Elements of Programming

Alexander Stepanov   Sean Parent   Paul McJones

July 13, 2007

Contents I

Contents II

Intended audience

- This book is for programmers who want to enhance their understanding of the craft
- While this book does not have any formal prerequisites, programming maturity and an appreciation of mathematics are assumed
- While this book uses `C++`, its fundamental concepts are not language-specific

Approach

- Programs are mathematical objects
- Mathematics provides tools for
  - reasoning
  - deriving
  - composing
  - optimizing
- Mathematics applied to programming addresses
  - security
  - reliability
  - performance
  - productivity
- The success of this approach requires years of individual study and organized effort by the community

Blending algorithms, formal methods, and programming

1. Start with an existing algorithm
2. Find the minimal requirements on which it depends
3. Implement it generically and efficiently
4. Place it in the taxonomy of requirements and algorithms, adjusting the taxonomy as necessary

The choice of C++

- C defines an abstract machine that in turn defines modern processor architectures
- C++ extends C with mechanisms for writing abstract versions of algorithms and data structures without sacrificing efficiency
- A small subset of C++ suffices
- We are interested in translating programs in the book to other languages and we welcome volunteers

C++ usage conventions

- We avoid implicit conversions to make our code strongly typed with respect to concepts
- However we take advantage of two implicit conversions to make the code clearer:
    - conversion of integer constants (e.g., 0) to types that model Integer
    - conversion of values of type `T` to values of `const T`
- We do not use postincrement ++ or depend on the return value of ++
- We omit `inline` for clarity, but in production code a programmer needs to carefully insert it in all the appropriate places

Outline of the book

- Part I: Algorithms on mathematical abstractions
    - Chapter 1 - Regular types and regular functions
    - Chapter 2 - Algorithms on algebraic structures
    - Chapter 3 - Orderings
    - Chapter 4 - Combining theories
- Part II: Algorithms on abstractions of memory
    - Chapter 5 - Refining theories of iterators
    - Chapter 6 - Permutations and rearrangements
    - Chapter 7 - Rotations
    - Chapter 8 - Partition and sorting
    - Chapter 9 - Node-based algorithms
    - Chapter 10 - Data structures
- Part III: Metamathematics of programming
    - Chapter 11 - Taxonomy of software components
    - Chapter 12 - Algebra of concepts
    - Chapter 13 - Linquistic requirements

Contents I

Contents II

Contents III

Types

Definition

A *type* is a way of interpreting data

Examples

From C++:

- built-in types
- classes
- unions

Concepts

Definition

A *concept* on a type `T` is:

- a set of type functions on `T` that return *affiliated types*
- a set of functions on `T` and its affiliated types
- a set of axioms expressed in terms of these functions

Concepts and models

Definition

- A type is called a *model* of a concept that it satisfies
    - We also say a type *models* a concept
- A given type could model more than one concept

Examples of concepts from C++

Examples

From C++:

- Integral type
  - Unsigned integral type
    - Models: `unsigned`, `unsigned long`, `unsigned char`
  - Signed integral type
    - Models: `int`, `long`, `char`
- Pointer type
  - Models: `int*`, `char*`
- `vector<T>`
  - Models: `vector<int>`
- Bidirectional iterator
  - Models: `list<int>::iterator`

Foundational concept

- First-classness
    - Can be a function argument/result
    - Can be stored in a location
- Equational reasoning
    - Basis of mathematics

Arithmetic:	$\frac{2}{4} = \frac{1}{2}$
Geometry:	$\triangle ABC = \triangle ACB$
Algebra:	$(a + b)^2 = a^2 + 2ab + b^2$

- First-classness + equational reasoning = regular types

Regular type

- Values of a type modeling `Regular`
    - Can be stored in any standard-conforming container
        - `vector`, `map`, `...`, `my_container`
    - Can be used with any standard-conforming algorithm
        - `swap`, `sort`, `...`, `my_algorithm`
    - Allow our programs to be reasoned about, transformed, optimized

Definition of regular type

Definition

- A *regular* type is one with

copy constructor:	`T a = b;`
assignment:	`a = b;`
equality:	`a == b`

- It shares common semantics of many built-in types
  - `int`, `double`, `char*`
  - But `C/C++` arrays are not regular

Axioms for regular

Axioms

```
T a = b; assert(a == b);
a = b;   assert(a == b);
```

Properties of equality

- Equality is symmetric, reflexive, and transitive
  - But not every equivalence relation is equality
- In Chapter 3 we will study orderings
- We will later learn to define equality for many types

Function

Definition

A *function* is something that can be applied to a list of arguments, producing a result: $f(x_0, x_1, \ldots, x_n) \rightarrow \mathrm{result}$

Examples

C++ provides several kinds:

- functions
- pointers to functions
- function objects

Partial function

- Many functions that are *total* (everywhere defined) in mathematics are *partial* in computer science
  - addition on integral types and pointers
  - multiplication on doubles
- There is a (potentially implicit) function, `is_defined`, that tells us whether a particular function is defined for a particular input

Regular function

Definition

A function f is *regular* if and only if
$x_0 = y_0, \ldots, x_{n-1} = y_{n-1} \Rightarrow f(x_0, \ldots, x_{n-1}) = f(y_0, \ldots, y_{n-1})$

Examples

- `sin`, `abs`, `sqrt` are regular
- `getc`, `clock` and `rand` are not regular

Axiom of equality

Axiom

$a = b \Leftrightarrow$ for any regular function f, $f(a) = f(b)$

- This is *not* a constructive definition of equality

Generic definition of equality

Definition

- For all built-in types, equality is defined
- For a composite type, two elements are *equal* if all corresponding essential parts are equal

Example

Both components of a `pair` are its essential parts

Program optimization

- Regular types and regular functions allow us to reason about and transform our programs
    - Common subexpression elimination
    - Loop hoisting
    - Copy propagation

Unary function

Definition

A *unary function* $f : T \to U$ maps a value x in T to another value $f(x)$ in U

Transformation

Definition

- A *transformation* is a unary function $f : T \to T$ mapping a value x in T to another value $f(x)$ in T
- In this book all the transformations are assumed to be regular unless explicitly stated otherwise

Composition

- Transformations are self-composable
    - $f(x), f(f(x)), \ldots$
- This simple ability allows us to define interesting algorithms

Definition

- $f^0(x) = x$
- $f^{n+1}(x) = f^n(f(x))$

Finiteness

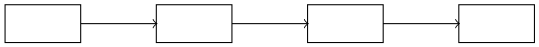
- In this book, there is a general assumption of *finiteness*
- All types are assumed to have a finite number of distinct values
- Memory is assumed to have a finite number of locations

Orbits, reachability, and cycles

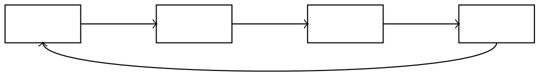
Definitions

- An element y is *reachable* from x if for some i, $f^i(x) = y$
- An *orbit* of an element x under a transformation f is the set of all elements reachable from x
- An element x is *cyclic* under f if $f(x)$ is defined and x is reachable from $f(x)$
- An orbit of x is *terminating* if none of its elements are cyclic
- An orbit of x is *non-terminating* if any of its elements are cyclic
- An orbit of x is *circular* if x is cyclic
- An orbit of x is ρ-*shaped* if it is non-terminating and not circular

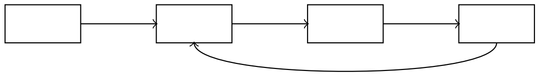
Orbits



terminating



circular



ρ-shaped

Sizes

Definitions

- The *size* of an orbit is the number of distinct elements in it
- The *handle size* of an orbit is the number of non-cyclic elements in it
- The *cycle size* of an orbit is the number of cyclic elements in it
- We refer to them, respectively, as n, h, and c

Fact

$$n = h + c$$

From definition to algorithm

```
bool is_reachable(T from, T to, F f);

bool is_terminating(T x, F f);
bool is_circular(T x, F f);
bool is_rho_shaped(T x, F f);
```

- We can combine the last three into:

```
enum orbit_kind_t {terminating, circular, rho_shaped};

orbit_kind_t orbit_kind(T x, F f);
```

is_reachable_terminating

```
template
    <typename T, // T models Regular
     typename F> // F models Transformation on T
bool is_reachable_terminating(T from, T to, F f)
{
    // Precondition: orbit of from is terminating
    while (true) {
        if (from == to) return true;
        if (!is_defined(f, from)) return false;
        from = f(from);
    }
}
```

Algorithms versus procedures

Definition

An *algorithm* is a procedure that terminates for all inputs in its domain

- `is_reachable_terminating` does not always terminate for inputs not satisfying the precondition; consider the example:
    - `f` is identity function on `T`
    - `from != to`
- Precondition is that orbit of `from` is terminating
- A correct precondition converts a procedure into an algorithm

is_reachable_terminating_or_circular

```
template
    <typename T, // T models Regular
     typename F> // F models Transformation on T
bool is_reachable_terminating_or_circular(T from, T to, F f)
{
    // Precondition: orbit of from is terminating or circular
    T original = from;
    while (true) {
        if (from == to) return true;
        if (!is_defined(f, from)) return false;
        from = f(from);
        if (from == original) return false;
    }
}
```

Algorithmic intuition

- If two cars, `fast` and `slow`, start along a path, `fast` will catch up with `slow` if and only if there is a cycle
    - If there is no cycle, `fast` will reach the end of the path before `slow`
    - If there is a cycle, by the time `slow` enters the cycle, `fast` will already be there, and will catch up eventually

is_reachable

```
template
    <typename T, // T models Regular
     typename F> // F models Transformation on T
bool is_reachable(T from, T to, F f)
{
    T fast = from;
    T slow = from;
    while (true) {
        if (fast == to)           return true;
        if (!is_defined(f, fast)) return false;
        fast = f(fast);
        if (fast == to)           return true;
        if (!is_defined(f, fast)) return false;
        fast = f(fast);
        slow = f(slow);
        if (fast == slow)         return false;
    }
}
```

Observation

- We need to prove termination and analyze the complexity of this algorithm
- Before we do that, we present some variations, one of which will be slightly simpler to analyze
- In general we need to find algorithms corresponding to all the properties of orbits we introduced

is_terminating_0

```
template
    <typename T, // T models Regular
     typename F> // F models Transformation on T
bool is_terminating_0(T x, F f)
{
    T fast = x;
    T slow = x;
    while (true) {
        if (!is_defined(f, fast)) return true;
        fast = f(fast);
        if (!is_defined(f, fast)) return true;
        fast = f(fast);
        slow = f(slow); // unnecessary when !is_defined(f, fast)
        if (fast == slow)         return false;
    }
}
```

is_terminating

```
template
    <typename T, // T models Regular
     typename F> // F models Transformation on T
bool is_terminating(T x, F f)
{
    if (!is_defined(f, x))        return true;
    T fast = x;
    T slow = x;
    while (true) {
        fast = f(fast);
        if (!is_defined(f, fast)) return true;
        fast = f(fast);
        if (!is_defined(f, fast)) return true;
        slow = f(slow);
        if (fast == slow)         return false;
    }
}
```

Correctness of `is_terminating`

- The movement of `fast` is guarded by a call of `is_defined`
- The movement of `slow` is unguarded, because by the regularity of `f`, `slow` will be traversing the same orbit as `fast`
- If there is no cycle, `is_defined` will eventually return false because of finiteness
- If there is a cycle, then once `slow` enters the cycle, the distance from `fast` to `slow` decreases by 1 on each iteration

Complexity of `is_terminating`

- Terminating case
  - f is applied to `fast` $n - 1$ times
  - f is applied to `slow` $(n - 1)/2 - 1$ times
  - f is applied $3(n - 1)/2 - 1$ times total
- Circular case
  - f is applied to `fast` $2n$ times
  - f is applied to `slow` n times
  - f is applied $3n$ times total

Complexity of `is_terminating` – ρ-shaped case

- The first cyclic point reached is the *connection point*
- The point where `fast` catches `slow` is the *collision point*
- $h = mc + r$, where $0 \leqslant r < c$
- When `slow` reaches the connection point, `fast` is r steps ahead
- If $r = 0$, we are done
- If $r \neq 0$, it will take $c - r$ iterations for `fast` to catch `slow`

Adding them all up – ρ-shaped case

- $r = 0$
  - f is applied to `slow` h times
  - f is applied to `fast` $2h$ times
  - f is applied $3h$ times total
- $r \neq 0$
  - f is applied to `slow` $h + (c - r)$ times
  - f is applied to `fast` $2(h + (c - r))$ times
  - f is applied $3(h + (c - r))$ times total
- These can be unified as $3(h + (-h \bmod c))$

Facts we will use later

- The distance to the collision point for the ρ-shaped case is $h + (-h \bmod c)$
- The distance from the collision point to the connection point is $h \bmod c = r$

orbit_kind

```
template
    <typename T, // T models Regular
     typename F> // F models Transformation on T
orbit_kind_t orbit_kind(T x, F f)
{
    if (!is_defined(f, x))        return terminating;
    T fast = x;
    T slow = x;
    while (true) {
        fast = f(fast);
        if (!is_defined(f, fast)) return terminating;
        if (fast == x)            return circular;
        fast = f(fast);
        if (!is_defined(f, fast)) return terminating;
        if (fast == x)            return circular;
        slow = f(slow);
        if (fast == slow)         return rho_shaped;
    }
}
```

orbit_point

```
template
    <typename T, // T models Regular
     typename F> // F models Transformation on T
pair<orbit_kind_t, T> orbit_point(T x, F f)
{
    typedef pair<orbit_kind_t, T> Pair;
    if (!is_defined(f, x))        return Pair(terminating, x);
    T fast = x;
    T slow = x;
    while (true) {
        fast = f(fast);
        if (!is_defined(f, fast)) return Pair(terminating, fast);
        fast = f(fast);
        if (!is_defined(f, fast)) return Pair(terminating, fast);
        slow = f(slow);
        if (fast == slow)         break;
    }
    if (fast == x)                return Pair(circular, x);
    while (true) {
        x    = f(x);
        fast = f(fast);
        if (fast == x)            return Pair(rho_shaped, x);
    }
}
```

Correctness of `orbit_point`

- The terminating and circular cases are the same as `is_terminating` and `orbit_kind`
- The final loop terminates because
    - the connection point is r steps away from the collision point
    - h steps from `x` takes us to the connection point
    - h $(= mc + r)$ steps from the collision point takes us to the connection point and then m times around the cycle

Complexity of `orbit_point`

- Terminating case
  - $\mathtt{f}$ is applied $3(n-1)/2-1$ times
- Circular case
  - $\mathtt{f}$ is applied $3n$ times
- ρ-shaped case:
  - $\mathtt{f}$ is applied $3(h+(-h \bmod c))+2h = 5h+3(-h \bmod c)$ times

Applications

- Determining whether a linked structure contains a cycle
    - See for example the Common Lisp function `list-length`
- Determining the length and the period of a random number generator – see for example:

  Donald Knuth.
  Exercise 3.1.6.
  The Art of Computer Programming, Volume 2, 3rd edition, 1998, page 7.
  Credits two-pointer orbit detection to R.W. Floyd.

- Our code works for both cases

Problem: representing sizes

- What type to use for n, h, and c?
- We need a way to find a type big enough to encode the orbit size for a given type `T` – a *type function*

Definition of distance type

Definition

- The *distance type* for a type `T` is an integral type that allows us to measure the number of function applications that transform one element of `T` into another element
- If a type occupies k bits, its distance type can be represented with an unsigned integral type occupying k bits
- This avoids an infinite tower of types
- (It is difficult to have a *count type* that could count the number of elements in any collection of type `T`, because that would require an extra value)

distance_type

```
template <typename T> // T models Countable
struct distance_type
{
    typedef size_t type;
};
#define DISTANCE_TYPE(T) typename distance_type<T>::type

template <>
struct distance_type<short>
{
    typedef unsigned short type;
};
```

Reflections on type functions

- Someday one will be able to define a type function in the same way as a normal function
- This is the accepted way to do it in `C++`
- It is limited to use only inside templates
  - Because of the use of `typename`

distance

```
template
    <typename T, // T models Regular
     typename F> // F models Transformation on T
DISTANCE_TYPE(T) distance(T first, T last, F f)
{
    assert(is_reachable(first, last, f));
    DISTANCE_TYPE(T) n(0);
    while (first != last) {
        first = f(first);
        n = n + 1;
    }
    return n;
}
```

Representing sizes

- Each of these fits into `distance_type`: $\begin{cases} n - 1 \\ h \\ c - 1 \end{cases}$

table_function

```
struct table_function
{
    typedef int argument_type;
    typedef int result_type;

    table_function(const int* p_a, size_t n_a)
        : p(p_a), n(n_a) {}
    result_type operator()(argument_type& x) const {
        return p[x];
    }
    friend bool is_defined(const table_function& f, int x) {
        return 0 <= x && size_t(x) < f.n;
    }

    const int* p;
    size_t n;
};
```

output_orbit_info

```
template
    <typename T, // T models Regular
     typename F> // F models Transformation on T
void output_orbit_info(T x, F f)
{
    ostream& o = cout;
    pair<orbit_kind_t, T> p = orbit_point(x, f);
    T y = p.second;
    switch (p.first) {
    case terminating:
        o << "terminating with n-1 " << distance(x, y, f)
          << " and termination point " << y; break;
    case circular:
        o << "cyclic with c-1 " << distance(f(x), x, f); break;
    case rho_shaped:
        o << "rho-shaped with h " << distance(x, y, f)
          << " and c-1 " << distance(f(y), y, f)
          << " and connection point " << y;
    }
    o << endl;
}
```

run_table_function

```
int run_table_function()
{
    cout <<
      "Enter sequence of integers terminated by\n"
      "EOF character (Ctrl-D in Linux and Mac, "
      "Ctrl-Z in Windows);\n"
      "enter empty sequence to end" << endl;
    while (true) {
        istream_iterator<int> f(cin), l;
        vector<int> v(f, l);
        clearerr(stdin); // the real world strikes again
        cin.clear();
        if (v.empty()) return 0;
        output_orbit_info(0, table_function(&v[0], v.size()));
    }
}
```

random_function

```
int random_function(int x)
{
    srand(x);
    return rand();
}
```

// Sadly there is no functional version of rand

Default `is_defined`

```
template
    <typename F, // F models Transformation on T
     typename T> // T models Regular
bool is_defined(const F&, const T&) { return true; }
```

- Defaults save time for programmers
- Defaults have unintended consequences

run_random_function

```
int run_random_function()
{
    cout <<
      "Enter an integer or zero to end program." << endl;
    while (true) {
        int x;
        cin >> x;
        if (x == 0) return 0;
        output_orbit_info(x, random_function);
    }
}
```

main

```
int main()
{
    using namespace book;
    return run_table_function();
}

int main()
{
    using namespace book;
    return run_random_function();
}
```

Exercise

Exercise

Run `run_random_function` enough times to decide whether the cycle length generated by `std::rand` on your platform is satisfactory

Conclusions

- Practical algorithms can be described using basic mathematical theories
- Nomenclature helps
    - e.g., orbit kinds and sizes
- Equational reasoning on types and functions is essential
- Abstract code facilitates reasoning and complexity analysis

Reading

James C. Dehnert and Alexander A. Stepanov.
Fundamentals of Generic Programming.
Report of the Dagstuhl Seminar on Generic Programming, Schloss Dagstuhl, Germany, April 1998, also appears in Lecture Notes in Computer Science (LNCS) volume 1766, pages 1-11.
First introduced regular types.

Project

- There are other algorithms for orbit analysis:

  R.T. Sedgwick, T.G. Szymanski and A.C. Yao.
  The complexity of finding cycles in periodic functions.
  Proc. 11th SIGACT Meeting, 1979, pages 376-390.

  Richard P. Brent.
  An improved Monte Carlo factorization algorithm.
  BIT, Volume 20, 1980, pages 176-184.

  Leon S. Levy.
  An improved list-searching algorithm.
  Information Processing Letters, Volume 15, Issue 1, August 1982, pages 43-45.

- Write generic versions of these algorithms and compare their efficiencies

Contents I

Contents II

Contents III

Abstract mathematics

- *Abstract mathematics* is the method of presenting theorems in their most general setting
- It evolved as a solution to the foundational crisis in mathematics
- It led to:
    - great increase in soundness
    - dramatic reuse (shortening of text books)
    - cross-fertilization between different branches

Branches of abstract mathematics

- Set theory
  - Collections, functions, and orderings
- Algebra
  - Theory of addition and multiplication
- Topology
  - Theory of continuity
- ...

