High-Performance Cross-Platform Architecture:

C++20 Innovations

NOAH STEIN

Cppcon
The C++ Conference

2024
September 15 - 20

About Me

- 35-year career in video games and embedded software
- Started using C++ in 1995
- First cross-platform project in 1994

Cross-Platform Architecture Goals

- Take advantage of all platforms
- Focus on the compiler
- Minimize boilerplate and unnecessary code
- Minimize redundant code
- Minimize modifying existing code
- Minimize preprocessor macros

The Design

- Implement a family of quaternion classes, an illustrative example from a larger project
- Project build issues
- Inclusion of platform-specific header files
- Concept hierarchies
- Class and Function Design

OCP: The Open–Closed Principle in C++

- Open for extension
- Closed for modification
- Implemented via delegated polymorphism

> Since programs that conform to the open–closed principle are changed by adding new code, rather than by changing existing code, they do not experience the cascade of changes exhibited by non-conforming programs.
>
> — Robert C. Martin (1996)

Example: Drawing Shapes — The C Way Before OCP

Shape Data

```
typedef enum {
  circle,
  square
} shape_type;

typedef struct {
  shape_type type;
  int radius;
} circle_shape;

typedef struct {
  shape_type type;
  int side_length;
} square_shape;
```

Drawing Functions

```
void draw_circle(circle_shape *c);
void draw_square(square_shape *s);

void draw_shapes(shape** shapes, int n) {
  int i;
  for (i = 0;i < n; i++) {
    switch(shapes[i]->type) {
      case circle:

  draw_circle((circle_shape*)(shapes[i]));
        break;

      case square:

  draw_square((square_shape*)(shapes[i]));
        break;
    }
  }
}
```

Example: Drawing Shapes — The Polymorphic C++ Way

Shape Classes

```
// shape.h
class shape {
public:
  virtual void draw() const = 0;
};

// circle.h
class circle : public shape {
  int radius;
public:
  void draw() const override;
};

// square.h
class square : public shape {
  int side_length;
public:
  void draw() const override;
};
```

Draw Function

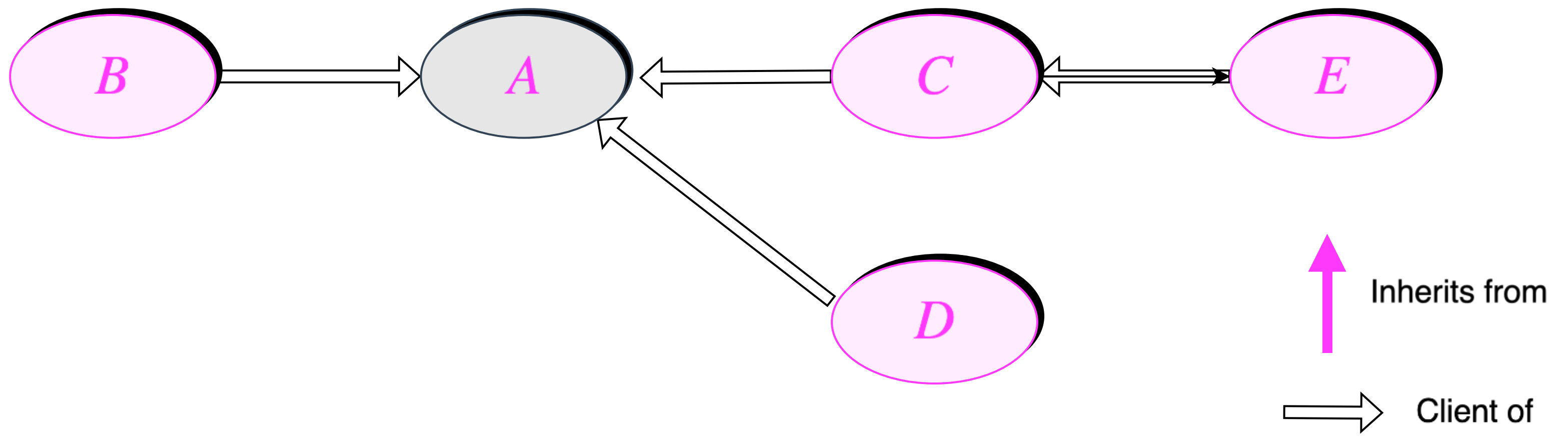
```
#include "shape.h"

void draw_shapes(std::vector<shape*> shapes) {
  for (auto const * s : shapes) {
    s->draw();
  }
}
```

OCP: The Original Open–Closed Principle

- If code has been made public, changes should not affect existing code.
- Re-use is through direct inheritance, not delegated polymorphism.
- Has been unrealistic in C++

