Operation Counting Adaptors for STL

Gillmer J. Derge, Markus Kuhn, Josh Wilmes
{*dergeg,kuhnm,wilmesj*}*@rpi.edu*

December 3, 1998

Contents

Abstract

Algorithms are often analyzed in terms of their complexity, which is frequently defined in terms of execution time. In generic programming, however, discussions of time requirements are not always practical, since the time needed to perform an operation on one data type may vary quite a bit from that required for another. One way to obtain a more useful complexity measure for generic algorithms is to think in terms of the abstract operations performed on the data. Thus, for example, a sort algorithm could be measured by the number of comparisons and assignments it performs. This report presents a number of adaptors for the C++ Standard Template Library (STL) that allow the programmer to count the operations done on data types. The classes also provide a rich set of reporting and data analysis functions.

$Revision: 1.64 $ ($Date: 1998/12/03 16:36:23 $)

Chapter 1

User's Guide

1.1 Introduction

When evaluating algorithms, generic algorithms in particular, it can be useful to have a measure of the operations performed by the algorithms. For example, one of the differences between a typical Quicksort versus a heap sort is that the heap sort performs fewer comparisons but more assignments. Although the general rule of thumb is that Quicksort is more efficient than heap sort, if comparisons are relatively expensive computationally compared to assignments, then the Quicksort might be less desirable for that particular application.

If it were easy to obtain counts of operations, it would be far easier to trade off the relative advantages and disadvantages of different algorithms. The library presented in this report provides just such a facility.

The library implements a set of "counting adaptors." These adaptors redefine the operations that are supported by a type to increment a counter and then defer to the base operation. For example, an adapted `int` would support an addition operator which would increment a count of additions and then perform a normal `int` addition.

There are adaptors for each of the data types typically used by STL algorithms: iterators, container value types, container difference types and functors. Additionally there is a `recorder` class that provides for both data analysis and control over the counting process.

1.2 Counting Adaptors

The general approach to analyzing the algorithm of a function `f()` is to adapt each argument of `f()` with a counting type. Then all of the work `f()` does on its arguments will be recorded. Additionally any `Container` used by the algorithm (either directly or via iterators) should be modified to contain counting elements. This is better illustrated by a simple example.

Suppose it was desired to analyze the computational requirements of the STL sort algorithm. A non-counting call to sort a vector of integers might look as shown below.

⟨non-counting sort example 2a⟩ ≡

```
vector<int> v;
// initialize v
sort(v.begin(), v.end());
◇
```

Used in part ?.

In order to count the operations done by `sort()` two changes are made. First the vector is changed to be a vector of counting integers. Second the iterators passed to sort are made to be counting iterators.

⟨counting sort example 2b⟩ ≡

```
vector<value_counter<int> > v;
typedef vector<value_counter<int> > V;
// initialize v
sort(iterator_counter<V::iterator>(v.begin()),
     iterator_counter<V::iterator>(v.end()));
◇
```

Used in part ?.

Now all of the operations performed on the iterators and their values will be counted. In fact, any operations performed on the `difference_type` of the iterators will also be counted. The counting iterators automatically use a counting `difference_type`.

There are five adaptor types that would typically be relevant to a library user and a sixth that is generally used only internally.

- `iterator_counter` adapts STL iterators
- `value_counter` adapts the `value_type` of STL `Containers`
- `generator_counter` adapts generator functors
- `unary_function_counter` adapts unary functors
- `binary_function_counter` adapts binary functors

The sixth type, `difference_counter` adapts the `difference_type` of iterators and as mentioned above, is not typically useful outside the library's internals.

An example using a `binary_function_counter` is shown below.

⟨counting sort example, with binary function 3a⟩ ≡

```
vector<value_counter<int> > v;
typedef vector<value_counter<int> > V;
// initialize v
less<int> compare;
sort(iterator_counter<V::iterator>(v.begin()),
     iterator_counter<V::iterator>(v.end()),
     binary_function_counter<less<int> >(compare));
```

◇

Used in part ?.

1.3 Recording Results

Access to the operation counts is done via the `recorder` class. Objects of type `recorder` save snapshots of values of the counters. These can then be reported and analyzed statistically.

A typical approach to using the adaptors would be to run the algorithm several times, collecting a statistical sample of the performance. This can be done using the `recorder` class as shown below.

⟨`recorder` example 3b⟩ ≡

```
recorder<> r;
for (int i = 0; i < niterations; ++i) {
  // invoke algorithm
  r.record();
}
```

◇

Used in part ?.

The `record()` method both saves the current values of the counters and resets them to zero. The values may also be reset independently by calling the `reset()` method.

Often in addition to calling the algorithm itself, the loop will perform some setup steps to prepare the test data. For example, to test a sort algorithm one would likely create a randomized sequence of data. Since these setup operations will be working with counting types, the setup work will also be shown in the results. This should be prevented, since it will bias the analysis.

To this end, it is possible to pause and resume counting via a `recorder` object. The code below shows a modified version of the previous example that includes setup operations. Any processing done between the pause and resume will not be reflected in the counting results.

⟨`recorder` pause/resume example 4a⟩ ≡

```
recorder<> r;
for (int i = 0; i < niterations; ++i) {
  r.pause(); {
    // setup test data
  } r.resume();

  // invoke algorithm
  r.record();
}
◇
```

Used in part ?.

Since it is important to match a call to `pause()` with a corresponding call to `resume()` it is a useful idiom to create a nested block of code with braces ({}) as shown above as a visual cue indicating the statements whose operations will not be counted.

1.4 Reporting and Analyzing Results

As discussed in the previous section, after the analysis is completed, the `recorder` will contain a statistical sample of counts from different trials of the algorithm. One will normally want to report some statistical property of that sample — perhaps median, mean, minimum or maximum. By default the `recorder` will report the median of the trials via its output operator, since that statistic is less sensitive to outliers.

⟨`recorder` output example 1 4b⟩ ≡

```
cout << r;  // report median
◇
```

Used in part ?.

It is also possible to show the minimum or maximum.

⟨`recorder` output example 2 4c⟩ ≡

```
r.show_median_counts(false);
r.show_min_counts(true);
r.show_max_counts(true);
cout << r;  // report min and max but not median
◇
```

Used in part ?.

The Reference Manual describes ways in which to get more advanced and generalized reporting features.

Under normal use the operation counts for each category of adaptor are reported separately and as an overall total. Thus, there will be listings for `iterator_counter` operations, `value_counter` operations, `difference_counter` operations, etc. along with the total.

Occasionally it may be useful to obtain counts at a finer level of granularity. For example, it may be important to distinguish between operations on vector iterators versus list iterators. In this case it is possible to supply a name to the counting adaptor which will be used in reporting. This name must be an external identifier.

⟨`iterator_counter` name example 5⟩ ≡

```
char name1[] = "list<int>::iterator";
char name2[] = "vector<int>::iterator";

typedef iterator_counter<list<int>::iterator, name1> iterator1;
typedef iterator_counter<vector<int>::iterator, name2> iterator2;
◇
```

Used in part ?.

In the above example, any operations performed on `iterator1` or `iterator2` will be reported separately from other `iterator_counter` types.

Chapter 2

Reference Manual

2.1 Recorder

The `recorder` provides several functions to control its recording of counters:

- `void pause()` pauses all recording until resume() is called.
- `void resume()` resumes recording.
- `void record()` records all current counts as an iteration and calls `reset()` to begin a new iteration.
- `void reset()` begins a new iteration by clearing the recorded counts.
- `void insert(string, counter_vector*)` adds a `counter_vector` to the `recorder`'s global registry, so that its counts will be recorded. This funciton should not need be called explicitly, because `counter_vector`'s constructor calls it automatically.

2.1.1 Evaluators

A number of functions are provided to customize the behavior of the `recorder` output operator.

- **Operator Masking Functions**
  The following functions allow the user to show or hide particular counted operations in the report output.

Function	Default	Description	Example
`show_all_ops()` `hide_no_ops()`	on	Show All Operators	`r.show_all_ops();` `r.hide_no_ops();`
`hide_all_ops()` `show_no_ops()`	off	Hide All Operators	`r.show_all_ops();` `r.show_no_ops();`
`show_op(operator_type)`	n/a	Show a Particular Operator	`r.show_op(r.PLUS_CTOR);`
`hide_op(operator_type)`	n/a	Hide a Particular Operator	`r.hide_op(r.PLUS_CTOR);`
`show_op(operator_type, bool)`	n/a	Show/Hide Particular Ops	`r.show_op(r.PLUS,true);`
`show_op(InputIterator, InputIterator)`	n/a	Show/Hide Group of Ops	`vector<bool> o = (true,false);` `r.show_op(o.begin(),o.end());`

- **Snapshot Presentation Options**
  The default is to show the median count for each operation across all snapshots.

Function	Default	Description
`show_min_counts(bool)` `show_min_counts()`	false	Include Minimum Counts
`show_max_counts(bool)` `show_max_counts()`	false	Include Maximum Counts
`show_mean_counts(bool)` `show_mean_counts()`	false	Include Mean Counts
`show_median_counts(bool)` `show_median_counts()`	true	Include Median Counts
`show_all_counts(bool)` `show_all_counts()`	false	Include All Counts
`show_zero_counts(bool)` `show_zero_counts()`	false	Include Zero Counts

2.2 iterator_counter<Iterator,Difference,Name,Counter>

2.2.1 Description

`iterator_counter` is an iterator adaptor that counts how often each method of an iterator is called. The counting is stored in a recorder. With this adaptor all iterators can be adapted.

2.2.2 Example of use

⟨`iterator_counter` example of use 7⟩ ≡

```
typedef vector<int> int_vector;

typedef iterator_counter<int_vector::iterator>
          counted_vector_iterator;
```

```
recorder<> r;
int_vector v;

for (int i = 0; i < 5; i++) {
  v.push_back(i+100);
}

// external constructor
counted_vector_iterator iter1(v.begin());

// copy constructor
counted_vector_iterator iter2 = iter1;

//preincrement
++iter1;

r.record();
cout << r;
```
◇

Used in part ?.

In this example, one external constructor, one copy constructor, and one preincrement is counted.

2.2.3 Definition

Defined in `counters.h`

2.2.4 Template parameters

Parameter	Description	Default
Iterator	The base iterator class. All Operation on this class are counted.	
Difference	The difference type, which is normally the adapted difference of the base class.	`difference_counter<typename iterator_traits<Iterator>:: difference_type, Iterator>`
Name	The Name, which is used for the recorder. If it is null, "`iterator_counter`" is used.	null
Counter	The type which is used for the counting.	unsigned long

2.2.5 Model of

Random Access Iterator

2.2.6 Type requirements

The base iterator type (that is, the template parameter Iterator) must be a Random Access Iterator or an Iterator, which implement a subset of the Random Access Iterator. As an consequence, each STL Iterator can be adapted. The `Difference` type is normally the `difference_type` of the base iterator adapted by `difference_counter`.

2.2.7 Public base classes

None.

2.2.8 Members

Member	Where Defined	Description
`iterator_counter()`	Default Constructible	The default constructor
`iterator_counter (const iterator_counter& x)`	Assignable	The copy constructor
`iterator_counter& operator= (const iterator_counter& y)`	Assignable	The assignment operator
`iterator_counter (const Iterator& x)`	`iterator_counter`	see below
`Iterator base()`	`iterator_counter`	see below
`bool operator== (const iterator_counter& x const iterator_counter& y)`	Equality Comparable	Compares two iterators for equality.
`bool operator< (const iterator_counter& x, const iterator_counter& y)`	LessThan Comparable	Determines whether the first argument precedes the second.
`reference operator* ()`	Trivial Iterator	The dereference operator
`iterator_counter& operator++ ()`	Forward Iterator	Preincrement
`iterator_counter operator++ (int)`	Forward Iterator	Postincrement
`iterator_counter& operator-- ()`	Bidirectional Iterator	Predecrement
`iterator_counter operator-- (int)`	Bidirectional Iterator	Postdecrement
`iterator_counter operator+ (const Difference& n)`	Random Access Iterator	Iterator addition
`iterator_counter operator+ (const Difference& n, const iterator_counter& n)`	Random Access Iterator	Iterator addition
`iterator_counter& operator+= (const Difference& n)`	Random Access Iterator	Iterator addition
`iterator_counter operator- (const Difference& n)`	Random Access Iterator	Iterator subtraction
`Difference operator- (const iterator_counter& i)`	Random Access Iterator	Iterator subtraction
`iterator_counter& operator-= (const Difference& y)`	Random Access Iterator	Iterator subtraction
`reference operator[] (const Difference& n)`	Random Access Iterator	Random access to an element.

2.2.9 New Members

These members are not defined in the Random Access Iterator requirements, but are specific to `iterator_counter`.

Member	Description
`iterator_counter (const Iterator& n)`	This constructor takes the object n of the base class and create an `iterator_counter` with it.
`Iterator base()`	Returns the current value of the `iterator_counter`'s base iterator.

2.2.10 Notes

None.

2.3 difference_counter<Difference,Iterator,Name,Counter>

2.3.1 Description

`difference_counter` is an adaptor around a `difference_type` from an iterator. It counts how often each method of the `difference_type` is called. The counting is stored in a recorder.

2.3.2 Example of use

⟨`difference_counter` example of use 11⟩ ≡

```
typedef vector<int> int_vector;

typedef difference_counter<int_vector::difference_type,
                           int_vector::iterator>
          counted_vector_difference;

recorder<> r;
int_vector v;

for (int i = 0; i < 5; i++) {
  v.push_back(i+100);
}

// external constructor
counted_vector_difference diff1(v.end() - v.begin());

// copy constructor
counted_vector_difference diff2 = diff1;

//preincrement
++diff1;

r.record();
cout << r;
```
◇

Used in part ?.

In this example, one external constructor, one copy constructor, and one preincrement is counted.

2.3.3 Definition

Defined in `counters.h`

2.3.4 Template parameters

Parameter	Description	Default
Difference	The base difference, which is normally	
Iterator	The base iterator class of which this is the difference.	
Name	The Name, which is used for the recorder. If it is null, "`difference_counter`" is used.	null
Counter	The type which is used for the counting.	unsigned long

2.3.5 Model of

Assignable, Default Constructible, Equality Comparable, LessThan Comparable

2.3.6 Type requirements

The base difference type (that is, the template parameter Difference) must be a Difference of an Iterator. The Iterator has to be the corresponding Iterator.

2.3.7 Public base classes

None.

2.3.8 Members

Member	Where Defined	Description
`difference_counter()`	Default Constructible	The default constructor
`difference_counter` `(const difference_counter& x)`	Assignable	The copy constructor
`difference_counter& operator=` `(const difference_counter& y)`	Assignable	The assignment operator
`difference_counter`	`difference_counter`	see below
`(const Difference& x)` `Difference base()`	`difference_counter`	see below
`operator int()`	`difference_counter`	see below

`bool operator==`
`(const difference_counter& x`
`const difference_counter& y)`
	Equality Comparable	Compares two iterators for equality.

`bool operator<`
`(const difference_counter& x,`
`const difference_counter& y)`
	LessThan Comparable	Determines whether the first argument precedes the second.

`difference_counter& operator++ ()`		Preincrement
`difference_counter operator++ (int)`		Postincrement
`difference_counter& operator-- ()`		Predecrement
`difference_counter operator-- (int)`		Postdecrement
`difference_counter operator<<` `(int x)`		Shift left
`difference_counter& operator<<=` `(int x)`		Shift left
`difference_counter operator>>` `(int x)`		Shift right
`difference_counter& operator>>=` `(int x)`		Shift right
`difference_counter operator+` `(const difference_counter& y)`		addition
`Iterator operator+` `(const Iterator& y)`	Random Access Iterator	Iterator addition

`Iterator operator+`
`(const Iterator& i,`
`const difference_counter& n)`
	Random Access Iterator	Iterator addition

`difference_counter& operator+=` `(const difference_counter& y)`		addition

`Iterator& operator+=`
`(const Iterator& i,`
`const difference_counter& y)`
	Random Access Iterator	Iterator addition

`difference_counter operator-` `(const difference_counter& y)`		subtraction

`Iterator operator-`
`(const Iterator& i,`
`const difference_iterator& n)`
	Random Access Iterator	Iterator subtraction

`difference_counter& operator-=` `(const difference_counter& y)`		subtraction

`Iterator& operator-=`
`(const Iterator& i,`
`const Difference& y)`
	Random Access Iterator	Iterator subtraction

`difference_counter operator*` `(const difference_counter& y)`		multiplication
`difference_counter operator*=` `(const difference_counter& y)`		multiplication
`difference_counter operator/` `(const difference_counter& y)`		division
`difference_counter operator/=` `(const difference_counter& y)`		division

2.3.9 New Members

These members are not defined in the concepts requirements, but are specific to `difference_counter`.

Member	Description
`difference_counter (const Difference& x)`	This constructor takes the object n of the base class and create an `difference_counter` with it.
`Difference base()`	Returns the current value of the `difference_counter`'s base difference.

2.3.10 Notes

None

2.4 value_counter<Value,Name,Counter>

2.4.1 Description

`value_counter` is an adaptor around an Element to which an iterator points. It counts how often each method on an element is called. The counting is stored in a recorder.

2.4.2 Example of use

⟨`value_counter` example of use 14⟩ ≡

```
typedef value_counter<int> counted_int;

recorder<> r;

counted_int val1(1234);
counted_int val2(val1);

bool b = (val1 == val2);

r.record();
cout << r;
```
◇

Used in part ?.

In this example, one external constructor, one copy constructor and one Equality is counted.

2.4.3 Definition

Defined in `counters.h`

2.4.4 Template parameters

Parameter	Description	Default
Value	The base value class , which become adapted.	
Name	The Name, which is used for the recorder. If it is null, "`difference_counter`" is used.	null
Counter	The type which is used for the counting.	unsigned long

2.4.5 Model of

Assignable, Default Constructible, Equality Comparable, LessThan Comparable

2.4.6 Type requirements

The base value type must be at least a model of Assignable and Default Constructible.

2.4.7 Public base classes

None.

2.4.8 Members

Member	Where Defined	Description
`value_counter()`	Default Constructible	The default constructor
`value_counter` `(const value_counter& x)`	Assignable	The copy constructor
`value_counter& operator=` `(const value_counter& y)`	Assignable	The assignment operator
`value_counter` `(const Value& x)`	`value_counter`	see below
`Value base()`	`value_counter`	see below

`bool operator==`
`(const value_counter& x`
`const value_counter& y)`
	Equality Comparable	Compares two iterators for equality.

`bool operator<`
`(const value_counter& x,`
`const value_counter& y)`
	LessThan Comparable	Determines whether the first argument precedes the second.

`value_counter& operator++ ()`		Preincrement
`value_counter operator++ (int)`		Postincrement
`value_counter& operator-- ()`		Predecrement
`value_counter operator-- (int)`		Postdecrement
`value_counter operator<<` `(int x)`		Shift left
`value_counter& operator<<=` `(int x)`		Shift left
`value_counter operator>>` `(int x)`		Shift right
`value_counter& operator>>=` `(int x)`		Shift right
`value_counter operator+` `(const value_counter& y)`		addition
`value_counter& operator+=` `(const value_counter& y)`		addition
`value_counter operator-` `(const value_counter& y)`		subtraction
`value_counter& operator-=` `(const value_counter& y)`		subtraction
`value_counter operator*` `(const value_counter& y)`		multiplication
`value_counter operator*=` `(const value_counter& y)`		multiplication
`value_counter operator/` `(const value_counter& y)`		division
`value_counter operator/=` `(const value_counter& y)`		division

2.4.9 New Members

These members are not defined in the concepts requirements, but are specific to `value_counter`.

