Variadic expansion in examples

Michał Dominiak
Nokia Networks
griwes@griwes.info

Outline

1. Introduction to variadic templates
2. Variadic syntax and recursive techniques
3. Tuple unpacking
4. A SFINAE technique with variadic packs
5. Variadic expansion of expressions
6. Expansion of lambda blocks
7. Implementing a variant type

Templates

```
std::vector<int> vector_of_ints = { 1, 2, 3 };
std::vector<std::string> vector_of_strings = { "abc", "def" };
```

Templates

```
std::vector<int> vector_of_ints = { 1, 2, 3 };
std::vector<std::string> vector_of_strings = { "abc", "def" };

std::vector<bool> not_a_container = { true, false };
```

Templates

```
template<typename T>
const T & min(const T & t, const T & u)
{
    return t < u ? t : u;
}
```

Templates

```
template<typename T, typename U>
decltype(auto) min(T && t, U && u)
{
    return t < u ? std::forward<T>(t) : std::forward<U>(u);
}
```

Templates

```
template<typename T, typename U>
auto min(T t, U u)
{
    return t < u ? t : u;
}
```

The need for variable number of arguments

- What about getting a minimal value among more than two, all of (possibly) different types?

The need for variable number of arguments

- What about getting a minimal value among more than two, all of (possibly) different types?
- What about binding functions to their arguments (partial application)?

The need for variable number of arguments

- What about getting a minimal value among more than two, all of (possibly) different types?
- What about binding functions to their arguments (partial application)? (We do not speak of `std::bind1st` and `std::bind2nd`...)

The need for variable number of arguments

- What about getting a minimal value among more than two, all of (possibly) different types?
- What about binding functions to their arguments (partial application)? (We do not speak of `std::bind1st` and `std::bind2nd`...)
- What about types parametrized on an arbitrary about of types?

The need for variable number of arguments

- What about getting a minimal value among more than two, all of (possibly) different types?
- What about binding functions to their arguments (partial application)? (We do not speak of `std::bind1st` and `std::bind2nd`...)
- What about types parametrized on an arbitrary about of types? Concrete example: variant types.

Boost approach: preprocessor

Let's talk about `boost::variant`...

Boost approach: preprocessor

Let's talk about `boost::variant`... a poster child for people saying that templates are what's wrong with C++.

Boost approach: preprocessor

Let's talk about `boost::variant`... a poster child for people saying that templates are what's wrong with C++.

```
#define BOOST_VARIANT_AUX_DECLARE_PARAMS \
    BOOST_PP_ENUM( \
          BOOST_VARIANT_LIMIT_TYPES \
        , BOOST_VARIANT_AUX_DECLARE_PARAMS_IMPL \
        , T \
        ) \
    /**/

template < BOOST_VARIANT_AUX_DECLARE_PARAMS > class variant;
```

Boost approach: preprocessor

```
template < typename T0 = detail::variant::void_ , typename T1 = detail::va
    , typename T2 = detail::variant::void_ , typename T3 = detail::variant
    , typename T4 = detail::variant::void_ , typename T5 = detail::variant
    , typename T6 = detail::variant::void_ , typename T7 = detail::variant
    , typename T8 = detail::variant::void_ , typename T9 = detail::variant
    , typename T10 = detail::variant::void_ , typename T11 = detail::varia
    , typename T12 = detail::variant::void_ , typename T13 = detail::varia
    , typename T14 = detail::variant::void_ , typename T15 = detail::varia
    , typename T16 = detail::variant::void_ , typename T17 = detail::varia
    , typename T18 = detail::variant::void_ , typename T19 = detail::varia
> class variant;
```

Recursive approach

```
struct tail
{
};

template<std::size_t Index, typename T, typename Tail = tail>
struct type_list
{
};
```

Recursive approach

```
struct tail
{
};

template<std::size_t Index, typename T, typename Tail = tail>
struct type_list
{
};

using list = type_list<0, int, type_list<1, float>>;
```

Outline

1. Introduction to variadic templates
2. Variadic syntax and recursive techniques
3. Tuple unpacking
4. A SFINAE technique with variadic packs
5. Variadic expansion of expressions
6. Expansion of lambda blocks
7. Implementing a variant type

Syntax

```
template<typename... Ts>
auto min(Ts... ts);
```

Syntax

```
template<typename... Ts>
auto min(Ts... ts);
min(1, 2, 3);
min(1, 2.f, 3.0, 'a');
```

