

## Section A.2 A Brief Tour of the Algorithms

### Section A.3 Random Numbers

This Appendix contains additional details about the algorithms and random number parts of the library. We also provide a list of all the names we used from the standard library along with the name of the header that defines that name.

In [Chapter 10](#) we used some of the more common algorithms and described the architecture that underlies the algorithms. In this Appendix, we list all the algorithms, organized by the kinds of operations they perform.

In § [17.4](#) (p. [745](#)) we described the architecture of the random number library and used several of the library's distribution types. The library defines a number of random number engines and 20 different distributions. In this Appendix, we list all the engines and distributions.

## A.1. Library Names and Headers

Our programs mostly did not show the actual `#include` directives needed to compile the program. As a convenience to readers, [Table A.1](#) lists the library names our programs used and the header in which they may be found.

**Table A.1. Standard Library Names and Headers**

Name	Header
abort	<code>&lt;cstdlib&gt;</code>
accumulate	<code>&lt;numeric&gt;</code>
allocator	<code>&lt;memory&gt;</code>
array	<code>&lt;array&gt;</code>
auto_ptr	<code>&lt;memory&gt;</code>
back_inserter	<code>&lt;iterator&gt;</code>
bad_alloc	<code>&lt;new&gt;</code>
bad_array_new_length	<code>&lt;new&gt;</code>
bad_cast	<code>&lt;typeinfo&gt;</code>
begin	<code>&lt;iterator&gt;</code>
bernoulli_distribution	<code>&lt;random&gt;</code>
bind	<code>&lt;functional&gt;</code>
bitset	<code>&lt;bitset&gt;</code>
boolalpha	<code>&lt;iostream&gt;</code>
cerr	<code>&lt;iostream&gt;</code>
cin	<code>&lt;iostream&gt;</code>
cmatch	<code>&lt;regex&gt;</code>
copy	<code>&lt;algorithm&gt;</code>
count	<code>&lt;algorithm&gt;</code>
count_if	<code>&lt;algorithm&gt;</code>
cout	<code>&lt;iostream&gt;</code>

cref	<functional>
csub_match	<regex>
dec	<iostream>
default_float_engine	<iostream>
default_random_engine	<random>
deque	<deque>
domain_error	<stdexcept>
end	<iterator>
endl	<iostream>
ends	<iostream>
equal_range	<algorithm>
exception	<exception>
fill	<algorithm>
fill_n	<algorithm>
find	<algorithm>
find_end	<algorithm>
find_first_of	<algorithm>
find_if	<algorithm>
fixed	<iostream>
flush	<iostream>
for_each	<algorithm>
forward	<utility>
forward_list	<forward_list>
free	cstdlib

front_inserter	<iterator>
fstream	<fstream>
function	<functional>
get	<tuple>
getline	<string>
greater	<functional>
hash	<functional>
hex	<iostream>
hexfloat	<iostream>
ifstream	<fstream>
initializer_list	<initializer_list>
inserter	<iterator>
internal	<iostream>
ios_base	<ios_base>
isalpha	<cctype>
islower	<cctype>
isprint	<cctype>
ispunct	<cctype>
isspace	<cctype>
istream	<iostream>
istream_iterator	<iterator>
istringstream	<sstream>
isupper	<cctype>
left	<iostream>
less	<functional>
less_equal	<functional>
list	<list>
logic_error	<stdexcept>
lower_bound	<algorithm>
lround	<cmath>

make_move_iterator	<iterator>
make_pair	<utility>
make_shared	<memory>
make_tuple	<tuple>
malloc	cstdlib
map	<map>
max	<algorithm>
max_element	<algorithm>
mem_fn	<functional>
min	<algorithm>
move	<utility>
multimap	<map>
multiset	<set>
negate	<functional>
noboolalpha	<iostream>
normal_distribution	<random>
noshowbase	<iostream>
noshowpoint	<iostream>
noskipws	<iostream>
not1	<functional>
nothrow	<new>
nothrow_t	<new>
nounitbuf	<iostream>
nouppercase	<iostream>
nth_element	<algorithm>
oct	<iostream>
ofstream	<fstream>
ostream	<iostream>
ostream_iterator	<iterator>
ostringstream	<sstream>
out_of_range	<stdexcept>
pair	<utility>
partial_sort	<algorithm>
placeholders	<functional>

placeholders::_1	<functional>
plus	<functional>
priority_queue	<queue>
ptrdiff_t	<cstddef>
queue	<queue>
rand	<random>
random_device	<random>
range_error	<stdexcept>
ref	<functional>
regex	<regex>
regex_constants	<regex>
regex_error	<regex>
regex_match	<regex>
regex_replace	<regex>
regex_search	<regex>
remove_pointer	<type_traits>
remove_reference	<type_traits>
replace	<algorithm>
replace_copy	<algorithm>
reverse_iterator	<iterator>
right	<iostream>
runtime_error	<stdexcept>
scientific	<iostream>
set	<set>
set_difference	<algorithm>
set_intersection	<algorithm>
set_union	<algorithm>
setfill	<iomanip>
setprecision	<iomanip>
setw	<iomanip>

