

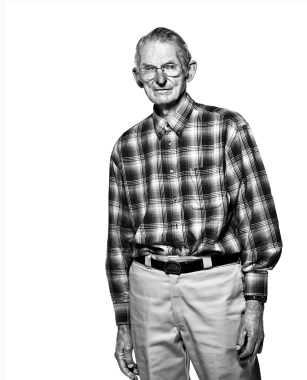
6 Algorithmic Journeys with Concepts

Taras Shevchenko

Rails Reactor / Giphy

The Software Industry is Not Industrialized

Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and time-space performance.



A Familiar Example. Douglas McIlroy about sin

Dimensions along which we wish to have variability:

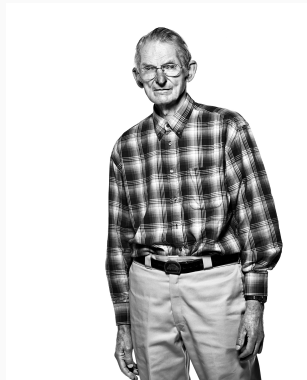
1. precision, for which perhaps ten different approximating functions might suffice
2. floating vs fixed computation
3. argument ranges $[0, \pi/2]$, $[0, 2\pi]$, also $[-\pi/2, \pi/2]$, $[-\pi, \pi]$, $[-big, +big]$
4. robustness - ranging from no argument validation through signaling of complete loss of significance, to signaling of specified range violations

1. Choices

- 1.1 precision
- 1.2 robustness
- 1.3 generality
- 1.4 generality
- 1.5 algorithm
- 1.6 interfaces and error-handling

2. Application Areas

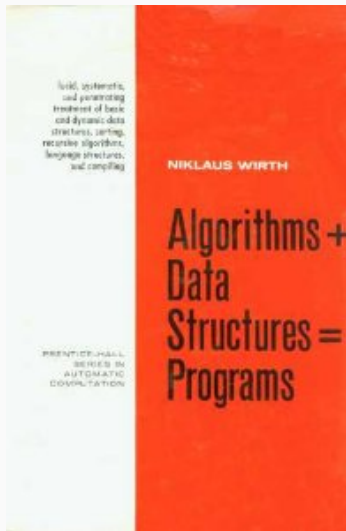
- 2.1 numerical approximation
- 2.2 I/O
- 2.3 2d and 3d geometry
- 2.4 text processing
- 2.5 storage management



Donald Knuth

1. Fundamental Algorithms
2. Seminumerical Algorithms
3. Sorting and Searching
4. Combinatorial Algorithms
5. Syntactic Algorithms
6. The Theory of Context-free Languages
7. Compiler Techniques





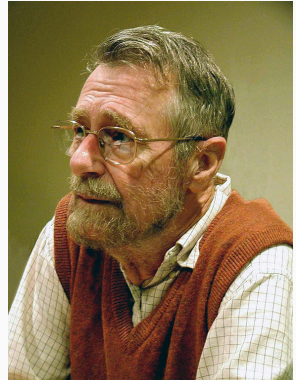
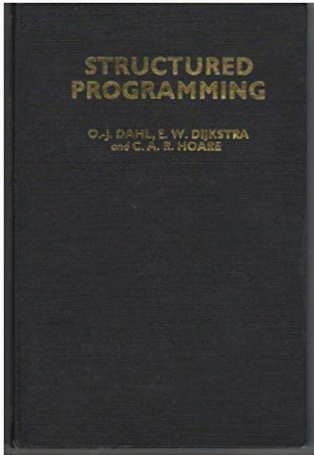


Figure 1: We need structured programming

1977 ACM Turing Award Lecture

The 1977 ACM Turing Award was presented to John Backus at the ACM Annual Conference in Seattle, October 27. In introducing the recipient, Jon E. Hummel, Chairman of the Awards Committee, made the following comments and read a portion of the final citation. The full announcement is in the September 1977 issue of *Communications*, page 651.

Probably there is nobody in the room who has not heard of Fortran and most of you have probably read at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably about as many people who have heard the letters BNF but don't necessarily know what they stand for. Well, it is for Backus, and the other letters are explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing Award.

The short form of his citation is for "pioneering, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publications of formal procedures for the specification of programming languages."

The most significant part of the full citation is as follows:

Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical com-

putations called Fortran. This team group designed the first system to translate Fortran programs into machine language. They employed novel optimizing techniques to generate fast machine-language programs. Many other compilers for the language were developed, first on IBM machines, and later on virtually every make of computer. Fortran was adopted as a U.S. standard in 1960.

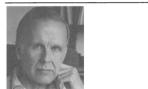
During the latter part of the 1950s, Backus served on the international committees which developed Algol 58 and a later version, Algol 68. The language Algol, and its distinctive computer, received broad acceptance in Europe as a means for describing programs and as a formal means of publishing the algorithms on which the programs are based.