Member	Description
`value_counter (const Value& x)`	This constructor takes the object x of the base class and create an `value_counter` with it.
`Value base()`	Returns the current value of the `value_counter`'s base value.

2.4.10 Notes

None.

2.5 Generator Counting Adaptor

2.6 Unary Function Counting Adaptor

2.7 Binary Function Counting Adaptor

Chapter 3

Design Issues

3.1 Implicit Constructors

Many operations implicitly invoke a constructor, perhaps more than once. For example, the post-increment operator normally returns a new object containing the result of the increment. Should these calls be counted explicitly or should they be considered to be an understood part of the operation that invokes the constructor?

The library does not count these calls separately. The motivation is that in order to properly analyze the operation counting results, the user must understand the relative costs of each operation. Since those costs may vary from type to type, it should not be up to the library to capture that information. Furthermore, counting the implicit constructor invocations that occur at the adaptor level could result in inaccurate result reporting. Consider the following example.

⟨Implicit Constructors example 1 18⟩ ≡

```
self& operator++ (int) {
  return *this;
}
```
◇

Used in part ?.

This is not an unrealistic example. In fact, this code is essentially equivalent to the post-increment operator of the STL `back_insert_iterator`. In this case, there is no implicit call of the constructor. The post-increment operator is extremely efficient. However, the most natural implementation of a counting post-increment operator would count two constructor calls, as shown below.

⟨Implicit Constructors example 2 19a⟩ ≡

```
self operator++ (int) {
  ++counters[POST_INCREMENT];
  self return_value (*this);  // first copy ctor call
  base_iterator++;
  return return_value; // second copy ctor call
}
```
◇

Used in part ?.

For this reason, the counting adaptors implement a special private constructor that is not counted.

3.2 Default Counter Type

In the general case, operation counts may become quite large, larger even than the built-in C^{++} integer types are able to manage. To accommodate this situation, all of the counting adaptors are parameterized on the type of the counter. This allows the user to choose an alternate integral type, for instance one that models integers of arbitrary length.

Although this feature is useful it is also somewhat inconvenient to use, since to override the default, each instantiation of a counting class must specify the counter type parameter. To simplify the library's use, the default type is actually given by a C preprocessor constant, `DEFAULT_COUNTER_TYPE`. If this is predefined at compile time, its value will be used as the default counter type. Otherwise the library default of `unsigned long` will be substituted.

⟨default counter type 19b⟩ ≡

```
#ifndef DEFAULT_COUNTER_TYPE
#  define DEFAULT_COUNTER_TYPE unsigned long
#endif
```
◇

Used in part 85a.

3.3 Enumeration Constants

One option considered for the indices used to refer to elements of the counter vector was to index by member function pointers. This would result in a syntax such as is shown below.

⟨Enumeration Constants example 1 19c⟩ ≡

```
// increment the count of assignments
++counters[&operator=];
```
◇

Used in part ?.

This approach has the desirable property that the counters are tied very directly and intuitively to the function that they are counting. There are a number of problems with this approach, however, that eventually caused it to be abandoned.

The most critical problem was that not all of the counted operations are implemented as member functions. Thus, indexing by member function pointer is not meaningful in some contexts. For example, a constructor is not a regular member function. Similarly, comparison operations such as `operator==` are generally implemented as global functions.

For this reason the choice was made to use enumerated numeric constants with names chosen to correspond to the operations. This presented an additional question as to where to define the enumeration type. It could have global scope or it could be nested in one of the library's classes.

The advantage to global scope is ease of reference. If the enumeration is nested in another class, the constants must be prefixed by the class name and scope resolution operator.

⟨Enumeration Constants example 2 20a⟩ ≡

```
// global enumeration type
enum { CONSTANT1, CONSTANT2, /* ... */ };

// ...
++counters[CONSTANT1];
◇
```

Used in part ?.

⟨Enumeration Constants example 3 20b⟩ ≡

```
// nested enumeration type
class Foo {
 public:
  enum { CONSTANT1, CONSTANT2, /* ... */ };
};

// ...
++counters[Foo::CONSTANT1];
◇
```

Used in part ?.

Based on this argument it would appear that a global type is better, however, it was feared that the constants would infringe on the user's namespace. Thus, they would need to be prefixed somehow, perhaps resulting in names such as `COUNTERS_CONSTANT1`. At this point then both approaches are quite similar.

The final decision was to utilize `nuweb`'s ability to define repeated "parts" in order to include the enumeration as a nested type in each counting adaptor. Then the adaptors themselves can refer to the enumeration constants without any prefix. Library users will not normally need to use the constants directly, but if that is necessary, they can be accessed through the `recorder` class which declares its enumeration to be public.

3.4 Counter Vectors

The `counter_vector` class derived from a standard vector, and is used to hold counts for each operation within each counter adaptor. Each adaptor instantiates a `counter_vector`, and registers this vector with the `recorder`.

Registration of an adaptor's `counter_vector` is done automatically by the `counter_vector`'s constructor. It calls `recorder<>::insert()`, which adds the vector to the `recorder`'s counter registry.

3.5 Printable Names

To provide for flexibility in the reporting of results, it is desirable to be able to associate a printable name with each counting adaptor type. In this way, the counts for operations applied to `int`'s may be reported separately from those on `float`'s and `vector<int>`'s for example. This can be an important distinction since the relative cost of operations on each data type may vary.

At the same time, however, it should not be necessary for the library user to explicitly name each adaptor class that is used. A suitable default should be chosen by the library if a name is not explicitly given. This presents a number of problems.

The approach chosen is to parameterize the adaptor types with an optional `Name` argument. One alternative considered for the default was the type name provided by the C^{++} runtime type identification mechanism. The standard does not, however, provide any specific requirements on that name and in practice it is not readily interpreted by a human.

As a result, the default value does not make reference to the parameterized type of the adaptor. For example, all `iterator_counter` variants will be reported as "iterator_counter" regardless of the underlying iterator type. However, if the user desires finer grained reporting a more specific name may be given as a template parameter.[1]

This results in class declarations such as the following.

⟨Printable Names example 1 21⟩ ≡

```
template <const char* const Name = "iterator_counter">
class iterator_counter {
  // ...
};
◇
```

Used in part ?.

Unfortunately, this is not actually legal C^{++} code. A template argument of pointer type must be an object with external linkage. String literals have internal linkage. As a result, it is necessary to use a special default (non-pointer[2]) object that represents the default value. The template argument can later be tested for equality to the default object, and appropriate actions may be taken. This provides the revised class declaration shown below.

[1]The `Name` template parameter is only available if the `COUNTERS_USE_PRINTABLE_NAMES` preprocessor variable is defined. It is not by default because it does not work with `g++ 2.8.1`.

[2]Strangely enough, arrays are considered to be non-pointers in this situation. So a char[] works when a char* would not.

⟨Printable Names example 2 22⟩ ≡

```
template <const char* const Name = counter_vector<>::null>
class iterator_counter {
  // ...
  // example use of Name:
  //     (*Name == '\0') ? "iterator_counter" : Name
};
```
◇

Used in part ?.

All counting adaptors should use this or another similar mechanism to obtain the name used for reports and counter registration.

3.6 Printable Names Bug

There is yet another problem related to the issue of printable names for the counting class adaptors. Many compilers are unable to handle the default argument for the `Name` parameter. For example, `g++ 2.8.1` encounters an "internal compiler error" when the counting adaptors are used.

For this reason the printable name feature is guarded by conditional compilation directives. For the time being, since it appears to be a problem more often than not, the default is not to use this feature. If the library user wishes to use printable names and has a compiler that is capable enough for the feature, the defining preprocessor constant `COUNTERS_USE_PRINTABLE_NAMES` will enable the feature.

3.7 Evaluation

3.8 Compiler Problems

Development was done primarily with `g++ 2.8.1` and `egcs 2.91.58`. Each had definite issues. g++ has a bad habit of crashing with a very unhelpful "internal compiler error" message. egcs, while generally a lot stricter about adhering to the standard and better with the error messages, had some odd linking issues with the `counter_tags` array and some of the iterator tests.

3.9 Source Code Legibility

Nuweb tends to introduce a lot of blank lines in the resulting source code files even if we disable the `#line` printing option. This makes them somewhat hard to browse. A one-line perl script can be run from the Makefile (the `source` target) to clean up the source files, should one wish to read them directly. This script simply removes the `#line` directives and collapses multiple blank lines to a single one.

Chapter 4

Source Code

4.1 General

The following C preprocessor directives are available for customizing the behavior of the code for specific compilers or environments:

Variable	Default	Description
`COUNTERS_USE_PRINTABLE_NAMES`	`undefined`	Define to enable the Name template parameter on the counters. It must be undefined for g++ 2.8.1 due to a compiler bug.
`DEFAULT_COUNTER_TYPE`	`unsigned long`	Type to use for operation counts.

4.2 Printable Names Bug

As discussed in the Design Issues chapter, there is a problem with using a default argument for the counting adaptors' printable name (see section 3.5, page 21 and section 3.6, page 22). Thus that part of the code is conditionalized. There are four forms in which the Name parameter may be referenced. The code shown below is used later by each class when implementing aspects of the printable name feature.

⟨printable names bug definitions 24a⟩ ≡

```
#ifdef COUNTERS_USE_PRINTABLE_NAMES
#  define COUNTERS_NAME_DECL_DEFAULT \
          const char* const Name = counter_vector<>::null,
#  define COUNTERS_NAME_DECL const char* const Name,
#  define COUNTERS_NAME_PARM Name,
#  define COUNTERS_NAME_TEST (*Name == '\0')
#else
#  define COUNTERS_NAME_DECL_DEFAULT
#  define COUNTERS_NAME_DECL
#  define COUNTERS_NAME_PARM
#  define COUNTERS_NAME_TEST true
#endif
◇
```

Used in part 85a.

To simplify totalling the operation counts computed by the different adaptors an enumeration type is provided to code the types of operations. The counts themselves are kept in a vector indexed by the enumerations.

⟨counter index enumeration 24b⟩ ≡

```
enum operator_type {
  DEFAULT_CTOR, COPY_CTOR,
  OTHER_CTOR, BASE, DTOR, GENERATION,
  ASSIGNMENT, CONVERSION, LESS_THAN, EQUALITY,
  POST_INCREMENT, PRE_INCREMENT,
  POST_DECREMENT, PRE_DECREMENT,
  PLUS, MINUS, TIMES, DIVIDE, MODULO,
  LEFT_SHIFT, RIGHT_SHIFT,
  PLUS_ASSIGN, MINUS_ASSIGN,
  TIMES_ASSIGN, DIVIDE_ASSIGN, MODULO_ASSIGN,
  LEFT_SHIFT_ASSIGN, RIGHT_SHIFT_ASSIGN,
  DEREFERENCE, SUBSCRIPT, FUNCTION_CALL,
  OSTREAM, ISTREAM,
  NCOUNTERS,
};
◇
```

Used in parts 34b, 47c, 59b, 78, 80c, 82a, 84b.

This enumeration is made available to all of our classes by using nuweb's macro capability to replicate this definition to all of our classes. They are accessible to other code as `counter_vector<>::operator_type`.

For reporting purposes, a `counter_tags` array is defined in the recorder to map these enums to textual descriptions.

⟨`counter_tags` definition 25a⟩ ≡

```
    "x()", "x(y)",
    "x(?)", "base", "~x", "gen",
    "op=", "cast()", "op<", "op==",
    "op++", "++op",
    "op--", "--op",
    "op+", "op-", "op*", "op/", "op\%",
    "op<<", "op>>",
    "op+=", "op-=", "op*=", "op/=", "op\%=",
    "op<<=", "op>>=",
    "*op", "op[]", "op()",
    "cout<<", "cin>>",
◇
```

Used in parts 29c, 32.

Each counter adaptor needs to keep track of a count for each of these count types. Rather than using a vector of counters (of `DEFAULT_COUNTER_TYPE`), indexed by `operator_types`, a `counter_vector` class is defined. This class is just a vector, but it adds a constructor which automatically registers the counter with the recorder class (discussed in the Design Issues section, page 21).

⟨`counter_vector` 25b⟩ ≡

```
template <class Counter = DEFAULT_COUNTER_TYPE> class recorder;

template <class Counter = DEFAULT_COUNTER_TYPE>
class counter_vector : public vector<Counter> {
 public:
  counter_vector () : vector<Counter>(recorder<>::NCOUNTERS) {}
  counter_vector (const char* const name) : vector<Counter>() {
    recorder<Counter>::insert(name, this);
  }
  counter_vector (const char* const name, size_type n)
    : vector<Counter>(n) {
    recorder<Counter>::insert(name, this);
  }
  static char null[];
};
template <class Counter>
char counter_vector<Counter>::null[] = "";
◇
```

Used in part 85a.

4.3 Recorder

The `recorder` class is used for tracking and reporting the results of operation counting. By default a `recorder` manages all counting adaptors. Alternatively it is possible to create a recorder for a specific subset of the adaptors or operations.

The `recorder` class keeps a static registry of the `counter_vector`s for all counting adaptor classes. This is used primarily when creating a default recorder, to allow it to record all known counters. Each `counter_vector` is automatically added to the registry when it is constructed by calling `recorder`'s `insert()` method. As a static member, the registry is therefore shared between all recorder instances (if more than one registry is declared).

⟨`recorder` registry of all counters declaration 26a⟩ ≡

```
static map_type* registry;
◇
```

Used in part 34b.

⟨`recorder` registry of all counters definition 26b⟩ ≡

```
template <class Counter>
recorder<Counter>::map_type* recorder<Counter>::registry;
◇
```

Used in part 34b.

⟨`recorder insert()` method 26c⟩ ≡

```
static bool insert (const string& x, counter_vector<Counter>* y) {
  if (registry == NULL)
    registry = new map_type();
  registry->insert(map_type::value_type(x, y));
  return true;
}
◇
```

Used in part 34b.

The `recorder` uses two member variables to keep track of the counting information. First is a map that contains pointers to all of the `Counter` vectors managed by the `recorder`. This map is analagous to the registry but contains only those counters that are being managed by a particular `recorder` instance.

⟨`recorder` counter map 26d⟩ ≡

```
typedef multimap<string, counter_vector<Counter>*> map_type;
map_type counters;
◇
```

Used in part 34b.

The second member variable contains snapshots of the counter values at various instances in time. Typically the library user would perform some set of tests, then record the counter values and reset them to zero. The `recorder`'s reporting functions provide a variety of ways to analyze the counter snapshots.

The snapshots are a three-dimensional data structure (see Figure 4.1). The first index is over the snapshots themselves and thus represents time or iterations. Then each snapshot has two dimensions. The first is the counter adaptor type which is

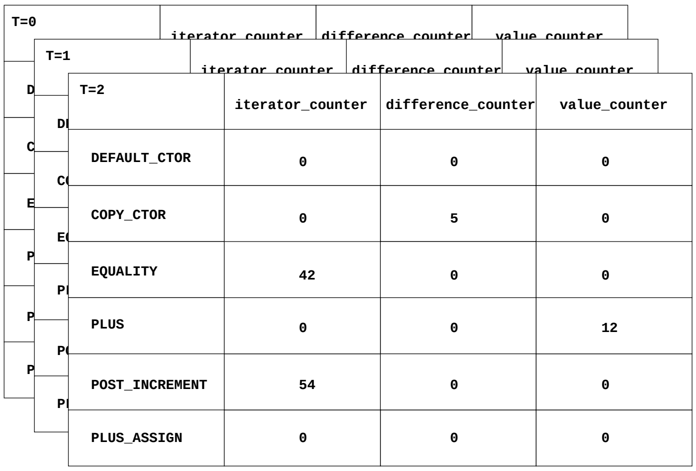

Figure 4.1: Snapshot Data Structure

represented by a string. The second index is the operation which is represented by a constant of the `operator_type` enumeration.

⟨`recorder` snapshot data 27a⟩ ≡

```
typedef multimap<string, counter_vector<Counter> > snapshot_type;
vector<snapshot_type> snapshots;

typedef map<string, counter_vector<Counter> > datum_type;
vector<datum_type> data;
◇
```

Used in part 34b.

Again, by default a `recorder` is initialized to manage all known counters.

⟨`recorder` constructor 27b⟩ ≡

```
recorder () : counters(*registry), data() {
  report_init();
}
◇
```

Used in part 34b.

Typical use of the counting adaptor classes will involve running a series of test sequences creating multiple data points for analysis. After each experiment the `record()` method should be called. This saves the current values of the counters in a snapshot and resets their values to 0. The library user may also temporarily pause counting via the `pause()` and `resume()` members. This is useful when performing setup steps to prepare for a test, such as creating the test data. It is not generally desirable to include these steps in the algorithm analysis.

⟨recorder `pause()` method 28a⟩ ≡

```
void pause () {
  snapshots.push_back(snapshot_type());
  snapshot_type& snapshot = snapshots.back();
  for (map_type::iterator i = counters.begin();
       i != counters.end();
       ++i)
    snapshot.insert(snapshot_type::value_type(i->first,
                                              *i->second));
}
```

◇

Used in part 34b.

⟨recorder `resume()` method 28b⟩ ≡

```
void resume () {
  snapshot_type& snapshot = snapshots.back();
  map_type::iterator mi;
  snapshot_type::iterator si;
  for (mi = counters.begin(), si = snapshot.begin();
       (mi != counters.end()) && (si != snapshot.end());
       ++mi, ++si)
    *mi->second = si->second;
  snapshots.pop_back();
}
```

◇

Used in part 34b.

⟨recorder `record()` method 28c⟩ ≡

```
void record () {
  data.push_back(datum_type());
  datum_type& datum = data.back();
  for (map_type::iterator i = counters.begin();
       i != counters.end();
       ++i) {
    counter_vector<Counter>& elt = datum[i->first];
    if (elt.size() < i->second->size())
      elt.reserve(i->second->size());
    for (size_t j = 0; j < i->second->size(); ++j)
      elt[j] += (*i->second)[j];
  }
  reset();
}
```

◇

Used in part 34b.

⟨recorder reset() method 29a⟩ ≡

```
void reset () {
  for (map_type::iterator i = counters.begin();
       i != counters.end();
       ++i)
    for (size_t j = 0; j < i->second->size(); ++j)
      (*i->second)[j] = 0;
}
◇
```

Used in part 34b.

Finally there are a variety of methods for reporting and analyzing the results. These use an array that maps indices in the counter vectors to printable strings.

⟨recorder print string declaration 29b⟩ ≡

```
static const char* const counter_tags[NCOUNTERS];
◇
```

Used in part 34b.

⟨recorder print string definition 29c⟩ ≡

```
template <class Counter>
const char* const recorder<Counter>::counter_tags[NCOUNTERS] = {
⟨counter_tags definition 25a⟩
};
◇
```

Used in part 34b.

