Notes on the Foundations of Programming
Volume II

Alexander A. Stepanov
Matthew A. Marcus

Draft of April 6, 2005

What thou lovest well remains,
the rest is dross
—Ezra Pound, *Cantos*

Contents

Part 1

Basic Algorithms

CHAPTER 1

The Machine Model

1.1. The Significance of Memory

In the first part of the course we spent a lot of time trying to convince you that computer science is fundamentally mathematics. But computer scientists are not just rehashing old material. The primary contribution of computer science to mathematics is the notion of *memory*. The people who invented the notion of memory were mathematicians. A computer with memory is often referred to as a Von Neumann computer. We would be hard pressed to find a programmer nowadays who is not familiar with the notion of memory. In this part of the course is we will figure out what memory really is. From this we will obtain iterators and different kinds of memories. We will begin by discussing the history of the notion of memory in computer science. Some people in computer science, e.g. functional programmers, would like to get rid of all state – we will demonstrate why this idea is wrong.

The notions of address and pointer took some time to develop. Memories used to be very small – machines had no pointers at all. Originally, on many machines, the instructions were structured with an op-code followed by several fields which would be: address1, address2, address3. (Sometimes there was even address4. When present, the fourth address was used for branching. Some machines had drum memory. While one instruction was being executed the drum would rotate past the next instruction. Clever people would optimize their code so that the next instruction would be located exactly on the right part of the drum. The fourth address would contain its address. The fourth address also eliminated goto.) Using three addresses allowed instructions like $a = b + c$. Three address architectures were common. Such architectures were possible because the address space was small and the word size was large. In fact there was no notion of bytes – just words. The word size was determined by the number of bits needed to hold floating point numbers, for example 60 bits. This allowed sufficient room for instructions to contain the three address fields.

Note that the only way of iterating through an array of values at that time was to modify the address fields of the instruction. Self modifying code was considered to be an essential part of programming. People had not yet discovered index registers. Pointers were not really needed from the hardware perspective, but they gradually evolved because they simplified the programming process.

An understanding of computer architecture is a required part of a programmer's core knowledge. We recommend one and a half books. First, Hennessy and Patterson's book [**5**] is the best way to come up to speed on modern computer architectures. In fact, we suggest rereading it every time a new edition is published. The other half-book is the second part of Blaauw and Brooks [**2**]. They call it "A Computer Zoo". It catalogues many of the historically important computer architectures, helping the reader gain an understanding of why things are the way they are today. Brooks led the development of OS/360 and Blaauw was responsible for the instruction set of the IBM 360. It is also worth reading the Mythical Man Month [**3**] where Brooks wittily describes the process of developing the OS and how things could go wrong.

People started to discover that the notion of an address as a separate object was a good idea. The first computers did not have pointers, nor did the first programming language: Fortran. Nor did the "improved" Fortran: Algol 60. You could easily implement a linked list in these languages. Programmers had to use, for example, indices into an array. (How would one implement a linked list in a language which had neither pointers nor record types with arrays being the only aggregation mechanism?) It became clear that index registers were needed, although originally they were not necessarily big enough to hold a whole address. The push for addressability/pointers was coming from application writers. This made life harder for compilers, introducing aliasing problems (preventing reordering optimizations, etc.). Compiler writers had an easier time with Fortran since there were no pointers at all and it was designed to facilitate optimization.

As people moved from processing numbers to also processing data, architectural questions arose concerning the appropriate minimum unit of addressability. Not all machines offered (the relatively expensive) byte addressability. Apparently, when Hennessy was designing the first MIPS computer he wanted to make it only word addressable, in the full spirit of RISC. (Originally the answer to such questions was not clear, for example, why not have bit addressability? The IBM Stretch architecture offered bit addressability. People gradually realized that this would make the address space too large and it would require expensive hardware. By trimming things at the bottom of the tree you could save a lot of address space machinery.) In any case, Unix and C required byte addressability. By about 1970, the idea that we call the *Common Machine Architecture*, or the *C machine model* was established. This assumes the presence of byte addressability, pointers, and two kinds of numbers: reals (floating point – support for automatically dealing with mantissa, exponent) as well as fixed point (integers). The IBM 360 established the standard that everything was (8-bit) byte addressable. At the time people thought that all characters could fit into 8 bits. Prior to that there were attempts to have 6 bit byte (this worked well with 36, or 60 bit words). Powers of two were not native to computer science. One of the great computer architecture of all time, Seymour Cray's Control Data 6600, used a word size of 60, with no byte addressability. Instruction words were divided into four 15-bit chunks. It could be viewed as a form of VLIW since in some sense it went in parallel. Eventually, 8-bit bytes became the norm.

Another attempted architecture was to simplify instructions by offering, say, a single hardware '+' instruction that would work both for fixed and floating point numbers. The idea behind these so-called *tagged architectures,* such as Bob Barton's Burroughs 5500, which used an Algol-60 instruction set, was that sets of bits in memory would be *tagged* with a data type. This resembles the notion of a finite/closed dynamically typed system. Of course such an idea is not extensible; it breaks down once a language supports record types.

Parallel arrays were used instead of record types – every field of a record would be distributed into a different array. A record would be a united across all of the arrays by a common index. Even though we now have record types we shouldn't discount this technique since it still has uses today. For example, if we are just looking at the first field, parallel arrays can be better since we the rest of the record does not need to be dragged into the cache. In the 1960s people eventually figured out that we could have record types where types could be aggregated. Record types, or structs, require byte addressability since thus allows natural representation. Also remember that records are generally padded. Words are loaded by word boundary even though we have byte addressability.

There is an old philosophical problem going back to the 14th century (see for example Copleston's *History of Philosophy*): Given two things of the same form what makes them different? It is not the particular value of the thing. In the middle ages the best answer was probably that developed by Duns Scotus who said that there was a mysterious thing called "haecceitas" or "thisness". In order to distinguish between different things, it is not sufficient that the attributes are the same, but there is some notion of thisness, or in computers, an address. While we can change an object's attributes, its address stays the same.

What are the minimal mathematical properties of an address – what can we do with an address? We need to be able to test for equality. An address has to also allow access to data, or a *dereferencing* operation. An address serves the same role as a name in philosophy: something which stands for something else, something which can be dereferenced. But there is another requirement. It is not enough to refer to someone as, say, "the fat Russian guy with 9 different recordings of Wagner's Ring", even if that would uniquely identify him. Addresses also need to offer a "fast" form of access to the value. Fast is not very easy to define in this context. For example memory access is not in fact constant time – in hardware memory access is typically logarithmic in the length of the address. So we will just require the fastest class of access for a given context.

1.2. Generic Algorithms

John Backus, the father of programming languages, came up with a grand vision that the problems with memory could be solved by introducing what he calls *functional programming*. Though we will refute a number of the ideas, his Turing award [**1**] lecture is brilliant and instructive and must be considered required reading. There he introduced the language FP, and a second order language FFP. He upheld the idea of dealing with computation without memory/state. Instead of having global storage, programs copy it around. For example, to change one element of a sequence, a new sequence with the changed elements must be generated. The problem with this approach is that it is very expensive. The notion of an address is in fact quite useful.

Ken Iverson's APL inspired John Backus's FP work. Iverson became an assistant professor at Harvard, somewhere around 1959-1960. He realized that very often we work on large objects such as matrices. He asked why have to write loop to work with them? He decided to create a language based on vectors and matrices. His great contribution was the idea of introducing an *operator* (e.g. higher order function) that could be applied to one function to create another. For example, the reduction operator '/', when applied to the binary '+' function in APL, produces a new function that +/ which adds all the elements in a vector, e.g. $+/1234$ yields 10. $*/1234$ gives 24. His operators were later referred to by Backus as "functional forms". In STL they are called generic algorithms. In APL it is not possible for users to define new functional forms.

In 1961 Iverson published a book called "A Programming Language" (APL). However, he did not publish enough and was denied tenure at Harvard, so he moved to IBM research. Lots of physicists learned APL. Iverson convinced the hardware guys to use APL as their hardware description language. They defined the semantics of the IBM 360 series in APL. This was probably the first example of a formal definition of a real computer system. This is why Brooks and Blaauw use APL as a hardware description language in their book. Eventually he left IBM and went to I.P. Sharp, a Canadian time sharing company that sold a lot of APL to financial analysts. APL always remained a small cultish language. He also tried to use APL as an instructional language in several somewhat misguided but brilliant books to teach high school algebra. He received a Turing award in 1979.

There are two fundamental ideas in Backus besides the idea of abolishing state. One is the idea of functional forms, that were later called generic algorithms. Though Iverson recognized this idea, Backus really brought it out. Functional forms are higher order functions, that is, they take a function and produce a new function. Go back and think about *power* – it is a functional form. The second idea is that functional forms are associated with mathematical laws. For example, reverse followed by reduction is the same as reduction followed by reverse (assuming an associative and commutative function).

The problem with APL and FP is that they do not support user-defined functional forms. We need to have a way of creating new ones. And we still don't have a way of describing their mathematical properties. For example, said that *power* only makes sense when operation is associative, but we cannot ask the compiler to check this. So if we take functional programming, keep the functional forms and the algebraic laws, but add extensibility, and allow state together with a generalized notion of addresses we end up with *generic programming*.

EXERCISE 1.1. Read Iverson's papers [**7**] and [**6**].
Read Backus's paper [**1**]

CHAPTER 2

Introductory Algorithms

2.1. Iterators

We don't like the term iterator since, for example, trivial iterators don't have any ability to iterate – navigation is a separate notion. Coordinate, position or address would be preferable. Historically speaking, iterators were a difficult discovery, especially coming from the functional programming perspective which prohibited addresses. Addresses were desirable, not for efficiency, but because it was impossible to describe things conceptually without them, e.g. what does something like *find* return, particularly when it doesn't succeed? Lisp can return a universal bottom marker, *NIL*. But there is no such thing in a strongly typed language. In functional programming, to replace a node in a list we need the address of the field in the node, not the address of the value.

Many people tried to solve this problem. They wanted to be able to write a *find* (they called it *position*) that would operate on both vectors and lists. In Common Lisp they introduced the notion of a *sequence* that could be either a list or a vector. Their *position* function returned an integer index. This works fine if the sequence happened to be a vector but it is quite inefficient in case the sequence is a list (since indexes do not provide constant time access to the elements in a linked list). They also tried to figure out how to specify a subsequence (e.g. with two integers). The reason that they failed was that they tried to come up with a type-less (uniform) way of indicating both subsequences and positions for all of types of sequences (Lisp doesn't have overloading).

The notion of iterators arose when we realization that iterators for vectors didn't have to be of the same type as iterators for lists, and also that it was OK for some types of iterators could be more powerful then others. The first idea was to introduce a notion called "coordinates" or "positions". Every data structure S would have an associated position type P such that there was a function *dereference* taking the pair to the value type, e.g. dereference: $(S, P) \rightarrow V$. You can see these ideas in [**8**]. Then we tried to look at the universe of C (after working in Ada). It became clear that very often a data structure is not available to an algorithm, e.g. it was necessary to also support pointers as iterators. Then, running quickly through five years worth of ideas, he realized that you could combine the pair of types into single type known as a *position*. It is straightforward to combine a pair of types S, P into a new type I so that dereference would be a function of a single argument. This strategy opens the door to using other lone types, for example non-pairs such as pointer types, as iterators too.

For a long time it was hard to figure out for how to write a generic copy algorithm. One reason for this was that at the time we were working in Ada. Ada has arrays that can be indexed by integral types. Ada also allows programmers to use a restricted integral type, for example integers in the range of 1 to 5. The difficulty was caused by the fact that an Ada array can be indexed by a restricted type containing exactly as many indices in the range as there are elements in the array. This prevented *copy* from working, and it even find *from* working. The range needs to be 1 larger than the number of elements. It was impossible to write *find* since there was no way of returning a position to indicate that an element was not found. Starting with 1 there are six possible subranges in an array of five elements, since we must account for the empty sub-range. So we need 1 more *position* than we have elements. Since Ada did not allow this *find*'s had to also return a boolean.

C guarantees that for every array there is a valid pointer past the end. This doesn't mean that it can be dereferenced. The fact that iterators need one extra position has nothing to do with STL, C or C++. It is a mathematical necessity. For example partition a sequence of two elements, (A B), into good and bad elements. There are three possible partition points, not two: either only A good, A and B are good or none are good. How many possible points about which to rotate? 3. How many possible insertion points? 3. Operations on sequences of n elements require $n + 1$ positions. Even in Microsoft Word there are $n + 1$ cursor positions in a buffer of n characters.

2.2. copy

We will develop the generic copy algorithm and it will gradually lead us through much that we need to know. It will also relate back to architecture.

```
while(first != last) /* we don't use < since it may not even exist
                        for out iterator type */
```

We also use ++ for moving forward. While this is regrettable it is necessary since it is important to be consistent when extending a language. The idioms for pointers needed to still work. That is, copy continues with

```
*result++=*first++;
```

We would rather have written dereference() then * – we are not particularly fond of post-increment. That is we would like to write, say

```
assign(first, result);
advance(first);
advance(result);
```

The idea of adding functions such as the above to the library was considered, but rejected because code needed to look like C. That was one of the difficult design decisions. At some point one has to take the host language seriously. We still regret using member functions in STL. For example one obtains the size of a vector via v.size() – it would have been better to say size(v), for two reasons. The minor reason is that it saves typing one character. But more important, the first form cannot be used uniformly for STL sequences and built-in C arrays. But in those days object-orientedness was king on the standards committee and it would not have been possible to get approval for the second form.

