Object-Oriented Programming for Scientific Computing

Template Metaprogramming

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

18. Juli 2017

Calculating the Square Root

We can calculate the square root of N as follows, using nested intervals:

```
#include<iostream>

template<std::size_t N, std::size_t L = 1, std::size_t H = N>
struct Sqrt
{
  enum{ mid = (L+H+1)/2 };
  enum{ value = (N<mid*mid)
                ? (std::size_t)Sqrt<N,L  ,mid-1>::value
                : (std::size_t)Sqrt<N,mid,H    >::value };
};

template<std::size_t N, std::size_t M>
struct Sqrt<N,M,M>
{
  enum{ value = M };
};

int main()
{
  std::cout << Sqrt<9>::value << "␣"
            << Sqrt<42>::value << std::endl;
}
```

Template Instantiations

- Calculating `Sqrt<9>` first leads to the execution of:

```
Sqrt<9,1,9>::value
    = (9<25) ? Sqrt<9,1,4>::value : Sqrt<9,5,9>::value
    = Sqrt<9,1,4>::value
```

- As a result `Sqrt<9,1,4>` is calculated next:

```
Sqrt<9,1,4>::value
    = (9<9) ? Sqrt<9,1,2>::value : Sqrt<9,3,4>::value
    = Sqrt<9,3,4>::value
```

- The next recursion step is then:

```
Sqrt<9,3,4>::value
    = (9<16) ? Sqrt<9,3,3>::value : Sqrt<9,4,3>::value
    = Sqrt<9,3,3>::value = 3
  \item This produces the desired result.
```

Template Instantiations

- However, there is a problem with the ternary operator

```
<condition> ? <true-path> : <false-path>
```

- The compiler generates not just the relevant part, but also the one that won't be used and isn't needed.
- This means it has to expand the next recursion level on that side as well (although the result will ultimately be discarded), and therefore the complete binary tree is assembled.
- For the square root of `N` this leads to `2N` template instantiations. This puts a large strain on the resources of the compiler (both runtime and memory), and limits the scope of the technique.
- This can be avoided using `std::conditional<B,T1,T2>`, which was introduced in C++11. We will examine its working mechanism by redefining it using a different name.

Type Selection at Compile Time

We can get rid of these unnecessary template instantiations by simply selecting the correct type and evaluating it directly.
This can be carried out with a small metaprogram that corresponds to an `if` statement (also called a "compile time type selection").

```
// Reimplementation of std::conditional<B,T1,T2> (C++11)

// Definition including specialization for the true case
template<bool B, typename T1, typename T2>
struct IfThenElse
{
  typedef T1 ResultType;
};

// Partial specialization for the false case
template<typename T1, typename T2>
struct IfThenElse<false,T1,T2>
{
  typedef T2 ResultType;
};
```

Improved Calculation of the Square Root

Using our meta-`if` statement we can implement the square root as follows:

```
template<std::size_t N, std::size_t L = 1, std::size_t H = N>
struct Sqrt
{
  enum{ mid = (L+H+1)/2 };

  using ResultType = typename IfThenElse<(N < mid*mid),
                     Sqrt<N,L,mid-1>, Sqrt<N,mid,H> >::ResultType;

  enum{ value = ResultType::value };
};

template<std::size_t N, std::size_t M>
struct Sqrt<N,M,M>
{
  enum{ value = M };
};
```

This requires only about $log_2(N)$ template instantiations!

Turing Completeness of Template Metaprogramming

Template meta programs may include:

- State variables: the template parameters.
- Loops: using recursion.
- Conditional execution: using the ternary operator `?:` or template specialization (e.g. the meta-`if`, or `std::conditional`, `std::is_same` and `std::enable_if`).
- Integer calculations.

This is sufficient to perform any calculation, as long as there isn't any restriction on the number of recursive instantiations and number of state variables (which doesn't imply that it is a good idea to calculate everything with template metaprogramming) .