Algebraic structures

Examples

- One type, one operation
  - Semigroup, monoid, group
- One type, two interrelated operations
  - Ring, field
- Two interrelated types
  - Module (group and ring), vector space (group and field)

Binary operation

Definition

- A *binary operation* is a regular binary function $f : T \times T \rightarrow T$
- As usual, many binary operations are partial and there is an implicit function `is_defined(op, x, y)`

Semigroup

Definition

- A binary operation $\circ$ is *associative* if $(a \circ b) \circ c = a \circ (b \circ c)$
- A *semigroup* is a set with an associative operation

Additive and multiplicative semigroups

Definition

- A semigroup is *commutative* if $a \circ b = b \circ a$
- A commutative semigroup whose operation is `operator+` is called *additive*
- A semigroup whose operation is `operator*` is called *multiplicative*

Reflection on overloading

Reflection

Progress in mathematics often involves extending an operator to a new domain

Example

$+$ on

- natural numbers
- integers
- rationals
- polynomials
- matrices

Reflection on overloading

Example

Properties of +:

- + is commutative and associative
    - (`C++ operator+` on strings notwithstanding)
- * distributes over +

Reflection

In programming, operators should be associated with generally-assumed semantics and complexity properties

Powers

$$a^2 = a \circ a$$
$$a^3 = a \circ a \circ a$$

$$a^1 = a$$
$$a^n = \underbrace{a \circ a \circ \cdots \circ a}_{n}$$

$$a^n \circ a^m = a^{n+m}$$

Properties of powers

Lemma

$a^n \circ a^m = a^m \circ a^n = a^{n+m}$ (powers of the same element commute)

Lemma

$(a^n)^m = a^{nm}$

Exercise

Prove the lemmas

Monoid

Definition

A semigroup element $\mathtt{i}$ is called an *identity* if $\mathtt{a} \circ \mathtt{i} = \mathtt{i} \circ \mathtt{a} = \mathtt{a}$

Exercise

Prove identity is unique

Definition

- A *monoid* is a semigroup with identity
- There is a function $\mathtt{identity\_element} : \mathrm{Op} \to \mathrm{T}$ such that $\mathtt{identity\_element}(\circ) = \mathtt{i}$

Additive and multiplicative monoids

Definitions

- An *additive monoid* T is an additive semigroup with $x + 0 = 0 + x = x$
- A *multiplicative monoid* T is a multiplicative semigroup with $x \times 1 = 1 \times x = x$

Default identity_element

```
template <typename T> // T models Multiplicative Monoid
T identity_element(const multiplies<T>&)
{
return 1;
}

template <typename T> // T models Additive Monoid
T identity_element(const plus<T>&)
{
return 0;
}
```

Group

Definitions

- A *group* is a monoid with a transformation *inverse* satisfying $\mathrm{a} \circ \mathrm{inverse}(\mathrm{a}) = \mathrm{identity\_element}(\circ) = \mathrm{inverse}(\mathrm{a}) \circ \mathrm{a}$
- There is a function $\mathrm{inverse\_operation} : \mathrm{Op} \to \mathrm{F}$ such that $\mathrm{inverse\_operation}(\circ) = \mathrm{inverse}$

Default inverse_operation

```
template <typename T> // T models Additive Group
negate<T> inverse_operation(const plus<T>&)
{
    return negate<T>();
}

template <typename T> // T models Multiplicative Group
struct reciprocal
{
    T operator()(const T& x) const {
        return identity_element(multiplies<T>()) / x;
    }
};
template <typename T> // T models Multiplicative Group
reciprocal<T> inverse_operation(const multiplies<T>&)
{
    return reciprocal<T>();
}
```

Extending powers to non-positive exponents

$$
\begin{aligned}
a^0 &= \mathtt{identity\_element}(\circ) \\
a^n \circ a^0 &= a^{n+0} = a^n \\
a^{-n} &= \mathtt{inverse}(a^n) \\
a^n \circ a^{-n} &= a^{n-n} = a^0
\end{aligned}
$$

Lemma

$(a^{-1})^n = a^{-n}$

Exercise

Prove the lemma

Concept `Integer`

- Models include:
  - all `C++` integral types, signed and unsigned
  - bignums
- Operations are `+ - * / %` obeying all the standard axioms
- Models may be partial
  - The operations obey the axioms only where they are defined
    - `n < n+1` only if `n < MAX_ELEMENT(I)`
  - All properties of mathematical integers hold for the domain of definition

Why partial models work

- Any terminating computation uses only a finite subset of integers
- It is our job as programmers to choose a type that holds all the values actually used

slow_power_positive

```
template
    <typename T, // T models Multiplicative Semigroup
     typename I> // I models Integer
T slow_power_positive(T a, I n)
{
    assert(n > 0);
    if (n == 1)
        return a;
    else
        return a * slow_power_positive(a, n - 1);
}
```

slow_power_positive with operation

```
template
    <typename T,  // T models Regular
     typename I,  // I models Integer
     typename Op> // Op models Semigroup Operation on T
T slow_power_positive(T a, I n, Op op)
{
    assert(n > 0);
    if (n == 1)
        return a;
    else
        return op(a, slow_power_positive(a, n - 1, op));
}
```

slow_power_nonnegative

```
template
    <typename T,  // T models Regular
     typename I,  // I models Integer
     typename Op> // Op models Monoid Operation on T
T slow_power_nonnegative(T a, I n, Op op)
{
    assert(n >= 0);
    if (n == 0)
        return identity_element(op);
    else
        return slow_power_positive(a, n, op);
}
```

Observation

- When computing a^n, we perform $n - 1$ operations
- We never perform the operation with the identity element
  - Except when a is the identity element
- It is not worth adding an extra check for the case of a being the identity element
  - This would slow down the average case
  - The caller can do this if it is important

slow_power

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op> // Op models Group Operation on T
T slow_power(T a, I n, Op op)
{
    if (n < 0)
    {
        a = inverse_operation(op)(a);
        n = -n;
    }
    return slow_power_nonnegative(a, n, op);
}
```

Over-abstraction

- We were tempted to introduce another, weaker abstraction (*groupoid*, or semigroup without associativity) and provide two algorithms (`left_power`, `right_power`) for it
- But we resisted because we do not have a good application for this algorithm
- Later we will find a use, with the `accumulate` algorithm

Reflection

Do not introduce an abstraction without necessity

- You need a problem to justify the solution

Silly way to multiply

```
cout « slow_power(99, 100, plus<int>());
```

Ancient Egyptian discovery

- Ahmes described (circa 1650 BC) a different power algorithm that gives an efficient way to multiply
- It is called the *Russian peasant algorithm* since western travelers observed 19th century Russian peasants using it

Exploiting associativity

$$\begin{aligned} aaaaaaa &= (aa)(aa)(aa)a \\ a^n &= (a^2)^{n/2} a^{n \bmod 2} \end{aligned}$$

$a^n = (a^2)^{n/2} a^{n \bmod 2}$

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op>  // Op models Monoid Operation on T
T power_nonnegative_0(T a, I n, Op op)
{
    assert(n >= 0);
    if (n == 0)
        return identity_element(op);
    else if (n == 1)
        return a;
    else
        return op(power_nonnegative_0(op(a, a), n / 2, op),
                  power_nonnegative_0(a, n % 2, op));
}
```

Reflection on recursion

Reflection

- Recursion is a powerful technique for converting a mathematical definition into an algorithm
- It could require unnecessary storage for intermediate results
    - Logarithmic here
- It introduces unnecessary function call overhead

$$a^n = \begin{cases} (a^2)^{n/2} & n \text{ even} \\ (a^2)^{n/2}a & n \text{ odd} \end{cases}$$

```
template
    <typename T,  // T models Regular
     typename I,  // I models Integer
     typename Op> // Op models Semigroup Operation on T
T power_positive_0(T a, I n, Op op)
{
    assert(n > 0);
    if (n == 1)
        return a;
    else if (n % 2 == 0)
        return power_positive_0(op(a, a), n / 2, op);
    else
        return op(power_positive_0(op(a, a), n / 2, op), a);
}
```

Eliminating common subexpression

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op>  // Op models Semigroup Operation on T
T power_positive_1(T a, I n, Op op)
{
    assert(n > 0);
    if (n == 1)
        return a;
    T result = power_positive_1(op(a, a), n / 2, op);
    if (n % 2 != 0)
        result = op(result, a);
    return result;
}
```

Introducing accumulation variable

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op> // Op models Semigroup Operation on T
T power_accumulate_nonnegative_0(T r, T a, I n, Op op)
{
    assert(n >= 0);
    if (n == 0)
        return r;
    if (n % 2 != 0) r = op(r, a);
    return power_accumulate_nonnegative_0(r, op(a, a), n / 2, op);
}
```

Recursion invariant: $\mathtt{r}\mathtt{a}^{\mathtt{n}}$

Almost iterative

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op> // Op models Semigroup Operation on T
T power_accumulate_nonnegative_1(T r, T a, I n, Op op)
{
    assert(n >= 0);
    if (n == 0)
        return r;
    if (n % 2 != 0) r = op(r, a);
    a = op(a, a);
    n = n / 2;
    return power_accumulate_nonnegative_1(r, a, n, op);
}
```

Recursion invariant: $\mathtt{ra}^{\mathtt{n}}$

Iterative

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op>  // Op models Semigroup Operation on T
T power_accumulate_nonnegative_2(T r, T a, I n, Op op)
{
    assert(n >= 0);
    while (true) {
        if (n == 0)
            return r;
        if (n % 2 != 0) r = op(r, a);
        a = op(a, a); // wasted on last iteration
        n = n /2;
    }
}
```

Loop invariant: ra^n

Rotating the loop: reordering

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op>  // Op models Semigroup Operation on T
T power_accumulate_nonnegative_3(T r, T a, I n, Op op)
{
    assert(n >= 0);
    while (true) {
        if (n == 0)
            return r;
        if (n % 2 != 0) r = op(r, a);
        n = n / 2;      // reorder
        a = op(a, a); // independent statements
    }
}
```

Loop invariant: ra^n

Rotating the loop: duplicating exit

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op> // Op models Semigroup Operation on T
T power_accumulate_nonnegative_4(T r, T a, I n, Op op)
{
    assert(n >= 0);
    while (true) {
        if (n == 0)
            return r;
        if (n % 2 != 0) r = op(r, a);
        n = n / 2;
        if (n == 0)
            return r; // early exit
        a = op(a, a);
    }
}
```

Loop invariant: ra^n

Rotating the loop: hoisting exit

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op> // Op models Semigroup Operation on T
T power_accumulate_nonnegative_5(T r, T a, I n, Op op)
{
    assert(n >= 0);
    if (n == 0)
        return r; // moved first test out of loop
    while (true) {
        if (n % 2 != 0) r = op(r, a);
        n = n / 2;
        if (n == 0)
            return r;
        a = op(a, a);
    }
}
```

Loop invariant: ra^n

Dependency between conditions

- n will reach 0 only from 1
- 1 is odd
- The second condition will only be true when the first condition is true:

```
if (n % 2 != 0) r = op(r, a);
n = n / 2;
if (n == 0)
    return r;
```

Utilizing dependency

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op>  // Op models Semigroup Operation on T
T power_accumulate_nonnegative_6(T r, T a, I n, Op op)
{
    assert(n >= 0);
    if (n == 0)
        return r;
    while (true) {
        bool odd = n % 2 != 0;
        n = n / 2;
        if (odd) {
            r = op(r, a);
            if (n == 0)
                return r;
        }
        a = op(a, a);
    }
}
```

Specializing for positive n

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op>  // Op models Semigroup Operation on T
T power_accumulate_positive_0(T r, T a, I n, Op op)
{
    assert(n > 0);
    while (true) {
        bool odd = n % 2 != 0;
        n = n / 2;
        if (odd) {
            r = op(r, a);
            if (n == 0)
                return r;
        }
        a = op(a, a);
    }
}
```

(Nearly final) power_accumulate_nonnegative

```
template
    <typename T,  // T models Regular
     typename I,  // I models Integer
     typename Op> // Op models Semigroup Operation on T
T power_accumulate_nonnegative_7(T r, T a, I n, Op op)
{
    assert(n >= 0);
    if (n == 0)
        return r;
    else
        return power_accumulate_positive_0(r, a, n, op);
}
```

Eliminating accumulation variable

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op> // Op models Semigroup Operation on T
T power_positive_2(T a, I n, Op op)
{
    assert(n > 0);
    n = n - 1;
    return power_accumulate_nonnegative_7(a, a, n, op);
}
```

Observation

- When `n` is 16, we do 7 operations where only 4 are needed
- When `n` is odd, this code is fine

Factoring out powers of 2

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op> // Op models Semigroup Operation on T
T power_positive_3(T a, I n, Op op)
{
    assert(n > 0);
    while (n % 2 == 0) {
        a = op(a, a);
        n = n / 2;
    }
    n = n - 1;
    if (n == 0)
        return a;
    else
        return power_accumulate_positive_0(a, a, n, op);
}
```

Observation

- `power_accumulate_positive_0` checks if `n` is odd on the first pass
- We know `n` is even at that point
- One extra operation!

Unrolling one iteration of `power_accumulate_positive_0`

```
template
    <typename T,   // T models Regular
     typename I,   // I models Integer
     typename Op> // Op models Semigroup Operation on T
T power_positive_4(T a, I n, Op op)
{
    assert(n > 0);
    while (n % 2 == 0) {
        a = op(a, a);
        n = n / 2;
    }
    n = n / 2;
    if (n == 0)
        return a;
    else
        return power_accumulate_positive_0(a, op(a, a), n, op);
}
```

Operations on exponent

- In the final version we used these operations:
  - `n = n / 2;`
  - `n % 2 == 0`
  - `n == 0`
- General `/` and `%` are very expensive
- For all integral types, if we know `n` is non-negative, we can use shifts and masks

Concept `Halvable Integer`

```
// T models Halvable Integer
bool is_positive(const T&);
bool is_negative(const T&);
bool is_zero(const T&);
bool is_even(const T&);
bool is_odd(const T&);
void halve_non_negative(T&);
```

Default Halvable Integer

```
// T models Integral
template <typename T>
bool is_positive(const T& a) { return 0 < a; }

template <typename T>
bool is_negative(const T& a) { return a < 0; }

template <typename T>
bool is_zero(const T& a) { return a == 0; }

template <typename T>
bool is_even(const T& a) { return (a & 1) == 0; }

template <typename T>
bool is_odd(const T& a) { return !is_even(a); }

template <typename T>
void halve_non_negative(T& a) { a >>= 1; }
```

power_accumulate_positive

```
template
    <typename T,   // T models Regular
     typename I,   // I models Halvable Integer
     typename Op>  // Op models Semigroup Operation on T
T power_accumulate_positive(T r, T a, I n, Op op)
{
    assert(is_positive(n));
    while (true) {
        bool odd = is_odd(n);
        halve_non_negative(n);
        if (odd) {
            r = op(r, a);
            if (is_zero(n))
                return r;
        }
        a = op(a, a);
    }
}
```

power_accumulate_nonnegative

```
template
    <typename T,  // T models Regular
     typename I,  // I models Halvable Integer
     typename Op> // Op models Monoid Operation on T
T power_accumulate_nonnegative(T r, T a, I n, Op op)
{
    assert(!is_negative(n));
    if (is_zero(n))
        return r;
    else
        return power_accumulate_positive(r, a, n, op);
}
```

power_positive

```
template
    <typename T,   // T models Regular
     typename I,   // I models Halvable Integer
     typename Op> // Op models Semigroup Operation on T
T power_positive(T a, I n, Op op)
{
    assert(is_positive(n));
    while (is_even(n)) {
        a = op(a, a);
        halve_non_negative(n);
    }
    halve_non_negative(n);
    if (is_zero(n))
        return a;
    else
        return power_accumulate_positive(a, op(a, a), n, op);
}
```

power_nonnegative

```
template
    <typename T,  // T models Regular
     typename I,  // I models Halvable Integer
     typename Op> // Op models Monoid Operation on T
T power_nonnegative(T a, I n, Op op)
{
    assert(!is_negative(n));
    if (is_zero(n))
        return identity_element(op);
    else
        return power_positive(a, n, op);
}
```

power

```
template
    <typename T,  // T models Regular
     typename I,  // I models Halvable Integer
     typename Op> // Op models Group Operation on T
T power(T a, I n, Op op)
{
    if (is_negative(n)) {
         a = inverse_operation(op)(a);
         n = -n;
    }
    return power_nonnegative(a, n, op);
}
```

Complexity

- Let u = number of significant bits in $n > 0$
- Let v = number of one-bits in n
- Let p = number of operations performed
- Then $p = (u - 1) + (v - 1) = u + v - 2$

Exercise

Prove that `power` achieves this complexity

`power` is not optimal

- For $\mathtt{n} = 15$, $u = 4$ and $v = 4$, so $u + v - 2 = 6$
- But $15 = 3 \times 5$, so $a^{15} = (a^3)^5$
  - a^3 takes 2 operations
  - a^5 takes 3 operations
  - $(a^3)^5$ takes 5 total
- There are faster ways for 23, 27, 39, 43, . . .

Applications

- The main application of fast exponentiation is cryptography

Donald Knuth.
Secret factors.
The Art of Computer Programming, Vol. 2, pages 403-407.
RSA.

Donald Knuth.
Improved primality tests.
The Art of Computer Programming, Vol. 2, pages 394-396.
Primality testing.

Manindra Agrawal, Neeraj Kayal, and Nitin Saxena.
PRIMES is in P.
Annals of Mathematics, 160 (2004), pages 781-793.
Primality testing.

Reflections on deriving useful interfaces

- `power_accumulate_nonnegative` is useful whenever one wants to compute ax^n
- Even `power_accumulate_positive` is useful in a situation when it is known that $n > 0$

Reflection

While developing a component, it is often possible to discover additional useful interfaces

Reflection on code transformations

Reflection

- Compilers can perform code transformations when the semantics of the operations are known
- Currently this is only for built-in types
- Someday we will be able to tell the compiler the semantics of *our* operations

annotated_plus

```
template <typename T> // T models Additive Monoid
struct annotated_plus
{
    ostream* o;
    annotated_plus(ostream& s) : o(&s) { }
    T operator()(const T& x, const T& y)
    {
        T tmp = x + y;
        *o << x << " + " << y << " == " << tmp << endl;
        return tmp;
    }
};

template <typename T> // T models Additive Monoid
T identity_element(const annotated_plus<T>&)
{
    return 0;
}
```

annotated_negate

```
template <typename T> // T models Additive Monoid
struct annotated_negate
{
    ostream* o;
    annotated_negate(ostream& s) : o(&s) { }
    T operator()(const T& x)
    {
        T tmp = -x;
        *o << "0 - " << x << " == " << tmp << endl;
        return tmp;
    }
};

template <typename T> // T models Additive Group
annotated_negate<T> inverse_operation(annotated_plus<T>& plus)
{
    return annotated_negate<T>(*plus.o);
}
```

run_egyptian_multiplication

```
int run_egyptian_multiplication()
{
    cout
      << "Enter two integers or two zeros to end program."
      << endl;
    while (true)
    {
        int x, n;
        cin >> x; cin >> n;
        if (x == 0 && n == 0)
            return 0;
        cout << "\nslow_power\n";
        slow_power(x, n, annotated_plus<int>(cout));
        cout << "\npower\n";
        power(x, n, annotated_plus<int>(cout));
    }
}
```

Fibonacci numbers

Definition

$$f_0 = 0$$
$$f_1 = 1$$
$$f_{n+2} = f_{n+1} + f_n$$

Example

0, 1, 1, 2, 3, 5, 8, 13, . . .

Recursive way to calculate f_n

```
int fibonacci_recursive(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci_recursive(n - 1)
             + fibonacci_recursive(n - 2);
}
```

Iterative way to calculate f_n

```
int fibonacci_iterative(int n)
{
    if (n == 0)
        return 0;
    int fib_i = 0;
    int fib_j = 1;
    while (n != 1) {
        int next = fib_i + fib_j;
        fib_i = fib_j;
        fib_j = next;
        n = n - 1;
    }
    return fib_j;
}
```

Fibonacci matrices

$$F_1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$F_n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

$$F_1 F_n = \begin{pmatrix} f_{n+1} + f_n & f_n + f_{n-1} \\ f_{n+1} & f_n \end{pmatrix} = \begin{pmatrix} f_{n+2} & f_{n+1} \\ f_{n+1} & f_n \end{pmatrix} = F_{n+1}$$

$$F_n = \underbrace{F_1 F_1 \ldots F_1}_{n} = F_1^n$$

$$F_m F_n = F_1^m F_1^n = F_1^{m+n} = F_{m+n}$$

Fibonacci matrices

$$F_m = \begin{pmatrix} f_{m+1} & f_m \\ f_m & f_{m-1} \end{pmatrix}$$
$$F_n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$
$$F_m F_n = \begin{pmatrix} f_{m+1}f_{n+1} + f_m f_n & f_{m+1}f_n + f_m f_{n-1} \\ f_m f_{n+1} + f_{m-1}f_n & f_m f_n + f_{m-1}f_{n-1} \end{pmatrix}$$

Observations

- We can represent the matrix with a `pair` corresponding to the second row
    - `pair::first` is f_n and `pair::second` is f_{n-1}
- The identity $f_{n+1} = f_{n-1} + f_n$ allows us to compute the bottom row of the product

fibonacci_multiplies

```
template <typename I> // I models Integer
struct fibonacci_multiplies
{
    pair<I, I> operator()(const pair<I, I>& x,
                          const pair<I, I>& y) const
    {
        return pair<I, I>(
            x.first * (y.first + y.second) + x.second * y.first,
            x.first * y.first + x.second * y.second);
    }
};
```

fibonacci

```
template <typename I> // I models Integer
I fibonacci(int n)
{
    assert(n >= 0);
    if (n == 0)
        return 0;
    fibonacci_multiplies<I> op;
    return power_positive(pair<I, I>(1, 0), n, op).first;
}
```

run_fibonacci

