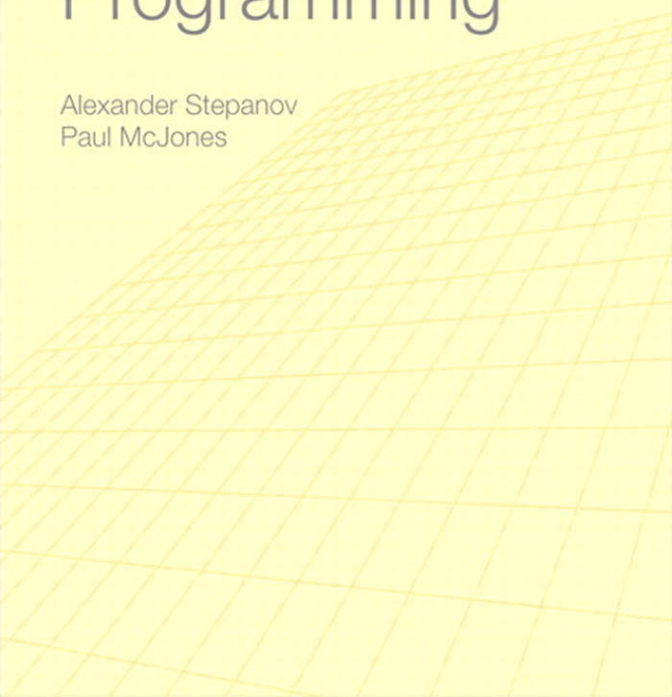




Elements of Programming

Alexander Stepanov
Paul McJones



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:
International Sales
international@pearson.com

Visit us on the Web: www.informit.com/aw

Library of Congress Cataloging-in-Publication Data

Stepanov, Alexander A.

Elements of programming/Alexander Stepanov, Paul McJones.
p. cm.

Includes bibliographical references and index.

ISBN 0-321-63537-X (hardcover : alk. paper)

1. Computer programming. 2. Computer algorithms. I. McJones, Paul. II. Title.

QA76.6.S726 2009

005.1-dc22

2009007604

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

ISBN-13: 978-0-321-63537-2

ISBN-10: 0-321-63537-X

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, June 2009

Preface

This book applies the deductive method to programming by affiliating programs with the abstract mathematical theories that enable them to work. Specification of these theories, algorithms written in terms of these theories, and theorems and lemmas describing their properties are presented together. The implementation of the algorithms in a real programming language is central to the book. While the specifications, which are addressed to human beings, should, and even must, combine rigor with appropriate informality, the code, which is addressed to the computer, must be absolutely precise even while being general.

As with other areas of science and engineering, the appropriate foundation of programming is the deductive method. It facilitates the decomposition of complex systems into components with mathematically specified behavior. That, in turn, is a necessary precondition for designing efficient, reliable, secure, and economical software.

The book is addressed to those who want a deeper understanding of programming, whether they are full-time software developers, or scientists and engineers for whom programming is an important part of their professional activity.

The book is intended to be read from beginning to end. Only by reading the code, proving the lemmas, and doing the exercises can readers gain understanding of the material. In addition, we suggest several projects, some open-ended. While the book is terse, a careful reader will eventually see the connections between its parts and the reasons for our choice of material. Discovering the architectural principles of the book should be the reader's goal.

We assume an ability to do elementary algebraic manipulations.¹ We also assume familiarity with the basic vocabulary of logic and set theory at the level of undergraduate courses on discrete mathematics; Appendix A summarizes the notation that we use. We provide definitions of a few concepts of abstract algebra when they are

1. For a refresher on elementary algebra, we recommend Chrystal [1904].

needed to specify algorithms. We assume programming maturity and understanding of computer architecture² and fundamental algorithms and data structures.³

We chose C++ because it combines powerful abstraction facilities with faithful representation of the underlying machine.⁴ We use a small subset of the language and write requirements as structured comments. We hope that readers not already familiar with C++ are able to follow the book. Appendix B specifies the subset of the language used in the book.⁵ Wherever there is a difference between mathematical notation and C++, the typesetting and the context determine whether the mathematical or C++ meaning applies. While many concepts and programs in the book have parallels in STL (the C++ Standard Template Library), the book departs from some of the STL design decisions. The book also ignores issues that a real library, such as STL, has to address: namespaces, visibility, inline directives, and so on.

Chapter 1 describes values, objects, types, procedures, and concepts. Chapters 2–5 describe algorithms on algebraic structures, such as semigroups and totally ordered sets. Chapters 6–11 describe algorithms on abstractions of memory. Chapter 12 describes objects containing other objects. The Afterword presents our reflections on the approach presented by the book.

Acknowledgments

We are grateful to Adobe Systems and its management for supporting the Foundations of Programming course and this book, which grew out of it. In particular, Greg Gilley initiated the course and suggested writing the book; Dave Story and then Bill Hensler provided unwavering support. Finally, the book would not have been possible without Sean Parent's enlightened management and continuous scrutiny of the code and the text. The ideas in the book stem from our close collaboration, spanning almost three decades, with Dave Musser. Bjarne Stroustrup deliberately evolved C++ to support these ideas. Both Dave and Bjarne were kind enough to come to San Jose and carefully review the preliminary draft. Sean Parent and Bjarne Stroustrup wrote the appendix defining the C++ subset used in the book. Jon Brandt reviewed multiple drafts of the book. John Wilkinson carefully read the final manuscript, providing innumerable valuable suggestions.

2. We recommend Patterson and Hennessy [2007].

3. For a selective but incisive introduction to algorithms and data structures, we recommend Tarjan [1983].

4. The standard reference is Stroustrup [2000].

5. The code in the book compiles and runs under Microsoft Visual C++ 9 and g++ 4. This code, together with a few trivial macros that enable it to compile, as well as unit tests, can be downloaded from www.elementsofprogramming.com.

The book has benefited significantly from the contributions of our editor, Peter Gordon, our project editor, Elizabeth Ryan, our copy editor, Evelyn Pyle, and the editorial reviewers: Matt Austern, Andrew Koenig, David Musser, Arch Robison, Jerry Schwarz, Jeremy Siek, and John Wilkinson.

We thank all the students who took the course at Adobe and an earlier course at SGI for their suggestions. We hope we succeeded in weaving the material from these courses into a coherent whole. We are grateful for comments from Dave Abrahams, Andrei Alexandrescu, Konstantine Arkoudas, John Banning, Hans Boehm, Angelo Borsotti, Jim Dehnert, John DeTreville, Boris Fomitchev, Kevlin Henney, Jussi Ketonen, Karl Malbrain, Mat Marcus, Larry Masinter, Dave Parent, Dmitry Polukhin, Jon Reid, Mark Ruzon, Geoff Scott, David Simons, Anna Stepanov, Tony Van Eerd, Walter Vannini, Tim Winkler, and Oleg Zabluda.

Finally, we are grateful to all the people who taught us through their writings or in person, and to the institutions that allowed us to deepen our understanding of programming.

Transformations and Their Orbits

This chapter defines a transformation as a unary regular function from a type to itself. Successive applications of a transformation starting from an initial value determine an orbit of this value. Depending only on the regularity of the transformation and the finiteness of the orbit, we implement an algorithm for determining orbit structures that can be used in different domains. For example, it could be used to detect a cycle in a linked list or to analyze a pseudorandom number generator. We derive an interface to the algorithm as a set of related procedures and definitions for their arguments and results. This analysis of an orbit-structure algorithm allows us to introduce our approach to programming in the simplest possible setting.

2.1 Transformations

While there are functions from any sequence of types to any type, particular classes of signatures commonly occur. In this book we frequently use two such classes: *homogeneous predicates* and *operations*. Homogeneous predicates are of the form $T \times \dots \times T \rightarrow \text{bool}$; operations are functions of the form $T \times \dots \times T \rightarrow T$. While there are n-ary predicates and n-ary operations, we encounter mostly unary and binary homogeneous predicates and unary and binary operations.

A *predicate* is a functional procedure returning a truth value:

$$\begin{aligned} \text{Predicate}(P) &\triangleq \\ &\text{FunctionalProcedure}(P) \\ &\wedge \text{Codomain}(P) = \text{bool} \end{aligned}$$

A homogeneous predicate is one that is also a homogeneous function:

$$\begin{aligned} \text{HomogeneousPredicate}(\mathbb{P}) &\triangleq \\ &\text{Predicate}(\mathbb{P}) \\ &\wedge \text{HomogeneousFunction}(\mathbb{P}) \end{aligned}$$

A *unary predicate* is a predicate taking one parameter:

$$\begin{aligned} \text{UnaryPredicate}(\mathbb{P}) &\triangleq \\ &\text{Predicate}(\mathbb{P}) \\ &\wedge \text{UnaryFunction}(\mathbb{P}) \end{aligned}$$

An *operation* is a homogeneous function whose codomain is equal to its domain:

$$\begin{aligned} \text{Operation}(\text{Op}) &\triangleq \\ &\text{HomogeneousFunction}(\text{Op}) \\ &\wedge \text{Codomain}(\text{Op}) = \text{Domain}(\text{Op}) \end{aligned}$$

Examples of operations:

```
int abs(int x) {
    if (x < 0) return -x; else return x;
} // unary operation
```

```
double euclidean_norm(double x, double y) {
    return sqrt(x * x + y * y);
} // binary operation
```

```
double euclidean_norm(double x, double y, double z) {
    return sqrt(x * x + y * y + z * z);
} // ternary operation
```

Lemma 2.1 $\text{euclidean_norm}(x, y, z) = \text{euclidean_norm}(\text{euclidean_norm}(x, y), z)$

This lemma shows that the ternary version can be obtained from the binary version. For reasons of efficiency, expressiveness, and, possibly, accuracy, the ternary version is part of the computational basis for programs dealing with three-dimensional space.

