Italian C++ Community

++it

www.italiancpp.org

Vector++ 17

Davide Bianchi

Italian C++ Conference 2019
June 15, Milan

HOST

POLITECNICO MILANO

PATRON

Community Crumbs

SPONSORS

KDAB

JFrog | CONAN C/C++ package manager

JET BRAINS

develer

AIV ACCADEMIA ITALIANA VIDEOGIOCHI

sigeo Servizi Informativi Geografici

HEXAGON | Leica Geosystems

Why vectors?

1. Static size
2. No need for move semantic
3. Widely used
4. Well defined operations
5. Countless implementations

The basics

We want these...

```
int integers[3]
float floats[3]
```

...to behave like them...

```
int
float
double
```

...when used like that.

```
auto more_floats = integers * floats;
```

First needs

We need the compiler to write the following:

```
float more_floats[3];
more_floats[0] = integers[0] * floats[0];
more_floats[1] = integers[1] * floats[1];
more_floats[2] = integers[2] * floats[2];
```

```
decltype(integers[0] * floats[0])
```

```
((more_floats[Indices] = integers[Indices] * floats[Indices]), ...);
```

Index sequences and fold expressions

```
template<typename T, typename U, size_t... Indices>
auto multiply(T lhs[], U rhs[], std::index_sequence<Indices...>)
{
    decltype(std::declval<T>() * std::declval<U>()) result[sizeof...(Indices)];

    ((result[Indices] = lhs[Indices] * rhs[Indices]), ...);

    return result;
}
```

```
int integers[3] = { 1, 2, 3 };
float floats[3] = { 1.f, 2.f, 3.f };

auto result = multiply(integers, floats, std::make_index_sequence<3>{});
static_assert(std::is_same_v<decltype(result[0]), decltype(floats[0])>, ":D");
```

Cleaning up

We need to use the operator* instead of the multiply function...

...so we put everything inside a class

```
template<typename T, size_t... Indices>
class vector_
{
public:
    static constexpr size_t Size = sizeof...(Indices);

    vector_(const std::array<T, Size>& init_data) { ((data[Indices] = init_data[Indices]), ...); }

    T& operator[](size_t index) { return data[index]; }
    T operator[](size_t index) const { return data[index]; }

private:
    std::array<T, Size> data;
};
```

Update the multiply function...

```
template<typename T, typename U, size_t... Indices, typename Ret = decltype(std::declval<T>() * std::declval<U>())>
auto multiply(const vector_<T, Indices...>& lhs, const vector_<U, Indices...>& rhs) -> vector_<Ret, Indices...>
{
    return { {(lhs[Indices] * rhs[Indices])...} };
}
```

...that can now deduce the index list

And implement the operator*

```
template<typename T, typename U>
auto operator*(const T& lhs, const U& rhs) -> decltype(multiply(lhs, rhs))
{
    return multiply(lhs, rhs);
}
```

```
using vector3int = vector_<int, 0, 1, 2>;
using vector3float = vector_<float, 0, 1, 2>;

vector3int integers{ { 1, 2, 3 } };
vector3float floats{ {1.f, 2.f, 3.f} };

auto result = integers * floats;
static_assert(std::is_same_v<decltype(result), decltype(floats)>, ":D");
```

Swizzlers

We want swizzlers!

```
auto result = integers.zyx * floats.yyx;
```

We consider swizzlers to be a special type that can represent the data inside our vector under different shapes, and can «decay» to its representing vectory type.

```
union
{
    T data[3];

    struct
    {
        T x;
        T y;
        T z;
    };

    swizzler<0, 0> xx;
    swizzler<0, 1> xy;
    swizzler<0, 2> xz;
```

And swizzlers issues

```
template<typename T, typename U, size_t... Indices, typename Ret = decltype(std::declval<T>() * std::declval<U>())>
auto multiply(const vector_<T, Indices...>& lhs, const vector_<U, Indices...>& rhs) -> vector_<Ret, Indices...>
{
    return { {(lhs[Indices] * rhs[Indices])...} };
}
```

This can now come as a swizzler!

We need to add a decay step before using it.

Decay implementation

```
template<typename T>
struct is_vector_ : std::false_type
{};

template<typename T, size_t... Ns>
struct is_vector_<vector_<T, Ns...>> : std::true_type
{};

template<typename T, size_t N, size_t... Indices, size_t.
struct is_vector_<swizzler<vector_<T, Ns...>, T, N, Indic
{};

template<typename T>
struct is_vector : is_vector_<remove_cvref_t<T>>
{};
```

C++20

```
template <class T>
constexpr auto decay(const T& t)
{
    if constexpr (is_vector_v<T>)
    {
        return t.decay();
    }
    else
    {
        return t;
    }
}
```

Swizzlers support

Just by adding this...

```
template<typename T, typename U>
auto multiply(const T& lhs, const U& rhs) -> decltype(multiply(details::decay(lhs), details::decay(rhs)))
{
    return multiply(details::decay(lhs), details::decay(rhs));
}
```

...we can make them work

```
auto r1 = v3i * v3f.yyx;
auto r2 = v3i.zyz * v3f.yyx;
static_assert(std::is_same_v<decltype(r1), decltype(v3f)>, ":)");
static_assert(std::is_same_v<decltype(r2), decltype(v3f)>, ":)");
```

SIMD-ish

```
auto r1 = math::pow(v3f, v3i);     // vector3<float>
auto r2 = math::pow(v3f, 2);       // vector3<float>
auto r3 = math::pow(3.1415, v3i); // vector3<double>
auto r4 = math::pow(v3i, 5);       // also vector3<double>
```

How do we get these types?

```
auto error = math::pow(v3f, "foo");
```

no instance of function template "math::pow" matches the argument list
argument types are: (vec3f, const char [4])

How do we get something like this...