```
int run_fibonacci()
{
    cout
      << "Enter an integer or a negative integer to end program."
      << endl;
    while (true)
    {
        int n;
        cin >> n;
        if (n < 0)
            return 0;
        cout << "fibonacci("<< n << ") == "
             << fibonacci<long long>(n) << endl;
    }
}
```

Conclusions

- Algorithms are *generic* by nature
  - They can be used with different models satisfying the same requirements
- Algorithms are affiliated with algebraic structures: semigroup, monoid, . . .
- Stepwise refinement leads from mathematical definitions to efficient code
- Intended models lead to specialized structures: halvable integer
- Mathematics leads to surprising algorithms: `fibonacci`

Reading

N. Bourbaki.
Chapter IV: Structures, and Summary of Results, Section 8: Scales of Sets. Structures.
Theory of Sets, Springer-Verlag, 2004.
The canonical exposition of algebraic structures.

D. Kapur, D.R. Musser, and A.A. Stepanov.
Operators and Algebraic Structures.
Proceedings of the 1981 conference on Functional programming languages and computer architecture, pages 59-63.
One of the first presentations of algorithms on algebraic structures.

Project 1

Charles M. Fiduccia.
An efficient formula for linear recurrences.
SIAM Journal of Computing, Volume 14, Number 1, February 1985, page 106-112.
Generalizes the fibonacci algorithm to arbitrary linear recurrences.

Project

Create a library implementing Fiduccia's algorithm

Project 2

Donald E. Knuth.
Addition chains.
The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd edition, Addison-Wesley, San Francisco, 1998, pages 465-481.
Describes a way to do minimal-operation exponentiation using addition chains.

Project

Implement a useful library doing power optimally for exponents known at compile time

Contents I

Contents II

Contents III

The importance of linear ordering

- Equality allows comparing individual elements easily
- But it only allows dealing with sets of values very slowly
- A linear ordering allows organizing elements so that their sets can be
    - matched: intersection, subset, union
    - searched

Relation

Definitions

- A *predicate* is a boolean-valued regular function
- A *relation* is a type-homogeneous binary predicate
    - Other books use this term for more general non-unary predicates

Transitivity

Definition

A relation r is *transitive* if $r(a, b) \wedge r(b, c) \Rightarrow r(a, c)$

Ordering

Definition

An *ordering* is a transitive relation

Examples

- Equality
- Equality of the first character
- Reachability in an orbit
- Divisibility

Strict versus reflexive

Definition

- An ordering r is *strict* if $\neg r(a, a)$
- An ordering r is *reflexive* if $r(a, a)$
- An ordering r is *weakly reflexive* if $r(a, b) \Rightarrow r(a, a) \wedge r(b, b)$

Examples

- Reflexive: all examples on the previous slide
- Strict: proper factor

Symmetric versus asymmetric

Definition

- A relation r is *symmetric* if $r(a, b) \Rightarrow r(b, a)$
- A relation r is *asymmetric* if $r(a, b) \Rightarrow \neg r(b, a)$

Exercise

Prove that a symmetric ordering is weakly reflexive

Examples

- Symmetric: sibling
- Asymmetric: parent

Equivalence

Definition

An *equivalence* is a reflexive, symmetric ordering

Examples

- Equality
- Geometrical congruence
- $a = b \pmod{6}$

Equivalence and equality

Lemma

If r is an equivalence relation, $a = b \Rightarrow r(a, b)$

Exercise

Prove the lemma

Symmetric complement

Definition

If r is a relation, the *symmetric complement* of r is $\neg r(a, b) \land \neg r(b, a)$

Total ordering

Definition

- A strict ordering is called a *total ordering* if its symmetric complement is equality
  - (Trichotomy law)
- In this book, we always assume strictness

Exercise

Prove that a total ordering is asymmetric

Total ordering on pairs

```
template
    <typename T1, // T1 models Totally Ordered
     typename T2> // T2 models Totally Ordered
struct lexicographically_less_than
{
    bool operator()(const pair<T1, T2>& a, const pair<T1, T2>& b) const
    {
        return a.first < b.first
              || !(b.first < a.first) && a.second < b.second;
    }
};
```

Weak ordering

Definition

- A strict ordering is called a *weak ordering* if its symmetric complement is an equivalence relation
    - (Weak trichotomy law)

Exercise

Prove that a total ordering is a weak ordering

Exercise

Prove that a weak ordering is asymmetric

Weak ordering on first component of pairs

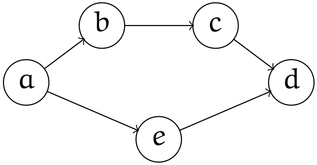
```
template
    <typename T1, // T1 models Totally Ordered
     typename T2> // T2 models Totally Ordered
struct less_first
{
    bool operator()(const pair<T1, T2>& a, const pair<T1, T2>& b) const
    {
        return a.first < b.first;
    }
};
```

Weak ordering on second component of pairs

```
template
    <typename T1, // T1 models Totally Ordered
     typename T2> // T2 models Totally Ordered
struct less_second
{
    bool operator()(const pair<T1, T2>& a, const pair<T1, T2>& b) const
    {
        return a.second < b.second;
    }
};
```

Not every strict ordering is weak

- Suppose a relation $\propto$ on $\{a, b, c, d, e\}$ is $a \propto b \propto c \propto d, a \propto e \propto d$:



$$\begin{tikzcd} & b \arrow[r] & c \arrow[dr] & \\ a \arrow[ur] \arrow[dr] & & & d \\ & e \arrow[urr] & & \end{tikzcd}$$

- The symmetric complement $\diamond$ of $\propto$ is $b \diamond e, e \diamond b, e \diamond c, c \diamond e$
- If $\diamond$ were an equivalence relation it would have to include $b \diamond c$ and $c \diamond b$ by transitivity

Partial ordering

- There are algorithms that deal with orderings that are not weak: algorithms on partially-ordered sets
- The most important is *topological sort*:

  Donald Knuth.
  Topological sorting.
  The Art of Computer Programming, Volume 1, Addison-Wesley, 1997, pages 261-268.

- Operations as simple as `max` and `min` do not make sense without a weak order

Multiple orderings on a type

- There is one equality relation on a type `T`
- There can be many equivalence relations
- Similarly there can be many orderings

Natural total ordering

Definition

The total ordering on a type that is consistent with algebraic operations on it is called the *natural total ordering*

$$\begin{aligned} a &< \text{successor}(a) \\ a < b &\Rightarrow \text{successor}(a) < \text{successor}(b) \\ a < b &\Rightarrow a + c < b + c \\ a < b \wedge 0 < c &\Rightarrow ca < cb \end{aligned}$$

Default ordering

- Sometimes a type does not have a natural total ordering
  - Complex numbers
  - Iterators on linked lists
- We still want a total ordering to put values into sets, so we always define the *default ordering*
  - Lexicographic ordering for complex numbers
  - Address-based ordering for iterators on linked lists
- When the natural order exists, it coincides with the default ordering

Definition

`less<T>` defines the default ordering for `T`

Overloading

Definition

`operator<` is reserved for the natural total ordering

Definition

`operator<` determines the corresponding `operator>`, `operator<=`, and `operator>=`:

$$a > b \Leftrightarrow b < a$$
$$a \leqslant b \Leftrightarrow \neg(b < a)$$
$$a \geqslant b \Leftrightarrow \neg(a < b)$$

Intervals

Definition

- A *closed interval* $[a, b]$ is the set of all elements x such that $a \leqslant x \leqslant b$
- An *open interval* (a, b) is the set of all elements x such that $a < x < b$
- A *half-open on right interval* $[a, b)$ is the set of all elements x such that $a \leqslant x < b$
- A *half-open on left interval* $(a, b]$ is the set of all elements x such that $a < x \leqslant b$
- A *half-open interval* is our short-hand for half-open on right
- These definitions generalize to weak orderings

min for `const T&`

```
template <typename T> // T models Totally Ordered
const T& min(const T& a, const T& b)
{
    if (b < a) return b;
    else       return a;
}

template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
const T& min(const T& a, const T& b, R r)
{
    if (r(b, a)) return b;
    else         return a;
}
```

min for T&

```
template <typename T> // T models Totally Ordered
T& min(T& a, T& b)
{
    if (b < a) return b;
    else       return a;
}

template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
T& min(T& a, T& b, R r)
{
    if (r(b, a)) return b;
    else         return a;
}
```

Why call by const reference

- The type `T` could be arbitrarily large
- Consider the complexity of `min`, where N is the cost of reading a value of type `T`
  - Worst case: 2N
  - Average: constant (first words differ)
- If we used call-by-value, we would have to add an additional 6N to copy the arguments and result
- For small `T`, peephole optimization should eliminate the unneeded stores/loads

Why call by non-const reference

- To modify the smaller object
  ```
  min(a, b) += 3;
  ```
- C++ requires two distinct versions
- From now on, we usually show the non-const version

Passing the ordering

- We pass the ordering operation by value because it is small: a function pointer or a function object with little or no state
- `C++` does not allow us to default the weak ordering operation to either the normal ordering or the default ordering
- From now on we will show the weak ordering version

Stability

Definition

An algorithm is *stable* if it respects the original order of equivalent elements

Example

- Stable sort
    - Multiple passes compose naturally
        - Sorting by first name and then by last name results in expected order
- Stable partition
    - Even/odd partition of $\{1, 2, 3, 4, 5\}$ results in $\{1, 3, 5, 2, 4\}$, not $\{1, 5, 3, 4, 2\}$ as produced by a fast partitioning algorithm
- Stability sometimes increases the complexity

Non-descending order

Definition

- A sequence $\ldots, a, \ldots, b, \ldots$ is in *non-descending* order if $\neg r(b, a)$
- In this book we always assume non-descending order for sorted sequences

sort_2

```
template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
void sort_2(T& a, T& b, R r)
{
    if (r(b, a)) swap(a, b);
}
```

- `sort_2` is stable
- It does the minimal amount of work since it does not swap equivalent elements

Stability of min

- Natural postcondition for `sort_2`:

```
sort_2(a, b, r);
assert(&a == &min(a, b, r)
    && &b == &max(a, b, r));
```

- This would not be the case if we implemented `min` with this body:

```
if (r(a, b)) return a;
else         return b;
```

max

```
template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
T& max(T& a, T& b, R r)
{
    if (r(b, a)) return a;
    else         return b;
}
```

Semantic interdependence of related functions

Reflection

To understand the semantics of a function, we often need to look at the semantics of related functions

min_3 and max_3

```
template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
T& min_3(T& a, T& b, T& c, R r)
{
    return min(min(a, b, r), c, r);
}

template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
T& max_3(T& a, T& b, T& c, R r)
{
    return max(max(a, b, r), c, r);
}
```

Implementing order selection

- `min` and `max` are extreme cases; often other cases, such as select 2nd of 4 or median of 3, are needed
- Writing order selections is somewhat complicated and can be helped by decomposition into simpler subproblems

median_3

```
template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
T& median_3(T& a, T& b, T& c, R r)
{
    if (r(b, a)) return median_3_stage1(b, a, c, r);
    else         return median_3_stage1(a, b, c, r);
}

template <typename T, typename R> // T, R as above
T& median_3_stage1(T& a, T& b, T& c, R r)
{
    assert(!r(b, a));         // a, b are sorted
    if (!r(c, b)) return b; // a, b, c are sorted
    else          return max(a, c, r); // not b
}
```

Exercise

Exercise

Show that `median_3` is stable

Complexity of `median_3`

- `median_3` does 3 comparisons in the worst case
- The function does 2 comparisons only when `c` is `max_3(a, b, c)` and that happens in one-third of the cases
- The average number of comparison is $2\frac{2}{3}$

Selecting second smallest

- Finding second smallest implies finding smallest
  - If after finding second smallest, two candidates for smallest remain, then the second smallest is not second smallest!
- Finding second smallest from n elements requires at least $n + \log n - 2$ comparisons
  - Finding second out of four requires 4 comparisons

select_2nd_4

```
template <typename T, typename R>
T& select_2nd_4(T& a, T& b, T& c, T& d, R r) {
    if (r(b, a)) return select_2nd_4_stage1(b, a, c, d, r);
    else         return select_2nd_4_stage1(a, b, c, d, r);
}

template <typename T, typename R>
T& select_2nd_4_stage1(T& a, T& b, T& c, T& d, R r) {
    // a ⩽ b
    if (r(d, c)) return select_2nd_4_stage2(a, b, d, c, r);
    else         return select_2nd_4_stage2(a, b, c, d, r);
}

template <typename T, typename R>
T& select_2nd_4_stage2(T& a, T& b, T& c, T& d, R r) {
    // a ⩽ b, c ⩽ d
    if (r(c, a)) return min(a, d, r);
    else         return min(b, c, r);
}
```

Exercises

Exercise

Is `select_2nd_4` stable?

Exercise

Implement `select_3rd_4`

median_5

```
template <typename T, typename R>
T& median_5(T& a, T& b, T& c, T& d, T& e, R r) {
    if (r(b, a)) return median_5_stage1(b, a, c, d, e, r);
    else         return median_5_stage1(a, b, c, d, e, r);
}

template <typename T, typename R>
T& median_5_stage1(T& a, T& b, T& c, T& d, T& e, R r) {
    // a ⩽ b
    if (r(d, c)) return median_5_stage2(a, b, d, c, e, r);
    else         return median_5_stage2(a, b, c, d, e, r);
}

template <typename T, typename R>
T& median_5_stage2(T& a, T& b, T& c, T& d, T& e, R r) {
    // a ⩽ b, c ⩽ d
    if (r(c, a)) return select_2nd_4_stage1(a, b, d, e, r);
    else         return select_2nd_4_stage1(c, d, b, e, r);
}
```

run_select_2nd_4

```
int run_select_2nd_4()
{
    typedef pair<int, int> T;
    int p[] = {1, 2, 2, 3};
    T t[] = {T(p[0], 1), T(p[1], 2), T(p[2], 3), T(p[3], 4)};
    while (true) {
        cout << "2nd of (" << p[0] << " " << p[1]
             << " " << p[2] << " " << p[3] << ") is ";
        T r = select_2nd_4(t[0], t[1], t[2], t[3], less_first<int, int>());
        cout << r.first << "/" << r.second << endl;
        if (!next_permutation(p, p+sizeof(p)/sizeof(int), less<int>()))
            return 0;
    }
}
```

run_median_5

```
int run_median_5()
{
    int p[5] = {1, 2, 3, 4, 5};
    while (true) {
        cout << "median of (" << p[0] << " " << p[1]
             << " " << p[2] << " " << p[3]
             << " " << p[4] << ") is "
             << median_5<int, less<int> >(p[0], p[1], p[2], p[3], p[4],
                                             less<int>())
             << endl;
        if (!next_permutation(p, p+sizeof(p)/sizeof(int), less<int>()))
            return 0;
    }
}
```

Exercises

Exercise

Show that `median_5` is not stable

Exercise

Design a stable version of `median_5`

Exercise

Find an algorithm for median of 5 that does slightly fewer comparisons on average

Conclusions

- Weak orderings allow efficient algorithms on sets of elements
    - Allow binary search
- The axioms of ordering provide the interface to connect specific orderings with general purpose algorithms
- Overloaded operators should preserve their semantics in mathematics
- The way a parameter is passed and a result is returned (value, const ref, ref) is an important part of an interface
- Inline procedures and call by reference allow a minimal representation of decision trees

Reading

- Our treatment of ordering is inspired by:

  N. Bourbaki.
  Chapter 3, Section 1: Order relations. Ordered Sets.
  Theory of Sets, Springer, 2004, pages 131-148.

Project

- Using material from

  Donald E. Knuth.
  Section 5.3: Optimum Sorting.
  The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd edition, Addison-Wesley, 1998.

  create a library for generic minimum-comparison networks for sorting, merging, and selection
- For sorting and merging networks, minimize not only the number of comparisons, but the number of data movements
- Assure stability of these networks

Contents I

Contents II

Contents III

Axioms combine theories

- Combining different structures on the same type
- Combining different types

Ring

Definition

A *(commutative)*[1] *ring* is

- An additive group on a set
- A commutative multiplicative monoid on the same set
- And a distributive law (axiom) combining the two:

$$x(y + z) = xy + xz$$

Example

Integers are a ring under addition and multiplication

[1]The rings in this book are all commutative, but a *noncommutative ring* is one with a noncommutative monoid operation

Module

Definition

A *module over a ring* is a commutative group with

- An affiliated type, which must be a ring
- Interrelating laws (α and β are ring elements; x and y are group elements):

$$\alpha(\beta x) = (\alpha\beta)x$$
$$(\alpha + \beta)x = \alpha x + \beta x$$
$$\alpha(x + y) = \alpha x + \alpha y$$
$$1x = x$$

Example

Polynomials with integer coefficients constitute a module over integers

An algorithmically-induced structure

Definition

Any commutative group is a module over integers with the operation nx defined as $\underbrace{x \circ x \circ \ldots \circ x}_{n}$

- This operation can be computed by `power` using no more than $2 \log n$ group operations
- We extend the operation to $n \leqslant 0$ as in Chapter 2

Ordered additive group

Definition

An *ordered additive group* is a set with

- An additive group operation
- A total ordering
- An axiom relating addition and ordering:[2]

$$x < y \Rightarrow x + z < y + z$$

[2]Exactly the same axiom defines *ordered additive monoid* ; many of the results in this chapter naturally generalize to such monoids but we leave this generalization as an exercise

Absolute value

```
template <typename T> // T models Ordered Additive Group
T abs(const T& x)
{
    if (x < 0) return -x;
    else       return  x;
}
```

- The correctness of `abs` depends on

$$x < 0 \Rightarrow x + (-x) < 0 + (-x) \Rightarrow 0 < -x$$

- We use the notation $|x|$ for the absolute value of x

Subtraction on an additive group

Definition

For an additive group, *subtraction* is defined as $a - b = a + (-b)$

Properties of absolute value

$$\begin{aligned} |x - y| &= |y - x| \\ |x + y| &\leqslant |x| + |y| \\ |x - y| &\geqslant |x| - |y| \end{aligned}$$

$$\begin{aligned} |x| = 0 &\Rightarrow x = 0 \\ x \neq 0 &\Rightarrow |x| > 0 \end{aligned}$$

Exercise

Prove these properties

Inducing division with remainder on an ordered additive group

- As repeated addition induces multiplication, repeated subtraction should induce division
- But an extra condition is needed to ensure termination

Archimedean group

Definition

An *Archimedean group* is an ordered additive group such that[3] for every a and $b > 0$ in the group, there exists an integer n such that $0 \leqslant a - nb < b$. This induces *division* of a by b

- n is called the quotient
- $a - nb$ is called the remainder
- For negative b, quotient and remainder are defined by $b < a - nb \leqslant 0$

Exercise

Prove that integers, rationals, and reals are Archimedean groups

[3]This Axiom of Archimedes is usually written as "there exists an integer m such that $a < mb$" but this obscures the relationship to division and does not hold for partial models

slow_remainder

```
template <typename T> // T models Archimedean Group
T slow_remainder(T a, T b)
{
    assert(b != 0);
    if (a < 0)
        if (b < 0)
            while (a <= b) a = a - b;
        else
            while (a < 0)  a = a + b;
    else
        if (b < 0)
            while (0 < a)  a = a + b;
        else
            while (b <= a) a = a - b;
    return a;
}
```

QUOTIENT_TYPE(T)

```
template <typename T> // T models Archimedean Group
struct quotient_type
{
    typedef size_t type;
};
#define QUOTIENT_TYPE(T) typename quotient_type<T>::type

template <>
struct quotient_type<int>
{
    typedef int type;
};
```

slow_quotient_remainder

```
template <typename T> // T models Archimedean Group
pair<QUOTIENT_TYPE(T), T> slow_quotient_remainder(T a, T b)
{
    assert(b != 0);
    QUOTIENT_TYPE(T) n(0);
    if (a < 0)
        if (b < 0)
            while (a <= b) { n = n + 1; a = a - b; }
        else
            while (a < 0)  { n = n - 1; a = a + b; }
    else
        if (b < 0)
            while (0 < a)  { n = n - 1; a = a + b; }
        else
            while (b <= a) { n = n + 1; a = a - b; }
    return pair<QUOTIENT_TYPE(T), T>(n, a);
}
```

slow_quotient

```
template <typename T> // T models Archimedean Group
QUOTIENT_TYPE(T) slow_quotient(T a, T b)
{
    return slow_quotient_remainder(a, b).first;
}
```

From slow quotient to quotient

- Repeated doubling leads to fast (logarithmic complexity) power
- An inverse algorithm is possible for quotient and remainder
- As with power, the Egyptians used this algorithm to do division

remainder_nonnegative

```
template <typename T> // T models Archimedean Group
T remainder_nonnegative(T a, T b)
{
    assert(0 <= a && 0 < b);
    if (a < b) return a;
    if (a - b < b) return a - b;
    a = remainder_nonnegative(a, b + b);
    if (a < b) return a;
    return a - b;
}
```

- The first comparison is not needed in the recursive calls
- It could be eliminated by the introduction of a helper function

remainder

```
template <typename T> // T models Archimedean Group
T remainder(T a, T b)
{
    assert(b != 0);
    T r;
    if (a < 0)
        if (b < 0) {
            r = -remainder_nonnegative(-a, -b);
        } else {
            r = remainder_nonnegative(-a,  b);
            if (r != 0) r = b - r;
        }
    else
        if (b < 0) {
            r = remainder_nonnegative(a, -b);
            if (r != 0) r = b + r;
        } else {
            r = remainder_nonnegative(a,  b);
        }
    return r;
}
```

quotient_remainder_nonnegative

```
template <typename T> // T models Archimedean Group
pair<QUOTIENT_TYPE(T), T>
quotient_remainder_nonnegative(T a, T b)
{
    assert(!(a < 0) && 0 < b);
    if (a < b)
        return pair<QUOTIENT_TYPE(T), T>(0, a);
    if (a - b < b)
        return pair<QUOTIENT_TYPE(T), T>(1, a - b);
    pair<QUOTIENT_TYPE(T), T> q
        = quotient_remainder_nonnegative(a, b + b);
    QUOTIENT_TYPE(T) n = q.first + q.first;
    a = q.second;
    if (a < b)
        return pair<QUOTIENT_TYPE(T), T>(n, a);
    else
        return pair<QUOTIENT_TYPE(T), T>(n + 1, a - b);
}
```

quotient_remainder

```
template <typename T> // T models Archimedean Group
pair<QUOTIENT_TYPE(T), T> quotient_remainder(T a, T b)
{
    assert(b != 0);
    pair<QUOTIENT_TYPE(T), T> q_r;
    if (a < 0)
        if (b < 0) {
            q_r = quotient_remainder_nonnegative(-a, -b);
            q_r.second = -q_r.second;
        }
        else {
            q_r = quotient_remainder_nonnegative(-a,  b);
            if (q_r.second != 0) {
                q_r.second = b - q_r.second;
                q_r.first = q_r.first + 1;
            }
            q_r.first = -q_r.first;
        }
    else
        if (b < 0) {
            q_r = quotient_remainder_nonnegative( a, -b);
            if (q_r.second != 0) {
                q_r.second = b + q_r.second;
                q_r.first = q_r.first + 1;
            }
            q_r.first = -q_r.first;
        }
        else
            q_r = quotient_remainder_nonnegative( a,  b);
        return q_r;
}
```

quotient

