How I learned to Stop Worrying and to Love the C++ Type System

"I observed that type errors almost invariably reflected either a silly programming error or a conceptual flaw in the design."
-- Bjarne Stroustrup, The Design and Evolution of C++

Prof. Peter Sommerlad
Director of IFS
C++Now 2019

@PeterSommerlad
peter.cpp@sommerlad.ch

Type = ({values}, {operations})

set of values
set of operation on values

What is a Type?

Type Theory often taught far away from use of types in programming

- **A Type denotes**
  - a set of possible values and
  - a set of possible operations on these values

Type Theory associates a term (expression) with a type

- **Cardinality(=size) of the first set denotes the least number of bits required for representing all values**
  - languages and hardware might impose additional overhead (e.g. alignment, run-time type information)
- **Operations include operators, functions, in general all possible uses of the values**
  - operations define the meaning of the values
  - also define possible conversions (implicit or explicit)

"Types provide meaning to programs"

- **This is often only taught implicitly from using a programming language**
  - it took me decades to actually learn about it well enough and I am still learning...

How many types does a language (need to) support?

- **SNOBOL - everything is a string**
- **Lisp - everything is a list (except if it is an atom - number, string, symbol)**
- **Smalltalk - everything is an object (but with mechanism to define classes, unlimited extensibility)**
- **FORTRAN -**
- **AWK -**
- **Javascript -**
- **Bash -**
- **ASM -**
- **λ-calculus -**

Object-orientation was first to allow user-defined types as "first-class" entities (Simula)

- **Variable names encode type: numbers and string (and arrays)**
  - `A=42`
  - `A$="The Answer"`
- **Conversion from integers to floating point implicit**
- **Conversion between numbers and strings explicit:**
  - `STR$(42.0)`
  - `INT(3.14)`
  - `VAL("42")`

★No own types possible

★just 2.5 types

★No extension of operations

Img Source: https://upload.wikimedia.org/wikipedia/commons/3/33/ZXSpectrum48k.jpg - Bill Bertram 2005 CC SA

My second experience with types: PASCAL (old style 1980s)

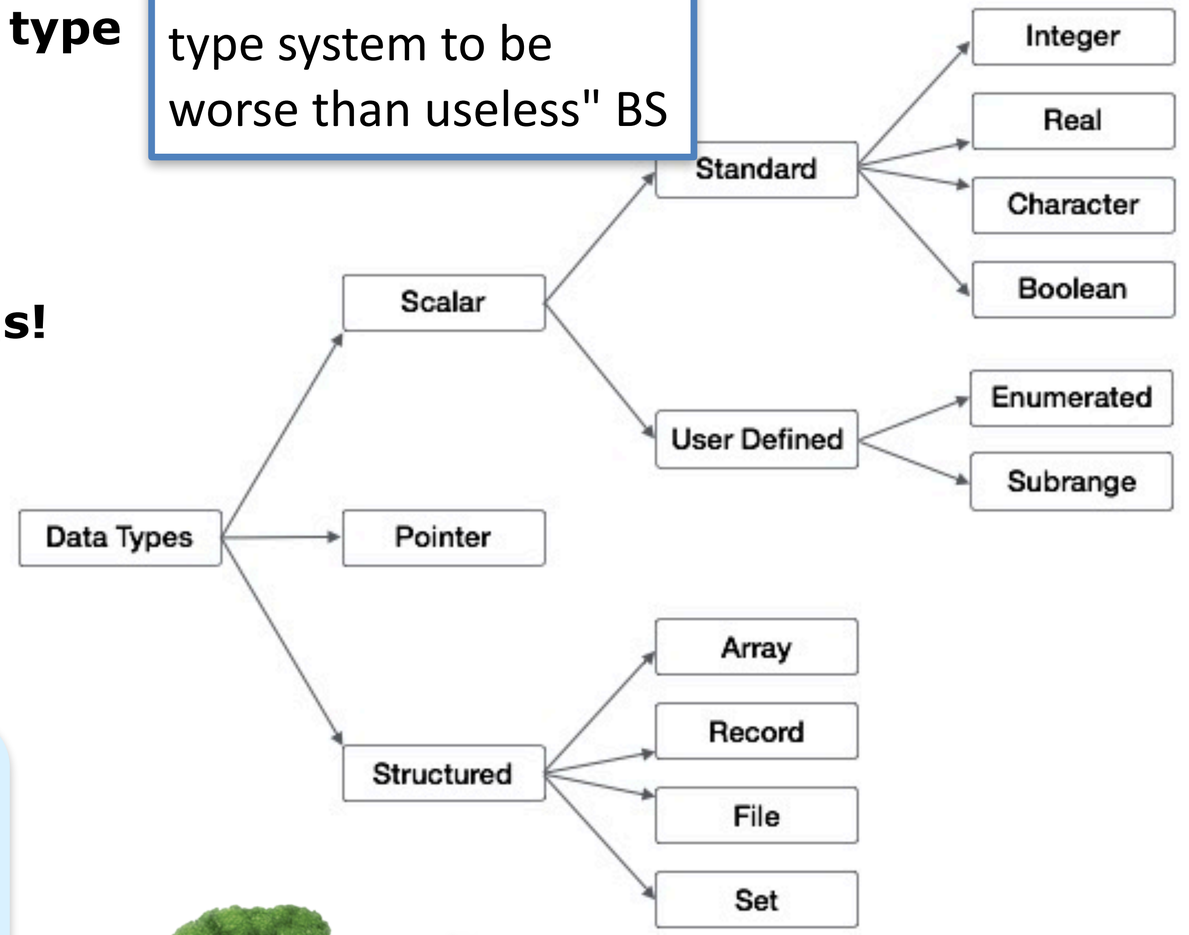
- **Variables to be declared up-front stating their type**
- **No implicit conversions**
  - not even from INTEGER to REAL
- **Array type includes dimension - also for strings!**
- **"product" and "sum" types**
  - RECORD and variant RECORD (CASE .. OF)
- **"Incomprehensible" (first) Pointer**

