantiation of function template
      specialization 'std::__1::__sort<std::__1::__less<A, A> &, A *>' requested here
    __sort<_Comp_ref>(__first, __last, __comp);
    ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:4126:12: note: in instantiation of function template
      specialization 'std::__1::sort<A *, std::__1::__less<A, A> >' requested here
    _VSTD::sort(__first, __last, __less<typename iterator_traits<_RandomAccessIterator>::value_type>());
           ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:4142:12: note: in instantiation of function template
      specialization 'std::__1::sort<A *>' requested here
    _VSTD::sort(__first.base(), __last.base());
           ^
long_error.cpp:12:8: note: in instantiation of function template specialization 'std::__1::sort<A>' requested here
  std::sort(begin(v), end(v));
       ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/utility:574:1: note: candidate template ignored: could not match
      'pair<type-parameter-0-0, type-parameter-0-1>' against 'const A'
operator< (const pair<_T1,_T2>& __x, const pair<_T1,_T2>& __y)
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/iterator:702:1: note: candidate template ignored: could not match
      'reverse_iterator<type-parameter-0-0>' against 'const A'
operator<(const reverse_iterator<_Iter1>& __x, const reverse_iterator<_Iter2>& __y)
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/iterator:1143:1: note: candidate template ignored: could not match
      'move_iterator<type-parameter-0-0>' against 'const A'
operator<(const move_iterator<_Iter1>& __x, const move_iterator<_Iter2>& __y)
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/iterator:1515:1: note: candidate template ignored: could not match
      '__wrap_iter<type-parameter-0-0>' against 'const A'
operator<(const __wrap_iter<_Iter1>& __x, const __wrap_iter<_Iter2>& __y) _NOEXCEPT_DEBUG
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/tuple:1182:1: note: candidate template ignored: could not match
      'tuple<type-parameter-0-0...>' against 'const A'
operator<(const tuple<_Tp...>& __x, const tuple<_Up...>& __y)
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/memory:2956:1: note: candidate template ignored: could not match
      'unique_ptr<type-parameter-0-0, type-parameter-0-1>' against 'const A'
operator< (const unique_ptr<_T1, _D1>& __x, const unique_ptr<_T2, _D2>& __y)
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/memory:3014:1: note: candidate template ignored: could not match
      'unique_ptr<type-parameter-0-0, type-parameter-0-1>' against 'const A'
operator<(const unique_ptr<_T1, _D1>& __x, nullptr_t)
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/memory:3023:1: note: candidate template ignored: could not match
      'unique_ptr<type-parameter-0-0, type-parameter-0-1>' against 'const A'
operator<(nullptr_t, const unique_ptr<_T1, _D1>& __x)
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/memory:4794:1: note: candidate template ignored: could not match
      'shared_ptr<type-parameter-0-0>' against 'const A'
operator<(const shared_ptr<_Tp>& __x, const shared_ptr<_Up>& __y) _NOEXCEPT
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/memory:4864:1: note: candidate template ignored: could not match
      'shared_ptr<type-parameter-0-0>' against 'const A'
operator<(const shared_ptr<_Tp>& __x, nullptr_t) _NOEXCEPT
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/memory:4872:1: note: candidate template ignored: could not match
      'shared_ptr<type-parameter-0-0>' against 'const A'
operator<(nullptr_t, const shared_ptr<_Tp>& __x) _NOEXCEPT
^
In file included from long_error.cpp:1:
In file included from /usr/local/Cellar/llvm/7.0.1/include/c++/v1/vector:270:
In file included from /usr/local/Cellar/llvm/7.0.1/include/c++/v1/__bit_reference:15:
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:3718:20: error: no matching function for call to '__sort3'
    unsigned __r = __sort3<_Compare>(__x1, __x2, __x3, __c);
                   ^~~~~~~~~~~~~~~~~
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:3938:20: note: in instantiation of function template
      specialization 'std::__1::__sort4<std::__1::__less<A, A> &, A *>' requested here
            _VSTD::__sort4<_Compare>(__first, __first+1, __first+2, --__last, __comp);
                   ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:4117:5: note: in instantiation of function template
      specialization 'std::__1::__sort<std::__1::__less<A, A> &, A *>' requested here
    __sort<_Comp_ref>(__first, __last, __comp);
    ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:4126:12: note: in instantiation of function template
      specialization 'std::__1::sort<A *, std::__1::__less<A, A> >' requested here
    _VSTD::sort(__first, __last, __less<typename iterator_traits<_RandomAccessIterator>::value_type>());
           ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:4142:12: note: in instantiation of function template
      specialization 'std::__1::sort<A *>' requested here
    _VSTD::sort(__first.base(), __last.base());
           ^
long_error.cpp:12:8: note: in instantiation of function template specialization 'std::__1::sort<A>' requested here
  std::sort(begin(v), end(v));
       ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:3678:1: note: candidate template ignored: substitution
      failure [with _Compare = std::__1::__less<A, A> &, _ForwardIterator = A *]
__sort3(_ForwardIterator __x, _ForwardIterator __y, _ForwardIterator __z, _Compare __c)
^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:3744:20: error: no matching function for call to '__sort4'
    unsigned __r = __sort4<_Compare>(__x1, __x2, __x3, __x4, __c);
                   ^~~~~~~~~~~~~~~~~
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:3941:20: note: in instantiation of function template
```

A closer look

- Always read the error from the start ! Still, it is not nice.