C++11: constexpr

C++11 introduces a simple alternative to template metaprogramming: expressions that are already evaluated at compile time. In such a `constexpr` only variables or functions which are `constexpr` themselves may be used.

```
#include<iostream>

int x1 = 7;
constexpr int x2 = 7;

constexpr int x3 = x1; // Error, x1 is not a constexpr
constexpr int x4 = x2;

constexpr int Fac(int n)
{
  return n<2 ? 1 : n * Fac(n-1);
}

int main()
{
  std::cout << Fac(10) << std::endl;
}
```

C++11: constexpr

It must be possible to evaluate a `constexpr` at compile time:

```
void f(int n)
{
    constexpr int x = Fac(n); // Error, n isn't known at
                              // time of translation
    int f10 = Fac(10);        // Correct
}

int main()
{
    const int ten = 10;
    int f10 = Fac(10);        // Also correct
}
```

C++11: constexpr

This will even work for objects of classes whose constructor is simple enough to be defined as `constexpr`:

```
struct Point
{
  int x,y;
  constexpr Point(int x_, int y_) : x(x_), y(y_)
  {}
};

constexpr Point origin(0,0);
constexpr int z = origin.x;

constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2)};
constexpr x = a[1].x;   // x becomes 1
```

- `constexpr` functions may not have the return type `void` and neither variables nor functions may be defined within them (this also applies to `constexpr` constructors).
- The function body can only contain declarations and a single `return` statement.

C++11: auto

C++11 introduces the `auto` keyword for automatic type deduction. This can be used when a type is awkward to write or depends on template parameters:

```
auto val = functionWithSomeStrangeReturnType();
```

In C++11, return types of functions can also be `auto`, but then the actual type has to be specified after the argument list:

```
auto functionReturningInt() -> int
{
  return 5;
}
```

Starting with C++14, this trailing return type can be omitted when it can be deduced from the function body (i.e. consistent return type when more than one return statement is present):

```
auto functionReturningInt()
{
  return 5;
}
```

C++11: Range-Based `for`

C++11 also introduces range-based for loops, which are already known from other languages, e.g. Python. These work with anything that provides `begin` and `end` methods returning iterators or that can use the free `begin` and `end` functions.

Instead of writing:

```
typename Container::iterator it;
for (it = container.begin(); it != container.end(); ++it)
{
  sum += *it;
}
```

One can now use the shorter and more elegant version:

```
// for read-only access
for (const auto& element : container)
{
  sum += element;
}

// for write access
for (auto&& element : container)
{
  element = 1.;
}
```

C++11: Enum Classes

```
enum {RED, GREEN, BLUE}; // C enum
enum class Color {RED, GREEN, BLUE}; // C++11 enum
```

- C enums are simply constant integer values with a name.
- The same name may be used only once across all enums.
- Such an enum can be used in exactly the same places where any other integer value can be used.
- C++11 introduces enum classes, so that each enum class is its own type and has its own name and thus namespace. This way both problems described above are solved.
- Such enums can only be cast to integer explicitly, and therefore C enums remain useful for template metaprogramming.

C++11: Enum Classes

```
#include<iostream>

enum class TimeIntegration : unsigned char
    {EE = 2, IE = 4, CN = 8, BDF2 = 16};

// uses int as internal data type
enum class SpatialIntegration {CCFV,FCFV,FE,DG};

template<TimeIntegration T, SpatialIntegration S>
void DoTimeStep()
{
  // explicit conversion to int possible
  std::cout << (unsigned int)T << "␣" << (int) S << std::endl;
}

int main()
{
  // scope has to be included
  DoTimeStep<TimeIntegration::CN,SpatialIntegration::FE>();
  TimeIntegration ti = TimeIntegration::IE;
  ti = 1;  // not possible, no implicit conversion
  SpatialIntegration si = ti; // not possible, wrong type
}
```

C++11: Explicit Conversion

- In C++, constructors with one argument are automatically used for type conversion.
- This isn't always desired, e.g. it may be sensible to construct an $n \times n$ matrix by calling a constructor with one argument, but numbers should not automatically be turned into matrices.
- In C++11, constructors can be made `explicit`.
- In this case the constructor will only be used when it is explicitly called, for example with `Matrix(5)`.