The reporting functions shown below can output a delimited (with tabs by default) table of results. The `report_headings()` method simply prints list of column titles for the table. The `report_table` function lists out one line of counts for each printable name. Finally the `report_totals()` method prints the total over all printable names.

These functions take a sequence operation which operates on the snapshots in the time dimension. Thus, for a sequence of snapshots it is possible to obtain the median, minimum, maximum or some other statistic.

⟨recorder report_headings() method 29d⟩ ≡

```
ostream& report_headings (ostream& out,
                          const char* delim = "\t") const {
  out << "Type" << delim;
  copy(counter_tags,
       counter_tags + sizeof(counter_tags)/sizeof(counter_tags[0]),
       ostream_iterator<const char* const>(out, delim));
  return out << endl;
}
◇
```

Used in part 34b.

⟨recorder `report_table()` method 30⟩ ≡

```
template <class SequenceOperation>
ostream& report_table (ostream& out,
                       const SequenceOperation& op,
                       const char* delim = "\t") const {
  counter_vector<Counter> tmp;
  for (datum_type::const_iterator i = data.front().begin();
       i != data.front().end();
       ++i) {
    out << i->first << delim;

    for (size_t j = 0; j < i->second.size(); ++j) {
      tmp.erase(tmp.begin(), tmp.end());
      for (size_t k = 0; k < data.size(); ++k)
        tmp.push_back(data[k].find(i->first)->second[j]);

      Counter x = *op(tmp.begin(), tmp.end());
      out << x << delim;
    }

    out << endl;
  }
  return out;
}
```
◇

Used in part 34b.

⟨recorder report_totals() method 31⟩ ≡

```
template <class SequenceOperation>
ostream& report_totals (ostream& out,
                        const SequenceOperation& op,
                        const char* delim = "\t") const {
  out << "Total" << delim;
  counter_vector<Counter> tmp;
  for (size_t op_idx = 0; op_idx < NCOUNTERS; ++op_idx) {
    tmp.erase(tmp.begin(), tmp.end());

    for (size_t j = 0; j < data.size(); ++j) {
      tmp.push_back(0);

      datum_type::const_iterator k;
      for (k = data[j].begin(); k != data[j].end(); ++k)
        if (op_idx == GENERATION)
          tmp[j] = max(tmp[j], k->second[op_idx]);
        else
          tmp[j] += k->second[op_idx];
    }

    // At this point, tmp is a vector of counts of this operation.
    // apply the functor to get a single count from this set (max,
    // min, etc)
    Counter x = *op(tmp.begin(), tmp.end());

    out << x << delim;
  }
  return out << endl;
}
◇
```

Used in part 34b.

⟨recorder report() method 32⟩ ≡

```
template <class SequenceOperation>
ostream& report (ostream& out,
                 const SequenceOperation& op) const {
  counter_vector<Counter> tmp;

  // this seems to be necessary for egcs, although I am not
  // sure why.  For some reason the linker can not find the
  // regular static copy of counter_tags.

  const char* const counter_tags[NCOUNTERS] = {
    ⟨counter_tags definition 25a⟩
  };

  for (size_t op_idx = 0; op_idx < NCOUNTERS; ++op_idx) {
    if (! op_filter[op_idx]) continue;

    tmp.erase(tmp.begin(), tmp.end());

    for (size_t j = 0; j < data.size(); ++j) {
      tmp.push_back(0);

      datum_type::const_iterator k;
      for (k = data[j].begin(); k != data[j].end(); ++k)
        if (op_idx == GENERATION)
          tmp[j] = max(tmp[j], k->second[op_idx]);
        else
          tmp[j] += k->second[op_idx];
    }

    // At this point, tmp is a vector of counts of this operation.
    // apply the functor to get a single count from this set (max,
    // min, etc)
    Counter x = *op(tmp.begin(), tmp.end());

    if (! show_zero && x == 0) continue;

    out << setw(6) << setiosflags(ios::right);
    out << counter_tags[op_idx];
    out << resetiosflags (ios::left);
    out << " = " << x << endl;
  }
  return out;
}
◇
```

Used in part 34b.

⟨`median_element` functor 33⟩ ≡

```
template <class ForwardIterator>
ForwardIterator median_element(ForwardIterator first,
                               ForwardIterator last) {
  typedef typename iterator_traits<ForwardIterator>::value_type T;

  // because nth_element is mutative, we must copy the range to a
  // temporary container.
  vector<T> tmp(last - first);
  copy (first,last,tmp.begin());

  vector<T>::iterator midpoint = (tmp.begin() +
                                  (tmp.end() - tmp.begin()) / 2);
  nth_element(tmp.begin(), midpoint, tmp.end());

  // OK, now find the midpoint value in the original container and
  // return that iterator.
  ForwardIterator median = find(first,last,*midpoint);

  return median;
}
```

◇

Used in part 85a.

⟨`recorder` output operator 34a⟩ ≡

```
friend ostream& operator<< (ostream& out,
                            const recorder<Counter>&rhs) {

  typedef typename counter_vector<Counter>::const_iterator cvi;

  if (rhs.show_all) {
    // (not implemented)
  }

  if (rhs.show_median) {
    out << "MEDIAN" << endl << "======" << endl;
    rhs.report(out, &median_element<cvi>);
  }

  if (rhs.show_mean) {
    // (not implemented)
    //out << "MEAN" << endl << "====" << endl;
    //rhs.report(out, &mean_element<cvi>);
  }

  if (rhs.show_min) {
    out << "MIN" << endl << "===" << endl;
    rhs.report(out, &min_element<cvi>);
  }

  if (rhs.show_max) {
    out << "MAX" << endl << "===" << endl;
    rhs.report(out, &max_element<cvi>);
  }

  return out;
}
```
◇

Used in part 34b.

⟨`recorder` 34b⟩ ≡

```
template <class Counter>
class recorder {

  ⟨recorder counter map 26d⟩
  ⟨recorder snapshot data 27a⟩

  ⟨recorder print string declaration 29b⟩
  ⟨recorder registry of all counters declaration 26a⟩

  ⟨recorder output operator 34a⟩
```

```
public:
 ⟨counter index enumeration 24b⟩

 ⟨recorder constructor 27b⟩

 ⟨recorder pause() method 28a⟩
 ⟨recorder resume() method 28b⟩
 ⟨recorder record() method 28c⟩
 ⟨recorder reset() method 29a⟩

 ⟨recorder insert() method 26c⟩

 // control over which operations are reported
 ⟨recorder report_init() method 36a⟩
 ⟨recorder op_filter definition 36b⟩
 ⟨recorder show_all_ops() method 36c⟩
 ⟨recorder hide_all_ops() method 37a⟩
 ⟨recorder show_op() method 37b⟩
 ⟨recorder show_op() sequence method 38a⟩
 ⟨recorder hide_op() method 37d⟩
 ⟨recorder show_op(bool) method 37c⟩

 // control over which counts are reported
 ⟨recorder show_ count flags 38b⟩
 ⟨recorder show_min_counts() method 38c⟩
 ⟨recorder show_max_counts() method 38d⟩
 ⟨recorder show_mean_counts() method 39a⟩
 ⟨recorder show_median_counts() method 39b⟩
 ⟨recorder show_all_counts() method 39c⟩
 ⟨recorder show_zero_counts() method 39d⟩

 ⟨recorder report() method 32⟩
 ⟨recorder report_headings() method 29d⟩
 ⟨recorder report_table() method 30⟩
 ⟨recorder report_totals() method 31⟩
};

⟨recorder print string definition 29c⟩

⟨recorder registry of all counters definition 26b⟩
◇
```

Used in part 85a.

4.4 Evaluators

4.4.1 Options

The behavior of either output format can be customized with the following option settings.

Export Format

Operator Options

By default, all operators are included in the output of the reporting functions.

Defaults are set by the `report_init` function, which is called by the `recorder` constructor.

⟨`recorder report_init()` method 36a⟩ ≡

```
void report_init() {
  show_all_ops();
  show_min = false;
  show_max = false;
  show_mean = false;
  show_median = true;
  show_all = false;
  show_zero = false;
}
```

◇

Used in part 34b.

Each of the counted operations is selectively shown or hidden in the output based upon the values of this array.

⟨`recorder op_filter` definition 36b⟩ ≡

```
bool op_filter[NCOUNTERS];
```

◇

Used in part 34b.

This array is manipulated with the following functions:

- **Show all Operators**

  ⟨`recorder show_all_ops()` method 36c⟩ ≡

  ```
  void show_all_ops() {
    for (size_t i = 0; i < NCOUNTERS; ++i)
      op_filter[i] = true;
  }

  void hide_no_ops() { show_all_ops(); }
  ```

  ◇

  Used in part 34b.

- **Hide all Operators**

⟨recorder `hide_all_ops()` method 37a⟩ ≡

```
void hide_all_ops() {
  for (size_t i = 0; i < NCOUNTERS; ++i)
    op_filter[i] = false;
}

void show_no_ops() { hide_all_ops(); }
```
◇

Used in part 34b.

- **Show an Operator**

⟨recorder `show_op()` method 37b⟩ ≡

```
void show_op(operator_type op) {
   show_op(op,true);
}
```
◇

Used in part 34b.

- **Show or Hide an Operator**

⟨recorder `show_op(bool)` method 37c⟩ ≡

```
void show_op(operator_type op, bool state) {
    op_filter[op] = state;
}
```
◇

Used in part 34b.

- **Hide an Operator**

⟨recorder `hide_op()` method 37d⟩ ≡

```
void hide_op(operator_type op) {
   show_op(op,false);
}
```
◇

Used in part 34b.

- **Container of Bools**

  Another useful option is to pass in a pair of iterators over a container of bools representing whether or not to show each operation.

⟨recorder show_op() sequence method 38a⟩ ≡

```
template <class InputIterator>
void show_op (InputIterator begin, InputIterator end) {
  for (operator_type op = 0; begin != end; ++op)
    show_op(op, *begin);
}
◇
```

Used in part 34b.

Count Options

The default is to show the median.

We use some simple boolean flags to track what will be done.

⟨recorder show_ count flags 38b⟩ ≡

```
bool show_min, show_max, show_mean, show_median, show_all,
     show_zero;
◇
```

Used in part 34b.

- **Include Minimum Counts from the Recorded Iterations**

  ⟨recorder show_min_counts() method 38c⟩ ≡

  ```
  void show_min_counts(bool b) {
     show_min = b;
  }

  void show_min_counts() { show_min_counts(true); }
  ◇
  ```

  Used in part 34b.

- **Include Maximum Counts from the Recorded Iterations**

  ⟨recorder show_max_counts() method 38d⟩ ≡

  ```
  void show_max_counts(bool b) {
     show_max = b;
  }

  void show_max_counts() { show_max_counts(true); }
  ◇
  ```

  Used in part 34b.

- **Include Mean Counts from the Recorded Iterations**

⟨recorder show_mean_counts() method 39a⟩ ≡

```
void show_mean_counts(bool b) {
   show_mean = b;
}

void show_mean_counts() { show_mean_counts(true); }
```

◇

Used in part 34b.

- **Include Median Counts from the Recorded Iterations**

⟨recorder show_median_counts() method 39b⟩ ≡

```
void show_median_counts(bool b) {
   show_median = b;
}

void show_median_counts() { show_median_counts(true); }
```

◇

Used in part 34b.

- **Include All Counts from the Recorded Iterations**

⟨recorder show_all_counts() method 39c⟩ ≡

```
void show_all_counts(bool b) {
   show_all = b;
}

void show_all_counts() { show_all_counts(true); }
```

◇

Used in part 34b.

- **include zero count results**

⟨recorder show_zero_counts() method 39d⟩ ≡

```
void show_zero_counts(bool b) {
   show_zero = b;
}

void show_zero_counts() { show_zero_counts(true); }
```

◇

Used in part 34b.

4.5 Counting Adaptors

There are a number of conventions that should be followed when writing a counting adaptor to work with the `recorder` class.

All counting adaptors should take three template parameters: a name to be used in reports[1], and the type being adapted.

The operations counts themselves should be stored in a `counter_vector`. The anonymous enumeration described in section 4.2, page 24, should be used for the vector's indices.

⟨declare and register counters 40a⟩ ≡

```
static counter_vector<Counter> counters;
```
◇

Used in parts 47c, 59b, 78, 80c, 82a, 84b.

The `counter_vector` automatically registers itself with the `recorder`'s registry when it its constructor is called.

The `counter_vector` should be statically initialized as follows (this is from the `unary_function_counter`, other counters should be nearly identical):

⟨`unary_function_counter` counter variable definition 40b⟩ ≡

```
template <class AdaptableUnaryFunction,
          COUNTERS_NAME_DECL
          class Counter>
counter_vector<Counter>
unary_function_counter<AdaptableUnaryFunction,
                       COUNTERS_NAME_PARM
                       Counter>::
counters(COUNTERS_NAME_TEST ?
             "unary_function_counter" : "",
         NCOUNTERS);
```
◇

Used in part 82a.

The `COUNTERS_NAME_` macros are described in more detail in section 4.1, page 23.

Another variable that any counter adaptor should include is `generation`, defined as follows:

⟨declare generation 40c⟩ ≡

```
Counter generation;
```
◇

Used in parts 47c, 59b, 78, 80c, 82a, 84b.

Any constructors for the counter should initialize `generation` to 1. The intent behind `generation` is much like the conventional meaning of the term. A copy of an object can be considered a child or a new generation.

Thus the non-copy constructors correspond to the first generation of an object. Each copy constructor invocation increases generation by 1. Note that the generation is

[1] Actually, the name parameter should be declared using the `COUNTERS_NAME_` macros, because some compilers can not handle the name parameter at all (see section 3.5, page 21 and section 3.6, page 22).

kept on a per-object basis, unlike the `counter_vector`, which is `static`, and therefore shared among all counter adaptors of that type.

A copy constructor should set the `counter_vector[GENERATION]` to its generation if it is larger than `counter_vector[GENERATION]`'s current value. It is not useful to accumulate the generation itself, since that would simply represent a total of all constructor calls. That information is already available.

Counting other operations is mostly straightforward. In general the function should increment the appropriate counter and call the adapted class's implementation.

4.5.1 Iterator Counting Adaptor

As is true for all counting adaptors, there are template parameters to specify the adapted type, a printable name and the type of the counter itself. The latter two have default values.

There is one additional parameter that allows the user to specify the `difference_type` of the counting iterator. By default this is the `difference_type` of the base iterator, adapted for counting. Specifying the type explicitly allows control over the type's printable name and its counter type.

⟨`iterator_counter` template parameter list 41a⟩ ≡

```
template <class Iterator,
          class Difference =
              difference_counter<typename iterator_traits<Iterator>
                                          ::difference_type,
                                 Iterator>,
          COUNTERS_NAME_DECL_DEFAULT
          class Counter = DEFAULT_COUNTER_TYPE>
```
◇

Used in part 47c.

⟨`iterator_counter` typedef for `iterator_traits` 41b⟩ ≡

```
  typedef typename iterator_traits<Iterator>::iterator_category
          iterator_category;
  typedef typename iterator_traits<Iterator>::value_type
          value_type;
  typedef Difference difference_type;
  typedef typename iterator_traits<Iterator>::pointer pointer;
  typedef typename iterator_traits<Iterator>::reference reference;
```
◇

Used in part 47c.

To simplify further declarations within this class, the type definition `self` is defined as follows:

⟨`iterator_counter` typedef self 42a⟩ ≡

```
        typedef iterator_counter<Iterator, Difference,
                                 COUNTERS_NAME_PARM
                                 Counter>
                self;
```
◇

Used in part 47c.

Since operations must be declared to work on different combinations of adapted and unadapted iterators, additional types are declared to simplify these definitions.

⟨`iterator_counter` typedef for operations 42b⟩ ≡

```
       typedef Iterator UnadaptedIterator;
       typedef typename iterator_traits<Iterator>::difference_type
               UnadaptedDifference;
       typedef Difference AdaptedDifference;
```
◇

Used in part 47c.

The counter vector is defined below, initializing the printable name based on the template parameter.

⟨`iterator_counter` counter variable definition 42c⟩ ≡

```
     template <class Iterator, class Difference,
               COUNTERS_NAME_DECL
               class Counter>
     counter_vector<Counter>
     iterator_counter<Iterator, Difference,
                      COUNTERS_NAME_PARM
                      Counter>::
     counters(COUNTERS_NAME_TEST ?
                "iterator_counter" : "",
              NCOUNTERS);
```
◇

Used in part 47c.

STL concepts define which operations must be counted. In the following table these operations are shown.

	Expression	Defined in Concept
Default constructor	`I()` `I i;`	Default Constructible Output Iterator

Copy constructor	`I(i)`
	`I i(j);`
	`I i = j;`
		Assignable
		Output Iterator

Assignment	`i = j`	Assignable
Swap	`swap (i,j)`	Assignable
Equality	`i == j`	Equality Comparable
Inequality	`i != j`	Equality Comparable
Less	`i < j`	LessThan Comparable
Greater	`i > j`	LessThan Comparable
Less or equal	`i <= j`	LessThan Comparable
Greater or equal	`i >= j`	LessThan Comparable
Dereference	`*i`	Trivial Iterator
Dereference assignment	`*i = t`	Trivial Iterator Output Iterator
Member access	`i->m`	Trivial Iterator
Preincrement	`++i`	Input Iterator Output Iterator
Postincrement	`(void) i++`	Input Iterator Output Iterator
Postincrement and assign	`*i++ = t;`	Output Iterator
Postincrement and dereference	`*i++`	Input Iterator
Postincrement	`i++`	Forward Iterator
Predecrement	`--i`	Bidirectional Iterator
Postdecrement	`i--`	Bidirectional Iterator

Iterator addition	`i += n`
	`i + n`
	`n + i`
		Random Access Iterator

Iterator subtraction	`i -= n` `i - n`	Random Access Iterator
Difference	`i - j`	Random Access Iterator
Element operator	`i[n]`	Random Access Iterator
Element assignment	`i[n] = t`	Random Access Iterator

I	the class iterator
i,j	object of the class iterator
n	object of the class `difference_type`
t	object of the class `value_type`

The default constructor sets the generation to 1.

⟨`iterator_counter` default constructor 43⟩ ≡

```
iterator_counter () : //base_iterator(),
                      generation(1) {
  ++counters[DEFAULT_CTOR];
}
◇
```

Used in part 47c.

The copy constructor takes the generation and increments it by one. If this is now the highest generation, it is set to the counter.

⟨`iterator_counter` copy constructor 44a⟩ ≡

```
iterator_counter (const self& x)
     : base_iterator(x.base_iterator),
       generation(x.generation + 1) {
  ++counters[COPY_CTOR];
  counters[GENERATION] = max(counters[GENERATION], generation);
}
```
◇

Used in part 47c.

⟨`iterator_counter` destructor 44b⟩ ≡

```
~iterator_counter () { ++counters[DTOR]; }
```
◇

Used in part 47c.

In adddition to the default constructor and the copy constructor there is a constructor which takes an object of the adapted iterator class as an argument. This is necessary for some iterator classes which call special constructors. For example, a back insert iterator takes a container as an argument of its constructor. This external constructor is called first and then the resulting iterator object is given to the counting adapter with this constructor. If any other operation is called, befor the iterator object is given to the counting adaptor, that operation won't be counted.