★type construction possible

★functions and procedures

★annoying rigidity: conversions, size

"I had found Pascal's type system to be worse than useless" BS

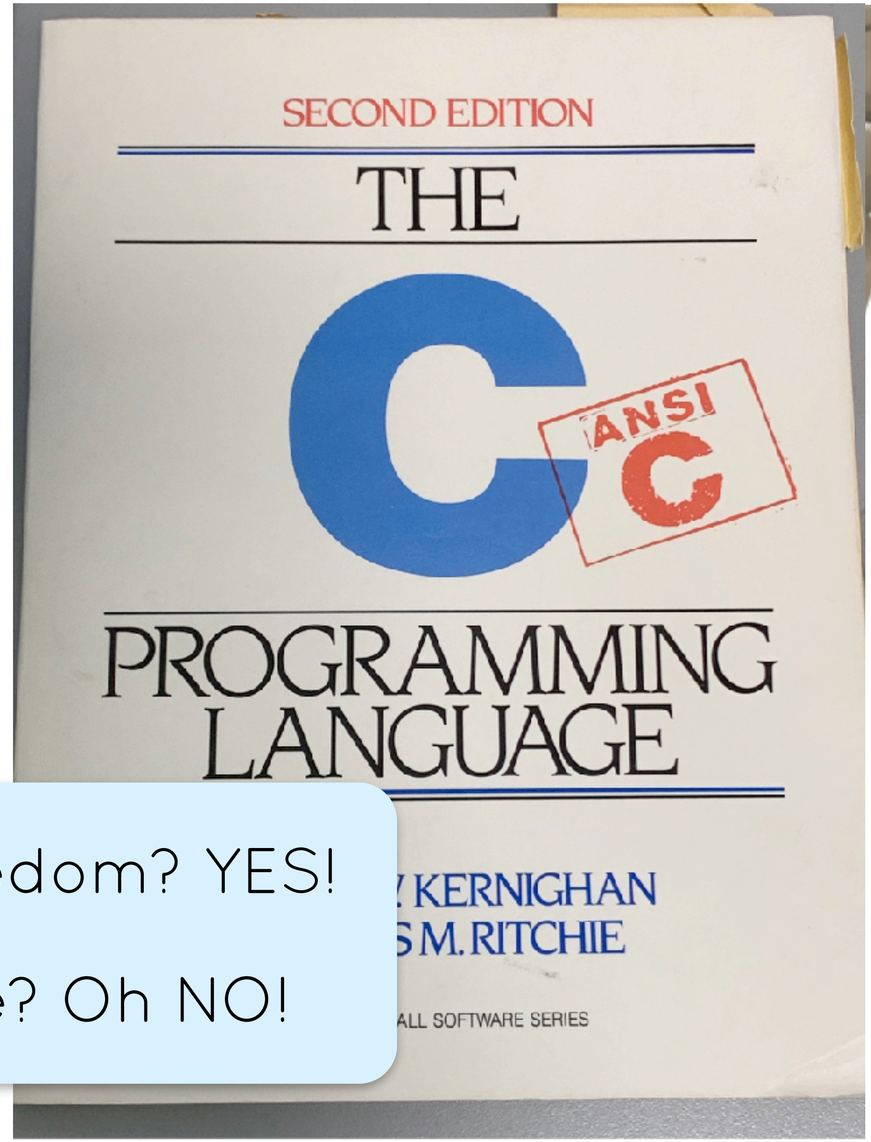


source: https://www.tutorialspoint.com/pascal/pascal_data_types.htm

"Quiche eaters use Pascal"

Then came C- full power with full responsibility

- **all the usual**
  - many implicit conversions (real 🧗 know it)
- **type construction including pointers, arrays, product types (struct) and sum types (union)**
  - defines the size
- **arrays decay to pointers**☹️
  - programmer knows how to take care
- **power of arbitrary memory reinterpretation**💪🏽
  - early: no checks at all on function calls
  - varargs
  - casting void*
- **Many many conventions (not consistent)**



★Freedom? YES!

★Safe? Oh NO!

- **first popular user-defined types**
- **simulates, what OO languages provide**
  - behavior is prime, not representation
- **often taught, but ignored by practitioners**
  - "I want to see how it works"
- **requires discipline**
  - to not access memory behind the pointer
- **overhead through hiding**
  - loses inlining capability

```
#ifndef SRC_STACK_H_
#define SRC_STACK_H_

// Abstract Data Type in C

#ifdef __cplusplus
extern "C" {
#endif

struct Stack* makeStack();
void destroyStack(struct Stack* s);
int  countStack(struct Stack* s);
void pushStack(struct Stack* s, int i);
int  topOfStack(struct Stack* s);
void popStack(struct Stack* s);

#ifdef __cplusplus
}
#endif


#endif /* SRC_STACK_H_ */
```

A Type in C (and C++)

typedef does not define a type but an alias

- **determines sizeof(T)**
- **determines interpretation of underlying memory**
- **determines validity of built-in operators**
- **determines validity of being passed to functions**
  - including operator overloads
- **determines registers/memory used for arguments and return values**
  - ABI specification

C++:

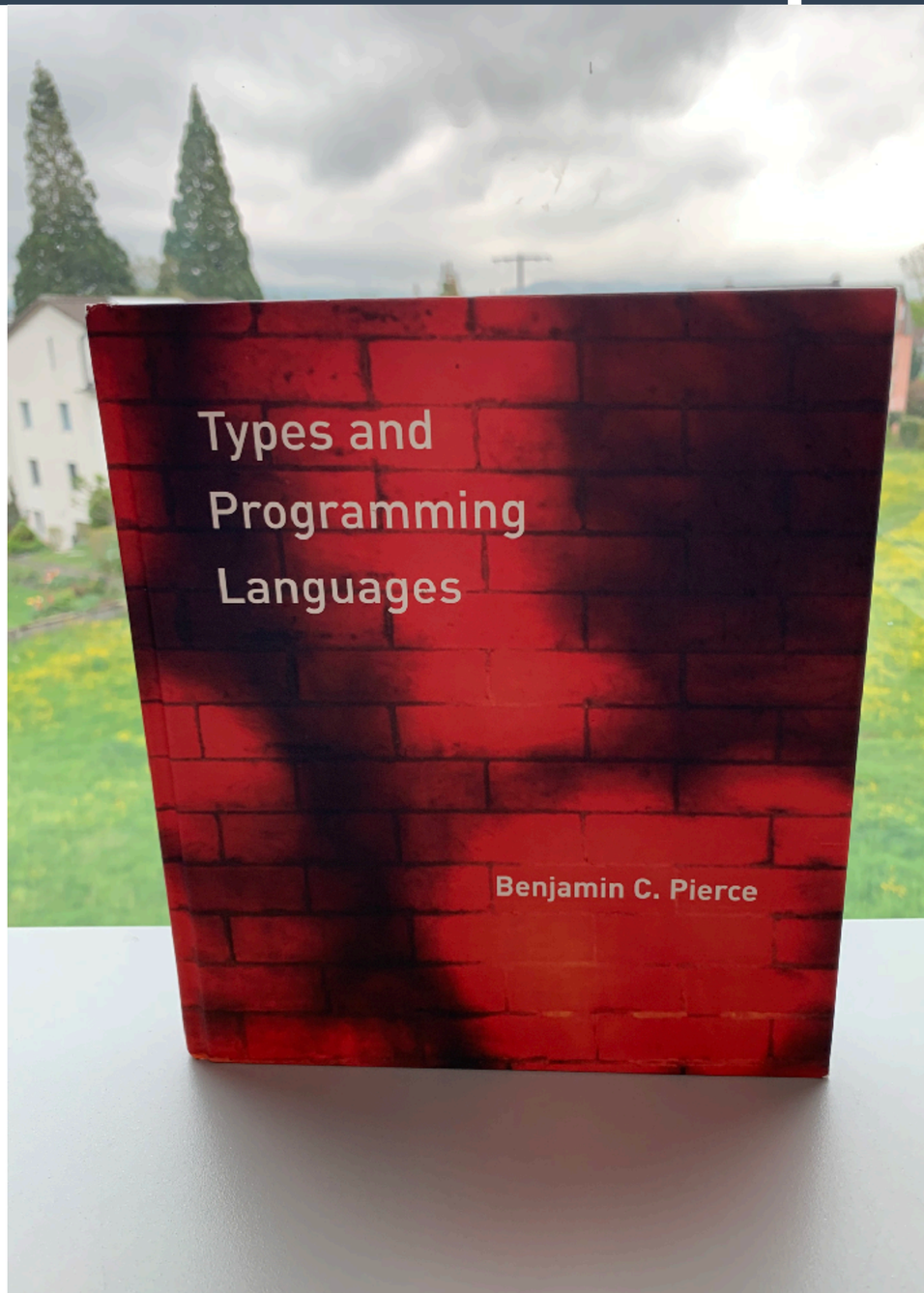
- **determines selection of function overload**
- **can determine instantiation of templates**
- **can represent values as types for meta-programming**