C++11: Custom Literals

It would be nice if it was possible to define more literals in addition to the builtin ones, e.g.:

```
"Hi!"s                 // string, not ``zero-terminated array of char''
1.2i                   // imaginary number
123.4567891234df       // decimal floating point (IBM)
101010111000101b       // binary number (in C++14 as 0b101010111000101)
123s                   // seconds (actually in std::chrono since C++14)
123.56km               // kilometers
1234567890123456789012345678901234567890x   // extended-precision
```

Some of these habe been included in C++14:

```
"Hi!"s                 // std::string literal
1.2i                   // imaginary number
0b101010111000101      // binary number (note the position of suffix b))
123s                   // seconds (defined in std::chrono)
```

Note that there is no ambiguity between string and seconds, since the arguments have different type.

C++11: Custom Literals

Since C++11, literals can also be user-supplied (their suffix has to start with a '_', however). Here are implementations for imaginary numbers and strings, as they would be needed before C++14:

```
// imaginary literal   (introduced in C++14 without "_" prefix)
constexpr complex<double> operator"" _i(long double d)
{
    return {0,d}; // returns the appropriate complex number
}

// std::string literal (introduced in C++14 without "_" prefix)
std::string operator"" _s (const char* p, size_t n)
{
    return string(p,n);
}
```

This can be used as follows:

```
template<class T> void f(const T&);
f("Hello");       // hands const char* to function
f("Hello"_s);     // hands string to function
f("Hello"_s+"there");  // works, because first operand is string

auto z = 2+1_i;   // complex(2,1)
```

C++11: Raw String Literals

If one wants to use a backslash in a string, then it has to be written as \\. This makes the string difficult to read, especially in the newly introduced regular expressions:

```
// Hopefully getting this example right...
string s = "\\w\\\\\\w";
```

In a raw string literal, each character is simply written directly as such:

```
std::string s = R"(\w\\w)"; // I'm pretty sure I got that right
std::string path = R"(c:\path\to\foo.exe)";
```

The initial proposal for the introduction of raw string literals has been motivated by the following example:

```
"('(?:[^\\\']|\\.)*'|\"(?:[^\\\"]|\\.)*\")|"
// Are the five backslashes correct or not?
// Even experts become easily confused.
```

C++11: Raw String Literals

A raw string literal starts with `R"(` and ends with `)"` .

```
R"("quoted string")"     // the string is "quoted string"
```

Should it happen that the combination `"(` or `)"` occurs in the string, then an arbitrary combination of characters can be inserted between the parenthesis and the quotation marks to make the delimiter unique:

```
// "quoted string containing the usual terminator ("])"
R"***("quoted string containing the usual terminator ("])")***"
```

Short version of the above regular expression:

```
"'([^\\\\']|\\\\.)*'|\"([^\\\\\"]|\\\\.)*\""
```

Equivalent raw string literal:

```
R"('([^\\']|\\.)*'|\"([^\\\"]|\\.)*\")"
```

C++11: Regular Expressions

As in many other programming languages, regular expressions can also be used in C++11:

```
#include<regex>
#include<iostream>
#include<string>

int main()
{
  std::regex name_re(R"--(([a-zA-Z]+)\s+([a-zA-Z]+))--");
  std::string name="Santa Claus";
  if(regex_match(name.begin(),name.end(),name_re))
    std::cout << "Hello" << name << std::endl;
  else
    std::cout << "Who are you?" << std::endl;
}
```

A Modern C++ printf

The function `printf()` has been a simple C function in C++, but since C++11 a templatized type-safe variant can be written. This can be applied as follows:

```
#include"print.h"

int main()
{
  const char* pi = "pi";
  Printf("The␣value␣of␣%s␣is␣about␣%g␣(unless␣you␣live␣in␣%s).\n",
       pi, 3.14159, "Indiana");
  const std::string name = "Stefan";
  int age = 24;
  float grade = 1.3;
  Printf(
  R"(The␣student␣%s,␣%d␣years␣old,␣has␣received␣the␣grade␣%g.
He␣is␣among␣the␣top␣10␣%%.
)",name,age,grade);
}
```

