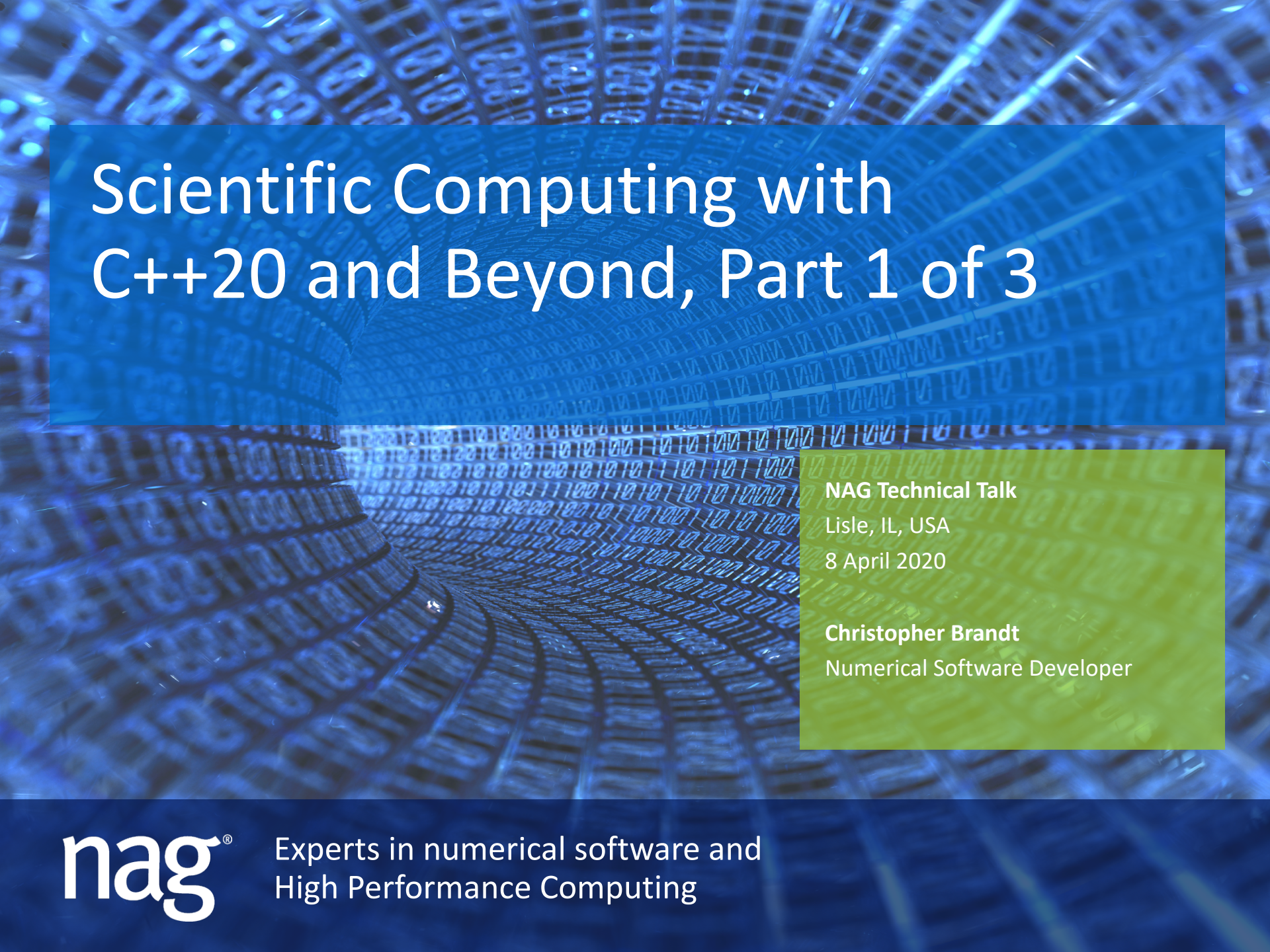


Scientific Computing with C++20 and Beyond, Part 1 of 3

NAG Technical Talk
Lisle, IL, USA
8 April 2020

Christopher Brandt
Numerical Software Developer

nag®

Experts in numerical software and
High Performance Computing

Overview

- Programming paradigms
  - Generic programming
  - Reference types
- New language features and extensions
  - concepts
  - span
  - mdspan
  - mdarray
  - blas
  - linear_algebra

Order of Topics

- Part 1
  - Generic programming
  - concepts
  - Reference types
  - span
- Part 2
  - mdspan
  - mdarray
- Part 3
  - blas
  - linear_algebra

Goals

- Provide a snapshot of current events in the C++ community
- Serve as an internal resource
  - For individual research
  - Incorporating ideas into future projects
- Materials include:
  - White papers
  - Proposals for the standard
  - Conference talks (slides)
  - Example codes (where implementations are available)