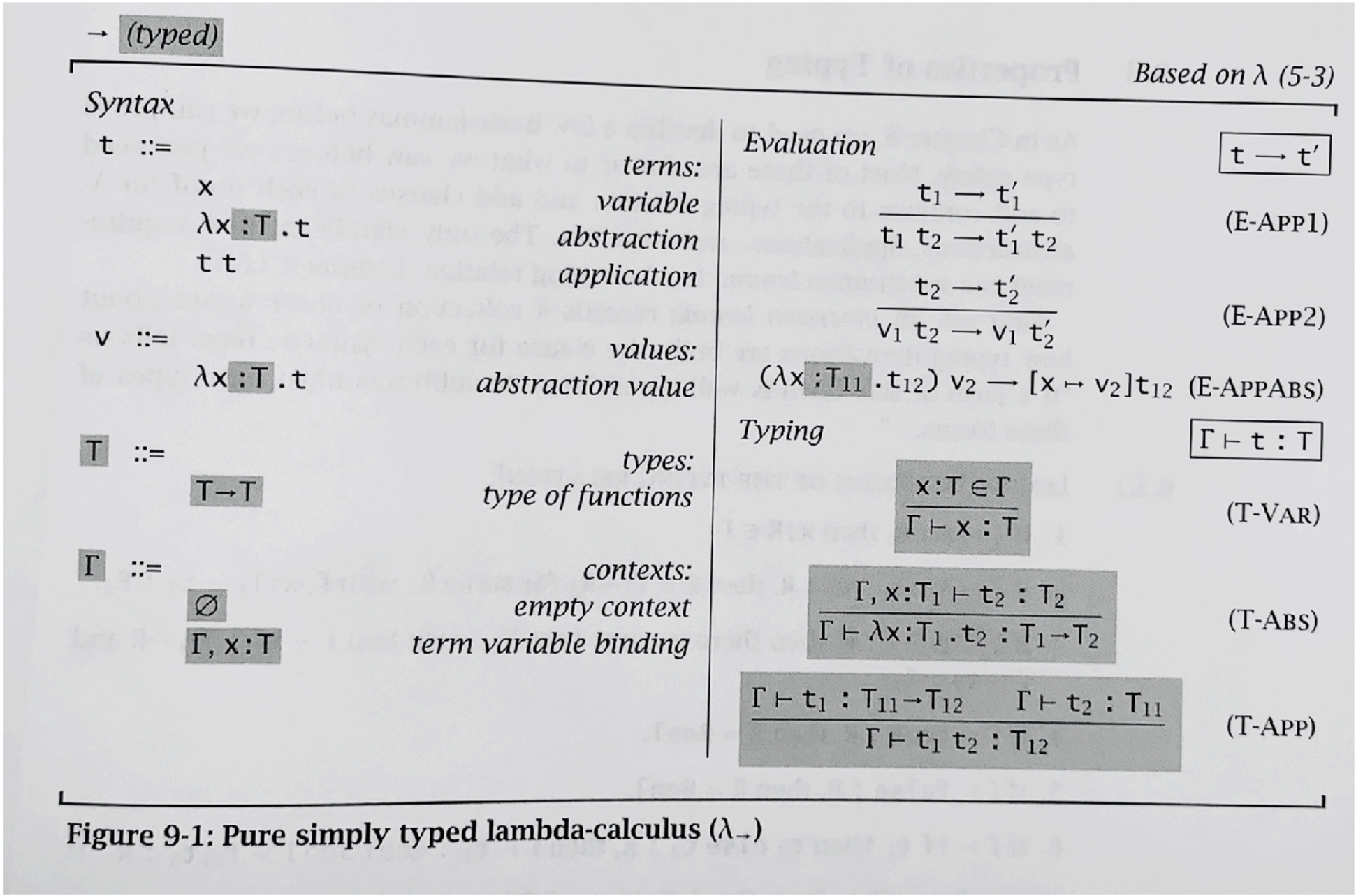
Type System

Provide meaning to programs
Keep track of the types of "program stuff"
Raise type errors

What is a Type System?

- **A Type System associates types with program elements**
  - this provides meaning (semantics) to the program
  - prevents mis-interpretation of values (data bits)
- **A compiler or interpreter can act upon type violations**
  - e.g., using an unsupported operation
  - e.g., combining types that you can not in an expression
- **A language can also support type coercion and conversion**
  - convert values to a different type to enable operations
- **Selection of operations, construction of "stuff" based on types**
  - overload resolution, template instantiation and selection





→ *(typed)*

Based on λ (5-3)

Syntax

t ::=		*terms:*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
v ::=		*values:*
	λx:T.t	*abstraction value*
T ::=		*types:*
	T→T	*type of functions*
Γ ::=		*contexts:*
	∅	*empty context*
	Γ, x:T	*term variable binding*

Evaluation $t \longrightarrow t'$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

Typing $\Gamma \vdash t : T$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