Syntax

```
template<typename... Ts>
auto min(Ts... ts);

min(1, 2, 3);
min(1, 2.f, 3.0, 'a');

template<typename... Ts>
class variant;
```

Syntax

```
template<typename... Ts>
auto min(Ts... ts);

min(1, 2, 3);
min(1, 2.f, 3.0, 'a');

template<typename... Ts>
class variant;

variant<int, float> v1;
variant<std::string, int, bool> v2;
```

Recursive unpacking

```
template<typename First, typename Second, typename... Tail>
auto min(First first, Second second, Tail... tail)
{
    return first < second ? min(first, tail...) : min(second, tail...);
}
```

Recursive unpacking

```
template<typename First, typename Second, typename... Tail>
auto min(First first, Second second, Tail... tail)
{
    return first < second ? min(first, tail...) : min(second, tail...);
}

template<typename Only>
auto min(Only only)
{
    return only;
}
```

Recursive unpacking

```
min(1, 2, 3, 4);
```

Recursive unpacking

```
min(1, 2, 3, 4);
```

```
auto min(int first, int second, int tail0, int tail1)
{
    return first < second ? min(first, tail0, tail1) : min(second, tail0, tail1);
}
```

Recursive unpacking

```
min(1, 2, 3, 4);
auto min(int first, int second, int tail0, int tail1)
{
    return first < second ? min(first, tail0, tail1) : min(second, tail0, tail1);
}
auto min(int first, int second, int tail0)
{
    return first < second ? min(first, tail0) : min(second, tail0);
}
```

Recursive unpacking

```
min(1, 2, 3, 4);
auto min(int first, int second, int tail0, int tail1)
{
    return first < second ? min(first, tail0, tail1) : min(second, tail0, tail1);
}
auto min(int first, int second, int tail0)
{
    return first < second ? min(first, tail0) : min(second, tail0);
}
auto min(int first, int second /* no tail - empty pack */)
{
    return first < second ? min(first) : min(second);
}
```

Outline

1. Introduction to variadic templates
2. Variadic syntax and recursive techniques
3. Tuple unpacking
4. A SFINAE technique with variadic packs
5. Variadic expansion of expressions
6. Expansion of lambda blocks
7. Implementing a variant type

Tuples

Tuple – a generic data structure containing values of several defined types.

Tuples

Tuple – a generic data structure containing values of several defined types.

```
template<typename... Ts>
class tuple;
```

Tuples

Tuple – a generic data structure containing values of several defined types.

```
template<typename... Ts>
class tuple;
```

```
std::tuple<int, float> t1 = { 1, 2.f };
auto t2 = std::make_tuple('a', true);
```

Tuples

Tuple – a generic data structure containing values of several defined types.

```
template<typename... Ts>
class tuple;

std::tuple<int, float> t1 = { 1, 2.f };
auto t2 = std::make_tuple('a', true);

auto v1 = std::get<0>(t1); // == 1
auto v2 = std::get<1>(t2); // == true
```

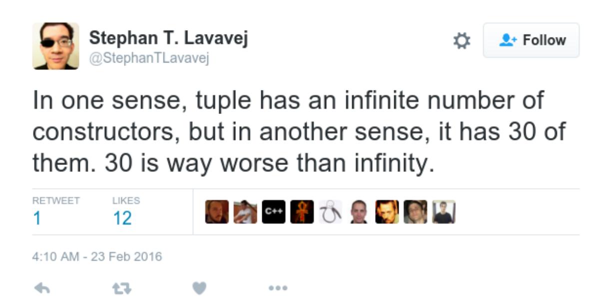
Tuples

`std::tuple` is also the bane of existence of the standard library developers.

Tuples

https://twitter.com/StephanTLavavej/status/702313387041038336

Tuples



https://twitter.com/StephanTLavavej/status/701967296978247680

Problem definition

- `std::tuple` used as generic storage.
- A function is passed in later on to be called with the stored arguments.

Problem definition

- `std::tuple` used as generic storage.
- A function is passed in later on to be called with the stored arguments.
- Easy to do in non-generic situations; harder with `std::tuple`, where `std::get` takes compile-time integers.