```
template <typename T> // T models Archimedean Group
QUOTIENT_TYPE(T) quotient(T a, T b)
{
    return quotient_remainder(a, b).first;
}
```

Complexity of `remainder` and `quotient_remainder`

- It is trivial to see that `remainder` and `quotient_remainder` are logarithmic

Exercise

Determine the exact counts of different operations

Divisibility on an Archimedean group

Definition

For $b \neq 0$, b *divides* $a \Leftrightarrow \mathrm{remainder}(a, b) = 0$

- b divides a is denoted by $b \setminus a$

Exercise

Prove $x \setminus a \wedge x \setminus b \Rightarrow x \setminus \mathrm{remainder}(a, b)$

Greatest common divisor

Definition

The *greatest common divisor* of a and b is a divisor of a and b that is divisible by any other divisor of a and b

- Greatest common divisor is denoted by $\gcd(a, b)$

Exercise

Prove that in an Archimedean group, $x \setminus a \wedge x \setminus b \Rightarrow x \leqslant |\gcd(a, b)|$

fast_subtractive_gcd

```
template <typename T> // T models Archimedean Group
T fast_subtractive_gcd(T a, T b)
{
    while (true) {
        if (b == 0) return a;
        a = remainder(a, b);
        if (a == 0) return b;
        b = remainder(b, a);
    }
}
```

- Note: Substituting `slow_remainder` for `remainder` gives the usual rendition of the subtractive gcd algorithm

Correctness of `fast_subtractive_gcd`

Proof.

1. Each assignment preserves the property of being divisible by any divisor of the original a and b
2. If one variable becomes zero, the other divides the original a and b
3. Therefore it returns the greatest common divisor

□

Termination of `fast_subtractive_gcd`

Exercise

Prove that it always terminates for integers and rationals

Exercise

Prove that it does not always terminate for reals

Euclidean algorithm group

Definition

An Archimedean group is called a *Euclidean algorithm group* when the `fast_subtractive_gcd` algorithm terminates for all inputs

Reflection

An algebraic structure can be defined with the axiom that a particular algorithm terminates

Generalizing the greatest common divisor algorithm

- We have a structure on which the computation of greatest common divisor is based on a particular remainder algorithm
- We can generalize by passing a remainder function as a parameter

euclidean_gcd for Euclidean module

```
template
    <typename T, // T models Additive Group
     typename R> // R models Euclidean Module Remainder on T
T euclidean_gcd(T a, T b, R rem)
{
    while (true) {
        if (b == 0) return a;
        a = rem(a, b);
        if (a == 0) return b;
        b = rem(b, a);
    }
}

template <typename T> // T models Euclidean Ring
T euclidean_gcd(T a, T b)
{
    return euclidean_gcd(a, b, modulus<T>());
}
```

Euclidean module

Definition

A module E with group G and ring of coefficients R is called a *Euclidean module* if it has operations

$$\begin{aligned} \mathtt{remainder} &: G \times (G - \{0\}) \to G \\ \mathtt{quotient} &: G \times (G - \{0\}) \to R \end{aligned}$$

such that for any $a, b \neq 0$ in G

$$a = \mathtt{quotient}(a, b) \cdot b + \mathtt{remainder}(a, b)$$

and the `euclidean_gcd` algorithm terminates

(Euclidean ring)

Definition

A ring R is called an *integral domain* if $ab = 0 \Rightarrow a = 0 \vee b = 0$

Definition

An integral domain is called a *Euclidean ring* if it has a *Euclidean norm* $w : (R - \{0\}) \Rightarrow \mathbb{N}$ satisfying

- $w(xy) \geqslant w(x)$
- R has remainder and quotient, and $\mathtt{remainder}(a, b) \neq 0 \Rightarrow w(\mathtt{remainder}(a, b)) < w(b)$

The fact that the norm decreases with application of remainder assures that `euclidean_gcd` terminates

(Euclidean module and Euclidean ring)

- Every ring is a module over itself
- This implies every Euclidean ring is a Euclidean module
- This implies our algorithm can be used both for Euclidean algorithm groups and the well-known structure of Euclidean rings, such as polynomials over reals

run_quotient_remainder

```
int run_quotient_remainder()
{
    pair<quotient_type<int>::type, int> p;
    for (int i = -10; i <= 10; ++i)
        for (int j = -10; j <= 10; ++j)
            if (j != 0) {
                p = quotient_remainder(i, j);
                assert( i == p.first * j + p.second
                        && ((0 <= p.second && p.second < j)
                            || (j < p.second && p.second <= 0)));
            }
    cout << "Enter two integers, or two zeroes to terminate" << endl;
    while (true) {
        int a, b; cin >> a; cin >> b;
        if (a == 0 && b == 0)
            return 0;
        cout << "remainder = " << remainder(a, b) << endl;
        p = quotient_remainder(a, b);
        cout << "quotient_remainder = "
             << p.first << "    " << p.second << endl;
    }
}
```

Conclusions

- Our task is to combine algorithms and mathematical structures into a seamless whole by describing algorithms in abstract terms and adjusting theories to fit algorithmic requirements
- This chapter is the result of the search to find the correct mathematical setting for subtractive gcd

Reading

Donald Knuth.
The Greatest Common Divisor, and Division of Polynomials.
The Art of Computer Programming, Volume 2, 3rd edition, 1998, Sections 4.5.2 and 4.6.1, pages 333-356 and 420-439.
Exhaustive coverage of both Euclidean and Stein (binary) gcd.

Pierre Samuel.
About Euclidean Rings.
Journal of Algebra, Volume 19, 1971, pages 282-301.
Exhaustive and relatively elementary treatment of Euclidean rings.

binary_gcd_nonnegative

- In 1961, Josef Stein discovered this algorithm:

```
template <typename T> // T models Integral
T binary_gcd_nonnegative(T a, T b) {
    if (a == 0) return b;
    if (b == 0) return a;
    int d = 0;
    while (a % 2 == 0 && b%2 == 0) {
        a /= 2; b /= 2; d += 1;
    }
    while (a % 2 == 0) a /= 2;
    while (b % 2 == 0) b /= 2;
    while (true)
        if (a < b) {
            b -= a; do b /= 2; while (b % 2 == 0);
        } else if (b < a) {
            a -= b; do a /= 2; while (a % 2 == 0);
        } else
    return a << d;
}
```

Generalizations of `binary_gcd_nonnegative`

Polynomials	See Knuth (Exercise 4.6.1.6, page 435 and Solution, page 673)
Gaussian integers	See Weilert
Other algebraic integer rings	See Damgård and Frandsen, and Agarwal and Frandsen

Project

Project

Find the correct abstract setting for binary gcd (Stein domain)

Additional reading for binary gcd

Andre Weilert.
(1+ i)-ary GCD Computation in $\mathbb{Z}[i]$ as an Analogue of the Binary GCD Algorithm.
J. Symbolic Computation (2000) 30, pages 605-617.

Ivan Bjerre Damgård and Gudmund Skovbjerg Frandsen.
Efficient algorithms for GCD and cubic residuosity in the ring of Eisenstein integers.
Proceedings of the 14th International Symposium on Fundamentals of Computation Theory, Lecture Notes in Computer Science 2751, Springer-Verlag (2003), pages 109-117.

Saurabh Agarwal and Gudmund Skovbjerg Frandsen.
Binary GCD Like Algorithms for Some Complex Quadratic Rings.
ANTS 2004, pages 57-71.

run_gcd

```
int run_gcd()
{
    cout << "Enter two integers, or two zeroes to terminate" << endl;
    while (true) {
        int a, b;
        cin >> a;
        cin >> b;
        if (a == 0 && b == 0)
            return 0;
        cout << "fast_subtractive_gcd = "
             << fast_subtractive_gcd(a, b) << endl;
        cout << "euclidean_gcd(modulus<int>()) = "
             << euclidean_gcd(a, b, modulus<int>()) << endl;
        cout << "euclidean_gcd = " << euclidean_gcd(a, b) << endl;
        cout << "binary_gcd_nonnegative(abs(), abs()) = "
             << binary_gcd_nonnegative(abs(a), abs(b)) << endl;
    }
}
```

Contents I

Contents II

Contents III

Warning for readers familiar with STL

- Our treatment of iterators departs significantly from the one in the Standard Template Library

Memory

Intuition

- *Memory* is a set of locations, each with an address and a value
- Getting or setting the value given an address is “fast”
- The association of a value with a location is changeable

Examples

- The physical and virtual address spaces of a computer
- The locations visited by an algorithm
- The locations owned by a data structure

Dereferencable concepts

Definition

- A type models *dereferencable* when it has
  - a type function `VALUE_TYPE` returning a regular type
  - *dereferencing functions* that allow for reading or writing of data
- A dereferencable type that has a function `source` models *readable*
- A dereferencable type that has a function `sink` models *writable*
- A dereferencable type that has both models *mutable*

`source` and `sink`

Definition

- If `a` is an expression of type `A`, then `source(a)` returns `VALUE_TYPE(A)`
- If `a` is an expression of type `A` and `v` is an expression of type `VALUE_TYPE(A)`, then `sink(a) = v` is a well-formed statement

Axiom

If `a1` is of a writable type `A1`, `a2` is of a readable type `A2`, and `a1` and `a2` designate the same location, then immediately after executing `sink(a1) = v`, `source(a2)` evaluates to `v`

Referential equivalence

Definition

- Two objects `i` and `j` of potentially different types are *referentially equivalent* if `sink(i)` can be used with the same effect as `sink(j)` and similarly for `source`
- We write $i \sim j$ if `i` and `j` are referentially equivalent

VALUE_TYPE for T and T*

```
template <typename T> // T models Regular
struct value_type
{
    typedef T type;
};

template <typename T> // T models Regular
struct value_type<T*>
{
    typedef T type;
};

#define VALUE_TYPE(T) typename value_type<T>::type
```

Dereferencing functions for T*

```
template <typename T> // T models Regular
const T& source(T* x)
{
    return *x;
}

template <typename T> // T models Regular
T& sink(T* x)
{
    return *x;
}
```

Proper and improper addresses

Definition

A type `A` is *properly dereferencable* if `VALUE_TYPE(A)` is different than `A`

Reflection

Many algorithms are useful if we make every regular type improperly dereferencable:

```
template <typename T> // T models Regular
const T& source(const T& x)
{
    return x;
}
```

Action

Definition

An *action* is a unary function that takes an argument by reference and mutates it

Definition

An action is *regular* if applying it to two equal but not identical objects maintains their equality

transformation_from_action

```
#define ELEMENT_TYPE(T) typename T::element_type

template <typename A> // A models Action
struct transformation_from_action
{
    typedef ELEMENT_TYPE(A) argument_type;
    typedef ELEMENT_TYPE(A) result_type;

    transformation_from_action() {}
    transformation_from_action(const A& a) : a(a) {}
    result_type operator()(argument_type x) { a(x); return x; }

    A a;
};
```

action_from_transformation

```
template <typename F> // F models Transformation
struct action_from_transformation
{
    typedef ARGUMENT_TYPE(F) element_type;

    action_from_transformation() {}
    action_from_transformation(const F& f) : f(f) {}
    void operator()(element_type& x) { x = f(x); }

    F f;
};
```

Duality of actions and transformations

- The two preceding functions show that for every action, there is a corresponding transformation, and vice versa
- The complexity of an action when implemented independently of the corresponding transformation could be smaller
    - Example: interchange the first and last elements of a sequence

Iterator

Definition

An *iterator* is a regular type with a constant-time action preincrement (++) and corresponding transformation `successor`

- ++ does not have to be regular

Models of iterator

- An iterator where ++ advances an input stream
- An iterator where ++ advances an output stream
- An iterator on a singly-linked list
- An iterator on a doubly-linked list
- An iterator on a one-dimensional array
- `int*`

Functions for iterator

```
template <typename I> // I models Iterator
I successor(I f)
{
    ++f;
    return f;
}

template <typename I> // I models Iterator
DISTANCE_TYPE(I) operator-(I l, I f)
{
    DISTANCE_TYPE(I) n(0);
    while (f != l) {
        n = n + 1;
        ++f;
    }
    return n;
}
```

More functions for iterator

```
template <typename I> // I models Iterator
void operator+=(I& f, DISTANCE_TYPE(I) n)
{
    while (n != 0) {
        ++f;
        n = n - 1;
    }
}

template <typename I> // I models Iterator
I operator+(I f, DISTANCE_TYPE(I) n)
{
    f += n;
    return f;
}
```

Ranges

Definition

A *range* is a sequence $f_0, f_1, \ldots, f_n$ of distinct iterators such that $f_{i+1} = \texttt{successor}(f_i) = \texttt{successor}^{i+1}(f_0)$

- The distinctness of the iterators ensures there are no cycles

Bounded and counted ranges

Definition

A *bounded* range is specified by two iterators, f and l, such that $f = f_0$ and $l = f_n$

Definition

A *counted* range is specified by an iterator f and a nonnegative integer n, such that $f = f_0$ and n is the index of the last iterator in the defining sequence

Semi-open and closed ranges

Definition

- A range r is *semi-open* if $x \in r \Leftrightarrow x = f_i$ for $i < n$
- An *empty* semi-open range is one specified by a single iterator
- n is the *size* of a semi-open range
- We designate a semi-open range as $[f, l)$ or $[f, n)$ depending on whether it is bounded or counted

Definition

- A range r is *closed* if $x \in r \Leftrightarrow x = f_i$ for $i \leqslant n$
- There are no empty closed ranges
- $n + 1$ is the *size* of a closed range
- We designate a closed range as $[f, l]$ or $[f, n]$ depending on whether it is bounded or counted

Readable, writable, and mutable ranges

Definition

A range r is *readable*, *writable*, or *mutable* depending on the kind of iterators in it

Partial compatibility of types

Definition

Partial compatibility is a symmetric relation on regular types; if types `T` and `U` are partially compatible then

- If `u` is a variable of type `U`, these are defined and equivalent:

  ```
  T t(u);            T t; t = u;
  ```

- If `u` is a variable of type `U` and `t` is a variable of type `T`, these are defined and equivalent:

  ```
  t = u;             t = T(u);
  ```

- There is a (potentially implicit) unary predicate `bool is_representable<T>(U)` such that if `is_representable<T>(u)`, then `u == U(T(u))`

We write `PARTIALLY_COMPATIBLE(T, U)` to indicate that T and U are partially compatible in the requirements for an algorithm

Compatibility of types

Definition

Two partially compatible types `T` and `U` are called *compatible* if `is_representable<T>` and `is_representable<U>` return `true` for all their arguments; in this case we write `COMPATIBLE(T, U)`

Preconditions for copying

- A readable range for input
- A writable range of sufficient size for output
- Partial compatibility between input and output value types
- Representability of values of the input range in the output value type

copy

```
template
    <typename I, // I models Readable Iterator
     typename O> // O models Writable Iterator
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I), VALUE_TYPE(O))
O copy(I f, I l, O r)
{
    while (f != l) {
        sink(r) = source(f);
        ++f;
        ++r;
    }
    return r;
}
```

Example using copy

```
template
    <typename N, // N models Integer
     typename O> // O models Writable Iterator
     // PARTIALLY_COMPATIBLE(N, VALUE_TYPE(O))
O iota(N n, O r) // like APL ι
{
    return copy<N, O>(0, n, r);
}
```

copy_output_bounded

```
template
    <typename I, // I models Readable Iterator
     typename O> // O models Writable Iterator
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I), VALUE_TYPE(O))
pair<I, O> copy_output_bounded(I f, I l, O r_f, O r_l)
{
    while (f != l && r_f != r_l) {
        sink(r_f) = source(f);
        ++f;
        ++r_f;
    }
    return pair<I, O>(f, r_f);
}
```

Reflection on returning useful information

- If an algorithm computes a quantity that could be of use to the client and would be otherwise expensive or impossible to obtain, it should be returned
    - It is worth the small constant time to return it

copy_n

```
template
    <typename I, // I models Readable Iterator
     typename O> // O models Writable Iterator
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I), VALUE_TYPE(O))
pair<I, O> copy_n(I f, DISTANCE_TYPE(I) n, O r)
{
    while (n != 0) {
        sink(r) = source(f);
        ++f;
        ++r;
        n = n - 1;
    }
   return pair<I, O>(f, r);
}
```

copy_n_unrolled

```
template
    <typename I, // I models Readable Iterator
     typename O> // O models Writable Iterator
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I), VALUE_TYPE(O))
pair<I, O> copy_n_unrolled(I f, DISTANCE_TYPE(I) n, I r)
{
    while (n >= 4) {
        sink(r) = source(f); ++f; ++r;
        sink(r) = source(f); ++f; ++r;
        sink(r) = source(f); ++f; ++r;
        sink(r) = source(f); ++f; ++r;
        n = n - 4;
    }
    return copy_n(f, n, r);
}
```

Forward iterator

Definition

A *forward iterator* is an iterator with a regular action `operator++`

Reflection

Two concepts can be different only by an axiom, without adding any new operations

Models of forward iterator

- An iterator on a singly-linked list
- An iterator on a doubly-linked list
- An iterator on a one-dimensional array
- `int*`

Advantages of regularity of `operator++`

Reflection

Forward iterators allow

- algorithms that maintain more than one iterator into a range
- multipass algorithms

Bidirectional iterator

Definition

A *bidirectional iterator* is a forward iterator with

- a regular constant-time action `operator--`
- a corresponding (constant-time) transformation `predecessor`

```
template <typename I> // I models Bidirectional Iterator
I predecessor(I f)
{
    --f;
    return f;
}
```

Axioms for bidirectional iterator

Axiom

For any iterator x in a range $[f, l)$:

$$x \neq l \Rightarrow \mathtt{predecessor}(\mathtt{successor}(x)) = x$$
$$x \neq f \Rightarrow \mathtt{successor}(\mathtt{predecessor}(x)) = x$$

Models of bidirectional iterator

- An iterator on a doubly-linked list
- An iterator on a one-dimensional array
- `int*`

Functions for bidirectional iterator

```
template <typename I> // I models Bidirectional Iterator
void operator-=(I& f, DISTANCE_TYPE(I) n)
{
    while (n != 0) {
        --f;
        n = n - 1;
    }
}

template <typename I> // I models Bidirectional Iterator
I operator-(I f, DISTANCE_TYPE(I) n)
{
    f -= n;
    return f;
}
```

Using copy with overlapping ranges

- If we are copying a range to a destination that begins within the source range, copy does not do what we want

copy_backward

```
template
    <typename I, // I models Readable Bidirectional Iterator
     typename O> // O models Writable Bidirectional Iterator
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I), VALUE_TYPE(O))
O copy_backward(I f, I l, O r)
{
    while (f != l) {
        --l;
        --r;
        sink(r) = source(l);
    }
    return r;
}
```

Indexed iterator

Definition

An *indexed iterator* of type `I` is a forward iterator with

- a <u>constant-time</u> mutating function `operator+=(I&, DISTANCE_TYPE(I))`
- a <u>constant-time</u> function `operator-(I, I)` returning a value of `DISTANCE_TYPE(I)`

Reflection

We are making secondary functions primitive

Lemma

Every indexed iterator has a constant-time function `operator+(I, DISTANCE_TYPE(I))`

Models of indexed iterator

- An iterator on a one-dimensional array
- `int*`

copy_n_unrolled_indexed

```
template
    <typename I, // I models Readable Indexed Iterator
     typename O> // O models Writable Indexed Iterator
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I), VALUE_TYPE(O))
pair<I, O> copy_n_unrolled_indexed(I f, DISTANCE_TYPE(I) n, O r)
{
    while (n >= 4) {
        sink(r)     = source(f);
        sink(r + 1) = source(f + 1);
        sink(r + 2) = source(f + 2);
        sink(r + 3) = source(f + 3);
    }
    return copy_n(f, n, r);
}
```

Reflection on aliasing

- Out-of-order execution of adjacent iterations speeds up the execution of `copy`
- A compiler cannot achieve this because it cannot determine that the source and destination ranges do not *alias*
- Programmers know when this is the case, and should be able to specify when such interleaved execution is legitimate
- A special statement `forall` will allow them to do so

copy_parallel for disjoint ranges

```
template
    <typename I, // I models Readable Indexed Iterator
     typename O> // O models Writable Indexed Iterator
     // VALUE_TYPE(I) == VALUE_TYPE(O)
void copy_parallel(I f, DISTANCE_TYPE(I) n, O r)
{
    forall(DISTANCE_TYPE(I) i, 0, n) {
        sink(r + i) = source(f + i);
    }
}
```

Random access iterator

Definition

A *random access iterator* is an indexed iterator type I with a signed integral DIFFERENCE_TYPE(I) large enough to contain distances and their negations and the following constant-time operations:

```
bool operator<(I, I);
void operator--();
void operator+=(I&, DIFFERENCE_TYPE(I));
void operator-=(I&, DIFFERENCE_TYPE(I));
I operator+(I, DIFFERENCE_TYPE(I));
I operator-(I, DIFFERENCE_TYPE(I));
  // +=, -=, +, and - accept negative values on the right
DIFFERENCE_TYPE(I) operator-(I, I);
  // - accepts iterators in either order
```

Axioms of random access iterator

Exercise

Write an appropriate set of axioms relating the operations to each other

Models of random access iterator

- `int*`

Equivalence of random access and indexed iterator

Theorem

For any function defined on a range of random access iterators, there is another function defined on indexed iterators with the same asymptotic complexity

Proof of the equivalence

Assuming these definitions:

```
typedef DISTANCE_TYPE(I) U;
typedef sign_extended<U> W;
```

where `sign_extended` is a templated struct adding a sign bit and appropriate integer operations, we rewrite the function with these substitutions:

`DIFFERENCE_TYPE(I)`	`W`
`i < j`	`i - f < j - f`
`i += n` when `n < 0`	`i = f + ((i - f) - U(-n))`
`i -= n`	`i += -n`
`i - j`	`W(i - f) - W(j - f)`

Reflection on random access and indexed iterator

Reflection

- The theorem shows the theoretical equivalence of these concepts in any context in which the beginning of ranges are known
  - `copy_backward` does not satisfy this requirement!
- In practice we have found there is no performance penalty for using the weaker concept in those algorithms

copy_backward

- copy_backward is not realizable for indexed iterators
- copy_backward_n, which is often just as useful, is realizable

```
template
    <typename I, // I models Readable Indexed Iterator
     typename O> // O models Writable Indexed Iterator
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I), VALUE_TYPE(O))
O copy_backward_n(I f, DISTANCE_TYPE(I) n, O r)
{
    while (n != 0) {
        n = n - 1;
        sink(r + n) = source(f + n);
    }
    return r;
}
```

Relationships between iterator theories

$$\begin{tikzcd}
 & & \mathrm{BI} \arrow[dr] & \\
\mathrm{It} \arrow[r] & \mathrm{FI} \arrow[ur] \arrow[dr] & & \mathrm{RI} \\
 & & \mathrm{II} \arrow[ur] &
\end{tikzcd}$$

Conclusions

- Refinement generates mathematical structures such as groups, Abelian groups, totally-ordered groups, and Archimedean groups
- It also generates concepts describing the fundamental notion of computer science, iterating through a data structure
- We have used three types of refinement, by adding
  - an operation
  - an axiom
  - a tighter complexity requirement

Reading

Alexander Stepanov and Meng Lee.
The Standard Template Library.
HP Laboratories Technical Report 95-11(R.1), November 14, 1995.
Introduced different categories of iterators and their axioms.

Intuition for segmented iterator

- There are data structures where ++ can be implemented faster within certain ranges or *segments*
  - hash tables
  - adjacency list representation of graphs
  - STL-like deques
- It is possible to optimize many algorithms for such structures by transforming inner loops into nested loops:
  - a loop over segments
  - a loop within a segment
- This results in a new dimension of the classification of iterators:
  - homogeneous
  - segmented

Segmented iterator

Definition

A *segmented iterator* is a forward iterator `I` with

- an affiliated iterator type `SEGMENT_ITERATOR(I)`
- constant-time regular operations:

```
SEGMENT_ITERATOR(I) begin(I);
SEGMENT_ITERATOR(I) end(I);
I operator+(I, SEGMENT_ITERATOR(I));
```

- For any $\mathrm{i} \in [\mathrm{f}, \mathrm{l}]$, `begin`, `end`, and `operator+` are defined
- For any $w \in [\texttt{begin}(\mathrm{i}), \texttt{end}(\mathrm{i})]$, $\mathrm{i} + w$ is defined

Axioms for segmented iterator

If

- $[\mathsf{f}, \mathsf{l})$ is a range of segmented iterators
- $\mathsf{i} \in [\mathsf{f}, \mathsf{l})$
- w is a segment iterator in $[\mathbf{begin}(\mathsf{i}), \mathbf{end}(\mathsf{i}))$

then the following hold:

1. $\mathsf{i} \sim \mathbf{begin}(\mathsf{i})$
2. $\mathbf{begin}(\mathsf{i} + w) = w$
3. $\mathsf{i} + \mathbf{begin}(\mathsf{i}) = \mathsf{i}$
4. $\mathsf{i} + (\mathbf{begin}(\mathsf{i}) + 1) = \mathsf{i} + 1$
5. $\mathbf{begin}(\mathsf{i}) + 1 \neq \mathbf{end}(\mathsf{i}) \Rightarrow \mathbf{begin}(\mathsf{i}) + 1 = \mathbf{begin}(\mathsf{i} + 1)$
6. $\mathbf{begin}(\mathsf{i}) \neq \mathbf{end}(\mathsf{i})$

copy_from_segmented

```
template
    <typename I, // I models Readable Segmented Iterator
     typename O> // O models Writable Output Iterator
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I), VALUE_TYPE(O))
O copy_from_segmented(I f, I l, O r)
{
    while (f + end(f) != l + end(l)) {
        // f and l are in different segments
        r = copy(begin(f), end(f), r);
        f = f + end(f);
    }
    // f and l are in the same segment
    return copy(begin(f), begin(l), r);
}
```

Project

- Implement segmented iterators for one or more data structures
  - STL-like `deque`
  - SGI STL-like hashed containers
- Produce segmented iterator versions of suitable (STL) algorithms
- Analyze if the axioms are independent, consistent, and complete

Contents I

Contents II

Contents III

Permutation

Definition

A function f is *one-to-one* if $f(x) = f(y) \Rightarrow x = y$

Definition

A *permutation* on a (finite) set is a one-to-one transformation on the set

Example of a permutation on $[0, 6)$

$$\begin{aligned} p(0) &= 5 \\ p(1) &= 2 \\ p(2) &= 4 \\ p(3) &= 3 \\ p(4) &= 1 \\ p(5) &= 0 \end{aligned}$$

Identity permutation

Definition

A *fixed point* of a transformation is an element x such that $f(x) = x$

Definition

The *identity permutation* on a set S, $\mathrm{identity}_S$, maps each element of S to itself; every element in S is a fixed point of $\mathrm{identity}_S$

Composition of permutations

Definition

If p and q are two permutations on a set S, the *composition* $q \circ p$ takes $x \in S$ to $q(p(x))$

Lemma

The composition of permutations is a permutation

Lemma

Composition of permutations is associative

Inverse of a permutation

Lemma

For every permutation p on a set S, there is an *inverse permutation* p^{-1} such that $\mathrm{p}^{-1} \circ \mathrm{p} = \mathrm{p} \circ \mathrm{p}^{-1} = \mathrm{identity}_S$

Permutation group

- The permutations on a set form a group under composition

Lemma

Every (finite) group is a subgroup of a permutation group of its elements where every permutation in a subgroup is generated by multiplying all the elements by an individual element

Example

$\times$	**1**	**2**	**3**	**4**
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

Multiplication mod5:
Every row and column of the multiplication table is a permutation

Cycles in a permutation

- Since a permutation on a set is one-to-one, the finiteness of the set implies that the orbit of each element is circular

Definition

A *cycle* is a circular orbit within a permutation

Definition

A *trivial cycle* is one with a cycle size of 1

- The element in a trivial cycle is a fixed point

Lemma

Every element in a permutation belongs to a unique cycle

Example of cycle decomposition

$$
\begin{aligned}
p(0) &= 5\\
p(1) &= 2\\
p(2) &= 4\\
p(3) &= 3\\
p(4) &= 1\\
p(5) &= 0
\end{aligned}
$$

$$p = (0\;5)(1\;2\;4)(3)$$

Structure of cycle decomposition

- Any permutation of a set with n elements contains $k \leqslant n$ disjoint cycles
- Each cycle is itself a permutation of this set, called a *cyclic permutation*
- Disjoint cyclic permutations commute
- Every permutation can be represented as a product of its cycles
- The inverse of a permutation is the product of the inverses of its cycles

Finite sets

Definition

A *finite set* S of size n is a set for which there exists a pair of functions

$$\mathrm{choose}_S : [0, n) \to S$$
$$\mathrm{index}_S : S \to [0, n)$$

satisfying

$$\mathrm{choose}_S(\mathrm{index}_S(x)) = x$$
$$\mathrm{index}_S(\mathrm{choose}_S(i)) = i$$

Index permutation of a permutation

Definition

If p is a permutation on a finite set S of size n, there is a corresponding *index permutation* p' on $[0, n)$ defined as

$$p'(i) = \mathrm{index}_S(p(\mathrm{choose}_S(i)))$$

Lemma

$$p(x) = \mathrm{choose}_S(p'(\mathrm{index}_S(x)))$$

- We will frequently define permutations by the corresponding index permutations

Rearrangement

Definition

A *rearrangement* is an algorithm that rearranges the elements of a mutable range to satisfy a given postcondition

Classifying rearrangements

Rearrangements are classified according to the following fundamental characteristics:

- Characterization of the postcondition
  - Postcondition kind
  - Stability
- Implementation constraints
  - Iterator requirements
  - Mutative versus copying
- Complexity
  - Space
  - Time

Postcondition kind

position-based The destination of a value depends only on its original position and not on the value itself; for example, "reverse the range"

bin-based The destination of a value depends only on the result of applying a k-way *bin function* to that value, and not directly on the value itself; for example, "move bad values before good ones"

- A useful subcase is predicate-based algorithms, where $k = 2$

ordering-based The destination of a value depends only on the outcome of applications of an ordering to pairs of values in the range; for example, "put the smallest value first"

Stability

Definition

A rearrangement is *stable* if it respects the original order of the range to the maximal extent possible while satisfying the postcondition

Examples

- Stable index partition
- Stable partition
- Stable sort

Iterator requirements

- For the same problem, there are often different algorithms for different iterator requirements

Examples

- `reverse_forward`
- `reverse_bidirectional`
- `reverse_indexed`

Mutative versus copying

Definition

Rearrangements as we have defined them are *mutative*

Definition

A rearrangement is *copying* if it sets a writable range to a rearranged copy of a readable range in way that satisifies a given postcondition

- A copying rearrangement could always be obtained by composing `copy` with the corresponding mutative rearrangement
- Often, however, there are faster algorithms that rearrange while copying

Space complexity

Definition

An algorithm is *in place* (or *in situ*) if it uses an amount of additional space that is (poly-)logarithmic in the size of the input

Definition

A *memory-adaptive* algorithm uses as much additional space as it can acquire to maximize performance

- A small percentage of additional space, while theoretically "linear," can lead to a large performance improvement

Definition

An algorithm *with buffer* requires the caller to provide a buffer of specified size

Time complexity

- Rearranging n elements might take $n + k$ assignments for some constant k
- $n \log n$ algorithms arise
  - Determining the final position may require collecting additional information, as in sorting
  - Absence of random access on some iterators requires multipass algorithms, as with in place random shuffle for forward iterators

cycle_2_default

```
template <typename I> // I models Mutable Iterator
void cycle_2_default(I x, I y)
{
    VALUE_TYPE(I) t = source(x);
            sink(x) = source(y);
            sink(y) = t;
}
```

Problems with `cycle_2_default`

- It could be much slower than necessary
  - `std::vector` copies element-by-element instead of interchanging the headers
- It could throw an exception due to unnecessary resource allocation
- We should exploit the fact it is based on a permutation

Underlying type

Definition

For any type `T`, the *underlying type* `U` is an affiliated type satisfying:

- `T` and `U` are compatible
- References with value types `T` and `U` may be reinterpretively cast into each other
- Construction of type `U` and assignment to type `U` never throw an exception
- An object of type `U` may only be used to hold temporary values while implementing a rearrangement of a range of `T` objects
- A reference to an object of type `U` may be reinterpretively cast to a reference to `T` and passed as a `const T&` parameter

We denote the underlying type of `T` as `UNDERLYING_TYPE(T)`

Motivation for underlying type

- The implementation of types `T` and `U` *could* exploit this restriction to save time
- As we shall see when implementing containers, underlying type allows multiple headers to point to the same data while performing rearrangements

Underlying type

```
template <typename T> // T models Regular
struct underlying_type
{
    typedef T type; // default
};

#define UNDERLYING_TYPE(T) typename underlying_type<T>::type

template <typename I> // I models Readable Iterator
const UNDERLYING_TYPE(VALUE_TYPE(I))&
underlying_source(I x)
{
    return reinterpret_cast<
                const UNDERLYING_TYPE(VALUE_TYPE(I))&>(source(x));
}

template <typename I> // I models Writable Iterator
UNDERLYING_TYPE(VALUE_TYPE(I))&
underlying_sink(I x)
{
    return reinterpret_cast<
                UNDERLYING_TYPE(VALUE_TYPE(I))&>(sink(x));
}
```

cycle_2

```
template <typename I> // I models Mutable Iterator
void cycle_2(I x, I y)
{
    UNDERLYING_TYPE(VALUE_TYPE(I)) t = underlying_source(x);
                   underlying_sink(x) = underlying_source(y);
                   underlying_sink(y) = t;
}
```

Underlying type and invariants

To be correct, it is sufficient for a sequence of code using `UNDERLYING_TYPE(T)` to:

- Be within a *critical section*: that is, avoid concurrent access
- Avoid any exceptions being thrown
- Restore the class invariants of `T` by the end of the sequence

Reflection

Disciplined violation of invariants to enhance performance is perfectly legitimate

cycle_left_3

```
template <typename I> // I models Mutable Iterator
void cycle_left_3(I x, I y, I z)
{
    UNDERLYING_TYPE(VALUE_TYPE(I)) t = underlying_source(x);
                  underlying_sink(x) = underlying_source(y);
                  underlying_sink(y) = underlying_source(z);
                  underlying_sink(z) = t;
}
```

swap

- We can interchange the contents of two variables using underlying type

```
#define UNDERLYING_REF(T) reinterpret_cast<UNDERLYING_TYPE(T)&>

template <typename T> // T models Regular
void swap(T& x, T& y)
{
    UNDERLYING_TYPE(T) tmp;
    tmp = UNDERLYING_REF(T)(x);
    UNDERLYING_REF(T)(x) = UNDERLYING_REF(T)(y);
    UNDERLYING_REF(T)(y) = tmp;
}
```

To-permutation and from-permutation

Definition

- Every rearrangement corresponds to two permutations of its range
    - A *to-permutation* mapping an iterator i to the iterator pointing to the destination of the element at i
    - A *from-permutation* mapping an iterator i to the iterator pointing to the origin of the element moved to i
- These two permutations are inverses of each other

do_cycle_from

```
template
    <typename I, // I models Mutable Forward Iterator
     typename P> // P models From-Permutation on I
void do_cycle_from(I i, P p)
{
    UNDERLYING_TYPE(VALUE_TYPE(I)) t = underlying_source(i);
    I f = i;
    I n = p(i);
    while (n != i) {
        underlying_sink(f) = underlying_source(n);
        f = n;
        n = p(n);
    }
    underlying_sink(f) = t;
}
```

Strict lower bound for rearrangement

- Using `do_cycle_from` for every nontrivial cycle of a rearrangement allows us to derive a formula for a lower bound on the number of assignments:

  $$n + c - t$$

  where n is the number of elements, c is the number of cycles, and t is the number of nontrivial cycles

do_cycle_to

Exercise

Implement `do_cycle_to` and compare the number of assignments it performs to `do_cycle_from`

Reverse permutation

Definition

The permutation p on a finite set with n elements defined by an index permutation $p(i) = (n - 1) - i$ is called the *reverse permutation* of the set

- The number of nontrivial cycles in a reverse permutation is $\lfloor n/2 \rfloor$; the number of trivial cycles is $n \bmod 2$
- $\lfloor n/2 \rfloor$ is the largest possible number of nontrivial cycles in a permutation

Reverse rearrangement

Definition

The *reverse rearrangement* of a range is the rearrangement induced by the reverse permutation

- The lower bound formula gives the number of assignments as $n + c - t = 3\lfloor n/2 \rfloor$

reverse_n_indexed

- The definition of reverse directly gives this:

```
template <typename I> // I models Mutable Indexed Iterator
void reverse_n_indexed(I f, DISTANCE_TYPE(I) n)
{
    DISTANCE_TYPE(I) i(0);
    while (i < n / 2) {
        cycle_2(f + i, f + ((n - 1) - i));
        i = i + 1;
    };
}
```

- The code does the lower bound number of assignments
- It works for forward iterators, but with quadratic number of iterator increments
- Since all the cycles are disjoint, this code benefits from `forall`

Return value of `reverse`

- It is tempting to define the `reverse` algorithms to return the range of elements that were not moved
  - The middle element when the size of the range is odd
  - The empty range between the two “middle” elements when the size of the range is even
- However, we do not know of any example when it is useful and, therefore, return `void`

reverse_bidirectional

```
template <typename I> // I models Mutable Bidirectional Iterator
void reverse_bidirectional(I f, I l)
{
    while (f != l && f != predecessor(l)) {
        --l;
        cycle_2(f, l);
        ++f;
    }
}
```

reverse_n_bidirectional

```
template <typename I> // I models Mutable Indexed Iterator
void reverse_n_bidirectional(I f, I l, DISTANCE_TYPE(I) n)
{
    assert(n <= l - f);
    DISTANCE_TYPE(I) i(0);
    while (i < n / 2) {
        --l;
        cycle_2(f, l);
        ++f;
        i = i + 1;
    }
}
```

- Passing $n < l - f$ effectively gives `reverse_until_n`

reverse_indexed

```
template <typename I> // I models Mutable Indexed Iterator
void reverse_indexed(I f, I l)
{
    reverse_n_indexed(f, l - f);
}
```

Intuition for divide and conquer reverse algorithm

1. Split the range into two parts
2. Reverse each part
3. Interchange the parts

Illustration of divide and conquer reverse algorithm

```
[a]  b  [c]  d  [e]  f  [g]  [h]  i  [j]  k  [l]  m  [n]
[c   b   a]  d  [g   f   e]  [j   i   h]  k  [n   m   l]
[g   f   e   d   c   b   a]  [n   m   l   k   j   i   h]
 n   m   l   k   j   i   h    g   f   e   d   c   b   a
```

swap_ranges_n

```
template
    <typename I1, // I1 models Mutable Iterator
     typename I2, // I2 models Mutable Iterator
     typename N>  // N models Integer
    // PARTIALLY_COMPATIBLE(VALUE_TYPE(I1), VALUE_TYPE(I2))
pair<I1, I2> swap_ranges_n(I1 f1, I2 f2, N n)
{
    while (n != 0) {
        cycle_2(f1, f2);
        ++f1;
        ++f2;
        n = n - 1;
    };
    return pair<I1, I2>(f1, f2);
}
```

reverse_n_recursive

```
template <typename I> // I models Mutable Forward Iterator
I reverse_n_recursive(I f, DISTANCE_TYPE(I) n)
{
    const DISTANCE_TYPE(I) h = n / 2;
    const DISTANCE_TYPE(I) r = n - 2 * h;
    if (h == 0)
        return f + n;
    I m = reverse_n_recursive(f, h);
    m += r;
    I l = reverse_n_recursive(m, h);
    swap_ranges_n(f, m, h);
    return l;
}
```

Correctness of `reverse_n_recursive`

Lemma

The reverse permutation $[0, n)$ is the only permutation satisfying $i < j \Rightarrow p(j) < p(i)$

1. This condition obviously holds for ranges of size 1
2. The recursive calls inductively establish that the condition holds within each half
3. `swap_ranges_n` reestablishes the condition between the halves (and the skipped middle element, if any)

Complexity of `reverse_n_recursive`

Lemma

For a range of length $n = \sum_{i=0}^{\lfloor \log n \rfloor} a_i 2^i$, where a_i is the i^{th} digit in the binary representation of n, the number of assignments equals $\frac{3}{2} \sum_{i=0}^{\lfloor \log n \rfloor} a_i i 2^i$

reverse_n_forward

```
template <typename I> // I models Mutable Forward Iterator
void reverse_n_forward(I f, DISTANCE_TYPE(I) n)
{
    reverse_n_recursive(f, n);
}
```

reverse_forward

```
template <typename I> // I models Mutable Forward Iterator
void reverse_forward(I f, I l)
{
    reverse_n_forward(f, l - f);
}
```

reverse_copy_n

```
template
    <typename B, // B models Mutable Bidirectional Iterator
     typename I> // I models Mutable Iterator
     // VALUE_TYPE(B) == VALUE_TYPE(I)
I reverse_copy_n(B l, DISTANCE_TYPE(I) n, I r)
{
    while (n != 0) {
        --l;
        sink(r) = source(l);
        ++r;
        n = n - 1;
    }
    return r;
}
```

reverse_n_with_buffer

```
template
    <typename I, // I models Mutable Forward Iterator
     typename B> // B models Mutable Bidirectional Iterator
     // UNDERLYING_TYPE(VALUE_TYPE(I)) == VALUE_TYPE(B)
I reverse_n_with_buffer(I f, DISTANCE_TYPE(I) n, B b)
{
    return reverse_copy_n(copy_n(f, n, b), n, f);
}
```

reverse_n_adaptive

```
template
    <typename I, // I models Mutable Forward Iterator
     typename B> // B models Mutable Bidirectional Iterator
     // UNDERLYING_TYPE(VALUE_TYPE(I)) == VALUE_TYPE(B)
I reverse_n_adaptive(I f, DISTANCE_TYPE(I) n, B b, DISTANCE_TYPE(I) n_b)
{
    const DISTANCE_TYPE(I) h = n / 2;
    const DISTANCE_TYPE(I) r = n - 2 * h;
    if (h == 0)
        return f + n;
    if (n <= n_b)
        return reverse_n_with_buffer(f, n, b);
    I m = reverse_n_adaptive(f, h, b, n_b);
    m += r;
    I l = reverse_n_adaptive(m, h, b, n_b);
    swap_ranges_n(f, m, h);
    return l;
}
```

Complexity of `reverse_n_adaptive`

Exercise

Derive a formula for the number of assignments performed by `reverse_n_adaptive` for given range and buffer sizes

Conjecture

Conjecture

There is no algorithm that reverses a forward iterator range with polylogarithmic additional space and in linear time

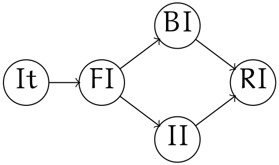
Selecting algorithms according to requirements

- It would be nice to have automatic selection from a family of algorithms based on the requirements the types satisfy
- The mechanism used by the current `C++` standard library is called *category dispatch*
- When concepts are added to `C++` there will be a more natural way of expressing it

Mechanics of category dispatch

- A tag type (containing no data) is defined for each category
- A type function is defined to obtain a type tag from a type
- Overloading is used to select from multiple signatures differing only by a tag type

Iterator tag types


$$\begin{tikzcd} & & BI \arrow[dr] & \\ It \arrow[r] & FI \arrow[ur] \arrow[dr] & & RI \\ & & II \arrow[ur] & \end{tikzcd}$$