Now copy will need arguments of some type for *first*, *last*, and *result*. But what do we return from copy? We must return the updated value of result. Philosophically we never want to throw away any possibly useful information that the algorithm computed. Inside copy we advance *result* and it might be useful to the caller to know where it ended up. So we end up with

```
template <typename I, // I models Input Iterator
          typename O> // O models Output Iterator
O copy(I first, I last, O result) {
  while(first != last) *result++=*first++;
  return result;
```

}

Unfortunately this is not as fast as it might be. To figure out why we mark all the places where we do work. The assignment is the most important operation, because it actually does the work. There is no way to reduce the number of assignments when copying n elements. The dereferencing operation also does some work. The rest is overhead, like the test to see whether we have reached the end of the range. If we could somehow block things together, and also avoid testing for the end all of the time, we could optimize things. There is one other thing to consider, and it serves as an example of why it is important to know about computer architecture. If we know that the memory latency is 8 cycles, it is tempting to say that it will take 16 cycles to move an item from one location to another. This can be avoided through a technique known as software pipelining. Unless we assist the compiler it will not be able to pipeline, and even then there are problems which may prevent it from taking place.

Blocking is connected with loop unrolling. It is worth noting that genericity can potentially affect performance. If we assume that we have at most have input iterators we can't unroll because there is no way to avoid checking for the end of range after every operation. So we want to eventually have a language with category dispatch where we can first write the function *copy* defined for input iterators, then write another version of *copy* defined for random access iterators, and so on. The compiler would pick the strongest possible version for the given category of iterator. Since such a mechanism does not exist in C++ we had to invent an ugly hack for STL.

2.2.1. Iterator Categories. We will discuss five kinds of iterator; *Input Iterators, Forward Iterators, Bidirectional Iterators, Random Access Iterators,* and *Output Iterators,* together with their relationships. We will introduce each category of iterator by providing a minimal *model* that satisfies the necessary requirements. We will also indicate some of the axioms that must be satisfied in each case.

We begin with *Input Iterators*. For our model we consider a slightly unusual form of singly linked list:

$$\begin{tikzcd} 0 \arrow[r] & 0 \arrow[r] & \dots \arrow[r] & 0 \arrow[r] & \times \\ b \arrow[u] & & & & e \arrow[u] \end{tikzcd}$$

The illustration above is meant to indicate that the Input Iterator b refers to the beginning of a list, each element of the list has a pointer to its successor, and the last element of the list points to an (invalid) element just past the end, as does the Input Iterator e. In our list, as long as it does not point past the end, b can be (repeatedly) dereferenced, written as $*b$, to get the value of the first element. It is important to require that the iterator must be dereferenceable multiple times. Otherwise we couldn't write a useful generic *find* algorithm that: 1) works on the weakest possible kind of Input Iterators and 2) returns an iterator to the found element. For our implementation would have to dereference the iterator internally when searching. But without the guarantee that it was safe to dereference an iterator multiple times the caller wouldn't be able to reliably dereference the returned iterator. In addition to the ability to dereference, we need to be able to advance b to the next element, which we write as $++b$. As soon as b is advanced, however, we assume that the element previously referred to by b is destroyed (e.g. garbage collected). For example, if we make the assignment $a = b$ then advance b via $++b$ we cannot even safely write the expression $a == b$; whatever b referred to and all other iterators that referred to the same place might be completely invalidated by advancing b. So algorithms that can work with *any* form of Input Iterator, including the weakest possible kind, must be single pass algorithms. Such algorithms are also referred to as *online* algorithms – they look only at local data. For, once an iterator is advanced through the sequence, the elements no longer exist.

We introduce *Forward Iterators* to support algorithms that may need more than one (forward) pass. To allow for multiple passes, Forward Iterators must meet the requirements of Input Iterators, but the right to destroy elements after navigating past them is forfeited. The usual model of a structure supporting Forward Iterators is a traditional singly linked list.

Starting with Forward Iterators it becomes possible to introduce a general notion of iterator equality. We know that $i == j \implies *i == *j$. Though not an axiom, it will also often be true that $i == j \implies \& * i = \& * j$. In many cases, we could go so far as to define iterator equality in terms of the latter identity.

An interesting question arose in the design of iterators: should an iterator be required to know when it can be safely incremented? The answer is no. Given a single iterator we don't know if it can be dereferenced or incremented. But given a pair of iterators, we can define the notion of a *valid range.* For a valid range $[i, j)$ we know that $i \neq j \implies *i$, that is, if i not equal to j then i can be dereferenced. Also for such a valid range we have the law that $i \neq j \implies + + i$, that is, if i is not equal to j then i can be incremented. We can keep incrementing and dereferencing the beginning iterator of a valid range until it becomes empty. The validity of the range will not be harmed. Nor will it be affected by copying the range. So it becomes possible to argue about algorithm correctness. Valid ranges come from containers via the *begin()* and *end()* functions. Of course mutating a container, by inserting an element for example, will in general invalidate a range.

In addition to providing all of the capabilities of Forward Iterators, *Bidirectional Iterators* must support backward navigation, which we write as *–b*. Bidirectional Iterators can typically be supported by data structures such as doubly linked lists. It is important to connect the new capability (decrement) of Bidirectional Iterators to the exisiting capabilities of Forward Iterators. We do so by introducing the following law $i == j \ \&\& + +i \implies j == - - (+ + i)$. That is, if $i == j$ and if it is legitimate to increment i, then the result of incrementing then decrementing i will still be equal to j. As a theorem (consequence of the axioms) we claim that $[i, j)\&\&i \neq j \implies - - j$.

Random Access Iterators add to Bidirectional Iterators the ability to calculate the (signed) distance between two iterators i and j, which we write as $j - i$. The model of a data structure that supports random access iterators is, essentially, a chunk of memory with the following caveats: we cannot assume that memory exists on either side of the chunk. That is, we may assume valid addresses exist only from the beginning to the element one past the end. Even the address two past the end is *not* guaranteed to exist. As a result we have to be careful with calculations. For iterator i and integers m and n we can't just write: $i + m - n$, instead need to write $i + (m - n)$. Random access iterators must also support the addition a (signed) distance. These capabilities are subject to the following connecting laws (when all quantities are defined)

$$(i + m) + n = i + (m + n)$$

$$j = i + 1 \implies j == + + i$$

$$i + (j - i) == j \tag{2.1}$$

None of these operations may consume more than constant time. It is worth underscoring that the key capability added by Random Access Iterators is the ability to efficiently calculate distance. The ability to jump doesn't help if you don't know whether $i + m$ would be past the end. One more law is necessary for Random Access Iterators in order to assure compatibility with C pointer idioms: $i + n == \&i[n]$. Of course $j == i + 1$ does *not* guarantee that $\& * j == (\& * i) + 1$ (consider *std::deque*. That is, consecutive iterators do not necessarily refer to values at consecutive addresses. Nor does $i == j$ necessarily imply that $+ + i == + + j$. But for dereferencing it holds, i.e. $i == j\&\& * i \Longrightarrow *i == *j$.

EXERCISE 2.1. Show why the axiom in Equation 2.1 is necessary by coming up with an example of an absurd or undesirable definition of iterator subtraction that would be allowed in the absence of this law.

EXERCISE 2.2. Why do we define the ability to subtract iterator *i* from iterator *j* for Random Access iterators? Would it be useful to have, say, Semi Random Access Iterators that supported addition of (signed) integers, but not iterator subtraction?

Many normal axioms (axioms that hold for the other types of iterators that we have discussed so far) break for Input Iterators. For Input Iterators it is not true that $i == j\&\& + +i \Longrightarrow + + i == + + j$. When thinking about iterator categories, it is most helpful to keep the minimal models presented above in mind.

We briefly mention the strange beasts known as *Output Iterators*. Output Iterators support only two (equivalent) expressions concerning navigation and dereferencing: $*i + + =$ (equivalently $*i$ =followed by $+ + i$). This is rather awkward to state linguistically, but the point is that the expression $*i$ is only valid on the left hand side of an assignment statement – it has no meaning in and of itself. It is not valid to assign to $*i$ twice without an intervening increment. The model for a structure supporting Output Iterators is a pipe, or an output stream.

2.2.2. copy_n and Loop Unrolling. We return to the problem of producing faster versions of copy that take advantage of loop unrolling. If we know the distance, *n*, between *first* and *last,* we can dispense with *last* and rewrite *copy* as follows as *copy_n*

```
template <typename I, // I models Input Iterator
          typename O, // O models Output Iterator
          typename N> // N models Integer
O copy_n_version1(I first, N n, O result) {
  while(--n) *result++=*first++;
  return result;
}
```

The profound point is that if we can obtain *n* we can move toward loop unrolling, *even for Input Iterators*. We haven't actually made things much faster yet – in the next version we will try to unroll the loop by a factor of 4. Typically people unroll by for or 8 but not more. The main reason that people don't go beyond 8 has to do with the law of diminishing returns. The point of loop unrolling is to gain a decent percent improvement in the ratio loop overhead to overall code. Starting with, say 30% loop overhead, unrolling by a factor of 4 leaves us with about 8% overhead. Unrolling by a factor of 8 brings it down to a 4% overhead. Overhead below 4% is commonly viewed as noise – results could vary from CPU to CPU, etc. In research we do unroll loops – 30% does not matter when we only want to demonstrate feasibility. But when it is time to transfer code to real applications then unrolling can be worth considering (only *after* fixing algorithms). Here is an unrolled version of *copy_n*

```
template <typename I, // I models Input Iterator
          typename O, // O models Output Iterator
          typename N> // N models Integer
O copy_n_version2(I first, N n, O result) {
  while(n >= 4) {
    *result++=*first++;
    *result++=*first++;
    *result++=*first++;
    *result++=*first++;
    n-=4;
  }
  switch(n) {
  case 3: *result++=*first++;
  case 2: *result++=*first++;
  case 1: *result++=*first++;
  default:
  }
  return result;
}
```

(Note the use of the C language feature that makes *break* statements optional). We have removed some loop overhead but the code is still not optimal. Unfortunately, we are doing 4 increments of the iterators. We may avoid the incrementing code by indexing. This gives the compiler the opportunity to hard-wire the displacements into the instructions and also to take advantage of possible parallelism.

```
template <typename I, // I models Random Access Iterator
          typename O, // O models Random Access iterator
          typename N> // N models Integer
O copy_n_version3(I first, N n, O result) {
  while(n >= 4) {
    result[0]=first[0];
    result[1]=first[1];
    result[2]=first[2];
    result[3]=first[3];
    n-=4; first+=4; result+=4;
  }
  switch(n){
  case 3: result[2]=first[2];
  case 2: result[1]=first[1];
  case 1: result[0]=first[0];
  default:
  }
  return result;
}
```

EXERCISE 2.3. Test all versions of copy and see which is the fastest. Compare the performance of the *copy*, the loop-unrolled versions of *copy_n* with unroll factor 4 vs. 8, with variations using ++'s in the body vs. versions using indices. Do all 5 of the above tests for arrays/pointers and for vectors/iterators, for a total of 10 tests.

Try to produce a decent framework for measurement – we will use it for other algorithms in the future. The framework should have several parameters that can be varied: One parameter should control the number of elements in the container; one parameter should control the number of times the test should be repeated; one (compile-time) parameter should control the type of the elements. We will need to be able to experiment with, say, ints, bytes, and structs containing N doubles. Don't forget to use high optimization settings when compiling.

When copying an array of 800 bytes would it even make sense to unroll the loop? We can estimate the number of cycles needed to copy 800 bytes as: 800÷(2×bus-width), since data needs to pass through the bus twice, when copying. Assuming a bus width of 32 we can assume that copying the data will take at least 12 cycles. The loop overhead for *copy_n*, consisting of a single comparison, is not worth trying to optimize away when the body does so much work, since the expectation is that on modern CPUs it will be executed in parallel with the other instructions.

Note that *copy_n* is very easy for compilers to unroll. It is also useful to know whether our compiler has the ability unroll loops in the case when it doesn't know *n*. In many cases it cannot. We hope that eventually the compiler will optimize all cases that today require hand unrolling, hand software pipelining, or hand blocking. Besides the fact that it would be nice to avoid unnecessary work, successful use of these techniques require the programmer to have knowledge of instruction ordering, parallelism, relative latencies and so on. But advances in CPU technology make it difficult to accurately evaluate and predict performance, especially across CPU releases. It is common knowledge that advances in compiler technology that will eliminate the need for such techniques, is "just around the corner". Unfortunately, such knowledge has been common since the 1960s, so we still need to know a little bit about low level optimization.

We discovered that changing *copy* to *copy_n* allows us to unroll. It fundamentally works with the same requirements as regular copy (Input Iterator, Output Iterator) since the inclusion of the *n* parameter solves the problem of when to stop . Could we replace every call to *copy* with a call to *copy_n*? No, since we need to know the length of the range. The algorithm *std::distance(i, j)* calculates this for us this with the precondition that [*i*, *j*) is a valid range. (This precondition is so common that we will no longer mention it.) Of course *distance* might have the side-effect of destroying the range. Why don't we have a function *i*s_valid *_range*(*i*, *j*)? Because if we don't know whether the range is valid then we can't even know whether it is safe to advance *i*, or whether advancing *i* would ever cause it to reach *j*; a loop with such a test may never even terminate. This is very important to remember. So we must always be careful to keep track of valid ranges.

In fact, in C++ < (less than) is not even defined for *void**. This is unfortunate since this prohibits us keeping *void** pointers in a set (sets require that elements are comparable). We believe that it is desirable for all types (including structs) to provide comparison and equality operations. But even if this was so, we still wouldn't be able to determine if two pointers constituted a valid range, since memory is not necessarily contiguous. Since we can't check range validity at runtime, we have to make sure at coding time to preserve iterator range validity.