```
clang++ -std=c++17 long_error.cpp
In file included from long_error.cpp:1:
In file included from /usr/local/Cellar/llvm/7.0.1/include/c++/v1/vector:270:
In file included from /usr/local/Cellar/llvm/7.0.1/include/c++/v1/__bit_reference:15:
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:721:71: error: invalid operands to binary expression
      ('const A' and 'const A')
    bool operator()(const _T1& __x, const _T1& __y) const {return __x < __y;}
                                                                  ~~~ ^ ~~~
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:3931:17: note: in instantiation of member function
      'std::__1::__less<A, A>::operator()' requested here
            if (__comp(*--__last, *__first))
                ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:4117:5: note: in instantiation of function template
      specialization 'std::__1::__sort<std::__1::__less<A, A> &, A *>' requested here
    __sort<_Comp_ref>(__first, __last, __comp);
    ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:4126:12: note: in instantiation of function template
      specialization 'std::__1::sort<A *, std::__1::__less<A, A> >' requested here
    _VSTD::sort(__first, __last, __less<typename iterator_traits<_RandomAccessIterator>::value_type>());
           ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/algorithm:4142:12: note: in instantiation of function template
      specialization 'std::__1::sort<A *>' requested here
    _VSTD::sort(__first.base(), __last.base());
           ^
long_error.cpp:12:8: note: in instantiation of function template specialization 'std::__1::sort<A>' requested here
  std::sort(begin(v), end(v));
       ^
/usr/local/Cellar/llvm/7.0.1/include/c++/v1/utility:574:1: note: candidate template ignored: could not match
      'pair<type-parameter-0-0, type-parameter-0-1>' against 'const A'
```

Why ?

- Because you see all the internal of the implementation
- It calls several functions which are implementation details and …
  …. at some point the operator < is missing !
- Same problem in Python (but at runtime).
  Long obscure traces of errors …

```
template<typename T>
T sqr (T const & x) {
 return x*x;
}
```

- The problem is to check that T satisfies some requirements *before* generating the code.

The notion of concept

- A concept is a set of constraints for a type.
- Example:
  - The element (of vector<A>, i.e. A) must be comparable
  - x<y must make sense.
- The generic algorithm only applies to the *category of types which models the concepts* (i.e. satisfy the constraints).

- C++20 : concept checked by compilers.
  Much better error messages

```
template<Sortable C>
void sort(C & c);
```

STL

- Concepts are an old idea in C++ (A. Stepanov, STL).
- STL is entirely based on concepts (iterators, e.g.) and generic algorithms.
- Until C++20 however, it was not explicit for the compilers, only written in documentation.

Case study

Zero cost abstraction and concepts

The puzzle

- A and B : two matrices n x n, real valued.
  A function ***trace***

- We want to write