In 1959, Backus presented a paper at the UNESCO conference in Paris on the syntax and semantics of a proposed international algebraic language. In this paper, he was the first to employ a formal technique for specifying the syntax of programming languages. The formal notation became known as BNF—standing for "Backus Normal Form," or "Backus Naur Form"—to recognize the further contributions by Peter Naur of Denmark.

Thus, Backus has contributed strongly both to the pragmatic world of problem-solving on computers and to the theoretical world of defining the interface between artificial languages and computational linguistics. Fortran reminds one of the most widely used programming languages in the world. Almost all programming languages are now described with some type of formal syntactic definition.^{1,2}

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



¹General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, in due form of time, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. If otherwise reprint a figure, table, other substantial excerpt, or the entire work require specific permission in form, reproduction, or systematic or multiple reproduction.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94133
© 1978 ACM 0001-0718/78/0000-0001 \$06.75

463

Conventional programming languages are growing ever more numerous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrecursive and nonsequential, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Communications
the ACM

August 1978
Volume 31
Number 3

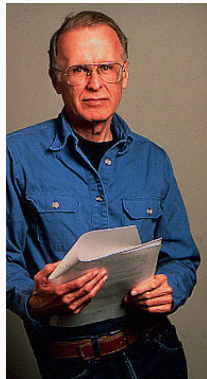


Figure 2: We need a few functional forms

Dmitri Mendeleev

ПЕРИОДИЧЕСКАЯ СИСТЕМА ЭЛЕМЕНТОВ Д. И. МЕНДЕЛЕЕВА

1	H																	2	He																																																																																																																
2	Li	3	Be	4	B	5	C	6	N	7	O	8	F	9	Ne																																																																																																																				
3	Na	10	Mg	11	Al	12	Si	13	P	14	S	15	Cl	16	Ar																																																																																																																				
4	K	18	Ca	19	Sc	20	Ti	21	V	22	Cr	23	Mn	24	Fe	25	Co	26	Ni	27	Cu	28	Zn	29	Ga	30	Ge	31	As	32	Se	33	Br	34	Kr																																																																																																
5	Rb	36	Sr	37	Y	38	Zr	39	Nb	40	Mo	41	Tc	42	Ru	43	Rh	44	Pd	45	Ag	46	Cd	47	In	48	Sn	49	Sb	50	Te	51	I	52	Xe																																																																																																
6	Cs	54	Ba	55	La	56	Hf	57	Ta	58	W	59	Re	60	Os	61	Ir	62	Pt	63	Au	64	Hg	65	Tl	66	Pb	67	Bi	68	Po	69	At	70	Rn	71	Fr	72	Ra	73	Ac	74	Ku	75	Th	76	Pa	77	U	78	Np	79	Pu	80	Am	81	Cm	82	Bk	83	Cf	84	Es	85	Fm	86	Md	87	No	88	Lr	89	Lu	90	Hf	91	Ta	92	W	93	Re	94	Os	95	Ir	96	Pt	97	Au	98	Hg	99	Tl	100	Pb	101	Bi	102	Po	103	At	104	Rn	105	Db	106	Sg	107	Bh	108	Hs	109	Mt	110	Ds	111	Rg	112	Cn	113	Nh	114	Fl	115	Mc	116	Lv	117	Ts	118	Og
7	Fr	80	Ra	81	Ac	82	Ku												83	Cf	84	Es	85	Fm	86	Md	87	No	88	Lr	89	Lu	90	Hf	91	Ta	92	W	93	Re	94	Os	95	Ir	96	Pt	97	Au	98	Hg	99	Tl	100	Pb	101	Bi	102	Po	103	At	104	Rn	105	Db	106	Sg	107	Bh	108	Hs	109	Mt	110	Ds	111	Rg	112	Cn	113	Nh	114	Fl	115	Mc	116	Lv	117	Ts	118	Og																																									

Ce	Pr	Nd	Pm	Sm	Eu	Gd	Tb	Dy	Ho	Er	Tm	Yb	Lu
58	59	60	61	62	63	64	65	66	67	68	69	70	71
140.90768	140.90768	140.90768	140.90768	150.35	151.964	157.25	158.92532	162.50010	164.93032	167.25930	168.93032	173.04470	174.96703
lanthanide	lanthanide	lanthanide	lanthanide	lanthanide	lanthanide	lanthanide	lanthanide	lanthanide	lanthanide	lanthanide	lanthanide	lanthanide	lanthanide

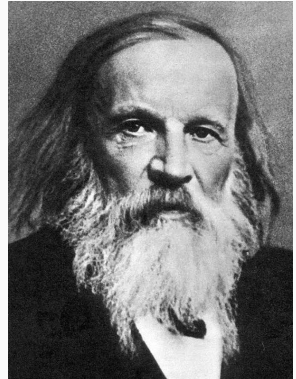


Figure 3: A Russian periodic table based on Dmitri Mendeleev's original table of 1869.

