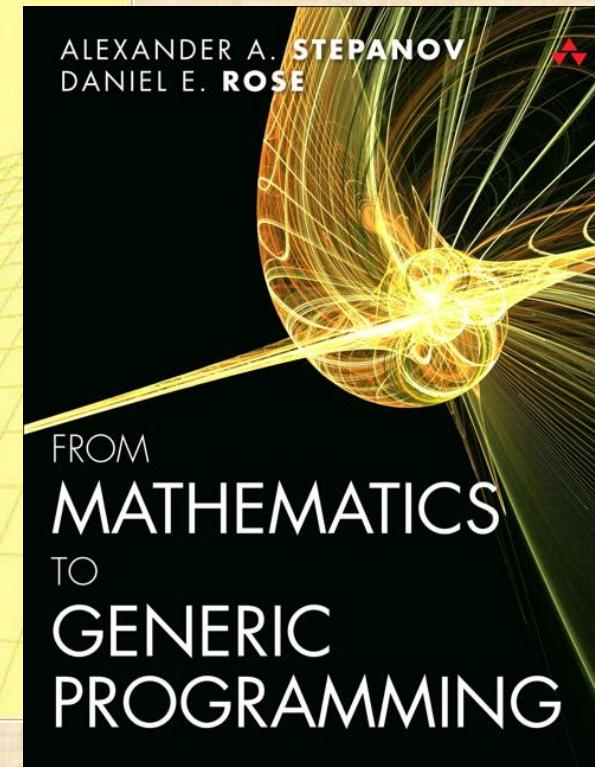
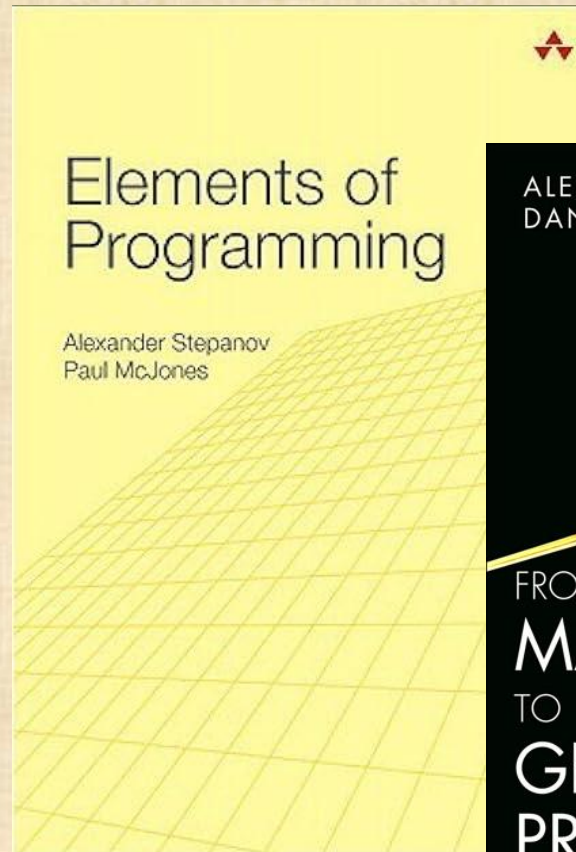




Concept-based Generic Programming

Bjarne Stroustrup
Columbia University
www.stroustrup.com



Generic programming

- A. Stepanov:
 - “Aim: The most general, most efficient, most flexible representation of concepts”
 - Don’t abstract for the sake of abstraction
 - Abstracting from concrete, efficient algorithms
 - Maintain (optimal) performance
- B. Stroustrup:
 - Represent separate concepts separately in code
 - Combine concepts freely wherever meaningful
 - Minimize overhead
 - Make generic code as simple as non-generic code





Three design requirements outlined in 1994 [D&E]

- **Generality**: Must be able to express more than I can imagine.
- **Uncompromised efficiency**: Generic code must not impose run-time overheads compared to roughly equivalent lower-level code, e.g., a generic, strongly typed vector is as efficient as a C array.
- **Statically type-safe interfaces**: The type system must be flexible enough to allow compile-time checking of most aspects of interfaces that do not depend on run-time values.
- Further requirements:
 - **Affordable**: Doesn't require expensive computers or slow compilers.
 - **Teachable**: "if we require PhDs from MIT, we have failed" – Kristen Nygaard