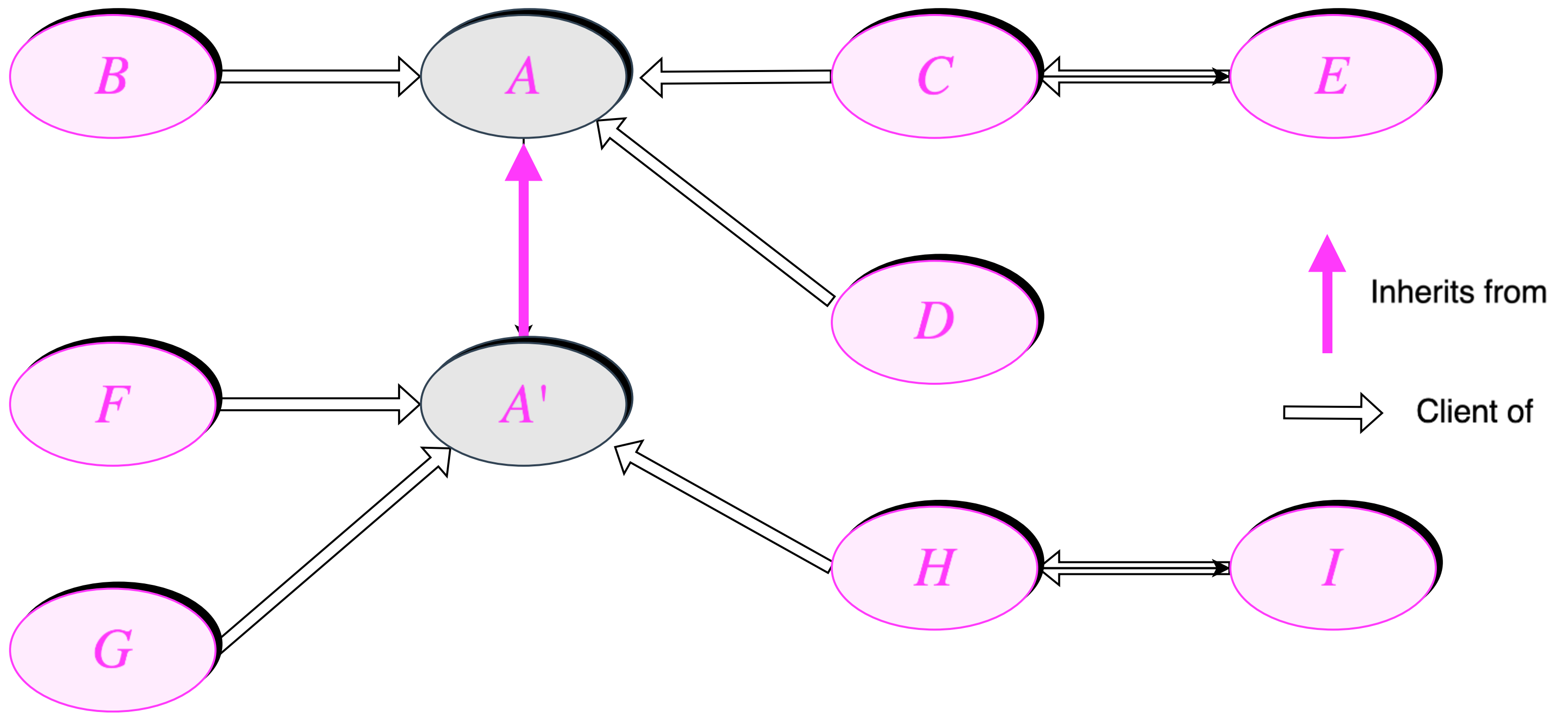
> The principles stated that a good module structure should be both closed and open:
>
> - Closed, because clients need the module's services to proceed with their own development, and once they have settled on a version of the module should not be affected by the introduction of new services they do not need.
> - Open, because there is no guarantee that we will include right from the start every service potentially useful to some client.
>
> — Bertrand Meyer (1988)

Meyer OPC: Adapting a Module to New Clients (Before)



Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.)

Meyer OPC: Adapting a Module to New Clients (After)



Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.)

OCP in C++: Strong and Weak Principles

- No OCP
  - Previous code modified
  - Change in current behavior and/or interface
  - Clients must change in response
- Weak OCP
  - Previous code modified
  - No change in current behavior and interface
  - Recompilation occurs
- Strong OCP
  - No previous code is modified in any way
  - Nothing is recompiled

Architecture Guidelines

- Very close to Bertrand Meyer’s original vision
- Not OOP, data and functions separate
- Adding new platforms has no effect on previously-implemented platforms
- Adding new revisions to a feature has no effect on previously-implemented revisions.

What is a Platform?

- A specific set of features
- A feature is an abstract unit of functionality requiring implementations that differ depending upon the target machine architecture.
- Features may be hardware: CPU architecture, SIMD instruction set, DMA controller, GPIO module, etc.
- Features may be software: OS, graphics API, etc.
- Features may not be totally orthogonal, e.g. x86/SSE, DirectX/Windows

Directory and File Structure

Flat

```
plt/simd
  Simd.h
  Neon32.h
  Sse.h
  Sse2.h
  …

plt/math
  Quat.h
  Quat_Common.h
  Quat_Neon32.h
  Quat_Sse.h
  Quat_Sse2.h
  …
```

