Object-Oriented Programming for Scientific Computing

Template Metaprogramming

Ole Klein

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

30. Juni 2015

Calculating the Square Root

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

```
#include<iostream>

template<std::size_t N, std::size_t L=1, std::size_t H=N>
struct Sqrt
{
public:
  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
```

- 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 other that is not used. This means it has to expand the next recursion level on that side as well (although the result will be discarded), which results in the assembly of the complete binary tree.
- 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.

Type Selection at Compile Time

We can get rid of the 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").

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

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

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
{
public:
  enum{ mid = (L+H+1)/2 };

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

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

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

This only requires 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`).
- Integer calculations.

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

Loop Unrolling

In the calculation of a scalar product, as in:

```
template<typename T>
inline T dot_product (int dim, T* a, T* b)
{
  T result = T();
  for (int i=0;i<dim;++i)
  {
    result += a[i]*b[i];
  }
  return result;
}
```

the compiler often optimizes the computation for large arrays. However, if small scalar products of the type

```
dp = dot_product(3,a,b);
```

are used most of the time, then this may be written more efficiently.

Loop Unrolling

```
// Primary template
template <int DIM, typename T>
class DotProduct
{
  public:
    static T result (T* a, T* b)
    {
        return *a * *b  +  DotProduct<DIM-1,T>::result(a+1,b+1);
    }
};

// Partial specialization as stopping criterion
template <typename T>
class DotProduct<1,T>
{
  public:
    static T result (T* a, T* b)
    {
        return *a * *b;
    }
};
```

Loop Unrolling

```
// for simplification
template <int DIM, typename T>
inline T dot_product (T* a, T* b)
{
    return DotProduct<DIM,T>::result(a,b);
}
```

Application: Loop Unrolling

```
#include <iostream>
#include "loop_unrolling.h"

int main()
{
    int a[3] = { 1, 2, 3};
    int b[3] = { 5, 6, 7};

    std::cout << "dot_product<3>(a,b) = "
              << dot_product<3>(a,b) << '\n';
    std::cout << "dot_product<3>(a,a) = "
              << dot_product<3>(a,a) << '\n';
}
```

Loop Unrolling for Random Access Container

```
// Primary template
template <int DIM, typename T, template<typename
    U,typename=std::allocator<U> > class vect>
struct DotProduct
{
    static T result(const vect<T> &a, const vect<T> &b)
    {
        return a[DIM-1]*b[DIM-1]  +
            DotProduct<DIM-1,T,vect>::result(a,b);
    }
};

// Partial specialization as stopping criterion
template <typename T, template<typename
    U,typename=std::allocator<U> > class vect>
struct DotProduct<1,T,vect>
{
    static T result(const vect<T> &a, const vect<T> &b)
    {
        return a[0] * b[0];
    }
};
```

Loop Unrolling for Random Access Container

```
// For simplification
template <int DIM, typename T, template<typename
    U,typename=std::allocator<U> > class vect>
inline T dot_product(const vect<T> &a, const vect<T> &b)
{
    return DotProduct<DIM,T,vect>::result(a,b);
}
```

Application: Loop Unrolling for Random Access Containe

```
#include <iostream>
#include <vector>
#include "loop_unrolling2.h"

int main()
{
    std::vector<double> a(3,3.0);
    std::vector<double> b(3,5.0);

    std::cout << "dot_product<3>(a,b) = "
              << dot_product<3>(a,b) << '\n';
    std::cout << "dot_product<3>(a,a) = "
              << dot_product<3>(a,a) << '\n';
}
```

constexpr

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

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
}
```

constexpr

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

```
struct Point
{
  int x,y;
  constexpr Point(int xx, int yy) : x(xx), y(yy)
  {}
};

constexpr Point origo(0,0);
constexpr int z = origo.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.

Template Metaprogramming Example: Numbers with Units

- When performing calculations with physical quantities errors may occur.
- The worst case scenario is comparing apples and oranges.
- The aim is the construction of a class which allows calculating with units.
- The implementation uses template metaprogramming. All calculations (except the 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))
```

Enum Classes

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

- In C enums are simply integer values with a name.
- The same name may be used only once.
- 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.

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
}
```

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)}
  {}
};
```

Explicit Conversion

- Constructors with one argument are used by C++ for automatic type conversion.
- This isn't always desired. In this example it shouldn't be possible to add or subtract a number without unit to/from one with unit.
- 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 `Quantity<M>(2.73)`.

Quantities: Addition and Subtraction

We can now calculate with quantities. For addition and subtraction, the unit of the two operands and the result have to be the same. All data types which have an appropriate `operator+` or `operator-` can be used.

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

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

Quantities: Multiplication and Division

During multiplication the units are added componentwise, while for division they are subtracted.

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

Quantities: Multiplication by Values without Unit

The unit remains the same when a quantity is multiplied by a value without unit.

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

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

Quantities: Division using Values without Unit

If a quantity is divided by a number without unit the same as for multiplication holds. If the quantity is the divisor, the signs of the components of the unit have to be switched.

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

template<typename U,typename V1, typename V2>
auto operator/(V1 y, Quantity<U,V2>
    x)->Quantity<Unit_negate<U>,decltype(y/x.val)>
{
  return Quantity<Unit_negate<U>,decltype(y/x.val)>(y/x.val);
}
```

