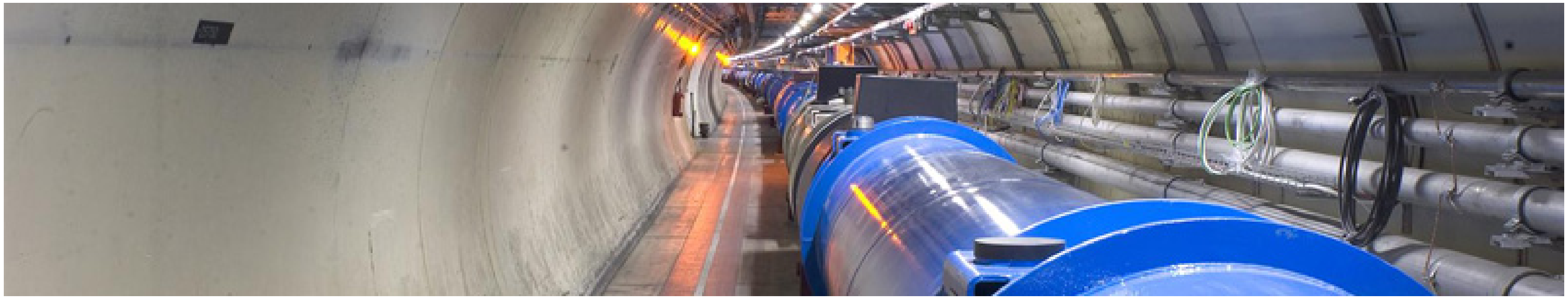
Concepts: The Future of Generic Programming (the future is here)

Bjarne Stroustrup

Morgan Stanley and Columbia University

www.stroustrup.com



Overview

- Generic programming, templates, and concepts
  - Aims and Status
  - Concepts and types
- Concept benefits
  - Overloading, readability, design
- What is a concept?
  - What makes a good concept?
- Defining concepts
  - How to use concepts well

Generic Programming

- David *Musser and Alex Stepanov (1989)*
  - Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software.

- My aims for C++
  - Make simple generic code as simple as non-generic code
  - Make more advanced generic code as easy to use and not that much more difficult to write
  - Not just for foundation libraries

Write better code!

- Cleaner
- Simpler
- More readable
- More maintainable
- Faster
- Less clever
- More general
- More usable and re-usable
- Type safe

Concepts can be a significant help

This is not a talk about language-technical details

Loop for more specialized concepts talks on the program

Concepts support status

- Concepts TS approved 2016
  - Available in GCC since GCC 6 (soon: Clang)
- Concepts in WP for C++20
  - Explicit **requires** clauses

    ```
    template<typename Iter> requires RandomAccesIterator<Iter> void sort(Iter,Iter);
    ```
  - Shorthand notation

    ```
    template<RandomAccesIterator Iter> void sort(Iter,Iter);
    ```
  - **requires** expressions
  - Basic concepts for Ranges in standard library
- Not quite yet
  - Ranges (defined using concepts) in standard library (soon)
  - A more function declaration like syntax (compromise being worked out)
  - Concept type-name introducers (won't make C++20)

GP is “just” programming

- My aims
  - Make simple generic code as simple as non-generic code
  - Make more advanced generic code as easy to use and not that much more difficult to write
  - Not just for foundation libraries
- Implies
  - Better type checking
    - concepts
  - Better syntax
    - shorter, more conventional
  - More similar organization of generic and “ordinary” code
    - Modules (not this talk)

Morgan Stanley

Generic programming: Templates

- 1980-1990
  - Use macros to express generic types and functions
- 1987-now; aims
  - Extremely general/flexible
    - "must be able to do much more than I can imagine"
  - Zero-overhead
    - Vector, Matrix, … to compete with C arrays
  - Well-specified interfaces
    - Implying overloading, good error messages, and maybe separate compilation
- 1994-now
  - Unconstrained templates
  - Header-only code organization
- 2003-now; aims
  - Concepts to precisely specify interfaces

