初心者でもわかる(かもしれない)

Boost.Hana

C++14世代のメタプログラミング

札幌C++勉強会 #9 2015/10/17

発表者

- @ignis_fatuus
- 住んでるとこ
  - 岐阜（なごやから電車で40分)。
- やってること
  - 某研究所で研究員, Ph.Dというやつ。
  - 実験解析とか物理シミュレーション。

対象者

- 新しめのC++に興味がある人
- メタプログラミングに興味がある人

Boost.Hanaとは

- メタプログラミング・コンパイルタイム計算ができる。
- ヘテロな(異なる型が混在した)コンテナがある。
- コンパイルタイム計算(型とか)もランタイム計算(値とか)も同じように書ける。
- 値計算を利用した型計算

- まだリリース版のBoostには含まれてない
- Official Boost Library
- Header-only

Disclaimers

This is now an official Boost library! The results of the formal review can be seen here. However, the library is still not included in a Boost release, and API stability is still not assured.

written in https://github.com/ldionne/hana

Requirements

Compiler/ Toolchain	Status
Clang >= 3.5.0	Fully working; tested on each push to GitHub
Xcode >= 6.3	Fully working; tested on each push to GitHub
GCC >= 5.1.0	Almost working; waiting for the GCC team to fix a couple of C++14-related bugs

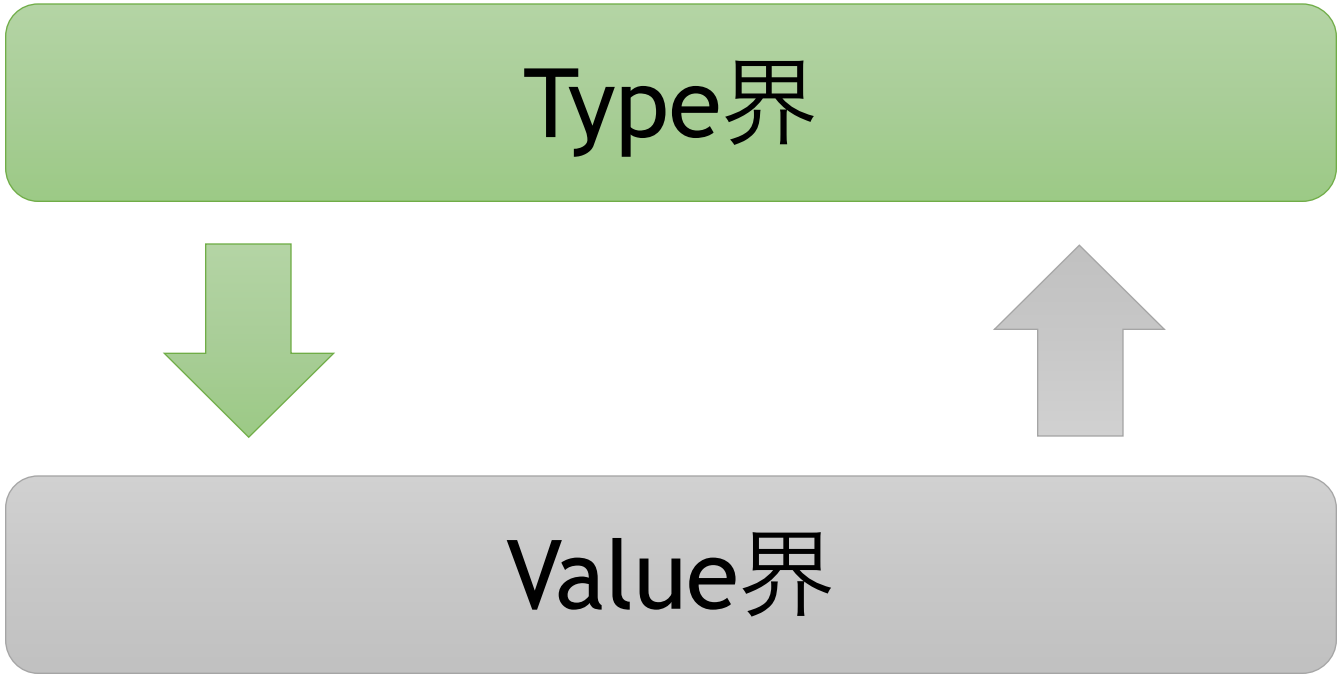
Boost.Hanaを学ぶには

- オススメ → Officialドキュメント
  - http://ldionne.com/hana/
  - 読みやすい。

- 詳細が気になったら
  - ソースコード読もう
  - Boost.Hanaは他のBoostライブラリへの依存度が低い。
  - C++14がわかるならHanaだけ読んでも理解できる。

