Sciware on Modern C++

Generic programming

Introduction to constraints & concepts

Olivier Parcollet

FLATIRON INSTITUTE
Center for Computational Quantum Physics

Generic programming in C++

- A function *sqr* that computes the square of x

```cpp
auto sqr (auto x) {
 return x*x;
}
```

$\Longleftrightarrow$

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

- Given *x* of type *T*, the compiler will instantiate (i.e. generate & compile) a *sqr* function for the type T

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

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

- Compare to Python : looks similar, but auto is resolved *at compile-time*

```python
def sqr(x):
 return x*x
```

A less trivial example : $\sqrt{x}$ with Babylonian algorithm

- Iterative method $y_{n+1} = \dfrac{y_n + x/y_n}{2}$, starting from $y_0 = \dfrac{1+x}{2}$. Fixed point : $y = x/y$

```cpp
auto babylonian(auto x, int N = 10) {
    auto y = (1+x)/2;
    for (int i = 0; i < N; ++i) y = (y + x/y)/2;
    return y;
}
```

```cpp
auto sqr (auto x) {
 return x*x;
}
```

- Code reuse !
  - double, float
  - High precision floating arithmetic (e.g. GMP, ARB library)
  - …
- What is the condition on x for the algorithm to compile & work ?

When an error occurs …

```cpp
auto babylonian(auto x, int N = 10);
```

```cpp
struct bad{};

int main() {
    auto b = bad{};
    auto r = babylonian(b);
}
```

```
Mac:~/ clang++ -std=c++20 babylon_error.cpp

babylon_error.cpp:3:15: error: invalid operands to binary expression ('int' and 'bad')
   auto t = (1+x)/2;
             ~^~
babylon_error.cpp:13:14: note: in instantiation of function template specialization 'babylonian<bad>'
requested here
    auto r = babylonian(b);
             ^
1 error generated.
```

When an error occurs …

- Error messages can be horribly long, e.g. with Standard Library

```
std::vector<A> v = …;
std::sort(begin(v), end(v));
```

- Forgot to implement the comparison *operator*< for A.

!?

```
In file included from /usr/local/opt/llvm/bin/../include/c++/v1/vector:275:
In file included from /usr/local/opt/llvm/bin/../include/c++/v1/__algorithm/equal.h:13:
/usr/local/opt/llvm/bin/../include/c++/v1/__algorithm/comp.h:73:71: 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/opt/llvm/bin/../include/c++/v1/__algorithm/sift_down.h:44:34: note: in
instantiation of member function 'std::__less<A>::operator()' requested here
    if ((__child + 1) < __len && __comp(*__child_i, *(__child_i + difference_type(1)))) {
                                 ^
/usr/local/opt/llvm/bin/../include/c++/v1/__algorithm/make_heap.h:37:14: note: in
instantiation of function template specialization 'std::__sift_down<std::_ClassicAlgPolicy,
std::__less<A> &, std::__wrap_iter<A *>>' requested here
        std::__sift_down<_AlgPolicy>(__first, __comp_ref, __n, __first + __start);
             ^
/usr/local/opt/llvm/bin/../include/c++/v1/__algorithm/partial_sort.h:39:8: note: in
instantiation of function template specialization 'std::__make_heap<std::_ClassicAlgPolicy,
std::__less<A> &, std::__wrap_iter<A *>>' requested here
  std::__make_heap<_AlgPolicy>(__first, __middle, __comp);
       ^
/usr/local/opt/llvm/bin/../include/c++/v1/__algorithm/partial_sort.h:67:27: note: in
instantiation of function template specialization
'std::__partial_sort_impl<std::_ClassicAlgPolicy, std::__less<A> &, std::__wrap_iter<A *>,
std::__wrap_iter<A *>>' requested here
  auto __last_iter = std::__partial_sort_impl<_AlgPolicy>(__first, __middle, __last,
static_cast<_Comp_ref>(__comp));
                          ^
/usr/local/opt/llvm/bin/../include/c++/v1/__algorithm/sort.h:681:10: note: in instantiation of
function template specialization 'std::__partial_sort<std::_ClassicAlgPolicy, std::__less<A>,
std::__wrap_iter<A *>, std::__wrap_iter<A *>>' requested here
    std::__partial_sort<_AlgPolicy>(__first, __last, __last, __comp);
         ^
/usr/local/opt/llvm/bin/../include/c++/v1/__algorithm/sort.h:694:8: note: in instantiation of
function template specialization 'std::__sort_impl<std::_ClassicAlgPolicy, std::__wrap_iter<A
*>, std::__less<A>>' requested here
  std::__sort_impl<_ClassicAlgPolicy>(std::move(__first), std::move(__last), __comp);
       ^
/usr/local/opt/llvm/bin/../include/c++/v1/__algorithm/sort.h:700:8: note: in instantiation of
function template specialization 'std::sort<std::__wrap_iter<A *>, std::__less<A>>' requested
here
  std::sort(__first, __last, __less<typename
iterator_traits<_RandomAccessIterator>::value_type>());
```