Output:

```
The value of pi is about 3.14159 (unless you live in Indiana).
The student Stefan, 24 years old, has received the grade 1.3.
He is among the top 10 %.
```

C++11: Variadic Templates

The easiest case of `printf()` is the one where no arguments other than the format string exist:

```
#include<iostream>
#include<type_traits>
#include<stdexcept>

void Printf(const char* s)
{
  if (s == nullptr)
    return;
  while (*s)
  {
    // make sure that there aren't any arguments.
    // %% is a normal % in a format string.
    if (*s == '%' && *++s != '%')
          throw std::runtime_error("invalid format: missing
              arguments");
    std::cout << *s++;
  }
}
```

C++11: Variadic Templates

We now have to treat the case of `printf()` with several arguments. This requires a variable number of template arguments:

```
template<typename T, typename... Args> // note the "..."
void Printf(const char* s, T value, Args... args)
{
  while (s && *s)
  {
    if (*s == '%'){
      switch (*++s){
        case '%':
          break;
        case 's':
          if (!Is_C_style_string<T>() && !Is_string<T>())
            throw std::runtime_error("invalid format: not string");
          break;
        case 'd':
          if (!std::is_integral<T>())
            throw std::runtime_error("invalid format: not
                integral");
          break;
```

C++11: Variadic Templates

```
        case 'g':
          if (!std::is_floating_point<T>())
            throw std::runtime_error("invalid␣format:␣not␣floating␣
                point");
          break;
        default:
            throw std::runtime_error("invalid␣format:␣unrecognized␣
                option");
        }
        std::cout << value;
        return(Printf(++s,args...)); // here again note the "..."
      }
      std::cout << *s++;
  }
  throw std::runtime_error("invalid␣format:␣too␣many␣arguments");
}
```

- Using variadic templates, only the first element of the argument list is visible in each function call. `T` can be a different type for each call. The remainder may then be passed to the function again.
- The ellipses after `typename`, after the template type in the argument list and after the corresponding variable name in the next function call are important.

C++11: Variadic Templates

While the predicates `is_integral<T>()` and `is_floating_point<T>` are predefined, the predicates `Is_C_style_string<T>()` and `Is_string<T>()` have to be defined:

```
template<typename T>
bool Is_C_style_string()
{
  return std::is_same<T,const char*>() || std::is_same<T,char*>();
}

template<typename T>
bool Is_string()
{
  return std::is_same<T,const std::string>() ||
      std::is_same<T,std::string>();
}
```

C++11: Tuples

Another use of variadic templates are tuples, a generalization of pairs to any number of components.

- Their type can be generated automatically with the help of `auto` and `std::make_tuple`.
- The auxiliary function `std::get<i>` returns the i-th component of a tuple.

```
#include<tuple>
#include<string>

int main()
{
  std::tuple<std::string,int> t2("Mueller",123);
  auto t = std::make_tuple(std::string("Mayer"),10, 1.23);
  // t is of type tuple<string,int,double>
  std::string s = std::get<0>(t);
  int x = std::get<1>(t);
  double d = std::get<2>(t);
}
```

C++17: Fold Expressions

C++17 introduces fold expressions, which can automatically expand a parameter pack (the argument with ellipsis in variadic templates) and apply operators inbetween.

```
template<typename... Args>
int sum(Args&&... args)
{
  return (args + ... + 0);
}

template<typename... Args>
void print(Args&&... args)
{
  (std::cout << ... << args) << std::endl;
}
```

This can be used with 32 predefined binary operators, including all arithmetic and logic operators.

C++17: Fold Expressions

Call a function for each argument, despite not knowing the number of arguments beforehand (folds using the comma operator):

```
template<typename Func, typename... Args>
void callForAll(const Func& f, Args&&... args)
{
  ( f(args), ...);
}
```

Folds can be expanded to the left (right fold, ellipsis on the right) or to the right (left fold, ellipsis on the left), and each version can have an initial value (binary fold) or not (unary fold).