Every good design starts with a problem

Implicit narrowing conversions (nasty C legacy)

Let's define a set of concepts to eliminate narrowing conversions – at very low cost

- An integer converted to a type with too few bits to represent its value
 - e.g., **short x = 1'000'000**; assuming that a **short** is 16 bits
- An unsigned int with a representation interpreted as a (large) negative int after conversion to signed
 - e.g., **short x = 0b1000'0000'0000'0000u**;
- A negative integer interpreted as a (large) positive value after conversion to an unsigned
 - e.g., **unsigned x = -2**;
- A floating-point value with a decimal part truncated to its integer part when converted to an int
 - e.g., **int x = 7.8**;



Arithmetic types

- First, let's deal with only arithmetic types. We define a concept for that

```
template<typename T>  
concept Num = integral<T> || floating_point<T>;
```

Is T an integral type?

Is T a floating point type?

- A concept is a predicate (a function returning a **bool**) that can be evaluated at compile time and can take types as arguments.
 - **integral<T>** and **floating_point<T>** are supplied by the standard library
 - If needed, we can extend **Num** to handle non-standard types



Arithmetic types

- Next, define which conversions can possibly narrow

```
template<typename T, typename U>  
concept Can_narrow = (sizeof(T) < sizeof(U))           // too small  
                    || (integral<T> && floating_point<U>) // can round  
                    || ((sizeof(T) == sizeof(U))        // the U value can be too large  
                        && ((floating_point<T> && integral<U>)  
                            || (signed_integral<T> != signed_integral<U>))));
```

- Copied from the standard, but now it is code that we can use
 - Yes, it's complicated logic but that's why we are formalizing it



We need a function for testing values

```
template<Num T, Num U>
constexpr bool will_narrow(U u)
{
    if constexpr (!Can_narrow<T, U>)
        return false;
    if constexpr (signed_integral<T> && unsigned_integral<U>)
        if (numeric_limits<T>::max() < u)           // too large positive?
            return true;
    if constexpr (unsigned_integral<T> && signed_integral<U>)
        if (u < 0)
            return true;                             // negative?
    T t = u;                                          // potentially narrows
    return (t != u);                                // narrows?
}
```



Decide what to do with narrowing

- I use exceptions for very rare (i.e., exceptional) occurrences
 - For such, exceptions are more efficient than error codes plus tests
 - Implicit narrowing is very rare in good code

```
class Bad_value {};
```

```
template<Num T, Num U>  
constexpr T convert_to(U u)  
{  
    if (will_narrow<T>(u))  
        throw Bad_value{};  
    return T(u);  
}
```

// we only gets to here if the cast doesn't narrow



Test!

```
void test(int si, char ch, unsigned ui, double d)
{
    auto x0 = convert_to<int>(si);           // redundant
    auto x1 = convert_to<int>(ui);           // ui could be too large
    auto x2 = convert_to<char>(si);          // si could be too large
    auto x3 = convert_to<int>(ch);           // redundant
    auto x4 = convert_to<unsigned>(ch);       // redundant or ch could be negative
    auto x5 = convert_to<unsigned>(si);       // si could be negative
    auto x6 = convert_to<double>(si);        // redundant
    auto x7 = convert_to<int>(d);            // could truncate (e.g., if d is 7.8)
}
```

- Verbose, tedious, unlikely to be used consistently



Now make the testing implicit

```
template<Num T>
class Number {      // a T that doesn't suffer narrowing conversions
    T val;
public:
    template<Num U>
    constexpr Number(const U u) : val{convert_to<T>(u)} { }      // initialize

    template<Num U>
    constexpr void operator=(const U u) { val = convert_to<T>(u); }      // assign

    operator T() { return val; }      // extract value
};
```