shared_ptr	<memory>
showbase	<iostream>
showpoint	<iostream>
size_t	<cstddef>
skipws	<iostream>
smatch	<regex>
sort	<algorithm>
sqrt	<cmath>
sregex_iterator	<regex>
ssub_match	<regex>
stable_sort	<algorithm>
stack	<stack>
stoi	<string>
strcmp	<cstring>
strcpy	<cstring>
string	<string>
stringstream	<sstream>
strlen	<cstring>
strncpy	<cstring>
strtod	<string>
swap	<utility>

terminate	<exception>
time	<ctime>
tolower	<cctype>
toupper	<cctype>
transform	<algorithm>
tuple	<tuple>
tuple_element	<tuple>
tuple_size	<tuple>
type_info	<typeinfo>
unexpected	<exception>
uniform_int_distribution	<random>
uniform_real_distribution	<random>
uninitialized_copy	<memory>
uninitialized_fill	<memory>
unique	<algorithm>
unique_copy	<algorithm>
unique_ptr	<memory>
unitbuf	<iostream>
unordered_map	<unordered_map>
unordered_multimap	<unordered_map>
unordered_multiset	<unordered_set>
unordered_set	<unordered_set>
upper_bound	<algorithm>
uppercase	<iostream>
vector	<vector>
weak_ptr	<memory>

## A.2. A Brief Tour of the Algorithms

The library defines more than 100 algorithms. Learning to use these algorithms effectively requires understanding their structure rather than memorizing the details of each algorithm. Accordingly, in [Chapter 10](#) we concentrated on describing and understanding that architecture. In this section we'll briefly describe every algorithm. In the following descriptions,

- `beg` and `end` are iterators that denote a range of elements ([§ 9.2.1, p. 331](#)). Almost all of the algorithms operate on a sequence denoted by `beg` and `end`.
- `beg2` is an iterator denoting the beginning of a second input sequence. If present, `end2` denotes the end of the second sequence. When there is no `end2`, the sequence denoted by `beg2` is assumed to be as large as the input sequence denoted by `beg` and `end`. The types of `beg` and `beg2` need not match. However, it must be possible to apply the specified operation or given callable object to elements in the two sequences.
- `dest` is an iterator denoting a destination. The destination sequence must be able to hold as many elements as necessary given the input sequence.

- `unaryPred` and `binaryPred` are unary and binary predicates (§ 10.3.1, p. 386) that return a type that can be used as a condition and take one and two arguments, respectively, that are elements in the input range.
- `comp` is a binary predicate that meets the ordering requirements for `key` in an associative container (§ 11.2.2, p. 425).
- `unaryOp` and `binaryOp` are callable objects (§ 10.3.2, p. 388) that can be called with one and two arguments from the input range, respectively.

### A.2.1. Algorithms to Find an Object

These algorithms search an input range for a specific value or sequence of values.

Each algorithm provides two overloaded versions. The first version uses equality (`==`) operator of the underlying type to compare elements; the second version compares elements using the user-supplied `unaryPred` or `binaryPred`.

#### Simple Find Algorithms

These algorithms look for specific values and require *input iterators*.

##### [Click here to view code image](#)

```
find(beg, end, val)
find_if(beg, end, unaryPred)
find_if_not(beg, end, unaryPred)
count(beg, end, val)
count_if(beg, end, unaryPred)
```

`find` returns an iterator to the first element in the input range equal to `val`.  
`find_if` returns an iterator to the first element for which `unaryPred` succeeds;  
`find_if_not` returns an iterator to the first element for which `unaryPred` is false. All three return `end` if no such element exists.

`count` returns a count of how many times `val` occurs; `count_if` counts elements for which `unaryPred` succeeds.

##### [Click here to view code image](#)

```
all_of(beg, end, unaryPred)
any_of(beg, end, unaryPred)
none_of(beg, end, unaryPred)
```

Returns a `bool` indicating whether the `unaryPred` succeeded for all of the elements, any element, or no element respectively. If the sequence is empty, `any_of` returns `false`; `all_of` and `none_of` return `true`.

#### Algorithms to Find One of Many Values

These algorithms require *forward iterators*. They look for a repeated elements in the input sequence.

[Click here to view code image](#)

```
adjacent_find(beg, end)
adjacent_find(beg, end, binaryPred )
```

Returns an iterator to the first adjacent pair of duplicate elements. Returns `end` if there are no adjacent duplicate elements.

[Click here to view code image](#)

```
search_n(beg, end, count, val)
search_n(beg, end, count, val, binaryPred )
```

Returns an iterator to the beginning of a subsequence of `count` equal elements. Returns `end` if no such subsequence exists.

### Algorithms to Find Subsequences

With the exception of `find_first_of`, these algorithms require two pairs of *forward iterators*. `find_first_of` uses *input iterators* to denote its first sequence and *forward iterators* for its second. These algorithms search for subsequences rather than for a single element.