This means: The one operation that is written explicitly is actually the *last* to be executed, not the first, since the fold is realized using recursion.

C++17: Structured Bindings and Initialized `if`

C++17 adds two new constructs, structured bindings and a version of `if` with initializer.

Structured bindings make multiple return values possible by assigning names to components of an object:

```
auto [x,y,z] = f() // f returns object with three components
```

This works with C arrays, C-style structs, `std::array`, `std::pair` and `std::tuple`. `x`, `y` and `z` are then references of the entries / data members.

The `if` with initializer works similar to a `for` loop:

```
if (auto [x,y,z] = f(); x.isValid())
  {...}
else
  {...}
```

This avoids polluting the surrounding scope, as with the first entry of the `for` loop declaration.

Template Metaprogramming Example: Numbers with Units

Errors may occur when performing calculations with physical quantities.
The worst case scenario is comparing apples and oranges.

Goal is the construction of a class which allows calculating with units.

- The implementation uses template metaprogramming. All calculations (except conversion for input and output) are as fast as without units.
- The necessary tests are performed at compile time and automatically optimized out.

Units: Unit Class

We first introduce a template class for units:

```
template<int M, int K, int S>
struct Unit {
  enum { m = M, kg = K, s = S };
};

using M    = Unit<1,0,0>;  // Meters
using Kg   = Unit<0,1,0>;  // Kilogram
using S    = Unit<0,0,1>;  // Seconds
using MpS  = Unit<1,0,-1>; // Meter per second (m/s)
using MpS2 = Unit<1,0,-2>; // Meter per second squared (m/(s*s))
```

Units: Helper Classes

In order to calculate with units, we require some helper classes.
We build template functions with `using` declarations.

```
template<typename U1, typename U2>
struct Uplus {
  using type = Unit<U1::m+U2::m, U1::kg+U2::kg, U1::s+U2::s>;
};

template<typename U1, typename U2>
using Unit_plus = typename Uplus<U1,U2>::type;

template<typename U1, typename U2>
struct Uminus {
  using type = Unit<U1::m-U2::m, U1::kg-U2::kg, U1::s-U2::s>;
};

template<typename U1, typename U2>
using Unit_minus = typename Uminus<U1,U2>::type;

template<typename U>
using Unit_negate = typename Uminus<Unit<0,0,0>,U>::type;
```

Quantities

Now we can introduce a class in which the values are stored together with their units. Since the units are only used as a template parameter, this only affects the class type but does not require memory. In order to remain as flexible as possible, the data type is also a template parameter.

```
template<typename U, typename V=double>
struct Quantity {
  V val;
  explicit constexpr Quantity(V d) : val{d}
  {}
  template<typename V2>
  constexpr Quantity(Quantity<U,V2> d) : val{static_cast<V>(d.val)}
  {}
};
```

Quantities: Arithmetic Operations

We can now calculate with quantities. For a multiplication the units are added componentwise, while for a division they are subtracted. All data types which have an appropriate `operator*` or `operator/` can be used.

The appropriate data type of the return value is deduced using the C++11 `decltype` specifier.

```
template<typename U1, typename U2, typename V1, typename V2>
auto operator*(Quantity<U1,V1> x, Quantity<U2,V2> y)
{
  return
      Quantity<Unit_plus<U1,U2>,decltype(x.val*y.val)>(x.val*y.val);
}

template<typename U1, typename U2, typename V1, typename V2>
auto operator/(Quantity<U1,V1> x, Quantity<U2,V2> y)
{
  return
      Quantity<Unit_minus<U1,U2>,decltype(x.val/y.val)>(x.val/y.val);
}
```

Addition, subtraction and scaling with dimensionless constants can be implemented the same way.

Literals for Quantities

Now we can define a large amount of personal literals:

```
constexpr Quantity<M,long double> operator"" _m(long double value)
{
  return Quantity<M,long double> {value};
}

constexpr Quantity<Kg,long double> operator"" _kg(long double
    value)
{
  return Quantity<Kg,long double> {value};
}

constexpr Quantity<S,long double> operator"" _s(long double value)
{
  return Quantity<S,long double> {value};
}
```

These literals simply map a given value to the quantity of the desired unit.