Test!

```
void test(int i)
{
    Number<unsigned int> ii = 0;
    Number<char> cc = '0';

    ii = 2;           // OK
    ii = -2;          // throws
    cc = i;            // OK if i is within cc's range
    cc = -17;          // OK if char is signed; otherwise throws
    cc = 1234;         // throws if char is 8 bits
}
```



We need arithmetic operations

```
template<Num N1, Num N2>  
using Common = std::common_type_t<N1,N2>; // a type that can represent a N1 and a N2
```

```
template<Num N1, Num N2>  
auto operator+(Number<N1> x, Number<N2> y)  
{  
    return Number<Common<N1,N2>>{ x.val + y.val };  
}
```

```
template<Num N1, Num N2>  
auto operator*(Number<N1> x, Number<N2> y)  
{  
    return Number<Common<N1,N2>>{ x.val * y.val };  
}
```

```
// ... other operations, such as *, /, ...
```



We (badly) need comparison operations

- Remember **-1 < 2u** is **false**!
 - Mixed **signed** and **unsigned** computations first convert the **signed** to **unsigned**
 - Yes, those old rules are tricky

```
template<Num N1, Num N2>
bool operator<(Number<N1> x, Number<N2> y)
{
    if constexpr (is_signed_v<N1> && is_unsigned_v<N2>)
        if (x.val < 0)           // x.val would be converted into a positive number
            return true;
    return x.val < y.val ;
}
```

```
// ... other operations, such as ==, <=, ...
```



Test!

```
Number<int> test(double d, Number<int> i)
{
    auto x = d+i*10;           // x is a double
    Number<double> z = d+i*10; // OK
    return x*2 - d*z;          // will check: may narrow
}
```



Let's attack range errors

narrowing conversions is a source of range errors

- An example of the problem

```
const unsigned max = 100;  
int a[max];  
span<int> s {a, max-500};
```

// I mistyped 50. That -500 is converted to 4294966796

- A concept for “containers”
 - E.g., vector, std::array, built-in arrays

```
template<typename S>  
concept Spanable = ranges::contiguous_range<S>;
```



Let's attack range errors

narrowing conversions is a source of range errors

```
template<typename T>
class Span {
    T* p;           // elements pointed to
    unsigned n;     // number of elements pointed to by p
public:
    unsigned check(unsigned nn)                // in range?
    {
        if (nn < n) return nn;
        throw Span_range_error{};
    }

    Span(Spanable auto& s) : p{ data(s)}, n{size(s)} {} // [0:size); a Spannable has a size(S)

    T& operator[](Number<unsigned> i) { return p[check(i)]; } // check here
};
```



Test!

```
void test(Span<Number<int>> ssi, vector<double>& v)
{
    int x0 = ssi[10];           // OK if 10 is in range
    int x1 = ssi[-10];          // will throw

    Span<double> sv {v};
    int xx0 = sv[10];           // OK if 10 is in range
    int xx1 = sv[-1];           // will throw

    int a[100];
    Span<int> sa { aa };
    int xxx0 = sa[10];           // OK
    int xxx1 = sa[200];          // will throw
    int xxx2 = sa[sv[2]];        // will throw unless sv[2] is a positive integer
}
```



Element type deduction

- Don't repeat what the compiler (and you) already knows
 - Class Template Type Deduction (CTAD)

```
template<Spanable R>
```

```
Span(R&) -> Span<ranges::range_value_t<R>>; // use the range's element type
```

```
void test()
```

```
{
```

```
    int aa[100];
```

```
    Span s1 {aa}; // deduce Span to mean Span<int>{aa,100}
```

```
    // ...
```

```
    for (const auto x : s1)
```

```
        cout << x << '\n';
```

```
}
```



A more realistic span

- Initialize with ranges, sub-ranges, and pointers

```
template<class T>
class Span {
    // ...
    // initialize with a pointer and an explicit number of elements:
    Span(T* pp, Number<unsigned> nn) :p{ pp }, n{ nn } { }           // can't check

    Span(Spanable auto& s, Number<unsigned> nn)
        : p{ data(s) }, n{ size(s) } { n = check(nn); }             // [0:nn)

    Span(Spanable auto& s, Number<unsigned> low, Number<unsigned> high)
        : p{ data(s) }, n{ size(s) } { p+=check(low); n= check(high - low); }    [low:high)
};
```



