Thinking with templates

Aleksandra Rylund Glesaaen
aleksandra@glesaaen.com

C++ User Group Meeting
January 27th 2016

Disclaimer

Today’s topics

1. Understanding types
2. Transcending types

Goal

Write code that is easy to use correctly
but hard to use incorrectly

Understanding types

The information stored in types

```
double power(double, int);
```

The information stored in types

```
void start(Widget &);
```

The information stored in types

The type tells the compiler

- How much space an object needs in memory
- What operations can be carried out on the object

(lets ignore type specifiers for now)

The information stored in types

The type tells the compiler

- How much space an object needs in memory
- What operations can be carried out on the object

(lets ignore type specifiers for now)

The information stored in types

```
Widget w = Widget{} + Widget{};
```

The information stored in types

```
Widget w = Widget{} + Widget{};
```

What are the requirements on the `Widget` type for this line to compile?

Declaring typenames

Class declaration

```
struct my_struct { ... };

class my_class { ... };

enum class my_enum { ... };
```

Can be compared to variable declaration

Declaring typenames

Class declaration

```
class my_class
  : public my_base_class { ... };
```

Reads:

”The class my_class is a my_base_class”

Declaring typenames

Class declaration

```
class my_class
  : public my_base_class { ... };
```

Reads:

"The class my_class is a my_base_class"

Much like variable assignment

```
var my_var = my_base_var;
```

Declaring typenames
Type aliasing

```
typedef std::vector<double> Vec;

using WidgetFunction =
  std::function<void(Widget&)>;
```

Explicit interfaces

```
class Widget
{
public:
  using value_type = double;
  using refernce = double&;

  Widget();
  ~Widget();

  reference operator[](std::size_t);
  value_type calculate() const;

  void swap(Widget &);
};
```

Nested types

Types can be nested inside of other compound types

```
class Widget
{
  class iterator { ... };

  enum class AccessType { ... };

  using pointer = std::unique_ptr<DataType>;

  ...
};
```

They provide an extended explicit interface

Nested types

Example: Type array

```
struct TypeArray
{
  using one = double;
  using two = std::string;
  using three = WidgetFunction;
  using four = std::list<int>;
  using five = std::nullptr_t;
  ...
};
```

Nested types
Example: Type map

```
struct TypeMap
{
  struct one
  {
    using first = int;
    using second = std::string;
  };

  struct two
  {
    using first = double**;
    using second = function<void(double**)>;
  };
  ...
};
```

Nested types

The nested types can be accessed using ::

```
using x = TypeArray::three;
using y = TypeMap::two::first;
```

Nested types

Nested types should provide information about its parent type

```
class Widget
{
  using pointer = DataType*;

  pointer data() const;

  ...
};

Widget w {};
Widget::pointer data_ptr = w.data();
```

Nested types

Nested types should provide information about its parent type

```
class Widget
{
  using pointer = std::shared_ptr<DataType>;

  pointer data() const;

  ...
};

Widget w {};
Widget::pointer data_ptr = w.data();
```

Why should you care?

Types is the language your compiler speaks

- Better control over the compiler through type manipulation
- Type specific logic errors to be checked at compile time
- Can wirte code that are optimized and readable at the same time

Transcending types

Types as compile time variables

Standard scenario: dereference & swap

```
using VecIterator
  = std::vector<double>::iterator;

void
iterator_swap(VecIterator i1, VecIterator i2)
{
  double tmp = *i1;
  *i1 = *i2;
  *i2 = tmp;
}
```

It would be natural that this function should work for all iterators that can be dereferenced & assigned

Types as compile time variables

```
template <typename InputIt1, typename InputIt2>
void iterator_swp(InputIt1 it1, InputIt2 it2)
{
    ?    tmp = *it1;

  *it1 = *it2;
  *it2 = tmp;
}
```

Have we lost information?

The compiler should know what type *it1 gives when instansiating the template function

Types as compile time variables

This information that can be stored in nested types

```
template <typename InputIt1, typename InputIt2>
void iterator_swp(InputIt1 it1, InputIt2 it2)
{
  using deref_type
    = typename InputIt1::value_type;

  deref_type tmp = *it1;
  *it1 = *it2;
  *it2 = tmp;
}
```

But no way to make it compatible with built in types

Fundamental theorem of software engineering

We can solve any problem by introducing an extra level of indirection.

David J. Wheeler

Types as compile time variables

```
template <typename Iterator>
struct iterator_traits
{
  using value_type
    = typename Iterator::value_type;
  ...
};

template <typename Ptr>
struct iterator_traits<Ptr*>
{
  using value_type = Ptr;
  ...
};
```

Types as compile time variables

```
template <typename Iterator>
struct iterator_traits
{
  using value_type
    = typename Iterator::value_type;
  ...
};

template <typename Ptr>
struct iterator_traits<Ptr*>
{
  using value_type = Ptr;
  ...
};
```

blah blah blah...