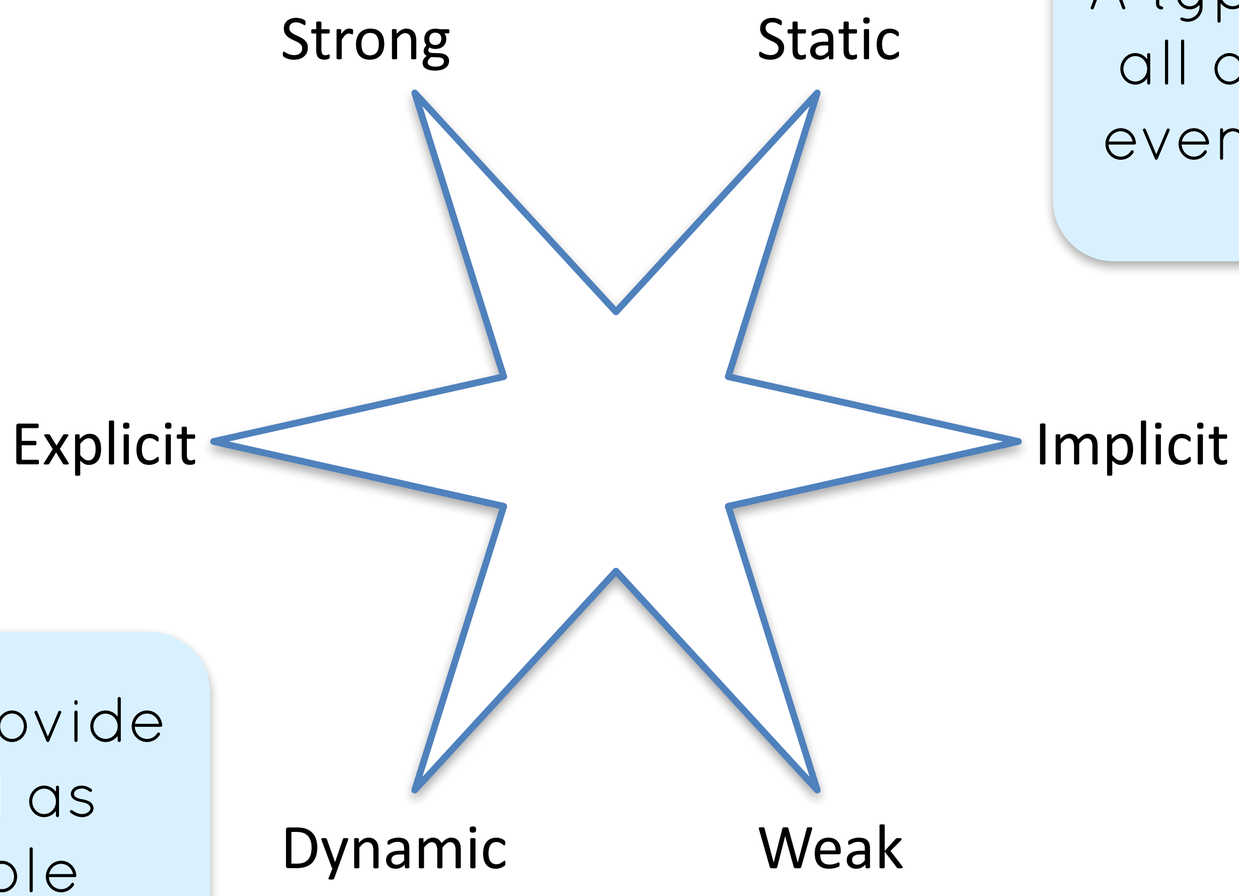
$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

Figure 9-1: Pure simply typed lambda-calculus ($\lambda_{\to}$)

Type System opposing views

A type system can have all of these properties even all opposing ones



C++ attempts to provide strong, static and as implicit as possible

Strong versus Weak Type System

- **Strong:**
  prevent mis-interpretation of data bits
  - safer and no doubt about type of an expression
  - easier to interpret by humans and tools
  - clear feedback when violated
  - cumbersome when conversions needed
  - no type punning
- **Weak:**
  allow implicit conversion and coercion
  - convenient: less annoying to programmers
  - confusion through intricate rules
  - coercion can lead to ambiguities with tie-breaking rules
  - re-interpretation of data bits allowed
  - ripple effects through data flow
  - type punning

★ weak typing needed for low-level efficiency

★ encapsulate that in strong-typed abstractions

Explicit versus Implicit Typs: often combined

- **Explicit**
  - Need to specify type in detail
  - can be annoying to state the obvious or be redundant:
    - String s = new String("hello"); // Java
  - often combined with strong typing
  - can hinder generic code, unless overloading is available
  - conversion always requires a function (or a cast)
- **Implicit (e.g. auto)**
  - type of element is deduced
  - less effort for programmer
  - can be confusing, because one needs to know all intrinsic tools, especially when automatic conversion can happen
  - tools help visualizing deduction
  - easier to spell generic code
  - should be combined with strong typing

★ Be only as explicit as you need to and rely on tools or knowledge for code understanding (almost always auto)

Static (compile-time) vs. Dynamic (run-time) Type checking

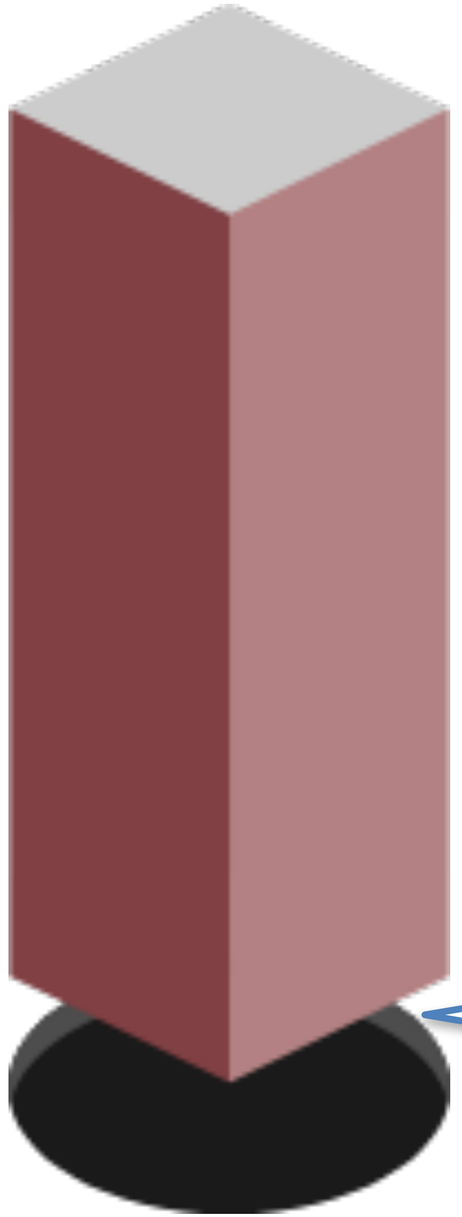
- **Static:**
  wrong type combination won't compile
  - safer
  - type errors can not lead to run-time errors
  - most of C++ type checking is static
  - some languages, like Haskell, have full static type checking: if the program compiles there can not be type errors

- **Dynamic:**
  type mismatch detection at run-time
  - still safe, but raise errors/exceptions
  - dynamic_cast<Base&>(derivedobject)
  - exception handling (today)
  - many typed interpreted languages have dynamic type checking

★ Prefer static type checks over dynamic run-time type errors

What can happen if types of things do not fit intent or together?

- **Garbage (e.g. ASM)**
- **Crash**
- **Undefined Behavior**
- **invalid value: NaN, "", nullptr**
- **Nullpointer Exception**
- **Signal**
- **Ignore and continue**
- **Implicit conversion**
- **Runtime error**
- **Compile error**
- **Compile warning**
- **SFINAE**



You want to detect bugs as early as possible!