Carl Linnaeus

Species Plantarum lists every species of plant known at the time, classified into genera. It is the first work to consistently apply binomial names and was the starting point for the naming of plants.



1. Definitions
2. Postulates
3. Common notions



Common Notions

1. Things which are equal to the same thing are also equal to one other.
2. If equals be added to equals, the wholes are equal.
3. If equals be subruacted from equals, the remainders are equal.
4. Things which coincide with one another are equal to one another.
5. The whole is greater than the part.

Basic idea

The essence of generic programming lies in the idea of concepts. A concept is a way of describing a family of related object types.

Natural Science	Mathematics	Programming	Programming Examples
genus	theory	concept	Integral, Character
species	model	type or class	uint8_t, char
individual	element	instance	01000001(65, 'A')

Definitions

1. Datum
2. Value
3. Value type
4. Object
5. Object type

Definition

A **datum** is a sequence of bits.

Example

01000001 is an example of a datum.

Definition

A **value** is a **datum** together with its interpretation.

Example

The **datum** 01000001 might have the interpretation of the integer 65, or the character “A”.

Explanation

Every **value** must be associated with a **datum** in memory; there is no way to refer to disembodied **values** in modern programming languages.

Definition

A **value type** is a set of values sharing a common interpretation.

Definition

An **object** is a collection of bits in memory that contain a **value** of a given **value type**.

Explanation

An **object** is immutable if the value never changes, and mutable otherwise. An object is unrestricted if it can contain any **value** of its **value type**.

Definition

An **object type** is a uniform method of storing and retrieving **values** of a given **value type** from a particular **object** when given its address.

Journey

How algorithms were selected?

1. useful
2. generic
3. fits into 1 slide

Operation

1. Copy construction
2. Assignment
3. Destruction

Semantic

$$\forall a \forall b \forall c : T \ a(b) \implies (b = c \implies a = c)$$

$$\forall a \forall b \forall c : a \leftarrow b \implies (b = c \implies a = c)$$

$$\forall f \in \text{RegularFunction} : a = b \implies f(a) = f(b)$$

Semiregular

```
template<typename T>
concept semiregular = std::is_default_constructible<T>::value &&
                     std::is_copy_constructible<T>::value &&
                     std::is_copy_assignable<T>::value &&
                     std::is_destructible<T>::value
```

Operation

1. Copy construction
2. Assignment
3. Equality
4. Destruction

Semantic

$$\forall a \forall b \forall c : T \ a(b) \implies (b = c \implies a = c)$$

$$\forall a \forall b \forall c : a \leftarrow b \implies (b = c \implies a = c)$$

$$\forall f \in \text{RegularFunction} : a = b \implies f(a) = f(b)$$


```
template<typename T>  
concept semiregular = semiregular<T> && is_equality_comparable<T>::value;
```

Relation

FunctionalProcedure(F) \triangleq F is a regular procedure defined on regular types : replacing its inputs with equal objects results in equal output objects.

*UnaryFunction(F) \triangleq FunctionalProcedure(F) \wedge Arity(F) = 1
 \wedge Domain : UnaryFunction \rightarrow Regular
 $F \mapsto \text{InputType}(F, 0)$*

*HomogeneousFunction(F) \triangleq FunctionalProcedure(F) \wedge Arity(F) > 0
 $\wedge (\forall i, j \in \mathbb{N})(i, j < \text{Arity}(F)) \implies (\text{InputType}(F, i) = \text{InputType}(F, j))$
 \wedge Domain : HomogeneousFunction \rightarrow Regular
 $F \implies \text{InputType}(F, 0)$*

```
template<typename F, typename... T>  
concept functional_procedure = (regular<typename std::invoke_result<F, T  
...>::type> || std::is_same<typename std::invoke_result<F, T...>::  
type, void>::value) && is_regular<T...>::value;
```

```
template<typename F, typename T>  
concept unary_function = functional_procedure<F, T> && regular<T>;
```

```
template<typename F, typename... T>  
concept homogeneous_function = functional_procedure<F, T...> && sizeof  
...(T) > 0 && all_same<T...>::value && all_regular<T>;
```

$$\text{Predicate}(P) \triangleq \text{FunctionalProcedure}(F) \wedge \text{Codomain}(P) = \text{bool}$$

$$\text{HomogeneousPredicate}(P) \triangleq \text{Predicate}(P) \wedge \text{HomogeneousFunction}(P)$$

$$\text{Relation}(R) \triangleq \text{HomogeneousPredicate}(R) \wedge \text{Arity}(R) = 2$$

```
template<typename F, typename... T>  
concept predicate = functional_procedure<F, T...> && std::is_same<  
    codomain_t<F, T...>, bool>::value;
```

```
template<typename F, typename... T>  
concept homogeneous_predicate = predicate<F, T...> &&  
    homogeneous_function<F, T...>;
```

```
template<typename R, typename T>  
concept relation = predicate<R, T, T>;
```