Automatic type deduction

auto

Used as a type definition, (mostly) carries out standard template type deduction on the right hand side of the assignment operator

decltype

Given a name or an expression, returns the name's or expression's type

Automatic type deduction

We can use auto to fix our type issue

```
template <typename InputIt1, typename InputIt2>
void iterator_swp(InputIt1 it1, InputIt2 it2)
{
  auto tmp = *it1;
  *it1 = *it2;
  *it2 = tmp;
}
```

Return type deduction

```
template <typename Func, typename Type>
    ?     map_function(Func f, Type val)
{

  return f.apply(val);
}
```

Return type deduction

```
template <typename Func, typename Type>
   ?    map_function(Func f, Type val)
{

  return f.apply(val);
}
```

We want whatever type this expression returns

Return type deduction
Attempt one

We know basically what we want

```
template <typename Func, typename Type>
decltype(f.apply(val))
map_function(Func f, Type val)
{
  return f.apply(val);
}
```

But how do we wrangle this information from the compiler?

Return type deduction
Attempt one

We know basically what we want

```
template <typename Func, typename Type>
decltype(f.apply(val))
map_function(Func f, Type val)
{
  return f.apply(val);
}
```

names f and val not yet declared

But how do we wrangle this information from the compiler?

Return type deduction

C++11 trailing return type

```
template <typename Func, typename Type>
auto map_function(Func f, Type val)
  -> decltype(f.apply(val))
{
  return f.apply(val);
}
```

Return type deduction

C++14 automatic return type deduction

```
template <typename Func, typename Type>
auto map_function(Func f, Type val)
{
  return f.apply(val);
}
```

Return type deduction

C++14 automatic return type deduction

```
template <typename Func, typename Type>
auto map_function(Func f, Type val)
{
  return f.apply(val);
}
```

Might not always produce the expected return type

Template type deduction

Assume the following piece of template pseudocode

```
template <typename Type>
void foo( ParamType param);

foo( expr );
```

The deduced types of `Type` and `ParamType` from *`expr`* depends on the form of ParamType

Template type deduction

```
template <typename Type>
void foo( ParamType param);
foo( expr);
```

Case 1:

ParamType is a reference or a pointer

(but not a && reference)

1. Ignore reference part of expr
2. Pattern-match expr’s type with ParamType to deduce Type

Template type deduction

```
template <typename Type>
void foo( ParamType param);

foo( expr);
```

Case 2:

ParamType is a universal reference

- If expr is an lvalue reference, ParamType will be deduced to be an lvalue reference
- If expr is an rvalue reference, standard rules apply

Template type deduction

```
template <typename Type>
void foo( ParamType param);

foo( expr );
```

Case 3:

ParamType is not a pointer nor a reference

1. Ignore reference and const part of expr
2. Pattern-match expr's type with ParamType to deduce Type

Return type deduction

This will slice any references from the return type

```
template <typename Func, typename Type>
auto map_function(Func f, Type val)
{
  return f.apply(val);
}
```

Return type deduction

Using decltype will pattern match correctly

```
template <typename Func, typename Type>
decltype(auto) map_function(Func f, Type val)
{
  return f.apply(val);
}
```

Template pattern matching

Can use the pattern matching to extract types from templates

```
template <typename Type>
Type extract(Widget<Type>) { ... }
```

Template pattern matching

...or the other way around

```
template <
  template Other,
  template <typename...> class Policy
>
Policy<Other> replace(Policy<Widget>) { ... }
```

Template pattern matching

Also good for restricting pattern matching when you know what patterns you expect to be valid

```
template <
  template <typename...> class CreationPolicy
>
class WidgetManager
  : public CreationPolicy<Widget>
{ ... }
```

Template pattern matching

Note that no implicit conversions are considered during type deduction

```
template <typename T>
void fill(std::vector<T> &v, T x);

std::vector<double> vec(6);
fill(vec, 1);
```

```
error: no matching function for call to 'fill'
  fill(vec, 1);
  ^---
note: candidate template ignored: deduced conflicting types for
parameter 'T' ('double' vs. 'int')
```

Template pattern matching

Note that no implicit conversions are considered during type deduction

```
template <typename T>
void fill(std::vector<T> &v, T x);

std::vector<double> vec(6);
fill(vec, 1);
```

Not even between built in types

Template pattern matching

Recursive pattern matching for variadic templates

```
void println(std::ostream & os)
{
  os << std::endl;
}

template <typename H, typename... T>
void println(std::ostream & os, const H & head, T... tail)
{
  os << head;
  if( sizeof...(tail) != 0)
    os << ", ";

  println(os,tail...);
}


println(std::cout, 7, 8.43, 'c', "Hello");
```