Boost.Hanaのアプローチ

- Type-levelの計算をValue-levelの計算を通して行う。



Arithmetic example

- Type界での整数表現

```
using one_t = integral_constant<int,1>;// type
using two_t = integral_constant<int,2>;
```

- Value界に降りる

```
auto one = one_t{}; // value
auto two = two_t{};
```

- Value界での法則

```
template<typename V, V v, typename U, U u>
auto operator+(integral_constant<V,v>,integral_constant<U,u>)
        {  return integral_constant<decltype(v+u), v+u>{}; }

auto three = one + two;  //  ok
```

```
template<class T, T v>
struct integral_constant {
   static constexpr T value = v;
   …
};
```

Arithmetic example

- Type界での整数表現

```
using one_t = integral_constant<int,1>;// type
using two_t = integral_constant<int,2>;
```

alias template (since c++11)

usingで型の別名がつけられる。

```
template<class T>
using new_type = long_name_type<T>;
new_type<int> v = ...;
```

```
template<class T, T v>
struct integral_constant {
  static constexpr T value = v;
  ...
};
```

constexpr (since c++11)

定数式

Arithmetic example

- Type界での整数表現
```
using one_t = integral_constant<int,1>;// type
using two_t = integral_constant<int,2>;
```
- Value界に降りる
```
auto one = one_t{}; // value
auto two = two_t{};
```

```
template<class T, T v>
struct integral_constant {
   static constexpr T value = v;
   …
};
```

型推論 auto (since c++11)

統一初期化記法(Uniform Initialization)(since c++11)

波カッコ{}で初期化できる。

Arithmetic example

- Type界での整数表現

```
using one_t = integral_constant<int,1>;// type
using two_t = integral_constant<int,2>;
```

- Value界に降りる

```
auto one = one_t{}; // value
auto two = two_t{};
```

- Value界での法則

```
template<typename V, V v, typename U, U u>
auto operator+(integral_constant<V,v>,integral_constant<U,u>)
        {  return integral_constant<decltype(v+u), v+u>{}; }
```

```
template<class T, T v>
struct integral_constant {
   static constexpr T value = v;
   …
};
```

演算子オーバーロード(c++03)

関数返り値の型推論(since c++14)

decltype (since c++11)
valueの型を取得(since c++11)

Arithmetic example

- Type界への帰還

```
auto three = one + two; // three は value
using three_t = ???     // three_t は
type
```

```
template<class T, T v>
struct integral_constant {
  static constexpr T value = v;
  …
};
```

Arithmetic example

- Type界への帰還

```
auto three = one + two; // three は value
using three_t = decltype(three);
                        // three_t は type
static_assert(three::value == 3, “”);
                        // ok, this is compile-time computation
```

```
template<class T, T v>
struct integral_constant {
   static constexpr T value = v;
   …
};
```

static_assert (since c++11)
コンパイル時にチェック。
実行時の値だとコンパイルエラー。

ポイント

- Value界での計算によってType界での結果が得られる。
- threeはcostexprではない。

Type界だけでやると

```
template <typename X, typename Y>
struct plus {
   using type =
      integral_constant<decltype(X::value + Y::value),  X::value + Y::value>;
};
using three_t = plus<one_t, two_t>::type; // メタ関数
```

- メタ関数でやろうとすると
  → 式が複雑化すると読みにくい
- Value界で計算
  → Runtimeと同じ記法： auto three = one + two
  読みやすい

メタ関数 (c++03)
引数→template引数, 返り値→::type
とみなす型操作関数

Example: Euclidean distance $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