"Science is about classification, not grand inspirations."– A. Stepanov

Generic Programming

Generic Programming – Background

- Intuition originated from the associative property of parallel add
  - Generalizing the operations
  - A generic interface is the result of a generic implementation
- Term originally coined by Alexander Stepanov and David Musser in 1988
- Culminated with the creation of the C++ Standard Library in 1998
- Generic design sits at the core of all new Standard Library additions

Generic Programming – Process

- Begin with a concrete representation of an efficient algorithm (or collection of concrete algorithms)
- Identify and classify the sets requirements on types and the operations on them
  - E.g., identifying the “concepts”
- Iteratively lift these generalizations from the concrete algorithm(s) to create generalized representations

Generic Programming – Introducing Concepts

- *"We call the set of axioms satisfied by a data type and a set of operations on it a concept."*
  - J. Dehnert and A. Stepanov, 'Fundamentals of Generic Programming', 1998
- A concept is a way of describing a family of related types

Natural Sciences	***Mathematics***	***C++***	***C++ Example***
family	axiom	template	T
genus	theory	concept	`integral`
species	model	type or class	`int`
individual	elements	instance	42

Generic Programming – Concrete lower_bound

```
int lower_bound(int x[ ], int n, int value)
{
  int first = 0;
  int last  = n;

  while (first != last) {
    int dist   = last  - first;
    int middle = first + (dist / 2);

    if (x[m] < value)
      first = middle + 1;
    else
      last  = middle;
  }

  return first;
}
```

Generic Programming – Generalizing lower_bound

```
template <class T1> // T1 models Regular
int lower_bound(T1 x[ ], int n, T1 value)
{
  int first = 0;
  int last  = n;

  while (first != last) {
    int dist   = last  - first;
    int middle = first + (dist / 2);

    if (x[m] < value)
      first = middle + 1;
    else
      last  = middle;
  }

  return first;
}
```

Generic Programming – Generalizing lower_bound

```
template <class T1, // T1 models Regular
          class T2> // T2 models Integral
T2 lower_bound(T1 x[ ], T2 n, T1 const& value)
{
  T2 first = 0;
  T2 last  = n;

  while (first != last) {
    T2 dist   = last  - first;
    T2 middle = first + (dist / 2);

    if (x[m] < value)
      first = middle + 1;
    else
      last  = middle;
  }

  return first;
}
```

Generic Programming – Generalizing lower_bound

```
template <class I, // I models RandomAccessIterator
          class T> // T is value_type<I>
I lower_bound(I first, I last, T const& value)
{
  while (first != last) {
    auto dist = last  - first;
    I middle  = first + (dist / 2);

    if (*middle < value)
      first = middle + 1;
    else
      last  = middle;
  }

  return first;
}
```

Generic Programming – Generalizing lower_bound

```
template <class I, // I models RandomAccessIterator
          class T> // T is value_type<I>
I lower_bound(I first, I last, T const& value)
{
  while (first != last) {
    auto dist = last  - first;      // compute distance
    I middle  = first + (dist / 2); // advance iterator

    if (*middle < value)
      first = middle + 1;           // advance iterator
    else
      last  = middle;
  }

  return first;
}
```

Generic Programming – Generic lower_bound

```
template <class I, // I models RandomAccessIterator
          class T> // T is value_type<I>
I lower_bound(I first, I last, T const& value)
{
  while (first != last) {
    auto dist = distance(first, last);
    I middle  = next(first, dist / 2);

    if (*middle < value)
      first = next(middle);
    else
      last  = middle;
  }

  return first;
}
```

Generic Programming – Generic distance

```
// Generic function that dispatches to implementation overloads
template <class I> // I models InputIterator
typename std::iterator_traits<I>::difference_type
distance(I first, I last)
{
  typename std::iterator_traits<I>::iterator_category iterator_tag;
  return distance_impl(first, llast, iterator_tag);
}
```

Generic Programming – Generic distance Implementation

```
// Overload for InputIterator
template <class I> // I models InputIterator
typename std::iterator_traits<I>::difference_type
distance_impl(I first, I last, std::input_iterator_tag)
{
  typename std::iterator_traits<I>::difference_type result{0};
  for (; first != last; ++first)
    ++result;

  return result;
}


// Overload for RandomAccessIterator
template <class I> // I models RandomAccessIterator
typename std::iterator_traits<I>::difference_type
distance_impl(I first, I last, std::random_access_iterator_tag)
{
  return last - first;
}
```

Generic Programming – Generic distance Implementation

```
template <class I> // I models InputIterator
typename std::iterator_traits<I>::difference_type
distance_impl(I first, I last, std::input_iterator_tag)
{
  typename std::iterator_traits<I>::difference_type result{0};
  for (; first != last; ++first)
    ++result;

  return result;
}

template <class I> // I models RandomAccessIterator
typename std::iterator_traits<I>::difference_type
distance_impl(I first, I last, std::random_access_iterator_tag)
{
  return last - first;
}

template <class I> // I models InputIterator
typename std::iterator_traits<I>::difference_type
distance(I first, I last)
{
  typename std::iterator_traits<I>::iterator_category iterator_tag;
  return distance_impl(first, last, iterator_tag);
}
```