```
double r = trace (A + B);
```

- Instead of

```
double r = 0;
for (int i = 0; i < n; ++i)
      r += A(i, i) + B(i, i);
```

- A priori, zero cost abstraction seems impossible:
  - A + B computed first, before calling `trace`.
  - Scales as N^2 while hand-written code is N

The trace function

- Assume we have a *square_matrix* class
- Let us implement the trace

```
double trace (square_matrix const & m) {
 double r = 0;
 int d = dim(m); // size of the matrix d x d
 for (int i=0; i<d; ++i) r += m(i,i);
 return r;
}
```

- Only things I used here :
  - m(i,j) returns the value of the matrix m_{ij}
  - dim(m) returns the dimension

Generic programming

- A generic version of the function

```
template<typename M>
double trace (M const & m) {
 double r = 0;
 int d = dim(m); // size of the matrix d x d
 for (int i=0; i<d; ++i) r += m(i,i);
 return r;
}
```

- What can M be ?
  - m(i,j) returns the value of the matrix m_{ij}
  - dim(m) returns the dimension
- *trace* makes sense (i.e. compiles) only when these constraints on M are true

The Matrix concept

- Main idea of generic programming:

  What matters is how an object behaves, not what it is.

- Example: Matrix concept

  The category of types that behave like a square matrix (of double)

  - m(i,j) returns the value of the matrix m_{ij}
  - dim(m) returns the dimension

- Trace will work for any type modelling Matrix concept

A simple square matrix

```
class square_matrix {
 int n;
 std::vector<double> data;

 public:
 square_matrix(int n);

 double operator()(int i, int j) const { return data[i + n * j]; }
 friend int dim(square_matrix const& m) { return m.n; }

 // Access to modify the matrix : not required by Matrix
 double & operator()(int i, int j) { return data[i + n * j]; }
};
```

- Models Matrix concept

Back to our question

```
double r = trace (A + B);
```

- The sum of 2 matrices is a lazy object that :
  just keeps a reference to A, B
  evaluates the actual sum only on demand.

```
template <typename A, typename B> struct lazy_addition {
 A const & a;
 B const & b;
 double operator()(int i, int j) const { return a(i, j) + b(i, j); }
 friend int dim(lazy_add const& x) { return dim(x.a); }
};
```

- Models Matrix concept

```
template <typename A, typename B>
lazy_addition<A, B> operator+(A const& a, B const& b){
 return {a, b};
}
```

- NB The addition is too general, it takes any type ! We will fix later…

What does the compiler do ?

```
double trace(lazy_addition const& m) {
 auto r = m(0, 0);
 int d = dim(m);
 for (int i = 1; i < d; ++i) r += m(i, i);
 return r;
}
```

```
template <typename A, typename B>
struct lazy_addition {
 A const& a;
 B const& b;

 double operator()(int i, int j)
const { return a(i, j) + b(i, j); }
```

- Replace and inline calls