Deep

```
plt/simd
  Simd.h
  Neon32.h
  Sse.h
  Sse2.h
  …

plt/math
  Quat.h
  Common/
    Quat_Common.h
    Vec_Common.h
    Mtx_Common.h
    …

  Neon32/
    Quat_Neon32.h
    Vec_Neon32.h
    Mtx_Neon32.h
    …
```

Including Headers

- Each feature has its own header file to define the common definitions
- The header file is responsible for including the platform-specific code
- A series of preprocessor macros handles generating the header file name to load
- Example:
  ```
      INCLUDE_SIMD(Quat)
  ```
  becomes:
  ```
      ”Quat_SSE2.h”
  ```

Header Inclusion Macros

- `#define INCLUDE_PLT(Feature, File) INCLUDE_BUILD_FILENAME(Feature, File)`
- `#define INCLUDE_PLT_FEATURE(Feature) INCLUDE_STRINGIZE(Feature.h)`
- `#define INCLUDE_BUILD_FILENAME(Feature, File) INCLUDE_STRINGIZE(File ## _ ## Feature.h)`
- `#define INCLUDE_STRINGIZE(String) #String`

- `#define INCLUDE_SIMD(File) INCLUDE_PLT(PLT_SIMD, File)`

Creating a Feature

- Define the feature in the build system
- Create a header file for the feature
- Create one header file for each unique implementation

The SIMD Feature in the Build System

- Place appropriate feature macro definition in toolchain files:

```
set(PLT_SIMD Common)
```

- In the common project, add the features to the preprocessor definitions:

```
add_compile_definitions(
      PLT_SIMD=${PLT_SIMD}
)
```

Simd.h: The SIMD Feature Header

```
#if !defined(PLT_SIMD)
  #error You must define PLT_SIMD.
#endif


#define INCLUDE_SIMD(File) INCLUDE_PLT(PLT_SIMD, File)


#include INCLUDE_PLT_LOCAL(PLT_SIMD))
```

Simd_Common.h: The SIMD Feature Header

```
namespace plt::simd
{
  struct Common {};
}
```

Quaternions, Mathematically Speaking (1/2)

- Four-dimensional complex number:

$$w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

- Where:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$

$$\mathbf{ij} = \mathbf{k} = -\mathbf{ji}$$

$$\mathbf{jk} = \mathbf{i} = -\mathbf{kj}$$

$$\mathbf{ki} = \mathbf{j} = -\mathbf{ik}$$

- Addition:

$$a + b = (a_w + b_w) + (a_x + b_x)\mathbf{i} + (a_y + b_y)\mathbf{j} + (a_z + b_z)\mathbf{k}$$

- Multiplication:

$$\begin{aligned} ab = &\ (a_w b_w - a_x b_x - a_y b_y - a_z b_z) \\ &+ (a_w b_x + a_x b_w + a_y b_z - a_z b_y)\mathbf{i} \\ &+ (a_w b_y - a_x b_z + a_y b_w + a_z b_x)\mathbf{j} \\ &+ (a_w b_z + a_x b_y - a_y b_x + a_z b_w)\mathbf{k} \end{aligned}$$

Quaternions, Mathematically Speaking (2/2)

- Conjugate:

$$\mathrm{q}^* = w - x\mathbf{i} - y\mathbf{j} - z\mathbf{k}$$

- Dot Product:

$$a \cdot b = a_w b_w + a_x b_x + a_y b_y + a_z b_z$$

- Norm:

$$\|q\| = \sqrt{q \cdot q} = \sqrt{w^2 + x^2 + y^2 + z^2}$$

- Multiplicative Inverse:

$$q^{-1} = \frac{q^*}{\|q\|^2} = \frac{q^*}{q \cdot q}$$

- Division:

$$\frac{a}{b} = ab^1 = \frac{ab^*}{b \cdot b}$$

Quaternion Concepts in Mathematics vs. C++ 20

- Mathematically, a quaternion is defined by its data and operations
- As a C++ concept, the quaternion is defined purely by its data
- The common implementation of operations rely only upon the concepts
- Optimized implementations conform to the concepts and common implementations

C++ Quaternion Concept

```cpp
template<typename Q>
concept Quaternion = requires(Q q)
{
  typename Q::Scalar;
  Arithmetic<typename Q::Scalar>;

  { q.w() } -> std::same_as<typename Q::Scalar>;
  { q.x() } -> std::same_as<typename Q::Scalar>;
  { q.y() } -> std::same_as<typename Q::Scalar>;
  { q.z() } -> std::same_as<typename Q::Scalar>;
};
```

Supporting Concepts

Arithmetic

```
template<typename T>
concept Arithmetic = std::is_arithmetic_v<T>;
```

MutuallyArithmetic

```
template<typename T, typename U>
concept MutuallyArithmetic = requires (T t, U u)
{
  requires Arithmetic<T>;
  requires Arithmetic<U>;

  { t + u };
  { t - u };
  { t * u };
  { t / u };
};
```

“Standard” Quaternion Type: Declaration and Data

```
template<typename S, typename I = plt::simd::PLT_SIMD>
class Quat
{
public:
  using Scalar = S;

private:
  Scalar w_, x_, y_, z_;

  // ... Constructors
  // ... Accessors
}
```