A procedure is *partial* if its definition space is a subset of the direct product of the types of its inputs; it is *total* if its definition space is equal to the direct product. We follow standard mathematical usage, where partial function includes total function. We call partial procedures that are not total *nontotal*. Implementations of some total functions are nontotal on the computer because of the finiteness of the representation. For example, addition on signed 32-bit integers is nontotal.

A nontotal procedure is accompanied by a precondition specifying its definition space. To verify the correctness of a call of that procedure, we must determine that the arguments satisfy the precondition. Sometimes, a partial procedure is passed as a parameter to an algorithm that needs to determine at runtime the definition space of the procedural parameter. To deal with such cases, we define a *definition-space predicate* with the same inputs as the procedure; the predicate returns true if and only if the inputs are within the definition space of the procedure. Before a nontotal procedure is called, either its precondition must be satisfied, or the call must be guarded by a call of its definition-space predicate.

Exercise 2.1 Implement a definition-space predicate for addition on 32-bit signed integers.

This chapter deals with unary operations, which we call *transformations*:

$Transformation(F) \triangleq$
 $Operation(F)$
 $\wedge UnaryFunction(F)$
 $\wedge DistanceType : Transformation \rightarrow Integer$

We discuss `DistanceType` in the next section.

Transformations are self-composable: $f(x)$, $f(f(x))$, $f(f(f(x)))$, and so on. The definition space of $f(f(x))$ is the intersection of the definition space and result space of f . This ability to self-compose, together with the ability to test for equality, allows us to define interesting algorithms.

When f is a transformation, we define its powers as follows:

$$f^n(x) = \begin{cases} x & \text{if } n = 0, \\ f^{n-1}(f(x)) & \text{if } n > 0 \end{cases}$$

To implement an algorithm to compute $f^n(x)$, we need to specify the requirement for an integer type. We study various concepts describing integers in Chapter 5. For now we rely on the intuitive understanding of integers. Their models include signed and unsigned integral types, as well as arbitrary-precision integers, with these operations and literals:

| | Specifications | C++ |
|------------|----------------|------|
| Sum | + | + |
| Difference | − | − |
| Product | · | * |
| Quotient | / | / |
| Remainder | mod | % |
| Zero | 0 | I(0) |
| One | 1 | I(1) |
| Two | 2 | I(2) |

where I is an integer type.

That leads to the following algorithm:

```
template<typename F, typename N>
    requires(Transformation(F) && Integer(N))
Domain(F) power_unary(Domain(F) x, N n, F f)
{
    // Precondition:  $n \geq 0 \wedge (\forall i \in \mathbb{N}) 0 < i \leq n \Rightarrow f^i(x)$  is defined
    while (n != N(0)) {
        n = n - N(1);
        x = f(x);
    }
    return x;
}
```

2.2 Orbits

To understand the global behavior of a transformation, we examine the structure of its *orbits*: elements reachable from a starting element by repeated applications of the transformation. y is *reachable* from x under a transformation f if for some $n \geq 0$, $y = f^n(x)$. x is *cyclic* under f if for some $n \geq 1$, $x = f^n(x)$. x is *terminal* under f if and only if x is not in the definition space of f . The *orbit* of x under a transformation f is the set of all elements reachable from x under f .

Lemma 2.2 An orbit does not contain both a cyclic and a terminal element.

Lemma 2.3 An orbit contains at most one terminal element.

If y is reachable from x under f , the *distance* from x to y is the least number of transformation steps from x to y . Obviously, distance is not always defined.

Given a transformation type F , $\text{DistanceType}(F)$ is an integer type large enough to encode the maximum number of steps by any transformation $f \in F$ from one element of $T = \text{Domain}(F)$ to another. If type T occupies k bits, there can be as many as 2^k values but only $2^k - 1$ steps between distinct values. Thus if T is a fixed-size type, an integral type of the same size is a valid distance type for any transformation on T . (Instead of using the distance type, we allow the use of any integer type in `power_unary`, since the extra generality does not appear to hurt there.) It is often the case that all transformation types over a domain have the same distance type. In this case the type function DistanceType is defined for the domain type and defines the corresponding type function for the transformation types.

The existence of DistanceType leads to the following procedure:

```
template<typename F>
  requires(Transformation(F))
DistanceType(F) distance(Domain(F) x, Domain(F) y, F f)
{
  // Precondition: y is reachable from x under f
  typedef DistanceType(F) N;
  N n(0);
  while (x != y) {
    x = f(x);
    n = n + N(1);
  }
  return n;
}
```

Orbits have different shapes. An orbit of x under a transformation is

| | |
|---------------------------------|---|
| <i>infinite</i> | if it has no cyclic or terminal elements |
| <i>terminating</i> | if it has a terminal element |
| <i>circular</i> | if x is cyclic |
| <i>ρ-shaped</i> | if x is not cyclic, but its orbit contains a cyclic element |

An orbit of x is *finite* if it is not infinite. Figure 2.1 illustrates the various cases.

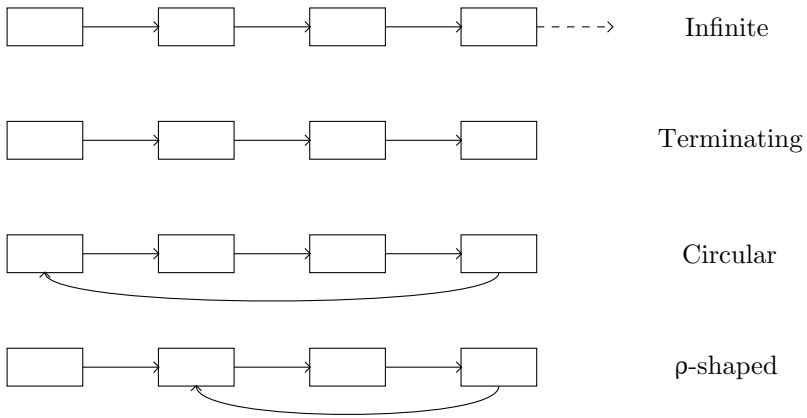


Figure 2.1 Orbit Shapes.

The *orbit cycle* is the set of cyclic elements in the orbit and is empty for infinite and terminating orbits. The *orbit handle*, the complement of the orbit cycle with respect to the orbit, is empty for a circular orbit. The *connection point* is the first cyclic element, and is the first element of a circular orbit and the first element after the handle for a ρ -shaped orbit. The *orbit size* o of an orbit is the number of distinct elements in it. The *handle size* h of an orbit is the number of elements in the orbit handle. The *cycle size* c of an orbit is the number of elements in the orbit cycle.

Lemma 2.4 $o = h + c$

Lemma 2.5 The distance from any point in an orbit to a point in a cycle of that orbit is always defined.

Lemma 2.6 If x and y are distinct points in a cycle of size c ,

$$c = \text{distance}(x, y, f) + \text{distance}(y, x, f)$$

Lemma 2.7 If x and y are points in a cycle of size c , the distance from x to y satisfies

$$0 \leq \text{distance}(x, y, f) < c$$

2.3 Collision Point

If we observe the behavior of a transformation, without access to its definition, we cannot determine whether a particular orbit is infinite: It might terminate or cycle back at any point. If we know that an orbit is finite, we can use an algorithm to determine the shape of the orbit. Therefore there is an implicit precondition of orbit finiteness for all the algorithms in this chapter.

There is, of course, a naive algorithm that stores every element visited and checks at every step whether the new element has been previously encountered. Even if we could use hashing to speed up the search, such an algorithm still would require linear storage and would not be practical in many applications. However, there is an algorithm that requires only a constant amount of storage.