```
double trace(lazy_addition const& m) {
 auto r = 0;
 int d = dim(m.a);
 for (int i = 0; i < d; ++i) r += m.a(i, i) + m.b(i,i);
 return r;
}
```

- The compiler rewrites the code for us
  - Exactly the hand written code
  - Scales like N, not N^2

Let us check

- Compare 3 code snippets (with Google Benchmarks)
  - With Trace (TRIQS library)

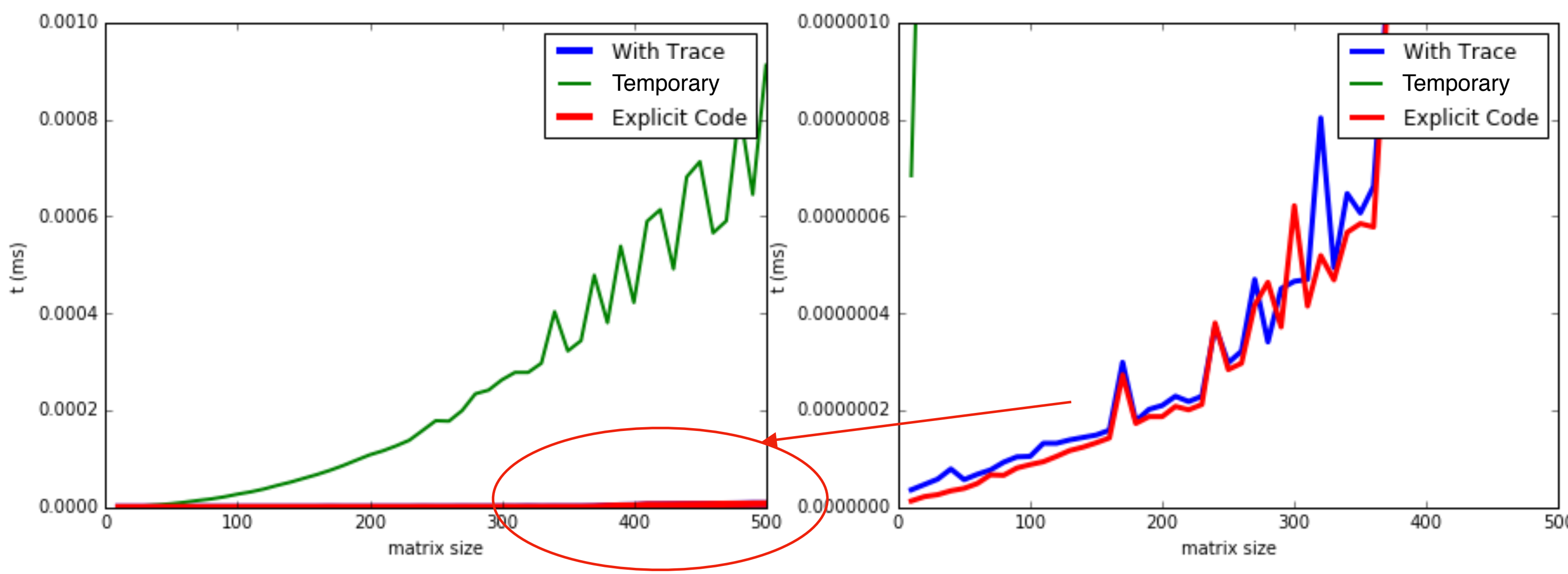
```
auto r = trace(A + B);
```

  - Explicit code (hand written)

```
for (int i = 0; i < N; ++i) r += A(i, i) + B(i, i);
```

  - Force temporary

```
auto r = trace(square_matrix{A + B});
```



Concept

Use C++20 feature to inform the compiler

Concept in C++20

- C++20 : make the compiler aware of concepts
- Matrix concept in C++20, e.g.

```
template <typename T> concept Matrix = requires(T m) {
  { m(0, 0)} -> std::convertible_to<double>;
  { dim(m) } -> std::convertible_to<int>;
 };
```

- *C++17 : simple "requires" implemented in gcc and clang, we can use concepts today, but I am showing the C++20 syntax here.*

Let us add concept to our code …

- The addition is restricted to types with the proper concept

```
template <Matrix A, Matrix B>
lazy_addition<A, B> operator+(A const& a, B const& b){
 return {a, b};
}
```

- Same thing for the trace

```
template<Matrix M>
double trace (M const & m) {
//…
}
```

- Compiler will issue clear error messages in other cases.
- No more long error message of template C++ code, including STL.

Concepts : simple error message

```
struct my_vector {}; // anything, not a matrix

int main() {
  auto v = my_vector{};
  double t = trace(v);
}
```

```
$ g++-6 -fconcepts trace_ab.cpp
trace_ab.cpp: In function 'int main()':
trace_ab.cpp:72:19:
  error: cannot call function 'double trace(const auto:1&) [with auto:1 = my_vector]'
  trace(my_vector{});
                   ^

trace_ab.cpp:9:8: note:   constraints not satisfied
 double trace(Matrix const& m) {
        ^~~~~
trace_ab.cpp:9:8: note:   concept 'Matrix<my_vector>' was not satisfied
```

- Concepts = simpler error messages

An open way to code ...

Classes modeling the Matrix concept

- square_matrix
- lazy_addition
- …

Other examples of classes modeling Matrix concept

- A matrix whose form is known analytically.