Design principles:

- generality
- Zero-overhead
- Provide good interfaces

1978 Type checking

- Unacceptable! (K&R C)

```
double sqrt();              // sqrt may take some arguments of some types

double d1 = sqrt(2);            // run-time crash
double d2 = sqrt("two");        // run-time crash

double sqrt(x) double x;
{
        // double uses one argument of type double
}
```

1978 Type checking

- 1979 response (C with Classes)

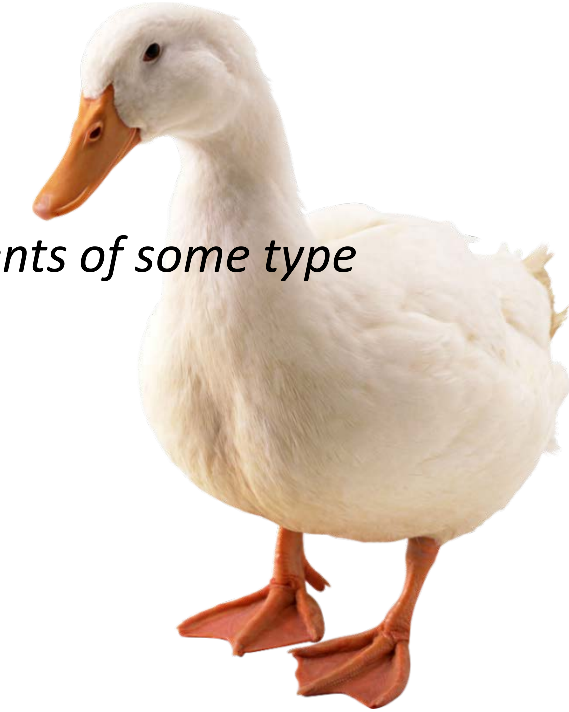
```
double sqrt(double);   // function argument specification and checking

double d1 = sqrt(2);           // correct answer
double d2 = sqrt ("two");      // compile-time error

double sqrt(double x)          // note: new (and uniform) syntax
{
        //  use x
}
```

Many initial responses were negative!
(too radical, not optional, not compatible)

- Immense improvement
  - Readability, maintenance, safe linking
  - Opened the door for overloading (and through that, generic programming)

1988 Type checking

- Acceptable? (no!)



```
template<class Iter>
void sort(Iter first, Iter last)      // takes two arguments of some type
{
        // uses the arguments as iterators
}

vector<int> vi;
list<int> lst;
vector<S> vs;   // S is struct S { int m; };

sort(vi.begin(),vi.end());           // OK
sort(lst.begin(),lst.end());         // Error: obscure error message
                                     // referring to the implementation
sort(vs.begin(),vs.end());           // Error: another obscure error message
```

Templates: A massive success

- **Because** of their great utility
  - Flexibility (Turing complete)
  - Type safety (better late detection than no detection)
  - Specialization (handle irregular types, template metaprogramming, and type-based optimizations)
  - Resulting run-time performance
- ***Despite*** their great flaws
  - Verbose syntax
  - Duck typing
  - Spectacularly bad error messages
  - Clumsy overloading
  - Weak/odd code organization
  - Slow compilation

Design aim:
Address all major flaws

Specify intent

2018 Type checking

- Precisely specify interfaces (using the concepts TS)

```
void sort(Sortable& c);   // Sortable has a random access sequence with <
```

A concept
specifying what it means to be sortable

```
sort(vi);       // OK
sort(lst);      // error: list<int> isn't Sortable (no [])
sort(vs);       // error: vector<S> isn't Sortable (no < for value type)

void sort(Sortable&  c)   // Sortable has a random access sequence with <
{
        std::sort(begin(c),end(c));
}
```