A more realistic span

- More type deduction

```
template<class T>  
Span(T*, Number<unsigned>) -> Span<T>;
```

```
template<Spanable R>  
Span(R&, Number<unsigned>) -> Span<ranges::range_value_t<R>>;
```

```
template<Spanable R>  
Span(R&, Number<unsigned>, Number<unsigned>) -> Span<ranges::range_value_t<R>>;
```



Test!

```
void test(vector<double>& v, int* p)
{
    int aa[100];
    Span s1 {aa};           // deduce Span to mean Span<int>{aa,100}
    Span s2 {aa,50};        // in case we want just half of the array
    Span s3 { aa,200};      // out of range: will throw

    Span sv1{v,10};         // first 10 elements
    Span sv2 {v,10,20};     // elements [10:20)

    int a[100];
    Span sa { aa, 10};      // first 10 elements
    Span sa { aa, 10, 20};  // elements [10:20)
    Span sp {p, 10};        // unchecked
}
```



More type deduction

- Now **Number** look verbose
 - “Don’t repeat the obvious

```
template<Num Init>
```

```
Number(const Init) -> Number<Init>; // deduce to the underlying type
```

```
Number n = 1; // Number<int>
```

```
Number n2 = 1u; // Number<unsigned>
```

```
Number n3 = 7.2; // Number<double>
```

- Be explicit, only when it makes code clearer

```
Number<double> d = 1;
```



Algorithms: Classic sort

```
template<typename Random_access_iterator>
void sort(Random_access_iterator first, Random_access_iterator last)
{
    // ... implementation ...
}
```

- This **sort** takes **first** and **last** of some type **Random_access_iterator**.
- The standard states precisely what properties are assumed for **first**, **last**, and **Random_access_iterator**, roughly:
 - **Random_access_iterator** must be a pointer-like type that can be used to iterate over a range of elements
 - **Random_access_iterator** must provide random access to that range
 - The elements pointed to by **Random_access_iterator** must be comparable with the **<** operator.
 - The **first** and **last** must define a half-open range [**first:last**).
- This style of code was spectacularly successful for decades
- It does not meet the third design criteria for C++ generic programming
 - interfaces should be precisely specified in code so that humans and compilers can read, understand, and use them.



Use

- Spectacularly successful
 - Efficient
 - Flexible
 - Despite appalling error messages

```
void test(vector<double>& vec, span<string>&ss, list<int>& lst)
{
    sort(vec.begin(),vec.end());           // vector of doubles in ascending order
    sort(ss.begin(),ss.end());              // span of strings in ascending order

    sort(lst.begin(),lst.end());            // error: list doesn't provide random access
}
```



Sort using concepts

```
template<random_access_iterator Iter, typename Pred = ranges::less>  
    requires sortable<Iter,Pred>  
void sort(Iter first, Iter last, Pred p = {});
```

- **Iter** must be an iterator providing random access to its elements (a **random_access_iterator**).
- Comparison of elements is done by a predicate, **Pred**, through such iterators; **sortable<Iter,Pred>** ensures that can be done.
- **Pred** is defaulted to a standard-library less-than operation (**ranges::less**). That **less** is hidden away in the **ranges** namespace.
- If the caller doesn't specify a comparison, the default **Pred** is used.