```
struct hilbert_matrix {
 int n;

 double operator()(int i, int j) const { return 1.0 / (i + j + 1); }
 friend int dim(hilbert_matrix const& m) { return m.n; }
};
```

- A matrix of rank 1 : $M_{ij} = X_i Y_j$. Store only X and Y

```
class rank1_matrix {
 std::vector<double> x, y; // a place to store the data

 public:
 rank1_matrix(std::vector<double> x, std::vector<double> y) :x(x), y(y) {}

 double operator()(int i, int j) const { return x[i] * y[j]; }
 friend int dim(rank1_matrix const& m) { return m.x.size(); }
};
```

Let us try it

```
int main() {

 auto m1 = square_matrix{4};
 auto m2 = hilbert_matrix{4};
 auto m3 = rank1_matrix{{0,1,2},{1,1,1}};

 std::cout << trace(m1) <<'\n'
           << trace(m1 + m2)  <<'\n'
           << trace(m1 + m2 + m3)  << std::endl;
}
```

- Compile and run ...

```
$ g++-6 -fconcepts trace_ab.cpp
$ ./a.out
4
5.67619
8.67619
```

Mathematical analogy

Analogy with mathematics

- Math
  - Notion of group.
  - General theorem that apply for every group
- Programming
  - Notion of concepts.
  - General algorithms that apply for every type which model the concept.
- Library design :
  - Find the most fruitful concepts for our field (e.g. solid state physics, quantum many-body problem)
  - A hierarchy of concepts, real type as leaf.
    Similar to Julia type system

Continue analogy

- The category of Matrix types is closed under addition.

  Matrix + Matrix → Matrix

- square_matrix is not :
  square_matrix + square_matrix != square_matrix

- Consequence for user's code …

```
square_matrix A, B;

auto X = A + B;   // What is X

Matrix auto X = A + B; // Same with concept checking

auto Y = square_matrix{A+B};
```

Exercise

- Write the abs function so that this code does not involve any temporary

```
auto r = trace (abs(a));
```

Full code available with slides


```
// g++-9 -std=c++17 -fconcepts trace_example.cpp -fsanitize=address -
O3

#include <iostream>
#include <vector>

// With concept TS (slight change for C++20).
template <typename T> concept bool Matrix = requires(T m) {
  { m(0, 0) } ->double; // std::convertible_to<double>;
  { dim(m) }  ->int;    // std::convertible_to<int>;
};

//  trace function
template <Matrix M> double trace(M const &m) {
  double r = 0;
  int d = dim(m); // size of the matrix d x d
  for (int i = 0; i < d; ++i)
    r += m(i, i);
  return r;
}

// baby matrix class
class square_matrix {
  int n;
  std::vector<double> data;

public:
  square_matrix(int n) : n(n), data(n * n, 1) {}

  double operator()(int i, int j) const { return data[i + n * j]; }
  friend int dim(square_matrix const &m) { return m.n; }

  // Access to modify the matrix : not required by Matrix
  double &operator()(int i, int j) { return data[i + n * j]; }
};

/
// lazy addition
template <typename A, typename B> struct lazy_addition {
  A const &a;
  B const &b;
  double operator()(int i, int j) const { return a(i, j) + b(i, j); }
  friend int dim(lazy_addition const &x) { return dim(x.a); }
};
```

```
// implement a + b
template <Matrix A, Matrix B>
lazy_addition<A, B> operator+(A const &a, B const &b) {
  return {a, b};
}

// ----  Other matrix classes  ------

struct hilbert_matrix {
  int n;

  double operator()(int i, int j) const { return 1.0 / (i + j + 1); }
  friend int dim(hilbert_matrix const &m) { return m.n; }
};