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
123s                // seconds
123.56km            // kilometers
12345678901234567890123456789012345678901234567890x   // extended-precision
```

Custom Literals

This is possible in C++11, e.g. for complex numbers and strings (their suffix has to start with a '_', however):

```
constexpr complex<double> operator"" _i(long double d)  //
    imaginary literal
{
    return {0,d};   // returns the appropriate complex number
}

std::string operator"" _s (const char* p, size_t n)   //
    std::string literal
{
    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+"Dolly");  // works, because first operand is string

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

Custom Literals

C strings may also be passed directly:

```
Bignum operator"" _x(const char* p)
{
    return Bignum(p);
}

void f(Bignum);
f(12345678901234567890123456789012345678901234567890_x);
```

but not always:

```
std::string operator"" _S(const char* p);  // this doesn't work

std::string blub = "one␣two"_S; // Error: no applicable literal
    operator
```

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:

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

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:\Programme\blub\blob.exe)";
```

The first 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.
```

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:

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

Short version of the above regular expression:

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

Equivalent raw string literal:

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

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="Torsten␣Will";
  if(regex_match(name.begin(),name.end(),name_re))
    std::cout << "Hello" << name << std::endl;
  else
    std::cout << "Who␣are␣you?" << std::endl;
}
```

String to Number and Number to String Conversion

```
#include<string>
#include"print.h"

int main()
{
  std::string sVal = "-2.47␣3.1415␣1e300␣42␣376857689403␣0xFF";
  size_t next=0;
  float fVal=std::stof(sVal,&next);
  sVal = sVal.substr(next);
  double dVal=std::stod(sVal,&next);
  sVal = sVal.substr(next);
  long double ldVal=std::stold(sVal,&next);
  sVal = sVal.substr(next);
  int iVal=std::stoi(sVal,&next);
  sVal = sVal.substr(next);
  long lVal= std::stol(sVal,&next);
  sVal = sVal.substr(next);
  unsigned long ulVal= std::stoul(sVal,&next,16);
  Printf("%d,␣%d,␣%d,␣%s,␣%g,␣%s\n",iVal,lVal,ulVal,
      std::to_string(fVal),dVal,std::to_string(ldVal));
}
```

Literals for Quantities with Floating Point Numbers

Now we can define a large amount of personal literals. First for quantities which are using floating point numbers as data type:

```
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};
}

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

Literals for Quantities with Floating Point Numbers

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

constexpr Quantity<Kg,long double> operator"" _mg(long double
    value)
{
  return Quantity<Kg,long double> {value/1000000};
}

constexpr Quantity<S,long double> operator"" _min(long double
    value)
{
  return Quantity<S,long double> {60*value};
}

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

Literals for Quantities with Floating Point Numbers

```
constexpr Quantity<S,long double> operator"" _d(long double value)
{
  return Quantity<S,long double> {86400*value};
}

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

constexpr Quantity<S,long double> operator"" _us(long double value)
{
  return Quantity<S,long double> {value/1000000};
}

constexpr Quantity<S,long double> operator"" _ns(long double value)
{
  return Quantity<S,long double> {value/1000000000};
}
```

Literals for Quantities with Integers

Then also for quantities that use integer data types:

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

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

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

And so on, as above for floating point numbers.

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
```

C++ Printf

The function `printf()` has been a simple C function in C++, but in C++11 this isn't necessarily the case any more. The application looks 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␣1␣%%.
)",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 1 %.
```

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

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("Bad Printf() format");
          break;
        case 'd':
          if (!std::is_integral<T>())
            throw std::runtime_error("Bad Printf() format");
          break;
```

Variadic Templates

```
        case 'g':
          if (!std::is_floating_point<T>())
            throw std::runtime_error("Bad Printf() format");
          break;
        default:
            throw std::runtime_error("Unknown Printf() format");
        }
        std::cout << value;
        return(Printf(++s,args...)); // here again note the "..."
      }
      std::cout << *s++;
  }
  throw std::runtime_error("Extra arguments provided to Printf");
}
```

- 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.

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>();
}
```

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);
}
```

External Templates

In projects that consist of many individual files templates are often instantiated in several places. This does not happen if they are declared with the keyword `extern`, for example in the header file `extern.h`:

```
#include<vector>

extern template class std::vector<int>;

int blub(std::vector<int>& a)
{
  return a[0];
}
```

Externe Templates

and also not in the main program `extern.cc`

```
#include"extern.h"

extern template class std::vector<int>;

int main()
{
  std::vector<int> x(5,5.);
  int y = blub(x);
}
```

Externe Templates

The actual instantiation is done explicitly in a well-defined location, in this example in the file `extern2.cc`:

```
#include<vector>

template class std::vector<int>;
```

It is important not to forget to link all the files together, in the simplest case by: `g++ extern2.cc extern.cc`