Problem definition

- `std::tuple` used as generic storage.
- A function is passed in later on to be called with the stored arguments.
- Easy to do in non-generic situations; harder with `std::tuple`, where `std::get` takes compile-time integers.
- Need to generate a list of consecutive integers from `0` to `sizeof...(Ts) - 1` at compile time.

std::make_integer_sequence

Generation of that integer list was so common that the committee deemed it useful to include a standard tool for that.

std::make_integer_sequence

Generation of that integer list was so common that the committee deemed it useful to include a standard tool for that.

```
template<typename T, T... Is>
struct integer_sequence;
```

std::make_integer_sequence

Generation of that integer list was so common that the committee deemed it useful to include a standard tool for that.

```
template<typename T, T... Is>
struct integer_sequence;

template<std::size_t... Is>
using index_sequence = integer_sequence<std::size_t, Is...>;
```

std::make_integer_sequence

Generation of that integer list was so common that the committee deemed it useful to include a standard tool for that.

```
template<typename T, T... Is>
struct integer_sequence;

template<std::size_t... Is>
using index_sequence = integer_sequence<std::size_t, Is...>;

auto sequence = std::make_index_sequence<3>();
// decltype(sequence) == std::index_sequence<0, 1, 2>
```

Tuple unpacking

```
template<typename... Ts>
auto do_something(std::tuple<Ts...> tuple)
{
    return do_something_impl(tuple, std::make_index_sequence<sizeof...(Ts)>());
}
```

Tuple unpacking

```
template<typename... Ts>
auto do_something(std::tuple<Ts...> tuple)
{
    return do_something_impl(tuple, std::make_index_sequence<sizeof...(Ts)>());
}

template<typename... Ts, std::size_t... Is>
auto do_something_impl(std::tuple<Ts...> tuple, std::index_sequence<Is...>)
{
    return function_to_call(std::get<Is>(tuple)...);
}
```

Outline

1. Introduction to variadic templates
2. Variadic syntax and recursive techniques
3. Tuple unpacking
4. A SFINAE technique with variadic packs
5. Variadic expansion of expressions
6. Expansion of lambda blocks
7. Implementing a variant type

SFINAE

SFINAE - Substitution Failure Is Not An Error

SFINAE

SFINAE - Substitution Failure Is Not An Error

```
template<bool B, typename T = void>
struct enable_if
{
    using type = T;
};
```

SFINAE

SFINAE - Substitution Failure Is Not An Error

```
template<bool B, typename T = void>
struct enable_if
{
    using type = T;
};

template<typename T>
struct enable_if<false, T>
{
};
```

SFINAE

```
template<typename T>
auto something(const T &)
    -> typename std::enable_if<interesting_trait<T>::value>::type;
```

SFINAE

```
template<typename T>
auto something(const T &)
    -> typename std::enable_if<interesting_trait<T>::value>::type;

template<typename T,
    typename std::enable_if<interesting_trait<T>::value, int>::type = 0>
auto something(const T &);
```

Variadic SFINAE

```
template<typename T,
    typename std::enable_if<interesting_trait<T>::value, int>::type...>
auto something(const T &);
```

Compiler bugs

Bug 11723 - Clang doesn't substitute into template parameter list of type template parameter pack if the pack is unused

Status: NEW

Product: clang
Component: C++11
Version: trunk
Platform: All All

Importance: P normal
Assigned To: Unassigned Clang Bugs

URL:
Keywords:

Depends on:
Blocks:

Reported: 2012-01-07 20:29 CST by Johannes Schaub
Modified: 2016-02-10 11:06 CST (History)
CC List: 11 users (show)

See Also:

https://llvm.org/bugs/show_bug.cgi?id=11723

Outline

1. Introduction to variadic templates
2. Variadic syntax and recursive techniques
3. Tuple unpacking
4. A SFINAE technique with variadic packs
5. Variadic expansion of expressions
6. Expansion of lambda blocks
7. Implementing a variant type

Contexts in which pack expansion is allowed

- expressions

Contexts in which pack expansion is allowed

- expressions
- list of base classes

Contexts in which pack expansion is allowed

- expressions
- list of base classes
- list of arguments (both template and function)

Contexts in which pack expansion is allowed

- expressions
- list of base classes
- list of arguments (both template and function)
- values or types passed as said arguments

Contexts in which pack expansion is not allowed

- declarations (to declare something for every element in a pack)

Contexts in which pack expansion is not allowed

- declarations (to declare something for every element in a pack)
- statements (to execute a statement for every element in a pack)

Problem definition

Objective: calling a function per every argument.