2018 Type checking

- Using the concepts TS

```
void sort(Sortable& c)   // a Sortable has a random-access sequence with <
{
        std::sort(begin(c),end(c));
}

void sort(List& c)       // a List has a sequence with <
{
        vector v {begin(c),end(c)};      // copy c into v (deduce element type)
        sort(v);                         // sort
        copy(v,c);                       // copy c back again (Ranges)
}

sort(vi);       // OK
sort(lst);      // OK
sort(vs);       // error: vector<S> isn't Sortable (no < for value type)
```

2018 Type checking

- Using the concepts TS

```
void sort(Sortable& c)   // a Sortable has a random-access sequence with <
{
        std::sort(begin(c),end(c));
}

void sort(List& c)            // a List has a sequence with <
{
        vector v {c};          // copy c into v (deduce element type) – soon I hope
        sort(v);               // sort
        copy(v,c);             // copy c back again (Ranges)
}

sort(vi);         // OK
sort(lst);        // OK
sort(vs);         // error: vector<S> isn't Sortable (no < for value type)
```

Types and concepts

- A type
  - Specifies the set of operations that can be applied to an object
    - Implicitly and explicitly
    - Relies on function declarations and language rules
  - Specifies how an object is laid out in memory
- A concept
  - Specifies the set of operations that can be applied to an object
    - Implicitly and explicitly
    - Relies on use patterns
      - reflecting function declarations and language rules
  - Says nothing about the layout of the object

My ideal: to be able use concepts wherever we use a type, in the same way,
Except for defining layout

Types and concepts

- They are very similar

```
template<typename T> concept Int = Same<T,int>;

Int x1 = 7;
int x2 = 9;
Int y1 = x1+x2;
int y2 = x2+x1;

void f(int&);        // a function
void f(Int&);        // a function template

void ff()
{
      f(x);
      f(y);
}
```

Not just for functions and classes

Technical issue

- Immovable opposition to the natural/conventional syntax in WG21
  - **void sort(Sortable&);** **//** *deemed confusing and error prone*
  - **void sort(Sortable auto&);** **//** *deemed much better by some*
- I don't see it
  - I have used and taught concepts for years
  - **void sort(Sortable&&);** **//** Key "anti" example: *rvalue or forward reference?*
- I guess I can live with
  - **void sort(Sortable&&);** **//** *error*
  - **void sort(Sortable auto&&);** **//** *forward reference*
  - But it breaks the equivalence between types and concepts

Concept benefits

Andrew Sutton, initial implementer

Concept benefits

- Support good design
  - Like classes did/does
- Readability, maintainability
  - Overuse of **auto** (unconstrained types) is a problem
- Overloading
  - Like functions, but much simpler

- Remember when we just had built-in types and no classes?
  - No, never in C++
  - Like C today
    - At least C has function prototypes

Morgan Stanley

Overloading

Design principles:

- Don’t force the user to do what a machine does better
- Zero overhead compared to unconstrained templates

- Overloading based on predicates

```
template<ForwardIterator Iter>
    void advance(Iter& i, int n) { while (n--) ++i; }

template<RandomAccessIterator Iter>
    void advance(Iter& i, int n) { i += n; }

void user(vector<string>& vs, list<string>& ls)
{
    auto pvs = vs.begin();   advance(pvs,2);   // use fast advance
    auto pls = ls.begin();   advance(pls,2);   // use slow advance
}
```

- We compute relationships among concepts, e.g.,

```
Input_iterator < Random_access_iterator        // no need to specify
```

Concepts simplifies design

- Overloading
  - Fundamental to C++ generic programming
- Fewer Traits
  - Many can be replaced by functions overloaded on concepts
  - Most can be made into implementation details
  - Note the **advance()** example used
    - No trait
    - No helper functions
- Conditional properties
  - Many **enable_if**s can be replaced by concepts