// -----------

class rank1_matrix {
  std::vector<double> x, y; // a place to store the data

public:
  rank1_matrix(std::vector<double> x, std::vector<double> y) :
     x(x), y(y) {}

  double operator()(int i, int j) const { return x[i] * y[j]; }
  friend int dim(rank1_matrix const &m) { return m.x.size(); }
};

// ------------------------

// main code

int main() {

  auto m1 = square_matrix{4};
  auto m2 = hilbert_matrix{4};
  auto m3 = rank1_matrix{{0, 1, 2, 3}, {1, 1, 1,1}};

  std::cout << trace(m1) << '\n'
            << trace(m1 + m2) << '\n'
            << trace(m1 + m3) << std::endl;
}
```

Type erasure

Template proliferation

- Another classical issue with generic programming.
- Small objects depends on 1 type (array)
- Composed objects depends on N types, … N is growing
- Illustrate on 2 examples

Reexamine the integrate function

- Integrate on [a, b]

```
template <typename F>
double integrate(F f, double a, double b) {
 const int N = 1000;
 double r = 0, step = (b - a) / N;
 for (int i = 0; i < N; ++i) r += f(a + step * i);
 return r;
}
```

- Apply it $\int_0^1 dx \cos(2x)$

```
double r1 = integrate( [](double x) { return std::cos(2 * x); },  0, 1);
```

Reexamine the integrate function

- Integrate on [a, b]

```
template <typename F>
double integrate(F f, double a, double b) {
 const int N = 1000;
 double r = 0, step = (b - a) / N;
 for (int i = 0; i < N; ++i) r += f(a + step * i);
 return r;
}
```

- For each function f,
  the compiler will generate a new function integrate !
- Do we want that ? Maybe or not.
- Can we do otherwise ?

Version 2

- Same code, but call with std::function. No template

```
template <typename F> double integrate(F f, double a, double b) {
  const int N = 1000;
  double r = 0, step = (b - a) / N;
  for (int i = 0; i < N; ++i)
    r += f(a + step * i);
  return r;
}

double integrate2(std::function<double(double)> f, double a, double b) {
  const int N = 1000;
  double r = 0, step = (b - a) / N;
  for (int i = 0; i < N; ++i)
    r += f(a + step * i);
  return r;
}
```

```
double r1 = integrate( [](double x) { return std::cos(2 * x); }, 0, 1);

double r1 = integrate2( [](double x) { return std::cos(2 * x); }, 0, 1);
```

Version 2

- Same code, but call with std::function. No template

```
template <typename F> double integrate(F f, double a, double b) {
  const int N = 1000;
  double r = 0, step = (b - a) / N;
  for (int i = 0; i < N; ++i)
    r += f(a + step * i);
  return r;
}

double integrate2(std::function<double(double)> f, double a, double b) {
  return integrate(f, a, b);
}
```

std::function

- A STL container that can contain any function of a given signature
- In particular any lambda
- At each call, there is an indirection/ decision to go to the code of the lambda. Cost !
- The compiler can not inline / see through std::function.

```
#include <functional>
int main() {

  std::function<double(double, double)> f;

  f = [](double x, double y) { return x +y;};
}
```

Version 2

```
template <typename F> double integrate(F f, double a, double b) {
  const int N = 1000;
  double r = 0, step = (b - a) / N;
  for (int i = 0; i < N; ++i)
    r += f(a + step * i);
  return r;
}