"put a square peg in a round hole"

What benefits do you get from the C++ type system

- **User-defined Types as "first class" entities**
- **Type safety (potentially)**
- **Type deduction (auto, template argument deduction)**
- **Static polymorphism:**
  - Overload resolution
  - Templates
- **Efficient Templates**
- **Compile-time computation**
  - even meta-programming analyzing and constructing types
- **Efficiency of generated code**
  - compilers employ type system to improve optimization

Why is making a type system for a programming language hard?

- **Type checking costs compile time - but can gain optimization opportunities**
- **Type deduction can be expensive (e.g., Swift includes a constraint solver)**
- **Systems programming requires to re-interpret bits breaking a sound type system**
- **Proving soundness can be required and hard**
- **Simplification (everything is a ...) can lead to inefficient code**
- **Incrementally building a language can lead to inconsistencies**
- **Engineering trade-offs are taken**
- **Users can get confused by sophistication of the type system**

- **If you think it is easy, try yourself, but understand theory behind!**
- **IMHO, there is no perfect type system for a programming language**

C++ type system weaknesses

Everything inherited from C
Backward compatibility
Enum oddities

C's Type System heritage: Weak and weaknesses

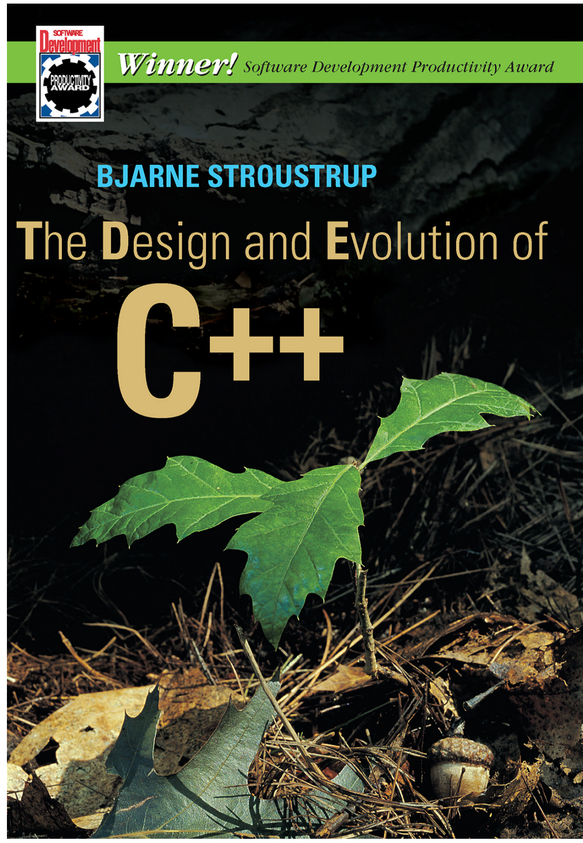
guidelines and static analysis tools, e.g., for safety will steer you away from these weak areas

- **(mostly) Arbitrary Casts reinterpreting memory**
- **Type Punning via casts from to (void*)**
- **char* convention**
- **Integral Promotion**
- **Implicit and Lossy Numeric Conversion**
- **Array-to-Pointer Decay**
- **typedef is not defining a new type**
- **Implementation-defined type size and signed-ness**
- **Undefined Behavior on "normal" arithmetic**
- ~~**no type deduction - no overload resolution possible - no user-defined operators**~~

```
ASSERT_EQUAL(0.0,sin(false));
ASSERT_EQUAL_DELTA(1.0,sin(true),0.2);
ASSERT_EQUAL(true,bool(sin(true)));
```

- **classes make a "strong" type system (BS - D&E)**
- **"I observed that type errors almost invariably reflected either a silly programming error or a conceptual flaw in the design." (BS - D&E)**
  - PS: "every cast is a indicator for a design improvement waiting"
- **"There are also facilities, such as explicit unchecked type conversions, for deliberately breaking the type system." (BS - D&E)**
- **Except for a few cases, almost all annoying flaws in the C++ type system are due to backward compatibility.**

C++'s Type System
minus
C's Type System
= strong type system



Some of the Problems with **primitive** built-in types

```
ASSERT_EQUAL(0.0,sin(false));
ASSERT_EQUAL_DELTA(1.0,sin(true),0.2);
ASSERT_EQUAL(true,bool(sin(true)));
```

- **Integral promotion (inherited from C)**
  - with very interesting rules no one can remember correctly, including bool and char as integer types
  - signed - unsigned mixtures in arithmetic
  - silent wrapping vs. undefined behavior on overflow, vs. signaling of overflow (want the carry bit!)
- **Automatic (numeric) conversions**
  - integers <-> floating points <-> bool
  - and that complicated with types with non-explicit constructors and conversion operators
- **Special values for floating point numbers**
  - +Inf, -Inf, NaN (often forgotten)

warnings often silenced with arbitrary casts

Do not make your class types implicitly convert!

Make comparison strict weak order or stronger!

Legacy: unqualified member function binding to temporary

value category member qualifier	const lvalue	xvalue (temporary)	lvalue	effect on *this
none	❌	✅	✅	side effect
const	✅	✅	✅	no change
const &	✅	✅	✅	no change
&	❌	❌	✅	side effect
&&	❌	✅	❌	consume guts
~~const &&~~		const(✅)		~~no change~~

old and new member function qualifiers do not mix well,
still mostly old style is used when overloads exist

- **Enumeration types are odd**
  - allow operator overloading, but not all useful operators
  - allow limitation of values, but no constructors or assignment to enforce constraints
  - can have implementation-defined underlying type (mitigation possible)
  - can spill enclosing scope with identifiers (mitigation possible)
- **IMHO it would be nice to have enum (class) types real types with**
  - constructors checking for validity
  - assignment operators ditto
  - (mutating) member functions, i.e., for bitmask enum types ensuring valid enum values
    - i.e. operator++ that wraps around with corresponding operator+=
    - bit operator with consistent combined assignment operator| and operator|=

How to better employ the C++ type system?

Use "strong-typed" wrappers!

Why not just "units" frameworks?

Where the standard library fails today!

C++ built-in numeric types are not good for abstracting domain values

- **multiple parameters of same type**
  - can not easily distinguish
  - people call for "named parameters"
    - IMHO: wrong approach for C++
- **type aliases do not help**
  - different for humans, not for compiler
- **Every domain or application has specific computations and thus specific types used**
  - often mapped to integers or floating point numbers without thinking
  - education problem (abstraction)
  - considered a performance issue

```
double consumption(double l, double km) {
  return l/(km/100.0);
}

void testConsumption1over1(){
  double const l { 1 } ;
  double const km { 1 } ;
  ASSERT_EQUAL(100.0, consumption(l, km));
}

void testConsumption40over500(){
  double const l { 40 };
  double const km { 500 };
  ASSERT_EQUAL(8.0, consumption(l, km));
}

void testConsumption40over500Wrong(){
  double const l { 40 };
  double const km { 500 };
  ASSERT_EQUAL(8.0, consumption(km, l)); // no check.
}
void testConsumptionEvenMoreStrange(){
  ASSERT_EQUAL(8.0,consumption(consumption(40,500),100));
}
```

```
consumption(km, l) or consumption(l,km)
```

Whenever you have a function taking multiple arguments of the same type,

it will be called wrongly!