- Concepts improve compile times
  - Compared to workarounds

Concepts simplifies design

- Conditional properties
  - **enable_if** is primitive and leads to very ugly complicate code
  - Concepts provide simple, elegant expression of conditions

```
template<typename T> class Ptr {
	// ...
	T* operator->() requires Is_class<T>;		// offer -> (only) if T is a class
};

template<typename T, typename U> class Pair {
	// ...
	template<class TT, class UU>
	   // offer constructor (only) for types that can be converted to the members
	   requires Convertible<TT,T> && Convertible<UU,U>
	Pair(const TT&, const UU&);
};
```

Workarounds do not scale

- The simplest **enable_if** workaround for the simplest example

```
template<typename T>
class Ptr {
    // ...

template<typename U>
    std::enable_if_t<is_class_v<U>,T*> operator->();

    // ...
};
```

- Sometimes you need both **enable_if<pred>** and **enable_if<!pred>**
- Sometimes there are two or more predicates to select on
- Some operations, notably constructors, do not have a simple syntactic place for **enable_if**.

Readability

- "every new useful feature will be misused and overused"
  - **auto**
- Observed problem *(slow reading, lots of browsing, errors)*
  - **auto ch = foobar(x,y);**
  - **if (auto ch = foobar(x,y)) ...**
- Response *(add comments everywhere **auto** is used in non-obvious ways)*
  - **auto ch = foobar(x,y);** **//** foobar() returns an input channel
  - **if (auto ch = foobar(x,y) /*** ch must be an input channel ***/) ...**
  - Comments: compilers don't read them, are imprecise, distract
- Constrain with concepts
  - **Input_channel ch = foobar(x,y);**
  - **if (Input_channel ch = foobar(x,y)) ...**

Readability

- Why?

```
template<typename InputIterator, typename Value>
InputIterator find(InputIterator first, InputIterator last, Value val)
{
    // ...
}
```

- History
  - In 1987, templates were so new and scary to many that they insisted on a “loud” syntax with a prefix keyword
    - When scared, (some) people always ask for (demand) “loud” syntax
    - Later, people complain about verbosity and clumsiness

Be careful what you ask for;
you might get it

Readability

- With concepts we can read declarations
  - And documentation and comments

```
template<InputIterator Iter, typename Value>
        requires EqualityComparable<Value_type<Iter>,Value>
Iter find(Iter first, Iter last, Value val);
```

Don’t confuse familiarity with simplicity

- With auto/typename we must read implementations
  - And documentation and comments
  - And pay close attention to naming

```
template<typename InputIterator, typename Value>
InputIterator find(InputIterator first, InputIterator last, Value val)
{
        // ... use InputIterator as an input iterator ...
        // ... compare *first to value using == ...
}
```

Readability

- Sequences expressed as pairs of iterators

```
template<InputIterator Iter, typename Value>
        requires EqualityComparable<Value_type<Iter>,Value>
Iter find(Iter first, Iter last, Value val);
```

Make simple things simple

- Ranges express sequences directly
  - "the whole sequence" is the simplest and most common case

```
template<InputRange Rng, typename Value>
        requires EqualityComparable<Value_type<Rng>,Value>
safe_iterator_t<Rng> find(Rng r, Value val)
{
        // ...
}
```

Readability

- Don’t expect optimal readability from
  - Older libraries converted to use concepts
    - They often need to be “bug compatible”
  - “Advanced foundation libraries”
    - They often have to offer extreme flexibility
- Design new libraries with readability in mind

Typed vs. untyped styles

- **auto** is the weakest concept
  - An unconstrained type
- In theory we could do without **auto** as a language construct
  - **template<typename T> concept Auto = true; //** *Auto means auto*
- Ideal:
  - Accept a concept wherever an **auto** is
  - Actually, that's backwards: Accept an **auto** wherever a concept is
    - More historically accurate
    - The committee accepted **auto** before concepts
- Historical factoid
  - I proposed **auto f(auto);** in 2003