Totally Ordered

property(*R* : *Relation*)

transitive : *R*

$r \mapsto (\forall a, b, c \in \text{Domain}(R))(r(a, b) \wedge r(b, c) \implies r(a, c))$

property(*R* : *Relation*)

total_ordering : *R*

$r \mapsto \text{transitive}(r) \wedge (\forall a, b \in \text{Domain}(R)) \text{ exactly one of following holds :}$

$r(a, b), r(b, a), \text{ or } a = b$

$\text{TotallyOrdered}(T) \triangleq \text{Regular}(T) \wedge <: T \times T \rightarrow \text{bool} \wedge \text{total_ordering}(<)$

Totally Ordered

```
template<typename T>  
concept totally_ordered = regular<T> && is_less_than_comprable<T>::value;
```

Journey 1

1. min
2. max

Journey #1

```
int min(int x, int y) {  
    if (y < x) {  
        return y;  
    }  
    return x;  
}
```

Journey #1

```
int min(int x, int y) {  
    if (y < x) {  
        return y;  
    }  
    return x;  
}
```

```
double min(double x, double y) {  
    if (y < x) {  
        return y;  
    }  
    return x;  
}
```

```
template<typename T>
T min(T x, T y) {
    if (y < x) {
        return y;
    }
    return x;
}
```

Dealing with large objects

```
template<typename T>
const T& min(const T& x, const T& y) {
    if (y < x) {
        return y;
    }
    return x;
}
```

Journey #1

```
template<typename T, typename P>
const T& min(const T& x, const T& y, P pred) {
    if (pred(y, x)) {
        return y;
    }
    return x;
}
```

Journey #1

```
struct employee {  
    std::string full_name;  
    int64_t salary;  
};  
  
void usage() {  
    employee e0{"Bjarne Stroustrup", 9999999ll};  
    employee e1{"Alex Stepanov", 9999999ll};  
    min(e0, e1, [](const auto& x, const auto& y) {  
        return x.salary < y.salary;  
    }).salary += 10000ll;  
}
```

```
template<typename T, typename P>
T& min(T& x, T& y, P pred) {
    if (pred(y, x)) {
        return y;
    }
    return x;
}
```

Journey #1

```
template<totally_ordered T>
const T& min(const T& x, const T& y) {
    if (y < x) {
        return y;
    }
    return x;
}
```

```
template<totally_ordered T>
T& min(T& x, T& y) {
    if (y < x) {
        return y;
    }
    return x;
}
```


Journey #1

```
template<typename T, weak_strict_ordering<T> R>
const T& min(const T& x, const T& y, R r) {
    if (r(y, x)) {
        return y;
    }
    return x;
}
```

```
template<typename T, weak_strict_ordering<T> R>
T& min(T& x, T& y, R r) {
    if (r(y, x)) {
        return y;
    }
    return x;
}
```

Journey #1

```
template<typename T, unary_function<T> Projection>  
requires totally_ordered<codomain_t<Projection, T>>  
const T& min(const T& x, const T& y, Projection projection) {  
    if (projection(y) < projection(x)) {  
        return y;  
    }  
    return x;  
}
```

```
template<typename T, unary_function<T> Projection>  
requires totally_ordered<codomain_t<Projection, T>>  
T& min(T& x, T& y, Projection projection) {  
    if (projection(y) < projection(x)) {  
        return y;  
    }  
    return x;  
}
```

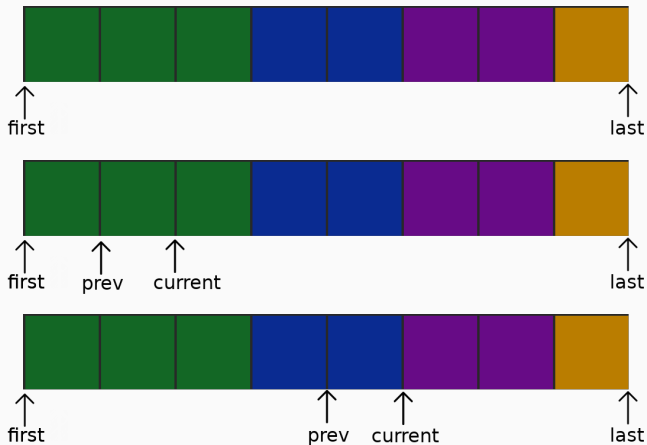
Journey #1

```
template<totally_ordered T>
const T& max(const T& x, const T& y) {
    if (y < x) {
        return x;
    }
    return y;
}
```