Generic Programming – next and advance

```
// Generic function defined in terms of another generic function
template <class I> // I models InputIterator
I next(I it, typename std::iterator_traits<I>::difference_type n = 1 )
{
  std::advance(it, n);
  return it;
}


// Generic function that dispatches to implementation overloads
// Design (not shown) is similar to that of std::distance
template <typename I, // I models InputIterator
          typename D> // D models Integral
void advance(I& it, D n)
{
  typename std::iterator_traits<I>::iterator_category iterator_tag;
  advance_impl(it, n, iterator_tag);
}
```

Generic Programming – Generic lower_bound

```
template <class I, // I models RandomAccessIterator
          class T> // T is value_type<I>
I lower_bound(I first, I last, T const& value)
{
  while (first != last) {
    auto dist = distance(first, last);
    I middle  = next(first, dist / 2);

    if (*middle < value)
      first = next(middle);
    else
      last  = middle;
  }

  return first;
}
```

Generic Programming – Generic lower_bound

```
template <class I, // I models ForwardIterator
          class T> // T is value_type<I>
I lower_bound(I first, I last, T const& value)
{
  while (first != last) {
    auto dist = distance(first, last);
    I middle  = next(first, dist / 2);

    if (*middle < value)
      first = next(middle);
    else
      last  = middle;
  }

  return first;
}
```

Generic Programming – Generic lower_bound

```
template <class I, // I models ForwardIterator
          class T> // T is value_type<I>
I lower_bound(I first, I last, T const& value)
{
  while (first != last) {
    auto dist = distance(first, last);
    I middle  = next(first, dist / 2);

    if (*middle < value)
      first = next(middle);
    else
      last  = middle;
  }

  return first;
}

// Contiguous sequential container
std::vector<int> v = { 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6 };

// Compute lower bound
auto it = lower_bound(v.begin(), v.end(), 4); // distance(v.begin(), it) = 6
```

Generic Programming – lower_bound for User Types

```
class Employee {
public:
  Employee() = default;
  Employee(int id, std::string const& f, std::string const& l)
    : id_(id), first_name_(f), last_name_(l) { }
  // ...

  int         id()         const { return id_; }
  std::string first_name() const { return first_name_; }
  std::string last_name()  const { return last_name_;  }
  // ...

private:
  int         id_;
  std::string first_name_;
  std::string last_name_;
  // ...
};
```

Generic Programming – lower_bound 3 Param Overload

```
template <class I, // I models ForwardIterator
          class T> // T is value_type<I>
I lower_bound(I first, I last, T const& value)
{
  while (first != last) {
    auto dist = distance(first, last);
    I middle  = next(first, dist / 2);

    if (*middle < value)
      first = next(middle);
    else
      last  = middle;
  }

  return first;
}
```

Generic Programming – lower_bound 4 Param Overload

```
template <class I, // I models ForwardIterator
          class T, // T is value_type<I>
          class C> // C models BinaryPredicate
I lower_bound(I first, I last, T const& value, C compare)
{
  while (first != last) {
    auto dist = distance(first, last);
    I middle  = next(first, dist / 2);

    if ( compare(*middle, value) )
      first = next(middle);
    else
      last  = middle;
  }

  return first;
}
```

Generic Programming – lower_bound Overload Example

```
// Define Employee objects
Employee e1{ 1, "John", "Doe" };
Employee e2{ 2, "Jeff", "Doe" };
Employee e3{ 4, "Josh", "Doe” };

// Initialize vector of employees
std::vector<Employee> e{ e1, e2, e3 };


// Heterogeneous comparison function
bool compare_id(Employee const& e, int id)
{ return e.id() < id; }


// Search for Employee id #3
auto it = lower_bound(e.begin(), e.end(), 3, compare_id);

// distance(e.begin(), it) = 2
```

Generic Programming – lower_bound Overload Set

```
// 4 parameter overload
template <class I, // I models ForwardIterator
          class T, // T is value_type<I>
          class C> // C models BinaryPredicate
I lower_bound(I first, I last, T const& value, C compare)
{
  while (first != last) {
    auto dist = distance(first, last);
    I middle  = next(first, dist / 2);

    if ( compare(*middle, value) )
      first = next(middle);
    else
      last  = middle;
  }

  return f;
}

// 3 parameter overload
template <class I, // I models ForwardIterator
          class T> // T is value_type<I>
I lower_bound(I first, I last, T const& value)
{
  return lower_bound(first, last, value, less<>{});
}
```

Generic Programming – upper_bound

```
// 4 parameter overload
template <class I, // I models ForwardIterator
          class T, // T is value_type<I>
          class C> // C models BinaryPredicate
I upper_bound(I first, I last, T const& value, C compare)
{
  auto dist = distance(first, last);

  while (dist > 0) {
    auto half = dist >> 1; I middle = first;
    advance(middle, half);

    if ( compare(value, *middle) ) dist = half;
    else {
      first = middle;
      ++first;
      dist -= half - 1;
    }
  }

return first;
}

// 3 parameter overload
// ... defined in terms of 4 parameter overload using std::less
```