“Standard” Quaternion Type: Constructors

```
Quat() = default;

Quat(Scalar w, Scalar x, Scalar y, Scalar z)
  noexcept(std::is_nothrow_copy_constructible_v<Scalar>)
: w_(w), x_(x), y_(y), z_(z)
{}

Quat(Scalar && w, Scalar && x, Scalar && y, Scalar && z)
  noexcept(std::is_nothrow_move_constructible_v<Scalar>)
: w_(move(w)), x_(move(x)), y_(move(y)), z_(move(z))
{}

template<Quaternion Q>
  requires std::convertible_to<typename Q::Scalar, Scalar>
Quat(const Q& rhs)
  noexcept(std::is_nothrow_convertible_v<typename Q::Scalar, Scalar>)
: Quat(Scalar{rhs.w()}, Scalar{rhs.x()}, Scalar{rhs.y()}, Scalar{rhs.z()})
{}
```

“Standard” Quaternion Type: Assignment Operator

```
template<Quaternion Q>
  requires std::convertible_to<typename Q::Scalar, Scalar>
Quat& operator=(const Q& rhs)
  noexcept(std::is_nothrow_convertible_v<typename Q::Scalar, Scalar>)
{
  w_ = Scalar{rhs.w()};
  x_ = Scalar{rhs.x()};
  y_ = Scalar{rhs.y()};
  z_ = Scalar{rhs.z()};
  return *this;
}
```

“Standard” Quaternion Type: Accessors

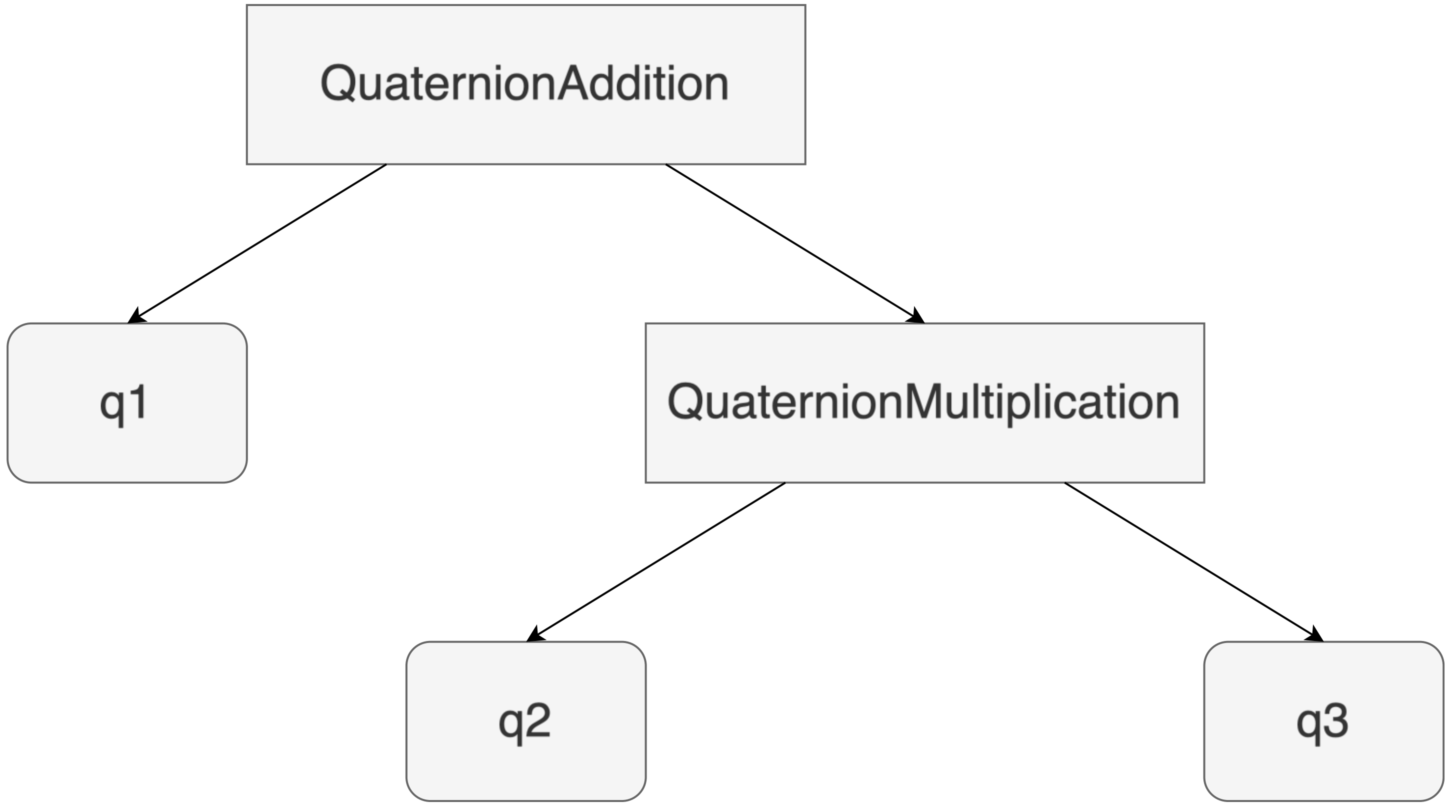
```
const Scalar & w() const { return w_; } noexcept
const Scalar & x() const { return x_; } noexcept
const Scalar & y() const { return y_; } noexcept
const Scalar & z() const { return z_; } noexcept
```