Whole Value Pattern (Ward Cunningham - CHECKS Pattern Language)

- *When parameterizing or otherwise quantifying a business (domain) model there remains an overwhelming desire to express these parameters in the most fundamental units of computation.*
  - *Not only is this no longer necessary (it was standard practice in languages with weak or no abstraction), it actually interferes with smooth and proper communication between the parts of your program and with its users.*
  - *Because bits, strings and numbers can be used to **represent almost anything**, any one in isolation **means** almost **nothing**.*
- **Therefore:**
- *Construct specialized values to quantify your domain model and use these values as the **arguments** of their messages and as the units of input and output.*

  Value Types
  - *Make sure these objects capture the whole quantity with all its implications beyond merely magnitude, but, keep them independent of any particular domain.*

    functions, operators
  - *Include format converters in your user-interface that can correctly and reliably construct these objects on input and print them on output.*

    constructors, I/O
  - ***Do not expect your domain model to handle string or numeric representations of the same information.***

    no implicit conversions

Do not use primitive types (int) or generic representation types (string) for your domain values!

```
double consumption(double l, double km) {
  return l/(km/100.0);
}
```

Apply "simplest strong" type wrapper: struct

- **simple aggregate types**
  - one for each domain value
  - no encapsulation
  - as efficient as normal
- **even works in C**
  - with slightly more elaborate spelling
  - structs can be passed by value
  - as efficient as primitive types
- **Disclaimer:**
  - ABI might not allow same efficiency

```
struct literGas{
  double l;
};
struct kmDriven{
  double km;
};
struct literPer100km {
  double consumption;
};

literPer100km consumption(literGas l, kmDriven km) {
  return {l.l/(km.km/100.0)}; // needs curlies
}
void testConsumption40over500(){
  literGas const l { 40 };
  kmDriven const km { 500 };
  ASSERT_EQUAL(8.0,consumption(l,km).consumption);
}

void testConsumption40over500Wrong(){
  literGas const l { 40 };
  kmDriven const km { 500 };
  ASSERT_EQUAL(8.0,consumption(km,l).consumption);
}
```

error: no matching function for call to 'consumption'

Simple Type Wrappers produce identical code

C++ source #1

Save/Load + Add new... CppInsights C++

```
1  struct literGas{
2      double l;
3  };
4  struct kmDriven{
5      double km;
6  };
7  struct literPer100km {
8      double consumption;
9  };
10
11 literPer100km consumption(literGas l, kmDriven km) {
12     return {l.l/(km.km/100.0)}; // needs curlies
13 }
14
```

C++ source #2

Save/Load + Add new... CppInsights C++

```
1
2
3
4
5
6
7
8
9
10
11 double consumption(double l, double km) {
12     return l/(km/100.0);
13 }
14
```

x86-64 clang 8.0.0 (Editor #1, Compiler #1) C++

x86-64 clang 8.0.0 -O3 -Wextra -Wall -Werror

11010 .LX0: lib.f: .text // \s+ Intel Demangle

```
1  .LCPI0_0:
2          .quad   4636737291354636288     # double 100
3  consumption(literGas, kmDriven):        # @consumption(literGas, kmD
4          divsd   xmm1, qword ptr [rip + .LCPI0_0]
5          divsd   xmm0, xmm1
6          ret
```

Output (0/0) x86-64 clang 8.0.0 - 1191ms (11692B)

x86-64 clang 8.0.0 (Editor #2, Compiler #2) C++

x86-64 clang 8.0.0 -O3 -Wextra -Wall -Werror

11010 .LX0: lib.f: .text // \s+ Intel Demangle

```
1  .LCPI0_0:
2          .quad   4636737291354636288     # double 100
3  consumption(double, double):                        # @consumption(
4          divsd   xmm1, qword ptr [rip + .LCPI0_0]
5          divsd   xmm0, xmm1
6          ret
```

Output (0/0) x86-64 clang 8.0.0 - 889ms (7877B)

https://godbolt.org/z/d-H6dm

Identical code, even in C

C source #1

```
typedef
struct literGas{
    double l;
} literGas;
typedef
struct kmDriven{
    double km;
} kmDriven;
typedef
struct literPer100km {
    double consumption;
} literPer100km;

literPer100km consumption(literGas l, kmDriven km) {
    literPer100km res = {1.1/(km.km/100.0)};
    return res;
}
```

C source #2

https://godbolt.org/z/-pAhZO

```
double consumption(double l, double km) {
    return l/(km/100.0);
}
```

x86-64 clang 8.0.0 (Editor #1, Compiler #1) C++

```
.LCPI0_0:
        .quad   4636737291354636288     # double 100
consumption(literGas, kmDriven):        # @consumption(literGas, kmD
        divsd   xmm1, qword ptr [rip + .LCPI0_0]
        divsd   xmm0, xmm1
        ret
```

Output (0/0) x86-64 clang 8.0.0 - 732ms (14348B)

x86-64 clang 8.0.0 (Editor #2, Compiler #2) C++

```
.LCPI0_0:
        .quad   4636737291354636288     # double 100
consumption(double, double):                    # @consumption(
        divsd   xmm1, qword ptr [rip + .LCPI0_0]
        divsd   xmm0, xmm1
        ret
```

Output (0/0) x86-64 clang 8.0.0 - cached (7880B)

Why not just a units library?

```
#include <iostream>
#include <phys/units/quantity.hpp>
#include <phys/units/io.hpp>
int main(){
  using namespace phys::units; // types
  using namespace phys::units::io; // operator <<(c
  std::cout << "Enter km driven:" << std::endl;
  double x{};
  std::cin >> x;
  quantity<length_d> dist = x * kilo * meter;
  std::cout << "Enter liters:" << std::endl;
  double y{};
  std::cin >> y;
  quantity<volume_d> vol = y * liter;
  std::cout << "You used " << 100 * y / x << " liters per 100 km\n";
  auto res = vol/dist;
  std::cout << "You used " << res << "\n"; // square meters, what?
}
```

```
Enter km driven:
500
Enter liters:
42
You used 8.4 liters per 100 km
You used 8.4e-08 m+2
```

- **Standard library is guilty of using built-ins as type aliases where they do not fit well**
  - size_t, size_type --> count elements = natural numbers including 0 - **absolute value**
  - ptrdiff_t, difference_type -> distance in contiguous sequences, difference between counts! - **relative value**

```
  size_type __n = std::distance(__first, __last); // implicit conversion to unsigned
   if (capacity() - size() ≥ __n) // aha to avoid warning in comparison
    {
std::copy_backward(__position, end(),
        this→_M_impl._M_finish
        + difference_type(__n)); // cast to the real thing again
std::copy(__first, __last, __position);
this→_M_impl._M_finish += difference_type(__n); // and cast again!
    }
```