Generic Programming – binary_search

```
// 4 parameter overload
template <class I, // I models ForwardIterator
          class T, // T is value_type<I>
          class C> // C models BinaryPredicate
bool binary_search(I first, I last, T const& value, C compare)
{
  f = lower_bound(first, last, value, compare);
  return ( !(first == last) && !(value < *first) );
}


// 3 parameter overload
template <class I, // I models ForwardIterator
          class T> // T is value_type<I>
bool binary_search(I first, I last, T const& value)
{
  return binary_search(first, last, value, less<>{});
}
```

Generic Programming – equal_range

```
// 4 parameter overload
template <class I, // I models ForwardIterator
          class T, // T is value_type<I>
          class C> // C models BinaryPredicate
std::pair<I, I> equal_range(I first, I last, T const& value, C compare)
{
  return std::make_pair( lower_bound(first, last, value, compare),
                         upper_bound(first, last, value, compare) );
}


// 3 parameter overload
template <class I, // I models ForwardIterator
          class T> // T is value_type<I>
inline std::pair<I, I> equal_range(I first, I last, T const& value)
{
  return equal_range(first, last, value, less<>{});
}
```

Generic Programming – lower_bound Recap

- Concrete lower_bound
  - Only accept a contiguous sequence of type int
- Generic lower_bound
  - Accept any sequential container (contiguous or non-contiguous)
  - Operates on any Regular type
  - Enable users to operate on their own types
  - Always providing the most efficient implementation possible
- Additionally
  - Identified 4 generic facilities that can be used in other functions
  - Enabled 2 functions to be defined in 2 lines of code by using lower_bound

Generic Programming – Properties of Generic Code

- Design
  - Easy to reason about
  - Easy to maintain
  - Easy to extend
  - Highly reusable
  - Highly scalable
- Performance
  - Generic code is *fast*

Generic Programming – References

- Presentations and conference talks:
  - Generic Programming, CppCon 2018 (S. Parent)
  - STL and Its Design Principles, Adobe Systems Inc., Jan. 2002 (A. Stepanov)
- Books:
  - Elements of Programming (A. Stepanov, P. McJones)
  - From Mathematics to Generic Programming (A. Stepanov, D. Rose)
- Papers:
  - Generic Programming (D. Musser, A. Stepanov)
  - Fundamentals of Generic Programming (J. Dehnert, A. Stepanov)
  - stepanovpapers.com

A set of requirements on a set of template arguments (C++20)

`concepts`

concepts – Motivation

- Concepts lets us constrain template parameters
  - Checking at the point of use
  - Improved compiler diagnostics
  - Direct expression of programmer intent within code
    - Core of the C++ philosophy
    - C++ Core Guildelines P.3
  - Specialize behaviour-based constraints
    - Template overloads
    - Class template specializations

concepts – Design

- Compile-time predicate expressing requirements on template parameters
  - Syntactic requirements – what operations, associated types are needed
  - Semantic requirement – behaviors required from the operations
  - Complexity requirements – performance requirements of operations
- Check syntactic requirements, and rely on the programmer for the semantics
- Not able to check for complexity

concepts – Concepts Library Arithmetic Concepts

```
namespace std {

  template <class T>
    concept integral = is_integral_v<T>;

  template <class T>
    concept signed_integral = integral<T> && is_signed_v<T>;

  template <class T>
    concept unsigned_integral = integral<T> && !is_signed_v<T>;

  template <class T>
    concept floating_point = is_floating_point_v<T>;

}
```

concepts – Concepts Library equality_comparable

```
namespace std {

  template <class T>
    concept weakly-equality-comparable-with = // exposition only
      requires(const remove_reference_t<T>& t,
               const remove_reference_t<U>& u) {
        { t == u } -> boolean;
        { t != u } -> boolean;
        { u == t } -> boolean;
        { u != t } -> boolean;
      };

  template <class T>
    concept equality_comparable = weakly-equality-comparable-with<T, T>;

}
```

concepts – Concepts Library Object Concepts

```
namespace std {

  template <class T>
    concept moveable = is_object_v<T>; && move_constructible<T> &&
                       assignable_from<T&, T> && swappable<T>;

  template <class T>
    concept copyable = copy_constructible<T> && moveable<T> &&
                       assignable_from<T&, T&> && assignable_from<T&, const T&> &&
                       assignable_from<T&, const T&>;

  template <class T>
    concept semiregular = copyable<T> && default_initializable<T>;

  template <class T>
    concept regular = semiregular<T> && equality_comparable<T>;

}
```

concepts – User-Defined Example

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

    { a += b } -> std::same_as<T&>;
    { a -= b } -> std::same_as<T&>;
    { a *= b } -> std::same_as<T&>;
    { a /= b } -> std::same_as<T&>;