General Operation Implementation

- Use expression trees
- Runs anywhere
- Works for any arithmetic scalar type
- Parameters defined using concepts

What is an Expression Tree? An Example

- Expression: q1+q2*q3
- Operators construct nodes in the tree
- Expression results in a tree:



Addition Example: Operator

```
template<Quaternion QL, Quaternion QR>
inline auto operator+(const QL & lhs, const QR & rhs) noexcept -> QuaternionAddition<QL, QR>
{
  return QuaternionAddition(lhs, rhs);
}
```

Addition Example: Quaternion Binary Expression

```
template<Quaternion QL, Quaternion QR>
  requires MutuallyArithmetic<typename QL::Scalar, typename QR::Scalar>
class QuaternionBinaryExpr : public QuaternionExpr
{
  using SL = typename QL::Scalar;
  using SR = typename QR::Scalar;

public:
  using Scalar = typename std::common_type<SL, SR>::type;
};

template<typename QL, typename QR>
using QuaternionBinaryType = typename QuaternionBinaryExpr<QL, QR>::Scalar;
```

Addition Example: Expression Node

```
template<Quaternion QL, Quaternion QR>
class QuaternionAddition : public QuaternionBinaryExpr<QL, QR>
{
  QL l_;
  QR r_;

public:
  using Scalar = QuaternionBinaryType<QL, QR>;

  QuaternionAddition(const QL & lhs, const QR & rhs) noexcept
  : l_(lhs), r_(rhs)
  {}

  Scalar w() const noexcept { return l_.w() + r_.w(); }
  Scalar x() const noexcept { return l_.x() + r_.x(); }
  Scalar y() const noexcept { return l_.y() + r_.y(); }
  Scalar z() const noexcept { return l_.z() + r_.z(); }
};
```

Scalar-valued Functions

```
template<Quaternion QL, Quaternion QR>
inline auto Dot(const QL & lhs, const QR & rhs) noexcept -> QuaternionBinaryType<QL, QR>
{
  return
    lhs.w() * rhs.w() +
    lhs.x() * rhs.x() +
    lhs.y() * rhs.y() +
    lhs.z() * rhs.z();
}
```

Usage of Common Implementation

```
Quat<float> eval(Quat<float> a, Quat<float> b, Quat<float> c) {
  return (a * b + c) / (a - c);
}
```

Common Implementation Complete

- The common implementation is a set of interrelated concepts, classes and functions.
- Quaternion-valued expressions will compile and run on any platform as long as the scalar types are supported on the platform
- The code is as optimal as the compiler that optimizes it.
- But we can do better…

Optimizing for the ARM Neon

- Update the build system
- Add a SIMD tag
- Add a template specialization for the Neon data
- Add Neon-optimized functions

Updating the Build System for ARM Neon32

- Change the SIMD macro to Neon32

```
    set(PLT_SIMD Neon32)
```

- May need to set a compiler flag to enable Neon

neon32.h: The ARM Neon32 Feature Header

```
#define SIMD_HAS_NEON32

#include <type_traits>
#include ”simd.h"

namespace plt::simd
{
  struct Neon32 : Common {};

  template<typename SIMD>
  concept Neon32Family = std::derived_from<SIMD, Neon32>;
}
```

Quat_Neon32.h: Quat<float, Neon> Class Declaration and Data

```
#include <arm_neon.h>

template<>
class Quat<float, plt::simd::Neon32>
{
  float32x4_t value_;

public:
  using Scalar = float;

  // ... Constructors
  // ... Accessors
};
```

Quat_Neon32.h: Quat<float, Neon> Constructors

```
Quat() = default;

Quat(Scalar w, Scalar x, Scalar y, Scalar z) {
  float vals[] = { w, x, y, z};
  value_ = vld1q_f32(vals);
}

template<Quaternion Q>
Quat(const Q& rhs)
  : Quat(static_cast<Scalar>(rhs.w()), static_cast<Scalar>(rhs.x()),
static_cast<Scalar>(rhs.y()),
  static_cast<Scalar>(rhs.z()))
{}

Quat(float32x4_t value) : value_(value) {}
```

Quat_Neon32.h: Quat<float, Neon> Accessors

```
template<>
class Quat<float, plt::simd::Neon32>
{
  // ...
  Scalar w() const noexcept { return vgetq_lane_f32(NeonVal(), 0); }
  Scalar x() const noexcept { return vgetq_lane_f32(NeonVal(), 1); }
  Scalar y() const noexcept { return vgetq_lane_f32(NeonVal(), 2); }
  Scalar z() const noexcept { return vgetq_lane_f32(NeonVal(), 3); }

  float32x4_t NeonVal() const noexcept { return value_; }
};
```

Where are We Now?