Use

- Identical to old style (compatibility)
 - Same performance as old style
 - Better and earlier error messages

```
void test(vector<double>& vec, span<string>&ss, list<int>& lst)
{
    sort(vec.begin(),vec.end());           // vector of doubles in ascending order
    sort(ss.begin(),ss.end());             // span of strings in ascending order

    sort(vec.begin(),vec.end(), ranges::greater{});    // descending order
    sort(ss.begin(),ss.end(), ranges::greater{});      // descending order

    sort(lst.begin(),lst.end());            // error: list doesn't provide random access
}
```



Use – Remaining problems

- A pair of iterators is supposed to specify a range
 - What if they don't? seen “in the wild”:
 - `sort(vec.end(),vec.begin());` *// begin and end flipped*
 - `sort(ss1.begin(),ss2.end());` *// not iterators into the same container*
 - It's verbose
 - 90+% of sorts are of a whole container



Sort using ranges

```
template<typename R, typename Pred = ranges::less>
    requires Sortable_range<R,Pred>
void sort(R& r, Pred p = {})
{
    sort(r.begin(), r.end(), p);           // one possible implementation
}

void test()
{
    vector<double> vd = {1, -2, 2, 3};
    vector<string> vs = {"d", "q", "a"};

    sort(vd);                             // sort ascending
    sort(vs, ranges::greater);             // sort descending
}
```



What if we wanted to sort lists?

The representation of lists is unsuitable for sorting

```
template<typename R, typename Pred = ranges::less>
concept Forward_sortable_range =
    ranges::forward_range<R>                // has begin()/end(), ++, ...
    && sortable<ranges::iterator_t<R>, Pred>; // compare elements using Pred

template<typename For, typename Pred = ranges::less>
    requires Forward_sortable_range<For, Pred>
void sort(For & r, Pred p = {})                // one possible implementation
{
    vector v {from_range, r};                  // copy the elements into a vector
    sort(v, p);                                // use the vector sort
    copy(r, v.begin());                        // copy the element back
}
```



Overloading picks the most constrained function

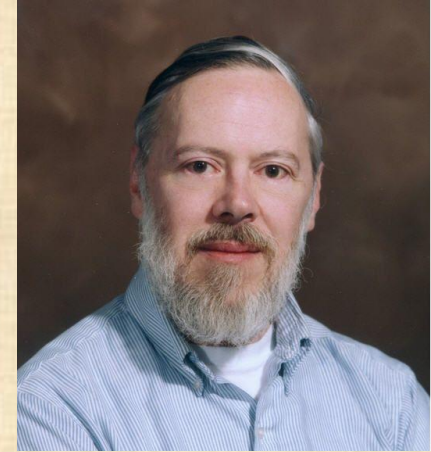
- We don't have to define hierarchies or priorities

```
void test()
{
    vector<double> vec = {1, -2, 2, 3};
    list<string> lst = {"d", "q", "a"};

    sort(vec);                // sort the vector ascending
    sort(lst, ranges::greater); // sort the list descending
}
```



Concepts



- We have always had the notion of concepts. For example,
 - ***C built-in types***: arithmetic and floating (from 1972 or so)
 - ***STL concepts***: iterators, sequences, and containers (since the early 1990s)
 - ***Mathematical concepts***: monad, group, ring, and field (for a couple of centuries)
 - ***Graph concepts***: edge, vertex, graph, and DAG (since 1736)
- No generic program could work unless the programmer had an idea of the concepts involved clear in mind. What is relatively new is that we can define them to be used in code.
- C++ concepts are simply compile-time functions (predicates) that can take type arguments.
- We must learn to use concepts effectively



Concepts are often built out of other concepts

```
template<typename R, typename Pred = ranges::less>
```

```
concept Forward_sortable_range =
```

```
    ranges::forward_range<R>
```

```
    && sortable<ranges::iterator_t<R>, Pred>;
```

```
// has begin()/end(), ++, ...
```

```
// compare elements using Pred
```



Concepts can be built out of basic language features

```
template<typename T, typename U = T>
concept equality_comparable = requires(T a, U b) {           // use patterns
    {a==b} -> Boolean;
    {a!=b} -> Boolean;
    {b==a} -> Boolean;
    {b!=a} -> Boolean;
}
```

- Use patterns
- The **requires** operator is a low-level mechanism for checking whether a construct is valid C++.
 - It is essential for expressing low-level requirements but best avoided in high-level code where named concepts (often built using **requires**) are more comprehensible and maintainable.