The following analogy helps to understand the algorithm. If a fast car and a slow one start along a path, the fast one will catch up with the slow one if and only if there is a cycle. If there is no cycle, the fast one will reach the end of the path before the slow one. If there is a cycle, by the time the slow one enters the cycle, the fast one will already be there and will catch up eventually. Carrying our intuition from the continuous domain to the discrete domain requires care to avoid the fast one skipping past the slow one.¹

The discrete version of the algorithm is based on looking for a point where fast meets slow. The *collision point* of a transformation f and a starting point x is the unique y such that

$$y = f^n(x) = f^{2n+1}(x)$$

and $n \geq 0$ is the smallest integer satisfying this condition. This definition leads to an algorithm for determining the orbit structure that needs one comparison of fast and slow per iteration. To handle partial transformations, we pass a definition-space predicate to the algorithm:

```
template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
             Domain(F) == Domain(P))
Domain(F) collision_point(const Domain(F)& x, F f, P p)
{
    // Precondition: p(x) ⇔ f(x) is defined
    if (!p(x)) return x;
```

1. Knuth [1997, page 7] attributes this algorithm to Robert W. Floyd.

```

Domain(F) slow = x;           // slow = f0(x)
Domain(F) fast = f(x);       // fast = f1(x)
                               // n ← 0 (completed iterations)
while (fast != slow) {       // slow = fn(x) ∧ fast = f2n+1(x)
    slow = f(slow);          // slow = fn+1(x) ∧ fast = f2n+1(x)
    if (!p(fast)) return fast;
    fast = f(fast);          // slow = fn+1(x) ∧ fast = f2n+2(x)
    if (!p(fast)) return fast;
    fast = f(fast);          // slow = fn+1(x) ∧ fast = f2n+3(x)
                               // n ← n + 1
}
return fast;                  // slow = fn(x) ∧ fast = f2n+1(x)
// Postcondition: return value is terminal point or collision point
}

```

We establish the correctness of `collision_point` in three stages: (1) verifying that it never applies f to an argument outside the definition space; (2) verifying that if it terminates, the postcondition is satisfied; and (3) verifying that it always terminates.

While f is a partial function, its use by the procedure is well defined, since the movement of `fast` is guarded by a call of p . The movement of `slow` is unguarded, because by the regularity of f , `slow` traverses the same orbit as `fast`, so f is always defined when applied to `slow`.

The annotations show that if, after $n \geq 0$ iterations, `fast` becomes equal to `slow`, then $\text{fast} = f^{2n+1}(x)$ and $\text{slow} = f^n(x)$. Moreover, n is the smallest such integer, since we checked the condition for every $i < n$.

If there is no cycle, p will eventually return false because of finiteness. If there is a cycle, `slow` will eventually reach the connection point (the first element in the cycle). Consider the distance d from `fast` to `slow` at the top of the loop when `slow` first enters the cycle: $0 \leq d < c$. If $d = 0$, the procedure terminates. Otherwise the distance from `fast` to `slow` decreases by 1 on each iteration. Therefore the procedure always terminates; when it terminates, `slow` has moved a total of $h + d$ steps.

The following procedure determines whether an orbit is terminating:

```

template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
             Domain(F) == Domain(P))
bool terminating(const Domain(F)& x, F f, P p)

```

```

{
  // Precondition: p(x) ⇔ f(x) is defined
  return !p(collision_point(x, f, p));
}

```

Sometimes we know either that the transformation is total or that the orbit is nonterminating for a particular starting element. For these situations it is useful to have a specialized version of `collision_point`:

```

template<typename F>
  requires(Transformation(F))
Domain(F)
collision_point_nonterminating_orbit(const Domain(F)& x, F f)
{
  Domain(F) slow = x;           // slow = f0(x)
  Domain(F) fast = f(x);       // fast = f1(x)
                                // n ← 0 (completed iterations)
  while (fast != slow) {       // slow = fn(x) ∧ fast = f2n+1(x)
    slow = f(slow);           // slow = fn+1(x) ∧ fast = f2n+1(x)
    fast = f(fast);           // slow = fn+1(x) ∧ fast = f2n+2(x)
    fast = f(fast);           // slow = fn+1(x) ∧ fast = f2n+3(x)
                                // n ← n + 1
  }
  return fast;                 // slow = fn(x) ∧ fast = f2n+1(x)
  // Postcondition: return value is collision point
}

```

In order to determine the cycle structure—handle size, connection point, and cycle size—we need to analyze the position of the collision point.

When the procedure returns the collision point

$$f^n(x) = f^{2n+1}(x)$$

n is the number of steps taken by `slow`, and $2n + 1$ is the number of steps taken by `fast`.

$$n = h + d$$

where h is the handle size and $0 \leq d < c$ is the number of steps taken by slow inside the cycle. The number of steps taken by fast is

$$2n + 1 = h + d + qc$$

where $q \geq 0$ is the number of full cycles completed by fast when slow enters the cycle. Since $n = h + d$,

$$2(h + d) + 1 = h + d + qc$$

Simplifying gives

$$qc = h + d + 1$$

Let us represent h modulo c :

$$h = mc + r$$

with $0 \leq r < c$. Substitution gives

$$qc = mc + r + d + 1$$

or

$$d = (q - m)c - r - 1$$

$0 \leq d < c$ implies

$$q - m = 1$$

so

$$d = c - r - 1$$

and $r + 1$ steps are needed to complete the cycle.

Therefore the distance from the collision point to the connection point is

$$e = r + 1$$

In the case of a circular orbit $h = 0$, $r = 0$, and the distance from the collision point to the beginning of the orbit is

$$e = 1$$

Circularity, therefore, can be checked with the following procedures:

```
template<typename F>
    requires(Transformation(F))
bool circular_nonterminating_orbit(const Domain(F)& x, F f)
{
    return x == f(collision_point_nonterminating_orbit(x, f));
}

template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
             Domain(F) == Domain(P))
bool circular(const Domain(F)& x, F f, P p)
{
    // Precondition: p(x) ⇔ f(x) is defined
    Domain(F) y = collision_point(x, f, p);
    return p(y) && x == f(y);
}
```

We still don't know the handle size h and the cycle size c . Determining the latter is simple once the collision point is known: Traverse the cycle and count the steps.

To see how to determine h , let us look at the position of the collision point:

$$f^{h+d}(x) = f^{h+c-r-1}(x) = f^{mc+r+c-r-1}(x) = f^{(m+1)c-1}(x)$$

Taking $h + 1$ steps from the collision point gets us to the point $f^{(m+1)c+h}(x)$, which equals $f^h(x)$, since $(m + 1)c$ corresponds to going around the cycle $m + 1$ times. If we simultaneously take h steps from x and $h + 1$ steps from the collision point, we meet at the connection point. In other words, the orbits of x and 1 step past the collision point converge in exactly h steps, which leads to the following sequence of algorithms:

```
template<typename F>
    requires(Transformation(F))
Domain(F) convergent_point(Domain(F) x0, Domain(F) x1, F f)
{
    while (x0 != x1) {
```



```

        x0 = f(x0);
        x1 = f(x1);
    }
    return x0;
}

template<typename F>
    requires(Transformation(F))
Domain(F)
connection_point_nonterminating_orbit(const Domain(F)& x, F f)
{
    return convergent_point(
        x,
        f(collision_point_nonterminating_orbit(x, f)),
        f);
}

template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
            Domain(F) == Domain(P))
Domain(F) connection_point(const Domain(F)& x, F f, P p)
{
    // Precondition:  $p(x) \Leftrightarrow f(x)$  is defined
    Domain(F) y = collision_point(x, f, p);
    if (!p(y)) return y;
    return convergent_point(x, f(y), f);
}

```

Lemma 2.8 If the orbits of two elements intersect, they have the same cyclic elements.

Exercise 2.2 Design an algorithm that determines, given a transformation and its definition-space predicate, whether the orbits of two elements intersect.

Exercise 2.3 For `convergent_point` to work, it must be called with elements whose distances to the convergent point are equal. Implement an algorithm `convergent_point_guarded` for use when that is not known to be the case, but there is an element in common to the orbits of both.

2.4 Measuring Orbit Sizes

The natural type to use for the sizes o , h , and c of an orbit on type T would be an integer count type large enough to count all the distinct values of type T . If a type T occupies k bits, there can be as many as 2^k values, so a count type occupying k bits could not represent all the counts from 0 to 2^k . There is a way to represent these sizes by using distance type.