⟨`iterator_counter` external constructor 44c⟩ ≡

```
iterator_counter (const Iterator& x) : base_iterator(x),
                                       generation(1) {
  ++counters[OTHER_CTOR];
}
```
◇

Used in part 47c.

The method `base` gives the the base iterator back.

⟨`iterator_counter` base 44d⟩ ≡

```
Iterator& base () {
  ++counters[BASE];
  return base_iterator;
}
```
◇

Used in part 47c.

Some of the counting adaptor's member functions need to call a constructor. In this case, this invocation of the constructor shoud not be counted. However, the generation should be kept. The following private constructor is provided for this purpose.

⟨`iterator_counter` uncounted constructor 45a⟩ ≡

```
iterator_counter (const self& x,
                  const bool count)

: base_iterator(x.base_iterator),
  generation(x.generation)
{
  if (count) {
    ++counters[COPY_CTOR];
  }
}

iterator_counter (const Iterator& x,
                  const Counter gen,
                  const bool count) : base_iterator(x),
                                      generation(gen) {
  if (count) {
    ++counters[OTHER_CTOR];
  }
}
◇
```

Used in part 47c.

The assignment makes a copy. The generation isn't increased.

⟨`iterator_counter operator=` , assignment 45b⟩ ≡

```
self& operator= (const self& y) {
  ++counters[ASSIGNMENT];
  base_iterator = y.base_iterator;
  return *this;
}
◇
```

Used in part 47c.

⟨`iterator_counter operator==`, equal 45c⟩ ≡

```
friend bool operator== (const self& x, const self& y) {
  ++self::counters[EQUALITY];
  return (x.base_iterator == y.base_iterator);
}
◇
```

Used in part 47c.

⟨`iterator_counter operator<` , less 46a⟩ ≡

```
friend bool operator< (const self& x, const self& y) {
  ++self::counters[LESS_THAN];
  return (x.base_iterator < y.base_iterator);
}
◇
```

Used in part 47c.

⟨`iterator_counter operator*` , dereference 46b⟩ ≡

```
reference operator* () const {
  ++counters[DEREFERENCE];
  return *base_iterator;
}
◇
```

Used in part 47c.

⟨`iterator_counter operator++`, preincrement 46c⟩ ≡

```
self& operator++ () {
  ++counters[PRE_INCREMENT];
  ++base_iterator;
  return *this;
}
◇
```

Used in part 47c.

⟨`iterator_counter operator++`, postincrement 46d⟩ ≡

```
self operator++ (int) {
  self return_value (*this, false);
  ++counters[POST_INCREMENT];
  base_iterator++;
  return self(return_value, false);
}
◇
```

Used in part 47c.

⟨`iterator_counter operator--`, predecrement 46e⟩ ≡

```
self& operator-- () {
  ++counters[PRE_DECREMENT];
  --base_iterator;
  return *this;
}
◇
```

Used in part 47c.

⟨iterator_counter operator--, postdecrement 47a⟩ ≡

```
self operator-- (int) {
  self return_value (*this, false);
  ++counters[POST_DECREMENT];
  base_iterator--;
  return self(return_value, false);
}
```

◇

Used in part 47c.

⟨iterator_counter operator<<, ostream 47b⟩ ≡

```
friend ostream& operator<< (ostream& o, const self& n) {
  return o << n.base_iterator;
}
```

◇

Used in part 47c.

⟨iterator_counter 47c⟩ ≡

```
⟨iterator_counter template parameter list 41a⟩

class iterator_counter {

  ⟨iterator_counter typedef for iterator_traits 41b⟩
  ⟨iterator_counter typedef self 42a⟩
  ⟨iterator_counter typedef for operations 42b⟩

  ⟨counter index enumeration 24b⟩
  ⟨declare and register counters 40a⟩

  Iterator base_iterator;
  ⟨declare generation 40c⟩

  ⟨iterator_counter uncounted constructor 45a⟩

 public:
  ⟨iterator_counter default constructor 43⟩
  ⟨iterator_counter copy constructor 44a⟩
  ⟨iterator_counter destructor 44b⟩
  ⟨iterator_counter external constructor 44c⟩
  ⟨iterator_counter base 44d⟩

  ⟨iterator_counter operator= , assignment 45b⟩
  ⟨iterator_counter operator==, equal 45c⟩
  ⟨iterator_counter operator< , less 46a⟩
  ⟨iterator_counter operator* , dereference 46b⟩
  ⟨iterator_counter operator++, preincrement 46c⟩
```

```
    ⟨iterator_counter operator++, postincrement 46d⟩
    ⟨iterator_counter operator--, predecrement 46e⟩
    ⟨iterator_counter operator--, postdecrement 47a⟩
    ⟨iterator_counter operator+ , addition, case 4 62a⟩
    ⟨iterator_counter operator+ , addition, case 6 62c⟩
    ⟨iterator_counter operator+ , addition, case 7 62d⟩
    ⟨iterator_counter operator+ , addition, case 9 63a⟩
    ⟨iterator_counter operator+=, addition assign, case 4 64b⟩
    ⟨iterator_counter operator+=, addition assign, case 6 64d⟩
    ⟨iterator_counter operator- , subtraction, case 4 66a⟩
    ⟨iterator_counter operator- , subtraction, case 6 66c⟩
    ⟨iterator_counter operator- , subtraction, case 7 66d⟩
    ⟨iterator_counter operator- , subtraction, case 8 66e⟩
    ⟨iterator_counter operator- , subtraction, case 9 67a⟩
    ⟨iterator_counter operator-=, subtraction assign, case 4 68b⟩
    ⟨iterator_counter operator-=, subtraction assign, case 6 68d⟩
    ⟨iterator_counter operator[], subscripton, case 1 69a⟩
    ⟨iterator_counter operator[], subscripton, case 2 69b⟩

    ⟨iterator_counter operator<<, ostream 47b⟩

};

⟨iterator_counter counter variable definition 42c⟩
◇
```

Used in part 85a.

4.5.2 Difference Type Counting Adaptor

⟨difference_counter template parameter list 48⟩ ≡

```
template <class Difference,
          class Iterator,
          COUNTERS_NAME_DECL_DEFAULT
          class Counter = DEFAULT_COUNTER_TYPE>
◇
```

Used in part 59b.

To simplify further declarations within this class, the type definition `self` is defined as follows:

⟨`difference_counter` typedef self 49a⟩ ≡

```
        typedef difference_counter<Difference, Iterator,
                                   COUNTERS_NAME_PARM
                                   Counter>
                 self;


        typedef typename iterator_traits<Iterator>::reference
           reference;
```
◇

Used in part 59b.

Since operations must be declared to work on different combinations of adapted and unadapted iterators, additional types are declared to simplify these definitions.

⟨`difference_counter` typedef for operations 49b⟩ ≡

```
       typedef Iterator UnadaptedIterator;

       typedef Difference UnadaptedDifference;
```
◇

Used in part 59b.

⟨`difference_counter` counter variable definition 49c⟩ ≡

```
     template <class Difference, class Iterator,
               COUNTERS_NAME_DECL
               class Counter>
     counter_vector<Counter>
     difference_counter<Difference, Iterator,
                        COUNTERS_NAME_PARM
                        Counter>::
     counters(COUNTERS_NAME_TEST ?
                  "difference_counter" : "",
              NCOUNTERS);
```
◇

Used in part 59b.

The `difference_type` provides the operations shown in the following table. Some these operations, which involve the class iterator, implemented in the `iterator_counter` class.

	Expression	Return Type	Defined in Concept
Default constructor	`N()` `N n;`	N	Default Constructible

Copy constructor	`N(o)`
	`N n(o);`
	`N n = o;`
		N	Assignable

Conversion	`int(n)`	int	
Assignment	`n = o`	`N&`	Assignable
Swap	`swap (n,o)`	void	Assignable
Equality	`n == o`	B	Equality Comparable
Inequality	`n != o`	B	Equality Comparable
Less	`n < o`	B	LessThan Comparable
Greater	`n > o`	B	LessThan Comparable
Less or equal	`n <= o`	B	LessThan Comparable
Greater or equal	`n >= o`	B	LessThan Comparable
Preincrement	`++n`	N	
Postincrement	`n++`	N	
Predecrement	`--n`	N	
Postdecrement	`n--`	N	

Addition	`n + o`
	`i + n`
	`n + i`
	`n += o`
	`i += n`
		N
		I
		I
		N
		I	

Subtraction	`n - o`
	`i - n`
	`n -= o`
	`i -= n`
		N
		I
		N
		I	

Multiplication	`n * o` `n *= o`	N N	
Division	`n / o` `n /= o`	N N	

N class `difference_type`
n,o object of class `difference_type`
I class iterator
i,j object of class iterator
B convertable to boolean

All constructors have to register the counters. The default constructor sets the generation to 1.

⟨`difference_counter` default constructor 50⟩ ≡

```
difference_counter () : //base_difference(),
                        generation(1) {
  ++counters[DEFAULT_CTOR];
}
```
◇

Used in part 59b.

The copy constructor takes the generation and increments it by one. If this is now the highest generation, then it is set to the counter.

⟨`difference_counter` copy constructor 51a⟩ ≡

```
difference_counter (const self& x)
: base_difference(x.base_difference),
  generation(x.generation + 1)
{
  ++counters[COPY_CTOR];
  counters[GENERATION] = max(counters[GENERATION], generation);
}
◇
```

Used in part 59b.

⟨`difference_counter` destructor 51b⟩ ≡

```
~difference_counter () { ++counters[DTOR]; }
◇
```

Used in part 59b.

In adddition to the default constructor and the copy constructor there is a constructor which takes an object of the adapted iterator class as an argument. This is necessary for some iterator classes which call special constructors. For example, a back insert iterator takes a container as an argument of its constructor. This external constructor is called first and then the resulting object is given to the counting adapter with this constructor.

⟨`difference_counter` external constructor 51c⟩ ≡

```
difference_counter (const Difference& x) : base_difference(x),
                                           generation(1) {
  ++counters[OTHER_CTOR];
}
◇
```

Used in part 59b.

Some of the counting adaptor's member functions need to call a constructor. In this case, this invocation of the constructor shoud not be counted. However, the generation should be kept. The following private constructor is provided for this purpose.

⟨`difference_counter` uncounted constructor 52a⟩ ≡

```
difference_counter (const self& x, const bool count)
     : base_difference(x.base_difference),
       generation(x.generation)
{
  if (count) {
    ++counters[COPY_CTOR];
  }
}

difference_counter (const Difference& x, const Counter gen,
                    const bool count)
     : base_difference(x), generation(gen) {
  if (count) {
    ++counters[OTHER_CTOR];
  }
}
```
◇

Used in part 59b.

⟨`difference_counter` int(), conversion 52b⟩ ≡

```
operator int() const {
  ++counters[CONVERSION];
  return int(base_difference);
}
```
◇

Used in part 59b.

With the operation base is it possible to get the base value.

⟨`difference_counter` base() 52c⟩ ≡

```
Difference base() const {
  ++counters[BASE];
  return base_difference;
}
```
◇

Used in part 59b.

The operation base is used by the `iterator_counter` and is not counted.

⟨`difference_counter` uncounted base() 53a⟩ ≡

```
Difference base(const bool count) const {
  if (count)
     ++counters[BASE];
  return base_difference;
}
◇
```

Used in part 59b.

The assignment makes a copy. The generation isn't increased.

⟨`difference_counter operator=` , assignment 53b⟩ ≡

```
self& operator= (const self& y) {
  ++counters[ASSIGNMENT];
  base_difference = y.base_difference;
  return *this;
}
◇
```

Used in part 59b.

⟨`difference_counter operator==`, equal 53c⟩ ≡

```
friend bool operator== (const self& x, const self& y) {
  ++self::counters[EQUALITY];
  return (x.base_difference == y.base_difference);
}
friend bool operator== (const self& x, const Difference& y) {
  ++self::counters[EQUALITY];
  return (x.base_difference == y);
}
friend bool operator== (const Difference& x, const self& y) {
  ++self::counters[EQUALITY];
  return (x == y.base_difference);
}
◇
```

Used in part 59b.

⟨`difference_counter operator<` , less 54a⟩ ≡

```
friend bool operator< (const self& x, const self& y) {
  ++self::counters[LESS_THAN];
  return (x.base_difference < y.base_difference);
}
friend bool operator< (const self& x, const Difference& y) {
  ++self::counters[LESS_THAN];
  return (x.base_difference < y);
}
friend bool operator< (const Difference& x, const self& y) {
  ++self::counters[LESS_THAN];
  return (x < y.base_difference);
}
◇
```

Used in part 59b.

⟨`difference_counter operator++`, preincrement 54b⟩ ≡

```
self& operator++ () {
  ++counters[PRE_INCREMENT];
  ++base_difference;
  return *this;
}
◇
```

Used in part 59b.

⟨`difference_counter operator++`, postincrement 54c⟩ ≡

```
self operator++ (int) {
  self return_value (*this, false);
  ++counters[POST_INCREMENT];
  base_difference++;
  return self(return_value, false);
}
◇
```

Used in part 59b.

⟨`difference_counter operator--`, predecrement 54d⟩ ≡

```
self& operator-- () {
  ++counters[PRE_DECREMENT];
  --base_difference;
  return *this;
}
◇
```

Used in part 59b.

⟨`difference_counter operator--`, postdecrement 55a⟩ ≡

```
self operator-- (int) {
  self return_value (*this, false);
  ++counters[POST_DECREMENT];
  base_difference--;
  return self(return_value, false);
}
```
◇

Used in part 59b.

⟨`difference_counter operator<<`, left shift 55b⟩ ≡

```
self operator<< (int x) const {
  ++counters[LEFT_SHIFT];
  return self(base_difference << x, false);
}
```
◇

Used in part 59b.

⟨`difference_counter operator>>`, right shift 55c⟩ ≡

```
self operator>> (int x) const {
  ++counters[RIGHT_SHIFT];
  return self(base_difference >> x, false);
}
```
◇

Used in part 59b.

⟨`difference_counter operator<<=`, left shift assign 55d⟩ ≡

```
self& operator<<= (int x) {
  ++counters[LEFT_SHIFT_ASSIGN];
  base_difference <<= x;
  return *this;
}
```
◇

Used in part 59b.

⟨`difference_counter operator>>=`, right shift assign 55e⟩ ≡

```
self& operator>>= (int x) {
  ++counters[RIGHT_SHIFT_ASSIGN];
  base_difference >>= x;
  return *this;
}
```
◇

Used in part 59b.

The multiplication operation is broken into three cases:

expr	argument	ret type	counted	case
`n * n`	difference, difference	difference	no	
	AdaptDiff, difference	AdaptDiff	yes	1
	difference, AdaptDiff	AdaptDiff	yes	2
	AdaptDiff, AdaptDiff	AdaptDiff	yes	3

⟨`difference_counter operator*` , multiply, case 1 56a⟩ ≡

```
self operator* (const UnadaptedDifference& y) const {
  ++counters[TIMES];
  return self(base_difference * y, generation, false);
}
◇
```

Used in part 59b.

⟨`difference_counter operator*` , multiply, case 2 56b⟩ ≡

```
friend self operator* (const UnadaptedDifference& x,
                       const self& y) {
  ++counters[TIMES];
  return self(x * y.base_difference, y.generation, false);
}
◇
```

Used in part 59b.

⟨`difference_counter operator*` , multiply, case 3 56c⟩ ≡

```
self operator* (const self& y) const {
  ++counters[TIMES];
  return self(base_difference * y.base_difference,
              max(generation, y.generation), false);
}
◇
```

Used in part 59b.

The multiplication assignment operation is broken into three cases:

expr	argument	ret type	counted	case
`n *= n`	difference, difference	difference	no	
	AdaptDiff, difference	AdaptDiff	yes	1
	difference, AdaptDiff	difference	yes	2
	AdaptDiff, AdaptDiff	AdaptDiff	yes	3

⟨`difference_counter operator*=`, multiply assign, case 1 57a⟩ ≡

```
self& operator*= (const UnadaptedDifference& y) {
  ++counters[TIMES_ASSIGN];
  base_difference *= y;
  return *this;
}
◇
```

Used in part 59b.

⟨`difference_counter operator*=`, multiply assign, case 2 57b⟩ ≡

```
friend UnadaptedDifference& operator*= (UnadaptedDifference& x,
                                        const self& y) {
  ++counters[TIMES_ASSIGN];
  x *= y.base_difference;
  return x;
}
◇
```

Used in part 59b.

⟨`difference_counter operator*=`, multiply assign, case 3 57c⟩ ≡

```
self& operator*= (const self& y) {
  ++counters[TIMES_ASSIGN];
  base_difference *= y.base_difference;
  generation = max(generation, y.generation);
  return *this;
}
◇
```

Used in part 59b.

The division operation is broken into three cases:

expr	argument	ret type	counted	case
`n / n`	difference, difference	difference	no	
	AdaptDiff, difference	AdaptDiff	yes	1
	difference, AdaptDiff	AdaptDiff	yes	2
	AdaptDiff, AdaptDiff	AdaptDiff	yes	3

⟨`difference_counter operator/` , divide, case 1 57d⟩ ≡

```
self operator/ (const UnadaptedDifference& y) const {
  ++counters[DIVIDE];
  return self(base_difference / y, generation, false);
}
◇
```

Used in part 59b.

⟨`difference_counter operator/` , divide, case 2 58a⟩ ≡

```
friend self operator/ (const UnadaptedDifference& x,
                       const self& y) {
  ++counters[DIVIDE];
  return self(x / y.base_difference, y.generation, false);
}
◇
```

Used in part 59b.

⟨`difference_counter operator/` , divide, case 3 58b⟩ ≡

```
self operator/ (const self& y) const {
  ++counters[DIVIDE];
  return self(base_difference / y.base_difference,
              max(generation, y.generation), false);
}
◇
```

Used in part 59b.

The division assignment operation is broken into three cases:

expr	argument	ret type	counted	case
`n /= n`	difference, difference	difference	no	
	AdaptDiff, difference	AdaptDiff	yes	1
	difference, AdaptDiff	difference	yes	2
	AdaptDiff, AdaptDiff	AdaptDiff	yes	3

⟨`difference_counter operator/=`, divide assign, case 1 58c⟩ ≡

```
self& operator/= (const UnadaptedDifference& y) {
  ++counters[DIVIDE_ASSIGN];
  base_difference /= y;
  return *this;
}
◇
```

Used in part 59b.

⟨`difference_counter operator/=`, divide assign, case 2 58d⟩ ≡

```
friend UnadaptedDifference& operator/= (UnadaptedDifference& x,
                                         const self& y) {
  ++counters[DIVIDE_ASSIGN];
  x /= y.base_difference;
  return x;
}
◇
```

Used in part 59b.

⟨`difference_counter operator/=`, divide assign, case 3 59a⟩ ≡

```
self& operator/= (const self& y) {
  ++counters[DIVIDE_ASSIGN];
  base_difference /= y.base_difference;
  generation = max(generation, y.generation);
  return *this;
}
```
◇

Used in part 59b.