Issues with generic programming


```
auto babylonian(auto x, int N = 10);
```

- Issue :
  - Bad error message as we see the implementation details.
  - User does not know what x can be !
  - By the way, Python has the same problem (no types).
- Solution
  - Constraints on the type of x
  - Express them in the signature of the function (so that the user can know)
  - Check the constraints *before* generating the code (to have a meaningful error message).
- C++20 concepts.

Concept : a set of constraints on a type

- Two new C++ keywords : concept, requires

  - A function $T \to$ bool
    executed at compile time:
    does T satisfy these constraints ?

```
#include <concepts>
template<typename T>
concept Number = requires(T x, T y) {
 {x+y} -> std::same_as<T>;
 {x-y} -> std::same_as<T>;
 {x*y} -> std::same_as<T>;
 {x/y} -> std::same_as<T>;
 {T{0}}; // T can construct from an int
};
```

- Constrain the algorithm

```
auto babylonian(Number auto x, int N = 10) {
 // Same as before
}
```

Terse syntax

```
template<Number T>
T babylonian(T x, int N = 10) {
 // Same as before
}
```

General syntax

```
template<typename T>
requires(Number<T>)
T babylonian(T x, int N = 10) {
// Same as before
}
```

Concepts : meaningful error message

```
auto babylonian(auto x, int N = 10);
```

```
struct bad{};

int main() {
    auto b = bad{};
    auto r = babylonian(b);
}
```

- Does not enter the logic of the function, and subsequent calls

```
babylon_error.cpp:22:14: error: no matching function for call to 'babylonian'
    auto r = babylonian(b);
             ^~~~~~~~~~
babylon_error.cpp:12:6: note: candidate template ignored: constraints not satisfied [with x:auto = bad]
auto babylonian(Number auto x, int N = 10) {
     ^
babylon_error.cpp:12:17: note: because 'bad' does not satisfy 'Number'
auto babylonian(Number auto x, int N = 10) {
                ^
babylon_error.cpp:5:5: note: because 'x + y' would be invalid: invalid operands to binary expression ('bad' and 'bad')
 {x+y} -> std::same_as<T>;
    ^
1 error generated.
```

Concept, algorithms, types

Concept

A set of a constraints on a type
e.g. Number

Types

A "category" of types satisfying the concept

e.g. double, float, gmp

Algorithms

A set of generic algorithms working with any type satisfying the concept

e.g. sqr, babylonian, …

Main idea of generic programming

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

Concept, algorithms, types

Concept

A set of a constraints on a type
e.g. Number

Types

A "category" of types satisfying the concept

e.g. double, float, gmp

Algorithms

A set of generic algorithms working with any type satisfying the concept

e.g. sqr, babylonian, …

- Separation data & algorithm
- Composability : # *types* × # *algorithms* possibilities

Concept, algorithms, types

Concept

A set of a constraints on a type

e.g. Number

Types

A "category" of types satisfying the concept

e.g. double, float, gmp

Algorithms

A set of generic algorithms working with any type satisfying the concept

e.g. sqr, babylonian, …

Part of function signature, i.e. library documentation

```
auto babylonian(Number auto x, int N = 10);
```

Case study

A tiny matrix class with a few functions

Goal : zero overhead abstraction

- Clear & simple code, without performance penalty.
- Temporaries elimination, e.g. for $a, b \in M_n(\mathbb{R})$

```
double r = trace (a + b);
```

$\Longleftrightarrow$

```
double r = 0;
for (int i = 0; i < n; ++i)
  r += a(i, i) + b(i, i);
```

How ? a + b computed before calling trace, so $O(n^2)$ instead of $O(n)$?