Typed vs. untyped styles

- Generally, we prefer to rely on types
- Types *(generally, when used well)*
  - document intended use
  - improves readability
  - enable overloading
  - help catch errors
  - help the compiler write good error messages
  - help optimizers
- Overly general types cause problems
  - **void* p;** // *p can point to any object*
  - **template<typename T>** // *T can be any type*
  - **[](auto x) { … }** // *x can be of any type*

My hope/expectation

- Concepts will change the way we think about
  - Programming
    - Not just about generic programming
  - Design
  - Interfaces
  - Types
- It's not just another support for "business as usual"
  - It's major
  - I define "major" as "changing the way we think"

This will take years

Individuals can do better
than the community as a whole

Concepts weren't born yesterday

- 1981: Alex Stepanov: "Algebraic structures"
- 1988: My attempts to find a way of constraining template arguments (failed)
- 1994: STL was specified in terms of concepts
- 2003: "Texas" design: use patterns and many alternatives
- 2003:"Indiana" design: functions signatures and initial implementation
- 2006: "Texas" + "Indiana" merger (leading to C++0x concepts and failure)
- 2011: Palo Alto meeting (leading to the Concepts TS and GCC implementation)
- 2012:"Concepts are predicates" design and implementation
- 2017: Concepts TS + GCC implementation + Ranges TS

- C++20: (most in current WP)

What is a concept?

"Concepts are all about semantics"

– Alex Stepanov ("father of the STL")

What is a concept?

- Concepts are compile-time predicates
  - **ForwardIterator<T>**: Is **T** a forward iterator?
  - **EqualityComparable<T,U>**: Can a **T** be compared to a **U** using **==** or **!=**?
- Concepts are fundamental
  - They represent fundamental concepts of an application area
  - Concepts are come in “clusters” describing an application area
    - Monoid, group, field, and ring
    - Input, forward, bidirectional, and random access operators

What is a concept?

- We have always had concepts
  - Every successful generic library has some form of concepts
    - In the designer's head
    - In the documentation
    - In comments
  - C/C++ built-in type concepts: arithmetic and floating
  - STL concepts like iterators, sequences, and containers
  - Mathematical concepts like monad, group, ring, and field
  - Graph concepts like edges and vertices, graph, DAG, ...
- We have added direct language support
  - Making using concepts easier than not using them
- We must learn to use the language support effectively

What makes a concept good?

- Represent a fundamental concept of a domain
  - Good concepts are carefully designed (or discovered)
- A concept is ***not*** the minimal requirements for an implementation
  - An implementation does not define the requirements
  - Requirements should be stable
- Has semantics
  - For a **Number**, the operations (**+**,**-**,*****, and **/**) must obey the usual rules
  - “**HasPlus**” is not a concept
- Good concepts support interoperability
  - There are relatively few good concepts
  - We can remember a good concept

Concepts

- Concepts are ***not*** types of types
  - Single-argument concepts are almost types of objects
- Concepts are not “type classes”
  - Implicit conversions, mixed-type operations, ...
  - not defined in terms of sets of functions, ...
- Most – concepts take more than one argument

```
requires InputIterator<Iter>
        && Assignment Compatible<Value_type<Iter>,Value>
```

- Like template, concepts can take value arguments

```
template<typename B, typename T, Size SZ> concept Buffer =
    Regular<T>                              // T is well-behaved
    && Integer<Size>                        // Size can be used as an integer
    && requires(B b, Size i) { {b[i]} -> T&; };
```

Operations come in "clusters"

- Useful concepts describe "clusters" of operations
  - E.g. algebra: The mathematicians used centuries to work out the few meaningful concepts in this area
    - Monoid, group, ring, field, vector space, …
- An operation typically cannot be defined in isolation
  - Numbers: **+**, **-**, *****, **/**, **+=**, **-=**, **++**, **--**, …
  - Iterators: **++**, **--**, *****, **[ ]**
  - Stacks: **push()**, B
- Only rarely does a concept characterize a single operation
  - "HasPlus" is very suspect
    - is **std::string** a HasPlus?
    - Do we need a separate "HasMinus"? (no way!)
- Just like for types

Ideal: "plug and play"

- Express requirements for algorithms in terms of fundamental and complete concepts
  - ***Not***: "specify the minimal requirements for an function template"
- For example

```
template<ForwardIterator Iter, typename Val>
Val sum(Iter first, Iter last, Val acc)
{
        while (first!=last) {
                acc += *first;
                ++first;
        }
        return acc;
}
```

Incrementable or Number?
Number!