Encoding intent in types

The C++ Guidelines have suggested a new template type to signal resource ownership

```
template <typename T>
using owner = T;
```

1. The code signals the intent of the programmer
2. Can be checked by the compiler

Encoding intent in types

```
owner<Widget*> FactoryMethod() {...};
```

← factory methods need to return owner types

```
Widget* w1 = FactoryMethod();

Widget* w2 = new Widget {};


auto w3 = FactoryMethod();
Widget* w4 = w3;
...
delete w4;
```

Encoding intent in types

```
owner<Widget*> FactoryMethod() {...};

Widget* w1 = FactoryMethod();
```

error: information on ownership lost

```
Widget* w2 = new Widget {};


auto w3 = FactoryMethod();
Widget* w4 = w3;
...
delete w4;
```

Encoding intent in types

```
owner<Widget*> FactoryMethod() {...};

Widget* w1 = FactoryMethod();

Widget* w2 = new Widget {};
```

error: cannot assign newed objects to non-owners

```
auto w3 = FactoryMethod();
Widget* w4 = w3;
...
delete w4;
```

Encoding intent in types

```
owner<Widget*> FactoryMethod() {...};

Widget* w1 = FactoryMethod();

Widget* w2 = new Widget {};


auto w3 = FactoryMethod();
Widget* w4 = w3;
...
delete w4;
```

ok: a raw pointer simply points to something

Encoding intent in types

```
owner<Widget*> FactoryMethod() {...};

Widget* w1 = FactoryMethod();

Widget* w2 = new Widget {};


auto w3 = FactoryMethod();
Widget* w4 = w3;
...
delete w4;
```

error: cannot delete non-owners

Implicit interfaces

```
template <typename Widget, typename Operator>
void check_and_apply(Widget &w, Operator op)
{
  if (w.size() > 10 and !w.bad())
    op.apply(w);
}
```

The expressions in the functon body make up the template's implicit interface

Curiously recurring template pattern

```
template <typename T>
class Base { ... };

class Derived
  : public Base<Derived> { ... };
```

Curiously recurring template pattern

Allows for static polymorphism

```
template <typename Type>
class Base
{
public:
  Type& self()
  {
    return static_cast<Type&>(*this);
  }

  void implementation()
  {
    self().implementation();
  }
};
```

Curiously recurring template pattern

Allows for static polymorphism

```
class Derived
  : public Base<Derived>
{
public:
  void implementation() { ... };
};

template <typename Type>
void call(Base<Type> widget)
{
  widget.implementation();
}
```

Curiously recurring template pattern

Makes it easier to put things in a common box

```
template <typename Val>
class unary { ... };

template <typename LVal, typename RVal>
class binary { ... };

template <typename LValU, RValU>
auto operate(unary<LValU> left, unary<RValU> right)
 -> binary<unary<LValU>, unary<RValU>> { ... };

template <typename LValU, RValBL, RValBR>
auto operate(unary<LValU> left, binary<RValBL,RValBR> right)
 -> binary<unary<LValU>, binary<RValBL,RValBR>> { ... };

...
```

Curiously recurring template pattern

Makes it easier to put things in a common box

```
template <typename Type>
class base { ... };

template <typename Val>
class unary : public base<unary<Val>> { ... };

template <typename LVal, typename RVal>
class binary : public base<binary<LVal,RVal>> { ... };

template <typename LExpr, RExpr>
auto operate(base<LExpr> left, base<RExpr> right)
 -> binary<LExpr,RExpr> { ... };
```

Building trees

```
struct plus {};
struct minus {};
struct times {};
struct divide {};

template <typename Expr>
struct base_expr {};

template <typename Op, typename Le, typename Re>
struct binary_expr : base_expr<binary_expr<Op,Le,Re>> {};

struct val : base_expr<val> {};
```

Building trees

```
template <typename Le, typename Re>
auto operator+(base_expr<Le>, base_expr<Re>)
  -> binary_expr<plus,Le,Re>
{
  return {};
}

template <typename Le, typename Re>
auto operator-(base_expr<Le>, base_expr<Re>) {...}

template <typename Le, typename Re>
auto operator*(base_expr<Le>, base_expr<Re>) {...}

template <typename Le, typename Re> {...}
auto operator/(base_expr<Le>, base_expr<Re>) {...}

int main()
{
  val v;
  auto expr = v + v - v * v / (v + v);
}
```

Resources

[1] **C++ core guidelines.**
`https://github.com/isocpp/CppCoreGuidelines`.

[2] **C++ reference.**
`http://cppreference.com`.

[3] **cppcon: The c++ conference.**
`http://cppcon.org`.

[4] **S. Meyers.**
Effective Modern C++.
O'Reilly Media, 2014.