Literals for Quantities

Other literals perform automatic conversion to the underlying unit:

```
constexpr Quantity<M,long double> operator"" _km(long double value)
{
  return Quantity<M,long double> {1000*value};
}

constexpr Quantity<Kg,long double> operator"" _g(long double value)
{
  return Quantity<Kg,long double> {value/1000};
}

constexpr Quantity<S,long double> operator"" _h(long double value)
{
  return Quantity<S,long double> {3600*value};
}
```

Literals for other data types, e.g. `long long`, and other units, e.g. micrograms or days, can be defined in the same way.

Quantities: Squaring and Comparison

We also want to be able to square quantities and compare them.

```
template<typename U, typename V>
Quantity<Unit_plus<U,U>,V> square(Quantity<U,V> x)
{
  return Quantity<Unit_plus<U,U>,V>(x.val*x.val);
}

template<typename U,typename V>
bool operator==(Quantity<U,V> x, Quantity<U,V> y)
{
  return x.val==y.val;
}

template<typename U,typename V>
bool operator!=(Quantity<U,V> x, Quantity<U,V> y)
{
  return x.val!=y.val;
}
```

Quantities: Larger and Smaller

Checks for inequality should even be possible for different data types when the unit matches:

```
}

template<typename U,typename V1, typename V2>
bool operator<(Quantity<U,V1> x, Quantity<U,V2> y)
{
  return x.val<y.val;
}

template<typename U,typename V1, typename V2>
bool operator>(Quantity<U,V1> x, Quantity<U,V2> y)
{
  return x.val>y.val;
}
```

Quantities: Output

We would like to be able to display quantities with the correct units. This can be achieved with a simple function and an overloaded output operator:

```
std::string suffix(int u, const char* x)
{
  std::string suf;
  if (u) {
    suf += x;
    if (1<u) suf += '0' + u;
    if (u<0) {
      suf += '-';
      suf += '0' - u;
    }
  }
  return suf;
}

template<typename U, typename V>
std::ostream& operator<<(std::ostream& os, Quantity<U,V> v)
{
  return os << v.val << suffix(U::m,"m") << suffix(U::kg,"kg") <<
      suffix(U::s,"s");
}
```

Quantities: Application Example

Now we can calculate with our quantities:

```
#include"units.h"

int main()
{
  Quantity<M,double> x {10.5};
  Quantity<S,int> y {2};
  Quantity<MpS,double> v = x/y;
  v = 2*v;
  auto distance = 10_m;
  Quantity<S,double> time = 20_s;
  auto speed = distance/time;
  Quantity<MpS2,double> acceleration = distance/square(time);

  std::cout << "Speed = " << speed << " Acceleration = " <<
      acceleration << std::endl;
}
```

Output:

```
Velocity = 0.5ms-1 Acceleration = 0.025ms-2
```

Summary

- Template metaprograms can perform calculations at compile time. They are based on elaborate template structures and recursion, and therefore hard to read.
- C++11 introduced `constexpr` as an alternative, compile-time calculations that look almost like regular C++.
- The introduction of `auto`, `decltype`, and range-based for loops in C++11 makes several verbose C++ idioms obsolete.
- Variadic templates, introduced in C++11, vastly expand the possibilities of generic programming. Fold expressions (C++17) are a direct extension of this.

Further Reading

Some highly relevant topics have been omitted in this course, these include:

- The I/O stream library
- The C++ thread support library (C++11)
- C++ random number generators (C++11)
- The C++ filesystem library (C++17)
- The mathematical function library (C++17)
- Lambda functions (C++11) and generic lambdas (C++14)
- Unordered associative containers (C++11)

A good starting point for these topics is en.cppreference.com.

Further information regarding best practices, C++ design choices and the philosophy of modern C++ can be found in the C++ Super FAQ (isocpp.org/faq).

A short presentation of the changes introduced by C++11/14/17 is available on multiple internet sites, one of them being e.g. `https://github.com/AnthonyCalandra/modern-cpp-features`.