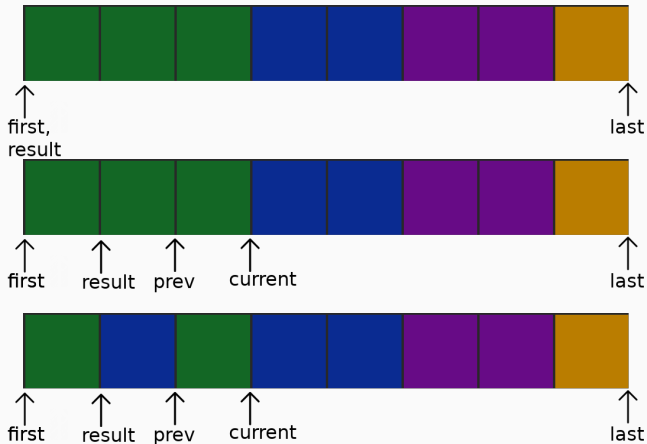
```
template<totally_ordered T>
T& max(T& x, T& y) {
    if (y < x) {
        return x;
    }
    return y;
}
```

1. `unique_count`
2. `unique_copy`
3. `unique`

Journey #2. Intuition behind unique_count and unique_copy



Journey #2. Intuition behind unique



$Iterator(T) \triangleq Regular(T) \wedge$
 $DistanceType : Iterator \rightarrow Integer \wedge$
 $successor : T \rightarrow T \wedge$
successor is not necessarily – regular

$ForwardIterator(T) \triangleq Iterator(T) \wedge regular_unary_function(successor)$

Concepts

```
template<typename I>
concept iterator = std::is_same<std::forward_iterator_tag, typename std::
    iterator_traits<I>::iterator_category>::value ||
    std::is_same<std::input_iterator_tag, typename std::
    iterator_traits<I>::iterator_category>::value ||
    std::is_same<std::output_iterator_tag, typename std::
    iterator_traits<I>::iterator_category>::value ||
    std::is_same<std::bidirectional_iterator_tag, typename
    std::iterator_traits<I>::iterator_category>::
    value ||
    std::is_same<std::random_access_iterator_tag, typename
    std::iterator_traits<I>::iterator_category>::
    value;
```

```
template<typename I>
concept forward_iterator = iterator<I> && std::is_base_of<std::
    forward_iterator_tag, typename std::iterator_traits<I>::
    iterator_category>::value;
```


$BidirectionalIterator(T) \triangleq ForwardIterator(T) \wedge$
 $predecessor : T \rightarrow T \wedge$
 $predecessor \text{ takes constant type } \wedge$
 $(\forall i \in T) \text{successor}(i) \text{ is defined} \implies$
 $predecessor(\text{successor}(i)) \text{ is defined}$
 $\text{and equals to } i \wedge$
 $(\forall i \in T) \text{predecessor}(i) \text{ is defined} \implies$
 $\text{successor}(\text{predecessor}(i)) \text{ is defined}$
 $\text{and equals to } i$

Concepts. Bidirectional Iterator

```
template<typename I>
concept bidirectional_iterator = iterator<I> && std::is_base_of<std::
    bidirectional_iterator_tag, typename std::iterator_traits<I>::
    iterator_category>::value;
```

Concepts. Indexed Iterator.

$\text{IndexedIterator}(T) \triangleq \text{ForwardIterator}(T) \wedge$
 $\quad + : T \times \text{DifferenceType}(T) \rightarrow T \wedge$
 $\quad - : T \times T \rightarrow \text{DifferenceType}(T) \wedge$
 $\quad + \text{ takes constant time}$
 $\quad - \text{ takes constant time}$

Concepts. Random Access Iterator

$\text{RandomAccessIterator}(T) \triangleq \text{BidirectionalIterator}(T) \wedge$

$\text{IndexedIterator}(T) \wedge$

$\text{TotallyOrdered}(T) \wedge$

$(\forall i, j \in T) i < j \iff i \prec j \wedge$

$\text{DifferenceType} :$

$\text{RandomAccessIterator} \rightarrow \text{Integer} \wedge$

$++ : T \times \text{DifferenceType}(T) \rightarrow T \wedge$

$-- : T \times \text{DifferenceType}(T) \rightarrow T \wedge$

$- : T \times T \rightarrow \text{DifferenceType}(T) \wedge$

$<$ takes constant time \wedge

$-$ between and iterator and an integer
takes constant time

Concepts. Random Access Iterator

```
template<typename I>
concept random_access_iterator= iterator<I> && std::is_base_of<std::
    random_access_iterator_tag, typename std::iterator_traits<I>::
    iterator_category >::value;
```

$$\begin{aligned} \text{Readable}(T) &\triangleq \text{Regular}(T) \wedge \\ &\quad \text{ValueType} : \text{Readable} \rightarrow \text{Regular} \wedge \\ &\quad \text{source} : T \rightarrow \text{ValueType}(T) \end{aligned}$$

```
template<typename T>
concept readable = std::is_same<decltype(*std::declval<T>()),
    value_type_t<T>&>::value || std::is_same<decltype(*std::declval<T>()),
    const value_type_t<T>&>::value;
```