2.3. distance and Compile Time Dispatching

In order to be able to call *copy_n*, we need to calculate the distance between *first* and *last*. We want the implementation of *distance* to be as fast as possible for a given iterator category. For Input Iterators we won't be able to do better than writing a loop. However, a single subtraction will suffice for Random Access Iterators. C++ doesn't provide direct support for selecting the best implementation on the basis of Iterator category. As far as the compiler is concerned, the notion of Iterator category doesn't even exist – it is simply a naming convention that we use when naming template parameters. So we can't just write different overloaded versions of distance to achieve the efficiency that we want. We illustrate how to write a version of *distance* that will use *compile-time dispatch* on the Iterator category to select the most efficient implementation. *distance* is very simple algorithmically, so we can focus on the techniques needed for compile-time dispatching in C++.

Our requirement is that the library will provide a single function template, *distance,* that will in turn select and call the appropriate helper function template based on the Iterator category. For Input Iterators the body of the helper function implementation will look something like

```
while(first != last ) { ++n; ++first;}
return n;
```

When STL was first being written, it was not at all clear what the function signature could be. Clearly, *distance* needs to return some kind of integer type. But what it needs to return depends on the type of iterator. For iterators that happen to be implemented as classes it is reasonable to write

```
template <typename Iterator>
typename Iterator::difference_type
distance(Iterator first, Iterator last);
```

That is, we can require that the author of any iterator class provide a nested *typedef* called difference_type that indicates the correct type for *n*. But such a signature will not be of much use in cases when the iterator is a pointer.

In the development of STL when exploring possible solutions to the problems presented by *distance,* we came up with a related algorithm, *advance_by* in which the caller is required to supply the type of *n*:

```
template <typename T1, //T1 models Incrementable
          typename T2> //T2 models Incrementable
T2 advance_by(T1 first, T1 last, T2 n) {
  while(first != last){++n; ++first;}
  return n;
}
```

The problem of specifying the return type is avoided by asking the caller to supply the type $T2$ in *advance_by*. Very often we can simplify interfaces and avoid complicated dispatching tricks, like those demonstrated below, by extending the set of types in a function template signature. Many times, after introducing the additional type(s) we find other uses for them. Here *T2* was added to avoid having to determine a return type. But the introduction of the additional parameter turned out to provide additional benefits. It is possible to call *advance_by* for types $T2$ which don't come equipped with a zero element, since the caller can supply the initial value.

Iverson's paper helps build up nomenclature for the important techniques that we can use in code. The APL notation doesn't matter, but it is important to be able to recognize and communicate about reduction, for example. In the case of *advance_by* we need a better name, but the idea is the point. We replaced Input Iterator with Incrementable in *advance_by* since we noticed that we didn't actually require the ability to dereference. Recall that PeanoTypes had only local zeros. These ideas aren't born full grown off the top of our head. Fragments of code that recur when working on algorithms are identified and evolved. We work to find the right name for the idea. We cannot afford, as a field, to have proprietary abstractions. The hope is that eventually we will work from a common set of abstractions and nomenclature. STL succeeded to a small degree, though people still fail to understand the abstractions. (Even the SGI STL documentation exhibits a misunderstanding of the requirements on < and ==.) Computer science needs to have common nomenclature in order to rise above the level of while statements.

While the implementation of *distance* (respectively: *advance_by*) given above might suffice for most classes of iterator, it would be a terrible implementation for Random Access Iterators (respectively: Incrementable types equipped with + and -) because, as we mentioned above, we could use a more efficient one-line subtraction instead. We return to our goal of writing to write a single piece of code, that can figure out the fastest possible implementation to call (at compile time). If we had concepts in C++, we could do a very simple thing:

```
/* ignoring return type for the moment */
distance(InputIterator first, InputIterator last)
{while(first != last) {/*...*/}}

/* ignoring return type for the moment */
distance(RandomAccessIterator first, RandomAccessIterator last)
{ return last - first; }
```

We can't do this in C++ since *InputIterator* and *RandomAccessIterator* are just words as far as the compiler is concerned. This is a major limitation of C++. concepts only exist in our minds. We simulate them with templates.

The first simulation technique employed when developing the STL was to extend the signature of *distance* with a third parameter to carry an *iterator tag*. We predefine *tag* classes, one for each iterator category, for example

```
struct InputIterator_tag {};
struct ForwardIterator_tag : public InputIterator_tag {};
// ...
```

We use inheritance since we want algorithms that require an Input Iterator to accept, say, a Forward Iterator in case there is no more efficient version of the algorithm available. This is one of the few legitimate uses of inheritance in C++. (We will have more to say about inheritance later.) We add the additional parameter to the *distance* helper function templates to indicate which category of iterator is being supplied. In this way the best implementation can be selected. For example

```
template <typename I> //I models InputIterator
inline /*return type discussed later*/
distance(I first, I last, InputIterator_tag)
{ /*...*/; while(first != last) {/*...*/}; /*...*/; }

template <typename R> //R models Random Access
inline /*return type discussed later*/
distance(R first, R last, RandomAccessIterator_tag)
{ return last - first; }
```

The tag type has no meaning at runtime. In fact the value passed as the third parameter is ignored – we use it only as a mechanism to drive the type system to select the appropriate implementation. The top level *distance* will arrange for the appropriate helper to be called by supplying a tag value as the third parameter. In order to determine the tag value from a given iterator type, we define a function template *iterator_category* that returns return a value of the type of its iterator category tag. Note that this style leads to a lot of glue code that occupies many lines but doesn't generate a single instruction.

```
template <typename T>
inline RandomAccessIterator_tag iterator_category(T*)
{ return RandomAccessIterator_tag(); }
```

This is just an ugly way of writing that "all pointers are random access iterators". *iterator_category* can be implemented for iterator classes by requiring that all iterator classes supply their category as a nested typedef named *iterator_category.* Of course nested typedefs are just another C++ embarrassment since what is really needed are type functions. In any case, we can now write our function as

```
template <typename I>
inline typename I::iterator_category iterator_category(I)
{ return typename I::iterator_category(); }
```

We can use these same dispatching techniques to calculate the return type for *distance* at compile time. We can't directly use the *iterator_category* function. Instead we create a function named difference_type that will indicate the type to be returned by the *distance* algorithm for a given iterator. For pointer types this will be *ptrdiff_t:*

```
template <typename T>
inline ptrdiff_t* difference_type(T*)
{ return (ptrdiff_t*)0; }
```

We actually return a pointer to the distance type for technical reasons (to avoid the cost of construction and copying). STL requires that all iterator classes supply the *difference_type* typedef inside so that it is possible to implement difference_type as follows

```
template <typename I>
inline typename I::difference_type* difference_type(I)
{ return (typename T::difference_type*)0; }
```

We can sketch the complete implementation of *distance*

```
template <typename I, //I models Input Iterator
          typename D> // D models Integer
D distance(I first, I last, InputIterator_tag, D*)
{D n=0; /*...*/}

template <typename R, //I models Random Access Iterator
          typename D> // D models Integer
D distance(R first, R last, RandomAccessIterator_tag, D*)
{ return last - first; }
```

The top level distance originally looked something like

```
template <typename I,// I models Input Iterator
          typename D>// D models Integer
inline void distance(I first, I last, D& n) {
        n = distance(first, last,
                     iterator_category(first),
                     difference_type(first));
}
```

EXERCISE 2.1. Without looking at an STL implementation, complete the implement a *distance* function template that dispatches to an appropriate helper algorithm for each iterator category. Complete and make use of the *iterator_category* and difference_type function templates that we sketched above.

Unfortunately, because of the lack of uniformity between user-defined and built-in types in C++, the above techniques were not sufficient to allow us to write *distance* with the desired signature – we needed a way to calculate the return type. In particular, it was not possible to use the difference_type function template to specify a return type for *distance.* So in the next solution the *iterator_category* and difference_type functions were replaced by *iterator_traits.* For every type of iterator, *I* (iterator class or pointer), it is possible to instantiate the traits class *std::iterator_traits<I>*. The role of the traits class is to provide the necessary typedefs for all of the types associated to the iterator *I*. Then, finally, it becomes possible to write the Input Iterator form of *distance* as

```
template <class I> // I models Input Iterator
inline typename iterator_traits<I>::difference_type
distance(I first, I last) {
  return distance(first, last,
                  typename iterator_traits<I>::iterator_category());
}
```

Here is an implementation of *iterator_traits* for iterator classes

```
template <typename I>
struct iterator_traits {
  typedef typename I::difference_type    difference_type;
  typedef typename I::value_type         value_type;
  typedef typename I::reference          reference;
  typedef typename I::pointer            pointer;
  typedef typename I::iterator_category  iterator_category;
};
```

One of the principal motivations for adding partial specialization to C++ was to allow *iterator_traits* to work for pointers too

```
template <typename T>
struct iterator_traits<T*> {
  typedef ptrdiff_t                        difference_type;
  typedef T                                value_type;
  typedef T&                               reference;
  typedef T*                               pointer;
  typedef std::random_access_iterator_tag  iterator_category;
};
```

EXERCISE 2.2. Complete the implementation of a *distance* function template that dispatches to an appropriate helper algorithm for each iterator category. This version of distance should return its value instead of taking a reference parameter to hold its result. Make use of the *iterator_traits* class templates provided above. You will need to slightly modify the signature and implementation of the *distance* helper routines from the previous exercise.

EXERCISE 2.3. Implement *copy* using the same techniques. For Random Access Iterators it should dispatch to *copy_n*.

EXERCISE 2.4. Implement the *advance* algorithm using the same techniques.

2.4. Analyzing a Program

In the first section of the book we assigned as one of the problems a task of writing a program to determine if a sequence is a derangement. Now we will use a solution of this program given by one of our strong students as a way to discuss how to evolve a piece of code.

The solution given to us was:

```
template <typename T>
bool is_derangement(const T* start, const T* end)
{
  T pos = 0;
  while (start != end) {
    if (*start++ == pos++)
      return false;
  }
  return true;
}
```

Let us be clear about it, this code is good; it is an adequate solution for the problem and there is often no need to touch it. We, however, love rewriting code. As a matter of fact, we believe that what distinguishes a born programmer, a programmer for whom programming is not just a job, but a calling, is love for code rewriting. Anybody can enjoy writing code, but only a born programmer loves rewriting code for no particular reason.

Let us start with a couple of trivial issues. There is certain unnecessary flamboyance in the choice of variable names. We always use [*first, last*)to describe an input range. While it might seem to be irrelevant, changing the boring [*first, last*) to [*start, end*) might cause some unneeded wonderment. Is it just a different way of naming things, or is there some desire to indicate that the range is not just a regular range? In general, we try to use names in a consistent way. While it is perfectly legal to have a floating point variable called *n*, it is confusing since a reader of the code will promptly forget the declaration and assume that *n* is an integer. While there is a distinguished programming tradition at MIT to name all variables *foo, bar, baz, honoze,* etc, we should admire them for producing some amazing programming artifacts, but we should not imitate their peculiar habits. It is also good to avoid abbreviations: we can use the time at the keyboard to think about what we are writing. So, we rewrite and obtain:

```
template <typename T>
bool is_derangement(const T* first, const T* last)
{
  T position = 0;
  while (first != last) {
    if (*first++ == position++)
      return false;
  }
  return true;
}
```

Now, we observe that our template type parameter is called *T*. What do we mean by it? Could we instantiate this function with any type *T*? It appears not to be the case, since not every type would allow us to do a post-increment on it and to initialize it with 0. Though C++ does not allow us to say it formally, we better indicate our intent:

```
template <typename T> // T models Integer
bool is_derangement(const T* first, const T* last)
{
  T position = 0;
  while (first != last) {
    if (*first++ == position++)
      return false;
  }
  return true;
}
```

Requiring a type to be an integer type is, of course, an overkill. We could have specified Peano Integer as our constraint since only 0 and *successor* are really required. But then we would be suffering from premature over-generalization. We do not know of any important type that is a Peano Integer, yet not a full Integer. And, as we shall eventually see, a far better type interface will become evident.

Here we need to ask a question about 0. Do we really need 0? What if our future client decides to index their permutations from 1? Couldn't we accommodate them? Indeed a simple trick of letting initial position be an argument solves this problem:

```
template <typename T> // T models Integer
bool is_derangement(const T* first,
                    const T* last,
                    T position)
{
  while (first != last) {
    if (*first++ == position++)
      return false;
  }
  return true;
}
```

Now we observe that we do not need *T* to be an Integer. The only condition on *T* now is that it is Incrementable:

```
template <typename T> // T models Incrementable
bool is_derangement(const T* first,
                    const T* last,
                    T position)
{
  while (first != last) {
    if (*first++ == position++)
      return false;
  }
  return true;
}
```

For example, now we can encode our permutations as an array of pointers into the array itself. (We will need to wrap pointer into a structure; doing so makes for a good C++ exercise. This could even be done with iterators into a container. We can define a container of structures that contain iterators to the container. Try doing it.)

This forces us to consider why we need to deal with pointers. Since we are working with a single pass, forward moving algorithm, we can rewrite it as:

```
template <typename It,  // It models InputIterator
          typename T>   // T  models Incrementable
                        // defined: It i; T x; *i == x;
bool is_derangement(It first,
                    It last,
                    T position)
{
  while (first != last) {
    if (*first++ == position++)
      return false;
  }
  return true;
}
```

It is not necessary for the return type of the dereferencing of the iterator to be the incrementable type. It is only required to be equality comparable.

