Programming Techniques for Scientific Simulations I

Templates, type traits, generic programming

Metaprogramming: Preview

- “Template metaprogramming is a family of techniques to create new types and compute values at compile time”
- Standard function: Zero+ parameters and a return value (or void)

```
int int_func(double x, int N) {
  return x;
}
```

- Meta function: A struct/class with zero+ template parameters and zero(+) return types or values

```
template <int X>
struct ReturnValue {
  static constexpr int value = X;
};
```

Return value in public member “value”

```
template <typename T>
struct ReturnType {
  using type = T;
};
```

Return value in public member “type”

Type traits: motivation

- Recall the generic minimum example from week 3

```
template <typename T>
T const& min(T const& x, T const& y) {
  return x < y ? x : y;
}
```

- We want to allow

```
min(1., 2)
min(1, 2.)
// etc
```

- Manual solution from week 3

```
template <typename R, typename T, typename U>
R min(T const& x, U const& y) {
  return x < y ? x : y;
}
```

- Which allows

```
min<int>(1., 2)
min<double>(1, 2.)
// etc
```

Type traits: motivation

- We want to allow the addition of two arrays:

```
template <typename T>
SArray<T> operator+(SArray<T> const&, SArray<T> const&)
```

- How do we add two "different" arrays? E.g. `int` plus `double`.

```
template <typename T, typename U>
SArray<?> operator+(SArray<T> const&, SArray<U> const&)
```

- We could again

```
template <typename R, typename T, typename U>
SArray<R> operator+(SArray<T> const&, SArray<U> const&)
```

- What is the result type?
  - We want to "calculate" with types!
- The solution is a technique called **traits**

Type traits: motivation

- We want to do something like

```
template <typename T, typename U>
typename min_type<T,U>::type min(T const& x, U const& y) {
  return x < y ? x : y;
}
```

- And

```
template <class T, class U>
SArray<typename sum_type<T,U>::type>
operator +(const SArray<T>&, const SArray<U>&)
```

- The keyword `typename` is needed here so that C++ knows the member is a type and not a variable or function
  - This is required to parse the program code correctly - it would not be able to check the syntax otherwise… Needed with template dependent types
- How to compute types `min_type` & `sum_type`?

Type traits: minimum example

- A definition of `min_type`
  - Empty template type to trigger error messages if used

```
template <typename T, typename U> struct min_type { };
```

  - Partially specialized valid templates

```
template <typename T> struct min_type<T, T> { typedef T type; };
```

  - Fully specialized valid templates

```
template <> struct min_type<float, double> { typedef double type; };
template <> struct min_type<double, float> { typedef double type; };
template <> struct min_type<float, int> { typedef float type; };
template <> struct min_type<int, float> { typedef float type; };
// ...
```

- What is?

```
min(1, 2)
min(1, 2.3f)
```

→

```
min_type<?, ?>::type = ?
```

Type traits: minimum example

- A definition of `min_type`
  - Empty template type to trigger error messages if used

```
template <typename T, typename U> struct min_type { };
```

  - Partially specialized valid templates

```
template <typename T> struct min_type<T, T> { typedef T type; };
```

  - Fully specialized valid templates

```
template <> struct min_type<float, double> { typedef double type; };
template <> struct min_type<double, float> { typedef double type; };
template <> struct min_type<float, int> { typedef float type; };
template <> struct min_type<int, float> { typedef float type; };
// ...
```

- What is?

```
min(1, 2)      →  min_type<int, int>::type = int
min(1, 2.3f)   →  min_type<int, float>::type = float
```

Type traits: simple array example

- A definition of `sum_type`
  - Empty template type to trigger error messages if used

```
template <typename T, typename U> struct min_type { };
```

  - Partially specialized valid templates

```
template <class T> struct sum_type<T, T> { typedef T type; };
```

  - Fully specialized valid templates

```
template <> struct sum_type<double, float> { typedef double type; };
template <> struct sum_type<float, double> { typedef double type; };
template <> struct sum_type<float, int> { typedef float type; };
template <> struct sum_type<int, float> { typedef float type; };
// ...
```

Old style traits

- In C++98 traits were big "blobs":

```
template<>
struct numeric_limits<int> {
  static const bool is_specialized = true;
  static const bool is_integer = true;
  static const bool is_signed = true;
  ...
};
```

- Later it was realized that this was ugly:
  - A traits class is a "meta function", a function operating on types
  - A blob like numeric limits takes one argument, and returns many different values
  - This is not the usual design for functions!

