The Point Challenge: Returning Different Types for the Same Operation

Amir Kirsh
Strong Types

Distance distance(Duration t, Speed s) {
  return t * s; // distance = time * speed
}

// usage:
auto d = distance(1.5h, 10_kmh); // 1.5h = 1.5 hours (std::chrono duration)
// 10_kmh = 10 km/h (with our own literal type)

// following would fail to compile:
auto d1 = distance(1h, 1h);
auto d2 = distance(10_km, 10_kmh);
Strong Types - not every operation is allowed!

// future / historical point in time - ok
Time operator+(Time p, Duration t);
Time operator+(Duration t, Time t);

// Duration between points in time - ok
Duration operator-(Time p1, Time p2);

// adding or subtracting durations - ok
Duration operator+(Duration d1, Duration d2);
Duration operator-(Duration d1, Duration d2);

// But the following should NOT be allowed:
operator+(Time p1, Time p2); // 12:00pm + 9:00am = ??
Point Arithmetics

Point p1 {3, 7};  // assume proper ctor
Point p2 {10, 10};
Point p3 = p1 + p2;

Should we allow adding two points?
Point Arithmetics

Point p1 {3, 7};  // assume proper ctor
Point p2 {10, 10};
Point p3 = p1 + p2;

Should we allow adding two points?
What does {13, 17} represent?

Maybe there is a point in adding Point and PointDiff...
But two points?
Point Arithmetics

Point p1 {3, 7}; // assume proper ctor
Point p2 {10, 10};

Point p3 = p1 + p2;

On the other hand maybe there is a need…
Point Arithmetics

Point p1 {3, 7}; // assume proper ctor
Point p2 {10, 10};

Point p3 = p1 + p2;

On the other hand maybe there is a need...

Averaging:

Point middle = (p1 + p2) / 2;
Adding two Points

auto twoPoints = p1 + p2;

Then, averaging:

Point middle = twoPoints / 2;
Adding two Points

```cpp
auto twoPoints = p1 + p2; // what should be the type of `twoPoints`?
```

Then, averaging:

```cpp
Point middle = twoPoints / 2;
```
Multiplying by N

auto twoPoints = p1 * 2;

Should we allow that?
Multiplying by N

// p3 should be closer to p2 (in ratio ⅓ <= ⅔):

Point p3 = (p1 + p2 * 2) / 3;

// however, (p2 * 2) shouldn’t be a point!
Dividing by N

// p3 should be closer to p2 (in ratio \( \frac{1}{3} \) <= \( \frac{2}{3} \)):

Point p3 = p1 / 3 + 2 * p2 / 3;

// however, (p1 / 3) and (2 * p2 / 3) shouldn’t be points!
The Challenge

**Allow:**

- Adding Points (2 points, 3 points, N points)
- Multiplying and Dividing by any number
- Result cannot be used as a *Point* unless getting it back to a Single Unit Point

**Rules:**

- We would rely only on compile time information
- Implementation shouldn’t be specific to class Point
Explaining the rules (pseudo code)

```plaintext
auto twoPoints = p1 + p2; // twoPoints is NOT a Point

// middle is a Point, but only since 2 is known at compile time
Point middle = twoPoints / 2;

auto thirdOfP1 = p1 / 3; // is NOT a Point
auto twoThirdsOfP2 = p2 * 2 / 3; // is NOT a Point

Point closerToP2 = thirdOfP1 + twoThirdsOfP2; // is a Point!
```
Step 1

// that’s not a good approach... just let’s review it...
TwoPoints operator+(Point p1, Point p2) {
    return TwoPoints{p1, p2};
}

// for class TwoPoints
Point TwoPoints::operator/(int num) {
    // how can we tell if num == 2 and the operation is allowed?
}
// “base template” for Divider - we do not allow dividing by any number rather than 2
template<class T, int num> struct Divider;

// specialized version for Divider
template<class T> struct Divider<T, 2> {
    static T divide() { return T{}; }
};

Point TwoPoints::operator/(int num) {
    return Divider<Point, num>::divide();
}

Oops...
Step 1 - before C++17 - Specialization

// “base template” for Divider - we do not allow dividing by any number rather than 2
template<class T, int num> struct Divider;

// specialized version for Divider
template<class T> struct Divider<T, 2> {
    static T divide() { return T{}; }
};

template<int num> Point TwoPoints::operator/(int) {
    return Divider<Point, num>::divide();
}
Step 1 - before C++17 - Specialization