⟨`difference_counter` 59b⟩ ≡

```
⟨difference_counter template parameter list 48⟩
class difference_counter {

  ⟨difference_counter typedef self 49a⟩
  ⟨difference_counter typedef for operations 49b⟩

  ⟨counter index enumeration 24b⟩
  ⟨declare and register counters 40a⟩

  Difference base_difference;
  ⟨declare generation 40c⟩

public:
  ⟨difference_counter uncounted constructor 52a⟩
  ⟨difference_counter uncounted base() 53a⟩

public:
  ⟨difference_counter default constructor 50⟩
  ⟨difference_counter copy constructor 51a⟩
  ⟨difference_counter destructor 51b⟩
  ⟨difference_counter external constructor 51c⟩
  ⟨difference_counter int(), conversion 52b⟩
  ⟨difference_counter base() 52c⟩

  ⟨difference_counter operator= , assignment 53b⟩
  ⟨difference_counter operator==, equal 53c⟩
  ⟨difference_counter operator< , less 54a⟩
  ⟨difference_counter operator++, preincrement 54b⟩
  ⟨difference_counter operator++, postincrement 54c⟩
  ⟨difference_counter operator--, predecrement 54d⟩
  ⟨difference_counter operator--, postdecrement 55a⟩
  ⟨difference_counter operator<<, left shift 55b⟩
  ⟨difference_counter operator>>, right shift 55c⟩
  ⟨difference_counter operator<<=, left shift assign 55d⟩
  ⟨difference_counter operator>>=, right shift assign 55e⟩
  ⟨difference_counter operator+ , addition, case 1 61a⟩
  ⟨difference_counter operator+ , addition, case 2 61b⟩
```

```
    ⟨difference_counter operator+ , addition, case 3 61c⟩
    ⟨difference_counter operator+ , addition, case 5 62b⟩
    ⟨difference_counter operator+ , addition, case 8 62e⟩
    ⟨difference_counter operator+=, addition assign, case 1 63b⟩
    ⟨difference_counter operator+=, addition assign, case 2 63c⟩
    ⟨difference_counter operator+=, addition assign, case 3 64a⟩
    ⟨difference_counter operator+=, addition assign, case 5 64c⟩
    ⟨difference_counter operator- , subtraction, case 1 65a⟩
    ⟨difference_counter operator- , subtraction, case 2 65b⟩
    ⟨difference_counter operator- , subtraction, case 3 65c⟩
    ⟨difference_counter operator- , subtraction, case 5 66b⟩
    ⟨difference_counter operator-=, subtraction assign, case 1 67b⟩
    ⟨difference_counter operator-=, subtraction assign, case 2 67c⟩
    ⟨difference_counter operator-=, subtraction assign, case 3 68a⟩
    ⟨difference_counter operator-=, subtraction assign, case 5 68c⟩
    ⟨difference_counter operator* , multiply, case 1 56a⟩
    ⟨difference_counter operator* , multiply, case 2 56b⟩
    ⟨difference_counter operator* , multiply, case 3 56c⟩
    ⟨difference_counter operator*=, multiply assign, case 1 57a⟩
    ⟨difference_counter operator*=, multiply assign, case 2 57b⟩
    ⟨difference_counter operator*=, multiply assign, case 3 57c⟩
    ⟨difference_counter operator/ , divide, case 1 57d⟩
    ⟨difference_counter operator/ , divide, case 2 58a⟩
    ⟨difference_counter operator/ , divide, case 3 58b⟩
    ⟨difference_counter operator/=, divide assign, case 1 58c⟩
    ⟨difference_counter operator/=, divide assign, case 2 58d⟩
    ⟨difference_counter operator/=, divide assign, case 3 59a⟩
  };

  ⟨difference_counter counter variable definition 49c⟩
  ◇
```

Used in part 85a.

4.5.3 Mixed Operations on Iterator and Difference Types

The following operations involve operations between `iterator` and `difference_type` arguments.

The addition assignment operation is broken into nine cases:

expr	argument	ret type	counted in	
`n + o`	Difference, Difference	Difference	-	
	AdaptDiff, Difference	AdaptDiff	`difference_counter`	1
	Difference, AdaptDiff	AdaptDiff	`difference_counter`	2
	AdaptDiff, AdaptDiff	AdaptDiff	`difference_counter`	3
`i + n`	Iterator, Difference	Iterator	-	
	AdaptIter, Difference	AdaptIter	`iterator_counter`	4
	Iterator, AdaptDiff	Iterator	`difference_counter`	5
	AdaptIter, AdaptDiff	AdaptIter	`iterator_counter`	6
`d + n`	Difference, Iterator	Iterator	-	
	Difference, AdaptIter	AdaptIter	`iterator_counter`	7
	AdaptDiff, Iterator	Iterator	`difference_counter`	8
	AdaptDiff, AdaptIter	AdaptIter	`iterator_counter`	9

n, o object of class `difference_type`
i object of class iterator

⟨`difference_counter operator+` , addition, case 1 61a⟩ ≡

```
self operator+ (const UnadaptedDifference& y) const {
  ++counters[PLUS];
  return self(base_difference + y, generation, false);
}
◇
```

Used in part 59b.

⟨`difference_counter operator+` , addition, case 2 61b⟩ ≡

```
friend self operator+ (const UnadaptedDifference& x,
                       const self& y) {
  ++self::counters[PLUS];
  return self(x + y.base_difference, y.generation, false);
}
◇
```

Used in part 59b.

⟨`difference_counter operator+` , addition, case 3 61c⟩ ≡

```
self operator+ (const self& y) const {
  ++counters[PLUS];
  return self(base_difference + y.base_difference,
              max(generation, y.generation), false);
}
◇
```

Used in part 59b.

⟨`iterator_counter operator+` , addition, case 4 62a⟩ ≡

```
self operator+(const UnadaptedDifference& n) const {
  ++counters[PLUS];
  return self(base_iterator + n, generation, false);
}
```
◇

Used in part 47c.

⟨`difference_counter operator+` , addition, case 5 62b⟩ ≡

```
friend Iterator operator+ (const UnadaptedIterator& i,
                           const self& n) {
  ++self::counters[PLUS];
  return Iterator(i + n.base_difference);
}
```
◇

Used in part 59b.

⟨`iterator_counter operator+` , addition, case 6 62c⟩ ≡

```
self operator+(const AdaptedDifference& n) const {
  ++counters[PLUS];
  return self(base_iterator + n.base(false), generation, false);
}
```
◇

Used in part 47c.

⟨`iterator_counter operator+` , addition, case 7 62d⟩ ≡

```
friend self operator+(const UnadaptedDifference& n,
                      const self& i) {
  ++self::counters[PLUS];
  return self(n + i.base_iterator, i.generation, false);
}
```
◇

Used in part 47c.

⟨`difference_counter operator+` , addition, case 8 62e⟩ ≡

```
Iterator operator+ (const UnadaptedIterator& i) {
  ++counters[PLUS];
  return Iterator(i + base_difference);
}
```
◇

Used in part 59b.

⟨`iterator_counter operator+` , addition, case 9 63a⟩ ≡

```
friend self operator+ (const AdaptedDifference& n,
                       const self& i) {
  ++self::counters[PLUS];
  return self (i.base_iterator + n.base(false),
               i.generation, false);
}
```
◇

Used in part 47c.

The addition assignment operation is broken into six cases:

expr	argument	ret type	counted in	
`n += o`	Difference, Difference	Difference	-	
	AdaptDiff, Difference	AdaptDiff	`difference_counter`	1
	Difference, AdaptDiff	Difference	`difference_counter`	2
	AdaptDiff, AdaptDiff	AdaptDiff	`difference_counter`	3
`i += n`	Iterator, Difference	Iterator	-	
	AdaptIter, Difference	AdaptIter	`iterator_counter`	4
	Iterator, AdaptDiff	Iterator	`difference_counter`	5
	AdaptIter, AdaptDiff	AdaptIter	`iterator_counter`	6

n, o object of class `difference_type`
i object of class iterator

⟨`difference_counter operator+=`, addition assign, case 1 63b⟩ ≡

```
self& operator+= (const UnadaptedDifference& y) {
  ++counters[PLUS_ASSIGN];
  base_difference += y;
  return *this;
}
```
◇

Used in part 59b.

⟨`difference_counter operator+=`, addition assign, case 2 63c⟩ ≡

```
friend UnadaptedDifference& operator+= (UnadaptedDifference& x,
                                        const self& y) {
  ++counters[PLUS_ASSIGN];
  x += y.base_difference;
  return x;
}
```
◇

Used in part 59b.

⟨`difference_counter operator+=`, addition assign, case 3 64a⟩ ≡

```
self& operator+= (const self& y) {
  ++counters[PLUS_ASSIGN];
  base_difference += y.base_difference;
  generation = max(generation, y.generation);
  return *this;
}
```
◇

Used in part 59b.

⟨`iterator_counter operator+=`, addition assign, case 4 64b⟩ ≡

```
self& operator+= (const UnadaptedDifference& y) {
  ++counters[PLUS_ASSIGN];
  base_iterator += y;
  return *this;
}
```
◇

Used in part 47c.

⟨`difference_counter operator+=`, addition assign, case 5 64c⟩ ≡

```
friend Iterator& operator+= (UnadaptedIterator& i,
                             const self& n) {
  ++counters[PLUS_ASSIGN];
  i += n.base_difference;
  return i;
}
```
◇

Used in part 59b.

⟨`iterator_counter operator+=`, addition assign, case 6 64d⟩ ≡

```
self& operator+= (const AdaptedDifference& n) {
  ++counters[PLUS_ASSIGN];
  base_iterator += n.base(false);
  return *this;
}
```
◇

Used in part 47c.

The subtraction operation is broken into nine cases:

expr	argument	ret type	counted in	
`n - o`	Difference, Difference	Difference	-	
	AdaptDiff, Difference	AdaptDiff	`difference_counter`	1
	Difference, AdaptDiff	AdaptDiff	`difference_counter`	2
	AdaptDiff, AdaptDiff	AdaptDiff	`difference_counter`	3
`i - n`	Iterator, Difference	Iterator	-	
	AdaptIter, Difference	AdaptIter	`iterator_counter`	4
	Iterator, AdaptDiff	Iterator	`difference_counter`	5
	AdaptIter, AdaptDiff	AdaptIter	`iterator_counter`	6
`i - j`	Iterator, Iterator	Difference	-	
	Iterator, AdaptIter	AdaptDiff	`iterator_counter`	7
	AdaptIter, Iterator	AdaptDiff	`iterator_counter`	8
	AdaptIter, AdaptIter	AdaptDiff	`iterator_counter`	9

n, o   object of class `difference_type`
i, j   object of class iterator

⟨`difference_counter operator-` , subtraction, case 1 65a⟩ ≡

```
self operator- (const UnadaptedDifference& y) const {
  ++counters[MINUS];
  return self(base_difference - y, generation, false);
}
```
◇

Used in part 59b.

⟨`difference_counter operator-` , subtraction, case 2 65b⟩ ≡

```
friend self operator- (const UnadaptedDifference& x,
                       const self& y) {
  ++counters[MINUS];
  return self(x - y.base_difference, y.generation, false);
}
```
◇

Used in part 59b.

⟨`difference_counter operator-` , subtraction, case 3 65c⟩ ≡

```
self operator- (const self& y) const {
  ++counters[MINUS];
  return self(base_difference - y.base_difference,
              max(generation, y.generation), false);
}
```
◇

Used in part 59b.

⟨`iterator_counter operator-` , subtraction, case 4 66a⟩ ≡

```
self operator- (const UnadaptedDifference& n) const {
  ++counters[MINUS];
  return self(base_iterator - n, generation, false);
}
◇
```

Used in part 47c.

⟨`difference_counter operator-` , subtraction, case 5 66b⟩ ≡

```
friend Iterator operator- (const UnadaptedIterator& i,
                           const self& n) {
  ++counters[MINUS];
  return Iterator(i - n.base_difference);
}
◇
```

Used in part 59b.

⟨`iterator_counter operator-` , subtraction, case 6 66c⟩ ≡

```
self operator- (const AdaptedDifference& n) const {
  ++counters[MINUS];
  return self(base_iterator - n.base(false), generation, false);
}
◇
```

Used in part 47c.

⟨`iterator_counter operator-` , subtraction, case 7 66d⟩ ≡

```
friend AdaptedDifference operator- (const UnadaptedIterator& x,
                                    const self& y) {
  ++counters[MINUS];
  return AdaptedDifference(x - y.base_iterator, 1, false);
}
◇
```

Used in part 47c.

⟨`iterator_counter operator-` , subtraction, case 8 66e⟩ ≡

```
AdaptedDifference operator- (const UnadaptedIterator& i) {
  ++counters[MINUS];
  return AdaptedDifference(base_iterator - i, 1, false);
}
◇
```

Used in part 47c.

⟨`iterator_counter operator-` , subtraction, case 9 67a⟩ ≡

```
AdaptedDifference operator- (const self& i) {
  ++counters[MINUS];
  return AdaptedDifference (base_iterator -
                            i.base_iterator, 1, false);
}
◇
```

Used in part 47c.

The subtraction assignment operation is broken into six cases:

expr	argument	ret type	counted in	
`n -= o`	Difference, Difference	Difference	-	
	AdaptDiff, Difference	AdaptDiff	`difference_counter`	1
	Difference, AdaptDiff	Difference	`difference_counter`	2
	AdaptDiff, AdaptDiff	AdaptDiff	`difference_counter`	3
`i -= n`	Iterator, Difference	Iterator	-	
	AdaptIter, Difference	AdaptIter	`iterator_counter`	4
	Iterator, AdaptDiff	Iterator	`difference_counter`	5
	AdaptIter, AdaptDiff	AdaptIter	`iterator_counter`	6

n, o object of class `difference_type`
i object of class iterator

⟨`difference_counter operator-=`, subtraction assign, case 1 67b⟩ ≡

```
self& operator-= (const UnadaptedDifference& y) {
  ++counters[MINUS_ASSIGN];
  base_difference -= y;
  return *this;
}
◇
```

Used in part 59b.

⟨`difference_counter operator-=`, subtraction assign, case 2 67c⟩ ≡

```
friend UnadaptedDifference& operator-= (UnadaptedDifference& x,
                                        const self& y) {
  ++counters[MINUS_ASSIGN];
  x -= y.base_difference;
  return x;
}
◇
```

Used in part 59b.

⟨`difference_counter operator-=`, subtraction assign, case 3 68a⟩ ≡

```
self& operator-= (const self& y) {
  ++counters[MINUS_ASSIGN];
  base_difference -= y.base_difference;
  generation = max(generation, y.generation);
  return *this;
}
◇
```

Used in part 59b.

⟨`iterator_counter operator-=`, subtraction assign, case 4 68b⟩ ≡

```
self& operator-= (const UnadaptedDifference& y) {
  ++counters[MINUS_ASSIGN];
  base_iterator -= y;
  return *this;
}
◇
```

Used in part 47c.

⟨`difference_counter operator-=`, subtraction assign, case 5 68c⟩ ≡

```
friend Iterator& operator-= (UnadaptedIterator& i,
                             const self& n) {
  ++counters[MINUS_ASSIGN];
  i -= n.base_difference;
  return i;
}
◇
```

Used in part 59b.

⟨`iterator_counter operator-=`, subtraction assign, case 6 68d⟩ ≡

```
self& operator-= (const AdaptedDifference& n) {
  ++counters[MINUS_ASSIGN];
  base_iterator -= n.base(false);
  return *this;
}
◇
```

Used in part 47c.

The subscript operation is broken into three cases:

expr	argument	ret type	counted in	
`i[n]`	Iterator, Difference	Iterator	-	
	AdaptIter, Difference	AdaptIter	`iterator_counter`	1
	AdaptIter, AdaptDiff	AdaptIter	`iterator_counter`	2

n, o object of class `difference_type`
i object of class iterator

⟨`iterator_counter operator[]`, subscripton, case 1 69a⟩ ≡

```
reference operator[] (const UnadaptedDifference& n) const {
  ++counters[SUBSCRIPT];
  return base_iterator[n];
}
◇
```

Used in part 47c.

⟨`iterator_counter operator[]`, subscripton, case 2 69b⟩ ≡

```
reference operator[](const AdaptedDifference& n) const {
  ++counters[SUBSCRIPT];
  return base_iterator[n.base(false)];
}
◇
```

Used in part 47c.

4.5.4 Value Type Counting Adaptor

⟨`value_counter` template parameter list 69c⟩ ≡

```
template <class Value,
          COUNTERS_NAME_DECL_DEFAULT
          class Counter          = DEFAULT_COUNTER_TYPE>
◇
```

Used in part 78.

To simplify further declarations within this class, the type definition `self` is defined as follows:

⟨`value_counter` typedef self 69d⟩ ≡

```
   typedef value_counter<Value,
                         COUNTERS_NAME_PARM
                         Counter> self;
◇
```

Used in part 78.

⟨`value_counter` counter variable definition 70⟩ ≡

```
template <class Value,
          COUNTERS_NAME_DECL
          class Counter>
counter_vector<Counter>
value_counter<Value,
              COUNTERS_NAME_PARM
              Counter>::
counters(COUNTERS_NAME_TEST ?
             "value_counter" : "",
         NCOUNTERS);
◇
```

Used in part 78.

The STL concepts define which operation are required. These required operations are shown in the following table. Some iterator adaptors have additional type requirements.

	Expression	Defined in Concept
Default constructor	`T()` `T t;`	Default Constructible

Copy constructor	`T(t)`
	`T t(s);`
	`T t = s;`
		Assignable

Assignment	`t = s`	Assignable
Swap	`swap (t,s)`	Assignable
Equality	`t == s`	Equality Comparable
Inequality	`t != s`	Equality Comparable
Less	`t < s`	LessThan Comparable
Greater	`t > s`	LessThan Comparable
Less or equal	`t <= s`	LessThan Comparable
Greater or equal	`t >= s`	LessThan Comparable
Pre-increment	`++t`	
Post-increment	`t++`	
Pre-decrement	`--t`	
Post-decrement	`t--`	
Left shift	`t << i`	
Right shift	`t >> i`	
Ostream	`cout << t`	ostream_iterator
Istream	`cin >> t`	istream_iterator

T the class `value`
t,s object of the class `value`
i object of type `int`

The default constructor sets the generation to 1.

⟨`value_counter` default constructor 71a⟩ ≡

```
value_counter () : //value_iterator(),
                   generation(1) {
  ++counters[DEFAULT_CTOR];
}
◇
```

Used in part 78.

The copy constructor takes the generation and increment it by one. If this is now the highest generation, it is set to the counter.

⟨`value_counter` copy constructor 71b⟩ ≡

```
value_counter (const self& x)
     : base_value(x.base_value),
       generation(x.generation + 1) {
  ++counters[COPY_CTOR];
  counters[GENERATION] = max(counters[GENERATION], generation);
}
◇
```

Used in part 78.

⟨`value_counter` destructor 71c⟩ ≡

```
~value_counter () { ++counters[DTOR]; }
◇
```

Used in part 78.

In adddition to the default constructor and the copy constructor there is a constructor which takes an object of the adapted iterator class as an argument. This is necessary for some iterator classes which call special constructors. For example, a back insert iterator takes a container as an argument of its constructor. This external constructor is called first and then the resulting object is given to the counting adapter with this constructor.