```
struct iterator_tag               {};
struct forward_iterator_tag       {};
struct bidirectional_iterator_tag {};
struct indexed_iterator_tag       {};
struct random_access_iterator_tag {};
```

Iterator category type function

```
template <typename T> // T models Iterator
struct iterator_category
{
    typedef iterator_tag category;
};

#define ITERATOR_CATEGORY(T) typename iterator_category<T>::category

template <typename T> // T models Regular
struct iterator_category<T*>
{
    typedef random_access_iterator_tag category;
};
```

Iterator category dispatch for `reverse_n` and `reverse`

```
template <typename I> // I models Mutable Forward Iterator
void reverse_n(I f, DISTANCE_TYPE(I) n)
{
    reverse_n(f, n, ITERATOR_CATEGORY(I)());
}

template <typename I> // I models Mutable Forward Iterator
void reverse(I f, I l)
{
    reverse(f, l, ITERATOR_CATEGORY(I)());
}
```

Iterator category dispatch cases for `reverse_n`

```
template <typename I> // I models Mutable Forward Iterator
void reverse_n(I f, DISTANCE_TYPE(I) n, forward_iterator_tag)
{
    reverse_n_forward(f, n);
}

template <typename I> // I models Mutable Bidirectional Iterator
void reverse_n(I f, DISTANCE_TYPE(I) n, bidirectional_iterator_tag)
{
    reverse_n_bidirectional(f, n);
}

template <typename I> // I models Mutable Indexed Iterator
void reverse_n(I f, DISTANCE_TYPE(I) n, indexed_iterator_tag)
{
    reverse_n_indexed(f, n);
}

template <typename I> // I models Mutable Random Access Iterator
void reverse_n(I f, DISTANCE_TYPE(I) n, random_access_iterator_tag)
{
    reverse_n_indexed(f, n);
}
```

Iterator category dispatch for reverse

```
template <typename I> // I models Mutable Forward Iterator
void reverse(I f, I l, forward_iterator_tag)
{
    reverse_forward(f, l);
}

template <typename I> // I models Mutable Bidirectional Iterator
void reverse(I f, I l, bidirectional_iterator_tag)
{
    reverse_bidirectional(f, l);
}

template <typename I> // I models Mutable Indexed Iterator
void reverse(I f, I l, indexed_iterator_tag)
{
    reverse_indexed(f, l);
}

template <typename I> // I models Mutable Random Access Iterator
void reverse(I f, I l, random_access_iterator_tag)
{
    reverse_indexed(f, l);
}
```

Conclusions

- Permutations lead us back to algebra, specifically group theory
- A permutation of a range induces a rearrangement
- A taxonomy of rearrangements was presented
- Weakening iterator requirements often raises interesting algorithmic problems
- A convention known as iterator category dispatch allows automatic selection of the appropriate algorithm
- We introduced a new class of algorithms, memory-adaptive algorithms

Reading

Donald Knuth.
Section 1.2.5: Permutations and Factorials.
The Art of Computer Programming, Volume 1, Addison-Wesley, 1997, pages 45-50.

Contents I

Contents II

Contents III

Definition of rotation

Definition

The permutation p on a finite set with n elements defined by an index permutation $p(i) = (i + k) \bmod n$ is called the k-*rotation* of the set

Lemma

The inverse of a k-rotation on an n-element set is an $(n - k)$-rotation

Least common multiple

Definition

The *least common multiple* of integers a and b, written $\mathrm{lcm}(a, b)$, is the smallest integer m such that $a \setminus m$ and $b \setminus m$

Lemma

$\mathrm{lcm}(a, b)$ is a multiple of a and b that divides any other multiple of a and b

Connecting lcm and gcd

Theorem

$a \cdot b = \text{lcm}(a, b) \cdot \text{gcd}(a, b)$

Proof.

If the prime decompositions of a and b are respectively $\prod p_i^{u_i}$ and $\prod p_i^{v_i}$, then $a \cdot b = \prod p_i^{u_i + v_i}$, $\text{lcm}(a, b) = \prod p_i^{\max(u_i, v_i)}$, and $\text{gcd}(a, b) = \prod p_i^{\min(u_i, v_i)}$, and the result follows from the useful identity $x + y = \max(x, y) + \min(x, y)$ □

(Divisibility lattice of integers)

Definition

An ordered set E is said to be a *lattice* if every subset consisting of two elements of E has a least upper bound and a greatest lower bound in E

Example

The set of integers $\geqslant 1$, ordered by the relation "m divides n" between $m n$ and n, is a lattice; the least upper bound of $\{m, n\}$ is $\mathrm{lcm}(m, n)$, and the greatest lower bound is $\gcd(m, n)$

- The definition and example are adapted from:

  N. Bourbaki.
  Chapter 3, Section 1.11.
  Theory of Sets, Springer, 2004, pages 145-146.

Cycle structure of k-rotation on n elements

- An element with index i is in the cycle

$$\{i, (i+k) \bmod n, (i+2k) \bmod n, \ldots\} = \{(i+uk) \bmod n\}$$

- The length of the cycle is the smallest positive integer m such that $i = (i + mk) \bmod n$
- This is equivalent to $mk \bmod n = 0$, which shows the length of the cycle to be independent of i
- Since m is the smallest positive number such that $mk \bmod n = 0$, it is obvious that $\mathrm{lcm}(k, n) = mk$
- $m = \frac{\mathrm{lcm}(k,n)}{k} = \frac{kn}{\gcd(k,n)k} = \frac{n}{\gcd(k,n)}$
- The number of cycles, therefore, is $\gcd(k, n)$

Disjoint cycles of k-rotation on n elements

- Consider two different elements in a cycle, $(i + uk) \bmod n$ and $(i + vk) \bmod n$
- The distance between them is $|(i+uk) \bmod n-(i+vk) \bmod n| = (u-v)k \bmod n = (u-v)k-pn$ where $p = \text{quotient}((u-v)k, n)$
- Since both k and n are divisible by $d = \gcd(k, n)$, so is the distance
- Therefore the distance between different elements in the same cycle is $\geqslant d$
- Elements with indices in $[0, d)$ belong to disjoint cycles

Interface for `rotate`

- k-rotation rearrangement of a range $[f, l)$ is equivalent to interchanging the relative positions of $[f, m)$ and $[m, l)$, where $m = f + ((l - f) - k) = l - k$
- Experimentally, m is a more useful input than k (and when forward or bidirectional iterators are involved, it avoids performing linear-time operations to compute m from k)
- Experimentally, returning the iterator $m' = f + k$ pointing to the new position of element at f is useful for many other algorithms
- Joseph Tighe suggests returning a pair, m and m', in the order constituting a valid range; while it is an interesting suggestion and preserves all the information, we do not yet know of a compelling use of such interface

From-permutation for k-rotation in terms of f, m, l

- To-permutation: $p(i) = (i + k) \bmod n$
- From-permutation: $p^{-1}(i) = (i + (n - k)) \bmod n$
- $n = l - f$
- Since m goes to f, $m + k = f + n \Rightarrow n - k = m - f$
- $k = n - (m - f) = l - m$
- $i < k \Rightarrow i + (n - k) < n \Rightarrow p^{-1}(i) = i + (n - k)$
- $i \geqslant k \Rightarrow p^{-1}(i) = i + (n - k) - n = i - k$

A from-permutation for indexed iterator

```
template <typename I> // I models Mutable Indexed Iterator
struct k_rotate_from_permutation_indexed
{
    I f;
    DISTANCE_TYPE(I) k;
    DISTANCE_TYPE(I) n_minus_k;

    k_rotate_from_permutation_indexed(I f, I m, I l) :
        f(f), k(l - m), n_minus_k(m - f) {}
    I operator()(I x) {
        DISTANCE_TYPE(I) i = x - f;
        if (i < k)
            i = i + n_minus_k;
        else
            i = i - k;
        return f + i;
    }
};
```

- The absence of $<$ and $-$ for indexed iterators costs us a couple of extra operations

A from-permutation for random access iterators

```
template <typename I> // I models Mutable Random Access Iterator
struct k_rotate_from_permutation_random_access
{
    DISTANCE_TYPE(I) k;
    DISTANCE_TYPE(I) n_minus_k;
    I m_prime;

    k_rotate_from_permutation_random_access(I f, I m, I l) :
        k(l - m), n_minus_k(m - f), m_prime(f + (l - m)) {}
    I operator()(I x) {
        if (x < m_prime)
            return x + n_minus_k;
        else
            return x - k;
    }
};
```

rotate_indexed_helper

```
template
    <typename I, // I models Mutable Indexed Iterator
     typename P> // P models From-Permutation on I
I rotate_indexed_helper(I f, I m, I l, P p)
{
    DISTANCE_TYPE(I) d = euclidean_gcd(m - f, l - m);
    DISTANCE_TYPE(I) i = 0;
    while (i < d) {
        do_cycle_from(f + i, p);
        ++i;
    }
    return f + (l - m);
};
```

This algorithm was first published by:

William Fletcher and Roland Silver.
Algorithm 284: Interchange of Two Blocks of Data.
CACM, Volume 9, Number 5, May 1966, page 326.

rotate_indexed and rotate_random_access

```
template <typename I> // I models Mutable Indexed Iterator
I rotate(I f, I m, I l, indexed_iterator_tag)
{
    k_rotate_from_permutation_indexed<I> p(f, m, l);
    return rotate_indexed_helper(f, m, l, p);
};

template <typename I> // I models Mutable Random Access Iterator
I rotate(I f, I m, I l, random_access_iterator_tag)
{
    k_rotate_from_permutation_random_access<I> p(f, m, l);
    return rotate_indexed_helper(f, m, l, p);
};

template <typename I> // I models Mutable Forward Iterator
I rotate(I f, I m, I l)
{
    if (m == f) return l;
    if (m == l) return f;
    return rotate(f, m, l, ITERATOR_CATEGORY(I)());
}
```

Complexity of `rotate` for indexed iterators

- The number of assignments is $n + c - t = n + \gcd(n, k)$
- On average, gcd $\ll n$

Loop fusion

- Assuming uniform distribution of rotation points, the expected value of $k = n/2$
- That gives very bad *locality of reference*
- Since the cycles are disjoint, we can improve locality of reference by operating on adjacent elements in different cycles
- This technique is called *loop fusion*, and was first discussed in this paper:

A.P. Yershov.
ALPHA–An Automatic Programming System of High Efficiency.
J. ACM, Volume 13, Number 1, January 1966, pages 17-24.

do_fused_cycles_from_with_buffer

```
template
    <typename I, // I models Mutable Forward Iterator
     typename P, // P models From-Permutation on I
     typename B> // B models Mutable Forward Iterator
     // UNDERLYING_TYPE(VALUE_TYPE(I)) == VALUE_TYPE(B)
void do_fused_cycles_from_with_buffer(I i, P p, B b, DISTANCE_TYPE(I) n)
{
    copy_n(i, b, n);
    I f = i;
    I next = p(i);
    while (next != i) {
        copy_n(next, f, n);
        f = n;
        n = p(n);
    }
    copy_n(b, f, n);
}
```

rotate_indexed_helper_fused

```
template
    <typename I, // I models Mutable Indexed Iterator
     typename P> // P models From-Permutation on I
I rotate_indexed_helper_fused(I f, I m, I l, P p)
{
    const int fusion_factor = 16;
    UNDERLYING_TYPE(VALUE_TYPE(I)) buffer[fusion_factor];
    DISTANCE_TYPE(I) d = gcd(m - f, l - m);
    DISTANCE_TYPE(I) i = 0;
    while (i + fusion_factor < d) {
        do_fused_cycles_from_with_buffer(f + i, p, buffer, fusion_factor);
        i = i + fusion_factor;
    }
    do_fused_cycles_from_with_buffer(f + i, p, buffer, d - i);
    return f + (l - m);
};
```

Issues with fused rotate

- Loop fusion is an important optimization technique that programmers need to know
- In this particular case, it is not that beneficial since often there are not too many loops to fuse
  - $\gcd(n, k) = 1$ with probability $\approx 60\%$
- There are algorithms that do more more assignments but have good locality of reference and work for weaker iterator requirements

Connection of rotate and reverse

Lemma

The k-rotation on $[0, n)$ is the only permutation p such that

1. $i < n - k \wedge n - k \leqslant j < n \Rightarrow p(j) < p(i)$
2. $i < j < n - k \vee n - k \leqslant i < j \Rightarrow p(i) < p(j)$

- The reverse permutation will satisfy condition 1, but not 2
- Applying reverse to subranges $[0, n - k)$ and $[n - k, n)$ and then applying reverse to the entire range will satisfy both conditions

rotate_three_reverses

- This insight gives us the following:

```
template <typename I> // I models Bidirectional Iterator
void rotate_three_reverses(I f, I m, I l)
{
    reverse(f, m);
    reverse(m, l);
    reverse(f, l);
}
```

- The only difficulty is finding the return value, m'

reverse_until

- The following auxiliary function allows us to find the return value without doing any extra work[4]

```
template <typename I> // I models Mutable Bidirectional Iterator
pair<I, I> reverse_until(I f, I m, I l)
{
    while (f != m && m != l) {
        --l;
        cycle_2(f, l);
        ++f;
    }
    return pair<I, I>(f, l);
}
```

[4]It was suggested to us by Raymond Ho and Wilson Lo

rotate for bidirectional iterator

```
template <typename I> // I models Mutable Bidirectional Iterator
I rotate(I f, I m, I l, bidirectional_iterator_tag)
{
    reverse(f, m);
    reverse(m, l);
    pair<I, I> p = reverse_until(f, m, l);
    reverse(p.first, p.second);
    if (m = p.first)
        return p.second;
    else
        return p.first;
}
```

Complexity of `rotate` for bidirectional iterator

Lemma

The number of assignments is $3(\lfloor n/2 \rfloor + \lfloor k/2 \rfloor + \lfloor (n-k)/2 \rfloor)$, which gives us $3n$ when n and k are both even and $3(n-2)$ in every other case

swap_ranges

```
template
    <typename I1, // I1 models Mutable Iterator
     typename I2> // I2 models Mutable Iterator
    // PARTIALLY_COMPATIBLE(VALUE_TYPE(I1), VALUE_TYPE(I2))
pair<I1, I2> swap_ranges(I1 f1, I1 l1, I2 f2, I2 l2)
{
    while (f1 != l1 && f2 != l2) {
        cycle_2(f1, f2);
        ++f1;
        ++f2;
    };
    return pair<I1, I2>(f1, f2);
}
```

- For $m \in [f, l)$ such that $l - m = m - f$, `swap_ranges(f, m, m, l)` rotates $[f, l)$ about m

Intuition for forward iterator rotate algorithm

$m = m'$	`swap_ranges` is sufficient
$m' < m$	after `swap_ranges`, $[f, m')$ are in the final position; we need to rotate $[m', l)$ around m
$m < m'$	after `swap_ranges`, $[f, m)$ are in the final position; we need to rotate $[m, l)$ around m'

This algorithm was first published by:

David Gries and Harlan Mills.
Swapping Sections.
Technical Report 81-452, Department of Computer Science, Cornell University, January 1981.

rotate_0

```
template <typename I> // I models Mutable Forward Iterator
void rotate_0(I f, I m, I l)
{
    I c = m;
    while (true) {
        pair<I, I> p = swap_ranges(f, m, m, l);
        if (p.first == m && p.second == l)
            return;
        if (p.first == m) {
            m = p.second;
            f = p.first;
        } else {
            assert(p.second == l);
            f = p.first;
        }
    }
}
```

Annotated `rotate_0`

```
template <typename I> // I models Mutable Forward Iterator
void rotate_0_annotated(I f, I m, I l)
{
                                                    DISTANCE_TYPE(I) u = m - f;
                                                    DISTANCE_TYPE(I) v = l - m;
    I c = m;
    while (true) {
        pair<I, I> p = swap_ranges(f, m, m, l);
        if (p.first == m && p.second == l)          assert(u == v);
            return;
        if (p.first == m) {                         assert(v > u);
            m = p.second;
            f = p.first;                                v = v - u;
        } else {
            assert(p.second == l);                  assert(u > v);
            f = p.first;                                u = u - v;
        }
    }
}
```

- u and v compute subtractive gcd of the initial lengths

Complexity of `rotate` for forward iterator

Lemma

The number of assignments is $3(n - \gcd(n, k))$

rotate_unguarded

- Inlining `swap_ranges` and small optimizations give:

```
template <typename I> // I models Mutable Forward Iterator
void rotate_unguarded(I f, I m, I l)
{
    assert(f != m && m != l);
    I c = m;
    while (c != l) {
        cycle_2(f, c);
        ++f;
        ++c;
        if (f == m)
            m = c;
        if (c == l)
            c = m;
    }
}
```

- The first time the second `if` clause is satisfied, `f` equals `m'`

rotate for forward iterator

```
template <typename I> // I models Mutable Forward Iterator
I rotate(I f, I m, I l, forward_iterator_tag)
{
    assert(f != m && m != l);
    I m_prime = l;
    I c = m;
    while (c != l) {
        cycle_2(f, c);
        ++f;
        ++c;
        if (f == m)
            m = c;
        if (c == l) {
            if (m_prime == l) m_prime = f;
            c = m;
        }
   }
   return m_prime;
}
```

- The guards in the category dispatch `rotate` guarantee the assertion

Conclusions

- Rotation rearrangements allow interchanging the order of adjacent ranges of different sizes
- Applying elementary number theory allows us to determine the cycle structure of rotations
- Interesting algorithms exist for forward, bidirectional, and indexed iterator algorithms
- Complexity is often dominated by locality of reference concerns rather than minimizing operation counts

Reading

Donald Knuth.
Section 1.3.3: Permutations and Factorials.
The Art of Computer Programming, Volume 1, Addison-Wesley, 1997, pages 164-185.
In particular, see problems 34 and 35.

Project 1: industrial-strength `rotate`

- Theoretically `rotate` for indexed iterators is much faster than the other two algorithms because it performs approximately $\frac{1}{3}$ as many assignments, but its locality of reference is poor
- Design a benchmark comparing performance of all three algorithms for different array sizes, element sizes, and rotation amounts
- Based on the results of the benchmark, design a composite algorithm that appropriately uses one of the three algorithms depending on its inputs

Project 2: Research in position-based rearrangements

- We have presented two kinds of position-based rearrangement algorithms: `reverse` and `rotate`
  - Later we will present another example, `random_shuffle`
- There are, however, many other examples of such algorithms; you can find some of them in the section of Knuth given in Reading as well as in the *Collected Algorithms of the ACM* (see, for example, numbers 302, 380, and 467)
- Develop a taxonomy of position-based rearrangements, discover additional algorithms, and produce a library

Contents I

Contents II

Contents III

Disclaimer

- This chapter contains only code
- We are posting it so that the desperate reader can find out where the story is going

find_if and find_if_not

```
template
    <typename I,  // I models Readable Iterator
     typename P>  // P models Unary Predicate on VALUE_TYPE(I)
I find_if(I f, I l, P p)
{
    while (f != l && !p(source(f)))
        ++f;
    return f;
}

template
    <typename I,  // I models Readable Iterator
     typename P>  // P models Unary Predicate on VALUE_TYPE(I)
I find_if_not(I f, I l, P p)
{
    while (f != l && p(source(f)))
        ++f;
    return f;
}
```

is_partitioned

```
template
    <typename I,   // I models Readable Iterator
     typename P>   // P models Unary Predicate on VALUE_TYPE(I)
bool is_partitioned(I f, I l, P p)
{
    return l == find_if_not(find_if(f, l, p), l, p);
}
```

- If we know the partition point m we can use:

```
template
    <typename I,   // I models Readable Iterator
     typename P>   // P models Unary Predicate on VALUE_TYPE(I)
bool is_partitioned(I f, I m, I l, P p)
{
    return m == find_if(f, m, p) && l == find_if_not(m, l, p);
}
```

partition_point_n

```
template
    <typename I,   // I models Readable Forward Iterator
     typename P>   // P models Unary Predicate on VALUE_TYPE(I)
I partition_point_n(I f, DISTANCE_TYPE(I) n, P p)
{
    while (n != 0) {
        DISTANCE_TYPE(I) h = n / 2;
        I m = f + h;
        if (!p(source(m))) {
            n = n - h - 1; f = m + 1;
        } else
            n = h;
    }
    return f;
}

template
    <typename I,   // I models Readable Forward Iterator
     typename P>   // P models Unary Predicate on VALUE_TYPE(I)
I partition_point(I f, I l, P p)
{
    return partition_point_n(f, l - f, p);
}
```

Intuition for partition for forward iterator

- ***** TO BE SUPPLIED *****
- The algorithm is due to Nico Lomuto, cited in:

  Jon Bentley.
  Programming Pearls.
  CACM, Volume 27, Number 4, April 1984, pages 287-291.

partition_forward_0

```
template
    <typename I,    // I models Mutable Forward Iterator
     typename P>    // P models Unary Predicate on VALUE_TYPE(I)
I partition_forward_0(I f, I l, P p)
{
    I m = f;
    I c = f;
    while (c != l) {
        assert(none(f, m, p) && all(m, c, p));
        if (!p(source(c))) {
            cycle_2(c, m);
            ++m;
        }
        ++c;
    }
    return m;
}
```

- Every false element at the beginning of the range is needlessly but harmlessly interchanged with itself

`partition` for forward iterator