An orbit could potentially contain all values of a type, in which case o might not fit in the distance type. Depending on the shape of such an orbit, h and c would not fit either. However, for a ρ -shaped orbit, both h and c fit. In all cases each of these fits: $o - 1$ (the maximum distance in the orbit), $h - 1$ (the maximum distance in the handle), and $c - 1$ (the maximum distance in the cycle). That allows us to implement procedures returning a triple representing the complete structure of an orbit, where the members of the triple are as follows:

| Case | m0 | m1 | m2 |
|----------------|---------|---------|------------------|
| Terminating | $h - 1$ | 0 | terminal element |
| Circular | 0 | $c - 1$ | x |
| ρ -shaped | h | $c - 1$ | connection point |

```
template<typename F>
    requires(Transformation(F))
triple<DistanceType(F), DistanceType(F), Domain(F)>
orbit_structure_nonterminating_orbit(const Domain(F)& x, F f)
{
    typedef DistanceType(F) N;
    Domain(F) y = connection_point_nonterminating_orbit(x, f);
    return triple<N, N, Domain(F)>(distance(x, y, f),
                                   distance(f(y), y, f),
                                   y);
}

template<typename F, typename P>
    requires(Transformation(F) &&
             UnaryPredicate(P) && Domain(F) == Domain(P))
triple<DistanceType(F), DistanceType(F), Domain(F)>
orbit_structure(const Domain(F)& x, F f, P p)
{
    // Precondition: p(x) ⇔ f(x) is defined
```

```

typedef DistanceType(F) N;
Domain(F) y = connection_point(x, f, p);
N m = distance(x, y, f);
N n(0);
if (p(y)) n = distance(f(y), y, f);
// Terminating: m = h - 1 ∧ n = 0
// Otherwise: m = h ∧ n = c - 1
return triple<N, N, Domain(F)>(m, n, y);
}

```

Exercise 2.4 Derive formulas for the count of different operations (f , p , equality) for the algorithms in this chapter.

Exercise 2.5 Use `orbit_structure_nonterminating_orbit` to determine the average handle size and cycle size of the pseudorandom number generators on your platform for various seeds.

2.5 Actions

Algorithms often use a transformation f in a statement like

```
x = f(x);
```

Changing the state of an object by applying a transformation to it defines an *action* on the object. There is a duality between transformations and the corresponding actions: An action is definable in terms of a transformation, and vice versa:

```
void a(T& x) { x = f(x); } // action from transformation
```

and

```
T f(T x) { a(x); return x; } // transformation from action
```

Despite this duality, independent implementations are sometimes more efficient, in which case both action and transformation need to be provided. For example, if a transformation is defined on a large object and modifies only part of its overall state, the action could be considerably faster.

Exercise 2.6 Rewrite all the algorithms in this chapter in terms of actions.

Project 2.1 Another way to detect a cycle is to repeatedly test a single advancing element for equality with a stored element while replacing the stored element at ever-increasing intervals. This and other ideas are described in Sedgewick, et al. [1979], Brent [1980], and Levy [1982]. Implement other algorithms for orbit analysis, compare their performance for different applications, and develop a set of recommendations for selecting the appropriate algorithm.

2.6 Conclusions

Abstraction allowed us to define abstract procedures that can be used in different domains. Regularity of types and functions is essential to make the algorithms work: fast and slow follow the same orbit because of regularity. Developing nomenclature is essential (e.g., orbit kinds and sizes). Affiliated types, such as distance type, need to be precisely defined.

Index

\rightarrow (function), 231
– (additive inverse), in additive group, 67
 \wedge (and), 231
– (difference)
 in additive group, 67
 in cancellable monoid, 72
 of integers, 18
 of iterator and integer, 111
 of iterators, 93
 \times (direct product), 231
 \in (element), 231
= (equality), 7
 for array_k, 212
 for pair, 210
 \triangleq (equals by definition), 12, 231
 \Leftrightarrow (equivalent), 231
 \exists (exists), 231
 \forall (for all), 231
> (greater), 62
 \geq (greater or equal), 62
 \Rightarrow (implies), 231
[] (index)
 for array_k, 211
 for bounded_range, 214
 \neq (inequality), 7, 62
 \cap (intersection), 231
< (less), 62
 for array_k, 212
 natural total ordering, 61
 for pair, 210
 \leq (less or equal), 62
 \mapsto (maps to), 231
 \neg (not), 231
 \vee (or), 231
 \mathbf{a}^n (power of associative operation), 32
 f^n (power of transformation), 17
< (precedes), 95

\leq (precedes or equal), 95
 \cdot (product)
 of integers, 18
 in multiplicative semigroup, 66
 in semimodule, 69
/ (quotient), of integers, 18
[f, l] (range, closed bounded), 94
[[f, n]] (range, closed weak or counted), 94
[f, l) (range, half-open bounded), 94
[[f, n) (range, half-open weak or counted), 94
 \subset (subset), 231
+ (sum)
 in additive semigroup, 66
 of integers, 18
 of iterator and integer, 92
 \cup (union), 231

A

abs algorithm, 16, 71
absolute value, properties, 71
abstract entity, 1
abstract genus, 2
abstract procedure, 13
 overloading, 43
abstract species, 2
accumulation procedure, 46
accumulation variable
 elimination, 39
 introduction, 35
action, 28
acyclic descendants of bifurcate coordinate,
 116
additive inverse (–), in additive group, 67
AdditiveGroup concept, 67
AdditiveMonoid concept, 67
AdditiveSemigroup concept, 66

- address, 4
 - abstracted by iterator, 89
- add_to_counter algorithm, 199
- advance_tail machine, 135
- algorithm. *See* machine
 - abs, 16, 71
 - add_to_counter, 199
 - all, 97
 - bifurcate_compare, 131
 - bifurcate_compare_nonempty, 130
 - bifurcate_equivalent, 129
 - bifurcate_equivalent_nonempty, 128
 - bifurcate_isomorphic, 126
 - bifurcate_isomorphic_nonempty, 125
 - circular, 25
 - circular_nonterminating_orbit, 25
 - collision_point, 22
 - collision_point_nonterminating_orbit, 23
 - combine_copy, 160
 - combine_copy_backward, 162
 - combine_linked_nonempty, 138
 - combine_ranges, 196
 - compare_strict_or_reflexive, 57–58
 - complement, 50
 - complement_of_converse, 50
 - connection_point, 26
 - connection_point_nonterminating_orbit, 26
 - convergent_point, 26
 - converse, 50
 - copy, 152
 - copy_backward, 155
 - copy_bounded, 153
 - copy_if, 158
 - copy_n, 154
 - copy_select, 158
 - count_if, 97, 98
 - cycle_from, 173
 - cycle_to, 173
 - distance, 19
 - euclidean_norm, 16
 - exchange_values, 164
 - fast_subtractive_gcd, 78
 - fibonacci, 46
 - find, 96
 - find_adjacent_mismatch, 103
 - find_adjacent_mismatch_forward, 106, 135
 - find_backward_if, 112
 - find_if, 97
 - find_if_not_unguarded, 102
 - find_if_unguarded, 101
 - find_last, 136
 - find_mismatch, 102
 - find_n, 101
 - find_not, 97
 - for_each, 96
 - for_each_n, 101
 - gcd, 80
 - height, 122
 - height_recursive, 118
 - increment, 91
 - is_left_successor, 119
 - is_right_successor, 120
 - k_rotate_from_permutation_indexed, 180
 - k_rotate_from_permutation_random_
 - access, 180
 - largest_doubling, 75
 - lexicographical_compare, 129
 - lexicographical_equal, 127
 - lexicographical_equivalent, 127
 - lexicographical_less, 130
 - lower_bound_n, 109
 - lower_bound_predicate, 108
 - median_5, 61
 - memory-adaptive, 177
 - merge_copy, 163
 - merge_copy_backward, 163
 - merge_linked_nonempty, 141
 - merge_n_adaptive, 206
 - merge_n_with_buffer, 202
 - none, 97
 - not_all, 97
 - orbit_structure, 28
 - orbit_structure_nonterminating_orbit, 27
 - partitioned_at_point, 191
 - partition_bidirectional, 194
 - partition_copy, 160
 - partition_copy_n, 160
 - partition_linked, 140
 - partition_point, 107
 - partition_point_n, 107
 - partition_semistable, 192
 - partition_single_cycle, 194
 - partition_stable_iterative, 201
 - partition_stable_n, 197
 - partition_stable_n_adaptive, 197
 - partition_stable_n_nonempty, 197