- The class alone compiles, links, and passes tests
- Quaternions will now use Neon registers
- The algorithms are all common implementations
- Data is moved into general-purpose registers for computations
- Depending on the platform, may see a performance gain at this stage

Quat<float, Neon32> Functions

```
template<plt::simd::Neon32Family SIMD>
inline auto operator-(Quat<float, SIMD> q) -> Quat<float, SIMD>
{
  float32x4_t value = q.NeonVal();
  float32x4_t negation = (-q).NeonVal();
  float32x4_t result = vcopyq_laneq_f32(negation, 0, value, 0);
  return Quat<float, SIMD>(result);
}
```

Usage of Platform-specific Implementation

```
Quat<float> eval(Quat<float> a, Quat<float> b, Quat<float> c) {
  return (a * b + c) / (a - c);
}
```

Adding a Platform with SSE

- Update the build scripts
- Add a SIMD tag
- Add a template specialization for the SSE data
- Add SSE-optimized functions
- SSE has 128-bit registers, including 4x floats, excluding 2x doubles

Updating the Build System for SSE

- Create a new toolchain file
- Set the SIMD feature macro:
  ```
  set(PLT_SIMD SSE)
  ```
- No change is required to the other tool chain file or the common build files

sse.h: The SSE Feature Header

```
#define SIMD_HAS_SSE

#include <type_traits>
#include ”simd.h"

namespace plt::simd
{
  struct Sse : Common {};

  template<typename SIMD>
  concept SseFamily = std::derived_from<SIMD, Sse>;
}
```

Quat_Sse.h: Quat<float, Sse> Declaration and Constructors

```
#include <immintrin.h>

template<>
class Quat<float, plt::simd::Sse>
{
  __m128 value_;


public:
  using Scalar = float;

  // ...

  Quat(Scalar w, Scalar x, Scalar y, Scalar z) {
    value_ = _mm_setr_ps(w, x, y, z);
  }

  Quat(__m128_t value) : value_(value) {}
  // ...
};
```

Quat_Sse.h: Quat<float, sse> Accessors

```
template<>
class Quat<float, plt::simd::Neon32>
{
  // ...
  Scalar w() const noexcept { return _mm_cvtss_f32(SseVal()); }

  Scalar x() const noexcept { return _mm_cvtss_f32(_mm_shuffle_ps(SseVal(), SseVal(),
    _MM_SHUFFLE(1, 1, 1, 1))); }

  Scalar y() const noexcept { return _mm_cvtss_f32(_mm_shuffle_ps(SseVal(), SseVal(),
    _MM_SHUFFLE(2, 2, 2, 2))); }

  Scalar z() const noexcept { return _mm_cvtss_f32(_mm_shuffle_ps(SseVal(), SseVal(),
    _MM_SHUFFLE(3, 3, 3, 3))); }

  __m128 SseVal() const noexcept { return value_; }
};
```

Quat_Sse Functions

```
template<plt::simd::Sse SIMD>
inline auto Dot(Quat<float, SIMD> lhs, Quat<float, SIMD> rhs) -> float
{
  __m128 squares = _mm_mul_ps(lhs.SseVal(), rhs.SseVal());
  __m128 badc = _mm_shuffle_ps(squares, squares, _MM_SHUFFLE(2, 3, 0, 1));
  __m128 pairs = _mm_add_ps(squares, badc);
  __m128 bbaa = _mm_shuffle_ps(pairs, pairs, _MM_SHUFFLE(0, 1, 2, 3));
  __m128 dp = _mm_add_ps(pairs, bbaa);
  float result = _mm_cvtss_f32(dp);
  return result;
}
```

Two Platforms Now

- For this platform, as with Neon, we now build using SSE registers
- We now have two platforms whose optimizations are completely independent
- Except for include guards and checking PLT_SIMD is defined, there are no #if or #ifdef macros anywhere
- Each header file unconditionally includes the appropriate intrinsic header
- No feature-specific code is ever fed into the compiler for a platform that does not have that version of the feature

Handling Feature Revisions

- With data class and functions written, we have a program optimized for SSE.
- Now we need to support a platform with SSE2
- SSE2 adds support for a 128-bit register with two doubles
- Now we have two optimized types Quat<float, Sse2> and Quat<double, Sse2>
- Same steps, different hoops

Sse2.h: The SSE Feature Header

```
#define SIMD_HAS_SSE2

#include ”Sse.h"

namespace plt::simd
{
  struct Sse2 : Sse {};

  template<typename SIMD>
  concept Sse2Family = SseFamily<SIMD> && std::derived_from<SIMD, Sse2>;
}
```

Quat_Sse2.h: Quat<float, Sse2> Class

```
#include "Quat_Sse.h"

template<>
class Quat<float, plt::simd::Sse2> : public Quat<float, plt::simd::Sse>
{
  using Quat<float, plt::simd::Sse>::Quat;
};
```