$Writable(T) \triangleq \text{ValueType} : \text{Writable} \rightarrow \text{Regular} \wedge$
 $(\forall x \in T)(\forall v \in \text{ValueType}(T)) \text{ sink}(x) \leftarrow v$
is a well – formed statement

Concept: writable

```
template<typename T, typename U = ValueType(T)>  
concept writable = requires(T it, U x) { *it = x; };
```

Concept: writable

```
template<typename T>
concept additive_semigroup = regular<T> && std::is_same<decltype(T() + T
    ()), T>::value;
```

```
template<typename T>
concept additive_monoid = additive_semigroup<T>; // 0 \in T,
    identity_element(T);
```

unique_count

```
N unique_count(It first, It last, N n, R r) {  
    if (first == last) { return n; }  
    // some algorithm  
    return n;  
}
```

unique_count

```
N unique_count(It first, It last, N n, R r) {  
    if (first == last) { return n; }  
    It previous = first;  
    ++first;  
    // some algorithm  
    return n;  
}
```

unique_count

```
N unique_count(It first, It last, N n, R r) {  
    if (first == last) { return n; }  
    It previous = first;  
    ++first;  
    ++n;  
  
    if (!r(*previous, *first)) {  
        ++n;  
    }  
  
    return n;  
}
```

unique_count

```
N unique_count(It first, It last, N n, R r) {  
    if (first == last) { return n; }  
    It previous = first;  
    ++first;  
    while (first != last) {  
        if (!r(*previous, *first)) {  
            ++n;  
        }  
        ++previous;  
        ++first;  
    }  
    ++n;  
    return n;  
}
```

unique_count

```
template<forward_iterator It, additive_monoid N, relation<value_type_t<It
    >> R>
N unique_count(It first, It last, N n, R r) {
    if (first == last) { return n; }
    It previous = first;
    ++first;
    while (first != last) {
        if (!r(*previous, *first)) {
            ++n;
        }
        ++previous;
        ++first;
    }
    ++n;
    return n;
}
```

```
template<forward_iterator It, additive_monoid N>  
requires(readable<It>)  
N unique_count(It first, It last, N n) {  
    return unique_count(first, last, n, std::equal_to<value_type_t<It, It  
        >());  
}
```


unique_copy

```
template<forward_iterator It, output_iterator Out, relation<value_type_t<
    It>> R>
requires readable<It> && writable<Out, value_type_t<It>>
Out unique_copy(It first, It last, Out out, R r) {
    if (first == last) { return out; }
    *out = *first;
    ++out;
    It previous = first; ++first;
    while (first != last) {
        if (!r(*previous, *first)) {
            *out = *first;
            ++out;
        }
        ++first;
        ++previous;
    }
    return out;
}
```

```
template<forward_iterator It, output_iterator Out>
requires readable<It> && writable<Out, value_type_t<It>>
Out unique_copy(It first, It last, Out out) {
    return cppcon::unique_copy(first, last, out, std::equal_to<
        value_type_t<It>>());
}
```

unique

```
template<forward_iterator It, relation<value_type_t<It>> R>
requires readable<It> && writable<It>
It unique(It first, It last, R r) {
    if (first == last) { return last; }
    It result = first; ++first;
    while (first != last) {
        if (r(*result, *first)) {
            ++first;
        } else {
            ++result;
            *result = *first;
            ++first;
        }
    }
    ++result;
    return result;
}
```

Journey #2. unique

```
template<forward_iterator It>
requires(regular<value_type_t<It>> && readable<It> && writable<It>)
It unique(It first, It last) {
    return cppcon::unique(first, last, std::equal_to<value_type_t<It>
>>());
}
```

1. frequencies
2. transform_subgroups
3. squash_subgroups

Journey #3. frequencies

```
template<forward_iterator It, output_iterator Out>
requires readable<It> && writable<Out, std::pair<value_type_t<It>, size_t
    >>
Out frequencies(It f, It l, Out out) {
    typedef size_t N;
    while (f != l) {
        It it = f;
        N n = 1;
        ++f;
        auto r = find_not(f, l, *it, n);
        f = r.first;
        *out = { *it, r.second };
        ++out;
    }
    return out;
}
```

Journey #3. frequencies

```
template<forward_iterator It, output_iterator Out, relation<value_type_t<
    It>> R>
requires readable<It> && writable<Out, std::pair<value_type_t<It>, size_t
    >>
Out frequencies(It f, It l, Out out, R r) {
    typedef size_t N;
    while (f != l) {
        It start = f;
        N n = 1;
        ++f;
        auto r = find_if_not(f, l, *start, r, n);
        f = r.first;
        *out = { *start, r.second };
        ++out;
    }
    return out;
}
```