⟨`value_counter` external constructor 71d⟩ ≡

```
value_counter (const Value& v) : base_value(v),
                                 generation(1) {
  ++counters[OTHER_CTOR];
}
◇
```

Used in part 78.

The method `base` gives the the base value back.

⟨`value_counter` base 72a⟩ ≡

```
Value& base() {
  ++counters[BASE];
  return base_value;
}
◇
```

Used in part 78.

Some of the counting adaptor's member functions need to call a constructor. In this case, this invocation of the constructor shoud not be counted. However, the generation should be kept. The following private constructor is provided for this purpose.

⟨`value_counter` uncounted constructor 72b⟩ ≡

```
value_counter (const self& x,
               const bool count)
  : base_value(x.base_value),
    generation(x.generation) {
  if (count)
    ++counters[COPY_CTOR];
}

value_counter (const Value& v,
               const Counter gen,
               const bool count) : base_value(v),
                                   generation(gen) {
  if (count)
    ++counters[OTHER_CTOR];
}
◇
```

Used in part 78.

The assignment makes a copy. The generation isn't increased.

⟨`value_counter operator=` , assignment 72c⟩ ≡

```
self& operator=(const self& y) {
  ++counters[ASSIGNMENT];
  base_value = y.base_value;
  return *this;
}
◇
```

Used in part 78.

⟨`value_counter operator==`, equal 73a⟩ ≡

```
friend bool operator==(const self& x, const self& y) {
  ++self::counters[EQUALITY];
  return (x.base_value == y.base_value);
}
◇
```

Used in part 78.

⟨`value_counter operator<` , less 73b⟩ ≡

```
friend bool operator<(const self& x, const self& y) {
  ++self::counters[LESS_THAN];
  return (x.base_value < y.base_value);
}
◇
```

Used in part 78.

⟨`value_counter operator++`, preincrement 73c⟩ ≡

```
self& operator++() {
  ++self::counters[PRE_INCREMENT];
  ++base_value;
  return *this;
}
◇
```

Used in part 78.

⟨`value_counter operator++`, postincrement 73d⟩ ≡

```
self operator++ (int) {
  self return_value (*this, false);
  ++counters[POST_INCREMENT];
  base_value++;
  return self(return_value, false);
}
◇
```

Used in part 78.

⟨`value_counter operator--`, predecrement 73e⟩ ≡

```
self& operator--() {
  ++self::counters[PRE_DECREMENT];
  --base_value;
  return *this;
}
◇
```

Used in part 78.

⟨value_counter operator--, postdecrement 74a⟩ ≡

```
self operator-- (int) {
  self return_value (*this, false);
  ++counters[POST_DECREMENT];
  base_value--;
  return self(return_value, false);
}
◇
```

Used in part 78.

⟨value_counter operator<<, left shift 74b⟩ ≡

```
self operator<< (int x) const {
  ++counters[LEFT_SHIFT];
  return self(base_value << x, false);
}
◇
```

Used in part 78.

⟨value_counter operator<<=, left shift assign 74c⟩ ≡

```
self& operator<<= (int x) {
  ++counters[LEFT_SHIFT_ASSIGN];
  base_value <<= x;
  return *this;
}
◇
```

Used in part 78.

⟨value_counter operator>>, right shift 74d⟩ ≡

```
self operator>> (int x) const {
  ++counters[RIGHT_SHIFT];
  return self(base_value >> x, false);
}
◇
```

Used in part 78.

⟨value_counter operator>>=, right shift assign 74e⟩ ≡

```
self& operator>>= (int x) {
  ++counters[RIGHT_SHIFT_ASSIGN];
  base_value >>= x;
  return *this;
}
◇
```

Used in part 78.

⟨`value_counter operator+` , addition 75a⟩ ≡

```
friend self operator+ (const self& x, const self& y) {
  ++self::counters[PLUS];
  return self(x.base_value + y.base_value,
              max(x.generation, y.generation), false);
}
```
◇

Used in part 78.

⟨`value_counter operator+=`, addition assign 75b⟩ ≡

```
self& operator+= (const self& y) {
  ++counters[PLUS_ASSIGN];
  base_value += y.base_value;
  generation = max(generation, y.generation);
  return *this;
}
```
◇

Used in part 78.

⟨`value_counter operator-` , subtraction 75c⟩ ≡

```
friend self operator- (const self& x, const self& y) {
  ++self::counters[MINUS];
  return self(x.base_value - y.base_value,
              max(x.generation, y.generation), false);
}
```
◇

Used in part 78.

⟨`value_counter operator-=`, subtraction assign 75d⟩ ≡

```
self& operator-= (const self& y) {
  ++counters[MINUS_ASSIGN];
  base_value -= y.base_value;
  generation = max(generation, y.generation);
  return *this;
}
```
◇

Used in part 78.

⟨`value_counter operator*` , multiply 76a⟩ ≡

```
friend self operator* (const self& x, const self& y) {
  ++self::counters[TIMES];
  return self(x.base_value * y.base_value,
              max(x.generation, y.generation), false);
}
```
◇

Used in part 78.

⟨`value_counter operator*=`, multiply assign 76b⟩ ≡

```
self& operator*= (const self& y) {
  ++counters[TIMES_ASSIGN];
  base_value *= y.base_value;
  generation = max(generation, y.generation);
  return *this;
}
```
◇

Used in part 78.

⟨`value_counter operator/` , divide 76c⟩ ≡

```
friend self operator/ (const self& x, const self& y) {
  ++self::counters[DIVIDE];
  return self(x.base_value / y.base_value,
              max(x.generation, y.generation), false);
}
```
◇

Used in part 78.

⟨`value_counter operator/=`, divide assign 76d⟩ ≡

```
self& operator/= (const self& y) {
  ++counters[DIVIDE_ASSIGN];
  base_value /= y.base_value;
  generation = max(generation, y.generation);
  return *this;
}
```
◇

Used in part 78.

⟨`value_counter operator<<`, ostream 77a⟩ ≡

```
friend ostream& operator<< (ostream& out, const self& t) {
  ++counters[OSTREAM];
  return out << t.base_value;
}
◇
```

Used in part 78.

⟨`value_counter operator>>`, istream 77b⟩ ≡

```
friend istream& operator>> (istream& i, self& t) {
  ++self::counters[ISTREAM];
  return i >> t.base_value;
}
◇
```

Used in part 78.

```
⟨value_counter 78⟩ ≡

    ⟨value_counter template parameter list 69c⟩

    class value_counter {

      ⟨value_counter typedef self 69d⟩

      ⟨counter index enumeration 24b⟩
      ⟨declare and register counters 40a⟩

      Value base_value;
      ⟨declare generation 40c⟩

      ⟨value_counter uncounted constructor 72b⟩

     public:
      ⟨value_counter default constructor 71a⟩
      ⟨value_counter copy constructor 71b⟩
      ⟨value_counter destructor 71c⟩
      ⟨value_counter external constructor 71d⟩
      ⟨value_counter base 72a⟩

      ⟨value_counter operator= , assignment 72c⟩
      ⟨value_counter operator==, equal 73a⟩
      ⟨value_counter operator< , less 73b⟩
      ⟨value_counter operator++, preincrement 73c⟩
      ⟨value_counter operator++, postincrement 73d⟩
      ⟨value_counter operator--, predecrement 73e⟩
      ⟨value_counter operator--, postdecrement 74a⟩
      ⟨value_counter operator<<, left shift 74b⟩
      ⟨value_counter operator<<=, left shift assign 74c⟩
      ⟨value_counter operator>>, right shift 74d⟩
      ⟨value_counter operator>>=, right shift assign 74e⟩
      ⟨value_counter operator+ , addition 75a⟩
      ⟨value_counter operator+=, addition assign 75b⟩
      ⟨value_counter operator- , subtraction 75c⟩
      ⟨value_counter operator-=, subtraction assign 75d⟩
      ⟨value_counter operator* , multiply 76a⟩
      ⟨value_counter operator*=, multiply assign 76b⟩
      ⟨value_counter operator/ , divide 76c⟩
      ⟨value_counter operator/=, divide assign 76d⟩
      ⟨value_counter operator<<, ostream 77a⟩
      ⟨value_counter operator>>, istream 77b⟩

    };

    ⟨value_counter counter variable definition 70⟩
    ◇
```

Used in part 85a.

4.5.5 Function Counting Adaptors

Generator Counting Adaptor

The `generator_counter` class is virtually identical to `unary_function_counter`. For details on the implementation the reader should refer to the discussion of that class in section 4.5, page 39.

⟨`generator_counter` template parameter list 79a⟩ ≡

```
template <class AdaptableGenerator,
          COUNTERS_NAME_DECL_DEFAULT
          class Counter = DEFAULT_COUNTER_TYPE>
◇
```

Used in part 80c.

⟨`generator_counter` counter variable definition 79b⟩ ≡

```
template <class AdaptableGenerator,
          COUNTERS_NAME_DECL
          class Counter>
counter_vector<Counter>
generator_counter<AdaptableGenerator,
                  COUNTERS_NAME_PARM
                  Counter>::
counters(COUNTERS_NAME_TEST ?
             "generator_counter" : "",
         NCOUNTERS);
◇
```

Used in part 80c.

⟨`generator_counter` constructors 79c⟩ ≡

```
generator_counter () : base_function(), generation(1) {
  ++counters[DEFAULT_CTOR];
}
generator_counter (const AdaptableGenerator& x)
  : base_function(x), generation(1) {
  ++counters[OTHER_CTOR];
}
◇
```

Used in part 80c.

⟨`generator_counter` copy constructor 80a⟩ ≡

```
generator_counter (const self& x)
  : base_function(x.base_function), generation(x.generation + 1) {
  ++counters[COPY_CTOR];
  counters[GENERATION] = max(counters[GENERATION], generation);
}
◇
```

Used in part 80c.

⟨`generator_counter` counting operations 80b⟩ ≡

```
self& operator= (const self& rhs) {
  ++counters[ASSIGNMENT];
  base_function = rhs.base_function;
  return *this;
}
~generator_counter () { ++counters[DTOR]; }

result_type operator() () const {
  ++counters[FUNCTION_CALL];
  return base_function();
}
◇
```

Used in part 80c.

⟨`generator_counter` 80c⟩ ≡

```
⟨generator_counter template parameter list 79a⟩
class generator_counter {
  typedef typename AdaptableGenerator::result_type result_type;
  typedef generator_counter<AdaptableGenerator, Counter> self;

  ⟨counter index enumeration 24b⟩
  ⟨declare and register counters 40a⟩

  AdaptableGenerator base_function;
  ⟨declare generation 40c⟩

 public:
  ⟨generator_counter constructors 79c⟩
  ⟨generator_counter copy constructor 80a⟩
  ⟨generator_counter counting operations 80b⟩
};

⟨generator_counter counter variable definition 79b⟩
◇
```

Used in part 85a.

Unary Function Counting Adaptor

⟨**unary_function_counter** template parameter list 81a⟩ ≡

```
template <class AdaptableUnaryFunction,
          COUNTERS_NAME_DECL_DEFAULT
          class Counter = DEFAULT_COUNTER_TYPE>
```
◇

Used in part 82a.

⟨**unary_function_counter** constructors 81b⟩ ≡

```
unary_function_counter () : base_function(), generation(1) {
  ++counters[DEFAULT_CTOR];
}
unary_function_counter (const AdaptableUnaryFunction& x)
  : base_function(x), generation(1) {
  ++counters[OTHER_CTOR];
}
```
◇

Used in part 82a.

⟨**unary_function_counter** copy constructor 81c⟩ ≡

```
unary_function_counter (const self& x)
  : base_function(x.base_function), generation(x.generation + 1) {
  ++counters[COPY_CTOR];
  counters[GENERATION] = max(counters[GENERATION], generation);
}
```
◇

Used in part 82a.

⟨**unary_function_counter** counting operations 81d⟩ ≡

```
self& operator= (const self& rhs) {
  ++counters[ASSIGNMENT];
  base_function = rhs.base_function;
  return *this;
}
~unary_function_counter () { ++counters[DTOR]; }

result_type operator() (const argument_type& arg) const {
  ++counters[FUNCTION_CALL];
  return base_function(arg);
}
```
◇

Used in part 82a.

⟨`unary_function_counter` 82a⟩ ≡

```
⟨unary_function_counter template parameter list 81a⟩
class unary_function_counter
  : public unary_function<
        typename AdaptableUnaryFunction::argument_type,
        typename AdaptableUnaryFunction::result_type> {

  typedef unary_function_counter<AdaptableUnaryFunction,
                                 Counter> self;

  ⟨counter index enumeration 24b⟩
  ⟨declare and register counters 40a⟩

  AdaptableUnaryFunction base_function;
  ⟨declare generation 40c⟩

 public:
  ⟨unary_function_counter constructors 81b⟩
  ⟨unary_function_counter copy constructor 81c⟩
  ⟨unary_function_counter counting operations 81d⟩
};

⟨unary_function_counter counter variable definition 40b⟩
◇
```

Used in part 85a.

Binary Function Counting Adaptor

The `binary_function_counter` class is virtually identical to `unary_function_counter`. For details on the implementation of counters in general, the reader should refer to the discussion in section 4.5, page 39.

⟨`binary_function_counter` template parameter list 82b⟩ ≡

```
template <class AdaptableBinaryFunction,
          COUNTERS_NAME_DECL_DEFAULT
          class Counter = DEFAULT_COUNTER_TYPE>
◇
```

Used in part 84b.

⟨`binary_function_counter` counter variable definition 83a⟩ ≡

```
template <class AdaptableBinaryFunction,
          COUNTERS_NAME_DECL
          class Counter>
counter_vector<Counter>
binary_function_counter<AdaptableBinaryFunction,
                        COUNTERS_NAME_PARM
                        Counter>::
counters(COUNTERS_NAME_TEST ?
             "binary_function_counter" : "",
         NCOUNTERS);
```
◇

Used in part 84b.

⟨`binary_function_counter` constructors 83b⟩ ≡

```
binary_function_counter () : base_function(), generation(1) {
  ++counters[DEFAULT_CTOR];
}
binary_function_counter (const AdaptableBinaryFunction& x)
  : base_function(x), generation(1) {
  ++counters[OTHER_CTOR];
}
```
◇

Used in part 84b.

⟨`binary_function_counter` copy constructor 83c⟩ ≡

```
binary_function_counter (const self& x)
  : base_function(x.base_function), generation(x.generation + 1) {
  ++counters[COPY_CTOR];
  counters[GENERATION] = max(counters[GENERATION], generation);
}
```
◇

Used in part 84b.

⟨binary_function_counter counting operations 84a⟩ ≡

```
self& operator= (const self& rhs) {
  ++counters[ASSIGNMENT];
  base_function = rhs.base_function;
  return *this;
}
~binary_function_counter () { ++counters[DTOR]; }

result_type operator() (const first_argument_type& x,
                        const second_argument_type& y) const {
  ++counters[FUNCTION_CALL];
  return base_function(x, y);
}
◇
```

Used in part 84b.

⟨binary_function_counter 84b⟩ ≡

```
⟨binary_function_counter template parameter list 82b⟩
class binary_function_counter
  : public binary_function<
        typename AdaptableBinaryFunction::first_argument_type,
        typename AdaptableBinaryFunction::second_argument_type,
        typename AdaptableBinaryFunction::result_type> {

  typedef binary_function_counter<AdaptableBinaryFunction,
                                  Counter> self;

  ⟨counter index enumeration 24b⟩
  ⟨declare and register counters 40a⟩

  AdaptableBinaryFunction base_function;
  ⟨declare generation 40c⟩

 public:
  ⟨binary_function_counter constructors 83b⟩
  ⟨binary_function_counter copy constructor 83c⟩
  ⟨binary_function_counter counting operations 84a⟩
};

⟨binary_function_counter counter variable definition 83a⟩
◇
```

Used in part 85a.

4.6 Summary

"counters.h" 85a ≡

```
#ifndef COUNTERS_H
#define COUNTERS_H

#include <iostream>
#include <iomanip>
#include <string>
#include <typeinfo>
#include <algorithm>
#include <functional>
#include <numeric>
#include <map>
#include <vector>

⟨printable names bug definitions 24a⟩
⟨default counter type 19b⟩

⟨median_element functor 33⟩

⟨counter_vector 25b⟩
⟨recorder 34b⟩
⟨difference_counter 59b⟩
⟨iterator_counter 47c⟩
⟨value_counter 78⟩
⟨generator_counter 80c⟩
⟨unary_function_counter 82a⟩
⟨binary_function_counter 84b⟩

#endif  // COUNTERS_H
```

◇

4.7 Makefile

"Makefile" 85b ≡

```
# -*-Makefile-mode-*-
PROG=counters
TEST_SRC=test-cov.cc test-use.cc test-comp.cc test-inst.cc \
         test-corr.cc
SRC=Makefile $(PROG).h $(PROG).cc $(PROG).tex $(TEST_SRC)

CXXFLAGS=-g -Wall
CLEANUP = perl -i -ne 'print if (!/^\#line/ && (/\S/ .. !/\S/))'

# TEST_ROPE produces many warnings on gcc.
# TEST_HASH is a problem when hash<T> is not defined
# TEST_INVALID_VOID produces warnings about void exprs
```

```
# TEST_TEMPORARY_BUFFER produces warnings about
#                        temporary_buffer<T>
ALL_TEST_COV_FLAGS = -DTEST_ROPE -DTEST_HASH -DTEST_INVALID_VOID \
                     -DTEST_TEMPORARY_BUFFER
TEST_COV_FLAGS=-DTEST_HASH -DTEST_ROPE

.PHONY: all test doc dvi ps preview pspreview clean nuweb

all: $(PROG) doc nuweb doc

$(PROG): $(PROG).h $(PROG).cc
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $@ $@.cc

src source: counters.h $(TEST_SRC)
        $(CLEANUP) counters.h $(TEST_SRC)

test: test-cov test-use test-comp test-inst test-corr
test-cov: $(PROG).h test-cov.cc
        $(CXX) $(CXXFLAGS) $(TEST_COV_FLAGS) $(LDFLAGS) \
        -o $@ $@.cc
test-use: $(PROG).h test-use.cc
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $@ $@.cc
test-comp: $(PROG).h test-use.cc
        $(CXX) $(CXXFLAGS) $(LDFLAGS) \
        -DCOUNTERS_TEST_USE_TYPE='value_counter<int>' \
        -o $@ test-use.cc
test-inst: $(PROG).h test-inst.cc
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $@ $@.cc
test-corr: $(PROG).h test-corr.cc
        $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $@ $@.cc

doc: dvi
dvi: $(PROG).dvi
$(PROG).dvi: $(PROG).tex
        latex $(PROG).tex
        latex $(PROG).tex
ps: $(PROG).ps
$(PROG).ps: $(PROG).dvi
        dvips -o $@ $(PROG).dvi
preview: dvi
        xdvi $(PROG).dvi
pspreview: ps
        gv $(PROG).ps

clean:
        $(RM) core $(PROG){.??*,.h,} *{~,.o}

$(SRC) nuweb: $(PROG).w
        nuweb $(PROG).w
```
◇

Chapter 5

Test Plan and Results

All testing was performed independently on three platforms: Red Hat Linux 5.1 with `egcs 2.90.29`, Solaris 2.6 with `g++ 2.8.1` and Debian Linux 2.1 with `egcs-2.91.58`. Results were consistent on each platform and thus are not generally reported separately.