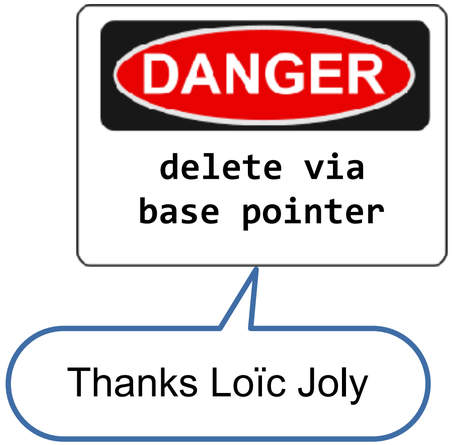
warnings often silenced with arbitrary casts

every cast is a indicator for a design improvement waiting

PSSST
Peters super simple strong typing framework
PS' simple strong typing framework
P. Sommerlad'S Strong typing framework

a proof of concept and work in progress

- **Just one member value**
  - wrap primitive types in a simple way
- **Easy Mix-in/Opt-in for operators on the domain**
  - without hindering adding DIY functionality
  - employ empty-base-class optimization (-> interesting limitations!)
- **Pretend that an aggregate is good enough**
  - C++17 extended the definition of aggregates
  - C++17 provides structured bindings to generically access the members of an aggregate
  - play with C++17 features
- **As simple code for users as possible**



- **CRTP base class**
  - provide value type and self
- **Operator-mix ins**
  - simplify specifying many with ops<>
  - Eq = Equality comparison
  - Out = generic operator>>
- **Parameterized operations tricky**
  - generically doable but noisy
  - ::template apply

```
struct kmDriven:strong<double,kmDriven>
,ScalarMultImpl<kmDriven,double>{};
```

```
using namespace Pssst;
struct literGas:strong<double,literGas>{};

struct literPer100km:strong<double,literPer100km>
,ops<literPer100km,Eq,Out>{};

struct kmDriven:strong<double,kmDriven>
,ops<kmDriven, ScalarMult<double>::template apply>{};

literPer100km consumption(literGas l, kmDriven km) {
  return {l.val/(km/100.0).val};
}

void testConsumption1over1(){
  literGas const l {1} ;
  kmDriven const km { 1 } ;
  ASSERT_EQUAL(literPer100km{100.0},consumption(l,km));
}
void testConsumption40over500(){
  literGas const l { 40 };
  kmDriven const km { 500 };
  ASSERT_EQUAL(literPer100km{8.0},consumption(l,km));
}
```

- **strong wrapper for CRTP**
  - needs to be first base to make EBO and single-brace-init work
  - can not merge the ops-mix-in mechanic here, because this would mean base-class initializers would rely on incomplete type
  - protect against storing references etc
  - naming convention for
    - value_type
    - val
- **all operator mix ins with CRTP arg**
  - ops applies this via inheritance
- **Mechanics can made work also with private member**

```
// apply multiple operator mix-ins
template <typename U,
  template <typename ...> class ...BS>
struct ops:BS<U>...{};

// strong first base
template <typename V, typename TAG>
struct strong {
  static_assert(std::is_object_v<V>);
  using value_type=V;
  V val;
};

// bind 2nd template argument
template<typename B,
template<typename...>class T>
struct bind2{
  template<typename A, typename ...C>
  using apply=T<A,B,C...>;
};
```

- **First step: Eq equality comparison**
- **Parameterized Boolean type?**
  - allow a strong Bool that does not participate in integral promotion!
- **structured binding to generically pick single value member**
  - requires == defined for member
  - could have used pass by value, but wanted to make sure it works well even with things like std::string as value_type
- **Out for generic value output**
  - together we can write Tests now

```
template <typename U, typename Bool=bool>
struct Eq{
  friend constexpr Bool
  operator==(U const &l, U const& r) noexcept {
    auto const &[vl]=l;
    auto const &[vr]=r;
    return {vl == vr};
  }
  friend constexpr Bool
  operator!=(U const &l, U const& r) noexcept {
    return !(l==r);
  }
};
template <typename U>
struct Out {
  friend std::ostream&
  operator<<(std::ostream &l, U const &r) {
    auto const &[v]=r;
    return l << v;
  }
};
```

- **"Eat your own dog food"**
- **bool is problematic**
  - but allows short-cut evaluation
- **Bool is type**
  - conversion operator allows use in bool contexts
- **Boolean are operations mix-ins**
  - no shortcut eval, but no side effect would be useful here

```
struct Bool:strong<bool,Bool>,ops<Bool,Boolean>{
  constexpr explicit operator bool() const {
    return val;
  }
};
template <typename U>
using CmpB = Order<U,Bool>;
template <typename U>
using EqB = Eq<U,Bool>;
```

comparison mix-ins with new Bool type

```
template <typename B>
struct Boolean {
  friend constexpr B
  operator || (B const &l, B const &r){
    auto const &[vl]=l;
    auto const &[vr]=r;
    return {vl || vr};
  }
  friend constexpr B
  operator && (B const &l, B const &r){
    auto const &[vl]=l;
    auto const &[vr]=r;
    return {vl && vr};
  }
  friend constexpr B
  operator !(B const &l){
    auto const &[vl]=l;
    return {! vl};
  }
};
```

Units beware: relative vs. absolute - often misunderstood or easily misapplied

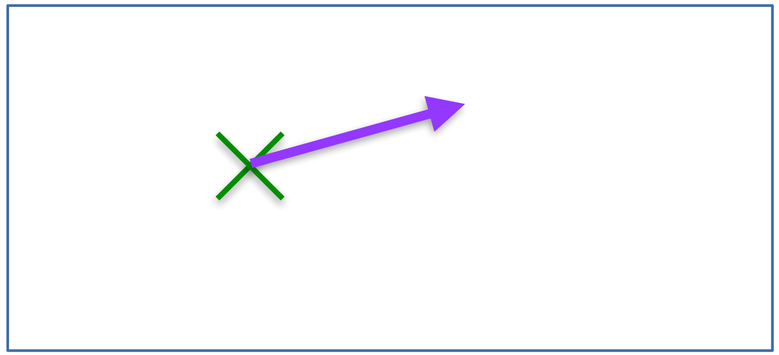
relative

relative

wrong result type!

```
    size_type __n = std::distance(__first, __last); // implicit conversion to unsigned
      if (capacity() - size() ≥ __n) // aha to avoid warning in comparison
       {
std::copy_backward(__position, end(),
            this→_M_impl._M_finish
            + difference_type(__n)); // cast to the real thing again
std::copy(__first, __last, __position);
this→_M_impl._M_finish += difference_type(__n); // and cast again!
```

- **<chrono> is a good example to follow:**
  - time_point and duration: tp1 - tp2 -> duration, tp + d -> time_point, **tp+tp -> nonsense**, d1 + d2 -> duration
- **position vs. direction (vector space vs affine space)**
  - Vec3d/Vec3 and similar are problematic, because identical representation is used for both roles
  - location and displacement
- **generic units must make this distinction**
  - easily forgotten in dimensional analysis



What operations to provide for 1d affine space?