New style traits

- Since C++03 all new traits are single-valued "functions"
  - Types are returned as the type member

```
template <typename T> struct min_type<T, T> { typedef T type; };
template <> struct min_type<float, double> { typedef double type; };
```

  - Constant values are returned as the value member

```
template<class T> struct is_integral { static const bool value=false; };
template<> struct is_integral<int> { static const bool value=true; };
```

Type traits: An average example

- Imagine an average function

```
template <typename T>
T average(std::vector<T> const& v) {
  T sum = 0;
  for (std::size_t i=0; i<v.size(); ++i) {
    sum += v[i];
  }
  return sum/v.size();
}
```

- Problems?
  E.g., what is?

```
std::vector<double> vd = {1., 2., 3., 4.};
average(vd);
std::vector<int>    vi = {1 , 2 , 3 , 4 };
average(vi);
```

Type traits: An average example

- Imagine an average function

```
template <typename T>
T average(std::vector<T> const& v) {
  T sum = 0;
  for (std::size_t i=0; i<v.size(); ++i) {
    sum += v[i];
  }
  return sum/v.size();
}
```

- Problems?
  E.g., what is?

```
std::vector<double> vd = {1., 2., 3., 4.};
average(vd);  →   10./4. = 2.5
std::vector<int>    vi = {1 , 2 , 3 , 4 };
average(vi);  →   int(10/4) = 2  ←   Want 2.5!
```

Type traits: An average example

- Imagine an average function

```
template <typename T>
typename average_type<T>::type average(std::vector<T> const& v) {
  typename average_type<T>::type sum = 0;
  std::cout << __PRETTY_FUNCTION__ << '\n';
  for (std::size_t i=0; i<v.size(); ++i) {
    sum += v[i];
  }
  return sum/v.size();
}
```

Type traits: An average example (manual solution)

- A definition of `average_type`
  - The general case

```
template <class T> struct average_type<T> { typedef T type; };
```

  - The special cases

```
template <> struct average_type<int> { typedef double type; };
// ... repeat for ALL integer types ...
```

There are quite a few... See here.

Type traits: An average example (automatic solution)

- A definition of `average_type`
  - The general case

```
template <typename T>
struct average_type {
  typedef typename helper<T, std::numeric_limits<T>::is_integer>::type type;
};
```

  - The "helper"
    - The general case

```
template <typename T, bool F>
struct helper { typedef T type; };
```

    - The special case for integers

```
template <typename T>
struct helper<T, true> { typedef double type; };
```

A type trait that is true for all integer arithmetic types: see here.

Generic Programming

- Templates provide direct support for generic programming in the form of programming using types as parameter
  - Function templates
  - Class templates
- A template is just a “blueprint”, only when used with specific template arguments it is generated: this is called instantiation
- A template puts requirements for its arguments (Stating them in code possible in C++20!)
- Templates are type-safe: no object can be misused -> compile error

Concepts / Named Requirements

- A concept is a set of requirements on types:
  - The **operations** the types must provide
  - Their **semantics** (i.e. the meaning of these operations)
  - Their time/space complexity
- A type that satisfies the requirements is said to **model** the concept
- A concept can extend the requirements of another concept, which is called **refinement**
- The standard defines few fundamental concepts, e.g.
  - CopyConstructible
  - Assignable
  - EqualityComparable
  - Destructible

  Regular type
- See e.g. https://en.cppreference.com/w/cpp/named_req

Documenting a function template

- In addition to
  - Preconditions
  - Postconditions
  - Semantics
  - Exception guarantees
- The documentation of a template function must include
  - Concept requirements on the types
- Note that the complete source code of the template function must be in a header file

Documenting your functions

- **Synopsis** of all functions, types and variables declared
- **Semantics**
  - What does the function do?
- **Requirements**
  - Concepts of template arguments
- **Preconditions**
  - What must be true before calling the function
- **Postconditions**
  - What you guarantee to be true after calling the function if the preconditions were true
- **Dependencies**
  - What does it depend on?
- **Exception guarantees** (will be discussed later)
- **References** or other additional material

Example documentation (Problem 4.2)

- **Synopsis:**

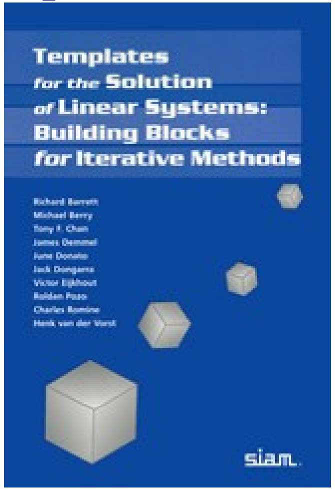
```
template<typename F, typename T>
T simpson(const T a, const T b, const unsigned bins, const F& func)
```

- **Semantics:**
  - `simpson` computes an approximation of the function `func(x)` over the interval `[min(a,b),max(a,b)]` using the composite Simpson rule with '`bins`' equally sized subintervals
- **Requirements:**
  - Concepts needed for type `F`: `F` needs to be a function or function object taking a single argument convertible from `T`, with return value convertible to `T`.
  - Concepts needed for type `T`: CopyConstructible, Assignable, `T` shall support arithmetic operations with double with result convertible to `T` with limited relative truncation errors.
- **Preconditions:**
  - The domain of the function `func(x)` has cover the interval `[min(a,b),max(a,b)]`
  - `'bins' > 0` convertible to unsigned
- **Postconditions**
  - The return value will approximate the integral of the function `func(x)` over the given interval
- **Dependencies:** None.
- **Exception guarantees:** no-throw
- **References:** ...

