Template Meta Programming

Programming Techniques for Scientific Simulation II

Plus Trait

```
template<typename  T>
struct has_plus_trait  {
    static constexpr bool value = false;
};
// we need to specialize our trait for
// every type that suppots addition
template<>
struct has_plus_trait<int>  {
    static constexpr bool value = true;
};

// together with enable_if, we can switch
// implementation depending on the trait

template<typename  T
       , typename  S = std::enable_if_t<has_plus_trait<T>::value>>
void fct(T const & t); // plus version

template<typename  T
       , typename  S = std::enable_if_t<!has_plus_trait<T>::value>>
void fct(T const & t); // no plus version
```

Plus Auto Detect

```
#include <type_traits>  // true / false_type

template<typename  T>
struct has_plus_trait  {

    // we use SFINAE in the default  argument  for S
    // declval<U>()  is just  a save way  to write U()
    // since U may  not  have  a default-ctor
    template<typename  U
           , typename  S = decltype(std::declval<U>()  + std::declval<U>())>
    static std::true_type  check(int);

    template<typename  U> //just catch anything else (variadic function)
    static std::false_type  check(...);

    // we want to check if check<T>  managed to substitute the first
    // check template.  If not, the return value is std::false_type  and
    // the std::false_type::value  is false
    static constexpr  bool value = decltype(check<T>(0))::value;
};
```

Variadic Functions

```
#include <stdarg.h> // used for variadic functions

// type-unsafe mean
double mean(int const & N, ...) { // ... here means variadic function
    va_list ap;       // the argument pack
    va_start(ap, N); // initializes argument pack

    double sum = 0;

    for(int i = 0; i < N; ++i)
        // hope and pray it's a double (also "increments" ap)
        sum += va_arg(ap, double);

    va_end(ap); // clean-up

    return sum / N;
}

// well known variadic function from c
printf("%s %s! %f\n", "Hello", "World", 1.1);
```

Variadic Templates

```
// stop the recursion
double accumulate() { return 0; }

// accumulate
// * Args and args is convention, could be any name
// * typename... specifies a variadic template type (must be at the end)
// * we specify the pattern "Args const &" and the use ...
//   to use this pattern for all types in the argument-pack
template<typename T, typename... Args>
double accumulate(T const & val, Args const &... args) {
    // again, pattern "args" and the expand it for all with ...
    return val + accumulate(args...);
}

// type-safe mean
template<typename... Args>
double mean(Args const &... args) {
    // sizeof...(args) or sizeof...(Args) returns how many argument are
    // found in the argument-pack
    return accumulate(args...) / sizeof...(args);
}
```

Ellipsis Operator . . .

```
...                // in a function signature: variadic c-function
typename...        // specifies a variadic template type (argument pack)
sizeof...          // returns the number of arguments in an argument pack

template<typename... Args>
void fct(Args &&... args) { // note univeral reference

    // ... expands a pattern (std::remove_reference<Args>)
    // for all types in Args
    using T = std::tuple<std::remove_reference<Args>...>;

    // ... expands a pattern (std::forward<Args>(args))
    // for all instances in args
    other_variadic_fct(std::forward<Args>(args)...);

    // illegal, can only expand in a "variadic context"
    args...;
}
```

Meta Programming

	runtime	constexpr	meta template
variables in a function	non const or const	non const or const	only const
variables in a namespace	non const or const	constexpr	only const
passing parameter to function as…	copy, ref, ptr	copy, ref, ptr, but only constexpr parameter	copy
side-effects of a function (aside from return-value)	via arguments (&, *) via global/local variables	none	none
return from function	fct(…) { return x; }		typename fct<…>::type
if construction	if … else …		template specialization
loops	for, while, do while		recursion
paradigm	multiple (generic, procedural, object-oriented …)		functional

std::tuple

```
template<typename... TN> // variadic declaration
struct tuple;

template<> // empty specialization
struct tuple<> {};

// derive recursively is no performance problem if inlining is not prevented
// manually or by virtual functions. There's no single virtual function
here!
template<typename T, typename... TN>// recursive specialization
struct tuple<T, TN...>: public tuple<TN...> {
    using super = tuple<TN...>;
    using value_type = T;

    // uref ctor
    template<typename U, typename... UN>
    tuple(U && u, UN &&... un): super(std::forward<UN>(un)...)
                              , elem(std::forward<U>(u)) {}

    value_type elem; // every element on each "level" has the same name
};
```

std::get

```
template<std::size_t I, typename TUP>
struct element_nr { // jump up in the super chain until ...
    using type = typename element_nr<I-1, typename TUP::super>::type;
};
// ... the index is 0, then just return the current tuple-class
template<typename TUP>
struct element_nr<0, TUP> {
    using type = TUP;
};
// for less typing
template<std::size_t I, typename TUP>
using element_nr_t = typename element_nr<I, TUP>::type;

// get<i>(tup) returns a reference to the i'th element of the tuple
template<std::size_t I, typename TUP>
typename element_nr_t<I, TUP>::value_type & get(TUP & tup) {
    using base = element_nr_t<I, TUP>;
    // the base:: tells tup, which of the many elem we need
    // tup.elem would just be first element
    return tup.base::elem;
}
```

accumulator with tags

```
namespace tag {
    struct count;
    struct mean;
    struct min;
}
// is always last in inheritance chain
template<typename T>
class accum_base;

template<typename T, typename tag, typename B>
class module; // this module doesn't exist unless...

// ... it is specialized, and hold a certain implementation
template<typename T, typename B>
class module<T, tag::count, B>: public B { /* count-impl */ };

template<typename T, typename B>
class module<T, tag::mean, B>: public B { /* mean-impl */ };

template<typename T, typename B>
class module<T, tag::min, B>: public B { /* min-impl */ };
```

accumulator with tags

```
// variadic declaration
template<typename T, typename... Tags>
struct accum_impl;
// empty specialization
template<typename T>
struct accum_impl<T> { using type = accum_base<T>; };
// recursive specialization
template<typename T, typename Tag, typename... Rest>
struct accum_impl<T, Tag, Rest...> {
    using type = module< T, Tag, typename accum_impl<T, Rest...>::type >;
};

// less typing
template<typename T, typename... Tags>
using accum = typename accum_impl<T, Tags...>::type;

// use it
// each accum only compiles the wanted features
accum<int, tag::min> a;
accum<int, tag::mean, tag::count> b;
accum<int, tag::count> c;
```

meta list

```
// a simple forward linked meta-list
template<typename T, typename N>
struct node {
    using type = T;
    using next = N;
};
// marks the end of the list
struct endnode;

// append one list to another one
template<typename L, typename L2>
struct concat_list_impl {
    using type = node< typename L::type
                     , typename concat_list_impl< typename L::next, L2>::type>;
};
// if the end of the first list is reached
template<typename L2>
struct concat_list_impl<endnode, L2> {
    using type = L2;
};

// for less typing
template<typename L1, typename L2>
using concat_list = typename concat_list_impl<L1, L2>::type;
```