Library Approaches for Strong Type Aliases

Anthony Williams

C++ now

2021
MAY 2-7
Aspen, Colorado, USA

Library Approaches for Strong Type Aliases

- Background — Why do we need this?
- Solutions — How can we do this?

Background

Background

There are 3 reasons for using Strong Type Aliases rather than common types:

- For correctness
- For clarity
- For extensibility

Background

The basic problem is that built-in types and common library types are **too common**.

Even hidden by `typedef` or `using`, they are indistinguishable.

Plus, implicit conversions make things too easily interchangeable.

Problem 1

Confusion about the order of parameters:

```
char const fillChar='0';
int const count=5;
std::string s(fillChar,count);

int const rows=3;
int const columns=4;
Matrix m{columns,rows};
```

Problem 2

Confusion about the units of a value:

```
void sleep(int seconds);
int const pause_milliseconds=5;
sleep(pause_milliseconds);

void accelerate_to(double metres_per_second);
double const target_speed_mph=4.3;
accelerate_to(target_speed_mph);
```

Problem 3

Confusion over meaning of a parameter:

```
int port=1234;
Socket s1{port};
Socket s2{socket(AF_INET,SOCK_STREAM,0)};


int8_t count=get_count();
std::cout<<"There are "<<count<<" elements\n";
```

Problem 4

Lack of extensibility:

Using a common type limits the interface to an exact set of operations. You cannot add additional domain-specific operations that apply to **only** your values.

Problem 5

Difficulty overloading operations:

```
void do_stuff(intfast16_t);
void do_stuff(intfast32_t);
```

Problem 6

Duplicate template instantiations or specializations:

```
template<typename T>
class Stuff{};

template class Stuff<intfast16_t>;
template class Stuff<intfast32_t>;

template<> class Stuff<intleast16_t>{};
template<> class Stuff<intleast32_t>{};
```

Problem 7

It is easy to use the wrong function:

```
using name_type=std::string;
using email_type=std::string;

name_type name="Anthony";
email_type email="anthony@justsoftwaresolutions.co.uk";
print_email(std::cout,email);
print_email(std::cout,name); // oops
```

Solutions

Whole Value

The solution is to create a unique type for each distinct usage.

This is the **Whole Value** idiom: the “whole value” of something includes the type and units:

- £4.32
- 27.6km
- “Anthony” is a **name**
- “anthony@justsoftwaresolutions.co.uk” is an **email address**

Whole Value in C++

The basic idea is to create a class for each use case rather than reusing a common type:

```
struct LengthInMetres{
  double value;
};
```

This quickly gets tedious, especially when you to add operations to your types.

Domain values

We can create a `template` for a specific domain of values, possibly with parameters for the underlying type and units.

- `std::duration<short,std::chrono::nano>`
- `length_t<int,std::kilo>`

We can easily add appropriate domain operations.

```
kmh_t speed=5_km / 2_h;
```

One-off values

Some values are not part of a larger domain:

- Names
- Email addresses
- Numeric IDs
- Row/column counts

But they may have common operations.

Common operations

- Arithmetic operations for numeric values

Common operations

- Arithmetic operations for numeric values
- Stream output

Common operations

- Arithmetic operations for numeric values
- Stream output
- Comparisons — equality or ordering

Common operations

- Arithmetic operations for numeric values
- Stream output
- Comparisons — equality or ordering
- Hashing — for use as a key to an `unordered_map`

Common operations

- Arithmetic operations for numeric values
- Stream output
- Comparisons — equality or ordering
- Hashing — for use as a key to an `unordered_map`
- Conversion to a string

Common operations

- Arithmetic operations for numeric values
- Stream output
- Comparisons — equality or ordering
- Hashing — for use as a key to an `unordered_map`
- Conversion to a string
- User-defined conversions

Eliminating boilerplate

Implementing the common operations for every one-off type is tedious, time consuming, and leads to duplicated code. $\Longrightarrow$ We need a solution that eliminates this boilerplate

There are fundamentally two general solutions in C++: **Macros** or **Templates**.

Macros are horrid, so that leaves us with `template`s.

Overall design

We want the declaration of a custom type to be as simple as possible:

```
using MyType=strong_typedef</* something */>;
```

More than this is too much boilerplate, and it quickly becomes easier to manually write a custom type.

Unique types

We want each use to have a unique type, otherwise it defeats the purpose.

We also need to specify the underlying type.

```
using MyType=strong_typedef<
   struct my_type_tag, // a tag type for uniqueness
   int,                // the underlying type
   /* other args */>;
```

Other requirements

- Explicitly constructible from underlying type
  So we can say `MyType(42)`
- **Not** implicitly convertible from underlying type
  So we can't accidentally pass values to the wrong argument.

```
int f(MyType mt);
int i=f(42); // won't compile
```

Additional operations

We want to make it easy to add operations like arithmetic, comparisons, etc.