Problem definition

Objective: calling a function per every argument.

```
template<typename... Args>
void foo(Args... args)
{
    bar(args)...; // doesn't work!
    // "expected expression"
    // "expected ; before ..."
    // "pack not expanded"
}
```

An attempt

```
template<typename... Args>
void swallow(Args &&...)
{
}
```

An attempt

```
template<typename... Args>
void swallow(Args &&...)
{
}

template<typename... Args>
void foo(Args... args)
{
    swallow(bar(args)...);
}
```

An attempt

```
template<typename... Args>
void foo(Args &&... args)
{
    swallow((bar(std::forward<Args>(args)), 0)...);
}
```

An attempt

```
template<typename... Args>
void f(Args... args)
{
    swallow((std::cout << args << ' ', 0)...);
}

int main()
{
    f(1, 2, "abc");
}
```

An attempt

```
$ clang++ -std=c++11 main.cpp && ./a.out
1 2 abc
```

An attempt

```
$ clang++ -std=c++11 main.cpp && ./a.out
1 2 abc

$ g++ -std=c++11 main.cpp && ./a.out
abc 2 1
```

Helpers

```
struct unit
{
};
```

Helpers

```
struct unit
{
};

struct swallow
{
    template<typename... Args>
    swallow(Args &&...)
    {
    }
};
```

A proper attempt

```
template<typename... Args>
void f(Args... args)
{
    swallow{ (std::cout << args << ' ', unit{})... };
}

int main()
{
    f(1, 2, "abc");
}
```

A proper attempt

```
$ clang++ -std=c++11 main.cpp && ./a.out
1 2 abc
```

A proper attempt

```
$ clang++ -std=c++11 main.cpp && ./a.out
1 2 abc

$ g++ -std=c++11 main.cpp && ./a.out
1 2 abc
```

Outline

1. Introduction to variadic templates
2. Variadic syntax and recursive techniques
3. Tuple unpacking
4. A SFINAE technique with variadic packs
5. Variadic expansion of expressions
6. Expansion of lambda blocks
7. Implementing a variant type

Motivation

- Runtime dispatch in cases of type erasure is usually implemented in terms of virtual function calls.

Motivation

- Runtime dispatch in cases of type erasure is usually implemented in terms of virtual function calls.
- Virtual function calls usually mean two pointer accesses.

Motivation

- Runtime dispatch in cases of type erasure is usually implemented in terms of virtual function calls.
- Virtual function calls usually mean two pointer accesses.
- For erasure where the possible types are known at compile time, *we can do better!*

Basic idea

Lambdas are expressions.

Basic idea

Lambdas are expressions.

```
template<typename... Ts>
void print(variant<Ts...> v)
{
    using visitor_type = void (*)(variant<Ts...>);
    static visitor_type handlers[] = {
        [](variant<Ts...> v) {
            using T = Ts;
            std::cout << get<index_of<T, Ts...>::value>(v) << std::endl;
        }...
    };
    handlers[v.index()](std::move(v));
}
```

Compiler bugs, again

Bug 47226 - [C++0x] GCC doesn't expand template parameter pack that appears in a lambda-expression

Status: NEW

Alias: None

Product: gcc
Component: c++ (show other bugs)
Version: 4.6.0

Importance: P3 normal
Target Milestone: ---
Assignee: Not yet assigned to anyone

URL:
Keywords:

Depends on:
Blocks: 54367
Show dependency tree / graph

Reported: 2011-01-08 20:58 UTC by Johannes Schaub
Modified: 2015-11-27 22:19 UTC (History)
CC List: 9 users (show)

See Also:
Host:
Target:
Build:
Known to work:
Known to fail:
Last reconfirmed: 2013-05-21 00:00:00

`https://gcc.gnu.org/bugzilla/show_bug.cgi?id=47226`

Helpers

```
template<typename T>
struct id
{
    using type = T;
}
```

A workaround

```
template<typename... Ts>
void print(variant<Ts...> v)
{
    using visitor_type = void (*)(variant<Ts...>);
    auto generator = [](auto type) {
        using T = typename decltype(type)::type;
        return [](variant<Ts...> v) {
            std::cout << get<index_of<T, Ts...>::value>(v) << std::endl;
        };
    };
    static visitor_type handlers[] = { generator(id<Ts>())... };
    handlers[v.index()](std::move(v));
}
```