double integrate2(std::function<double(double)> f, double a, double b) {
  return integrate(f, a, b);
}
```

- Solution 1 :
  - Optimized for each function. But multiple codes.
  - Good if function is very quick
- Solution 2 :
  - One function compiled only (in cpp e.g.)
  - Fine if the function is long to run.

Balance

- Compile time/static polymorphism: template, concepts, ...

vs

- Runtime polymorphism : type erasure (object orientation, …)
- Decision taken at compile time or at run time ?
- Small objects, critical parts : compile time
- Larger objects, less critical : run time.
- How to pass from one to the other ?
  Avoid “template proliferation” in the code.

Type erasure

- Concept :
  - Object is callable with a double, return a double
- Our lambda models the concepts (cos example)
- std::function<double(double)> also
- std::function<double(double)> can store any lambda modelling the concepts.

- General notion : type erasure
  - For each concept, there is one container that
    - Can store any object modelling the concept
    - Models the concept.

Example 2 : generic Monte Carlo

- Directly from the mc_generic in triqs.
- Problem : write a generic Monte Carlo ?

Monte Carlo sampling

- Partition function and operator averaging : (assume p(x) >0)

$$Z = \int_{\mathcal{C}} dx\, p(x), \qquad \langle A \rangle = \frac{1}{Z} \int_{\mathcal{C}} dx\, A(x) p(x)$$

Configuration space

Probability of configuration x
e.g. in classical model : $p(x) \propto e^{-\beta E(x)}$

- Principle : use a Markov chain in configuration space.
  - Average taken over the Markov chain.
  - Transition rate $W_{x \to y}$: probability to go from x to y
  - Detailed balance :

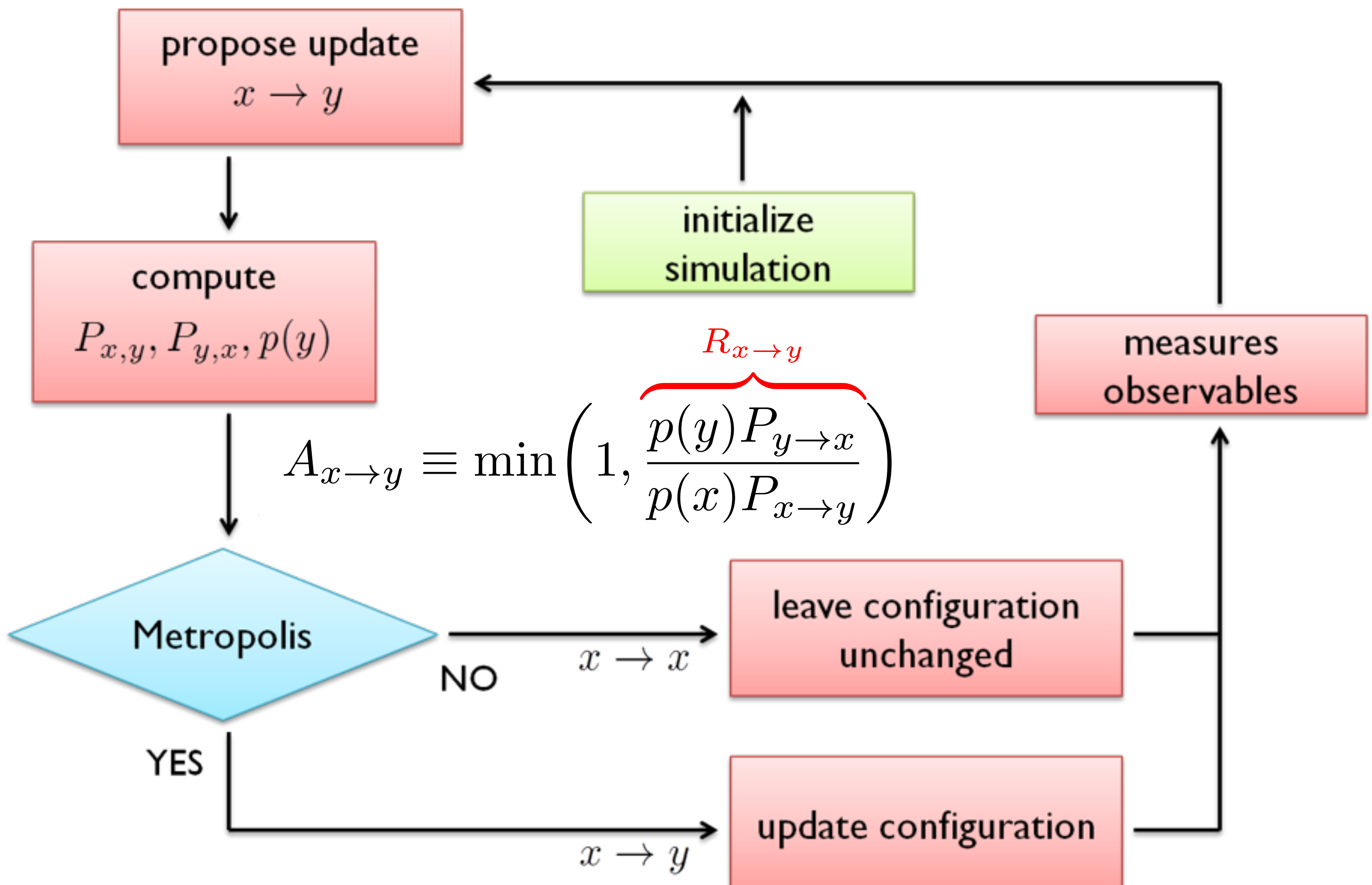
$$\frac{W_{x \to y}}{W_{y \to x}} = \frac{p(y)}{p(x)}$$

  - Ergodicity property :
    It is possible to reach y from x, ∀x,y in a finite number of steps.

Metropolis algorithm

N. Metropolis et al. J. Chem. Phys. 1953

$$W_{x\to y} = \overset{\text{Proposal prob.}}{P_{x\to y}}\,\overset{\text{Acceptance prob.}}{A_{x\to y}}$$



How to write a generic Monte Carlo ?

- A configuration (e.g. spins in Ising model)
- Moves
  one step of the Markov chain, modify configuration.
- Measures.
  Accumulate quantities occasionally along Markov chain

Move concept

- Moves are simple objects with a common concept