    { T{0} }; // Construct from zero
};
```

concepts – Direct Expression of Intent

```
template <typename T1, // T1 models Regular
          typename T2, // T2 models FloatingPoint
          typename T3> // T3 models SignedIntegral
void foo(T1 a, T2 b, T3 c)
{
  //
  // do fooable things
  //
};


template <typename T1, typename T2, typename T3>
  requires std::regular<T1>
        && std::floating_point<T2>
        && std::signed_integral<T3>
void foo(T1 a, T2 b, T3 c)
{
  //
  // do fooable things
  //
};
```

concepts – Usage and Notations

```
// Generic function (C++14)
template <typename T, typename U>
auto add(T const& a, U const& b)
{ return a + b; }

// Long form
template <typename T, typename U>
  requires Number<T> && Number <U>
auto add(T const& a, U const& b)
{ return a + b; }

// Short form
template <Number T, Number U>
auto add(T const& a, U const& b)
{ return a + b; }

// Trailing form
template <typename T, typename U>
auto add(T const& a, U const& b) requires Number<T> && Number <U>
{ return a + b; }
```

concepts – Constrained auto

```
// Automatic type deduction
auto add_auto(auto const& a, auto const& b)
{ return a + b; }

// Constrained automatic type deduction
Number auto add_constrained_auto(Number auto const& a, Number auto const& b)
{ return a + b; }


int a = 1;
int b = 2;

auto result1 = add_auto(a, b); // = 3

Number auto result2 = add_constrained_auto(a, b); // = 3


double x = 1.5;
double y = 2.2;

Number auto result3 = add_constrained_auto(x, y); // = 3.7
Number auto result4 = add_constrained_auto(x, b); // = 3.5
```

concepts – Compiler Diagnostics

```
template <typename T, typename U>
auto add_generic(T const& a, U const& b)
{ return a + b; }


std::string s1 = "abc";
std::string s2 = "xyz";

auto r = add_generic(s1, s2); // = "abcxyz"

std::floating_point auto r2 = add_generic(s1, s2); // error!
```

```
file_name.cxx: In function ‘int main()’:
file_name.cxx:20:51: error: deduced initializer does not satisfy placeholder constraints
   20 |   std::floating_point auto r2 = add_generic(s1, s2); // error!
      |                                                   ^
file_name.cxx:20:51: note: constraints not satisfied
In file included from file_name.cxx:5:
/.../include/c++/10.0.0/concepts:111:30: note: the expression ‘is_floating_point_v<_Tp>’ evaluated to
‘false’
  111 |     concept floating_point = is_floating_point_v<_Tp>;
      |                              ^~~~~~~~~~~~~~~~~~~~~~~~
```

concepts – Compiler Diagnostics

```
template <typename T, typename U>
auto add_generic(T const& a, U const& b)
{ return a + b; }


std::vector<std::string> v1 = { "abc" };
std::vector<std::string> v2 = { "xyz" };


auto result = add_generic(v1, v2); // error!
```

concepts – Compiler Diagnostics

```
/Users/.../include/c++/10.0.0/bits/basic_string.h:6139:5: note: candidate: ‘template<class _CharT, class _Traits,
class _Alloc> std::__cxx11::basic_string<_CharT, _Traits, _Allocator>
std::operator+(std::__cxx11::basic_string<_CharT, _Traits, _Allocator>&&, _CharT)’
 6139 |     operator+(basic_string<_CharT, _Traits, _Alloc>&& __lhs,
      |     ^~~~~~~~
/Users/.../include/c++/10.0.0/bits/basic_string.h:6139:5: note:   template argument deduction/substitution failed:
In file included from file_name.cxx:10:
file_name2.hxx:10:12: note:   types ‘std::__cxx11::basic_string<_CharT, _Traits, _Allocator>’ and ‘const
std::vector<std::__cxx11::basic_string<char> >’ have incompatible cv-qualifiers
   10 |   return a + b;
      |          ~~^~~
In file included from /Users/.../include/c++/10.0.0/bits/stl_algobase.h:67,
                 from /Users/.../include/c++/10.0.0/bits/char_traits.h:39,
                 from /Users/.../include/c++/10.0.0/ios:40,
                 from /Users/.../include/c++/10.0.0/ostream:38,
                 from /Users/.../include/c++/10.0.0/iostream:39,
                 from file_name.cxx:6:
/Users/.../include/c++/10.0.0/bits/stl_iterator.h:1026:5: note: candidate: ‘template<class _Iterator, class
_Container> constexpr __gnu_cxx::__normal_iterator<_Iterator, _Container> __gnu_cxx::operator+(typename
__gnu_cxx::__normal_iterator<_Iterator, _Container>::difference_type, const __gnu_cxx::__normal_iterator<_Iterator,
_Container>&)’
 1026 |     operator+(typename __normal_iterator<_Iterator, _Container>::difference_type
      |     ^~~~~~~~
/Users/.../include/c++/10.0.0/bits/stl_iterator.h:1026:5: note:   template argument deduction/substitution failed:
In file included from file_name.cxx:10:
file_name2.hxx:10:12: note:   ‘const std::vector<std::__cxx11::basic_string<char> >’ is not derived from ‘const
__gnu_cxx::__normal_iterator<_Iterator, _Container>’
   10 |   return a + b;
      |          ~~^~~.

