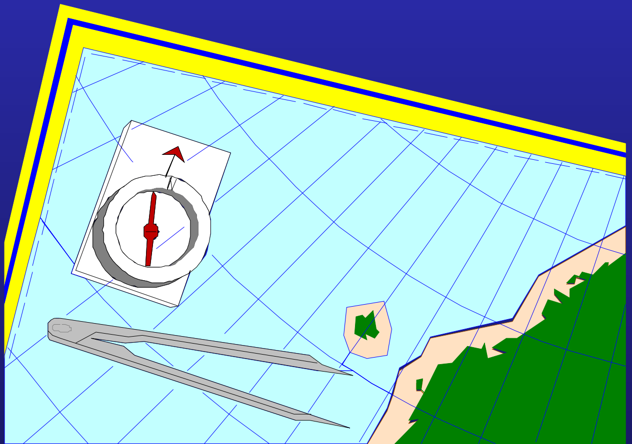
The C++ Type System is your Friend

NDC Oslo, June 2016

Hubert Matthews
hubert@oxyware.com



Why this talk?

Safe, performant, reusable code

- General desiderata (in priority order):
  - We want our code to help us prevent avoidable mistakes, preferably at compile time
  - We want the run-time cost of this safety to be zero compared to unsafe coding (or people will avoid doing it)
  - We want the code to be reusable and generic (i.e. a library) so we can avoid having to reimplement it every time

(Mostly) typeless programming

- Assembler
  - Integer can be used as an address and *vice versa*
  - Machine efficiency at the cost of programmer effort
  - Translate into the language – domain knowledge is embedded, not obvious or easy to decipher
  - Liberal use of comments (hopefully!)
  - High maintenance cost
- B, BCPL
  - Hardly any type safety
  - 3 * (4 + 5) gives the value 27
  - 3 (4 + 5) calls function at address 3 with value 9
- C preprocessor
  - Programming with strings

Machine-typed programming

- C and primitive-based C++
  - Avoids the type puns and mistakes of assembler
  - High machine efficiency
  - Better programmer efficiency
  - Uses the underlying machine types (int, float, typed pointers)
  - Adds structures and aggregates
  - Abstraction through files
  - Still have to translate domain into a program
  - Little opportunity for compile-time checking or proofs

Type-rich programming

- Higher-level C++
  - Uses the C++ type system extensively to create lightweight abstractions that increase the amount of domain knowledge in the program without sacrificing machine efficiency
  - The type system is a proof system – 100% compile-time checking if a construct is illegal
  - Well used, it can make code safer and more reusable
  - Stroustrup is a big fan of this approach

Little white lies:
C structs, C++ RTTI

The miracle of compilation

	Run time	Compile time	
C++	Machine types	Language types	Application types
C	Machine types	Language types	
asm	Machine types		

what this talk is focused on

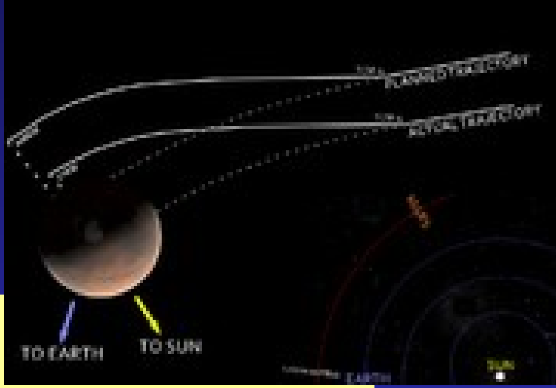
thrown away at run time (no memory or CPU overhead)

Primitive or typed API

```
// Is this y/m/d (ISO), d/m/y (European) or m/d/y (broken US)?

Date(int, int, int);

// Unambiguous and expressive

Date(Year, Month, Day);

// Helps with expressivity but not correctness as it's just a
// aliased type

using Year = int;       // just a type alias

// We need a completely separate type to get safety as well

class Day { /*...*/ };
```