Use patterns

- Concepts specify what a template must be able to do with its arguments
 - Not exactly what an argument type must be
- For example: in **requires(T a, U b) {a+b;}**, the **+** in **a+b** could be provided as any of
 - **X operator+(X,Y);** *// if a is an X and b is a Y*
 - **X X::operator+(const Y&);** *// if a is an X and b is of a class derived from Y*
 - **Y operator+(const X&, const Y&);** *// if an X can be implicitly constructed from a T
// and a Y can be implicitly constructed from a U*
 - **Y operator+(Y,X&);** *// if a Y can be implicitly constructed from a T
// and b is an X*
 - ... and many more ...
- That's important.
 - Handles mixed-mode arithmetic
 - Handles implicit conversions
 - Interface stability (as the definition of **+** changes)



Checking is implicit

- But if we want to catch problems early, we can test that types are **equality_comparable** using **static_assert**:

```
static_assert(equality_comparable<int,double>);           // succeeds
static_assert(equality_comparable<int>);                  // succeeds (U is defaulted to int)
static_assert(!equality_comparable<int,string>);          // succeeds
```



GP (with concepts) is an integral part of C++

- Not a sub-language isolated from the rest
 - **Uniform treatment of types**
 - Scope, creation, destruction
 - Lambdas
 - Generating types and functions
 - **Optional member functions**
 - Compile-time computation
 - Variadic templates
 - **Static reflection (C++26)**
 - ...





Uniform treatment of types

- From the earliest days of C++
 - User-defined or built-in
 - Resource owner (with constructors and destructors) or not

```
template<typename X>  
void test()  
{  
    X x;  
    new X;  
}
```



Optional member functions

- A smart pointer

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

```
void test(Ptr<int> pi, Ptr<pair<int,string>> pcd)  
{  
    auto x0 = *pi;           // OK: x0 is an int  
    auto x1 = *pcd;          // OK: x1 is a pair<int,string>  
  
    auto x2 = pi->value;      // error: Ptr<int> doesn't have a ->  
    auto x3 = pcd->first;     // OK: x3 is an int  
}
```



Optional member functions

- Consider **pair**:

```
template<typename T, typename U>
struct Pair {
    T t;
    U u;
public:
    // ...
    // offer constructor (only) for types that can be converted to the members:
    template<typename TT, typename UU>
        requires std::convertible_to<TT, T>
            && std::convertible_to<UU, U>
        Pair(const Pair<TT, UU>& pp) :t(pp.t), u(pp.u) {};
};
```



Optional member functions

- But **convertible_to<TT,T>** doesn't protect against narrowing conversions, so it might be better to simply check to see if the values used really results in narrowing:

```
template<class TT, class UU>
Pair(const Pair<TT, UU>& pp)
    :Pair{convert_to<T>(pp.t), convert_to<U>(pp.u)} {}
```

- But **convert to** is defined to take arithmetic types!
 - Just overload **convert_to**:

```
template<typename T, typename U>
constexpr T convert_to(U u)
{
    return T(u);
}
```



But but wait!

- We built a non-narrowing rule into C++ about 15 years ago

- `int x {7.2};` *// error: narrowing*

- We can use language rules to define concepts

```
template<typename S, typename T>
```

```
concept Can_narrow = requires(T t) { t = { std::declval<S>() } };
```

```
Can_narrow <int, float>;        // true
```

```
Can_narrow<int, long>;        // true on some systems and false on others
```

```
Can_narrow <int, char>;        // false
```

```
Can_narrow <string, char*>;    // false
```

```
Can_narrow <char*, string>;    // true
```

Don't confuse an example
with what it is meant to be an example of



GP with Static Reflection (C++26)

- A simple example: generate descriptors of class members
 - When we can do that, we can do a huge range of support for run-time uses
 - generating I/O operations
 - serialization to/from various formats
 - foreign language calls to/from C++
 - ...

```
struct data_member_descriptor
{
    string_view name;
    size_t offset;
    size_t size;
};
```



```
template <typename S>
constexpr auto get_layout()
{
    constexpr auto members = meta::nonstatic_data_members_of(^S);

    array<data_member_descriptor, members.size()> layout;

    for (int i = 0; i < members.size(); ++i)
        layout[i] = {
            meta::identifier_of(members[i]),    // member name
            meta::offset_of(members[i]),
            meta::size_of(members[i])
        };
    return layout;    // returns an array<member_descriptor, members.size()>
}
```