Now let us consider what our code is doing. It is finding the first place in a sequence that, in some sense, is pointing to itself. But what we return is just a Boolean value that indicates whether such an element exists. Our interface loses valuable information! What if we want to find an average distance to a first fixed point in a permutation? Our Boolean value will not help at all. Here we need to remember the following general rule: behind every "is?" there is "find." (We do not believe in the existence of things that can never be found.) It is almost always important to expose such find to the user:

```
template <typename It,  // It models InputIterator
          typename T>    // T  models Incrementable
                         // defined: It i; T x; *i == x;
It find_derangement(It first, It last, T position)
{
  while (first != last && *first != position) {
    ++first; ++position;
  }
  return first;
}
```

Now our new function returns the rightmost iterator into the sequence such that a range from first to it is a derangement. It is of course possible to provide a convenient shortcut:

```
template <typename It,  // It models InputIterator
          typename T>   // T  models Incrementable
                        // defined: It i; T x; *i == x;
inline
bool is_derangement(It first, It last, T position) {
  return last ==
    find_derangement(first, last, position);
}
```

If we look at our *find_derangement* function we observe that it looks remarkably like the well known STL function *mismatch*. As an exercise, we will ask you to find a way to unify these two algorithms. But first, we will take some time to completely implement an iterator.

2.5. Implementing an Iterator

EXAMPLE 2.1. We implement a relatively simple iterator called *value_iterator*. We would like to be able to fill a destination with the consecutive integers from 5 to 25 by writing *copy(value_iterator<int>(5), value_iterator<int>(25), result)*. We could implement this by maintaining an underlying vector of 20 elements, keeping an iterator into it, but we would be wasting memory. Instead, we implement an iterator with no underlying container to generate these sequences for any type that has the notion of a successor (i.e. a type that defines ++, a model of Incrementable). We begin by naming four of the five associated types:

```
template <typename T> //T models Incrementable
class value_iterator {
  public:
typedef T                                  value_type;
  typedef const T*                         pointer;
  /* const since these are generated values they are not mutable */
  typedef const T&                         reference;
  typedef std::forward_iterator_tag        iterator_category;
```

Why not use a stronger iterator category? We don't have any indication that we will need random access for this kind of iterator – we won't be sorting these things. We might occasionally want to have a bidirectional iterator, but it is important to consider the usage context so as to avoid adding unnecessary machinery. We view it is a generalized *iota* (from APL), where, for example, we my not always want to start with 1.

We get stuck when we try to supply the *typedef* for *difference_type*. We have no way to know the appropriate type to use, so we must ask the user to supply it. When *T* is an *int* we can just use *T*, but this won't work when *T* is, say, a pointer type. In that case we would want *difference_type* to be *ptrdiff_t*. So we rewrite:

```
template <typename T,    //T models Incrementable
          typename D=T>  //D models Integer
class value_iterator {
public:
  typedef T                                value_type;
  typedef const T*                         pointer;
  typedef const T&                         reference;
  typedef D                                difference_type;
  typedef std::forward_iterator_tag        iterator_category;
```

Now that we have defined these five typedefs we will use them in the implementation with no further mention of *T* or *D*.

```
private:
  value_type value;
```

We place the *private* section in between two *public* sections because it is better for a reader to be able to understand the code in a single pass. As in C we try to show each variable declaration before it is used. We believe that such considerations outweigh the need for "hiding" the private declarations at the end of the class. Defining the inline functions out of the body of the class would not help either. Moving such functions out of the class body triples the code size. It is important to keep code as short as possible, unless we do something algorithmically important. C++ is already verbose enough so we don't use information hiding that causes one-line functions to occupy many extra lines. Information hiding can be useful for large body of code, but not for simple abstractions.

Now we need some constructors. It is important to support the ability for clients of our library to write *T a; a = b;* This is possible for built-in types and should also be possible for user-defined types. We need to be able to express the idea that "it does not matter what the value is". We don't ignore the semantics of C – Ritchie was very smart. So we allow for default constructed iterators, but we cannot rely on others providing such a facility for us to use. We will also need a copy constructor, but we don't need to write it (nor must we write a destructor or an assignment) since the compiler-generated versions will suffice. We do need to provide a default constructor implementation, since the compiler will not generate one if we write any other constructors. Go figure. Continuing with constructors in the code:

```
public:
```

Note that it is more efficient to directly initialize the variable than it is to use a default construction followed by assignment. Also take note of our use of the *explicit* keyword to inhibit automatic conversions. Dennis Ritchie came up with the C type conversion rules as a sort of "poor man's substitute" for numeric genericity. This is one of the few cases where we believe that he made a bad compromise. The use of *explicit* will protect programmers from mistakes, like when they inadvertently pass a value of type *T* to a function that expects an argument of type *value_iterator<T>*. A more compelling example of the need for *explicit* occurs with *std::vector.* There, before C++ supported *explicit* constructors it was possible to pass an *int* to a function that expected a *vector* with no complaint from the compiler – the *int* would be implicitly converted into a vector using the single argument *int* constructor. When programmers first start using *explicit* constructors they are often confused by the following issue. Consider:

```
T x;
U y=x; /* If T and U are different types then this line */
U z(x); /* can have a different meaning from this line */
```

The first line means: implicitly construct an (anonymous) value of type U from the value x of type T, then assign it to y. The second line says to construct the value z of type U directly from the value x of type T. In case the T-conversion constructor of class U is marked as explicit, the first form above should generate an error, since the implicit construction is disallowed.

A long time ago Abelson & Sussman wrote the classic book "Structure and Interpretation of Computer Programs". Though it teaches a number of incorrect things, it is nevertheless a very great book. We strongly urge you to read it, no matter how long you have been programming. In the book they introduce the notion of a "first class object" (this has nothing to do with object-oriented programming). We don't believe that they got it completely right, but they were on the right track. The idea is that a first class object is something that can be: passed to a function, returned from a function, and stored/retrieved from a data structure. Every object in the language of the book, Scheme, is a first class object. Scheme has the very nice property that functions are first class objects, unlike in C++. In C++ arrays can not be passed to, or returned from a function. Our notion of *regular type* is an attempt to extend this idea of "first class objects". In particular, regular types must have a default constructor – otherwise we can not have an array of elements of these types. We want to write things which will work seamlessly within the entire language framework.

Now we need to create the six operators: *, ++ (two versions), *operator->*, *operator==*, *operator!=*. (By the way, we believe that C++ needs to always define *operator!=* as *!operator==*.)

```
reference operator*() const {return  value;}
reference operator->() const {return  &this->operator*();}
```

It is useful to define *operator->* in case the value type is a *struct* like *complex* and clients want to use the -> syntax to access members of the struct. We use the above implementation of *operator->* because it will work no matter how *operator** is implemented. That is, the code has the advantage of being paste-able into other contexts. Note that there are two versions of *operator++* in the final version of the class below. The post-increment version takes a dummy argument of type *int*. Also we implement binary operators as friends to preserve symmetry.

```
template <typename T,       // T models Incrementable
          typename D = T>   // D models Integer
class value_iterator {
  public:
  typedef std::forward_iterator_tag        iterator_category;
  typedef T                                value_type;
  typedef D                                difference_type;
  typedef const T&                         reference;
  typedef const T*                         pointer;
  private:
  value_type      value;
  public:
  value_iterator(){}
  explicit
  value_iterator(const value_type& v) :
    value(v){}
  reference operator*() const {
    return value;
  }
  pointer operator->() const {
    return &(operator*());
  }
  value_iterator& operator++() {
    ++value;
    return *this;
  }
  value_iterator operator++(int)  {
    value_iterator tmp = *this;
    ++*this;
    return tmp;
  }
  inline friend
  bool operator==(const value_iterator& x,
                  const value_iterator& y) {
                    return *x == *y;
                  }
  inline friend
  bool operator!=(const value_iterator& x,
                  const value_iterator& y) {
                    return !(x == y);
                  }
};
```

We offer the following criterion: a class is fully defined if the public interface of a class allows us to implement equality. If we need access to *private* data to implement equality then there is something wrong with the class.

EXAMPLE 2.2. We now extend *value_iterator* to work for Regular Types. Instead of using *operator++* we parameterize *value_iterator* to accept a function object to do the incrementing. That is, we weaken the requirement on the value type from Incrementable to Regular Type and require that a function object is passed in at runtime whose role is to advance to the next value. We will also need a good default value for the function object so that our clients will not have to do extra work in case the value type is already Incrementable. We begin by defining the default function object.

```
template <typename T> // T models incrementable
struct advance {
  T& operator()(T& x) const {return ++x;}
  friend inline
  bool operator==(const advance& x, const advance& y) {
    return true;
  }
};
```

We did not provide *typedef*s for argument type or result since we are not intending that *advance* will be used with *bind1st.* Even if we did, *bind1st* does not work correctly with reference return types. We return a reference from *operator()* to support the *advance(advance(x))* idiom. In a moment, we will explain why we took the trouble to make *advance* into a Regular Type by providing the equality operator.

```
template <typename T,     // T models Incrementable
          typename D=T, // D models Integer
          typename F=advance<T> >  // F models Mutator
class value_iterator { /* functional programmers call this a stream
  */
public:
  typedef T                               value_type;
  typedef const T&                        reference;
  typedef const T*                        pointer;
  typedef D                               difference_type;
  typedef std::forward_iterator_tag        iterator_category;
private:
  value_type              value;
  F                       fun;
public:
  value_iterator() : value(value_type()){}
  explicit
  value_iterator(const value_type& value, F f=F()) :
  value(value), fun(f){}
  reference operator*() const {
    return value;
  }
  pointer operator->() const {
    return &(operator*());
  }
  value_iterator& operator++() {
    fun(value);
    return *this;
  }
  value_iterator operator++(int)  {
    value_iterator tmp = *this;
    fun(value);
    return tmp;
  }
  inline friend
  bool operator==(const value_iterator& x,
  const value_iterator& y) {
    return *x == *y && x.fun == y.fun;
  }
  inline friend
  bool operator!=(const value_iterator& x,
  const value_iterator& y) {
    return !(x == y);
  }
};
```

When implementing equality for this version of *value_iterator,* it is not sufficient to check for equality of the corresponding *value* members. If incrementing is allowed (that is, if we are not at the end of a valid range) then the axiom $i{=}{=}j \Longrightarrow {+}{+}i{=}{=}{+}{+}j$ must hold. An instance of *value_iterator* that uses a *fun* which advances by 2 is not equal to an instance of *value_iterator* which advance by 3, even if their values are currently the same. So we take *fun* into account when implementing *value_iterator::operator==*. This is why we needed to supply *operator==* for *advance.*

Now we return to the problem posed at the end of Section 2.4.

EXERCISE 2.3. Use *value_iterator* to generate a sequences of odd numbers.

EXERCISE 2.4. Use *value_iterator* to unify *mismatch* and *find_derangement*.

2.6. Type Functions

We want to be clear that we do not like constructs like *iterator_traits*. They are necessary because C++ does not allow us to treat built-in types and user defined types in a uniform manner. At a deeper level, we are trying to establish certain fundamental relationships between types. To do so we require the fundamental notion of a type function. This takes some getting used to for C++ programmers since they are more familiar with functions that take values as parameters and return a value. A *type function* takes *types* as parameters and returns a type. The *value_type* of an iterator is really a type function: given an iterator type we somehow need to find its value type. Since type functions don't exist in C++ we are forced to write expressions like *value_type: std::vector<int>::iterator* instead of *value_type(iterator_type(vector_iterator<int>))* .

There are five types affiliated with each iterator. (In many respects this is three too many. *iterator_category* is not really desirable as a type. In a language that properly supported generic programming, dispatch to the appropriate function would not require anything like *iterator_category*.) Ignoring C++ for the moment, what are the two type functions that we really need to apply to an iterator type? For a given iterator type, we need to calculate the return type for *operator**. This is fundamental – it is not about C++. We have a very simple operation, the dereference operation, whose return type varies based on the type of iterator. Dereferencing takes a value of an iterator type I to a value of its associated value type V. Actually, we don't quite want V to be a value type – we want it to be a reference type. From a reference type it is possible to calculate a value type and a pointer type; these two calculations did not come into being because of anything having to do with iterators. They exist partly as a result of an over-ambitious attempt to support multiple different pointer and reference types. (At the time compilers offered, for example: *int**, *int far**, and *int long**, two different 32-bit pointer types that worked differently. For one of them the carry would never cross the 16-bit boundary under *operator++* !) We attempted to accommodate this in an abstraction called an *allocator* (combining memory and allocation). It couldn't possibly work, largely because *&* can not be overloaded, at least for built-in types. This is the reason that we have *pointer* and *reference* typedefs in iterators. Once again, although we would like to be able to write *reference(I)* to recover the reference type from an iterator of type *I,* in C++ we are forced to use the nested *typedef* idiom so we usually end up writing something like *typename iterator_traits<I>::reference* instead.

2.6.1. The Limitations of Object-Oriented Programming.

An iterator consists of a main type, let's call it *I,* together with two associated types: the type resulting from the dereference operation, *R,* and the distance type, *D*. That is, we view an iterator as a triple of types $\{I, R, D\}$. A number of algorithms depend on more than one of these types. When code depends on multiple types object-oriented-programming breaks down. The idea of object-oriented programming is to cluster pieces of code around a single type. Clustering is a wonderful idea, but we discovered that it needs to happen around multiple types, not just one. A typical kind of example that object-oriented people love to use is, say, to start with Semigroup from which they then derive Group, followed by Ring – then they stop. They forget to continue as far as Vector Space. This one is not derivable. A Vector Space requires a pair of types $\{V, S\}$, where V is the type of the Vectors (which, incidentally, must form a Group) and S is the type of the Scalars (which form a Field), and where certain other laws (such as the distributive law) must hold. The signatures of the operations upon a Vector Space are expressed in terms of both S and V.