Quat_Sse2.h: Quat<double, Sse2> Class Declaration and Data

```
template<>
class Quat<double, plt::simd::Sse2>
{
  __m128d wx_;
  __m128d yz_;

public:
  using Scalar = double;

  // ... Constructors
  // ... Accessors
}
```

Quat_Sse2.h: Quat<double, Sse2> Class Constructors

```
Quat() = default;

Quat(Scalar w, Scalar x, Scalar y, Scalar z)
{
  wx_ = _mm_set_pd(x, w);
  yz_ = _mm_set_pd(z, y);
}

template<Quaternion Q>
Quat(const Q& rhs)
: Quat(static_cast<Scalar>(rhs.w()), static_cast<Scalar>(rhs.x()), static_cast<Scalar>(rhs.y()),
  static_cast<Scalar>(rhs.z()))
{}

Quat(__m128d wx, __m128d yz)
: wx_(wx), yz_(yz)
{}
```

Quat_Sse2.h: Quat<double, Sse2> Accessors

```
Scalar w() const { return _mm_cvtsd_f64(SseWx()); }
Scalar x() const { return _mm_cvtsd_f64(_mm_unpackhi_pd(SseWx(), SseWx())); }
Scalar y() const { return _mm_cvtsd_f64 (SseYz()); }
Scalar z() const { return _mm_cvtsd_f64(_mm_unpackhi_pd(SseYz(), SseYz())); }

__m128d SseWx() const { return wx_; }
__m128d SseYz() const { return yz_; }
```

Quat<double, Sse2> Functions

```
template<plt::simd::Sse2 SIMD>
inline auto Dot(Quat<double, SIMD> lhs, Quat<double, SIMD> rhs) -> double
{
  __m128d w2x2 = _mm_mul_pd(lhs.SseWx(), rhs.SseWx());
  __m128d x2w2 = _mm_shuffle_pd(w2x2, w2x2, _MM_SHUFFLE2(0, 1));
  __m128d wx2wx2 = _mm_add_pd(w2x2, x2w2);

  __m128d y2z2 = _mm_mul_pd(lhs.SseYz(), rhs.SseYz());
  __m128d z2y2 = _mm_shuffle_pd(y2z2, y2z2, _MM_SHUFFLE2(0, 1));
  __m128d yz2yz2 = _mm_add_pd(y2z2, z2y2);

  __m128d dp = _mm_add_pd(wx2wx2, yz2yz2);
  double result = _mm_cvtsd_f64(dp);
  return result;
}
```

On to SSE3

- Add another toolchain file for the new platform that has SSE3
- Add another SIMD tag for SSE3 inheriting from SSE2
- No new register formats, so the float and double Quat specializations get pulled forward to Quat<float, Sse3> and Quat<double, Sse3>
- New instructions mean more-optimized implementations

Quat<float, Sse3> Functions

```
template<plt::simd::Sse3 SIMD>
inline auto Dot(Quat<float, SIMD> lhs, Quat<float, SIMD> rhs) -> float
{
  __m128 squares = _mm_mul_ps(lhs.SseVal(), rhs.SseVal()); // w^2, x^2, y^2, z^2
  __m128 add1st = _mm_hadd_ps(squares, squares); // w^2+x^2, y^2+z^2, w^2+x^2, y^2+z^2
  __m128 add2nd = _mm_hadd_ps(add1st, add1st); // w^2+x^2+y^2+z^2, ...
  float result = _mm_cvtss_f32(add2nd);
  return result;
}

template<plt::simd::Sse3 SIMD>
inline auto Dot(Quat<double, SIMD> lhs, Quat<double, SIMD> rhs) -> double
{
  __m128d w2x2 = _mm_mul_pd(lhs.SseWx(), rhs.SseWx()); // w^2, x^2
  __m128d y2z2 = _mm_mul_pd(lhs.SseYz(), rhs.SseYz()); // y^2, z^2
  __m128d add1 = _mm_hadd_pd(w2x2, y2z2); // w^2+x^2, y^2+z^2
  __m128d add2 = _mm_hadd_pd(add1, add1); // w^2+x^2+y^2+z^2, ...

  float result = _mm_cvtsd_f64(add2);
  return result;
}
```

Function Overload Resolution: The Outline

- Overload resolution is the determination of which function to call given a set of functions with the same name
- The compiler generates a list of candidates, trims it down to viable ones, and then picks the best match
- In the event there is no best match the compiler emits an ambiguous overload error
- With concepts, what constitutes the best match?

Concepts, Overload Resolution, and Subsumption

- Like enable_if<>, concepts constrain viability
- Unlike enable_if<>, concepts also partake in determining the best match
- For classes, a derived class has a higher priority than its base class
- Concepts do not derive from other concepts; however, other concepts may appear in the body of concept definition, analogous to the idea concepts may be composed but not inherited
- For concepts, a concept that subsumes another concept has the higher priority
- What is subsumption?