Gain information from compiler



Use of static reflection

- The results of static reflection is ordinary C++

```
struct X {  
    char a;  
    int b;  
    string c;  
};
```

```
constexpr auto Xd = get_layout<X>();
```

- Now **Xd** is an array of **data_member_descriptors** with the value **{{ "a",0, 1}, {"b",4,4}, {"c",8,24}}**
 - The literal strings are what you'd see looking through the **string_views**



Major concept design decisions

- Concepts are functions
 - OOP and GP
- Concepts can be partial constraints
- Concepts can take value arguments
 - not just type arguments
- Concepts are not just for template arguments
- Concept type matching
- Definition checking





Concepts are functions

- Concepts are functions
 - Flexibility
 - Generality
 - Cheap compile time
- We rely on use patterns
 - Specify what a type must be able to do, not how it must do it
 - Mixed mode arithmetic
 - Implicit conversions
- Concepts can have multiple arguments
 - Handles relationships among arguments
- Concepts can take value arguments
 - not just type arguments



Concepts can take value arguments

```
template<int S> concept Buffer_space = (1024 <= S) && is_power_of_two (S);
```

```
template<Element T, int S>  
    requires Buffer_space<S>
```

```
struct Buffer {  
    T buf[S];  
    // ...
```

```
};
```

```
void test0()
```

```
{
```

```
    Buffer<char, 100> b1;
```

```
    // error: buffer too small
```

```
    Buffer<int, 10000> b2;
```

```
    // error: size not binary
```

```
    Buffer<int, 2048> b3;
```

```
    // OK
```

```
}
```



Compile-time evaluation

```
constexpr bool is_power_of_two(int n)
{
    if (n == 0)
        return false;           // we don't accept 0
    while ((n & 1) == 0)         // shift 0s away
        n >>= 1;
    return (n == 1);            // is there only a single 1 left?
}
```



GP and OOP

- Classical OOP
 - Focused on defining types of objects, often in class hierarchies
 - Applies to one object at a time
 - Relies on name equivalence
 - Run-time selection among an open set of types
 - Inlining is difficult (typically impossible)
 - Defining an interface requires precisely specifying types
 - That requires significant foresight, conceptionally and in technical details
- GP
 - Focused on a function's requirements on its arguments
 - Often involving relationships among multiple arguments
 - Relies on use patterns (a generalization of structural equivalence)
 - Compile-time selection among types in scope
 - Inlining is easy
 - Doesn't offer interfaces in the ABI sense
 - Use OOP for that



GP and OOP are complimentary

- The classical OOP “draw all shapes example”

```
void draw_all(Drawable_range auto& r)          // a generic function
{
    for (auto& d : r)
        r->draw();          // calling functions that probably (not necessarily) virtual
}
```

- A very simple version of **Drawable_range** looks like this:

```
template<typename T>
concept Drawable = requires(T a) { a->draw(); };

template<typename R>
concept Drawable_range = ranges::forward_range<R> && Drawable<ranges::iterator_t<R>>;
```



Concepts can be partial constraints

- Consider a **Number** concept specified directly from language primitives:

```
template<typename T, typename U = T>
concept Number = requires(T x, U y) {           // arithmetic operations and a zero
    x+y;    x-y;    x*y;    x/y;
    x+=y;   x-=y;   x*=y;   x/=y;
    x=x;    // copy a T (not x=y)
    x=0;
};
```

```
template<typename T, typename U = T>           // symmetric
concept Arithmetic = Number<T,U> && Number<U,T>;
```



False lookup matches are avoided

- What's the odds that this gets matched by something that is not a number?

```
template<typename T, typename U = T>  
concept Number = requires(T x, U y) {  
    x+y;    x-y;    x*y;    x/y;  
    x+=y;   x-=y;   x*=y;   x/=y;  
    x=x;  
    x=0;  
};  
  
// arithmetic operations and a zero  
  
// copy a T (not x=y)
```

```
template<typename T, typename U = T>  
concept Arithmetic = Number<T,U> && Number<U,T>;  
  
// symmetric
```