... Instead of something like these

C2182	'_Value': illegal use of type 'void'
C2182	'_Elems': illegal use of type 'void'
C2182	'[]': illegal use of type 'void'
C2661	'pow': no overloaded function takes 3 arguments
C2440	'initializing': cannot convert from 'initializer list' to 'math::vector_<void,0,1,2>'

Deduce the return type: order

- At least one argument is a vector
- All the vectors are of the same order

```
template<typename T, typename... Rest>
inline constexpr size_t get_order()
{
    if constexpr (sizeof...(Rest) == 0)
    {
        return get_size_v<T>;
    }
    else if constexpr (get_size_v<T> == 1)
    {
        return get_order<Rest...>();
    }
    else if constexpr (get_order<Rest...>() == 1 || get_size_v<T> == get_order<Rest...>())
    {
        return get_size_v<T>;
    }
    return 0;
}
```

```
:value) ?
```

Deduce the return type: scalar type

```
vector_<std::invoke_result_t<F, get_scalar_type_t<U>...>, Ns...>
```

```
std::make_index_sequence<get_order_v<U...>>
```

```
[](auto&&... args) { return std::pow(std::forward<decltype(args)>(args)...); }
```

vec_invoke

```
template<size_t Index, typename F, typename... ArgsT>
auto vec_invoke(F& aFunction, ArgsT&... someArgs)
{
    return std::invoke(aFunction, get_val<Index>(someArgs)...);
}

template<typename F, size_t... Ns, typename... U>
auto vec_invoke_impl(F& aFunction, std::index_sequence<Ns...>, U&&... aRHS)
{
    return vector_<std::invoke_result_t<F, get_scalar_type_t<U>...>, Ns...>{ { vec_invoke<Ns>(aFunction, aRHS...)... } };
}

template<typename F, typename... U>
auto vec_invoke(F&& aFunction, U&&... aRHS)
{
    return vec_invoke_impl(aFunction, std::make_index_sequence<get_order_v<U...>>{}, decay(aRHS)...);
}
```

Swizzlers...

Managing errors

```
template<typename BaseT, typename ExpT>
inline auto pow(const BaseT& base, const ExpT& exp)
{
    return details::vec_invoke([](auto _base, auto _exp){ return std::pow(_base, _exp); }, base, exp);
}
```

```
template<typename BaseT, typename ExpT>
inline auto pow(const BaseT& base, const ExpT& exp) -> details::vector_def_t<
    decltype(std::pow(std::declval<details::get_scalar_type_t<BaseT>>(), std::declval<details::get_scalar_type_t<ExpT>>())),
    details::get_order_v<BaseT, ExpT>>
{
    return details::vec_invoke([](auto _base, auto _exp){ return std::pow(_base, _exp); }, base, exp);
}
```

```
template<typename T, typename ListT>
struct vector_def;

template<typename T, size_t... Indices>
struct vector_def<T, std::index_sequence<Indices...>>
{
    using type = vector_<T, Indices...>;
};

template<typename T, size_t N>
using vector_def_t = std::enable_if_t<(N > 1), typename vector_def<T, std::make_index_sequence<N>>::type>;
```

Adding macros

```
template<typename... ArgsT>
inline auto pow(ArgsT&&... args) -> details::vector_def_t<
    decltype(std::pow(std::declval<details::get_scalar_type_t<ArgsT>>()...)),
    details::get_order_v<ArgsT...>>
{
    return details::vec_invoke([](auto... _args){ return std::pow(_args...); }, std::forward<ArgsT>(args)...);
}
```

```
#define DEFINE_VECTOR_FUNCTION(function_name, function_impl) \
template<typename... ArgsT> \
inline auto function_name(ArgsT&&... args) -> \
::math::details::vector_def_t< \
    decltype(function_impl(std::declval<::math::details::get_scalar_type_t<ArgsT>>()...)), \
    ::math::details::get_order_v<ArgsT...>> \
{ return ::math::details::vec_invoke( \
    [](auto... _args) { return function_impl(_args...); }, \
    std::forward<ArgsT>(args)...); }

#define DECLARE_VECTOR_STD_FUNCTION(function_name) DEFINE_VECTOR_FUNCTION(function_name, std::function_name)
```

Constructions & Conversions

Standard constructions

```
vec3i v3i0{ v3i };
vec3i v3i1{ v3i.xxy };
vec3i v3i2{ v3i.y, v3i.zz };
```

Implicit conversion

```
vec3d v3d{ v3f };
v3d = v3f;
```

```
vec3f v3f{ v3i };
v3f = v3i;
```

Allowing implicit conversion

```
template<typename... Args, class = std::enable_if_t<details::get_total_size_v<Args...> == Size && Size != 1>>
vector_(Args&&... args)
{
    size_t i = 0;
    ((construct(i, details::decay(std::forward<Args>(args)))), ...);
}
```

```
auto construct(size_t& i, scalar_type value)
{
    data[i++] = value;
}

template<typename U, size_t... OtherIndices>
void construct(size_t& i, const vector_<U, OtherIndices...>& value)
{
    ((data[i++] = value[OtherIndices]), ...);
}
```

```
vector_type& operator=(const vector_type& other)
{
    ((data[Indices] = other[Indices]), ...);
    return *this;
}
```

Questions?