+ another 205 lines of error messages
```

concepts – Compiler Diagnostics

```
template <Number T, Number U>
auto add_concepts(T const& a, U const& b)
{ return a + b; }

std::vector<std::string> v1 = { "abc" };
std::vector<std::string> v2 = { "xyz" };


auto result = add_concepts(v1, v2); // error!
```

concepts – Compiler Diagnostics

```
number.hxx:7:18:   in requirements with ‘std::vector<std::__cxx11::basic_string<char> > a’,
‘std::vector<std::__cxx11::basic_string<char> > b’
number.hxx:8:7: note: the required expression ‘(a + b)’ is invalid
    8 |   { a + b }  -> std::same_as<T>;
      |     ~~^~~
number.hxx:9:7: note: the required expression ‘(a - b)’ is invalid
    9 |   { a - b }  -> std::same_as<T>;
      |     ~~^~~
number.hxx:10:7: note: the required expression ‘(a * b)’ is invalid
   10 |   { a * b }  -> std::same_as<T>;
      |     ~~^~~
number.hxx:11:7: note: the required expression ‘(a / b)’ is invalid
   11 |   { a / b }  -> std::same_as<T>;
      |     ~~^~~
number.hxx:12:5: note: the required expression ‘- a’ is invalid
   12 |   { -a }     -> std::same_as<T>;
      |     ^~
number.hxx:14:7: note: the required expression ‘a += b’ is invalid
   14 |   { a += b } -> std::same_as<T&>;
      |     ~~^~~~
number.hxx:15:7: note: the required expression ‘a -= b’ is invalid
   15 |   { a -= b } -> std::same_as<T&>;
      |     ~~^~~~
number.hxx:16:7: note: the required expression ‘a mult_expr b’ is invalid
   16 |   { a *= b } -> std::same_as<T&>;
      |     ~~^~~~
number.hxx:17:7: note: the required expression ‘a /= b’ is invalid
   17 |   { a /= b } -> std::same_as<T&>;
      |     ~~^~~~

+ another 15 lines
```

concepts – Implementations

- Compilers
  - GCC 10 (Trunk) using –std=c++2a flag

concepts – Reference

- Conference Talks
  - Concepts in 60, CppCon 2018, Andrew Sutton
  - Concepts: The Future of Generic Programming (The Future is Now), CppCon 2018, Bjarne Stroustrup
- Publications
  - Introducing Concepts, ACCU Overload Journal #129, Andrew Sutton
  - Defining Concepts, ACCU Overload Journal #131, Andrew Sutton
  - Overloading with Concepts, ACCU Overload Journal #136, Andrew Sutton
- Additional Reading
  - Concepts: The Future of Generic Programming, Bjarne Stroustrup

“Within C++, there is a much smaller and cleaner language struggling to get out” – B. Stroustrup

Reference Types

Reference Types – Introduction

- Description
  - A non-owning object that provides access to the storage of another object
- Similar terms
  - Reference type
  - Parameter type
  - Vocabulary type
  - Borrow type
  - View
  - Span

Reference Types – Properties

- Very light weight abstraction
  - Often configurable to store only a single pointer
  - Zero-overhead nature
- Pass-by-value semantics
  - Cheap to construct
  - Cheap to copy
  - Trivial destructor
- Contain useful methods for slicing
- Not an iterator (cannot be incremented, etc.)
- Shallow const-ness (similar to pointers)

Reference Types – Leveraging the Type System

```
// Run time configurable
class Ref_type_one {
public:
  Ref_type_one(double* d, std::size_t s) : data_(d), size_(s) { }
  double*     data() const { return data_; }
  std::size_t size() const { return size_; }
private:
 double*     data_;
 std::size_t size_;
};

// Compile time configurable
template <std::size_t Size>
class Ref_type_two {
public:
  Ref_type_two(double* d) : data_(d) { }
  double*     data() const { return data_; }
  std::size_t size() const { return Size;  }
private:
  double* data_;
};
```

Reference Types – Leveraging the Type System

```
std::vector<double>
  x{ 1.0, 2.0, 3.0, 4.0, 5.0 };