- **Additive (+,-, unary +,-,++,--)**
- **Scalar Multiplication**
  - no safety guarantee (yet)
  - only provide % for integral bases
- **Equality**
- **Order**
- **On application we need to provide scalar base through bind2**
  - that is what `::template apply` is about

```
template <typename V>
using Additive=ops<V,UPlus,UMinus,Add,Sub,Inc,Dec>;
template <typename V, typename BASE>
using Affine=ops<V,Additive,
  ScalarMult<BASE>::template apply,Eq,Order>;
```

```
template <typename R, typename BASE>
struct ScalarMultImpl :
ScalarModulo<R,BASE,std::is_integral_v<BASE>> {
  friend constexpr R&
  operator*=(R& l, BASE const &r) noexcept {
    auto &[vl]=l;
    vl *= r;
    return l;
  }
  friend constexpr R
  operator*(R l, BASE const &r) noexcept {
    return l*=r;
  }
  friend constexpr R
  operator*(BASE const & l, R r) noexcept {
    return r*=l;
  }
  friend constexpr R&
  operator/=(R& l, BASE const &r) noexcept {
    auto &[vl]=l;
    // need to check if r is 0 and handle error
    vl /= r; // times 1/r could be more efficient
    return l;
  }
  friend constexpr R
  operator/(R l, BASE const &r) noexcept {
    return l/=r;
  }
};
template<typename BASE>
using ScalarMult=bind2<BASE,ScalarMultImpl>;
```

- **Need an Origin**
  - defaults to zero
    - tricky mechanics to obtain underlying value type not shown
  - tricky mechanic via function object type
    - because when instantiated types can still be incomplete, need delegation
- **Delegates operations to affine space**
  - additive with affine space values
  - subtracting vector space values results in affine space value

```
template <typename ME, typename AFFINE,
          typename ZEROFUNC=default_zero<AFFINE>>
struct create_vector_space {
   using affine_space=AFFINE;
   using vector_space=ME;
   static inline constexpr affine_space origin=ZEROFUNC{}();
   affine_space offset;
   friend constexpr vector_space&
   operator+=(vector_space& l, affine_space const &r) noexcept {
      l.offset += r;
      return l;
   }
   friend constexpr vector_space
   operator+(vector_space l, affine_space const &r) noexcept {
      return l += r;
   }
   friend constexpr vector_space
   operator+(affine_space const & l, vector_space r) noexcept {
      return r += l;
   }
   friend constexpr vector_space&
   operator-=(vector_space& l, affine_space const &r) noexcept {
      l.offset -= r;
      return l;
   }
   friend constexpr vector_space
   operator-(vector_space l, affine_space const &r) noexcept {
      return l -= r;
   }
   // vs - vs = as
   friend constexpr affine_space
   operator-(vector_space const &l,vector_space const &r){
      return l.offset - r.offset;
   }
};
```

- **Kelvin**
  - zero is origin
  - provide Equality, Order and Output for tests
- **Celsius**
  - 273.15 degrees (K) is origin
  - ditto for Kelvin
- **generic conversion possible with same affine space:**

```
// must be vector spaces from same affine space
template<typename TO, typename FROM>
constexpr TO convertTo(FROM from) noexcept{
  static_assert(std::is_same_v<
      typename FROM::affine_space
      ,typename TO::affine_space>);
  return {(from.offset-(TO::origin-FROM::origin))};
}
```

```
// affine space: degrees (K and C)
struct degrees:strong<double,degrees>,
Affine<degrees,double>, ops<degrees,Out>{};

struct Kelvin:create_vector_space<Kelvin,degrees>
             ,ops<Kelvin,Eq,Order,Out>{};

struct CelsiusZero{
  constexpr degrees operator()() const noexcept{
    return {273.15};
  }
};

struct
Celsius:create_vector_space<Celsius,degrees,CelsiusZero>
              , ops<Celsius,Eq,Order,Out>{};

void thisIsADegreesTest() {
  degrees hotter{20};
  Celsius spring{15};
  auto x = spring+hotter;
  ASSERT_EQUAL(Celsius{35},x);
}
void testConversion(){
  Celsius mild{20};
  Kelvin k{convertTo<Kelvin>(mild)};
  ASSERT_EQUAL(Kelvin{293.15},k);
  ASSERT_EQUAL(mild, convertTo<Celsius>(k));
}
```

PSSST: no overhead, but more safety!

C++ source #1

```
#include "https://raw.githubusercontent.com/PeterSommerlad/Psst/master/cevelop-workspace/
using namespace Pssst;
struct literGas:strong<double,literGas>{};

struct literPer100km:strong<double,literPer100km>
,ops<literPer100km,Eq,Out>{};

//struct kmDriven0:strong<double,kmDriven0>
//,ops<kmDriven0, ScalarMult<double>::template apply>{};

struct kmDriven:strong<double,kmDriven>
,ScalarMultImpl<kmDriven,double>{};

static_assert(sizeof(double)==sizeof(kmDriven));

literPer100km consumption(literGas l, kmDriven km) {
    return {l.val/(km/100.0).val};
}
```

x86-64 gcc 9.1 (Editor #1, Compiler #1) C++

x86-64 gcc 9.1 | -O3 -Wextra -Wall -Werror -std=c++17 -Wno-missing-field-initializers

```
consumption(literGas, kmDriven):
        divsd   xmm1, QWORD PTR .LC0[rip]
        divsd   xmm0, xmm1
        ret
.LC0:
        .long   0
        .long   1079574528
```

Output (0/0) x86-64 gcc 9.1 - 1812ms (346301B)

C++ source #2

```
double consumption(double l, double km) {
    return l/(km/100.0);
}
```

x86-64 gcc 9.1 (Editor #2, Compiler #2) C++

x86-64 gcc 9.1 | -O3 -Wextra -Wall -Werror

```
consumption(double, double):
        divsd   xmm1, QWORD PTR .LC0[rip]
        divsd   xmm0, xmm1
        ret
.LC0:
        .long   0
        .long   1079574528
```

Output (0/0) x86-64 gcc 9.1 - 645ms (3464B)

https://godbolt.org/z/HujfT2

- **More tests for PSSST and experiments needed (LIW ?)**
  - integrate Safe Numerics or similar for safety?
  - MSVC seems to be bad in optimizing aggregates (see godbolt.org)
- **Learn to embrace your language's type system!**
  - Every type cast is a potential design issue!
  - Apply the Whole Value Pattern
  - Check out Ward Cunningham's CHECKS pattern language for more inspiration
- **Use strong type wrappers with care, but use them or DIY!**
  - regardless who's (Joe Boccara, Björn Fahller, Jonathan Müller, Boost)
- **Do not confuse a units-framework with strong type wrapper framework**
  - they serve similar but different purposes, many domains need their own units!

Questions?

peter.sommerlad@hsr.ch
@PeterSommerlad

By the way, thanks for the great piece of software! This is by far the best free IDE for C/C++ so far after trying basically all the free C/C++ IDE. motowizlee on github 27.04.2017