Outline

1. Introduction to variadic templates
2. Variadic syntax and recursive techniques
3. Tuple unpacking
4. A SFINAE technique with variadic packs
5. Variadic expansion of expressions
6. Expansion of lambda blocks
7. Implementing a variant type

std::aligned_storage

```
template<std::size_t Size,
    std::size_t Alignment = /* default alignment */>
struct aligned_storage
{
    using type = /* type of size Size, aligned with Alignment */;
};
```

std::aligned_storage

```
template<std::size_t Size,
    std::size_t Alignment = /* default alignment */>
struct aligned_storage
{
    using type = /* type of size Size, aligned with Alignment */;
};

template<std::size_t Size,
    std::size_t Alignment = /* default alignment */>
using aligned_storage_t = typename aligned_storage<Size, Alignment>::type;
```

Basics

```
template<typename... Ts>
class variant
{
    std::aligned_storage_t<
        max(sizeof(Ts)...),
        max(alignof(Ts)...)
    > storage;

    std::size_t tag;
};
```

Construction

```
template<typename T, typename std::enable_if<
    any_of<std::is_same<T, Ts>::value...>::value,
    int
>::type = 0>
variant(T t) : tag(index_of<T, Ts...>::value)
{
    new (&storage) T(std::move(t));
}
```

Copy construction

```
variant(const variant & other) : tag(other.tag)
{
    using visitor_type = void(*)(variant & self, const variant & other);
    auto generator = [](auto type) {
        using Arg = typename decltype(type)::type;
        return [](variant & self, const variant & other) {
            new (&self.storage) Arg(*reinterpret_cast<const Arg *>(&other.storage));
        };
    };
    static visitor_type copy_ctors[] = { generator(id<Ts>())... };
    copy_ctors[tag](*this, other);
}
```

Copy assignment

```
variant & operator=(const variant & other)
{
    using visitor_type = void (*)(variant & self, const variant & other);
    auto generator = [](auto type) {
        using T = typename decltype(type)::type;
        return [](variant & self, variant & other) {
            auto generator = [](auto type) {
                using Arg = typename decltype(type)::type;
                return [](variant & self, const variant & other) {
                    reinterpret_cast<T *>(&self.storage)->~T();
                    new (&self.storage) Arg(*reinterpret_cast<const Arg *>(&other.storage));
                    self.tag = other.tag;
                    return;
                };
            };
            static visitor_type assignment_helpers[] = { generator(id<Ts>())... };
            assignment_helpers[other.tag](self, other);
        };
    };
    static visitor_type copy_assignments[] = { generator(id<Ts>())... };
    copy_assignments[tag](*this, other);
    return *this;
}
```

Destruction

```
~variant()
{
    using dtor_type = void (*)(variant &);
    auto generator = [](auto type) {
        using Arg = typename decltype(type)::type;
        return [](variant & v) {
            reinterpret_cast<Arg *>(&v.storage)->~Arg();
        };
    };
    static dtor_type dtors[] = { generator(id<Args>())... };
    dtors[tag](*this);
}
```

Visitation

Note: the following code is a free function, friend with `variant`.

```
template<std::size_t N, typename... Ts>
const auto & get(const variant<Ts...> & variant)
{
    if (variant.tag != N)
    {
        throw invalid_variant_get(N, variant.tag);
    }

    return *reinterpret_cast<const nth<N, Ts...> *>(&variant.storage);
}
```

Visitation

```
template<typename... Ts, typename F>
auto fmap(const variant<Ts...> & var, F && f)
{
    using result_type = /* variant that can hold any of the return values */;
    using visitor_type = result_type (*)(const variant<Ts...> &, F &&);
    auto generator = [](auto type) {
        using T = typename decltype(type)::type;
        return [](const variant<Ts...> & v, F && f) -> result_type {
            return invoke(std::forward<F>(f), get<index_of<T, Ts...>::value>(v));
        };
    };
    static visitor_type visitors[] = { generator(id<Ts>())... };
    auto index = var.index();
    return visitors[index](var, std::forward<F>(f));
}
```

Links

- `https://github.com/griwes/reaverlib/blob/master/include/reaver/variant.h`
- `https://github.com/griwes/reaverlib/blob/master/tests/variant.cpp`