- Type界だけ

```
template <typename P1, typename P2>
struct distance {
  using xs = typename mpl::minus<typename P1::x,
                                 typename
P2::x>::type;
  using ys = typename mpl::minus<typename P1::y,
                                 typename
P2::y>::type;
  using type = typename sqrt<
                    typename mpl::plus<
                        typename mpl::multiplies<xs,
xs>::type,
                        typename mpl::multiplies<ys, ys>::type
                    >::type
                >::type;
};
static_assert(mpl::equal_to<
                 distance<point<mpl::int_<3>, mpl::int_<5>>,
                          point<mpl::int_<7>, mpl::int_<2>>
                >::type,   mpl::int_<5>>::value, "");
```

- Value界で計算

```
template <typename P1, typename P2>
constexpr auto distance(P1 p1, P2 p2) {
        auto xs = p1.x - p2.x;
        auto ys = p1.y - p2.y;
        return sqrt(xs*xs + ys*ys);
}

auto ic = distance(point(3_c, 5_c), point(7_c, 2_c))==5_c;
static_assert( decltype(ic)::value, "");
```

※ “distance”の返り値はIntegralConstant.

IntegralConstantとは

- 数値(など)を持った型，数値を型として表現したもの
  - Integral_constnt<int,1>, integral_constant<bool, true>
- 内包する数値はconstexprとして取り出せる
- Boost.Hanaではtemplate aliasとvariable templateによる略記可
  - template<int i> using int_ = integral_constant<int, i>;
  - template<int i> constexpr int_<i> int_c{};

int_<3>　→　型, integral_constant<int,3>

int_c<3>　→　値, integral_constant<int,3>{}

- hana::literals名前空間を使えばユーザ定義リテラルでも可
  - ex) 3_c, 5_c

IntegralConstantとは

- 数値(など)を持った型，数値を型として表現したもの
  - Integral_constnt<int,1>, integral_constant<bool, true>
- 内包する数値はconstexprとして取り出せる
- Boost.Hanaではtemplate aliasとvariable templateによる略記可
  - template<int i> using int_ = integral_constant<int, i>;
  - template<int i> constexpr int_<i> int_c{};

variable template (since c++14)
型に応じて変数定義を変えられる。

```
template<class T> std::string type{"default"};
template<> std::string type<int>{"int"};
template<> std::string type<char>{"char"};
std::cout << type<int> << std::endl;
std::cout << type<char> << std::endl;
```

IntegralConstant

- hana::tupleの要素指定
  - auto values = hana::make_tuple(1, ‘x’, 3.1415f);
  - float pi = values[2_c];

ヘテロなコンテナ

- Hana alogorithmに渡すpredicateのreturn value
  - auto result = hana::find_if(values, [](auto const& x) {
    return hana::traits::is_integral(x); // IntegralConstant
    });

- repeat (loop unrolling)

```
auto f = [](){ std::cout<< “Boost.Hana” << std::endl;};
hana::int_c<3>.times(f);  // equivalent to  f(), f(), f()
```

IntegralConstant

- hana::tupleの要素指定
  - auto values = hana::make_tuple(1, ‘x’, 3.1415f);
  - float pi = values[2_c];

ヘテロなコンテナ

- Hana alogorithmに渡すpredicateのreturn value
  - auto result = hana::find_if(values, [](auto const& x) {
    return hana::traits::is_integral(x); // IntegralConstant
    });

Generic lambda (since c++14)
ラムダ式の引数にautoを指定できる

- repeat (loop unrolling)

```
auto f = [](){ std::cout<< “Boost.Hana” << std::endl;};
hana::int_c<3>.times(f);  // equivalent to  f(), f(), f()
```

“times” implementation

```
template <typename T, T v>
template <typename F>
constexpr void times_t<T, v>::operator()(F&& f) const
{ go<T, ((void)sizeof(&f), v)>::without_index(static_cast<F&&>(f)); }

template <typename T, T N, typename = std::make_integer_sequence<T, N>>
struct go;

template <typename T, T N, T ...i>
struct go<T, N, std::integer_sequence<T, i...>> {
        using swallow = T[];

        template <typename F>
        static constexpr void without_index(F&& f){ (void)swallow{T{}, ((void)f(),
i)...}; }
};
```

integer_sequenceで展開してるだけ

“times” implementation

```
template <typename T, T v>
template <typename F>
constexpr void times_t<T, v>::operator()(F&& f) const
{ go<T, ((void)sizeof(&f), v)>::without_index(static_cast<F&&>(f)); }

template <typename T, T N, typename = std::make_integer_sequence<T, N>>
struct go;

template <typename T, T N, T ...i>
struct go<T, N, std::integer_sequence<T, i...>> {
    using swallow = T[];

    template <typename F>
    static constexpr void without_index(F&& f){ (void)swallow{T{}, ((void)f(),
i)...}; }
};
```

integer_sequenceで展開してるだけ

swallow ?