+= or + and =?

Design principle:

- An implementation isn't a specification
- Look for semantic coherence

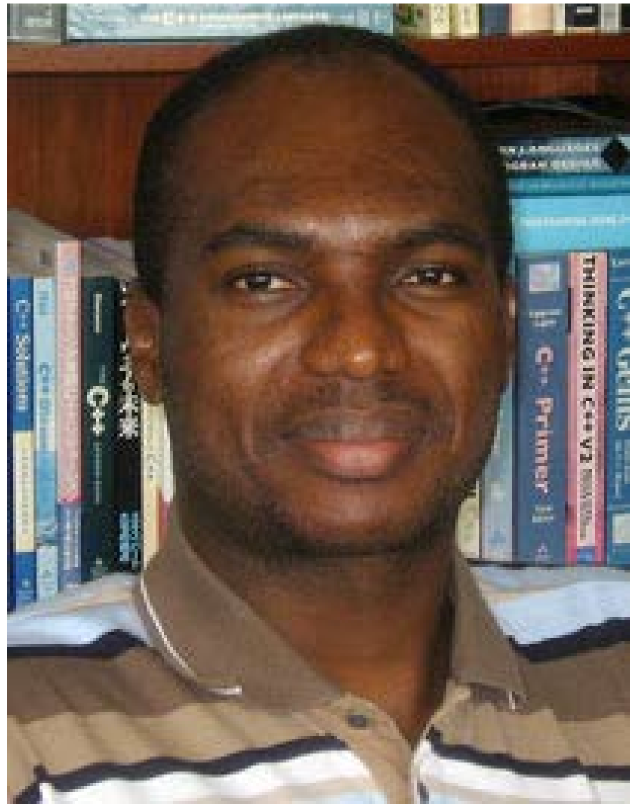
Copyable? Moveable?

Concepts: not just for algorithms

```
template<InputTransport Transport, MessageDecoder MessageAdapter>
class InputChannel {
public:
    using InputMessage =
                MessageAdapter::InputMessage<Transport::InputBuffer>;
    using MessageCallback = function<void(InputMessage&&)>;
    using ErrorCallback = function<void(const error_code&)>;

    template<typename... TransportArgs>
    InputChannel(TransportArgs&&... transportArgs)
          : _transport(std::forward<TransportArgs>(transportArgs)...)
    {}
    // ...
    Transport _transport;
};
```

Slightly simplified

Defining concepts



Gabriel Dos Reis – Designer and experimenter since 2002

Defining concepts

- First, compose from existing concepts

```
template<typename T>     // Build from other concepts
concept Sortable =
    Sequence<T>                      // has begin() and end()
    && Random_access<T>              // has [], +, etc.
    && Comparable<Value_type<T>>;    // has ==, !=,  <, <=, >, and >=
```

- Second, define using requires expressions

```
template<typename T>          // Build from primitive requirements
concept EqualityComparable =
    requires (T a, T b) {
        { a == b } -> bool;       // compare Ts with == return bool
        { a != b } -> bool;       // compare Ts with != return bool
    };
```

Defining concepts: Try for completeness

- Related operations and types

```
template<typename X> using Value_type = X::value_type;
template<typename X> using Iterator_of= X::iterator;
```

Design principle:

- Look for semantic coherence

```
template<typename T>
concept Sequence = requires(T t) {
    typename Value_type<T>;            // must have a value type
    typename Iterator_of<T>;           // must have an iterator type

    { begin(t) } -> Iterator_of<T>;    // must have begin() and end()
    { end(t) } -> Iterator_of<T>;

    requires InputIterator<Iterator_of<T>>;
    requires Same<Value_type<T>,Value_type<Iterator_of<T>>>;
};
```