- Creating separate types for values catches type errors at compile time

Physical types




```
typedef float meters, seconds;

meters m = 3.4;
seconds s = 1.5;
auto velocity = m + s;  // oops, probably meant / not +
                        // but it still compiles

typedef float feet;

feet f = 5.6;
meters m2 = m + f;      // physical units correct but
                        // measurement system wrong

// Mars Climate Orbiter crashed because of a pound-seconds
// and newton-seconds mismatch (1999)
```

- Lots of possibilities for simple errors that are hard to find and debug but easy to prevent

Whole Value pattern

```
class Year {
public:
  explicit Year(int y) : yr(y) {}
  operator int() const { return yr; }
private:
  int yr;
};

Year operator"" _yr(unsigned long long v) { return Year(v); }

Year y = 2016_yr;
```

explicit c/tr to avoid automatic conversions

user-defined conversion is safe: narrow to wide

- Holds a value but has no operations – all operations done on the base type (int, here) through widening conversion
- Safe way to pass values but not foolproof
- Repetitive when defining multiple types

Templates to the rescue

```
enum class UnitType { yearType, monthType, dayType };

template <UnitType U>
class Unit {
public:
  explicit Unit(int v) : value(v) {}
  operator int() const { return value; }
private:
  int value;
};

using Year = Unit<UnitType::yearType>;
using Month = Unit<UnitType::monthType>;
using Day = Unit<UnitType::dayType>;

Date(Year, Month, Day);          // now type-safe API
```

- Removes repetition across types
- As efficient as primitives; functions are inlined

Adding checking of values

```
template <UnitType U, int low, int high>
class Unit {
public:
  constexpr explicit Unit(int v) : value(v) {
    if (v < low || v > high) throw std::invalid_argument("oops");
  }
  constexpr operator int() const { return value; }
private:
  int value;
};

using Year = Unit<UnitType::yearType, 1900, 2100>;

Year tooSmall(1000);                     // throws at run-time
constexpr Year tooBig(2300);             // compile-time error
```

- Extra checking for types can be added for both run-time and compile-type checking
- Constexpr is very powerful keyword for this

Operations

- Up to now we have used conversions to allow us to operate on our types, which is simple but possibly error-prone as we can't control what operations are valid (we get everything that *int* can do)
- Essentially our types are just labels
- Let's add operations and remove the conversion (or make it explicit)

Operations

Please imagine all functions are constexpr – it makes the slides shorter!

```
template <UnitType U, int low, int high>
class Unit {
public:
  constexpr explicit Unit(int v) : value(v) {
    if (v < low || v > high) throw std::invalid_argument("oops");
  }
  constexpr explicit operator int() const { return value; }
private:
  int value;
};

Year operator+(Year yr, int i) { return Year(int(yr)+i); }
Year operator+(int i, Year yr) { return Year(int(yr)+i); }

// define only those operations that make
// sense in the domain for a given type
```

- Year+Year doesn't make sense but Year+int does, as does Year-Year

Operations

Type	Desirable operations	Non-sensical operations
Date	Date+int => Date int+Date => Date Date-Date => Date-int => Date Date < Date => bool Date == Date => bool	Date * int
Money	Money * float => Money Money / float => Money Money < Money => bool Money == Money => bool	Money + float Money – float

- Every type has its own set of operations
- How to make this generic?
- How do we avoid repetitive boilerplate code?

Reuse through client libraries

```
bool operator==(Year y1, Year y2) { return int(y1) == int(y2); }
bool operator<(Year y1, Year y2) { return int(y1) < int(y2); }

#include <utility>
using namespace std::rel_ops;       // defines <=,>=,>,!=

// namespace std { namespace rel_ops {
//   template <typename T>
//   bool operator>(T t1, T t2) { return t2 < t1; }
// }}

bool ge = y1 >= y2;
```

- Can't be used in the definition of a class
- Client has to decide to use these broad templates
- Only handles relational operators

Reuse through inheritance – CRTP

```
template <typename Derived>
class Ordered {
public:
  const Derived & derived() const {
        return static_cast<const Derived &>(*this);
  }
  bool operator>(const Ordered & rhs) const {
    return rhs.derived() < derived();
  }
};

class Year : public Ordered<Year> {
public:
  explicit Year(int i) : val(i) {}
  bool operator<(const Year & rhs) const { return val < rhs.val; }
private:
  int val;
};

int main() {
  Year y1(7), y2(5);
  assert(y1 > y2);      // true
}
```

downcast to Derived is safe

CRTP pattern: deriving from a template using yourself!