```
struct move1 {
 // ... any data, constructor, other method

 double attempt(); // returns Metropolis ratio Rx→y up to sign
 double accept();  // returns 1 or sign of Metropolis ratio
 void reject();    // Cleaning if necessary
};
```

```
struct move2 {
 // ... other details differs from move1, but same
 // interface

 double attempt(); // returns Metropolis ratio Rx→y up to sign
 double accept();  // returns 1 or sign of Metropolis ratio
 void reject();    // Cleaning if necessary
};
```

$$W_{x\to y} = P_{x\to y} A_{x\to y}$$

$$A_{x\to y} \equiv \min\left(1, \underbrace{\frac{p(y)P_{y\to x}}{p(x)P_{x\to y}}}_{R_{x\to y}}\right)$$

Type erasure

- Implement the type erasure for the move concept
- A systematic technique to do this, not presented here.

```
struct any_move {

 template<Move M>
   any_move(M&& m); // construct from any M with the Move concept.

 double attempt(); // returns Metropolis ratio Rx→y up to sign
 double accept();  // returns 1 or sign of Metropolis ratio
 void reject();    // Cleaning if necessary
};
```

- Take a std::vector<any_move> and write the generic Monte Carlo
- Complete separation of the MC logic, from the details of the moves
- Easy to add move, the user just write a simple class with the Move concept. No need to know the exact type in advance

Python : the ultimate type eraser

- Python object can be anything (dynamic type)
- The most general type eraser (see also std::any)
- Cf Python / C++ interface this afternoon

Thank you for your attention

Solution of the exercise

```
// Lazy call of the function object F
template <typename F, Matrix A> struct lazy_call {
 F f;
 A const& a;

 double operator()(int i, int j) const { return f(a(i, j)); }
 friend int dim(lazy_call const& x) { return dim(x.a); }
};

// A make function. Given f,a it builds lazy_call without having to
// specify F
template <typename F, Matrix A> lazy_call<F, A> make_lazy_call(F f, A const& a) {
 return {f,a};
}

// abs for Matrix concept
// f is a lambda that will apply abs to its argument
template <Matrix A> auto abs(A const& a) {
 auto f = [](auto const &x) { using std::abs; return abs(x);};
 return make_lazy_call(f, a);
}
```