- Techniques exists : "expression template", "metaprogramming".
  Used in all modern libraries : Eigen, triqs/nda (from CCQ), [precursor : Blitz++]
- C++20 concepts make them natural and easy to implement.

The Matrix concept

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

The type T behaves like a square matrix (of double)

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

Two algorithms

- Use only what is in the concept.

$$Tr\,M = \sum_{i=1}^{n} M_{ii}$$

$$sum(M) = \sum_{i,j=1}^{n} M_{ij}$$

```
double trace (Matrix auto 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;
}
```

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

- Also insulate sum from other overloads for unrelated types…

A simple matrix class

- Now I implement a simple class…

```
class square_matrix {
  int n;                       // size
  std::vector<double> data; // some storage

public:
  square_matrix(int n) : n(n), data(n * n, 0) {}

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

  // other methods ...
};
```

- Concepts are *non intrusive*

- Satisfy Matrix concept

```
static_assert(Matrix<square_matrix>);
```

- Hence algorithms work …

```
int main() {
  auto m = square_matrix{4};
  // ...
  auto t = trace(m)
}
```

Other Matrix classes …

```
struct rank1_matrix {
  std::vector<double> x, y;
  double operator()(int i, int j) const { return x[i] * y[j];}
  // ...
};
int dim(rank1_matrix const &m) { return m.x.size();}
```

$$M_{ij} = x_i y_j$$

$$M_{ij} = x\delta_{ij}$$

```
struct kronecker_matrix {
  int n;
  double x;
  //...
  double operator()(int i, int j) const { return (I==j ? x : 0);}
};
int dim(kronecker_matrix const &m) { return m.n; }
```

- Satisfy Matrix, hence previous algorithms (trace, sum) works

Concept, algorithms, types

Concept

Matrix

Types

square_matrix, rank1_matrix, kronecker_matrix, …

Algorithms

- *Functions Matrix* $\to \mathbb{R}$
  e.g. trace, sum, …

Concept, algorithms, types

Concept

Matrix

Types

square_matrix, rank1_matrix, kronecker_matrix, …

Algorithms

- *Functions Matrix* $\rightarrow \mathbb{R}$
  e.g. *trace, sum, …*
- *Functions Matrix* $\rightarrow$ *Matrix*
  e.g. $a \rightarrow \text{abs}(a)$
- *Functions Matrix x Matrix* $\rightarrow$ *Matrix*
  e.g. $(a, b) \rightarrow a + b$

Abs, addition


```
Matrix auto abs(Matrix auto const &m);
```

$$[\text{abs}(M)]_{ij} = |M_{ij}|$$

```
Matrix auto operator+(Matrix auto const &a, Matrix auto const &b);
Matrix auto operator-(Matrix auto const &a, Matrix auto const &b);
```

- Composability

```
int main() {
  auto a = square_matrix{2};
  auto b = rank1_matrix{{0, 3}, {1, 1}};

  double t = trace(a + b);
  double diff = sum(abs(a - b));

  Matrix auto s = a + b;
  Matrix auto s2 = square_matrix{a + b};
}
```

$$\|A - B\| = \sum_{i,j=1}^{n} |A_{ij} - B_{ij}|$$

$$\text{Tr}(A + B) = \sum_{i=1}^{n} A_{ii} + B_{ii}$$

Abs, addition

```
Matrix auto abs(Matrix auto const &m);
```

$$[\mathrm{abs}(M)]_{ij} = |M_{ij}|$$

```
Matrix auto operator+(Matrix auto const &a, Matrix auto const &b);
Matrix auto operator-(Matrix auto const &a, Matrix auto const &b);
```

- Composability

```
int main() {
  auto a = square_matrix{2};
  auto b = rank1_matrix{{0, 3}, {1, 1}};

  double t = trace(a + b);
  double diff = sum(abs(a - b));

  Matrix auto s = a + b;
  Matrix auto m3 = square_matrix{a + b};
}
```

```
class square_matrix {
//...
public:
  square_matrix(int n);
  square_matrix(Matrix auto const &m);
//...
};
```