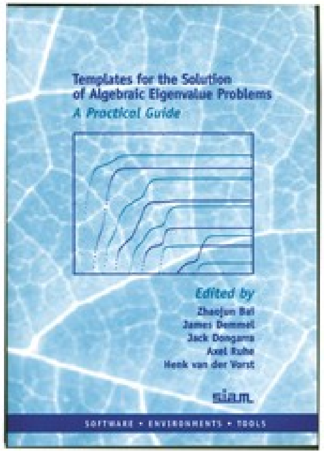
Examples: iterative algorithms for linear systems

- Barret et al., "Templates for the Solution of Linear Systems", 1994
https://doi.org/10.1137/1.9781611971538

https://www.netlib.org/linalg/html_templates/Templates.html



- Bai et al., "Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide", 2000
https://doi.org/10.1137/1.9780898719581



The power method

- Is the simplest eigenvalue solver
  - Returns the largest absolute eigenvalue and corresponding eigenvector

ALGORITHM 4.1: **Power Method for HEP**

(1) *start with vector* $y = z$, *the initial guess*
(2) **for** $k = 1, 2, \ldots$
(3) $\quad v = y / \|y\|_2$
(4) $\quad y = A\, v$
(5) $\quad \theta = v^* \, y$
(6) $\quad$ **if** $\|y - \theta\, v\|_2 \leq \epsilon_M |\theta|$, **stop**
(7) **end for**
(8) *accept* $\lambda = \theta$ *and* $x = v$

- Only requirements:
  - A is linear operator on a Hilbert space
  - Initial vector y is vector in the same Hilbert space
- Can we write the code with as few requirements as possible?

Generic implementation of the power method

- A possible generic implementation

```
OP A;                            // linear operator
V v, y;                          // vectors
T theta, tolerance, residual; // scalars
// ...
do {
  v = y / two_norm(y);               // line (3)
  y = A * v;                         // line (4)
  theta = dot(v, y);                 // line (5)
  residual = two_norm(y - theta * v); // line (6)
} while( residual > tolerance*abs(theta) );
```

ALGORITHM 4.1: **Power Method for HEP**

(1) *start with vector* $y = z$, *the initial guess*
(2) **for** $k = 1, 2, \ldots$
(3) $\quad v = y/\|y\|_2$
(4) $\quad y = A\,v$
(5) $\quad \theta = v^* y$
(6) $\quad$ **if** $\|y - \theta\, v\|_2 \leq \epsilon_M |\theta|$, **stop**
(7) **end for**
(8) *accept* $\lambda = \theta$ *and* $x = v$

Generic implementation of the power method

- A possible generic implementation

```
OP A;                              // linear operator
V v, y;                            // vectors
T theta, tolerance, residual;      // scalars
// ...
do {
  v = y / two_norm(y);                      // line (3)
  y = A * v;                                // line (4)
  theta = dot(v, y);                        // line (5)
  residual = two_norm(y - theta * v);       // line (6)
} while( residual > tolerance*abs(theta) );
```

ALGORITHM 4.1: **Power Method for HEP**

(1) *start with vector* $y = z$, *the initial guess*
(2) **for** $k = 1, 2, \ldots$
(3) $\quad v = y / \|y\|_2$
(4) $\quad y = A\,v$
(5) $\quad \theta = v^* y$
(6) $\quad$ **if** $\|y - \theta\, v\|_2 \le \epsilon_M |\theta|$, **stop**
(7) **end for**
(8) *accept* $\lambda = \theta$ *and* $x = v$

Concepts for the power method

- The triple of types (`T`, `V`, `OP`) models the Hilbert space concept if
  - `T` must be the type of an element of a field
  - `V` must be the type of a vector in a Hilbert space over that field
  - `OP` must be the type of a linear operator in that Hilbert space
- All the allowed mathematical operations in a Hilbert space have to exist:
  - Let `v`, `w` be of type `V`
  - Let `r`, `s` of type `T`
  - Let `A` be of type `OP`
  - The following must compile and have the same semantics as in the mathematical concept of a Hilbert space:
    - `r+s`, `r-s`, `r/s`, `r*s`, `-r` have return type `T`
    - `v+w`, `v-w`, `v*r`, `r*v`, `v/r` have return type `V`
    - `A*v` has return type `V`
    - `two_norm(v)` and `dot(v, w)` have return type `T`
    - …