Defining concepts

- You won’t get concept definitions right the first time
  - Use a library if you can (e.g., from the C++20 WP)
    - **<concepts>**
    - **<ranges>**
    - **<iterators>**
    - **<memory>**
    - …
  - Partial/incomplete concepts are useful
    - During development
    - As building blocks
    - Don’t use them as interfaces in application code

Defining concepts

- Avoid ad-hoc constraints

```
template<typename T> T sum(T& a, T& b)
    requires requires(T a, T b) { {a+b} -> T; };      // misuse

template<typename T> concept Addable = requires {
    { a+b } -> T;
    a+=b;     // can increment
    a=b;      // can copy
    T{0};     // can construct from zero
};

template<Addable T> T sum(T& a, T& b);          // better
```

- **“requires requires”** is usually a design error
  - Leads to incomplete constraints

Accidental match?

- (some) people worry about accidental matches
  - A type matches a concept if it provides the required properties
- Consider a classic bad example

```
template<typename T>            // suspicious: single property concept
concept Drawable = requires(T t) { t.draw(); } ;

class Shape { /* … */ void draw(); /* light up selected pixels on the screen */ };

class Cowboy { /* … */ void draw(); /* pull deadly weapon from holster */ };

template<Drawable D>
void draw_all(vector<D*>& v)  // ye olde draw all shapes example
{
        for (auto& x : v) v->draw();
}
```

Real concepts rarely accidentally match

- But a few do
  - Classical example: input iterator and forward iterator
- If they do
  - Add an operator to allow disambiguation (as done in the Ranges TS)
  - Use a traits class (as currently)
- Beware of single constraint “concepts”
  - HasPlus, HasMinus, incrementable, …
    - beware of the “dreaded *able”s
    - If you need them, let them express more than one constraint
  - Sometimes useful as implementation details for “general use concepts”

Design principles:
- Don’t let the tail wag the dog
- Keep simple things simple

Real concepts rarely accidentally match

- Many operations, related types, semantics

```
template<typename T>
concept Number = requires(T a, T b) {
        { a+b } -> T;
        { a-b } -> T;
        { a*b } -> T;
        { a/b } -> T;
        { -a } -> T;

        { a+=b } -> T&;
        { a-=b } -> T&;
        { a*=b } -> T&;
        { a/=b } -> T&;

        { T{0} };               // can construct a T from a zero

        // ...
};
```

Definition checking

- Concepts won't catch all type errors in template definitions

```
template<ForwardIterator Iter, typename val>
Iter find(Iter first, Iter last, const Val& val)          // fairly conventional find
{
	while (first!=last && *first!=val) first = first+1;  // catch early?
	return first;
}

void use(int arr[], list<int>& lst)
{
	auto p = find(arr[0],arr[10],7);          // error: int is not a forward iterator
	find q = find(lst.begin(), list.end());   // list<T>::iterator is a forward iterator
	                                          // BUT: late error!!
}
```

It is more important to allow a useful feature than to prevent every misuse [Str94]

- All type errors are caught eventually (instantiation time)

Definition checking: Why not?

- 90% of benefits come from point of use checking
  - And design improvements
- We know how to do definition checking (Gabriel Dos Reis experiments)
- Any significant change to a definition requires interface changes
  - What about debugging aids?
  - What about telemetry/logging?
- How to manage transition?
  - Can an unconstrained template call a constrained one?
    - Must: implies late checking
  - Can a constrained template call an unconstrained one?
    - Must: implies instantiation for checking
- Maybe never
  - Requires serious experimentation

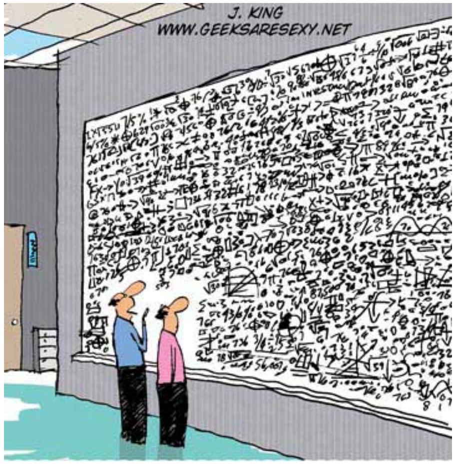
Definition checking: how to?