5.1 Usability

In order to assess the usability of the library, the tsort2 program for analysis of sort algorithms from [1] was modified to use the new library. This proved to be a reasonably simple task. The required changes are described below.

The original implementation included several files to define the counting adaptors. Those are all now defined in `counters.h`. This results in a short list of included headers.

⟨usability test include files 87⟩ ≡

```
#include <assert.h>
#include <algo.h>
#include <vector.h>
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include "counters.h"
◇
```

Used in part 89f.

The original implementation had four rather complicated typedefs to define the counting adaptor types. The new version eliminates the need to refer to the `difference_counter` explicitly and also simplifies the form of the other typedefs. The user does not need to specify as many redundant template parameters.

⟨usability test typedefs 88a⟩ ≡

```
#ifndef COUNTERS_TEST_USE_TYPE
#  define COUNTERS_TEST_USE_TYPE int
#endif

typedef COUNTERS_TEST_USE_TYPE U;
typedef value_counter<U> cdata;
typedef iterator_counter<vector<cdata >::iterator> citer;
◇
```

Used in part 89f.

The type, `U`, that is used to test the sorting is conditionalized primarily in order to facilitate the composition test (see 5.4 on page 111). Because of the way the vector is initialized, the type must have a constructor taking an `int` argument and must be able to be compared to an `int`.

The goal of the original implementation was to compare three sort algorithms: Introsort [2], Quicksort and heap sort. This was done using the STL sort algorithm as Quicksort. Since newer versions of STL have replaced that with Introsort, this was no longer reasonable. Thus, for the purposes of this test, the first two algorithms are both Introsort. This is not due to a limitation of the operation counting software but simply to the fact that a good Quicksort function was not available.

⟨usability test algorithm definitions 88b⟩ ≡

```
template <class Container>
void algorithm (int k, Container& x) {
  switch (k) {
//   case 0: introsort(citer(x.begin()), citer(x.end()));
  case 0: sort(citer(x.begin()), citer(x.end()));
     break;
  case 1: sort(citer(x.begin()), citer(x.end()));
     break;
  case 2: partial_sort(citer(x.begin()), citer(x.end()),
                       citer(x.end()));
     break;
  }
}

const char* const headings[number_of_algorithms] = {
  "Introsort        ",
  "Introsort        ",
  "Heapsort         "
};
◇
```

Used in part 89f.

The random number generator used in the original code was not necessarily portable. This was replaced by instantiation of the STL subtractive random number generator.

⟨usability test random number generator 89a⟩ ≡

```
subtractive_rng random(S);
◇
```

Used in part 89f.

The definition of the array of `recorder`'s used to track the statistics requires no template parameters compared with five in the original program.

⟨usability test recorder array 89b⟩ ≡

```
recorder<> stats[number_of_algorithms];
◇
```

Used in part 89f.

In a number of places the original code manipulated the operation counts manually in order to adjust for setup operations that were not part of the actual sort algorithms. That code has been replaced with the `recorder`'s `pause()/resume()` idiom.

⟨usability test randomize test vector 89c⟩ ≡

```
stats[0].pause(); {
  random_shuffle(x.begin(), x.end(), random);
  y = x;
} stats[0].resume();
◇
```

Used in part 89f.

⟨usability test restore test vector 89d⟩ ≡

```
stats[n].pause(); {
  x = y;
} stats[n].resume();
◇
```

Used in part 89f.

⟨usability test verify sorting 89e⟩ ≡

```
stats[n].pause(); {
  for (int z = 0; z < N; ++z)
    assert(x[z] == U(z));
} stats[n].resume();
◇
```

Used in part 89f.

Finally, the reporting code was changed to use the output operator rather than the `report()` method. The final program is shown below. It is identical to the original except as described in the preceding paragraphs.

`"test-use.cc"` 89f ≡

```
/*

Example program for measuring the computing time of
algorithms.  This program both measures times and counts
operations; for a simpler example of only measuring times,
see tsort1.cpp

*/

/*
 * Copyright (c) 1997 Rensselaer Polytechnic Institute
 *
 * Permission to use, copy, modify, distribute and sell
 * this software and its documentation for any purpose is
 * hereby granted without fee, provided that the above
 * copyright notice appear in all copies and that both that
 * copyright notice and this permission notice appear in
 * supporting documentation.  Rensselaer Polytechnic
 * Institute makes no representations about the suitability
 * of this software for any purpose.  It is provided "as
 * is" without express or implied warranty.
 */

⟨usability test include files 87⟩
⟨usability test typedefs 88a⟩

const int number_of_algorithms = 3;
const int number_of_trials = 7;

⟨usability test algorithm definitions 88b⟩

int main () {
  int N0, N1, N2, N, S;

  const int factor = 1000;

  cout << "All sequence sizes are in multiples of "
       << factor << ".\n";
  cout << "Input the smallest and largest sequence sizes: "
       << flush;
  cin >> N1 >> N2;

  cout << "Input random seed control (a positive integer): ";
  cin >> S;

  ⟨usability test random number generator 89a⟩

  ofstream ofs1("read.dat");
  ofstream ofs2("graph.dat");
```

⟨usability test recorder array 89b⟩

```
vector<cdata> x, y;
x.reserve(N2 * factor);
y.reserve(N2 * factor);

int repetitions = max(32 / N1, 1);
int n;

cout << endl;
ofs1 << endl;

for (N0 = N1; N0 <= N2; N0 *= 2) {
  N = N0 * factor;

  cout << "Size: " << setw(4) << N0 << "   Trials: " << flush;
  ofs1  << "Size: " << setw(4) << N0 << endl << flush;
  ofs2  << setw(4) << N0 << "\t" << flush;

  for (int i = 0; i < N; ++i)
    x.push_back(U(i));

  int p, q;
  for (p = 0; p < number_of_trials; ++p) {
    ⟨usability test randomize test vector 89c⟩

    cout << (p + 1) << ":" << flush;

    for (n = 0; n < number_of_algorithms; ++n) {
      for (q = 0; q < repetitions; ++q) {
        ⟨usability test restore test vector 89d⟩
        algorithm(n, x);
      }

      ⟨usability test verify sorting 89e⟩

      cout << (n + 1) << flush;
      stats[n].record();
    }
    cout << ", " << flush;
  }

  for (n = 0; n < number_of_algorithms; ++n) {
    cout << headings[n] << endl;
    stats[n].report_headings(cout);
    stats[n].report_table(cout,
        &median_element<counter_vector<>::const_iterator>);
    stats[n].report_totals(cout,
        &median_element<counter_vector<>::const_iterator>);
```

```
          ofs1 << headings[n] << endl;
          stats[n].report_headings(ofs1);
          stats[n].report_table(ofs1,
              &median_element<counter_vector<>::const_iterator>);
          stats[n].report_totals(ofs1,
              &median_element<counter_vector<>::const_iterator>);

          stats[n].report_totals(ofs2,
              &median_element<counter_vector<>::const_iterator>);
        }

        cout << endl;
        ofs1 << endl;
        ofs2 << endl;

        x.erase(x.begin(), x.end());

        if (repetitions > 1)
          repetitions /= 2;
      }
      return 0;
    }
```
◇

This test is by nature a very subjective "test." Its success or failure is defined by whether the new library is "easier to use" than the original. Although this judgement is a matter of opinion, there are a number of factual observations that can be made.

The new source code has 12 percent fewer lines and 22 percent fewer characters (since many code lines were simplified). Additionally the simple fact that most of the program remained unchanged is an indicator that using the new library does not require a large amount of user effort.

5.2 Coverage

One of the constraints on the library is that the counting adaptors must implement a sufficient set of operations to be useful within the context of STL. For example, the STL `inner_product()` function requires a type that supports addition and multiplication. Thus, it is important that the `value_counter` provide counting versions of those operations.

One way to verify this is to write a program that instantiates each STL data structure and calls each STL algorithm at least once using counting adaptors whenever possible. This was done by defining a variety of template functions that make it easy to create the structures and invoke the algorithms on a variety of data types.

It should be emphasized that the goal of this test is merely to force the compiler to reference each data structure and algorithm. Semantically the invocations may not make sense. In fact, it is most likely that if this program were executed it would either abort or would not terminate at all. This test succeeds simply when the program compiles cleanly.

To create and test the container types, two functions are used. The first creates one of each of the basic STL containers. The second function takes a particular container type and performs the relevant tests — in this case creating counting versions of each of the container's iterator types.

⟨coverage test of containers 93a⟩ ≡

```
template <class T>
void instantiate_containers () {
  instantiate_container(vector<T>());
  instantiate_container(deque<T>());
  instantiate_container(list<T>());
  instantiate_container(slist<T>());
  instantiate_container(bit_vector());
  instantiate_container(set<T>());
  instantiate_container(map<T, T>());
  instantiate_container(multiset<T>());
  instantiate_container(multimap<T, T>());
#ifdef TEST_HASH
  instantiate_container(hash_set<T>());
  instantiate_container(hash_map<T, T>());
  instantiate_container(hash_multiset<T>());
  instantiate_container(hash_multimap<T, T>());
#endif
  instantiate_container(basic_string<char>());
#ifdef TEST_ROPE
  instantiate_container(rope<char>());
#endif
}
◇
```

Used in part 104.

⟨coverage test of a single container 93b⟩ ≡

```
template <class Container>
void instantiate_container (const Container&) {
  volatile iterator_counter<typename Container::iterator> i;
  (void)i;
  volatile iterator_counter<typename Container::const_iterator> ci;
  (void)ci;
}
◇
```

Used in part 104.

Note the use of `volatile` here and elsewhere to ensure that the compiler does not optimize the unused variables out of the program.

Next the program tests the iterator manipulation functions, `distance()` and `advance()`. This is done by three functions. Since the iterator functions may be implemented differently for each of the STL iterator concepts, it is important to use each of those five concepts. The first test function does this. The second and third functions take a particular container or iterator type and invoke the iterator functions for that type.

⟨coverage test of iterator functions 94a⟩ ≡

```
template <class T>
void instantiate_iterator_functions () {
  instantiate_iterator_functions(istream_iterator<T>(cin),
                                 istream_iterator<T>());  // input
#ifdef TEST_HASH
  instantiate_iterator_functions(hash_set<T>());  // forward
#endif
  instantiate_iterator_functions(list<T>());  // reversible
  instantiate_iterator_functions(vector<T>());  // random access
}
◇
```

Used in part 104.

⟨coverage test of iterator functions on a container 94b⟩ ≡

```
template <class Container>
void instantiate_iterator_functions (Container x) {
  instantiate_iterator_functions(x.begin(), x.end());
}
◇
```

Used in part 104.

⟨coverage test of iterator functions on a given iterator 94c⟩ ≡

```
template <class InputIterator>
void instantiate_iterator_functions (const InputIterator& begin,
                                     const InputIterator& end) {
  iterator_counter<InputIterator> cbegin(begin), cend(end);
  distance(cbegin, cend);
  advance(cbegin, 1);
}
◇
```

Used in part 104.

Next the built-in iterator adaptor types are tested. Most of these are parameterized on the `value_type` of the iterator. In cases where the template parameter is a container or another iterator type, `list` is chosen arbitrarily as the base type.

⟨coverage test of iterator adaptors 95⟩ ≡

```
template <class T>
void instantiate_iterators () {
  typedef list<T> Container;
  typedef iterator_counter<Container::iterator> Iterator;
  Container x;
  volatile iterator_counter<istream_iterator<T> > i1(cin);
  (void)i1;
  volatile iterator_counter<ostream_iterator<T> > i2(cout);
  (void)i2;
  volatile iterator_counter<front_insert_iterator<Container> >
      i3(front_insert_iterator<Container>(x)); (void)i3;
  volatile iterator_counter<back_insert_iterator<Container> >
      i4(back_insert_iterator<Container>(x)); (void)i4;
  volatile iterator_counter<insert_iterator<Container> >
      i5(insert_iterator<Container>(x, x.begin())); (void)i5;
  volatile iterator_counter<reverse_iterator<Iterator> >
      i6(reverse_iterator<Iterator>()); (void)i6;
  volatile iterator_counter<raw_storage_iterator<Iterator, T> >
      i7(raw_storage_iterator<Iterator, T>(x.begin())); (void)i7;
#ifdef TEST_ROPE
  volatile iterator_counter<sequence_buffer<Container> >
      i8(sequence_buffer<Container>(x)); (void)i8;
#endif
}
◇
```

Used in part 104.

The largest portion of this test is algorithm instantiation. Like the iterator manipulation functions, some of the algorithms may have different implementations for different iterator concepts. Indeed some of the algorithms are restricted to only a subset of the iterator concepts. For this reason, as is explained in more detail below, the `iterator_traits` mechanism is used to determine an iterator's category.

The first function instantiates the algorithms for one of each iterator concept. In addition, there are several algorithms that are defined only for numeric types. These are treated separately in
`instantiate_numeric_algorithms()`.

⟨coverage test of algorithms 96a⟩ ≡

```
template <class T>
void instantiate_algorithms () {
  instantiate_algorithms(istream_iterator<T>(cin),
                         istream_iterator<T>());  // input
#ifdef TEST_HASH
  instantiate_algorithms(hash_set<T>());  // forward
#endif
  instantiate_algorithms(list<T>());  // reversible
  instantiate_algorithms(vector<T>());  // random access
  instantiate_numeric_algorithms();
}
◇
```

Used in part 104.

⟨coverage test of numeric algorithms 96b⟩ ≡

```
void instantiate_numeric_algorithms () {
  typedef value_counter<int> value_type;
  vector<value_type> x;
  iterator_counter<vector<value_type>::iterator> begin(x.begin()),
                                                 end(x.end());
  ostream_iterator<value_type> out(cout);

  iota(begin, end, value_type());
  accumulate(begin, end, value_type(0));
  accumulate(begin, end, value_type(0), plus<value_type>());
  inner_product(begin, end, begin, value_type(0));
  inner_product(begin, end, begin, value_type(0),
                plus<value_type>(), multiplies<value_type>());
  partial_sum(begin, end, out);
  partial_sum(begin, end, out, plus<value_type>());
  adjacent_difference(begin, end, out);
  adjacent_difference(begin, end, out, minus<value_type>());
#if 0
  power(value_type(2), value_type(2));
  power(value_type(2), value_type(2), multiplies<value_type>());
#endif
}
◇
```

Used in part 104.

The algorithm instantiation functions parameterized on `Container` and `Iterator` types simply compute the iterator's category and invoke the appropriate sub-function.

⟨coverage test of algorithms on a container 97a⟩ ≡

```
template <class Container>
void instantiate_algorithms (Container x) {
  instantiate_algorithms(x.begin(), x.end());
}
```
◇

Used in part 104.

⟨coverage test of algorithms on a given iterator 97b⟩ ≡

```
template <class Iterator>
void instantiate_algorithms (const Iterator& begin,
                             const Iterator& end) {
  iterator_counter<Iterator> cbegin(begin), cend(end);
  instantiate_algorithms(cbegin, cend,
                         iterator_traits<Iterator>::
                         iterator_category());
}
```
◇

Used in part 104.

The is a specialized algorithm instantiation function for each iterator category. Since `Random Access Iterators` are also `Bidirectional Iterators` the random access iterator function also calls the bidirectional iterator function. Similarly the bidirectional iterator function calls the forward iterator function and so on, chaining down to the level of input iterators. Each function invokes only those functions that are specifically defined for that particular iterator category.

⟨coverage test of algorithms on random access iterators 98⟩ ≡

```
template <class Iterator>
void instantiate_algorithms (Iterator& begin, Iterator& end,
                             random_access_iterator_tag) {
  typedef typename Iterator::value_type value_type;
  typedef equal_to<value_type> predicate_type;
  predicate_type f;
  binary_function_counter<predicate_type> bp(f);
  unary_function_counter<binder2nd<predicate_type> >
    up(bind2nd(f, value_type()));

#ifdef TEST_INVALID_VOID
  random_shuffle(begin, end); // random
  random_sample(begin, end, begin, end); // 1st pair fwd,
                                         // 2nd pair random
#endif
  sort(begin, end); // random
  sort(begin, end, bp); // random
  stable_sort(begin, end); // random
  stable_sort(begin, end, bp); // random
  partial_sort(begin, end, end); // random
  partial_sort(begin, end, end, bp); // random
  partial_sort_copy(begin, end, begin, end); // 1st pair input,
                                             // 2nd pair random
  partial_sort_copy(begin, end, begin, end, bp); //   "    "
  nth_element(begin, end, end); // random
  nth_element(begin, end, end, bp); // random
  push_heap(begin, end); // random
  push_heap(begin, end, bp); // random
  pop_heap(begin, end); // random
  pop_heap(begin, end, bp); // random
#ifdef TEST_INVALID_VOID
  make_heap(begin, end); // random
  make_heap(begin, end, bp); // random
#endif
  sort_heap(begin, end); // random
  sort_heap(begin, end, bp); // random
  is_heap(begin, end); // random
  is_heap(begin, end, bp); // random

  instantiate_algorithms(begin, end, bidirectional_iterator_tag());
}
◇
```

Used in part 104.

⟨coverage test of algorithms on bidirectional iterators 99a⟩ ≡

```
template <class Iterator>
void instantiate_algorithms (Iterator& begin, Iterator& end,
                             bidirectional_iterator_tag) {
  typedef typename Iterator::value_type value_type;
  typedef equal_to<value_type> predicate_type;
  predicate_type f;
  binary_function_counter<predicate_type> bp(f);
  unary_function_counter<binder2nd<predicate_type> >
    up(bind2nd(f, value_type()));
  ostream_iterator<value_type> out(cout);

  copy_backward(begin, end, end); // bidir
  reverse(begin, end); // bidir
  reverse_copy(begin, end, out); // bidir
#ifdef TEST_TEMPORARY_BUFFER
  inplace_merge(begin, end, end); // bidir
  inplace_merge(begin, end, end, bp); // bidir
#endif
  next_permutation(begin, end); // bidir
  next_permutation(begin, end, bp); // bidir
  prev_permutation(begin, end); // bidir
  prev_permutation(begin, end, bp); // bidir

  instantiate_algorithms(begin, end, forward_iterator_tag());
}
◇
```

Used in part 104.