V and *S* types do not enjoy equal status in a Vector Space. The Vectors are clearly more important. We argue that the axioms for Vector Spaces are of secondary importance. To abstract, we insist on starting from a particular thing such as Socrates, then abstracting to a type, Human, finally to a concept (Genus) Animal. The same approach works in mathematics: we start with 5, we abstract to the type int and from there we abstract to the Concept of Number. Fundamentally, a model of a Vector Space *is* a particular collection of Vectors. The crucial consequence of this is that the type of the Vector uniquely determines the type of the Scalar. The type of an Animal uniquely determines the type of food that the animal eats. The reverse is not true. There will always be one principal type for an abstraction, usually accompanied by several auxiliary types. We need the ability to include as part of our abstractions the type functions that let us recover these auxiliary types. Our Vector Space abstraction needs to include a type function *scalar* which can recover the type *S* from the type *V*, e.g. *scalar(V)=S*. Then, for a generic algorithm like *inner product,* we use the *scalar* type function to produce the signature *scalar(V) inner_product(const V& a, const V& b).*

Object-oriented programming works fine in the case of interfaces where only one type varies. For example, if *Cow* is derived from *Mammal* we might declare the virtual member function *void Mammal::sleep().* We would have no problem overriding sleep in the derived class: *void Cow::sleep().* We can depict this in the commutative diagram below:

$$\begin{tikzcd}
\text{Mammal} \arrow[r, "\text{sleep}"] \arrow[d] & \text{void} \arrow[d] \\
\text{Cow} \arrow[r, "\text{sleep}"] & \text{void}
\end{tikzcd}$$

Now we consider the binary function *mate*. What we want is *Mammal Mammal::mate(Mammal)* and *Cow Cow::mate(Cow),* visualized as:

$$\begin{tikzcd}
\text{Mammal} \times \text{Mammal} \arrow[r, "\text{mate}"] \arrow[d] & \text{Mammal} \arrow[d] \\
\text{Cow} \times \text{Cow} \arrow[r, "\text{mate}"] & \text{Cow}
\end{tikzcd}$$

But the object-oriented paradigm only allows for variation in the first parameter. Even with covariant return types, the best that we can do is *Mammal Mammal::mate(Mammal)* and *Cow Cow::mate(Mammal).* It goes without saying that this could be rather awkward for the cow. We illustrate these interfaces below:

$$\begin{tikzcd}
\text{Mammal} \times \text{Mammal} \arrow[r, "\text{mate}"] \arrow[d] & \text{Mammal} \arrow[d] \\
\text{Cow} \times \text{Mammal} \arrow[r, "\text{mate}"] & \text{Cow}
\end{tikzcd}$$

This is not just a fluke. The most important kind of operation, the binary operation, is not properly supported under the usual object-oriented single-dispatch model. Addition, multiplication, division are all binary operations, but they can't work in an object-oriented hierarchy!

There are further difficulties. Let us consider another function on our hierarchy: *eat*. A Mammal will eat MammalFood and a Cow will eat CowFood. Under our object-oriented decomposition, the best we might hope for would be:

$$\begin{tikzcd}
\text{Mammal} \times \text{MammalFood} \arrow[r, "\text{eat}"] \arrow[d] & \text{void} \arrow[d] \\
\text{Cow} \times \text{MammalFood} \arrow[r, "\text{eat}"] & \text{void}
\end{tikzcd}$$

But cows don't want any old mammal food, they want cow food, e.g. hay. Even multimethods won't help us here. The problem is that only cows know what they eat. There is a type dependency. We really need:

$$\begin{tikzcd}\text{Mammal} \times \text{food(Mammal)} \arrow[r, "\text{eat}"] \arrow[d] & \text{void} \arrow[d] \\ \text{Cow} \times \text{food(Cow)} \arrow[r, "\text{eat}"] & \text{void}\end{tikzcd}$$

where *food* is a type function. We are not aware of any existing object-oriented languages that support the ability to include type functions as a component of our abstractions. We tried for many years to come up with an object-oriented way of defining iterators, integers, or sequences. Everyone said it should be possible with object-orientation. Then we realized that the problem arose from the attempt to decompose things in an object-oriented manner. (Nowadays, object-orientation is equated with "good". We are not against good. When we refer to object-orientation we refer to existing languages and the specific techniques defined by proponents in certain books – we are not criticizing "what could be".) Such a strategy cannot work except in the case of single argument operations – the paradigm breaks down for general binary operations.

Today systems tend to be built using a number of different languages. One rarely sees activities such as scripting carried out in a language such as C++. We don't believe that there is any good reason why a single well-designed programming language couldn't support all programming tasks. Sadly there is not much interesting activity in programming language research nowadays. Even at the well-known POPL conference people are working on theoretical languages for which there are no programmers. A programming language must have users – graduate students writing type checkers for a language do not count. We want to be able to state everything we know about our program in our programming language. For example we would like to be able to state that an operation is commutative. Our ideal language would not have //-style comments. We want comments to be structured in a formal mathematical notation, and we them to be semantically checked by the compiler (to the extent that this is possible).

In C++ we are not even really faking it. We write *copy* with the words with *InputIterator*, *OutputIterator*. But we never define these things. We could globally replace the first with the word "Apple" and the second with the word "Orange" without changing the behavior of the program. These words only carry meaning in our mind – iterators are figments of our imagination in C++. This a huge problem. Consider the following situation.:

```
template <typename InputIterator,
          typename OutputIterator>
OutputIterator buggy_copy(InputIterator first,
                          InputIterator last,
                          OutputIterator result) {
  while (first < last) *result++=*last++;
  return result;
}
```

This will compile. It may even run for test cases that happen to use types for which < is be defined. In fact, this event is quite likely since people don't usually bother to test their algorithms against a minimal model. Instead they try only something like *buggy_copy<int*>*. Eventually, the person who wrote the above code will leave the company to work for Google. Another programmer will come along and instantiate *buggy_copy* for a list iterator (or some other iterator that is not Random Access). They will get an indecipherable thirty page long error message. STL works provided that it is used correctly. Otherwise the error messages are awful. The whole thing is done with smoke and mirrors.

There are some important techniques that we can use to test our algorithms against minimal models. One such technique is to create minimal model adaptors.

EXERCISE 2.1. Write adaptors for each iterator category that will help algorithm authors test the correctness of their iterator category requirements. For example, *ForwardIteratorAdaptor<I>* should have a restricted interface that only exposes the operations required to be provided by all Forward Iterators. Instantiate the faulty version of *buggy_copy* on *ForwardIteratorAdaptor<int*>*. InputIteratorAdaptor is far from trivial since ideally we would like its use to cause a compilation error when supplied to a two pass algorithm. But it is an open question whether it is even possible to for a class to detect two-pass algorithms at compile time. OutputIteratorAdaptor is hard, but not as hard. The others are straightforward.

EXERCISE 2.2. To get an idea of how the use of these adaptors impacts performance repeat the performance measurements from Question 2.3 replacing the types that you were using with their restricted/adapted versions. Do not alter any of the flavors of your *copy* algorithm.

The STL flaws that we encounter in C++ would be alleviated in a language that supported concepts. In particular, concepts allow us to declare that an iterator is something that comes equipped with some operations and a collection of type functions (associated types) all of which obey certain semantic constraints. We will have more to say about concepts later.

We will move on to more interesting algorithms momentarily, but first we note a few important facts about *copy*. One problem is that the version that we have been working with requires that *result* is not between *first* and *last*. When the ranges overlap, *copy* may fail (for our unrolled version). The case where the ranges could overlap is addressed by another function called *copy_backwards* (implemented using the strategy indicated by its name).

EXERCISE 2.3. Implement *copy_backwards*. Pay attention to the interface, in particular to the allowable iterator categories.

EXERCISE 2.4. Implement *find_if* (without looking even at the interface or implementation of the STL *find_if*) and *count_if*. Reasoning about what makes a good interface is important. The return type of *count_if* must be the *difference_type* of the iterator. This exercise will force you to learn *iterator_traits* and the non-intuitive C++ syntax needed to simulate type functions.

Part 2

Permutation Algorithms

CHAPTER 3

Introduction to Permutation Algorithms

Most interesting algorithms involving iterators come from a class of problems concerning permutations. We often need to reorder a group of elements. Let us recall a few definitions. A permutation is a one-to-one mapping of a finite set onto itself. A *cyclic permutation* or a *cycle* is a subset of a permutation whose elements trade places with one another. An element that remains unmoved by a permutation is called a *fixed point*. There can be at most n fixed points, in which case the permutation would be *trivial* (i.e. the identity permutation).

There are three distinct kinds of permutation operations. The first kind is index based. Under an *index based permutation operation* the destination of an element depends only on the element's original position, not on its value. The second kind of permutation operation takes an element's value into account when determining its destination. The typical example of such a *predicate based permutation operation* is *partition*. The problem that *partition* solves is to place the "good" elements before the "bad" elements, according to some unary predicate. The third kind of permutation operation compares two values to determine an element's destination. While *partition* uses a unary predicate, *comparison based permutation algorithms*, like *sort*, employ a binary comparison operation.

Remarkably, there are relatively few interesting algorithms within these three classes. The job of mathematicians is not to derive dozens of theorems, but to discover the few interesting ones. By the same token, the job of a computer scientist is not to produce a large number of algorithms, but to discover the important ones. We must act as filters, separating the good from the bad.

Permutation algorithms fall into three different classes. For permutations of n elements, algorithms that use at most $\log n$ additional storage are known as *in-place algorithms*. At the other end of the spectrum, we have algorithms that may use as much storage as needed. In practice this seldom amounts to more than n extra elements – $n/2$ often suffices. The third class, memory-adaptive algorithms, tends not to be described in the literature, but we view it as the most important. It is rarely the case that we have either as much memory as we like, or a logarithmic amount. Instead, there is usually *some* extra memory available. *Memory-adaptive algorithms* make use of whatever extra memory they are given to improve performance. For large classes of algorithms, a small increase in available storage (e.g. 10% or lower) allows for a significant improvement in running time.

In the chapters of this part we will study index-based permutation operations, predicate based permutation operations and comparison based permutation operations. First we prove a couple of useful theorems.

THEOREM 3.1. *Any permutation, p, on a finite set X can be decomposed into a product of disjoint cycles. Except for changes in the order of the cycles, this decomposition is unique.*

PROOF. We build our product of cycles as follows: for the first cycle we begin with an arbitrary element x of X and carry it to its image under the permutation (i.e. to $p(x)$), which in turn gets carried to its image, and so on. Since X is finite, we may continue this process until we first return to an element that has already been visited. Such an element must in fact be x, thereby completing our cycle, for x is the only element that has not already appeared as an image, and our function, being a restriction of the original permutation, must remain one-to-one. After the first cycle is complete, we generate more cycles by repeating the above process until there no longer remain elements of X which have not yet participated in a cycle. Finally, since each cycle is uniquely determined by the permutation, we note that our decomposition is unique (up to order of the cycles). □

THEOREM 3.2. *The total number of assignments needed to implement a permutation on a set of n elements is* ($n-$ *the number of trivial cycles* + *the number of non-trivial cycles*).

PROOF. Suppose that we have a permutation of length n with i non-trivial cycles and t trivial cycles. Notice that, in our machine model, any (sequence of) assignments will "destroy" at least one value. So, to implement a non-trivial cycle of length k we will need to perform (at least) $k + 1$ assignments, since permutations must remain 1 to 1. To see that $k + 1$ assignments are sufficient, consider the following procedure: first copy the “last” value in (some representation of) the cycle to a temporary location. Then copy the remaining $k - 1$ values (proceeding from right to left) to their final destinations. Finally copy the temporary value to its final destination. The total number of assignments required will be: one per element of each non-trivial cycle + one extra per non-trivial cycle (trivial cycles don’t contribute to the total number of assignments). In other words $(n - t) + i$. □

See the appendix for an alternative proof.

CHAPTER 4

Position Based Permutation Operations

In this chapter we discuss the three most important index based permutation operations: *reverse*, *rotate*, and *random_shuffle*. *reverse* is fairly straightforward, mapping 1 2 3 to 3 2 1. The *rotate* operation, though extremely useful, is unknown to many programmers. Rotating 1 2 3 around 3 results in 3 1 2. That is, rotate swaps (possibly unequal-sized) ranges. Interestingly, there are three different algorithms for the *rotate* operation, each one being suited for use with a different iterator category. We finish the chapter with *random_shuffle*.

4.1. reverse

4.1.1. reverse for Bidirectional and Random Access Iterators.

In order to reverse a range we need the ability to swap elements. We begin by implementing *swap*

```
template <typename T> // T models Regular Type
inline void swap(T& a, T& b) {
  T tmp=a;
  a=b;
  b=tmp;
}
```

We don't return anything because we didn't compute anything unknown to the caller. Observe that the cost of performing a *swap* is the same as the cost of three assignment statements.

It is possible to write swap using *no* extra storage at all. Such an implementation is not useful nowadays, since it runs more slowly then the version above. Nonetheless it is an interesting technique:

```
a^=b; // a == a0^b0 b == b0
b^=a; // a == a0^b0 b == a0
a^=b; // a == b0     b == a0
```

Nowadays the above implementation is not useful because it runs more slowly then the previous version. Given *swap* we can implement *reverse* for bidirectional iterators:

```
template <typename I> // I models Bidirectional Iterator
void reverse(I first, I last) {
  while(first != last && first != --last){
    swap(*first, *last);
    ++first;
  }
}
```

This code requires bidirectional iterators because it uses of the decrement operator. Could we make it a little bit faster? We are doing two comparisons per iteration. We can reduce the number of comparisons using our usual technique: we create *reverse_n*. This allows us to unroll the loop to wring a little bit of extra performance out of the algorithm when n is known.

```
template <typename I> /* I models Bidirectional Iterator */
void reverse_n(I first,
               I last,
               /* We retain this parameter since it may be expensive
                  compute in the Bidirectional Iterator case. */
               typename std::iterator_traits<I>::difference_type n) {
  while(n>1){
    swap(*first, *--last);
    ++first;
    n-=2;
  }
}
```

Now we can implement *reverse* for random access iterators in terms of *reverse_n*.

```
template <typename I> /* I models Random Access Iterator */
void reverse(I first,
             I last) {
  reverse_n(first, last, last - first);
}
```

EXERCISE 4.1. Implement a version of *reverse* that dispatches to *reverse_n* for random access iterators and to the previous version for bidirectional iterators.