Ref_type_one    r1( x.data(), 5 ); // size configured at run time
Ref_type_two<5> r2( x.data()    ); // size configured at compile time


auto s1 = r1.size();        // = 5
auto s2 = r2.size();        // = 5

auto sizeof1 = sizeof(r1); // = 16
auto sizeof2 = sizeof(r2); // = 8
```

Reference Types – Usage

- Most commonly used as function parameters
  - Reference type as a replacement for an overload sets is VERY powerful!
  - Easier for libraries and applications to support users' own types
  - Typically constructed within the function call itself
- Always be sure the underlying data exists longer than the reference type instance
  - Usage cases do exist
  - Proceed with caution

Reference Types – Examples

- C++ Standard Library
  - std::string_view (C++17)
  - std::span (C++20)
  - std::mdspan (C++23*)
- Open source C++ libraries
  - abseil::string_view github.com/abseil/abseil-cpp
  - gsl::span github.com/microsoft/GSL
  - gsl::string_span github.com/microsoft/GSL

Reference Types – References

- Conference Talks
  - Modern C++ API Design Part 1, CppCon 2018, Titus Winters
  - The Most Important Design Guideline, Scott Meyers
- Other Readings
  - Revisiting Regular Types, Titus Winters
  - A Can of span, Corentin Jabot

A non-owning view over a contiguous sequence of objects (C++20)

span

span – Motivation

- Provide both high performance and bounds-safe access to contiguous sequences of elements
  - Effectively replacing the pointer + size idiom
  - Drop-in replacement for passing a ContiguousContainer by reference
- Provide clear operations and semantics for working on subsets of contiguous sequences
  - Subview methods
  - Element access via subscript operator []
  - Iterator support
- VERY powerful tool for extending an API to accommodate a wide range of user-defined types

span – Design

- Simply a view over another object's contiguous storage
  - Never performs any free store allocations
  - Aims to replace pointer arithmetic and array indexing
- Supports both static-size and dynamic-size views
  - Dynamic-size (default) is provided at runtime
    - Conceptually, just a pointer and size field (not an implementation requirement)
  - Static-size is fixed at compile-time
    - Requires no storage size overhead beyond a single pointer (leveraging type system)
    - Extremely efficient type to use for access to fixed-length buffer
- Encouraged to use as a pass-by-value param type

span – Synopsis

```
namespace std {

  // Constants
  inline constexpr size_t dynamic_extent = numeric_limits<size_t>::max();

  // class template span
  template <class ElementType, size_t Extent = dynamic_extent>
    class span;

}
```

span – Construction

```
// Containers
int c[ ] =          { 1, 2, 3, 4, 5, 6 };
std::array<int, 6> a{ 1, 2, 3, 4, 5, 6 };
std::vector<int>   v{ 1, 2, 3, 4, 5, 6 };

// Dynamic-size span objects
std::span<int> ds1{ c };
std::span<int> ds2{ a };
std::span<int> ds3{ v.data(), v.size() };    // sizeof(dsN) = 16


// Fixed-size span objects
std::span<int, 6> fs1{ c };
std::span<int, 6> fs2{ a };
std::span<int, 6> fs3{ v.data(), v.size() }; // sizeof(fsN) = 8
```

span – Subviews

```
// Built-in array
int c[ ] = { 1, 2, 3, 4, 5, 6 };

// span objects
std::span<int>    ds{ c }; // dynamic-size
std::span<int, 6> fs{ c }; // fixed-size

// Create dynamic-size subspan objects
auto ds_sub1 = fs.first(3);
auto ds_sub2 = ds.last(3);
auto ds_sub3 = fs.subspan(2, 3);   // [offset, count)

// Create fixed-size subspan objects
auto fs_sub1 = fs.first<3>();
auto fs_sub2 = ds.last<3>();
auto fs_sub3 = fs.subspan<2, 3>(); // [offset, count)
```

span – Usage as Function Parameter

```
// Dynamic-size span parameter
void add_two(std::span<int> s)
{
  for (auto& i : s) i += 2;
}

// Fixed-size span parameter
template <std::size_t N>
void add_two(std::span<int, N> s)
{
  for (auto& i : s) i += 2;
}


std::vector<int> v{ 1, 2, 3, 4, 5, 6 };

// Invoke dynamic-size span overload
add_two( {v.data(), v.size()} );    // v = { 3, 4, 5, 6, 7, 8  }

// Invoke fixed-size span overload
add_two<6>( {v.data(), v.size()} ); // v = { 5, 6, 7, 8, 9, 10 }
```

span – Accommodating User Defined Types

```
// Dynamic-size span parameter
void add_two(std::span<int> s)
{
  for (auto& i : s) i += 2;
}

// Fixed-size span parameter
template <std::size_t N>
void add_two(std::span<int, N> s)
{
  for (auto& i : s) i += 2;
}


My_vector_type<int> my_vec{ 1, 0, 0, 1, 0, 1};

// Invoke dynamic-size span overload
add_two( {my_vec.storage(), my_vec.len()} );    // my_vec = { 3, 2, 2, 3, 2, 3 }