⟨coverage test of algorithms on forward iterators 99b⟩ ≡

```
template <class Iterator>
void instantiate_algorithms (Iterator& begin, Iterator& end,
                             forward_iterator_tag) {
  typedef typename Iterator::value_type value_type;
  typedef equal_to<value_type> predicate_type;
  predicate_type f;
  binary_function_counter<predicate_type> bp(f);
  unary_function_counter<binder2nd<predicate_type> >
    up(bind2nd(f, value_type()));
  ostream_iterator<value_type> out(cout);

  adjacent_find(begin, end); // fwd
  adjacent_find(begin, end, bp); // fwd
  find_first_of(begin, end, begin, end); // 2nd pair fwd
  find_first_of(begin, end, begin, end, bp); // 2nd pair fwd
  search(begin, end, begin, end); // fwd
  search(begin, end, begin, end, bp); // fwd
  search_n(begin, end, 1, value_type()); // fwd
```

```
  search_n(begin, end, 1, value_type(), bp); // fwd
  find_end(begin, end, begin, end); // fwd
  find_end(begin, end, begin, end, bp); // fwd
  copy_backward(begin, end, end); // bidir
  iter_swap(begin, end);
  swap_ranges(begin, end, begin); // fwd
  replace(begin, end, value_type(), value_type()); // fwd
  replace_if(begin, end, up, value_type()); // fwd
  fill(begin, end, value_type()); // fwd
  fill_n(out, 1, value_type());
//   generate(begin, end, /* generator */); // fwd // !!!
  remove(begin, end, value_type()); // fwd
  remove_if(begin, end, up); // fwd
  unique(begin, end); // fwd
  unique(begin, end, bp); // fwd
  rotate(begin, end, end); // fwd
  rotate_copy(begin, end, end, out); // fwd
  random_sample_n(begin, end, out, 1); // fwd
  partition(begin, end, up); // fwd
#ifdef TEST_TEMPORARY_BUFFER
  stable_partition(begin, end, up); // fwd
#endif
  is_sorted(begin, end); // fwd
  is_sorted(begin, end, bp); // fwd
  lower_bound(begin, end, value_type()); // fwd
  lower_bound(begin, end, value_type(), bp); // fwd
  upper_bound(begin, end, value_type()); // fwd
  upper_bound(begin, end, value_type(), bp); // fwd
  equal_range(begin, end, value_type()); // fwd
  equal_range(begin, end, value_type(), bp); // fwd
  binary_search(begin, end, value_type()); // fwd
  binary_search(begin, end, value_type(), bp); // fwd
  min_element(begin, end);
  min_element(begin, end, bp);
  max_element(begin, end);
  max_element(begin, end, bp);

  instantiate_algorithms(begin, end, input_iterator_tag());
}
◇
```

Used in part 104.

⟨coverage test of algorithms on input iterators 100⟩ ≡

```
template <class Iterator>
void instantiate_algorithms (Iterator& begin, Iterator& end,
                             input_iterator_tag) {
  typedef typename Iterator::value_type value_type;
  typedef equal_to<value_type> predicate_type;
  predicate_type f;
```

```
    binary_function_counter<predicate_type> bp(f);
    unary_function_counter<binder2nd<predicate_type> >
        up(bind2nd(f, value_type()));
    ostream_iterator<value_type> out(cout);

    for_each(begin, end, up);
    find(begin, end, value_type());
    find_if(begin, end, up);
    count(begin, end, value_type());
    count_if(begin, end, up);
    mismatch(begin, end, begin);
    mismatch(begin, end, begin, bp);
    equal(begin, end, begin);
    equal(begin, end, begin, bp);
    copy(begin, end, out);
    copy_n(begin, 1, out);
    swap(begin, end);
    transform(begin, end, out, identity<value_type>());
    transform(begin, end, begin, out,
              project1st<value_type, value_type>());
    replace_copy(begin, end, out, value_type(), value_type());
    replace_copy_if(begin, end, out, up, value_type());
//   generate_n(out, 1, /* generator */); // !!!
    remove_copy(begin, end, out, value_type());
    remove_copy_if(begin, end, out, up);
    unique_copy(begin, end, out);
    unique_copy(begin, end, out, bp);
    merge(begin, end, begin, end, out);
    merge(begin, end, begin, end, out, bp);
    includes(begin, end, begin, end);
    includes(begin, end, begin, end, bp);
    set_union(begin, end, begin, end, out);
    set_union(begin, end, begin, end, out, bp);
    set_intersection(begin, end, begin, end, out);
    set_intersection(begin, end, begin, end, out, bp);
    set_difference(begin, end, begin, end, out);
    set_difference(begin, end, begin, end, out, bp);
    set_symmetric_difference(begin, end, begin, end, out);
    set_symmetric_difference(begin, end, begin, end, out, bp);
    min(value_type(), value_type());
    min(value_type(), value_type(), bp);
    max(value_type(), value_type());
    max(value_type(), value_type(), bp);
    lexicographical_compare(begin, end, begin, end);
    lexicographical_compare(begin, end, begin, end, bp);
    lexicographical_compare_3way(begin, end, begin, end);
}
◇
```

Used in part 104.

The last form of STL class is the function object. The function below instantiates each of these on an arbitrary type. Certain functions are defined only for numeric types. Those are instantiated on an operation counting `int`, regardless of the type parameter of the function.

⟨coverage test of function objects 102⟩ ≡

```
template <class T>
void instantiate_functions () {
  plus<value_counter<int> >();
  minus<value_counter<int> >();
  multiplies<value_counter<int> >();
  divides<value_counter<int> >();
  modulus<value_counter<int> >();
  negate<value_counter<int> >();
  equal_to<T>();
  not_equal_to<T>();
  less<T>();
  greater<T>();
  less_equal<T>();
  greater_equal<T>();
  logical_and<value_counter<int> >();
  logical_or<value_counter<int> >();
  logical_not<value_counter<int> >();
  identity<T>();
  project1st<T, T>();
  project2nd<T, T>();
  select1st<pair<T, T> >();
  select2nd<pair<T, T> >();
}
◇
```

Used in part 104.

Finally the test program is completed by a function that applies each of the individual instantiation functions to an arbitrary type. The `main()` function then calls this function on a wide variety of fundamental and container types, in both their counting and non-counting forms.

⟨coverage test on type T 103⟩ ≡

```
template <class T>
void instantiate () {
  instantiate_containers<T>();
  instantiate_iterator_functions<T>();
  instantiate_iterators<T>();
#ifdef TEST_INVALID_VOID
  instantiate_algorithms<T>();
#endif
  instantiate_functions<T>();
}
◇
```

Used in part 104.

"test-cov.cc" 104 ≡

```
#include <iostream>
#include <string>
#include <vector>
#include <deque>
#include <list>
#include <slist>
#include <bvector.h>
#include <set>
#include <map>
#ifdef TEST_HASH
#include <hash_set>
#include <hash_map>
#endif
#ifdef TEST_ROPE
#include <rope>
#endif
#include <stack>
#include <queue>
#include <iterator>
#include <algorithm>
#include <functional>

#include "counters.h"

⟨coverage test of a single container 93b⟩
⟨coverage test of containers 93a⟩
⟨coverage test of iterator functions on a given iterator 94c⟩
⟨coverage test of iterator functions on a container 94b⟩
⟨coverage test of iterator functions 94a⟩
⟨coverage test of iterator adaptors 95⟩
⟨coverage test of algorithms on input iterators 100⟩
⟨coverage test of algorithms on forward iterators 99b⟩
⟨coverage test of algorithms on bidirectional iterators 99a⟩
⟨coverage test of algorithms on random access iterators 98⟩
⟨coverage test of algorithms on a given iterator 97b⟩
⟨coverage test of algorithms on a container 97a⟩
⟨coverage test of numeric algorithms 96b⟩
⟨coverage test of algorithms 96a⟩
⟨coverage test of function objects 102⟩

⟨coverage test on type T 103⟩

int main () {
  instantiate<int>();
}
◇
```

One will notice that a number of portions of this test are conditionalized. In practice it is difficult to instantiate all of the STL algorithms, even when the adaptors are implemented correctly. The problems are discussed below.

The hash-based containers in STL require that `hash<T>` be defined for the container's `value_type`. This means that each instantiated type must have a specialized definition for `hash<T>`. Furthermore, the hash include file produces warnings from the compiler independent from the counting adaptor library. For this reason, all hash-related code is conditionalized (it is included by default), and the functions are instantiated only for the `int` type.

The `rope` include file also produces warnings from the compiler. Thus, the `rope` code is also conditionalized (but included by default). The test succeeds — the program compiles — when the hash and rope code is included, but it does not proceed free of warnings.

Instantiating many of the algorithms produces an inexplicable error message about a void expression. Isolating the problematic code into a smaller test case does not reproduce the message. Thus, all of the algorithms can be instantiated on an individual basis, just not all at once in a single program. This would appear to be a compiler limitation. For this reason, those troublesome functions are conditionalized temporarily until the compiler is more able to handle large, nested template invocations.

There is one legitimate omission that was uncovered with this test. `stable_sort()` and `inplace_merge()` use the `temporary_buffer` type. Instantiating this type with counting adaptors produces an error which has not yet been resolved. For this reason, those algorithms are not included in the test.

Finally, even with all of these problems controlled by conditional compilation, when more than one type is instantiated in the `main()` function, the compiler frequently runs out of memory. With all of the template instantiations involved, the processing required to produce an executable is simply too much for the compiler.

With all these caveats, it does appear that the library is successful in covering the required operations. The massive complexity of this test presents compilation problems, but they do not appear to be due to the library itself. When the test's size is controlled carefully, it succeeds. Rephrased, the problems appear to be related to the ambitiousness of the test itself rather than to defficiencies in the library.

5.3 Instantiation

Because member functions of template classes are not instantiated unless they are invoked, it is possible for a program to contain undetected errors unless each function is referenced. This forces the compiler to process the function, checking it for syntax errors. The program below does this for each of the counting adaptor types. Like the coverage test, this test succeeds if the program compiles successfully. It is not necessarily appropriate to execute the resulting program.

"test-inst.cc" 105 ≡

```
#include <vector>
#include "counters.h"

//    istream& operator>> (istream& i, int v) {
//cout << "int >>" << endl;
//      return i;
```

```
//}

int main()
{
  typedef vector<int> int_vector;
  typedef int_vector::iterator        vector_iterator;
  typedef int_vector::difference_type vector_difference;

  typedef iterator_counter<vector_iterator>
           counted_vector_iterator;

  typedef difference_counter<vector_difference, vector_iterator>
           counted_vector_difference;

  typedef value_counter<int> counted_int;

  recorder<> r;

  int_vector v;

  for (int i = 0; i < 5; i++) {
    v.push_back (i+100);
  };

  bool b;
  int i;
  int d;
  vector_iterator unadIter;

  // iterator_counter //
  //////////////////////

  // default constructor
  counted_vector_iterator def_const_iter;

  // external constructor
  counted_vector_iterator iter1(v.begin());

  // base
  unadIter = iter1.base();

  // copy constructor
  counted_vector_iterator iter2 = iter1;

  // Assignment
  iter1 = iter2;

  // Equality
  b = (iter1 == iter2);
```

```
// Less
b = (iter1 < iter2);

// dereference
i = *iter1;

//preincrement
++iter1;

//postincrement
iter1++;

//predecrement
--iter1;

//postdecrement
iter1--;


// difference_counter //
////////////////////////

// default constructor
counted_vector_difference def_const_diff;

// external constructor
counted_vector_difference diff1(v.end()-v.begin());

// copy constructor
counted_vector_difference diff2 = diff1;

vector_difference unaptdiff = diff1.base();
(void)unaptdiff; // use the variable

// Assignment
diff1 = diff2;

// Equality
b = (diff1 == diff2);

// Less
b = (diff1 < diff2);

//preincrement
++diff1;

//postincrement
diff1++;

//predecrement
```

```
--diff1;

//postdecrement
diff1--;

//multiply
diff2 = diff1 * 5;
diff2 = 5     * diff1;
diff2 = diff1 * diff1;

//multiply assign
diff2 *= 5;
d     *= diff1;
diff2 *= diff1;

//divide
diff2 = diff1 / 5;
diff2 = 5     / diff1;
diff2 = diff1 / diff1;

//divide assign
diff2 /= 5;
d     /= diff1;
diff2 /= diff1;

// ostream
cout << diff1 << endl;


// mixed operation //
/////////////////////

//addition
diff2    = diff1     + 5;
diff2    = 5         + diff1;
diff2    = diff1     + diff1;

iter2    = iter1     + 5;
unadIter = v.begin() + diff1;
iter2    = iter1     + diff1;

iter2    = 5         + iter1;
unadIter = diff1     + v.begin();
iter2    = diff1     + iter1;


//addition assign
diff2    += 5;
d        += diff1;
diff2    += diff1;
```

```
iter2    += 5;
unadIter += diff1;
iter2    += diff1;

//subtraction
diff2    = diff1     - 5;
diff2    = 5         - diff1;
diff2    = diff1     - diff1;

iter2    = iter1     - 5;
unadIter = v.begin() - diff1;
iter2    = iter1     - diff1;

diff1    = v.begin() - iter2;
diff1    = iter1     - v.begin();
diff1    = iter1     - iter2;

//subtraction assign
diff2    -= 5;
d        -= diff1;
diff2    -= diff1;

iter2    -= 5;
unadIter -= diff1;
iter2    -= diff1;


i = iter1[5];
i = iter1[diff1];

// value_counter //
///////////////////

// default constructor
counted_int default_int;

// external constructor
counted_int val1(1234);

// copy constructor
counted_int val2 = val1;

// Assignment
val1 = val2;

// base
int unadval = val1.base();
(void)unadval; // use the variable
```

```
// Equality
b = (val1 == val2);

// Less
b = (val1 < val2);

//preincrement
++val1;

//postincrement
val1++;

//predecrement
--val1;

//postdecrement
val1--;

//multiply
val2 = val1 * val1;

//multiply assign
val2 *= val1;

//divide
val2 = val1 / val1;

//divide assign
val2 /= val1;

//addition
val2    = val1     + val1;

//addition assign
val2    += val1;

//subtraction
val2    = val1     - val1;

//subtraction assign
val2    -= val1;

// ostream
cout << val1 << endl;

// istream
if (false)
   cin >> val2;
```

```
    r.record();
    cout << r;
}
```
◇

5.4 Composition

It is important that if operation `X` is invoked on a counting adaptor, the adaptor must in turn invoke operation `X`, and not some other operation `Y`, on its base type. One convenient way to test this is to compose a counting adaptor with another. In other words, rather than using a `value_counter<int>`, use `value_counter<value_counter<int> >`. Then the operation counts produced by the outer counter must equal those from the inner counter.

In practice it is not easy to obtain independent results for the inner and outer counters. However, there is another way to obtain the same result. First the test is run on a non-nested counting type, such as `value_counter<int>`. These results are used as a baseline case.

Then when the program is run with composed adaptors, each operation on the outer type will result in two operations being counted. First the outer type will count the operation and defer to its base type, the inner counter. Then the inner counter will count the operation and defer to its base type. Thus, the results from the composed adaptors should show exactly double the counts as are in the baseline case.

This test was performed by reusing the code from the usability test. First the program was run using adapted integers as the type to be sorted. Then it was rerun using composed adaptors. The resulting operation counts were in fact doubled in the composed case, indicating proper transparency of the operations.

5.5 Correctness

The most obvious test to perform is a correctness test. It is important to ensure that the reported operation counts give an accurate representation of the actions performed by the program. Unfortunately it is difficult to do this type of test for anything but a trivial case.

The entire motivation behind implementing operation counting adaptors is that it is difficult to determine from looking at an algorithm exactly what operations are invoked. It is easy to miss subtle, implicit calls to constructors, conversion operators and the like. Thus, the only correctness tests that can be readily performed are simple examples where it is immediately clear from a visual inspection of the code what is happening.

This code is centered around a function that performs one call to each of several operations.

⟨correctness test basis function 112a⟩ ≡

```
void do_operations () {
  ⟨correctness test value_counter operations 112b⟩
  ⟨correctness test iterator_counter operations 112c⟩
  ⟨correctness test difference_counter operations 112d⟩
  ⟨correctness test unary_function_counter operations 113a⟩
  ⟨correctness test binary_function_counter operations 113b⟩
}
◇
```

Used in part 114b.

⟨correctness test `value_counter` operations 112b⟩ ≡

```
value_counter<int> vc;  // default ctor
vc = value_counter<int>(vc);  // assignment, copy ctor
vc++;  // post-increment
// 2 object dtors, 2 implicit dtors in post-increment
◇
```

Used in part 112a.

⟨correctness test `iterator_counter` operations 112c⟩ ≡

```
iterator_counter<int*> ic;  // default ctor
ic = iterator_counter<int*>(ic);  // assignment, copy ctor
ic++;  // post-increment
// 2 object dtors, 2 implicit dtors in post-increment
◇
```

Used in part 112a.

⟨correctness test `difference_counter` operations 112d⟩ ≡

```
difference_counter<size_t, int*> dc;  // default ctor
// assignment, copy ctor
dc = difference_counter<size_t, int*>(dc);
dc++;  // post-increment
// 2 object dtors, 2 implicit dtors in post-increment
◇
```

Used in part 112a.

⟨correctness test `unary_function_counter` operations 113a⟩ ≡

```
identity<int> id;
unary_function_counter<identity<int> > ufc(id);  // other ctor
// assignment, copy ctor
ufc = unary_function_counter<identity<int> >(ufc);
ufc(0);  // function call
// 2 object dtors
◇
```

Used in part 112a.

⟨correctness test `binary_function_counter` operations 113b⟩ ≡

```
equal_to<int> eq;
binary_function_counter<equal_to<int> > bfc(eq);  // other ctor
// assignment, copy ctor
bfc = binary_function_counter<equal_to<int> >(bfc);
bfc(0, 0);  // function call
// 2 object dtors
◇
```

Used in part 112a.

This basis function is then invoked repeatedly. Given a number, N, for each integer $i < N$ the function is called i times and the counts are recorded. Thus, assuming the counting is working properly, the `recorder` should contain snapshots with 1, 2, 3, ... N as the operation counts. The `recorder`'s reporting function is then called to print median, minimum and maximum operation counts.

Of course, if the iterations are performed in order (1, 2, 3, ... N) the work of the `recorder`'s evaluation functions is trivial. Thus, their order is randomized.

⟨correctness test iteration randomization 113c⟩ ≡

```
vector<size_t> ni(N);
iota(ni.begin(), ni.end(), 1);
random_shuffle(ni.begin(), ni.end());
◇
```

Used in part 114b.

⟨correctness test iterations 114a⟩ ≡

```
recorder<> r;
for (size_t i = 0; i < ni.size(); ++i) {
  for (size_t j = 0; j < ni[i]; ++j)
    do_operations();
  r.record();
}
r.report_headings(cout);
r.report_table(cout,
               &median_element<counter_vector<>::const_iterator>);
r.report_table(cout,
               &min_element<counter_vector<>::const_iterator>);
r.report_table(cout,
               &max_element<counter_vector<>::const_iterator>);
◇
```

Used in part 114b.

"test-corr.cc" 114b ≡

```
#include <iostream>
#include "counters.h"

⟨correctness test basis function 112a⟩

int main() {
  size_t N = 10;
  ⟨correctness test iteration randomization 113c⟩
  ⟨correctness test iterations 114a⟩
}
◇
```

Chapter 6

Future Plans

- **User-Defined Operations**

  Any vector indices higher than `NCOUNTERS` are available for user-defined operations. For example, one could create a `Container` counter that tracks invocations of `begin()`, `end()`, `insert()` and `erase()`. Those counts could be placed in the vector at indices `NCOUNTERS`, `NCOUNTERS + 1`, `NCOUNTERS + 2` and `NCOUNTERS + 3` respectively. Totals will not be reported for user-defined operations, since one counter's use for an index may differ from another's.

Bibliography

[1] D. R. Musser. Measuring Computing Times of Generic Algorithms. December, 1996.

[2] D. R. Musser. 'Introspective Sorting and Selection Algorithms'. *Software — Practice and Experience*, **27**(8), 983-993 (August 1997).