We can do this with the additional arguments:

```
using MyType=strong_typedef<
  struct my_type_tag,
  int,
  strong_typedef_properties::addable,
  strong_typedef_properties::equality_comparable>;
```

Value access

Access to the underlying value is via the `underlying_value` member function:

```
MyInt mi{42};
int& i=mi.underlying_value();
i+=99; // update internal value
```

Explicit conversion

Values can be explicitly converted to the underlying value:

```
MyInt mi{42};
int i=mi; // error
int j=static_cast<int>(mi); // OK
```

Basic definition

Our `strong_typedef` template provides basic operations itself:

```
template <typename Tag, typename Type, typename... Props>
class strong_typedef: /* bases */ {
public:
  constexpr strong_typedef() noexcept;
  explicit constexpr strong_typedef(Type value_);
  explicit constexpr operator Type const &() const noexcept;
  constexpr Type const &underlying_value() const noexcept;
  constexpr Type &underlying_value() noexcept;
private:
  Type value;
};
```

Property definitions

A property such as `pre_incrementable` is a type with a member template `mixin`.

```
struct pre_incrementable {
  template <typename Derived, typename ValueType>
  struct mixin {
    friend Derived &operator++(Derived &self) noexcept(
    noexcept(++std::declval<ValueType &>())) {
      ++self.underlying_value();
      return self;
    }
  };
};
```

Using properties in the implementation

Our template can thus derive from the mixins:

```
template <typename Tag, typename ValueType,
  typename... Properties>
class strong_typedef
  : public Properties::template mixin<
    strong_typedef<Tag, ValueType, Properties...>,
    ValueType>... {
  // ...
};
```

Using properties

Fine grained properties mean you can have fine-grained control over operations available on your types:

```
using SessionId=strong_typedef<
  struct session_id_tag, unsigned long long,
  strong_typedef_properties::post_incrementable,
  strong_typedef_properties::equality_comparable,
  strong_typedef_properties::ordered,
  strong_typedef_properties::streamable>;
```

`SessionId`s can be compared for equality or ordering, written to a stream, or incremented with `id++`, but general arithmetic and other operations are forbidden.

Combining properties

This still gets tedious if you have lots of common operations: equality **and** ordering comparisons, addition **and** subtraction, **all** the bitwise operations, etc.

The mixin-based scheme allows you to combine properties:

```
struct comparable {
  template <typename Derived, typename ValueType>
  struct mixin
    : ordered::template mixin<Derived, ValueType>,
      equality_comparable::template mixin<Derived, ValueType>
};
```

Hashing

A common requirement is support for `std::hash`, so the type can be used as a key in a `std::unordered_map`.

This requires specializing `std::hash`, since it has no extension points.

We only want it to work where the user has requested it.

Hashing

The mixin itself just derives from an empty struct:

```
struct hashable {
  struct base {};
  template <typename Derived, typename ValueType>
  struct mixin : base {};
};
```

Hashing

The specialization then checks for the base class:

```
template <typename Tag, typename Type, typename... Prop>
struct hash<strong_typedef<Tag, Type, Prop...>> {
  template <typename Arg>
  typename std::enable_if<
    std::is_base_of<
    strong_typedef_properties::hashable::base, Arg>::value,
    size_t>::type
  operator()(Arg const &arg) const noexcept(noexcept(
    std::hash<Type>()(std::declval<Type const &>()))) {
    return std::hash<Type>()(arg.underlying_value());
  }
};
```

Custom properties

Though the library provides properties for most basic things, you might want others. For example, a type based on `std::string` might want to provide `s.c_str()` or `s.substr()` operations.

This can be easily done by defining your own property type.

Custom properties

```
struct string_properties{
  template<typename Derived,typename ValueType>
  struct mixin{
    const char* c_str() const noexcept{
      return static_cast<const Derived&>(*this).
        underlying_value().c_str();
    }
    Derived substr(
      size_t pos,size_t length) const noexcept {
      return Derived(static_cast<const Derived&>(*this).
        underlying_value().substr(pos,length));
    }
  };
};
```

Example Uses

```
using SessionId=strong_typedef<
  struct session_id_tag, unsigned long long,
  stp::incrementable, stp::streamable,
  stp::comparable, stp::hashable>;

using UserName=strong_typedef<
  struct username_tag, std::string,
  stp::streamable, stp::comparable,
  stp::hashable>;

using Password=strong_typedef<
  struct password_tag, std::string>;
```

Debugging

gdb displays empty base classes by default if you display values, so all those mixins get printed, complete with their type names and template parameters.

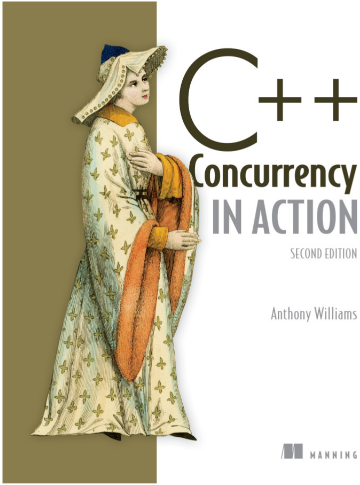
I added a custom pretty printer to print just the value; you might want to do the same for other debuggers.

```
(gdb) print st
{value=42}
```

Links

- jss::strong_typedef
  https://github.com/anthonywilliams/strong_typedef
- PSsst
  https://github.com/PeterSommerlad/PSsst
- WholeValue
  https://github.com/martinmoene/WholeValue
- Boost.Units
  https://www.boost.org/doc/libs/1_76_0/doc/html/boost_units.html
- mp-units
  https://github.com/mpusz/units

My Book



C++ Concurrency in Action
Second Edition

Covers C++17 and the Concurrency TS

C++20 addendum coming soon

cplusplusconcurrencyinaction.com

Questions?