```
template
    <typename I,   // I models Mutable Forward Iterator
     typename P>   // P models Unary Predicate on VALUE_TYPE(I)
I partition(I f, I l, P p, forward_iterator_tag)
{
    while (true) {
        if (f == l) return f;
        if (p(source(f))) break;
        ++f;
    }
    I c = successor(f);
    while (c != l) {
        if (!p(source(c))) {
            cycle_2(c, f);
            ++f;
        }
        ++c;
    }
    return f;
}
```

Complexity of `partition` for forward iterator

- xxx

partition for bidirectional iterator

```
template
    <typename I,   // I models Mutable Bidirectional Iterator
     typename P>   // P models Unary Predicate on VALUE_TYPE(I)
I partition(I f, I l, P p, bidirectional_iterator_tag)
{
    while (true) {
        while (true) {
            if (f == l) return f;
            if (p(source(f))) break;
            ++f;
        }
        while (true) {
            --l;
            if (f == l) return f;
            if (!p(source(f))) break;
        }
        cycle_2(f, l);
        ++f;
    }
}
```

partition for indexed iterator

```
template
    <typename I,   // I models Mutable Indexed Iterator
     typename P>   // P models Unary Predicate on VALUE_TYPE(I)
I partition(I f, I l, P p, indexed_iterator_tag)
{
    DISTANCE_TYPE(I) i = 0;
    DISTANCE_TYPE(I) j = l - f;
    while (true) {
        while (true) {
            if (i == j) return f + i;
            if (p(source(f + i))) break;
            i = i + 1;
        }
        while (true) {
            j = j - 1;
            if (i == j) return f + j;
            if (!p(source(f + j))) break;
        }
        cycle_2(f + i, f + j);
        i = i + 1;
    }
}
```

partition

```
template
    <typename I,   // I models Mutable Random Access Iterator
     typename P>   // P models Unary Predicate on VALUE_TYPE(I)
I partition(I f, I l, P p, random_access_iterator_tag)
{
    return partition(f, l, p, bidirectional_iterator_tag());
}
```

- We expect modern compilers generate equally good code if we called the indexed version in this case

```
template
    <typename I,   // I models Mutable Forward Iterator
     typename P>   // P models Unary Predicate on VALUE_TYPE(I)
I partition(I f, I l, P p)
{
    return partition(f, l, p, ITERATOR_CATEGORY(I)());
}
```

partition_copy_n

```
template
    <typename I,   // I models Readable Iterator
     typename O0,  // Of models Writable Iterator
     typename O1,  // Ot models Writable Iterator
     typename P>   // P models Unary Predicate on VALUE_TYPE(I)
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I), VALUE_TYPE(O0))
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I), VALUE_TYPE(O1))
pair<O0, O1> partition_copy_n(I f, DISTANCE_TYPE(I) n, O0 r0, O1 r1, P p)
{
    while (n != 0) {
        if (p(*f)) {
            sink(r1) = source(f); ++r1;
        } else {
            sink(r0) = source(f); ++r0;
        }
        ++f;
        n = n - 1;
    }
    return pair<O0, O1>(r0, r1);
}
```

- T.K. Lakshman suggested the interface

underlying_forward_iterator

```
template <typename I> // I models Iterator
struct underlying_forward_iterator
{
    typedef UNDERLYING_TYPE(VALUE_TYPE(I)) UT;
    typedef DISTANCE_TYPE(I) N;
    typedef underlying_forward_iterator UFI;
    I i;

    underlying_forward_iterator() {}
    underlying_forward_iterator(const I& x) : i(x) {}
    operator I() { return i; }
    void operator++() { ++i; }
    UFI operator+(N n) { return i + n; }
    friend N operator-(UFI x, UFI y) { return x.i - y.i; }
    friend bool operator==(const UFI& x, const UFI& y) { return x.i == y.i; }
    friend bool operator!=(const UFI& x, const UFI& y) { return !(x == y); }
    friend const UT& source(const UFI& x) { return underlying_source(x.i); }
    friend       UT& sink(UFI& x)         { return underlying_sink(x.i);   }
};
#define UFI(I) underlying_forward_iterator<I>
```

underlying_bidirectional_iterator

```
template <typename I> // I models Iterator
struct underlying_bidirectional_iterator
{
    typedef UNDERLYING_TYPE(VALUE_TYPE(I)) UT;
    typedef DISTANCE_TYPE(I) N;
    typedef underlying_bidirectional_iterator UBI;
    I i;

    underlying_bidirectional_iterator() {}
    underlying_bidirectional_iterator(const I& x) : i(x) {}
    operator I() { return i; }
    void operator++() { ++i; }
    void operator--() { --i; }
    UBI operator+(N n) { return i + n; }
    friend N operator-(UBI x, UBI y) { return x.i - y.i; }
    friend bool operator==(const UBI& x, const UBI& y) { return x.i == y.i; }
    friend bool operator!=(const UBI& x, const UBI& y) { return !(x == y); }
    friend const UT& source(const UBI& x) { return underlying_source(x.i); }
    friend       UT& sink(UBI& x)         { return underlying_sink(x.i);   }
};
#define UBI(I) underlying_bidirectional_iterator<I>
```

Type functions for underlying_forward_iterator

```
template <typename I> // I models Forward Iterator
struct value_type<underlying_forward_iterator<I> >
{
    typedef UNDERLYING_TYPE(VALUE_TYPE(I)) type;
};

template <typename I> // I models Forward Iterator
struct distance_type<underlying_forward_iterator<I> >
{
    typedef DISTANCE_TYPE(I) type;
};

template <typename I> // I models Forward Iterator
struct iterator_category<underlying_forward_iterator<I> >
{
    typedef forward_iterator_tag category;
};
```

stable_partition_n_with_buffer

```
template
    <typename I, // I models Mutable Forward Iterator
     typename B, // B models Mutable Forward Iterator
     typename P> // P models Unary Predicate on VALUE_TYPE(I)
     // UNDERLYING_TYPE(VALUE_TYPE(I)) == VALUE_TYPE(B)
pair<I, I> stable_partition_n_with_buffer(I f, DISTANCE_TYPE(I) n, B b, P p)
{
    pair<UFI(I), B> r = partition_copy_n(UFI(I)(f), n, UFI(I)(f), b, p);
    return pair<I, I>(I(r.first), I(copy(b, r.second, r.first)));
}
```

stable_partition_0 and stable_partition_1

```
template
    <typename I, // I models Mutable Forward Iterator
     typename P> // P models Unary Predicate on VALUE_TYPE(I)
pair<I, I> stable_partition_0(I f, P)
{
    return pair<I, I>(f, f);
}

template
    <typename I, // I models Mutable Forward Iterator
     typename P> // P models Unary Predicate on VALUE_TYPE(I)
pair<I, I> stable_partition_1(I f, P p)
{
    I l = successor(f);
    if (p(source(f)))
        return pair<I, I>(f, l);
    else
        return pair<I, I>(l, l);
}
```

stable_partition_merge

```
template <typename I> // I models Mutable Forward Iterator
pair<I, I> stable_partition_merge(pair<I, I> r0, pair<I, I> r1)
{
    I m = rotate(r0.first, r0.second, r1.first);
    return pair<I, I>(m, r1.second);
}
```

stable_partition_n

```
template
    <typename I, // I models Mutable Forward Iterator
     typename P> // P models Unary Predicate on VALUE_TYPE(I)
pair<I, I> stable_partition_n(I f, DISTANCE_TYPE(I) n, P p)
{
    if (n == 0)
        return stable_partition_0(f, p);
    if (n == 1)
        return stable_partition_1(f, p);
    pair<I, I> r0 = stable_partition_n(f, n / 2, p);
    pair<I, I> r1 = stable_partition_n(r0.second, n - n / 2, p);
    return stable_partition_merge(r0, r1);
}
```

stable_partition_n_adaptive

```
template
    <typename I, // I models Mutable Forward Iterator
     typename B, // B models Mutable Forward Iterator
     typename P> // P models Unary Predicate on VALUE_TYPE(I)
     // UNDERLYING_TYPE(VALUE_TYPE(I)) == VALUE_TYPE(B)
pair<I, I> stable_partition_n_adaptive(
    I f, DISTANCE_TYPE(I) n, B b, DISTANCE_TYPE(I) n_b, P p)
{
    if (n == 0)
        return stable_partition_0(f, p);
    if (n == 1)
        return stable_partition_1(f, p);
    if (n <= n_b)
        return stable_partition_n_with_buffer(f, n, b, p);
    pair<I, I> r0 = stable_partition_n_adaptive(f, n / 2, b, n_b, p);
    pair<I, I> r1 = stable_partition_n_adaptive(r0.second, n - n / 2, b, n_b, p);
    return stable_partition_merge(r0, r1);
}
```

find_out_of_order

```
template
    <typename I, // I models Readable Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
I find_out_of_order(I f, I l, R r)
{
    if (f == l) return l;
    VALUE_TYPE(I) v = source(f);
    ++f;
    while(f != l && !r(source(f), v)) {
        v = source(f);
        ++f;
    }
    return f;
}
```

is_sorted

```
template
    <typename I, // I models Readable Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
bool is_sorted(I f, I l, R r)
{
    return l == find_out_of_order(f, l, r);
}
```

lower_bound_n

```
template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
struct greater_than_or_equal_to_a
{
    const T& a;
    R r;
    greater_than_or_equal_to_a(const T& a, R r) : a(a), r(r) { }
    bool operator()(const T& x) { return !r(x, a); }
};

template
    <typename I, // I models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
I lower_bound_n(I f, DISTANCE_TYPE(I) n, const VALUE_TYPE(I)& a, R r)
{
    greater_than_or_equal_to_a<VALUE_TYPE(I), R> predicate(a, r);
    return partition_point_n(f, n, predicate);
}
```

upper_bound_n

```
template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
struct greater_than_a
{
    const T& a;
    R r;
    greater_than_a(const T& a, R r) : a(a), r(r) { }
    bool operator()(const T& x) { return r(a, x); }
};

template
    <typename I, // I models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
I upper_bound_n(I f, DISTANCE_TYPE(I) n, const VALUE_TYPE(I)& a, R r)
{
    greater_than_a<VALUE_TYPE(I), R> predicate(a, r);
    return partition_point_n(f, n, predicate);
}
```

binary_insertion_sort_n_0

```
template
    <typename I, // I models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
I binary_insertion_sort_n_0(I f, DISTANCE_TYPE(I) n, R r)
{
    if (n < 2) return f + n;
    I c = successor(f);
    DISTANCE_TYPE(I) i = 1;
    while (i != n) {
        I m = upper_bound_n(f, i, source(c), r);
        rotate(m, c, successor(c));
        ++c;
        i = i + 1;
    }
    return c;
}
```

rotate_last

```
template <typename I> // I models Mutable Forward Iterator
void rotate_last(I f, I last)
{
    rotate_last(f, last, ITERATOR_CATEGORY(I)());
}

template <typename I> // I models Mutable Forward Iterator
void rotate_last(I f, I last, forward_iterator_tag)
{
    while (f != last) {
        cycle_2(f, last);
        ++f;
    }
}

template <typename I> // I models Mutable Bidirectional Iterator
void rotate_last(I f, I last, bidirectional_iterator_tag)
{
    typedef UBI(I) U;
    UNDERLYING_TYPE(VALUE_TYPE(I)) tmp = underlying_source(last);
    copy_backward(U(f), U(last), U(successor(last)));
    underlying_sink(f) = tmp;
}
```

rotate_last, continued

```
template <typename I> // I models Mutable Indexed Iterator
void rotate_last(I f, I last, indexed_iterator_tag)
{
    UNDERLYING_TYPE(VALUE_TYPE(I)) tmp = source(last);
    copy_backward_n(f, last - f, UFI(I)(successor(f)));
    sink(f) = tmp;
}

template <typename I> // I models Mutable Random Access Iterator
void rotate_last(I f, I last, random_access_iterator_tag)
{
    rotate_last(f, last, bidirectional_iterator_tag());
}
```

binary_insertion_sort_n

```
template
    <typename I, // I models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
I binary_insertion_sort_n(I f, DISTANCE_TYPE(I) n, R r)
{
    if (n < 2) return f + n;
    I c = successor(f);
    DISTANCE_TYPE(I) i = 1;
    while (i != n) {
        I m = upper_bound_n(f, i, source(c), r);
        rotate_last(m, c);
        ++c;
        i = i + 1;
    }
    return c;
}
```

insertion_merge_n_0

- Precondition: $f1 = f0 + n0$ and $[f0, n0)$ and $[f1, n1)$ are sorted

```
template
    <typename I, // I models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
void insertion_merge_n_0(
    I f0, DISTANCE_TYPE(I) n0, I f1, DISTANCE_TYPE(I) n1, R r)
{
    DISTANCE_TYPE(I) i = 0;
    while (i != n1) {
        I m = upper_bound_n(f0, n0 + i, source(f1), r);
        if (m == f1) return;
        rotate_last(m, f1);
        ++f1;
        i = i + 1;
    }
}
```

insertion_merge_n

```
template
    <typename I, // I models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
void insertion_merge_n(
    I f0, DISTANCE_TYPE(I) n0, I f1, DISTANCE_TYPE(I) n1, R r)
{
    DISTANCE_TYPE(I) i = 0;
    while (i != n1 && n0 != 0) {
        I m = upper_bound_n(f0, n0, source(f1), r);
        if (m == f1) return;
        rotate_last(m, f1);
        n0 = n0 - (m - f0);
        f0 = successor(m);
        ++f1;
        i = i + 1;
    }
}
```

Intuition for in-place merge

- ***** TO BE SUPPLIED *****
- The algorithm is due to

Krzysztof Dudziński and Andrzej Dydek.
On a Stable Minimum Storage Merging Algorithm.
Information Processing Letters, Volume 12, Number 1, February 1981, pages 5-8.

merge_n

```
template
    <typename I, // I models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
void merge_n(I f0, DISTANCE_TYPE(I) n0, I f1, DISTANCE_TYPE(I) n1, R r)
{
    if (n0 + n1 < 16)
        return insertion_merge_n(f0, n0, f1, n1, r);
    I i, j;
    if (n0 > n1) {
        i = f0 + n0 / 2;
        j = lower_bound_n(f1, n1, source(i), r);
    } else {
        j = f1 + n1 / 2;
        i  = upper_bound_n(f0, n0, source(j), r);
    }
    I m = rotate(i, f1, j);
    merge_n(f0, i - f0, i, m - i, r);
    merge_n(m, j - m, j, n1 - (j - f1), r);
}
```

stable_sort_n

```
template
    <typename I, // I models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
I stable_sort_n(I f, DISTANCE_TYPE(I) n, R r)
{
    if (n < 16)
        return binary_insertion_sort_n(f, n, r);
    I m = stable_sort_n(f, n / 2, r);
    I l = stable_sort_n(m, n - n / 2, r);
    merge_n(f, n / 2, m, n - n / 2, r);
    return l;
}
```

merge_copy_n

```
template
    <typename I0, // I0 models Iterator
     typename I1, // I1 models Iterator
     typename O,  // O models Writable Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I0), VALUE_TYPE(O))
     // PARTIALLY_COMPATIBLE(VALUE_TYPE(I1), VALUE_TYPE(O))
triple<I0, I1, O> merge_copy_n(I0 f0, DISTANCE_TYPE(I0) n0,
                               I1 f1, DISTANCE_TYPE(I1) n1, O o, R r)
{
    while (n0 != 0 && n1 != 0) {
        if (r(source(f1), source(f0))) {
            sink(o) = source(f1); ++f1; n1 = n1 - 1;
        } else {
            sink(o) = source(f0); ++f0; n0 = n0 - 1;
        }
        ++o;
    }
    pair<I0, O> p0 = copy_n(f0, n0, o);
    pair<I1, O> p1 = copy_n(f1, n1, p0.second);
    return triple<I0, I1, O>(p0.first, p1.first, p1.second);
}
```

underlying_compare

```
template
    <typename T, // T models Regular
     typename R> // R models Relation on T
struct underlying_compare
{
    typedef UNDERLYING_TYPE(T) U;
    R r;
    underlying_compare(R r) : r(r) {}
    bool operator()(const U& x, const U& y)
    {
        return r(reinterpret_cast<const T&>(x), reinterpret_cast<const T&>(y));
    }
};
```

merge_n_with_buffer

```
template
    <typename I, // I models Mutable Forward Iterator
     typename B, // B models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
     // UNDERLYING_TYPE(VALUE_TYPE(I)) == VALUE_TYPE(B)
void merge_n_with_buffer(
    I f0, DISTANCE_TYPE(I) n0, I f1, DISTANCE_TYPE(I) n1, B b, R r)
{
    typedef underlying_compare<VALUE_TYPE(I), R> UR;
    copy_n(UFI(I)(f0), n0, b);
    merge_copy_n(b, n0, UFI(I)(f1), n1, UFI(I)(f0), UR(r));
}
```

merge_n_adaptive

```
template
    <typename I, // I models Mutable Forward Iterator
     typename B, // B models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
     // UNDERLYING_TYPE(VALUE_TYPE(I) == VALUE_TYPE(B)
void merge_n_adaptive(I f0, DISTANCE_TYPE(I) n0, I f1, DISTANCE_TYPE(I) n1,
                      B b, DISTANCE_TYPE(I) n_b, R r)
{
    if (n0 + n1 < 8) // ***** MEASURE AND TUNE
        return insertion_merge_n(f0, n0, f1, n1, r);
    if (n0 <= n_b)
        return merge_n_with_buffer(f0, n0, f1, n1, b, r);
    I i, j;
    if (n0 > n1) {
        i = f0 + n0 / 2;
        j = lower_bound_n(f1, n1, source(i), r);
    } else {
        j = f1 + n1 / 2;
        i = upper_bound_n(f0, n0, source(j), r);
    }
    I m = rotate(i, f1, j);
    merge_n_adaptive(f0, i - f0, i, m - i, b, n_b, r);
    merge_n_adaptive(m, j - m, j, n1 - (j - f1), b, n_b, r);
}
```

stable_sort_n_adaptive

```
template
    <typename I, // I models Mutable Forward Iterator
     typename B, // B models Mutable Forward Iterator
     typename R> // R models Weak Ordering on VALUE_TYPE(I)
     // UNDERLYING_TYPE(VALUE_TYPE(I)) == VALUE_TYPE(B)
I stable_sort_n_adaptive(
    I f, DISTANCE_TYPE(I) n, B b, DISTANCE_TYPE(I) n_b, R r)
{
    if (n < 16)
        return binary_insertion_sort_n(f, n, r);
    I m = stable_sort_n_adaptive(f, n / 2, b, n_b, r);
    I l = stable_sort_n_adaptive(m, n - n / 2, b, n_b, r);
    merge_n_adaptive(f, n / 2, m, n - n / 2, b, n_b, r);
    return l;
}
```

Conclusions

Reading

Project

Contents I

Contents II

Disclaimer

- This chapter contains only code
- We are posting it so that the desperate reader can find out where the story is going

TO DO

Describe node iterators; explain they always allow set_successor, but these algorithms do not require mutability (need not have sink)

reverse_append

```
template <typename I> // I models Node Iterator
I  reverse_append(I f, I l, I r)
{
    while (f != l) {
        I tmp = successor(f);
        set_successor(f, r);
        r = f;
        f = tmp;
    }
    return r;
}
```

reverse_nodes

```
template <typename I> // I models Node Iterator
I reverse_nodes(I f, I l)
{
    return reverse_append(f, l, l);
}
```

partition_nodes signature

```
template
    <typename I, // I models Node Iterator
     typename P> // P models Unary Predicate on VALUE_TYPE(I)
pair<pair<I, I>, pair<I, I> > partition_nodes(I f, I l, P p)
```

partition_nodes body

```
{
entry: I h0 = l; I t0 = l; I h1 = l; I t1 = l;
    if (f == l)        {                        goto exit; }
    if (p(source(f))) { h1 = f;                 goto s1;   }
    else               { h0 = f;                goto s0;   }
s0:    t0 = f; ++f;
    if (f == l)        {                        goto exit; }
    if (p(source(f))) { h1 = f;                 goto s3;   }
    else               {                        goto s0;   }
s1:    t1 = f; ++f;
    if (f == l)        {                        goto exit; }
    if (p(source(f))) {                         goto s1;   }
    else               { h0 = f;                goto s2;   }
s2:    t0 = f; ++f;
    if (f == l)        {                        goto exit; }
    if (p(source(f))) { set_successor(t1, f); goto s3;   }
    else               {                        goto s2;   }
s3:    t1 = f; ++f;
    if (f == l)        {                        goto exit; }
    if (p(source(f))) {                         goto s3;   }
    else               { set_successor(t0, f); goto s2;   }
exit:  return pair<pair<I, I>, pair<I, I> >
                 (pair<I, I>(h0, t0), pair<I, I>(h1, t1));
}
```

merge_nodes_non_empty

```
template
    <typename I, // I models Node Iterator
     typename R> // R models Weak Order on VALUE_TYPE(I)
triple<I, I, I> merge_nodes_non_empty(I f0, I l0, I f1, I l1, R r)
{
    typedef triple<I, I, I> RT;
entry: I h, t;
    if (r(source(f1), source(f0)))
                   { h = f1;                goto s1;              }
    else           { h = f0;                goto s0;              }
s0:    t = f0; ++f0;
    if (f0 == l0)  { set_successor(t, f1); return RT(h, t, l1); }
    if (r(source(f1), source(f0)))
                   { set_successor(t, f1); goto s1;              }
    else           {                        goto s0;              }
s1:    t = f1; ++f1;
    if (f1 == l1)  { set_successor(t, f0); return RT(h, t, l0); }
    if (r(source(f1), source(f0)))
                   {                        goto s1;              }
    else           { set_successor(t, f0); goto s0;              }
}
```

reduce_non_empty