- algorithm. *See* machine (*cont.*)
 - partition_stable_singleton, 196
 - partition_stable_with_buffer, 195
 - partition_trivial, 198
 - phased_applicator, 147
 - potential_partition_point, 191
 - power, 42
 - power_accumulate, 41
 - power_accumulate_positive, 41
 - power_left_associated vs. power_0, 34
 - power_right_associated, 33
 - power_unary, 18
 - predicate_source, 140
 - quotient_remainder, 85
 - quotient_remainder_nonnegative, 82
 - quotient_remainder_nonnegative_iterative, 83
 - reachable, 121
 - reduce, 99
 - reduce_balanced, 200
 - reduce_nonempty, 99
 - reduce_nonzeroes, 100
 - relation_source, 141
 - remainder, 84
 - remainder_nonnegative, 74
 - remainder_nonnegative_iterative, 75
 - reverse_append, 139, 140
 - reverse_bidirectional, 175
 - reverse_copy, 156
 - reverse_copy_backward, 156
 - reverse_indexed, 186
 - reverse_n_adaptive, 178
 - reverse_n_bidirectional, 175
 - reverse_n_forward, 177
 - reverse_n_indexed, 175
 - reverse_n_with_buffer, 176
 - reverse_swap_ranges, 167
 - reverse_swap_ranges_bounded, 167
 - reverse_swap_ranges_n, 168
 - reverse_with_temporary_buffer, 187, 225
- rotate, 187
- rotate_bidirectional_nontrivial, 182
- rotate_cycles, 181
- rotate_forward_annotated, 183
- rotate_forward_nontrivial, 184
- rotate_forward_step, 184
- rotate_indexed_nontrivial, 181
- rotate_nontrivial, 188
- rotate_partial_nontrivial, 185
- rotate_random_access_nontrivial, 181
- rotate_with_buffer_backward_nontrivial, 186
- rotate_with_buffer_nontrivial, 185
- select_0_2, 53, 63
- select_0_3, 54
- select_1_2, 54
- select_1_3, 55
- select_1_3_ab, 55
- select_1_4, 56, 59
- select_1_4_ab, 56, 59
- select_1_4_ab_cd, 56, 58
- select_2_3, 54
- select_2_5, 60
- select_2_5_ab, 60
- select_2_5_ab_cd, 59
- slow_quotient, 73
- slow_remainder, 72
- some, 97
- sort_linked_nonempty_n, 142
- sort_n, 207
- sort_n_adaptive, 207
- sort_n_with_buffer, 203
- split_copy, 158
- split_linked, 137
- subtractive_gcd, 78
- subtractive_gcd_nonzero, 77
- swap, 224
- swap_basic, 223
- swap_ranges, 165
- swap_ranges_bounded, 166
- swap_ranges_n, 166
- terminating, 23
- transpose_operation, 201
- traverse, 123
- traverse_nonempty, 118
- traverse_phased_rotating, 148
- traverse_rotating, 146
- underlying_ref, 224
- upper_bound_n, 109
- upper_bound_predicate, 109
- weight, 122
- weight_recursive, 117
- weight_rotating, 147
- aliased property, 150
- aliased write-read, 150
- aliased write-write, 159

all algorithm, 97
 ambiguous value type, 3
 amortized complexity, 219
 and (\wedge), 231
 annihilation property, 68
 annotation variable, 183
ArchimedeanGroup concept, 83
ArchimedeanMonoid concept, 72
 area of object, 227
 Aristotle, 77
 Arity type attribute, 11
 array, varieties, 220–221
 array_k type, 210
 Artin, Emil, 13
 assignment, 7
 for array_k, 211
 for pair, 210
 associative operation, 31, 98
 power of (a^n), 32
 associative property, 31
 exploited by power, 33
 partially_associative, 98
 of permutation composition, 170
 asymmetric property, 50
 attribute, 1
 auxiliary computation during recursion, 176
 Axiom of Archimedes, 72, 73

B

backward movement in range, 112
BackwardLinker concept, 134
 backward_offset property, 161
 basic singly linked list, 218
 begin
 for array_k, 211
 for bounded_range, 214
 for *Linearizable*, 213
 behavioral equality, 3, 228
BidirectionalBifurcateCoordinate concept,
 119–120
BidirectionalIterator concept, 111
BidirectionalLinker concept, 134
BifurcateCoordinate concept, 115
 bifurcate_compare algorithm, 131
 bifurcate_compare_nonempty algorithm, 130
 bifurcate_equivalent algorithm, 129
 bifurcate_equivalent_nonempty algorithm,
 128

bifurcate_isomorphic algorithm, 126
 bifurcate_isomorphic_nonempty algorithm,
 125
BinaryOperation concept, 31
 binary_scale_down_nonnegative, 41
 binary_scale_up_nonnegative, 41
 bisection technique, 107
 Bolzano, Bernard, 107
 bounded integer type, 87
 bounded range, 93
 bounded_range property, 93
 bounded_range type, 214
 Brandt, Jon, 193

C

CancellableMonoid concept, 72
 cancellation in monoid, 72
 categories of ideas, 1
 Cauchy, Augustin Louis, 107
 circular algorithm, 25
 circular array, 220
 circular doubly linked list, 218
 circular singly linked list, 218
 circular_nonterminating_orbit algorithm, 25
 closed bounded range ($[f, l]$), 94
 closed interval, 231
 closed weak or counted range ($[f, n]$), 94
 clusters of derived procedures, 62
 codomain, 10
 Codomain type function, 11
 Collins, George, 13
 collision point of orbit, 21
 collision_point algorithm, 22
 collision_point_nonterminating_orbit
 algorithm, 23
 combine_copy algorithm, 160
 combine_copy_backward algorithm, 162
 combine_linked_nonempty algorithm, 138
 combine_ranges algorithm, 196
 common-subexpression elimination, 35
 commutative property, 66
CommutativeRing concept, 69
CommutativeSemiring concept, 68
 compare_strict_or_reflexive algorithm,
 57–58
 complement algorithm, 50
 complement of converse of relation, 50
 complement of relation, 50

- complement_of_converse algorithm, 50
- complement_of_converse property, 104
- complexity
 - amortized, 219
 - of `empty`, 213
 - of indexing of a sequence, 213
 - of regular operations, 227
 - of source, 90
 - of successor, 92
- composite object, 215
- composition
 - of permutations, 170
 - of transformations, 17, 32
- computational basis, 6
- concept, 11
 - AdditiveGroup*, 67
 - AdditiveMonoid*, 67
 - AdditiveSemigroup*, 66
 - ArchimedeanGroup*, 83
 - ArchimedeanMonoid*, 72
 - BackwardLinker*, 134
 - BidirectionalBifurcateCoordinate*, 119–120
 - BidirectionalIterator*, 111
 - BidirectionalLinker*, 134
 - BifurcateCoordinate*, 115
 - BinaryOperation*, 31
 - CancellableMonoid*, 72
 - CommutativeRing*, 69
 - CommutativeSemiring*, 68
 - consistent, 87
 - DiscreteArchimedeanRing*, 86
 - DiscreteArchimedeanSemiring*, 85
 - EmptyLinkedBifurcateCoordinate*, 144
 - EuclideanMonoid*, 77
 - EuclideanSemimodule*, 80
 - EuclideanSemiring*, 79
 - examples from C++ and STL, 11
 - ForwardIterator*, 106
 - ForwardLinker*, 133
 - FunctionalProcedure*, 11
 - HalvableMonoid*, 74
 - HomogeneousFunction*, 12
 - HomogeneousPredicate*, 16
 - IndexedIterator*, 110
 - Integer*, 18, 40
 - Iterator*, 91
 - Linearizable*, 213
 - LinkedBifurcateCoordinate*, 144
 - modeled by type, 11
 - Module*, 70
 - MultiplicativeGroup*, 68
 - MultiplicativeMonoid*, 67
 - MultiplicativeSemigroup*, 66
 - NonnegativeDiscreteArchimedeanSemiring*, 86
 - Operation*, 16
 - OrderedAdditiveGroup*, 70
 - OrderedAdditiveMonoid*, 70
 - OrderedAdditiveSemigroup*, 70
 - Predicate*, 15
 - RandomAccessIterator*, 113
 - refinement, 11
 - Regular*, 11
 - Relation*, 49
 - relational concept, 69
 - Ring*, 69
 - Semimodule*, 69
 - Semiring*, 68
 - Sequence*, 216
 - TotallyOrdered*, 62
 - Transformation*, 17
 - type concept, 11
 - UnaryFunction*, 12
 - UnaryPredicate*, 16
 - univalent, 86
 - useful, 87
 - weakening, 11
- concept dispatch, 106, 187
- concept schema
 - composite object, 216
 - coordinate structure, 124
- concept tag type, 187
- concrete entity, 1
- concrete genus, 2
- concrete species, 2
- connectedness of composite object, 215
- connection point of orbit, 20
- connection_point algorithm, 26
- connection_point_nonterminating_orbit algorithm, 26
- connectors, 229
- consistency of concept's axioms, 87
- constant-size sequence, 216
- constructor, 7
- container, 213
- convergent_point algorithm, 26

converse algorithm, 50
 converse of relation, 50
 coordinate structure
 bifurcate coordinate, 115
 of composite object, 215
 concept schema, 124
 iterator, 89
 copy algorithm, 152
 copy constructor, 8
 for `array_k`, 211
 for `pair`, 210
 copy of object, 5
 copying rearrangement, 172
`copy_backward` algorithm, 155
`copy_backward_step` machine, 154
`copy_bounded` algorithm, 153
`copy_if` algorithm, 158
`copy_n` algorithm, 154
`copy_select` algorithm, 158
`copy_step` machine, 152
`counted_range` property, 93
`counter_machine` type, 200
`count_down` machine, 153
`count_if` algorithm, 97, 98
 cycle detection intuition, 21
 cycle in a permutation, 171
 cycle of orbit, 20
 cycle size, 20
`cycle_from` algorithm, 173
`cycle_to` algorithm, 173
 cyclic element under transformation, 18
 cyclic permutation, 171