What's the swallow ?

- templateパラメータパックの順序問題

```
static int counter=0;
int inc() { return counter++; }

template<class ...T>
void do_something(){    [](...){}   ((T::var = inc())...);    }

template<int> struct S { };
template<> struct S<0> { static int var; }; int S<0>::var = 0;
template<> struct S<1> { static int var; }; int S<1>::var = 0;
template<> struct S<2> { static int var; }; int S<2>::var = 0;

do_something <S<0>,S<1>,S<2>> (); // variadic templateに投げる

int main(){ std::cout << S<0>::var << S<1>::var << S<2>::var; }  // 出力結果は？
```

What's the swallow ?

- templateパラメータパックの順序問題

```
static int counter=0;
int inc() { return counter++; }

template<class ...T>
void do_something(){    [](...){}   ((T::var = inc())...);    }

template<int> struct S { };
template<> struct S<0> { static int var; }; int S<0>::var = 0;
template<> struct S<1> { static int var; }; int S<1>::var = 0;
template<> struct S<2> { static int var; }; int S<2>::var = 0;

do_something <S<0>,S<1>,S<2>> (); // variadic templateに投げる

int main(){ std::cout << S<0>::var << S<1>::var << S<2>::var; }  // 出力結果
は？
```

答え

GCCなら210

Clangなら012

(順序が決まってない)

What's the swallow ?

- templateパラメータパックの順序は決まってないがinitializer_listだと決まる

```
template<class ...T>
void do_swallow()
{
    using swallow = int[];
    (void)swallow{ (T::var = inc())... };
}
```

```
do_swallow <S<0>,S<1>,S<2>> ();
int main(){ std::cout << S<0>::var << S<1>::var << S<2>::var; }
```

012

GCCでもClangでも

型と値に対する統一的な記述

Hanaが使えない時代では…

```
// types (MPL)
using types = mpl::vector<int*, char&, void>;
using ts = mpl::copy_if<types,
                        mpl::or_<std::is_pointer<mpl::_1>,
                                 std::is_reference<mpl::_1>>>::type;
// values (Fusion)
auto values = fusion::make_vector(1, 'c', nullptr, 3.5);
auto vs = fusion::filter_if<std::is_integral<mpl::_1>>(values);
```

- 型と値の操作は別のライブラリ
- インターフェイスも全く異なる
- Hanaの目的 → 型と値の操作を統一的手法で行う

型と値に対する統一的な記述

Hanaを使うと…

```
// types
auto types = hana::tuple_t<int*, char&, void>; // types as value
auto ts = hana::filter(types, [](auto t) {
        return is_pointer(t) || is_reference(t);
});

// values
auto values = hana::make_tuple(1, 'c', nullptr, 3.5);
auto vs = hana::filter(values, [](auto t) {
        return is_integral(t);
});
```

Boost.Hana → 型でも値でも同じような書き方ができる。
読みやすい。

コンパイル時条件分岐

```
struct NotHas_xxx { };
struct Has_xxx { const int xxx; };

int main() {
    auto has_xxx = hana::is_valid(
                       [](auto&& x) -> decltype((void)x.xxx){});

    auto x = NotHas_xxx{};
    hana::if_(has_xxx(x),
            [](auto&& x) { std::cout << "has_xxx" << std::endl; },
            [](auto&& x) { std::cout << "not has_xxx" << std::endl; }
        )(x);
}
```

メンバ変数xxxを持っているかを判定する関数オブジェクト

- メンバ変数xxxを持つかどうかによってコンパイルタイムに呼び出される関数が決まる。

まとめ

- Boost.Hanaによってメタプログラミングが簡単になる。
- Boost.HanaはType計算をValue界で行う。
- 型と値を統一的な記法で扱える。