Reuse through inheritance – CRTP

- The cast in Ordered::derived() is checked at compile-time as it's a static_cast
- There is no overhead in terms of space or time
- All calls are resolved at compile time
- Compile-type polymorphism
- Using a virtual call instead would mean:
  - Larger class (vtable pointer)
  - Run-time dispatch (virtual call)
  - Can't be constexpr (forces run-time eval)
  - Probably not inlined
- Very common technique in libraries like Boost

Physical quantities

```
template <typename V, UnitSys U, int M, int L, int T>
class Quantity {
public:
  explicit Quantity(V v) : val(v) {}
  explicit operator V() const { return val; }
private:
  V val;
};

template <typename V, UnitSys U, int M, int L, int T>
auto operator+(Quantity<V, U, M, L, T> q1, Quantity<V, U, M, L, T> q2) {
  return Quantity<V, U, M, L, T>(V(q1) + V(q2));
}

template <typename V, UnitSys U,
                int M1, int L1, int T1, int M2, int L2, int T2>
auto operator/(Quantity<V, U, M1, L1, T1> q1,
                          Quantity<V, U, M2, L2, T>2 q2) {
  return Quantity<V, U, M1-M2, L1-L2, T1-T2>(V(q1) / V(q2));
}

using meters = Quantity<float, SIUnits, 0, 1, 0>;
using seconds = Quantity<float, SIUnits, 0, 0, 1>;

int main() {
  auto velocity = 23.1_meters / 1.5_secs;
  // auto error = 23.1_meters + 1.5_secs;    // compile-time error
}
```

Physical quantities and dimensions

- Allows us to define operations that convert types (here the dimension exponents are calculated to give new dimension values)
- Prevents physically impossible calculations
- Prevents mixing of measurement units (e.g. mixing SI Units and imperial units)
- Can be used for related "flavours" of types, such as multiple currencies that are "the same underlying thing" but with different units

Compile-time reflection

```
template <typename V, UnitSys U, int M, int L, int T>
class Quantity {
public:
  using value_type = V;
  static constexpr UnitSys unit_sys = U;
  static constexpr int mass_exponent = M;
  static constexpr int length_exponent = L;
  static constexpr int time_exponent = T;
  explicit Quantity(V v) : val(v) {}
  explicit operator V() const { return val; }
private:
  V val;
};

using length = Quantity<float, SIUnits, 0, 1, 0>;
using time = Quantity<length::value_type, length::unit_sys, 0, 0, 1>;

template <typename V, UnitSys U>
using Mass = Quantity<V, U, 1, 0, 0>;

template <typename Q>
void print_units(Q q) {
  if (Q::unit_sys == UnitSys::SIUnits)
    std::cout << “Using SI units\n”;
}
```

republish template parameters

create a compatible type using reflection

if statement based on constants will be removed

Tailoring operations – library code

```
template <typename T>
struct op_traits {
    static constexpr bool add_scalar = false;
    static constexpr bool add_value = false;
};

template <typename T, typename Requires =
        std::enable_if_t<op_traits<T>::add_scalar>>
auto operator+(T t, int i)
{
    return T{t.val+i};
}

template <typename T, typename Requires =
        std::enable_if_t<op_traits<T>::add_value>>
auto operator+(T t1, T t2)
{
    return T{t1.val+t2.val};
}

// same for operator+(int i, T t);
```

Tailoring operations – client code

```
struct Year { int val; };

template <>
struct op_traits<Year> {
    static constexpr bool add_scalar = true;
};

int main() {
    Year y1{10}, y2{5};
    //auto y3 = y1 + y2;                // compiler error
    auto y3 = y1 + 2;
}
```

- Library user defines what operations from the library are valid by setting the appropriate traits

Where are we now?