D

DAG (directed acyclic graph), 116
 datum, 2
 de Bruijn, N. G., 74
 default constructor, 8
 for `array_k`, 211
 for `pair`, 209
 default ordering, 62
 default total ordering, 62
 importance of, 228
 definition space, 9
 definition-space predicate, 17
 dependence of axiom, 86
 deref, 150
 derived relation, 50

descendant of bifurcate
 coordinate, 116
 destructor, 7
 for `pair`, 210
 difference (`-`)
 in additive group, 67
 in cancellable monoid, 72
 of iterator and integer, 111
 of iterators, 93
`DifferenceType` type function, 113
 direct product (\times), 231
 directed acyclic graph, 116
DiscreteArchimedeanRing concept, 86
DiscreteArchimedeanSemiring concept, 85
 discreteness property, 85
 disjoint property, 134
 disjointness of composite object, 216
 distance algorithm, 19
 distance in orbit, 19
`DistanceType` type function, 17, 91
 distributive property, holds for semiring,
 68
 divisibility on an Archimedean monoid,
 76
 division, 68
 domain, 10
`Domain` type function, 12
 double-ended array, 220
 doubly linked list, 218–219
 Dudziński, Krzysztof, 206
 dummy node doubly linked list, 218
 Dydek, Andrzej, 206
 dynamic-size sequence, 216

E

efficient computational basis, 6
 element (\in), 231
 eliminating common subexpression, 35
`empty`
 for `array_k`, 212
 for `bounded_range`, 214
 for *Linearizable*, 213
 empty coordinate, 144
 empty range, 95
EmptyLinkedBifurcateCoordinate
 concept, 144
 end

- for `array_k`, 211
- for `bounded_range`, 214
- for *Linearizable*, 213
- entity, 1
- equality
 - =, 7
 - ≠, 62
 - for `array_k`, 212
 - behavioral, 3, 228
 - equal** for *Regular*, 127
 - for objects, 5
 - for **pair**, 210
 - for regular type, 7
 - representational, 3, 228
 - structural, 228
 - for uniquely represented type, 3
 - for value type, 3
- equals by definition (\triangleq), 12, 231
- equational reasoning:, 4
- equivalence class, 51
- equivalence property, 51
- equivalent (\Leftrightarrow), 231
- equivalent coordinate collections, 126
- erasure in a sequence, 217
- Euclidean function, 79
- EuclideanMonoid* concept, 77
- EuclideanSemimodule* concept, 80
- EuclideanSemiring* concept, 79
- `eulerian_norm` algorithm, 16
- even, 41
- `exchange_values` algorithm, 164
- exists (\exists), 231
- expressive computational basis, 6

F

- `fast_subtractive_gcd` algorithm, 78
- fibonacci algorithm, 46
- Fibonacci sequence, 45
- find algorithm, 96
- `find_adjacent_mismatch` algorithm, 103
- `find_adjacent_mismatch_forward` algorithm, 106, 135
- `find_backward_if` algorithm, 112
- `find_if` algorithm, 97
- `find_if_not`, 97
- `find_if_not_unguarded` algorithm, 102
- `find_if_unguarded` algorithm, 101
- `find_last` algorithm, 136

- `find_mismatch` algorithm, 102
- `find_n` algorithm, 101
- `find_not` algorithm, 97
- finite order, under associative operation, 32
- finite set, 171
- first-last singly linked list, 218
- fixed point of transformation, 170
- fixed-size sequence, 216
- Floyd, Robert W., 21
- for all (\forall), 231
- ForwardIterator* concept, 106
- ForwardLinker* concept, 133
- `forward_offset` property, 162
- `for_each` algorithm, 96
- `for_each_n` algorithm, 101
- Frobenius, Georg Ferdinand, 32
- from-permutation, 172
- function, 2
 - , 231
 - on abstract entities, 2
 - on values, 3
- function object, 9, 96, 236
- functional procedure, 9
- FunctionalProcedure* concept, 11

G

- garbage collection, 230
- Gaussian integers, 40
 - Stein's algorithm, 81
- `gcd`, 76
 - Stein, 81
 - subtractive, 76
- `gcd` algorithm, 80
- genus, 2
- global state, 6
- `goto` statement, 148
- greater ($>$), 62
- greater or equal (\geq), 62
- greatest common divisor (`gcd`), 76
- group, 67
 - of permutations, 170

H

- `half_nonnegative`, 41
- half-open bounded range ($[f, l)$), 94
- half-open interval, 231
- half-open weak or counted range ($[f, n)$), 94
- HalvableMonoid* concept, 74

handle of orbit, 20
 handle size, 20
 header of composite object, 217
 height algorithm, 122
 height of bifurcate coordinate (DAG), 116
 height_recursive algorithm, 118
 Ho, Wilson, 182
 Hoare, C. A. R., 195
 homogeneous functional procedure, 10
HomogeneousFunction concept, 12
HomogeneousPredicate concept, 16

I

ideas, categories of, 1
 identity
 of concrete entity, 1
 of object, 5
 identity element, 65
 identity token, 5
 identity transformation, 170
 identity_element property, 65
 implies (\Rightarrow), 231
 inconsistency of concept, 87
 increasing range, 103
 increasing_counted_range property, 105
 increasing_range property, 105
 increment algorithm, 91
 independence of proposition, 86
 index ([])
 for array_k, 211
 for bounded_range, 214
 index permutation, 172
 index of segmented array, 221
 indexed iterator
 equivalent to random-access iterator, 113
IndexedIterator concept, 110
 inequality (\neq), 7
 standard definition, 62
 inorder, 118
 input object, 6
 input/output object, 6
 InputType type function, 11
 insertion in a sequence, 217
Integer concept, 18, 40
 interpretation, 2
 intersection (\cap), 231
 interval, 231
 into transformation, 169

invariant, 148
 loop, 37
 recursion, 36
 inverse of permutation, 170, 171
 inverse_operation property, 66
 isomorphic coordinate sets, 124
 isomorphic types, 86
 is_left_successor algorithm, 119
 is_right_successor algorithm, 120
 iterator adapter
 for bidirectional bifurcate coordinates,
 project, 124
 random access from indexed, 114
 reverse from bidirectional, 112
 underlying type, 224
Iterator concept, 91
 iterator invalidation in array, 221
 IteratorConcept type function, 187
 IteratorType type function, 133, 134, 213

K

Kislitsyn, Sergei, 55
 k_rotate_from_permutation_indexed
 algorithm, 180
 k_rotate_from_permutation_random_access
 algorithm, 180

L

Lagrange, J.-L., 107
 Lakshman, T. K., 159
 largest_doubling algorithm, 75
 less ($<$), 62
 for array_k, 212
 for bounded_range, 215
 less for *TotallyOrdered*, 130
 natural total ordering, 61
 for pair, 210
 less or equal (\leq), 62
 lexicographical_compare algorithm, 129
 lexicographical_equal algorithm, 127
 lexicographical_equivalent algorithm, 127
 lexicographical_less algorithm, 130
 limit in a range, 95
 linear ordering, 52
Linearizable concept, 213
 link rearrangement, 134
 on lists, 219
 linked iterator, 133

linked structures, forward vs. bidirectional, 219
LinkedBifurcateCoordinate concept, 144
 linker object, 133
 linker_to_head machine, 139
 linker_to_tail machine, 135
 links, reversing, 145
 list

- doubly linked, 218
- singly linked, 218

 Lo, Raymond, 182
 load, 4
 local part of composite object, 217
 local state, 6
 locality of reference, 143
 loop invariant, 37
 lower bound, 107
 lower_bound_n algorithm, 109
 lower_bound_predicate algorithm, 108

M

machine, 120

- advance_tail, 135
- copy_backward_step, 154
- copy_step, 152
- count_down, 153
- linker_to_head, 139
- linker_to_tail, 135
- merge_n_step_0, 205
- merge_n_step_1, 205
- reverse_copy_backward_step, 156
- reverse_copy_step, 155
- reverse_swap_step, 166
- swap_step, 165
- traverse_step, 121
- tree_rotate, 145

 maps to (\mapsto), 231
 marking, 118
 Mauchly, John W., 107
 median_5 algorithm, 61
 memory, 4
 memory-adaptive algorithm, 177
 merge, stability, 203
 mergeable property, 203
 merge_copy algorithm, 163
 merge_copy_backward algorithm, 163
 merge_linked_nonempty algorithm, 141
 merge_n_adaptive algorithm, 206

merge_n_step_0 machine, 205
 merge_n_step_1 machine, 205
 merge_n_with_buffer algorithm, 202
 mod (remainder), 18
 model, partial, 70
 models, 11
Module concept, 70
 monoid, 67
 multipass traversal, 106
MultiplicativeGroup concept, 68
MultiplicativeMonoid concept, 67
MultiplicativeSemigroup concept, 66
 multiset, 227
 Musser, David, 13
 mutable range, 151
 mutable_bounded_range property, 151
 mutable_counted_range property, 151
 mutable_weak_range property, 151
 mutative rearrangement, 172