Implementation abs

```
Matrix auto abs(Matrix auto const &m) {
  return lazy_mapped_matrix{m, [](auto &&x) { return std::abs(x); }};
}
```

- Mapping a function onto a Matrix (term by term) with $f(x) \equiv |x|$

$$M'_{ij} = f(M_{ij})$$

```
template <Matrix M, typename F> struct lazy_mapped_matrix { // implementation detail
  M const &m;
  F f;
  double operator()(int i, int j) const { return f(m(i, j)); }
  friend int dim(lazy_mapped_matrix const &m) { return dim(m.m); }
};
template<Matrix M, typename F> lazy_mapped_matrix(M, F)-> lazy_mapped_matrix <M, F>; // CTAD
```

Why lazy ?


```cpp
double sum(Matrix auto const &m) {
  double r = 0;
  int d = dim(m);
  for (int i = 0; i < d; ++i)
    for (int j = 0; j < d; ++j)
      r += m(i, j);
  return r;
}
```

```cpp
template <Matrix M, typename F> struct lazy_mapped_matrix {
  M const &m;
  F f;
  double operator()(int i, int j) const { return f(m(i, j)); }
  friend int dim(lazy_mapped_matrix const &m) { return dim(m.m); }
};
```

- The compiler take the generic trace and generates the code

```cpp
double sum(lazy_mapped_matrix<M,F> auto const &m) {
  double r = 0;
  int d = dim(m.m);
  for (int i = 0; i < d; ++i)
    for (int j = 0; j < d; ++j)
      r += m.f(m.a(i,i));
  return r;
}
```

$$||A|| = \sum_{i,j=1}^{n} |A_{ij}|$$

- Exactly the hand written code. **No temporaries**

Addition implementation

```
Matrix auto operator+(Matrix auto const &a, Matrix auto const &b) {
  return lazy_add{a, b};
}
```

- Lazy addition: just take references and wait to be called. Implementation detail, not for user

```
template <Matrix A, Matrix B> struct lazy_add {
  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); }
};
template <Matrix A, Matrix B> lazy_add(A, B) -> lazy_add<A, B>;
```

- Exercise : show that Tr(a+b) scales like *dim*, not *dim*^{2}

$$\mathrm{Tr}(A+B) = \sum_{i=1}^{n} A_{ii} + B_{ii}$$

Concept refinement

- *Issue* : for a rank 1 matrix, the sum algorithm is ridiculous …

$$sum(M) = \sum_{i,j=1}^{n} M_{ij} = \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{j=1}^{n} y_j\right) \quad \text{if} \quad M_{ij} = x_i y_j$$

- *Solution:*
  A more refined concept Matrix_of_rank1
  Compiler will choose the most refined one.

```
double sum(UnrelatedThing auto const &m) {
  // ...
}
```

```
double sum(Matrix auto const &m) {
  // ...
}
```

```
template <typename T>
concept Matrix_of_rank1 = Matrix<T> && requires(T const &m) {
  { m.get_x() } ;
  { m.get_y() };
};
```

```
double sum(Matrix_of_rank1 auto const &m) {
  // ...
}
```

Conclusion

- Generic programming, concept, constraint

- What I did not cover
  - More on concepts: refine them, combine them, concepts depending on 2 types, …
  - Feature to ease writing of generic code, take compile time decisions (e.g. *if constexpr, requires "on the fly", folds, variadic template, …)*
  - Type erasure : from compile time to run time polymorphism.

FLATIRON INSTITUTE

Lazy again [exercise]


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

```
template <Matrix A, Matrix B> struct lazy_add {
  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 &m) { return dim(m.a); }
};
```

- The compiler take the generic trace and generates the code

```
double trace(lazy_add<A,B> const & m) {
 double 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;
}
```

- Exactly the hand written code. **No temporaries**
- Scales like *dim*, not *dim*^{2}

$$\mathrm{Tr}(A+B) = \sum_{i=1}^{n} A_{ii} + B_{ii}$$