```
template
    <typename I,  // I models Readable Iterator
     typename Op> // Op models Semigroup Operation on VALUE_TYPE(I)
VALUE_TYPE(I) reduce_non_empty(I f, I l, Op op)
{
    VALUE_TYPE(I) r = source(f);
    ++f;
    while (f != l) {
        r = op(r, source(f));
        ++f;
    }
    return r;
}
```

reduce

```
template
    <typename I,  // I models Readable Iterator
     typename Op> // Op models Monoid Operation on VALUE_TYPE(I)
VALUE_TYPE(I) reduce(I f, I l, Op op, VALUE_TYPE(I) z)
{
    if (f == l) return z;
    return reduce_non_empty(f, l, op);
}
```

reduce_nonzeroes

```
template <typename I>  // I models Readable Iterator
I find_not(I f, I l, VALUE_TYPE(I) x)
{
    while (f != l && source(f) == x) ++f;
    return f;
}

template
    <typename I,   // I models Readable Iterator
     typename Op> // Op models Monoid Operation on VALUE_TYPE(I)
VALUE_TYPE(I) reduce_nonzeroes(I f, I l, Op op, VALUE_TYPE(I) z)
{
    f = find_not(f, l, z);
    if (f == l) return z;
    VALUE_TYPE(I) r = source(f);
    ++f;
    while (f != l) {
        if (source(f) != z) r = op(r, source(f));
        ++f;
    }
    return r;
}
```

add_to_counter

```
template
    <typename I,   // I models Mutable Forward Iterator
     typename Op>  // Op models Binary Operation on VALUE_TYPE(I)
VALUE_TYPE(I) add_to_counter(I f, I l, Op op, VALUE_TYPE(I) x, VALUE_TYPE(I) z)
{
    if (x == z) return z;
    while (f != l) {
        if (source(f) != z) {
            x = op(source(f), x);
            sink(f) = z;
        } else {
            sink(f) = x;
            return z;
        }
        ++f;
    }
    return x;
}
```

reduce_transposed_nonzeroes

```
template
    <typename I,  // I models Readable Forward Iterator
     typename Op> // Op models Monoid Operation on VALUE_TYPE(I)
VALUE_TYPE(I) reduce_transposed_nonzeroes(
    I f, I l, Op op, VALUE_TYPE(I) z)
{
    f = find_not(f, l, z);
    if (f == l) return z;
    VALUE_TYPE(I) r = source(f);
    ++f;
    while (f != l) {
        if (source(f) != z) r = op(source(f), r);
        ++f;
    }
    return r;
}
```

reduce_balanced

```
template
    <int k,         // $2^k > l - f$
     typename I,  // I models Readable Forward Iterator
     typename Op> // Op models Monoid Operation on VALUE_TYPE(I)
void reduce_balanced(I f, I l, Op op, VALUE_TYPE(I) z)
{
    VALUE_TYPE(I) counter[k];
    I* c_f = counter;
    I* c_l = counter;
    while (f != l) {
        VALUE_TYPE(I) carry = add_to_counter(c_f, c_l, op, source(f), z);
        if (carry != z) {
            sink(c_l) = carry;
            ++c_l;
        };
        ++f;
    }
    return reduce_transposed_nonzeros(c_f, c_l, op, z);
}
```

merger

```
template
    <typename I, // I models Node Iterator
     typename R> // R models Weak Order on VALUE_TYPE(I)
struct merger
{
    I l;
    R r;
    merger(I l, R r) : l(l), r(r) {}
    I operator()(I x, I y)
    {
        return merge_nodes_non_empty(x, l, y, l, r).first;
    }
};
```

sort_nodes

```
template
    <int k,        // $2^k > l - f$
     typename I, // I models Node Iterator
     typename R> // R models Weak Order on VALUE_TYPE(I)
I sort_nodes(I f, I l, R r)
{
    merger<I, R> merge_op(l, r);
    I counter[k];
    I* c_f = counter;
    I* c_l = counter;
    while (f != l) {
        I old_f = f;
        ++f;
        set_successor(old_f, l);
        I carry = add_to_counter(c_f, c_l, merge_op, old_f, l);
        if (carry != l) {
            sink(c_l) = carry;
            ++c_l;
        }
    }
    return reduce_transposed_nonzeroes(c_f, c_l, merge_op, l);
}
```

Contents I

Contents II

Contents III

Disclaimer

- This chapter contains only code
- We are posting it so that the desperate reader can find out where the story is going

Allocation and deallocation

```
void* raw_allocate(size_t n)
{
    return ::operator new(n);
}

template <typename T> // T models Regular
T* allocate(size_t n)
{
    return (T*)raw_allocate(n * sizeof(T));
}

void deallocate(void* p)
{
    ::operator delete(p);
}
```

construct_copy and destroy

```
template <typename T> // T models Regular
struct construct_copy
{
    const T* p;
    construct_copy(const T& x) : p(&x) {}
    void operator()(T& q)
    {
        new (&q) T(source(p));
    }
};

template <typename T> // T models Regular
void destroy(T& p)
{
    sink(&p).~T();
}
```

list_node and list_iterator

```
template <typename T> // T models Regular
struct list_node
{
    T value;
    list_node* successor;
    list_node(const T& v, list_node* p) : value(v), successor(p) {}
};

template <typename T> // T models Regular
struct list_iterator
{
    list_node<T>* p;
    list_iterator() : p(0) {}
    list_iterator(list_node<T>* p) : p(p) {}
};
```

list_iterator functions

```
template <typename T>
void operator++(list_iterator<T>& x) { x.p = source(x.p).successor; }

template <typename T>
void set_successor(list_iterator<T> x, list_iterator<T> y)
{ sink(x.p).successor = y.p; }

template <typename T>
bool operator==(list_iterator<T> x, list_iterator<T> y)
{ return x.p == y.p; }

template <typename T>
const T& source(list_iterator<T> x) { return source(x.p).value; }

template <typename T>
T& sink(list_iterator<T> x) { return sink(x.p).value; }
```

list_iterator type functions

```
template <typename T>
struct value_type< list_iterator<T> >
{
    typedef T type;
};

template <typename T>
struct distance_type< list_iterator<T> >
{
    typedef DISTANCE_TYPE(list_node<T>*) type;
};

template <typename T>
struct iterator_category< list_iterator<T> >
{
    typedef forward_iterator_tag category;
};
```

List node insert functions

```
template
    <typename T, // T models Regular
     typename F> // F models Constructor Object for T
list_iterator<T> insert_construct(list_iterator<T> i, F f)
{
    list_iterator<T> j(allocate< list_node<T> >(1));
    f(sink(j));
    set_successor(j, i);
    return j;
}

template <typename T> // T models Regular
list_iterator<T> insert(list_iterator<T> i, const T& x)
{
    return insert_construct(i, construct_copy<T>(x));
}
```

More list node insert functions

```
template
    <typename T, // T models Regular
     typename F> // F models Constructor Object for T
list_iterator<T> insert_after_construct(list_iterator<T> i, F f)
{
    list_iterator<T> j = insert_construct(successor(i), f);
    set_successor(i, j);
    return j;
}

template <typename T> // T models Regular
list_iterator<T> insert_after(list_iterator<T> i, const T& x)
{
    return insert_after_construct(i, construct_copy<T>(x));
}
```

list_insert_after_iterator

```
template <typename T> // T models Regular
struct list_insert_after_iterator
{
    list_iterator<T> h;
    list_iterator<T> t;
    list_insert_after_iterator() {}
    list_insert_after_iterator(list_iterator<T> h, list_iterator<T> t)
        : h(h), t(t) {}
};
```

(list_insert_after_iterator equality operator)

```
template <typename T> // T models Regular
bool operator==(const list_insert_after_iterator<T>& x,
                const list_insert_after_iterator<T>&y)
{
    return x.h == y.h && x.t == y.t;
}
```

list_insert_after_iterator functions

```
template <typename T> // T models Regular
void operator++(list_insert_after_iterator<T>& x) {}

template <typename T> // T models Regular
T& sink(list_insert_after_iterator<T>& x)
{
    if (x.t == list_iterator<T>()) {
        x.h = insert(x.h, T());
        x.t = x.h;
    } else
        x.t = insert_after(x.t, T());
    return sink(x.t);
}
```

List node erase functions

```
template <typename T> // T models Regular
list_iterator<T> erase_first(list_iterator<T> i)
{
    list_iterator<T> j = successor(i);
    destroy(sink(i));
    deallocate(i.p);
    return j;
}

template <typename T> // T models Regular
void erase_after(list_iterator<T> i)
{
    set_successor(i, erase_first(successor(i)));
}
```

list data structure

```
template <typename T> // T models Regular
struct list
{
    list_iterator<T> first;

    list() : first(0) {}

    template <typename I> // I models Readable Iterator to T
    list(I f, I l);

    list(const list& s);

    ~list();

    void operator=(list s) { swap(sink(this), s); }
};
```

list type functions

```
template <typename T> // T models Regular
struct underlying_type< list<T> >
{
    typedef list_iterator<T> type; // or ITERATOR(list<T>)
};

template <typename S> // S models Data Structure
struct iterator_type
{
    typedef S type;
};
#define ITERATOR(S) typename iterator_type<S>::type

template <typename T> // T models Regular
struct iterator_type< list<T> >
{
    typedef list_iterator<T> type;
};
```

list functions

```
template <typename T> // T models Regular
ITERATOR(list<T>) begin(const list<T>& x)
{
    return x.first;
}

template <typename T> // T models Regular
ITERATOR(list<T>) end(const list<T>& x)
{
    return list_iterator<T>();
}

template <typename S> // S models Data Structure
DISTANCE_TYPE(ITERATOR(S)) size(const S& x)
{
    return end(x) - begin(x);
}

template <typename S> // S models Data Structure
bool is_empty(const S& x)
{
    return begin(x) == end(x);
}
```

list construction

```
template
    <typename T, // T models Regular
     typename I> // I models Iterator on T
void insert(list<T>& s, ITERATOR(list<T>) i, I f, I l)
{
    s.first = copy(f, l, list_insert_after_iterator<T>(begin(s), i)).h;
}

template <typename T> // T models Regular
template <typename I> // I models Readable Iterator to T
list<T>::list(I f, I l) : first(0)
{
    insert(sink(this), end(sink(this)), f, l);
}

template <typename T> // T models Regular
list<T>::list(const list& s) : first(0)
{
    insert(sink(this), end(sink(this)), begin(s), end(s));
}
```

list destruction

```
template <typename T> // T models Regular
void erase_all(list<T>& s)
{
    while (!is_empty(s))
       s.first = erase_first(begin(s));
}

template <typename T> // T models Regular
list<T>::~list()
{
    erase_all(sink(this));
}
```

mismatch

```
template
    <typename I0, // I0 models Readable Iterator
     typename I1, // I1 models Readable Iterator
     typename R>  // R models Relation on VALUE(I0) and VALUE(I1)
pair<I0, I1> mismatch(I0 f0, I0 l0, I1 f1, I1 l1)
{
    while (f0 != l0 && f1 != l1 && r(source(f0), source(f1))) {
        ++f0; ++f1;
    }
    return pair<I0, I1>(f0, f1);
}
```

lexicographic_equal

```
template
    <typename I0, // I0 models Readable Iterator
     typename I1> // I1 models Readable Iterator
     // VALUE_TYPE(I0)== VALUE_TYPE(I1)
bool lexicographic_equal(I0 f0, I0 l0, I1 f1, I1 l1)
{
    while (true) {
        if (f0 == l0 && f1 == l1) return true;
        if (f0 == l0 || f1 == l1) return false;
        if (source(f0) != source(f1)) return false;
        ++f0; ++f1;
    }
}
```

lexicographic_less

```
template
    <typename I0, // I0 models Readable Iterator
     typename I1> // I1 models Readable Iterator
     // VALUE_TYPE(I0)== VALUE_TYPE(I1)
bool lexicographic_less(I0 f0, I0 l0, I1 f1, I1 l1)
{
    while (true) {
        if (f1 == l1) return false;
        if (f0 == l0) return true;
        if (source(f0) < source(f1)) return true;
        if (source(f0) > source(f1)) return false;
        ++f0; ++f1;
    }
}
```

Exercise

Improve the code by skipping initial equal elements, e.g., with mismatch

list == and <

```
template <typename T> // T models Regular
bool operator==(const list<T>& x, const list<T>& y)
{
    return lexicographic_equal(begin(x), end(x), begin(y), end(y));
}

template <typename T> // T models Regular
bool operator<(const list<T>& x, const list<T>& y)
{
    return lexicographic_less(begin(x), end(x), begin(y), end(y));
}
```

- These definitions work for any data structure

list partition

```
template
    <typename T, // T models Regular
     typename P> // P models Unary Predicate on T
void partition_list(list<T>& x, list<T>& y, P p)
{
    typedef ITERATOR(list<T>) I;
    pair< pair<I, I>, pair<I, I> > pp = partition_nodes(begin(x), end(x), p);
    x.first = pp.first.first;
    if (pp.second.first != end(x)) {
        set_successor(pp.second.second, begin(y));
        y.first = pp.second.first;
    }
}
```

list merge

```
template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
void merge(list<T>& x, list<T>& y, R r)
{
    if (is_empty(y)) return;
    if (is_empty(x)) x = y;
    else
        x.first = merge_nodes_non_empty(
                      begin(x), end(x), begin(y), end(y), r).first;
    y.first = end(y);
}
```

list sort

```
template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
void sort(list<T>& x, R r)
{
    x.first = sort_nodes<32>(begin(x), end(x), r);
}
```

array_header

```
template <typename T> // T models Regular
struct array_header
{
    T* m;
    T* l;
    T  a;
};
```

- Invariants, where $f = \&a$:
  - $[f, m)$ are constructed elements
  - $[m, l)$ are unconstructed (reserved) elements

allocate_array_header

```
template <typename T> // T models Regular
array_header<T>* allocate_array_header(DISTANCE_TYPE(T*) n)
{
    typedef array_header<T>* P;
    size_t size = sizeof(array_header<T>) + size_t(n - 1) * sizeof(T);
    P p(P(raw_allocate(size)));
    T* f = &sink(p).a;
    sink(p).m = f;
    sink(p).l = f + n;
    return p;
}
```

deallocate_array_header

```
template <typename T> // T models Regular
void deallocate_array_header(array_header<T>* p)
{
    deallocate(p);
}
```

array data structure

```
template <typename T> // T models Regular
struct array
{
    array_header<T>* p;

    array() : p(0) {}

    template <typename I> // I models Readable Iterator to T
    array(I f, I l);

    array(const array& x);

    ~array();

    void operator=(array x) { swap(sink(this), x); }
};
```

array type functions

```
template <typename T> // T models Regular
struct underlying_type< array<T> >
{
    typedef struct { array_header<T>* p; } type;
};

template <typename T> // T models Regular
struct iterator_type< array<T> >
{
    typedef T* type;
};
```

array functions

```
template <typename T> // T models Regular
ITERATOR(array<T>) begin(const array<T>& a)
{
    if (a.p == 0) return ITERATOR(array<T>)(0);
    return ITERATOR(array<T>)(&source(a.p).a);
}

template <typename T> // T models Regular
ITERATOR(array<T>) end(const array<T>& a)
{
    if (a.p == 0) return ITERATOR(array<T>)(0);
    return ITERATOR(array<T>)(source(a.p).m);
}

template <typename T> // T models Regular
ITERATOR(array<T>) end_of_storage(const array<T>& a)
{
    if (a.p == 0) return ITERATOR(array<T>)(0);
    return ITERATOR(array<T>)(source(a.p).l);
}
```

More array functions

```
template <typename T> // T models Regular
DISTANCE_TYPE(T*) size(const array<T>& a)
{
    return end(a) - begin(a);
}

template <typename T> // T models Regular
bool is_empty(const array<T>& a)
{
    return begin(a) == end(a);
}

template <typename T> // T models Regular
DISTANCE_TYPE(T*) capacity(const array<T>& a)
{
    return end_of_storage(a) - begin(a);
}

template <typename T> // T models Regular
bool is_full(const array<T>& a)
{
    return end(a) == end_of_storage(a);
}
```

reserve

```
template <typename T> // T models Regular
void reserve(array<T>& a, DISTANCE_TYPE(ITERATOR(array<T>)) n)
{
    if (n < size(a) || n == capacity(a)) return;
    typedef UFI(T*) U;
    array_header<T>* p = allocate_array_header<T>(n);
    T* f = &sink(p).a;
    copy(U(begin(a)), U(end(a)), U(f)); // never throws
    sink(p).m = f + size(a);
    deallocate_array_header(a.p);
    a.p = p;
}
```

push_back_construct

```
template <typename I> // I models Integer
I array_growth_function(I n)
{
    const I initial_size(1);
    return max(initial_size, I(2) * n);
}

template
    <typename T, // T models Regular
     typename F> // F models Constructor Object to T
void push_back_construct(array<T>& a, F f)
{
    if (size(a) == capacity(a))
        reserve(a, array_growth_function(size(a)));
    f(sink(source(a.p).m));
    ++sink(a.p).m;
}
```

push_back

```
template <typename T> // T models Regular
void push_back(array<T>& a, const T& x)
{
    push_back_construct(a, construct_copy<T>(x));
}
```

array_insert_iterator

```
template <typename T> // T models Regular
struct array_insert_iterator
{
    array<T>* p;
    array_insert_iterator() {}
    array_insert_iterator(array<T>& a) : p(&a) {}
};
```

(array_insert_iterator equality)

```
template <typename T> // T models Regular
bool operator==(array_insert_iterator<T> x,
                array_insert_iterator<T> y)
{
    return x.p == y.p;
}
```

array_insert_iterator functions

```
template <typename T> // T models Regular
void operator++(array_insert_iterator<T>& x) {}

template <typename T> // T models Regular
T& sink(array_insert_iterator<T>& x)
{
    push_back(sink(x.p), T());
    return sink(end(source(x.p)) - 1);
}

template
    <typename T, // T models Regular
     typename I> // I models Iterator on T
void append(array<T>& a, I f, I l)
{
    copy(f, l, array_insert_iterator<T>(a));
}
```

array erase functions

```
template <typename T> // T models Regular
void erase_back(array<T>& a)
{
    --sink(a.p).m;
    destroy(source(a.p).m);
}

template <typename T> // T models Regular
void erase_all(array<T>& a)
{
    while (!is_empty(a))
       erase_back(a);
    deallocate_array_header(a.p);
    a.p = 0;
}
```

array member functions

```
template <typename T> // T models Regular
template <typename I> // I models Readable Iterator to T
array<T>::array(I f, I l) : p(0) { append<T, I>(sink(this), f, l); }

template <typename T> // T models Regular
array<T>::array(const array& x) : p(0)
{
    reserve(sink(this), size(x));
    append<T>(sink(this), begin(x), end(x));
}

template <typename T> // T models Regular
array<T>::~array() { erase_all(sink(this)); }
```

array == and <

```
template <typename T>
bool operator==(const array<T>& x, const array<T>& y)
{
    return lexicographic_equal(begin(x), end(x), begin(y), end(y));
}

template <typename T>
bool operator<(const array<T>& x, const array<T>& y)
{
    return lexicographic_less(begin(x), end(x), begin(y), end(y));
}
```

fill_iterator

```
template
    <typename T, // T models Regular
     typename N> // N models Integer
struct fill_iterator
{
    T v;
    N n;
    fill_iterator() : n(0) {}
    fill_iterator(N n) : n(n) {}
    fill_iterator(const T& v, N n) : v(v), n(n) {}
};
```

fill_iterator functions

```
template
    <typename T, // T models Regular
     typename N> // N models Integer
void operator++(fill_iterator<T, N>& x) { ++x.n; }

template
    <typename T, // T models Regular
     typename N> // N models Integer
const T& source(fill_iterator<T, N> x) { return x.v; }
```

More fill_iterator functions

```
template
    <typename T, // T models Regular
     typename N> // N models Integer
void operator+=(fill_iterator<T, N>& x, N n) { x.n += n; }

template
    <typename T, // T models Regular
     typename N> // N models Integer
N operator-(const fill_iterator<T, N>& x, const fill_iterator<T, N>& y)
{
    return x.n - y.n;
}

template
    <typename T, // T models Regular
     typename N> // N models Integer
bool operator==(fill_iterator<T, N> x, fill_iterator<T, N> y)
{
    return x.n == y.n;
}
```

fill_iterator type functions

```
template
    <typename T, // T models Regular
     typename N> // N models Integer
struct value_type< fill_iterator<T, N> >
{
    typedef T type;
};

template
    <typename T, // T models Regular
     typename N> // N models Integer
struct distance_type< fill_iterator<T, N> >
{
    typedef N type;
};

template
    <typename T, // T models Regular
     typename N> // N models Integer
struct iterator_category< fill_iterator<T, N> >
{
    typedef indexed_iterator_tag category;
};
```

array sort

```
template
    <typename T, // T models Regular
     typename R> // R models Weak Ordering on T
void sort(array<T>& x, R r)
{
    typedef DISTANCE_TYPE(ITERATOR(array<T>)) N;
    typedef UNDERLYING_TYPE(T) U;
    typedef fill_iterator<U, N> F;
    N n = max(size(x) / N(10), N(100));
    array<U> buffer(F(0), F(n));
    stable_sort_n_adaptive(begin(x), size(x), begin(buffer), n, r);
}
```

Acknowledgments

- We are grateful for comments and suggestions from:
  - Andrei Alexandrescu
  - David Musser
  - Dave Parent
  - Dmitry Polukhin
  - Mark Ruzon
  - Geoff Scott
  - Walter Vannini
  - Oleg Zabluda

Index I

Index II

Index III

Index IV

Index V

Index VI

Index VII

Index VIII

Index IX

Index X

Index XI