N

natural total ordering, < reserved for, 61
 negative, 41
 nil, 134
 Noether, Emmy, 13
 noncircularity of composite object, 216
 none algorithm, 97
NonnegativeDiscreteArchimedeanSemiring concept, 86
 nontotal procedure, 17
 not (\neg), 231
 not_all algorithm, 97
 not_overlapped property, 157
 not_overlapped_backward property, 155
 not_overlapped_forward property, 153
 not_write_overlapped property, 159
 null link, 218

O

object, 4

- area, 227
- equality, 5
- starting address, 216
- state, 4

 object type, 4
 odd, 41
 one, 41
 one-to-one transformation, 169

onto transformation, 169
 open interval, 231
Operation concept, 16
 or (\vee), 231
 orbit, 18–20
 orbit_structure algorithm, 28
 orbit_structure_nonterminating_orbit
 algorithm, 27
OrderedAdditiveGroup concept, 70
OrderedAdditiveMonoid concept, 70
OrderedAdditiveSemigroup concept, 70
 ordering, linear, 52
 ordering-based rearrangement, 172
 output object, 6
 overloading, 43, 133, 144
 own state, 6
 ownership, of parts by composite
 object, 216

P

pair type, 11, 209
 parameter passing, 9
 part of composite object, 215–219
 partial model, 70
 partial procedure, 17
 partial (usage convention), 232
 partially formed object state, 7
 partially_associative property, 98
 partition algorithm, origin of, 195
 partition point, 105
 lower and upper bounds, 107
 partition rearrangement, semistable, 192
 partitioned property, 105
 partitioned range, 105
 partitioned_at_point algorithm, 191
 partition_bidirectional algorithm, 194
 partition_copy algorithm, 160
 partition_copy_n algorithm, 160
 partition_linked algorithm, 140
 partition_point algorithm, 107
 partition_point_n algorithm, 107
 partition_semistable algorithm, 192
 partition_single_cycle algorithm, 194
 partition_stable_iterative algorithm, 201
 partition_stable_n algorithm, 197
 partition_stable_n_adaptive algorithm, 197
 partition_stable_n_nonempty
 algorithm, 197

partition_stable_singleton algorithm, 196
 partition_stable_with_buffer algorithm, 195
 partition_trivial algorithm, 198
 permanently placed part of composite object,
 217
 permutation, 170
 composition, 170
 cycle, 171
 cyclic, 171
 from, 172
 index, 172
 inverse, 170, 171
 product of its cycles, 171
 reverse, 174
 rotation, 178
 to, 172
 transposition, 171
 permutation group, 170
 phased_applicator algorithm, 147
 pivot, 205
 position-based rearrangement, 172
 positive, 41
 postorder, 118
 potential_partition_point algorithm, 191
 power
 of associative operation (\mathbf{a}^n), 32
 powers of same element commute, 32
 of transformation (f^n), 17
 power algorithm, 42
 operation count, 34
 power_accumulate algorithm, 41
 power_accumulate_positive algorithm, 41
 power_right_associated algorithm, 33
 power_unary algorithm, 18
 precedence preserving link rearrangement,
 135
 precedes (\prec), 95
 precedes or equal (\preceq), 95
 precondition, 13
 predecessor
 of integer, 41
 of iterator, 111
Predicate concept, 15
 predicate-based rearrangement, 172
 predicate_source algorithm, 140
 prefix of extent, 220
 preorder, 118
 prime property, 14

- procedure, 6
 - abstract, 13
 - functional, 9
 - nontotal, 17
 - partial, 17
 - total, 17
- product (·)
 - of integers, 18
 - in multiplicative semigroup, 66
 - in semimodule, 69
- program transformation
 - accumulation-variable elimination, 39
 - accumulation-variable introduction, 35
 - common-subexpression elimination, 35
 - enabled by regular types, 35
 - forward to backward iterators, 112
 - relaxing precondition, 38
 - strengthening precondition, 38
 - strict tail-recursive, 37
 - tail-recursive form, 35
- project
 - abstracting platform-specific copy algorithms, 164
 - algorithms for bidirectional bifurcate algorithms, 123
 - axioms for random-access iterator, 113
 - benchmark and composite algorithm for rotate, 189
 - concepts for bounded binary integers, 87
 - coordinate structure concept, 131
 - cross-type operations, 14
 - cycle-detection algorithms, 29
 - dynamic-sequences benchmark, 222
 - dynamic-sequences implementation, 222
 - dynamic-sequences interfaces, 222
 - floating-point nonassociativity, 42
 - isomorphism, equivalence, and ordering using `tree_rotate`, 148
 - iterator adapter for bidirectional bifurcate coordinates, 124
 - linear recurrence sequences, 47
 - minimum-comparison stable sorting and merging, 61
 - nonhalvable Archimedean monoids, 75
 - order-selection stability, 61
 - reallocation strategy for single-extent arrays, 221
 - searching for a subsequence within a sequence, 114
 - setting for Stein gcd, 81
 - sorting library, 208
 - underlying type used in major library, 225
- projection regularity, 216
- proper underlying type, 223
- properly partial object state, 5
- properly partial value type, 2
- property
 - aliased, 150
 - annihilation, 68
 - associative, 31
 - asymmetric, 50
 - `backward_offset`, 161
 - `bounded_range`, 93
 - commutative, 66
 - `complement_of_converse`, 104
 - `counted_range`, 93
 - discreteness, 85
 - disjoint, 134
 - distributive, 68
 - equivalence, 51
 - `forward_offset`, 162
 - identity element, 65
 - `identity_element`, 65
 - `increasing_counted_range`, 105
 - `increasing_range`, 105
 - `inverse_operation`, 66
 - mergeable, 203
 - `mutable_bounded_range`, 151
 - `mutable_counted_range`, 151
 - `mutable_weak_range`, 151
 - notation, 14
 - `not_overlapped`, 157
 - `not_overlapped_backward`, 155
 - `not_overlapped_forward`, 153
 - `not_write_overlapped`, 159
 - partially associative, 98
 - partitioned, 105
 - prime, 14
 - `readable_bounded_range`, 95
 - `readable_counted_range`, 96
 - `readable_tree`, 123
 - `readable_weak_range`, 96
 - reflexive, 50
 - `regular_unary_function`, 14
 - `relation_preserving`, 103

property (*cont.*)

- strict, 50
- strictly_increasing_counted_range, 105
- strictly_increasing_range, 104
- symmetric, 50
- total_ordering, 51
- transitive, 49
- tree, 117
- trichotomy, 51
- weak trichotomy, 51
- weak_ordering, 52
- weak_range, 92
- writable_bounded_range, 150
- writable_counted_range, 150
- writable_weak_range, 150
- write_aliased, 159

proposition, independence of, 86
 pseudopredicate, 136
 pseudorelation, 137
 pseudotransformation, 91

Q

quotient (/), of integers, 18
 quotient
 in Euclidean semimodule, 80
 in Euclidean semiring, 79
 QuotientType type function, 72
 quotient_remainder algorithm, 85
 quotient_remainder_nonnegative algorithm, 82
 quotient_remainder_nonnegative_iterative algorithm, 83

R

random-access iterator, equivalent to indexed iterator, 113
RandomAccessIterator concept, 113
 range
 backward movement, 112
 closed bounded ($[f, l]$), 94
 closed weak or counted ($[f, n]$), 94
 empty, 95
 half-open bounded ($[f, l)$), 94
 half-open weak or counted ($[f, n)$), 94
 increasing, 103
 limit, 95
 lower bound, 107

mutable, 151
 partition point, 105
 partitioned, 105
 readable, 95
 size, 94
 strictly increasing, 103
 upper bound, 107
 writable, 150
 reachability
 of bifurcate coordinate, 116
 in orbit, 18
 reachable algorithm, 121
 readable range, 95
 readable_bounded_range property, 95
 readable_counted_range property, 96
 readable_tree property, 123
 readable_weak_range property, 96
 rearrangement, 172
 bin-based, 172
 copying, 172
 link, 134
 mutative, 172
 ordering-based, 172
 position-based, 172
 reverse, 174
 rotation, 179
 recursion invariant, 36
 reduce algorithm, 99
 reduce_balanced algorithm, 200
 reduce_nonempty algorithm, 99
 reduce_nonzeroes algorithm, 100
 reduction, 98
 reference counting, 230
 refinement of concept, 11
 reflexive property, 50
Regular concept, 11
 and program transformation, 35
 regular function on value type, 3
 regular type, 6–8
 regularity, 216, 217
 regular_unary_function property, 14
Relation concept, 49
 relational concept, 69
 relationship, 229
 relation_preserving property, 103
 relation_source algorithm, 141
 relaxing precondition, 38
 remainder