On some platforms, for bidirectional iterators, it turns out to be faster to call *distance* followed by *reverse_n*.

4.1.2. Sketch of Reverse for Forward Iterators. Now we move on to *reverse* for forward iterators. Before we explain how the algorithm works, we need to establish the meaning of *reverse*. A permutation, p, is said to reverse a range of elements if and only if, for all a and b in the range, if a precedes b then $p(b)$ precedes $p(a)$. Now, suppose that we wish to reverse the sequence of elements

$$\begin{array}{l} f \qquad\quad l \\ \boxed{abcde} \end{array}$$

We first select a position such as m, the point between a and b:

$$\begin{array}{|c|c|} \multicolumn{1}{l}{f} & \multicolumn{1}{l}{m \qquad l} \\ \hline a & bcde \\ \hline \end{array}$$

If we reverse the elements in $[f,m)$ then reverse the elements in $[m,l)$ we obtain

$$\begin{array}{|c|c|} \multicolumn{1}{l}{f} & \multicolumn{1}{l}{m \qquad l} \\ \hline a & edcb \\ \hline \end{array}$$

Now we rotate around m (i.e. we swap the two unequal-sized ranges $[f,m)$ and $[m,l)$) we get

$$\begin{array}{|c|c|} \multicolumn{1}{l}{f} & \multicolumn{1}{l}{m' \ l} \\ \hline edcb & a \\ \hline \end{array}$$

Clearly this procedure has had the effect of reversing the range. We try again, this time selecting a different *m*

$$\begin{array}{l|l|}
f & m \quad l \\
\hline
\multicolumn{1}{|l|}{abc} & de \\
\hline
\end{array}$$

Reversing each range gives

$$\begin{array}{l|l|}
f & m \quad l \\
\hline
\multicolumn{1}{|l|}{cba} & ed \\
\hline
\end{array}$$

and after the final rotation we once again see that the elements have been reversed:

$$\begin{array}{l|l|}
f & m' \quad l \\
\hline
\multicolumn{1}{|l|}{ed} & cba \\
\hline
\end{array}$$

To see that this works in general, consider a pair the elements x, y in a range to be reversed. If after choosing m, x and y are in the same sub-range then their relative order will be reversed when their sub-range is reversed. Their relative order will be unaffected by the *rotate*. On the other hand, if x and y are in different sub-ranges then their relative order will be unchanged when their sub-range is reversed, but their order will be reversed by the *rotate*. In either case the relative order of x and y is flipped, so the permutation described above really does constitute a reverse algorithm. Later on we will show how a dual algorithm is possible–we implement a *rotate* algorithm in terms of *reverse*. Starting from the key idea of the algorithm we begin by writing:

```
I m = select(first, last);
reverse(first, middle);
reverse(middle, last);
rotate(first, middle, last);
```

Even leaving aside the fact that we haven't defined the *select* procedure yet, the code above is incomplete, since we will have a non-terminating recursion. We need to insert a termination condition ahead of the above code:

```
if(f==l || successor(f)=l) return;
```

4.1.3. Towards Efficient Divide and Conquer Algorithms. We actually produced a whole family of *reverse* algorithms, parameterized by the selected rotation point. Each algorithm has a different complexity. The cost of the reverse will be essentially equal to the sum of the costs of all the rotates. We will see later that the complexity of rotating is linear in the number of elements. Consider the *reverse* algorithm obtained by using a *select* that always chooses the point one past the start for the rotation point. We depict its complexity in Figure 1.

From the Figure we see that we will have to rotate: a range of size n at the top level of the recursion, a range of size 1 and a range of size $n-1$ at the second level of recursion, and so on. Adding up these costs we get the total cost for *reverse*, $\mathrm{rev}(n)$ as

$$\mathrm{rev}(n) \approx (n + (n-1) + \ldots + 1) + ((n-1) \cdot 1) = \frac{n(n+1)}{2} + n - 1$$

So

$$O(\mathrm{rev}(n)) = O(\frac{n(n+1)}{2} + n - 1) = O(n^2).$$

This analysis is not only about the performance of a particular *reverse* algorithm. In general it applies whenever we are using a divide and conquer strategy with an operation of linear complexity, with a strategy of splitting into two sub-problems by picking off a

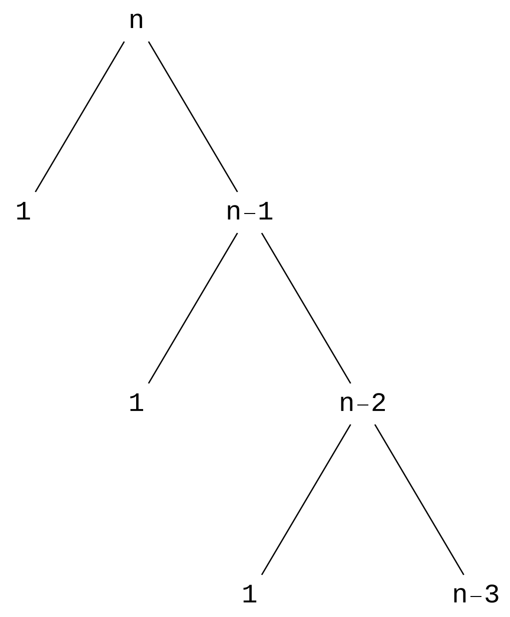

FIGURE 1.

single element from the beginning. In such cases we will see quadratic performance. The remedy is to use split into sub-problems in the middle. The complexity of this algorithm is represented in Figure 2 (for simplicity we consider the case when n is a power of 2). The complexity is now on the order of

$$O\left(\sum_{i=0}^{\lg n} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{\lg n} n\right) = O(n \cdot (1 + \lg n)) = O(n \lg n).$$

So, by switching our partitioning strategy from splitting at the first element to splitting in the middle we improve the complexity from $O(n^2)$ to $O(n \lg n)$.

We can make some further improvements. When n is even, instead of using *rotate*, we can use the cheaper *swap_ranges* (which swaps two equal-sized, possibly non-contiguous ranges).

EXERCISE 4.2. Implement *I2 swap_ranges(I1 first, I1 last, I2 first2).*

EXERCISE 4.3. Implement *std::pair<I1, I2> swap_ranges_n(I1 first, I2 first2, Integer n).*

Let's write some code for the forward iterator *reverse* algorithm based on the ideas that we have been discussing.

```
template <typename I> // I models Forward Iterator
void reverse(I first, I last) {
  typename std::iterator_traits<I>::difference_type
  n = std::distance(first, last);
  /* The above line has linear complexity.
     We will address this below. */
```

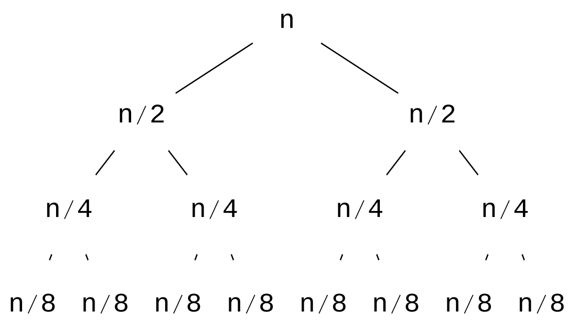

FIGURE 2.

```
    if(n<2) return;
    I middle = first;
    std::advance(middle, n/2);
    /* Advancing by n/2 doesn't quite work since n might
       be odd. But luckily, in that case the element
       in the middle doesn't need to be swapped.
       We patch for the odd case below.
    */
    I first2 = middle;
    if(n%2) ++first2;
    std::swap_ranges(first, middle, first2);
    reverse(first, middle);
    reverse(first2, last);
}
```

This version of *reverse* is no longer quadratic, it has complexity $O(n \log n)$. This is good, but there is still room for improvement. We do three traversals: once during the call to *distance*, again when calling *advance*, and the third time when calling *swap_ranges*. The third traversal dominates the runtime cost, and we will not be able to make it go away. But we can still enjoy some smaller gains by getting rid of the first two traversals. The technique is to extend the *reverse* function to return the information that we previously obtained by traversing.

It is easiest to see how to do this in the context of *reverse_n*. Here we avoid the call to distance since n is already given. We extend *reverse_n* to return an iterator referring to the end of the reversed range. Note that it doesn't matter whether we call *swap_ranges* before or after we do the reverse. By moving the calls to *reverse_n* ahead of the call to *swap_ranges* we can make use of the iterator returned by *reverse_n* to get rid of the call to *advance*.

Finally, we need to add code to return the end of the reversed range. It is trivial to figure out what to return in case $n < 2$. In addition, while the first call to *reverse_n* gives us *middle*, the second call to *reverse_n* gives us *result*. In code we have:

```
template <typename I, // I models Forward Iterator
          typename J> // J models Integer
```

```
I reverse_n( I first , J n )
{
  if (n == 0) return first;
  if (n == 1) return ++first;

  I middle = reverse_n(first, n/2); /* this eliminates the need
                                       for advance */
  if (n%2 == 1) ++middle;
  I result = reverse_n(middle, n/2);
  swap_ranges_n(first, middle, n/2);
  return result;
}
```

We find this code amazing because, at first glance, it is not at all clear where any work is being done! This implementation is in-place: it uses $\lg n$ additional storage for the stack during the recursion. It should be clear how to produce *reverse* from this *reverse_n*.

We saw in Theorem 3.2 that the minimum number of assignments needed to implement a permutation is $n + \text{ntc} - \text{tc}$ where ntc is the number of non-trivial cycles, and tc is the number of trivial cycles. For *reverse* this amounts to $n + n/2 - n\%2$. This tells us that *reverse* is the most expensive permutation to execute, because we have the largest possible number of non-trivial cycles. In particular, there is no point in trying to make our implementation of *reverse* (for bidirectional or forward iterators) any more efficient.

4.1.4. Memory Adaptive Algorithms. Unfortunately, it turns out that in practice the forward iterator version of *reverse* is not very useful. The typical example of a container that can only provide forward iterators is a singly-linked list. But for such lists we would usually implement *reverse* by directly manipulating the successors of the iterators. Nonetheless, the issues that we addressed in the *reverse* example are illustrative of those that appear in the more general class of top-down recursive algorithms.

Usually, when we provide a top-down recursive algorithm for an operation, we should also provide a memory adaptive version. During the recursion, as soon as the sub-problem becomes smaller than the auxiliary buffer, we can speed things up. To do so, we first copy the all of the remaining elements into the buffer. Assuming that the buffer provides at least bidirectional iterators, we can efficiently copy the elements back into place in reverse order, using the *reverse_copy* algorithm. We give the code for *reverse_copy* followed by the code for *reverse_n_adaptive*.

```
template <typename B, // B models Bidirectional iterator
          typename O> // O models Output Iterator
O reverse_copy(B first, B last, O result)
{
  while(first != last){
    --last;
    *result = *last;
    ++result;
  }
  return result;
}
```

Observe the asymmetry in the arguments to *reverse_copy* above. As an alternative to providing two bidirectional iterators to delimit the source range, we can provide a single input iterator referring to the beginning of the source range, along with a pair of bidirectional iterators that determine the destination range. In each case, to implement the algorithm, we need at least one of the ranges to provide bidirectional iterators so that we can walk backwards through that. Despite the fact that it is a bit too cute, the best name that we can think of for this algorithm is *copy_reverse*:

```
template <typename I, // I models Input Iterator
          typename B> // B models Bidirectional Iterator
I copy_reverse(I first, B result_first, B result_last)
{
  while(result_first != result_last){
    --result_last;
    *first = *result_last;
    ++first;
  }
  return first;
}
```

EXERCISE 4.4. Implement *reverse_copy_n*.

Here is the code for *reverse_n_adaptive*:

```
template <typename I,  // I models Forward Iterator
          typename I2> // I models Bidirectional Iterator
I reverse_n_adaptive(I first,
                     typename std::iterator_traits<I>::difference_type
                        n,
                     I2 buffer,
                     typename std::iterator_traits<I>::difference_type
                        buffer_length)
{
  if(n==0) return first;
  if(n==1) return ++first;
  if(n <= buffer_length) {
    return reverse_copy(buffer, copy_n(first, n, buffer), first);
  }
  I middle = reverse_n_adaptive(first, n/2, buffer, buffer_length);
  if(n%2==1) ++middle;
  I last = reverse_n_adaptive(middle, n/2, buffer, buffer_length);
  swap_ranges_n(first, middle, n/2);
  return last;
}
```

STL provides *get_temporary_buffer* which tries to return the largest available temporary buffer. It would be nice of this was useful in trying to write memory adaptive algorithms, but unfortunately it is implemented using *malloc*. System interfaces have a tendency to hide information that we need to know such as: the cache size, the size of physical memory, or the available temporary buffer size. Information hiding has gotten to the point where it is difficult to program effectively.

Adaptive algorithms are commonly useful, but somehow they are overlooked considered by the theoreticians, who employ an all-or-nothing approach. Even as little as 1% of additional memory can have dramatic effects. As we saw in the *reverse_adaptive_n*, the bottom of the recursion tree is where all the work gets done. But this is exactly where a small buffer of extra storage can have the most benefit.

EXERCISE 4.5. Graph the performance of the adaptive algorithm for different buffer sizes. Use an array, together with forward iterator adaptor (fairly large, say 100,000) and see how the performance varies when 1%, 10%, and 50% additional storage is available.