- Let's look at the generated code for an example that puts all of these things together to see how efficient it is (both code and data)
  - Constexpr and user-defined literals
  - Physical dimensions and unit types
  - CRTP for operator inheritance

```
int main()
{
    Distance d1 = 5.2_meters;
    Distance d2 = 4.6_meters;
    Time t = 2.0_secs;
    auto v = (d1+d2+Distance(d1 > d2)) / t;
    return int(float(v));
}
```

```
// generated code
// g++ -O3

  movl    $5, %eax
  ret

// return 5;
```

Templates and policies

- Another example: fixed-length strings that prevent the sort of basic buffer overflow bugs that traditionally haunt C programs

```
template <size_t N>
class FixedString {
public:
  static constexpr max_size = N;
  explicit FixedString(const char * p = “”) {
    strncpy(data, p, N);
    data[N-1] = 0;
  }
  size_t size() const { return strlen(data); }
private:
  char data[N];
};
```

truncates the incoming string – this is a policy decision

Templates and policies

- This class truncates its input. This may be what you want, but there are other options:
  - Add an entry to the diagnostic log and continue (if overflow is expected and OK)
  - Throw an exception (if overflow shouldn't happen)
  - Reboot the system (if overflow is a serious error)
  - Dump a stack track and jump into the debugger (during development and test)

Implementing policies

- Let's use a policy on overflow

```
struct Complain {
  static void overflow(size_t n, const char * s, const char * p) {
      std::cout << "Overflow of FixedString<" << n << "> and "
                << "contents " << s << " when adding " << p << std::endl;
  }
};

template <size_t N, typename OverflowPolicy = Complain>
class FixedString {
public:
  constexpr explicit FixedString(const char * p = "") {
    char * s = data;
    while (s-data != N-1 && (*s++ = *p++)) {}
    if (*(p-1) != 0) OverflowPolicy::overflow(N, data, p);
    *s = 0;
  }

private:
  char data[N];
};
```

```
FixedString<8> fs1(“hello”);        // no overflow
FixedString<5> fs2(“hello”);        // prints msg

template <size_t N>
using NoisyString = FixedString<N, ResetOnOverflow>;
```

Comparing policies and CRTP

- CRTP has to use a compile-time downcast to access the derived class' functionality (i.e. to get itself “mixed in”)
- CRTP is usually used for injecting library functionality

- Policies don't need a downcast as they are a pure “up call” to a static function
- Policies are useful for parametrising rules and validation logic (such as in constructors)

Constructor validation logic

- Let's use a policy to enforce that quantities are non-negative

```
struct NonNegChecker {
  constexpr NonNegChecker(float f) {
      if (f < 0) throw std::invalid_argument("oops!");
  }
};

template <UnitType U, int M, int L, int T, class CtrCheck=NonNegChecker>
class Quantity : public Ordered<Quantity<U, M, L, T>>, public CtrCheck {
public:
  constexpr explicit Quantity(float v) : CtrCheck(v), val(v) {}
  constexpr explicit operator float() const { return val; }
  bool operator<(Quantity other) const { return val < other.val; }
private:
  float val;
};
```

Constexpr constructor check

- Constexpr in effect interprets your code at compile time using a cut-down version of the compiler
- C++11 version is limited, C++14 is general
- Some limitations
  - Can't initialise the string directly
- If the CtrCheck constructor doesn't complete correctly because an exception has been thrown then this becomes a compiler error
- If it doesn't throw then no code is generated for CtrCheck

Effect of constructor validation logic

- So, what about the generated code?

```
constexpr Distance d0 = -1.1_meters;
```

```
(compiler error)
```

```
Distance d0 = -1.1_meters;
```

```
(throws at runtime)
```

```
int main()
{
    Distance d1 = 5.2_meters;
    Distance d2 = 4.6_meters;
    Time t = 2.0_secs;
    auto v = (d1+d2+Distance(d1 > d2)) / t;
    return int(float(v));
}
```

```
// generated code
// g++ -O3

  movl    $5, %eax
  ret
```

same as before

Summary

- Defining lightweight domain abstractions allows us to have safer code with more domain knowledge embedded in the code
- Zero or small runtime overhead in terms of CPU or memory
- Can create reusable domain-specific libraries

(Disclaimer: There is no guarantee your programs will end up being only a single instruction when using these techniques)