algorithm, 84
 in Euclidean semimodule, 80
 in Euclidean semiring, 79
 remainder (mod), of integers, 18
 remainder_nonnegative algorithm, 74
 remainder_nonnegative_iterative algorithm,
 75
 remote part of composite object, 217
 representation, 2
 representational equality, 3, 228
requires clause, 13
 syntax, 240
 resources, 4
 result space, 10
 returning useful information, 87, 96, 97,
 101–103, 106, 112, 152, 153, 159,
 163, 174, 179, 182, 211
 reverse rearrangement, 174
 reverse_append algorithm, 139, 140
 reverse_bidirectional algorithm, 175
 reverse_copy algorithm, 156
 reverse_copy_backward algorithm, 156
 reverse_copy_backward_step machine, 156
 reverse_copy_step machine, 155
 reverse_indexed algorithm, 186
 reverse_n_adaptive algorithm, 178
 reverse_n_bidirectional algorithm, 175
 reverse_n_forward algorithm, 177
 reverse_n_indexed algorithm, 175
 reverse_n_with_buffer algorithm, 176
 reverse_swap_ranges algorithm, 167
 reverse_swap_ranges_bounded
 algorithm, 167
 reverse_swap_ranges_n algorithm, 168
 reverse_swap_step machine, 166
 reverse_with_temporary_buffer algorithm,
 187, 225
 reversing links, 145
 Rhind Mathematical Papyrus
 division, 73
 power, 33
Ring concept, 69
 rotate algorithm, 187
 rotate_bidirectional_nontrivial
 algorithm, 182
 rotate_cycles algorithm, 181
 rotate_forward_annotated algorithm,
 183

rotate_forward_nontrivial algorithm, 184
 rotate_forward_step algorithm, 184
 rotate_indexed_nontrivial algorithm, 181
 rotate_nontrivial algorithm, 188
 rotate_partial_nontrivial algorithm, 185
 rotate_random_access_nontrivial algorithm,
 181
 rotate_with_buffer_backward_nontrivial
 algorithm, 186
 rotate_with_buffer_nontrivial algorithm, 185
 rotation
 permutation, 178
 rearrangement, 179

S

schema, concept, 124
 Schreier, Jozef, 55
 Schwarz, Jerry, 150
 segmented array, 221
 segmented index, 221
 select_0_2 algorithm, 53, 63
 select_0_3 algorithm, 54
 select_1_2 algorithm, 54
 select_1_3 algorithm, 55
 select_1_3_ab algorithm, 55
 select_1_4 algorithm, 56, 59
 select_1_4_ab algorithm, 56, 59
 select_1_4_ab_cd algorithm, 56, 58
 select_2_3 algorithm, 54
 select_2_5 algorithm, 60
 select_2_5_ab algorithm, 60
 select_2_5_ab_cd algorithm, 59
 semi (usage convention), 232
 semigroup, 66
Semimodule concept, 69
Semiring concept, 68
 semistable partition rearrangement, 192
 sentinel, 101
Sequence concept, 216
 extent-based models, 219
 linked models, 219
 set, 231
 single-ended array, 220
 single-extent array, 220
 single-extent index, 221
 single-pass traversal, 91
 singly linked list, 218
 sink, 149

- size
 - for `array_k`, 212
 - for `bounded_range`, 214
 - for *Linearizable*, 213
 - size of an orbit, 20
 - size of a range, 94
 - `SizeType` type function, 213
 - slanted index, 221
 - `slow_quotient` algorithm, 73
 - `slow_remainder` algorithm, 72
 - snapshot, 1
 - some algorithm, 97
 - `sort_linked_nonempty_n` algorithm, 142
 - `sort_n` algorithm, 207
 - `sort_n_adaptive` algorithm, 207
 - `sort_n_with_buffer` algorithm, 203
 - source, 90
 - space complexity, memory adaptive, 177
 - species
 - abstract, 2
 - concrete, 2
 - splicing link rearrangement, 219
 - `split_copy` algorithm, 158
 - `split_linked` algorithm, 137
 - stability, 52
 - of merge, 203
 - of partition, 192
 - of sort, 204
 - of sort on linked range, 142
 - stability index, 53
 - Standard Template Library, x
 - starting address, 4, 216
 - state of object, 4
 - Stein, Josef, 81
 - Stein gcd, 81
 - STL, x
 - store, 4
 - strengthened relation, 53
 - strengthening precondition, 38
 - strict property, 50
 - strict tail-recursive, 37
 - strictly increasing range, 103
 - `strictly_increasing_counted_range` property, 105
 - `strictly_increasing_range` property, 104
 - structural equality, 228
 - subpart of composite object, 216
 - subset (\subset), 231
 - subtraction, in additive group, 67
 - `subtractive_gcd` algorithm, 78
 - `subtractive_gcd_nonzero` algorithm, 77
 - successor
 - definition space on range, 94
 - of integer, 41
 - of iterator, 91
 - sum (+)
 - in additive semigroup, 66
 - of integers, 18
 - of iterator and integer, 92
 - `swap` algorithm, 224
 - `swap_basic` algorithm, 223
 - `swap_ranges` algorithm, 165
 - `swap_ranges_bounded` algorithm, 166
 - `swap_ranges_n` algorithm, 166
 - `swap_step` machine, 165
 - symmetric complement of a relation, 52
 - symmetric property, 50
- ## T
- tail-recursive form, 35
 - technique. *See* program transformation
 - auxiliary computation during recursion, 176
 - memory-adaptive algorithm, 177
 - operation–accumulation procedure duality, 47
 - reduction to constrained subproblem, 54
 - returning useful information, 87, 96, 97, 101–103, 106, 112, 152, 153, 159, 163, 174, 179, 182, 211
 - transformation–action duality, 28
 - useful variations of an interface, 38
 - `temporary_buffer` type, 187
 - terminal element under transformation, 18
 - terminating algorithm, 23
 - three-valued compare, 63
 - Tighe, Joseph, 179
 - to-permutation, 172
 - total object state, 5
 - total procedure, 17
 - total value type, 2
 - TotallyOrdered* concept, 62
 - `total_ordering` property, 51
 - trait class, 240
 - transformation, 17
 - composing, 17, 32

- cyclic element, 18
- fixed point of, 170
- identity, 170
- into, 169
- of program. *See* program transformation
- one-to-one, 169
- onto, 169
- orbit, 18
- power of (f^n), 17
- terminal element, 18
- Transformation* concept, 17
- transitive property, 49
- transpose_operation algorithm, 201
- transposition, 171
- traversal
 - multipass, 106
 - single-pass, 91
 - of tree, recursive, 119
- traverse algorithm, 123
- traverse_nonempty algorithm, 118
- traverse_phased_rotating algorithm, 148
- traverse_rotating algorithm, 146
- traverse_step machine, 121
- tree property, 117
- tree_rotate machine, 145
- trichotomy law, 51
- triple type, 11
- trivial cycle, 171
- twice, 41
- two-pointer header doubly linked list, 218
- type
 - array_k, 210
 - bounded_range, 214
 - computational basis, 6
 - counter_machine, 200
 - isomorphism, 86
 - models concept, 11
 - pair, 11, 209
 - regular, 6
 - temporary_buffer, 187
 - triple, 11
 - underlying_iterator, 225
 - visit, 118
- type attribute, 10
 - Arity, 11
- type concept, 11
- type constructor, 11
- type function, 11

- Codomain, 11
- DifferenceType, 113
- DistanceType, 17, 91
- Domain, 12
- implemented via trait class, 240
- InputType, 11
- IteratorConcept, 187
- IteratorType, 133, 134, 213
- QuotientType, 72
- SizeType, 213
- UnderlyingType, 223
- ValueType, 90, 149, 213
- WeightType, 115

U

- unambiguous value type, 3
- UnaryFunction* concept, 12
- UnaryPredicate* concept, 16
- underlying type, 164, 223
 - iterator adapters, 224
 - proper, 223
- UnderlyingType type function, 223
- underlying_iterator type, 225
- underlying_ref algorithm, 224
- union (\cup), 231
- uniquely represented object type, 5
- uniquely represented value type, 2
- univalent concept, 86
- upper bound, 107
- upper_bound_n algorithm, 109
- upper_bound_predicate algorithm, 109
- useful variations of an interface, 38
- usefulness of concept, 87

V

- value, 2
- value type, 2
 - ambiguous, 3
 - properly partial, 2
 - regular function on, 3
 - total, 2
 - uniquely represented, 2
- ValueType type function, 90, 149, 213
- visit type, 118

W

- weak (usage convention), 232
- weak-trichotomy law, 51

- weakening of concept, 11
- weak_ordering property, 52
- weak_range property, 92
- weight algorithm, 122
- WeightType type function, 115
- weight_recursive algorithm, 117
- weight_rotating algorithm, 147
- well-formed object, 5
- well-formed value, 2

- words in memory, 4
- writable range, 150
- writable_bounded_range property, 150
- writable_counted_range property, 150
- writable_weak_range property, 150
- write_aliased property, 159

X

- zero, 41