// Invoke fixed-size span overload
add_two<6>( {my_vec.storage(), my_vec.len()} ); // my_vec = { 5, 4, 4, 5, 4, 5 }
```

span – Implementations

- Compilers
  - GCC 10 (Trunk)
- Single-Header, Standard Conforming
  - C++11 and later: github.com/tcbrindle/span
- Non-Standard Conforming
  - Microsoft GSL: github.com/microsoft/GSL

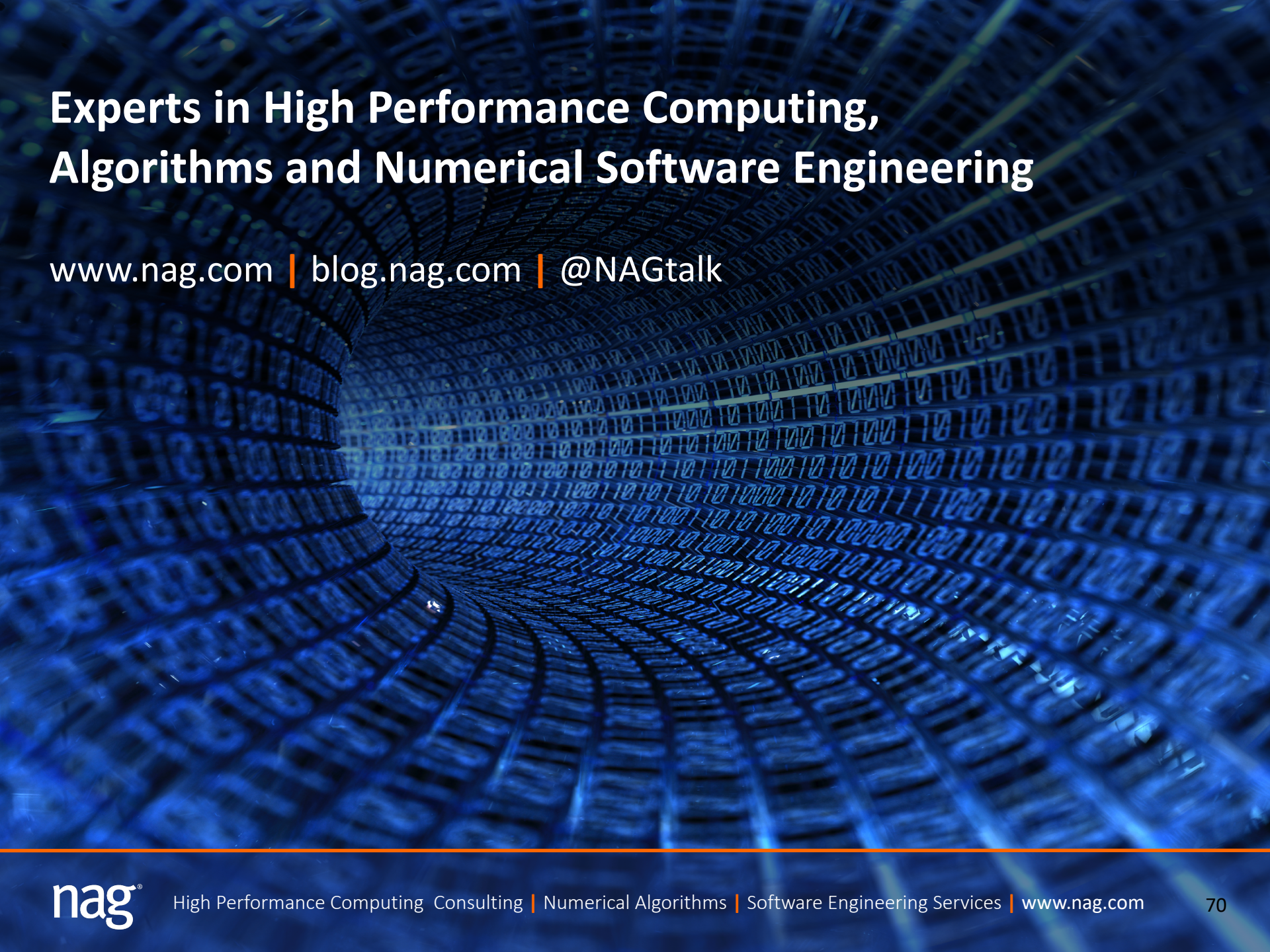
span – Reference

- C++ Standards Committee Papers
  - span: Bounds-Safe Views for Sequences of Objects (P0122R7)
  - Usability Enhancements for std::span (P1024R3)
  - Range Constructor for std::span (P1394R4)
- Best Practices
  - C++ Core Guidelines: github.com/isocpp/CppCoreGuidelines
- Conference Talks
  - Modern C++ API Design Part 1, CppCon 2018, Titus Winters

Preview of Part 2 and Part 3

- Part 2
  - mdspan
    - A non-owning multidimensional array reference
    - Sits a the core of numerous HPC applications
  - mdarray
    - The owning corollary of mdspan
- Part 3
  - blas
    - A free function linear algebra interface based on the BLAS
  - linear_algebra
    - Bringing “Matlab-like” functionality to the C++ Standard Library

Questions?


Experts in High Performance Computing,
Algorithms and Numerical Software Engineering
www.nag.com | blog.nag.com | @NAGtalk