- Check for particular types
  - **static_assert(Range<My_type>);**
  - **static_assert(EqualityComparable<My_type1,Foo>);**
  - **static_assert(My_algorithm<My_type1>);**
  - **static_assert(My_algorithm2<Mt_type1,My_type2>);**
  - …
- Use archetypes
  - **static_assert(My_algorithm<Archetype_for_X,Archetype_for_Y>);**
- Beware
  - It’s easy to forget exactly the same operation in a concept and an archetype

Use higher-level concepts

- Consider **std::merge()**:

```
template<typename For,
                 typename For2,
                 typename Out>
    requires ForwardIterator<For>
        && ForwardIterator<For2>
        && OutputIterator<Out>
        && Assignable<Value_type<For>,Value_type<Out>>
        && Assignable<Value_type<For2,Value_type<Out>>
        && Comparable<Value_type<For>,Value_type<For2>>
Out merge(For p, For  q, For2 p2, For2 q2, Out p);
```



"...And that, in simple terms, is what's wrong with your software design."

- Headache inducing, and **accumulate()** is worse
  - But that's what the standard says (For good reasons)
  - And that's just the syntax

Use higher-level concepts

- Better:

```
template<typename For, typename For2, typename Out>
        requires Mergeable<For,For2,Out>
Out merge(For p, For q, For2 p2, For2 q2, Out p);
```

- Better still (but not C++20):

```
Mergeable{For,For2,Out}
    Out merge(For p, For  q, For2 p2, For2 q2, Out p);
```

- The

  concept-name { identifier-list }

  notation introduces constrained names

Use higher-level concepts

- Now we just need to define **Mergeable**:

```
template<typename For, typename For2, typename Out>
concept Mergeable =
        ForwardIterator<For>
        && ForwardIterator<For2>
        && OutputIterator<Out>
        && Assignable<Value_type<For>,Value_type<Out>>
        && Assignable<Value_type<For2,Value_type<Out>>
        && Comparable<Value_type<For>,Value_type<For2>>;
```

Principles of Concept Design

- Raise importance of semantics in conceptual specifications
- Emphasize distinction between universal and ad-hoc requirements
- Make consistent sets of properties concrete as concepts
- Support reasoning by using regular types and functions

- Major inspiration:
  - Stepanov and McJones:
    Elements of Programming
    Addison-Wesley 2009

Concrete suggestions

- User-level/application-level concepts should have semantics
- Incomplete concepts are better than no concepts
- Use named concepts
  - Never **requires requires**
- Use **static_assert**s to
  - Check (sets of) types against concepts
  - Check algorithms against (sets of) types
- Define algorithms using general types
  - For plug-and-play
  - Not absolute minimal requirements for an implementation
- Constrain variables with concepts to improve readability
  - Tighten type checking compared to auto
  - Relax type checking compared to specific types

Try concepts! You’ll never go back

- Concepts help us develop better designs
  - Improve interoperability (“plug and play”)
  - Help focus on fundamental issues and semantics
- Concepts provide better specification of interfaces
- Concepts simplify code
  - Fewer traits and **enable_if**s
  - Simpler and more precise expression of ideas (“shorter code”)
- Concepts give precise and early error messages
  - And fewer errors

Design principles:

- Provide good interfaces
- Look for semantic coherence
- Don’t force users to do what machines do better
- Keep simple things simple