False lookup matches are avoided

- There are genuine examples of concepts that differ only semantically
 - The standard-library **forward_iterator** concept differs only from an **input_iterator** only in that a **forward_iterator** allows repeated traversals of its sequence.
- Simple solution:
 - introduce a syntactic difference indicating the semantic requirement.
 - For example: a **forward_iterator** must have a **forward_iterator_tag** and an **input_iterator** must not.
- Beware of “simple concepts” without meaningful semantics
 - Except as concepts used only as building blocks for other concepts
- Beware of single-function concepts
 - They are almost duck typing (only argument type checking protects)

Design principles:

- Don't let the tail wag the dog
- Keep simple things simple



Concepts are not just for template arguments

```
template<typename T>  
concept Integer = same_as<T,short> || same_as<T,int> || same_as<T,long>;
```

```
Integer auto x1 = 7;  
int x2 = 9;
```

```
Integer auto y1 = x1+x2;  
int y2 = x2+x1;
```

```
void f(int&);           // a function  
void f(Integer auto&);  // a function template
```

```
void ff()  
{  
    f(x1);  
    f(x2);  
}
```



Definition checking

- Consider:

```
void advance(input_iterator auto p, int n)
{
    log("advance({}, {})", p, n); // Should this work?
    p += n; // Should this work? Where to check?
}
```

- The interface doesn't mention **log**
- An **input_iterator** doesn't require **+=**
- In industrial code, infrastructure code is very common
 - Logging, debugging, tracing, monitoring, telemetry, ...



Definition checking

- Performance implications:

```
void advance(input_iterator auto p, int n) { /*... */ }
```

```
vector v = {0,1,2,3,4,5,6,7,8,9};
```

```
auto p = v.begin();
```

```
advance(p,2);
```

// Should this work? It has since C++98

- It works
- If it didn't, the performance of some standard-library algorithms would change from $O(n)$ to $O(n^2)$



Summary

- Generic Programming
 - Is “just” programming
 - *“The most general, most efficient, most flexible representation of concepts”*
 - Based on classical mathematics
 - Treats built-in and user-defined types uniformly (scope, naming, creation, cleanup)
 - Is complementary to classical Object-Oriented Programming
- Concepts
 - Are compile-time functions that can take multiple type and value arguments
 - not defined as classes with sets of function declarations – not (just) types of types
 - Specify what a generic function requires from its arguments
 - not how its arguments implement that
 - Can be used to select among alternatives
 - simple, flexible overloading

Generic Programming using concepts enables us to build our own type systems on top of what C++ offers by default



References

- A. Stepanov and P. McJones: Elements of Programming. Addison Wesley. 2009.
- A. Stepanov and D. Rose: From Mathematics to Generic Programming. Addison Wesley. 2014.
- B. Stroustrup: A Tour of C++ (3rd edition)
- B. Stroustrup: [C++20 Generic Programming](#). Video. 2020.
- B. Stroustrup: [Concepts: The future of generic programming](#)
- A. Sutton and B. Stroustrup: [Design of Concept Libraries for C++](#). Proc. SLE 2011. July 2011. Given the Most Influential Paper Award in 2021.
- B. Stroustrup: [The C++0x "Remove Concepts" Decision](#). Dr.Dobb's Journal. July 2009. The design that failed
- D. Gregor, J. Jarvi, J. Siek, B. Stroustrup, G. Dos Reis, Andrew Lumsdaine: [Concepts: Linguistic Support for Generic Programming in C++](#). OOPSLA'06. The design that failed.
- G. Dos Reis and B. Stroustrup: [Specifying C++ Concepts](#). POPL06. January 2006.
- Thomas: [Static, Dynamic Polymorphism, CRTP and C++20's Concepts](#) .
- B. Stroustrup: Concept-based Generic Programming. The written version of this talk: Blog@CACM.org (soon).