Journey #3. Frequencies

```
template<forward_iterator It, output_iterator Out0, output_iterator Out1>
requires readable<It> && writable<Out0, value_type_t<It>> && writable<
    Out1, size_t>
std::pair<Out0, Out1> frequencies(It f, It l, Out0 out0, Out1 out1) {
    typedef size_t N;

    while (f != l) {
        It start = f;
        N n = 1;
        ++f;
        auto r = find_not(f, l, *start, n);

        *out0 = *start;
        ++out0;

        *out1 = r.second;
        ++out1;

        f = r.first;
    }
    return {out0, out1};
}
```


Journey # 3. Frequencies

```
template<forward_iterator It, output_iterator Out0, forward_iterator Out1
>
requires readable<It> && writable<Out0> && writable<Out1>
std::pair<Out0, Out1> frequencies(It f, It l, Out0 out0, Out1 out1) {
    typedef value_type_t<Out1> N;

    while (f != l) {
        It it = f;
        N n = 1;
        ++f;
        It r = find_not(f, l, *it, n);

        *out0 = *it;
        ++out0;

        *out1 = r.second;
        ++out1;

        f = r.first;
    }
    return {out0, out1};
}
```

Journey # 3. Transform subgroups

```
template<forward_iterator It,
        output_iterator Out,
        relation<value_type_t<It>> P,
        functional_procedure<It, It> F>
requires readable<It> && writable<Out, codomain_t<F, It, It>>
Out transform_subgroups(It first, It last, Out out, P pred, F function) {
    if (first == last) { return out; }
    It previous = first;
    It fast = previous;
    ++fast;
    while (fast != last) {
        if (!pred(*previous, *fast)) {
            *out = function(first, fast);
            ++out;
            first = fast;
        }
        ++previous; ++fast;
    }
    *out = function(first, fast);
    return ++out;
}
```

Journey #3. Transform subgroups

```
template<forward_iterator It,
        output_iterator Out,
        relation<value_type_t<It>> P,
        functional_procedure<It, distance_type_t<It>> F>
requires readable<It> && writable<Out, codomain_t<F, It, distance_type_t<
    It>> >
Out transform_subgroups(It first, It last, Out out, P pred, F function) {
    if (first == last) { return out; }
    It previous = first;
    It fast = previous;
    ++fast;
    distance_type_t<It> n = 0;
    while (fast != last) {
        if (!pred(*previous, *fast)) {
            *out = function(first, n);
            n = 0;
            ++out;
            first = fast;
        }
        ++n;
        ++previous; ++fast;
    }
    *out = function(first, n);
    return ++out;
}
```

Journey # 3. Squash subgroups

```
template<forward_iterator It,
        output_iterator Out,
        relation<value_type_t<It>> Predicate,
        functional_procedure<It, It> F>
requires readable<It> && writable<Out, codomain_t<F, It, distance_type_t<
    It>>>
Out squash_subgroups(It first, It last, Predicate pred, F function) {
    return transform_subgroups(first, last, first, pred, function);
}
```

1. split
2. transform_splits

Journey #4. Intuition behind efficient split

Never was anything great achieved without danger.

↑ first ↑ last

This diagram illustrates a single split operation on the sentence. An upward arrow points to the first token 'Never', labeled 'first'. Another upward arrow points to the final token 'danger.', labeled 'last'.

Never was anything great achieved without danger.

↑ first ↑ middle ↑ last

This diagram illustrates a split operation with three points. Upward arrows point to the first token 'Never' (labeled 'first'), the middle token 'great' (labeled 'middle'), and the final token 'danger.' (labeled 'last').

Never was anything great achieved without danger.

↑ first ↑ middle ↑ last

This diagram illustrates a split operation with three points, similar to the previous one. Upward arrows point to the first token 'Never' (labeled 'first'), the middle token 'great' (labeled 'middle'), and the final token 'danger.' (labeled 'last').

Journey # 4. Split

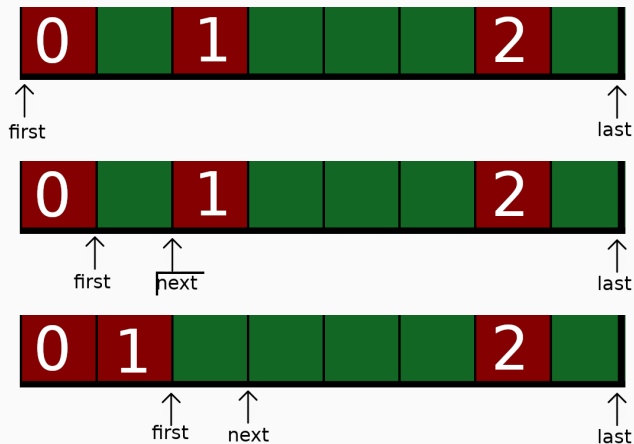
```
template<forward_iterator It0, forward_iterator It1, functional_procedure
    <It0, It0> F>
requires readable<It0> && readable<It1>
void split(It0 first0, It0 last0, It1 first1, It1 last1, F f) {
    while (first0 != last0) {
        It0 middle = std::search(first0, last0, first1, last1);
        f(first0, middle);
        first0 = middle;
        if (first0 != last0) {
            ++first0;
        }
    }
}
```