Subsumption in Code

- Sse2Family is defined as conforming to the sse_family concept and the additional constraint of SIMD deriving from Sse2. Thus, Sse2Family subsumes SseFamily
- When Sse2Derived is true, SseFamily is also true (but not necessarily the reverse); however, it does not subsume SseFamily

```
struct Sse : Common {};
struct Sse2 : Sse {};

template<typename SIMD>
concept SseFamily = std::derived_from<SIMD, Sse>;

template<typename SIMD>
concept Sse2Family = SseFamily<SIMD> &&
std::derived_from<SIMD, Sse2>;

template<typename SIMD>
concept Sse2Derived = std::derived_from<SIMD, Sse2>;
```

Feature Revision Without Subsumption

- The first multiplication function is defined in SSE
- A more-optimized one is implemented in SSE3
- Assume Sse3Family is defined only as a tag derived from Sse3
- This will fail to compile with an ambiguous overload error
- The valid parameter types of Sse3Family is a strict subset of those acceptable by SseFamily
- But the concepts are unrelated and thus ambiguous

```
template<typename SIMD>
concept Sse3Family = std::derived_from<SIMD, Sse3>;

template<plt::simd::SseFamily SIMD>
inline auto operator*(Quat<float, SIMD> lhs,
Quat<float, SIMD> rhs) -> Quat<float, SIMD>;

template<plt::simd::Sse3Family SIMD>
inline auto operator*(Quat<float, SIMD> lhs,
Quat<float, SIMD> rhs) -> Quat<float, SIMD>;
```

Feature Revision With Subsumption

- The Sse3Family concept refers to Sse2Family and thus subsumes it
- And the Sse2Family concept refers to SseFamily and thus subsumes it
- Subsumption is transitive thus Sse3Family subsumes SseFamily
- Consequently, the multiply constrained by Sse3Family is a better match than that constrained by SseFamily, and there is no ambiguity

```
template<typename SIMD>
concept Sse2Family = SseFamily<SIMD> &&
std::derived_from<SIMD, Sse3>;

template<typename SIMD>
concept Sse3Family = Sse2Family<SIMD> &&
std::derived_from<SIMD, Sse3>;

template<plt::simd::SseFamily SIMD>
inline auto operator*(Quat<float, SIMD> lhs,
Quat<float, SIMD> rhs) -> Quat<float, SIMD>;

template<plt::simd::Sse3Family SIMD>
inline auto operator*(Quat<float, SIMD> lhs,
Quat<float, SIMD> rhs) -> Quat<float, SIMD>;
```

The State of Affairs with SSE3

- The data definition of a Quat<float> is defined for SSE and then re-used for SSE2 and SSE3 with minimal boilerplate.
- A full set of arithmetic operations are implemented in SSE
- SSE2 inherits the optimized SSE functions without a single line of code
- SSE3 implements two more-optimized functions and inherits the rest
- When adding a new revision, all platforms still on older revision do not see any new code and build in exactly the same way as if no code was added

Fast Forward to AVX

- AVX is the next Intel SIMD standard after SSE4
- SSE uses 128-bit registers
- AVX uses 256-bit registers
- Quat<float> cannot take advantage of the wider register format, so it is pulled forward as usual
- Quat<double> can now fit the entire quaternion into a single register, so it gets a new data type

Quat<double, Avx> Class Data

- It now is quite like the original Quat<float, SSE> with “double” replacing “float” and “__m256d” replacing “__m128f”
- If your class and math functions are in the same header, compiling the full test suite now will result in a mass of failures
- The Avx tag derives from the SSE4 tag, so all the functions that operate on two __m128f members fails because we now have a single __m256 member
- What to do?

```
template<> class Quat<double, plt::simd::Avx>
{
   __m256d value_;

public:
   // ...
}
```

Dividing Code Into Separate Headers

- Separating the class and functions has been described as good for recompiles
- Additionally, it can avoid the problem of incompatible data across revisions
- Thus, instead of merely having Quat.h, Quat_Sse.h, etc., the project should also have QuatMath.h, QuatMath_Sse.h, etc.
- But this still would not avoid the compilation errors!

The Root Cause of the Errors

- The initial split is necessary but not sufficient to fix the errors
- The problem is that Quat<double, Avx> should not include the prior definitions
- But if QuatMath_Avx.h does not include QuatMath_Sse4.h then the Quat<float, Avx> functions won’t get included and all of a sudden Quat<float> is no longer optimized
- Thus, separate them into their own headers

Separate Math Headers AVX Example

- The math header merely includes headers for each supported scalar type
- QuatFloatMath_Avx.h includes QuatFloatMath_Sse4.h to inherit operations
- QuatDoubleMath_Avx.h would not include a prior header because it is incompatible with prior revisions

```
#include "QuatFloatMath_Avx.h"
#include "QuatDoubleMath_Avx.h"
```

How Could I Know the Structure From Day 1?

- You probably couldn’t
- Given how large code bases can be, I’d refactor when it became known
- Such a refactor would trigger a change in files seen by older platforms
- Thus, it’s a weak OCP that is nearly strong

Summary

- We can write optimized code for each platform
- We can isolate platforms
- We can minimize redundancy
- With C++ 20, concepts, and some discipline we can have it all at once

- https://github.com/noahstein/Ark