4.1.5. reverse for Node Based Iterators. Consider the following linked list:

$$\begin{array}{|c|}\hline i\\\hline 1\\\hline\end{array} \longrightarrow \begin{array}{|c|}\hline j\\\hline 2\\\hline\end{array} \longrightarrow \begin{array}{|c|}\hline k\\\hline 3\\\hline\end{array} \longrightarrow \begin{array}{|c|}\hline l\\\hline 4\\\hline\end{array} \longrightarrow \begin{array}{|c|}\hline end\\\hline \emptyset\\\hline\end{array}$$

Here we have four nodes, *i*, *j*, *k*, and *l* with **i==1, *j==2, *k==3, *l==4* and *successor(i)=j, successor(j)==k, successor(k)==l, successor(l)==end. end* is not dereferenceable, it exists simply for convenience to represent an off-the-end node. A natural way to reverse this data structure is to change the successors of each node so that we end up with:

$$\begin{array}{|c|}\hline end\\\hline \emptyset\\\hline\end{array} \longleftarrow \begin{array}{|c|}\hline i\\\hline 1\\\hline\end{array} \longleftarrow \begin{array}{|c|}\hline j\\\hline 2\\\hline\end{array} \longleftarrow \begin{array}{|c|}\hline k\\\hline 3\\\hline\end{array} \longleftarrow \begin{array}{|c|}\hline l\\\hline 4\\\hline\end{array}$$

We didn't change or copy any of the values, we still have **i==1, *j==2, *k==3, *l==4.* But now *successor(i)=end, successor(j)==i, successor(k)==j, successor(l)==k.* In general, when we are working with linked data structures (node-based structures) we can implement *reverse* without copying any values. The iterators that we have been discussing so far do not allow for successor changes. We need an additional operation which we will call *set_successor*. We will call (forward or stronger) iterators that support this operation *node-based iterators*, or *node iterators*.

In C++ there is no uniform and efficient notion of an empty list, so we borrow from lisp the idea of using the *reverse_append(first, last, result)* operation which reverses the range $[first, last)$ and then appends *result*. This algorithm eliminates the need to deal with empty lists. Then we can implement *reverse(first, last)* for node iterators as *reverse_append(first, last, last)*. Here is the code for *reverse_append*:

```
template <typename T> // T models Node Iterator
T reverse_append(T first, T last, T result) {
  while(first != last) {
    T next = successor(first);
 /* We must save first's successor as we will not be
    able to discover it after calling set_successor. */
    set_successor(first, result);
    result = first;
    first = next;
  }
  return result;
}
```

Notice that we save the *first*'s old successor before setting its new successor in order to retain the ability to properly navigate the list. We could simplify *reverse_append* by making *set_successor* return a value, but there are several possible choices with no clear winner (e.g. we might return the old value of the successor, or we might return the first argument).

EXERCISE 4.6. Implement a simple node iterator and use it to test your implementation of *reverse_append*. You do not need to implement a container.

We use the name *reverse_nodes* to distinguish this operation from *reverse*, since an operation that changes the successor of a node has markedly different semantics from an operation that changes the value.

While node iterators must be at least forward iterators, it is quite possible to have bidirectional node iterators. Node operation support is orthogonal to the notion of iterator categories. In general, we cannot form a simple hierarchical taxonomy of concepts.

4.2. rotate

The *rotate* operation is very useful since, in many algorithms, it is necessary to swap two ranges of unequal size. We saw one use of *rotate* when working towards a *reverse* algorithm for forward iterators (although we optimized away the call to *rotate*, in favor of a call to *swap_ranges*, during recursion elimination).

rotate is also useful when implementing data structure operations. Consider the problem of inserting elements from a range [*first, last*) into a *vector* at *position*. The standard C++ library provides this functionality as the member function template *vector::insert(I1 position, I2 first, I2 last)*. When *I2* is only an input iterator we don't know the size of the range [*first, last*), i.e. we don't know in advance how many elements are to be inserted. Naive algorithms insert elements into the *vector* one at a time. But then, for each of the *distance(first, last)* inserts, *distance(position, end())* elements must be moved out of the way (*end()* denotes the end of the *vector*). If we add to this the cost of copying each element of [*first, last*) into position, the total number of assignments required by this algorithm is:

$$distance(first, last) * (distance(position, end()) + 1)$$

There are even some versions of the standard C++ library that ship using this quadratic algorithm. A different algorithm, employing *rotate*, only requires one move of $[position, end())$. We begin by appending all of the elements in $[first, last)$ to the *vector*. Then we *rotate* the new elements into place. Roughly speaking, this reduces the number of assignments required from the (quadratic) product of the lengths of the two ranges, to the (linear) sum of the lengths. More precisely, if we represent the (linear) cost of rotating *n* elements as *rot(n)*, the number of assignments needed is:

$$distance(first, last) + rot(distance(first, last) + distance(position, end()))$$

We will have more to say about the cost of rotating later on. In C++ our algorithm looks like:

```
template <typename T,
          typename A>
template <typename I>
void vector<T,A>::insert(iterator position, I first, I last)
{
/* position may not remain valid after an insertion occurs. We save
   its offset so that we can later construct an iterator that refers
   to the beginning of the range to be rotated. We do the same for the
   original end so that we can later determine the rotation point. */

  difference_type  position_offset = position - begin();
  difference_type  end_offset = end() - begin();
```

```
    while(first != last)
      push_back(*first++);
    std::rotate(begin() + position_offset,
                begin() + end_offset,
                end());
  }
```

In this chapter we derive, analyze, and optimize three separate algorithms for the *rotate* operation: one algorithm for forward iterators, one for bidirectional iterators, and one for random access iterators.

Before we explain how the algorithms work, we need to be clear on the meaning of the *rotate* operation. $rotate(If, Im, Il)$ must swap the two adjacent ranges, $[f, m)$ and $[m, l)$. If $[f, m)$ has size a and $[m, l)$ has size b then the $rotate(f, m, l)$ operation is defined to have the same effect as a permutation that moves the elements in $[f, m)$ to the right by a positions, and moves the elements in $[m, l)$ to the left by b positions. We will revisit the return value later.

4.2.1. rotate for Bidirectional Iterators. At first, the problem of rotating appears to be quite difficult, as we must swap ranges of unequal size. Since *rotate* helped us discover how to implement *reverse*, we ask in turn whether *reverse* might help us to see how to implement *rotate*. Suppose that we wish to *rotate* the range $[f, l)$ containing the values 1 through 8 around the iterator m, which refers to 6, as in:

(4.1)
$$\begin{array}{|c|c|} \multicolumn{1}{l}{f} & \multicolumn{1}{l}{m \quad\ l} \\ \hline 12345 & 678 \\ \hline \end{array}$$

Reversing the entire range with $reverse(f, l)$ yields:

$$\begin{array}{|c|c|} \multicolumn{1}{l}{f} & \multicolumn{1}{l}{m' \qquad l} \\ \hline 876 & 54321 \\ \hline \end{array}$$

This puts each range in the correct location, though the elements within each range still need to be reversed. In the case of forward iterators, it may not be efficient to determine the image of m, shown above as m', under the rotation. So it is most convenient if we begin by reversing each of the small ranges in Equation 4.1. That is, we start by calling $reverse(f, n)$ and $reverse(m, l)$ to obtain:

$$\begin{array}{|c|c|} \multicolumn{1}{l}{f} & \multicolumn{1}{l}{m \quad\ l} \\ \hline 54321 & 876 \\ \hline \end{array}$$

Now, we reverse the entire range with a call to $reverse(f, l)$ to arrive at:

$$\begin{array}{|c|c|} \multicolumn{1}{l}{f} & \multicolumn{1}{l}{m' \qquad l} \\ \hline 678 & 12345 \\ \hline \end{array}$$

We call this the "three reverse" rotate algorithm. What is its complexity (in terms of the number of assignments required)? We know that it takes $n/2$ swaps to reverse a range of n elements. In the three reverse algorithm we reverse the entire range once, at a cost of $n/2$ swaps. We also reverse two smaller ranges, the sum of whose sizes is n, which requires another $n/2$ swaps. So the total cost is n swaps, or $3n$ assignments.

Is a *rotate* algorithm that requires $3n$ assignments good or bad? Any non-trivial rotation will consist of a permutation that moves every element–there will be no trivial cycles. By Theorem 3.2 the minimum number of assignments required will therefore be n + the number of cycles. This is substantially lower than $3n$ assignments. In the section on *rotate* for forward iterators we will show that, remarkably, the number of cycles is the gcd of the sizes of the two ranges! Then, in the section on *rotate* for random access iterators, we will describe an algorithm that requires only n + gcd assignments–the minimum possible number. Nevertheless, we will later see that sometimes three reverses is the best algorithm to use.

A well-designed *rotate* needs to return the image of m, we'll call it m'. For, we have seen that the caller might not be able to determine this value without traversing. Furthermore, this allows us to undo the effects of a *rotate*. In other words, *rotate*(f, *rotate*(f, m, l), l) is the identity permutation. Of course we want to return m' without doing any extra work. This is a constant challenge–we want to return the useful information without paying a performance penalty.

We modify the algorithm to return m' as follows. We begin as before by reversing the ranges $[f, m)$ and $[m, l)$. The third reverse, the one which reverses the entire range $[f, l)$, is normally implemented as:

```
while(f != l && f != --l){
  swap(*f, *l);
  ++f;
}
```

The key to efficiently returning m' is the observation that, in the course of reversing, when one iterator hits m the other iterator will be at m'. So we change our the third reverse, beginning with:

```
while(f != m && m != l){
  --l;
  swap(*f, *l);
  ++f;
}
```

That is, we reverse until one of the iterators hits the sentinel, m. We then: save m', complete the reverse of the middle range, and finally return m'. The algorithm for reversing until we hit a sentinel is useful in its own right, so we create a new interface, *reverse_until*. We commonly create new interfaces, since algorithms often come in clusters of useful variants. *reverse_until(I f, I m, I l)* will reverse the range $[f, l)$ as discussed above until one of the iterators hits m. We might be tempted to have it simply return m'. But then we would be throwing away a piece of useful information. The caller doesn't know which of the two ranges, $[m, m')$ or $[m', m)$ is the valid one. So we require that *reverse_until* returns, in a *pair<I, I>*, the valid range of elements that haven't yet been reversed. The caller can determine m' by testing whether m is equal to the first element of the pair–if not, then the pair's first element contains m', otherwise the second element does. Here is the implementation in C++:

```
template <typename I> // I models Bidirectional Iterator
std::pair<I,I>
reverse_until(I first, I middle, I last)
{
  while (first != middle && middle != last) {
    --last;
    swap(*first, *last);
    ++first;
  }
```

```
    return std::make_pair(first, last);
}
```

Now we can implement our improved *rotate* as:

```
template <typename I> // I models Bidirectional Iterator
I rotate(I first, I middle, I last) {
  if(first == middle) return last;
  if(middle == last) return first;

  reverse(first, middle);
  reverse(middle, last);

  std::pair<I, I> new_middle =
    reverse_until(first, middle, last);
  reverse(new_middle.first, new_middle.second);

  if (middle != new_middle.first) {
    return new_middle.first;
  } else {
    return new_middle.second;
  }
}
```

4.2.2. rotate for Forward Iterators. We wish to exchange the ranges $[f, m)$ and $[m, l)$, of sizes a and b respectively.

$$\begin{array}{|c|c|} \multicolumn{2}{l}{f\quad m\quad\quad l} \\ \hline a & \text{b} \\ \hline \end{array}$$

This time we are given only forward iterators. If one of the ranges is empty then there is no work to do. When $a = b$, we can call *swap_ranges* on the two equal sized ranges, and since $m' = m$ we can return m. We must also define our algorithm in case $a > b$ or $a < b$. We proceed inductively, that is, we show how to reduce the problem of rotating the entire range to a problem involving a smaller rotation.

First we consider the case where $[f, m)$ of size a is smaller than $[m, l)$ of size b. Let m' be the iterator with $distance(m, m') = a$. In the diagrams below, each range is labeled with its size.

$$\begin{array}{|c|c|c|} \multicolumn{3}{l}{f\quad m\quad m'\quad\quad l} \\ \hline a & a & b-a \\ \hline \end{array} \tag{4.2}$$

The idea is to first swap the two blocks of size a, then to rotate the resulting second and third blocks. The first task is easily accomplished with a call to *swap_ranges*. The task of swapping the two remaining blocks (of smaller total size) is accomplished with another call to *rotate* since the second and third ranges are not guaranteed to be the same size.

We elaborate the above idea: We begin by swapping the (equal sized) ranges $[f, m)$ and $[m, m')$:

$$\begin{array}{|c|c|c|} \multicolumn{3}{l}{f\quad m\quad m'\quad\quad l} \\ \hline a & a & b-a \\ \hline \end{array} \tag{4.3}$$

The swap has moved each element originally from $[m, m')$ to the left by a positions. By the definition of *rotate* the elements now in $[f, m)$ are in their final positions. The swap has moved the elements originally residing in $[f, m)$ to the right by a positions to $[m, m')$. *rotate* requires that these elements ultimately move to the right by b positions, so they must still move $b - a$ positions to the right. The elements in $[m', l)$ have not moved at all, so they have yet to move b positions to the left. All of the necessary remaining moves can be accomplished if we exchange the ranges $[m, m')$ and $[m', l)$. That is, we have reduced the problem of exchanging ranges of sizes a and b to the smaller problem of exchanging ranges of sizes a and $b - a$. Observe that our return value will be the same as that returned by "inner" rotation, *rotate(m, m', l)*.