Journey # 4. Transform split

```
template<forward_iterator It0, output_iterator Out, forward_iterator It1,
        functional_procedure<It0, It0> F>
requires readable<It0> && readable<It1> && writable<Out, codomain_t<F,
        It0, It0>>
Out transform_split(It0 first0, It0 last0, Out out, It1 first1, It1 last1
, F f) {
    auto step_size = std::distance(first1, last1);
    while (first0 != last0) {
        It0 middle = std::search(first0, last0, first1, last1);
        *out = f(first0, middle);
        ++out;
        first0 = middle;
        if (first0 != last0) {
            std::advance(first0, step_size);
        }
    }
    return out;
}
```


1. remove
2. remove_if
3. partition_semistable

Journey #5. Intuition behind the scene



Journey # 5. Remove

```
template<forward_iterator It>
requires readable<It> && writable<It>
It remove(It first, It last, const value_type_t<It>& value) {
    first = std::find(first, last, value);
    if (first == last) { return last; }
    It fast = first; ++fast;
    while (fast != last) {
        if (*fast == value) {
            ++fast;
        } else {
            *first = std::move(*fast);
            ++first; ++fast;
        }
    }
    return first;
}
```

Journey # 5. Remove if

```
template<forward_iterator It, unary_predicate<value_type_t<It>> P>
requires readable<It> && writable<It>
It remove_if(It first, It last, P pred) {
    first = std::find_if(first, last, pred);
    if (first == last) { return last; }
    It fast = first; ++fast;
    while (fast != last) {
        if (pred(*fast)) {
            ++fast;
        } else {
            *first = std::move(*fast);
            ++first; ++fast;
        }
    }
    return first;
}
```

Journey # 5. Semistable Partition

```
template<forward_iterator It, unary_predicate<value_type_t<It>> P>
requires readable<It> && writable<It>
It semistable_partition(It first, It last, P pred) {
    first = std::find_if(first, last, pred);
    if (first == last) { return last; }
    It fast = first; ++fast;
    while (fast != last) {
        if (pred(*fast)) {
            ++fast;
        } else {
            std::swap(*first, *fast);
            ++first; ++fast;
        }
    }
    return first;
}
```

Journey # 5. Semistable Partition

```
template<forward_iterator It0, forward_iterator It1, binary_predicate<
    value_type_t<It0>, value_type_t<It1>> P>
requires (readable<It0> && writable<It0>) && (readable<It1> && writable<
    It1>)
std::pair<It0, It1> semistable_partition(It0 first0, It0 last0, It1
    first1, P pred) {
    std::pair<It0, It1> r = find_if(first0, last0, first1, pred);
    first0 = r.first; first1 = r.second;
    if (first0 == last0) { return {first0, first1}; }
    It0 fast0 = first0; ++fast0;
    It1 fast1 = first1; ++fast1;
    while (fast0 != last0) {
        if (pred(*fast0, *fast1)) {
            ++fast0; ++fast1;
        } else {
            std::swap(*first0, *fast0);
            ++first0; ++fast0;
            std::swap(*first1, *fast1);
            ++first1; ++fast1;
        }
    }
    return {first0, first1};
}
```

Demo

Conclusion

Conclusion

1. Concepts are mathematical. They are not specific to C++.
2. Know as many algorithms as you can.
3. Algorithms come in groups.
4. Transform complicated loops into well-defined algorithms.
5. Use mathematics for everything you do.
6. Don't obey mathematical conventions in programming.
7. Prefer fast concrete algorithms to more general where concreteness gives you better performance.
8. Have a little Euclid, Knuth, Dijkstra in your mind and let them argue.
9. It is good to organise your code.

```
while (first != last) {  
    // write good code  
}
```

Sources and Related Materials

1. The Thirteen Books of the Elements, Vol. 1: Books 1-2 2nd ed. Edition by Thomas L. Heath (Author), Euclid (Author).
2. From Mathematics to Generic Programming 1st Edition by Alexander A. Stepanov (Author), Daniel E. Rose (Author).
3. Elements of Programming 1st Edition by Alexander A. Stepanov (Author), Paul McJones (Author).
4. Species Plantarum by Carolus Linnaeus.
5. [Source code for the lecture.](#)
6. [The latest version of this presentation.](#)