// “base template” for Divider - we do not allow dividing by any number rather than 2
template<class T, int num> struct Divider;

// specialized version for Divider
template<class T> struct Divider<T, 2> {
  static T divide() { return T{}; }
};

template<int num>
Point TwoPoints::operator/(Number<num>) {
  return Divider<Point, num>::divide();
}
Step 1 - before C++17 - class Number

template<int num> class Number {};
Step 1 - before C++17 - main

```cpp
int main() {
    Point p = TwoPoints{} / Number<2>();
}

http://coliru.stacked-crooked.com/a/7fda5ead44c0eef5
```
Step 2 - replace Number<int> with...

std::ratio<N, D>

Point p1 = TwoPoints{} / std::ratio<2>();

auto twoThirdsOfaPoint = Point{} * std::ratio<2, 3>();
Step 3 - use a generic “Aggregator” (instead of “TwoPoints”...)

template<class T, long Numerator = 1, long Denominator = 1>
class Aggregator {
    T t;
public:
    ... 
};

operator+ 
operator/ 
operator*

return either T or Aggregator<T, ...>
class Point {
    double x, y;

public:
    Point(double x1, double y1): x(x1), y(y1) {} 

    friend Aggregator<Point, 2> operator+(Point p1, Point p2) {
        return Aggregator<Point, 2>{Point{p1.x + p2.x, p1.y + p2.y}};
    }

    ...
}
Step 4 - “Aggregator” using C++17 if constexpr

template<class T1, long Numerator1, long Denominator1,
        long MultNum, long MultDenom>

auto constexpr operator*(Aggregator<T1, Numerator1, Denominator1> a,
                            std::ratio<MultNum, MultDenom> n) {

    if constexpr(Numerator1*MultNum != Denominator1*MultDenom) {
        return Aggregator<T1, Numerator1*MultNum, Denominator1*MultDenom>
                   {a.getT()};
    }
    else {
        return a.getT().unsafe_multiply(n);
    }
}
Step 4 - “Aggregator” using C++17 if constexpr

template<class T1, long Numerator1, long Denominator1,  
        long MultNum, long MultDenom>
auto constexpr operator*(Aggregator<T1, Numerator1, Denominator1> a,  
        std::ratio<MultNum, MultDenom> n) {

    if constexpr(Numerator1*MultNum != Denominator1*MultDenom) {
        return Aggregator<T1, Numerator1*MultNum, Denominator1*MultDenom>  
            {a.getT()};
    } else {
        return a.getT().unsafe_multiply(n);
    }
}
Step 4 - “Aggregator” using C++17 if constexpr

template<class T1, long Numerator1, long Denominator1,
        long MultNum, long MultDenom>
auto constexpr operator*(Aggregator<T1, Numerator1, Denominator1> a,
                         std::ratio<MultNum, MultDenom> n) {
    if constexpr(Numerator1*MultNum != Denominator1*MultDenom) {
        return Aggregator<T1, Numerator1*MultNum, Denominator1*MultDenom>
               {a.getT()};
    } else {
        return a.getT().unsafe_multiply(n);
    }
}
and it works...

Point p1 { 5, 10 }, p2 { 25, 30 };

std::cout << "p1 + p2: " << p1 + p2 << std::endl;

    // prints: p1 + p2: [ Aggregate (2/1) ] : {30,40}

std::cout << "(p1 + p2)/2: " << (p1 + p2) / std::ratio<2>() << std::endl;

    // prints: (p1 + p2)/2: {15,20}
...we can even calculate average

```cpp
auto s = sum(p1, p2, p3 * std::ratio<2>()); // variadic template method sum
std::cout << "average = " << s.average() << std::endl;
```

http://coliru.stacked-crooked.com/a/fd1e09191504e5e5

Same code, with friend functions implemented outside the template class:
https://godbolt.org/z/GsgL2F

(to comply with http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#2174)
See also (1)

Adi Shavit’s blog:
http://videocortex.io/2018/Affine-Space-Types

This talk ignores overflow and signed/unsigned issues.
To take that into account see:
See also (2)

boost::units
CppCon 2015: Robert Ramey “Boost Units”

https://github.com/nholthaus/units
https://github.com/pierreblavy2/unit_lite
https://github.com/bernedom/SI
https://github.com/mpusz/units

Jonathan Boccara’s NamedType:
https://github.com/joboccara/NamedType
void conclude(auto greetings) {
    while(still_time() && have_questions()) {
        ask();
    }
    greetings();
}

conclude([]{ std::cout << "Thank you!"; });