The case where $a > b$ is handled similarly. We again begin by swapping two equal sized blocks followed by a call to *rotate* on a smaller range. Elaborating, here we let m' be the iterator with $distance(f, m') = b$ as pictured below:

$$\begin{array}{|c|c|c|}
\multicolumn{1}{l}{f} & \multicolumn{1}{l}{m'} & \multicolumn{1}{l}{m \quad l} \\ \hline
b & a-b & b \\ \hline
\end{array}$$

We begin as before by swapping the equal sized ranges, in this case $[f, m')$ and $[m, l)$. The swap moves each element originally from $[m, l)$ to the left by a positions, to their final destinations in $[f, m')$. An important consequence is that here m' is the value that we must eventually return. The swap has also moved the elements originally from $[f, m')$ to the right by a positions. *rotate* requires that these elements ultimately move to the right by b positions so they must still be moved $a - b$ positions back to the left. The elements of $[m', m)$ have not moved at all, so they have yet to move b positions to the right. All of the necessary remaining moves can be accomplished if we exchange the ranges $[m', m)$ and $[m, l)$. That is, we have reduced the problem of exchanging ranges of sizes a and b to the problem of exchanging ranges of sizes $a - b$ and b. The final result will be:

$$\begin{array}{|c|c|c|}
\multicolumn{1}{l}{f} & \multicolumn{1}{l}{m'} & \multicolumn{1}{r}{l} \\ \hline
b & b & a-b \\ \hline
\end{array}$$

The forward iterator *rotate* algorithm was discovered by David Gries and Harlan Mills [**4**]. We will refer to it as the Gries-Mills algorithm.

4.2.2.1. *Analysis of Gries-Mills.* Let us determine the cost of the Gries-Mills algorithm, in terms of the number of assignments. In the case when $a < b$ we first swap the two ranges of size a. This costs a swaps and it moves exactly a elements to their final places. Then, in the sub-problem, we rotate smaller ranges of size a and $b - a$. In other words, we have reduced the size of our problem by a at a cost of a swaps. A similar analysis applies in the case when $a > b$, in which case we reduce the size of our problem by b at a cost of b swaps. In either case we reduce the size of our problem by $\min(a, b)$, at a cost of that many swaps. Just before the algorithm terminates, when the remaining ranges are of equal size d, we gain some efficiency, moving the final $2*d$ elements into place at a cost of only d swaps. Altogether, the cost in swaps of the *rotate* is $n -$ the size of the final (equal-sized) ranges.

What is be the size of the final ranges? Lets look a little bit more carefully at how the algorithm works. If $a < b$ we reduce the size of our problem by a and are left with the task of exchanging ranges of size a and $b-a$. If $a > b$ we reduce our problem to rotating ranges of size b and $a - b$. We continue in this manner until the remaining ranges have the same size. Writing this down in code we get:

```
int remaining_size(int a, int b)
{
  if(a < b) return remaining_size(a, b - a);
  if(b < a) return remaining_size(b, a - b);
  return a;
}
```

But this is exactly Euclid's algorithm for (subtractively) calculating the $gcd(a, b)$! So the final ranges have size $gcd(a, b)$ and the total cost of the algorithm is $n - gcd(a, b)$ swaps, or $3 * (n - gcd(a, b))$ assignments.

It is important to note that the number of cycles in the "complete" rotate permutation is the same as the number of cycles in the permutation corresponding to the "inner" rotate. We demonstrate this in the case where $a < b$ for $rotate(f, m, l)$ and $rotate(m, m', l)$ (see Equation 4.3). Consider what happens to an element at y in the range $[m, m')$. In the inner rotation, let x be the position of the element that gets moved to y, and let z be the position where the element at y gets moved, e.g. $(\ldots xyz \ldots)\ldots$. If we precede the inner rotation by swapping the ranges $[f, m)$ and $[m, m')$, it will have the following effects on the destinations of the elements: (i) the element at x will still be moved to y, (ii) the element at y will move to the left by a positions to, say, position y' and (iii) the element at y' will be moved to position z. The destinations of elements of the range $[m', l)$ will remain unchanged. In other words we transform the inner rotation into the complete rotation if, for each position y in $[m, m')$, we replace y in the cycle $(\ldots xyz \ldots)$ with yy' to get the resulting cycle $(\ldots xyy'z \ldots)$. This leaves the total number of cycles unchanged. The case where $a > b$ is handled similarly.

Another important lemma is that if positions x and y are in the same cycle then $x \equiv y \pmod{gcd(a, b)}$. To see this, consider what happens to an element at position i. We have:

$$i \to \begin{cases} i + b & \text{if } i + b < l \\ i - a & \text{otherwise} \end{cases} \tag{4.4}$$

Thus the distance between two positions in the same cycle will be of the form $d = ma + nb$ for some integers m, n. But $\gcd(a, b)$ divides all such numbers, so $d \equiv 0 \pmod{\gcd(a, b)}$ which completes the proof of the lemma. As a corollary, it follows that each position in a contiguous block of $gcd(a, b)$ elements is part of a different cycle. We will find this fact useful when implementing the random access iterator rotate algorithm.

4.2.2.2. *Implementation of Gries-Mills.* In the previous section we explained that our implementation strategy for swapping unequal sized ranges was to first swap two equal-sized ranges in order to order to reduce the original problem to a smaller one. We start from the heart of our implementation, then work our way out, patching as necessary. We could almost use $swap\_ranges(f, m, m)$ for the initial swap. Unfortunately, this will wreak havoc if $[f, m)$ is larger than $[m, l)$. Instead, we need a variant of *swap_ranges* that will swap as many elements of the two ranges as are present in the shorter range.

Once again, we create a new algorithm: *swap_ranges(I1 f1, I1 l1, I2 f2, I2 l2)*. What should it return? If the shorter range has length a we want to return the iterator, call it m, that is a past the beginning of the longer range. But, just as was the case with the *reverse_until* algorithm,there is another useful piece of information that we must return. It may not be efficient for the caller to determine which range the returned iterator is in. So here too we return a pair, this time of type *pair<I1, I2>*. The caller can determine m by testing whether $l1$ is equal to the first element of the pair–if not, then the pair's first element contains m, otherwise the second element does. Here is the code:

```
template <typename I1, // I1 models Forward Iterator
          typename I2> // I2 models Forward Iterator
std::pair<I1, I2>
swap_ranges(I1 f1, I1 l1, I2 f2, I2 l2)
{
    while (f1 != l1 && f2 != l2) {
      swap(*f1, *f2);
      ++f1;
      ++f2;
    }

    return std::make_pair(f1, f2);
}
```

We start implementing the Gries-Mills *rotate(I f, I m, I l)* algorithm from the inside out. The algorithm takes different actions according to the relative sizes, *a* and *b*, of the two ranges to be swapped, as indicated by the comments below.

```
    std::pair<I1, I2> tmp = swap_ranges(f, m, m, l);
    if(tmp.first == m && tmp.second == l)
      return; // a == b
    assert(tmp.first == m || tmp.second == l);
    /* Our assert comes from the definition of swap_ranges. */
     if(tmp.first == m) { //  a < b
      f = m;
     m = tmp.second;
     }else{ // b < a
      f = tmp.first;
    }
```

If we put this code inside of a **while**(**true**), a couple of issues remain. First, if one of the ranges is empty the loop won't terminate–the code above only works if neither range is empty. We can patch this by adding the precondition *assert(f != m && m != l)* . Again we have found a useful algorithmic fragment, which we will name *rotate_non_empty*. Another issue is that we are comparing *tmp.first* to *m* in two separate places. The standard way to fix this is to fuse the two conditions. Rewriting, we now have:

```
template <typename I> // I models Forward Iterator
void rotate_non_empty(I f, I m, I l)
{
    assert(f != m && m != l);
    while(true) {
      std::pair<I, I> tmp = swap_ranges(f, m, m, l);
      if(tmp.first == m){
        if(tmp.second == l) return; // a == b
        else { // a < b
          f = m;
         m = tmp.second;
        }
      }else{ // b < a
        f = tmp.first;
      }
    }
}
```

EXERCISE 4.1. We can still speed up the above code. As it stands now, we are calling *swap_ranges* from inside of a *while* loop. That is, we have a function call which results in two nested *while* loops. This is rather costly when we get down to the point of swapping many tiny ranges. Though *swap_ranges* was helpful in understanding how to develop the algorithm we now want to avoid the unnecessary overhead. Rewrite *rotate_non_empty* to use *swap* instead of *swap_ranges*.

Finally, we must patch the code to return m', the image of the dividing point, and we must handle the cases when one of the ranges is empty. How do we know when we have found m'? We discussed how to calculate the return value in Section . Recall that three cases occur for ranges $[f, m)$ and $[m, l)$, of sizes a, b respectively. If a equals b then we simply return m. If $a > b$ then after swapping ranges, m' is at $f + b$, which is the same as the return value of *swap_ranges(f, m, m, l)*. Finally, if $a < b$ we discovered that the return value was the same as that of the "inner" *rotate*. Below is an implementation of rotate that addresses these issues.

```
template <typename I> // I models Forward Iterator
void rotate(I f, I m, I l)
{
  if(f == m) return l;
  if(m == l) return f;
  while(true)
  {
    std::pair<I, I> tmp = swap_ranges(f, m, m, l);
    if(tmp.first == m){
      if(tmp.second == l) return m; // a == b
      else{ // a < b
        f = m;
        m = tmp.second;
      }
    }else{ // b < a
      rotate_non_empty(tmp.first, m, l);
      return tmp.first;
    }
  }
}
```

4.2.3. rotate for Random Access Iterators. We saw in the previous section that a rotation which swaps ranges of sizes a and b corresponds to a permutation with $\gcd(a, b)$ cycles. We also proved that any adjacent $\gcd(a, b)$ elements in the range belong to different cycles. In particular, the first gcd positions contain exactly one position from each cycle. Cycles can be implemented very efficiently for random access iterators: given a position i we can quickly find the next position in the cycle using Equation 4.4. For each position i in the first gcd elements our algorithm will efficiently perform the corresponding cycle...

Part 3

Appendices and Back Matter

APPENDIX A

Minimum Number of Assignments Needed to Implement a Cycle

Leslie Lamport contributed the material in this Appendix.

Let f be a permutation of $1, ..., n$. Assume there are n balls numbered 1 to n and an infinite set of boxes numbered $1, 2, ...$. Let a *configuration* C be an assignment of the balls to boxes that assigns each ball to exactly one box. The initial configuration C_{init} assigns each ball i to box i. The goal configuration C_{goal} assigns each ball i to box $f[i]$. A *move* consists of moving a ball from its current box to an empty box.

The object is to find the minimal number of moves that takes the initial configuration to the goal configuration.

Call box $f[i]$ the *destination* box of ball i.

Partition the balls into cycles, where a cycle is a minimal set S of balls such that if ball i is in S then ball $f[i]$ is in S. If S is a cycle and C is a configuration, define

```
dist(S, C) ≜
          IF every ball in S is in its destination box
             THEN 0
             ELSE the number of balls in S not in their destination boxes
                    +
                IF every destination box of a ball in S has a ball in S
                   THEN 1
                   ELSE 0
```

For any configuration C, define $movesToGo(C) \triangleq$ the sum over all cycles S of $dist(S, C)$

LEMMA A.1.

$$movesToGo(C) = 0 \iff C = C_{goal}$$

PROOF. Obvious. □

LEMMA A.2. *If a move takes configuration* C *to configuration* C', *then* $movesToGo(C') \geq movesToGo(C) - 1$.

PROOF. Let the move be a move of ball i.

(1) For all cycles S, if i is not in S then $dist(S, C') = dist(S, C)$
    Proof: pretty obvious.
(2) If i in S, then $dist(S, C') \geq dist(S, C) - 1$
    (a) CASE every ball in S is in its destination box
        Proof: Obvious, because in this case $dist(S, C) = 0$
    (b) CASE $\wedge$ not every ball in S is in its destination box
             $\wedge$ every destination box of a ball in S has a ball in S

Proof: In this case, i cannot be put into its destination box because that box already contains a ball in S. Therefore, it is put elsewhere. It's easy to see then that $dist(S, C') = dist(S, C) - 1$

(c) CASE $\wedge$ not every ball in S is in its destination box
$\wedge$ not every destination box of a ball in S has a ball in S
Proof: In this case, $dist(S, C)$ equals the number of balls in S not in their destination boxes, and moving i can decrease this value by at most 1 and hence can decrease $dist(S, C)$ by at most 1.

(d) QED
Proof: By propositional logic, Cases 2a, 2b, and 2c are exhaustive.

(3) QED
Proof: Follows trivially from 1 and 2 and the definition of movesToGo.

□

THEOREM A.3. *The number of moves to go from the initial configuration to the goal configuration is at least* $movesToGo(C_{init})$.

PROOF. Follows easily from Lemmas A.1 and A.2. □

Bibliography

1. John W. Backus, *Can programming be liberated from the von neumann style? a functional style and its algebra of programs.*, Commun. ACM **21** (1978), no. 8, 613–641.
2. G. A. Blaauw and Jr. F. P. Brooks, *Computer architecture: Concepts and evolution*, Addison Wesley, Reading, 1997.
3. Jr. F. P. Brooks, *The mythical man-month: Essays on software engineering, 20th anniversary edition*, Addison Wesley, Reading, 1995.
4. David Gries and Harlan Mills, *Swapping sections*, Tech. Report TR81-452, Cornell University Library, 1981.
5. J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach (3rd edition)*, Morgan Kaufmann, New York, 2003.
6. Kenneth E. Iverson, *Operators.*, ACM Trans. Program. Lang. Syst. **1** (1979), no. 2, 161–176.
7. ———, *Notation as a tool of thought.*, Commun. ACM **23** (1980), no. 8, 444–465.
8. Dave Musser and Alexander Stepanov, *Generic programming*, ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation (formerly SYMSAM, SYMSAC, EUROSAM, EUROCAL) (also sometimes in cooperation with the Symbolic and Algebraic Manipulation Groupe in Europe (SAME)), 1989.