[Click here to view code image](#)

```
search(beg1, end1, beg2, end2)
search(beg1, end1, beg2, end2, binaryPred )
```

Returns an iterator to the first position in the input range at which the second range occurs as a subsequence. Returns `end1` if the subsequence is not found.

[Click here to view code image](#)

```
find_first_of(beg1, end1, beg2, end2)
find_first_of(beg1, end1, beg2, end2, binaryPred )
```

Returns an iterator to the first occurrence in the first range of any element from the second range. Returns `end1` if no match is found.

[Click here to view code image](#)

```
find_end(beg1, end1, beg2, end2)
find_end(beg1, end1, beg2, end2, binaryPred )
```

Like `search`, but returns an iterator to the last position in the input range at which the second range occurs as a subsequence. Returns `end1` if the second subsequence is empty or is not found.

## A.2.2. Other Read-Only Algorithms

These algorithms require *input iterators* for their first two arguments.

The `equal` and `mismatch` algorithms also take an additional *input iterator* that denotes the start of a second range. They also provide two overloaded versions. The first version uses equality (`==`) operator of the underlying type to compare elements; the second version compares elements using the user-supplied `unaryPred` or `binaryPred`.

[Click here to view code image](#)

```
for_each(beg, end, unaryOp)
```

Applies the callable object (§ 10.3.2, p. 388) `unaryOp` to each element in its input range. The return value from `unaryOp` (if any) is ignored. If the iterators allow writing to elements through the dereference operator, then `unaryOp` may modify the elements.

[Click here to view code image](#)

```
mismatch(beg1, end1, beg2)
mismatch(beg1, end1, beg2, binaryPred)
```

Compares the elements in two sequences. Returns a `pair` (§ 11.2.3, p. 426) of iterators denoting the first elements in each sequence that do not match. If all the elements match, then the `pair` returned is `end1`, and an iterator into `beg2` offset by the size of the first sequence.

[Click here to view code image](#)

```
equal(beg1, end1, beg2)
equal(beg1, end1, beg2, binaryPred)
```

Determines whether two sequences are equal. Returns `true` if each element in the input range equals the corresponding element in the sequence that begins at `beg2`.

## A.2.3. Binary Search Algorithms

These algorithms require *forward iterators* but are optimized so that they execute much more quickly if they are called with *random-access iterators*. Technically speaking, regardless of the iterator type, these algorithms execute a logarithmic number of comparisons. However, when used with forward iterators, they must make a linear number of iterator operations to move among the elements in the sequence.

These algorithms require that the elements in the input sequence are already in order. These algorithms behave similarly to the associative container members of the same name (§ 11.3.5, p. 438). The `equal_range`, `lower_bound`, and

`upper_bound` algorithms return iterators that refer to positions in the sequence at which the given element can be inserted while still preserving the sequence's ordering. If the element is larger than any other in the sequence, then the iterator that is returned might be the off-the-end iterator.

Each algorithm provides two versions: The first uses the element type's less-than operator (`<`) to test elements; the second uses the given comparison operation. In the following algorithms, "`x` is less than `y`" means `x < y` or that `comp(x, y)` succeeds.

[Click here to view code image](#)

```
lower_bound(beg, end, val)
lower_bound(beg, end, val, comp )
```

Returns an iterator denoting the first element such that `val` is not less than that element, or `end` if no such element exists.

[Click here to view code image](#)

```
upper_bound(beg, end, val)
upper_bound(beg, end, val, comp )
```

Returns an iterator denoting the first element such that `val` is less than that element, or `end` if no such element exists.

[Click here to view code image](#)

```
equal_range(beg, end, val)
equal_range(beg, end, val, comp )
```

Returns a pair (§ 11.2.3, p. 426) in which the first member is the iterator that would be returned by `lower_bound`, and second is the iterator `upper_bound` would return.

[Click here to view code image](#)

```
binary_search(beg, end, val)
binary_search(beg, end, val, comp )
```

Returns a `bool` indicating whether the sequence contains an element that is equal to `val`. Two values `x` and `y` are considered equal if `x` is not less than `y` and `y` is not less than `x`.

#### A.2.4. Algorithms That Write Container Elements

Many algorithms write new values to the elements in the given sequence. These algorithms can be distinguished from one another both by the kinds of iterators they use to denote their input sequence and by whether they write elements in the input range or write to a given destination.

##### Algorithms That Write but Do Not Read Elements

These algorithms require an *output iterator* that denotes a destination. The `_n` versions take a second argument that specifies a count and write the given number of elements to the destination.

[Click here to view code image](#)

```
fill(beg, end, val)
fill_n(dest, cnt, val)
generate(beg, end, Gen)
generate_n(dest, cnt, Gen)
```

Assigns a new value to each element in the input sequence. `fill` assigns the value `val`; `generate` executes the generator object `Gen()`. A generator is a callable object (§ 10.3.2, p. 388) that is expected to produce a different return value each time it is called. `fill` and `generate` return `void`. The `_n` versions return an iterator that refers to the position immediately following the last element written to the output sequence.

### [Write Algorithms with Input Iterators](#)

Each of these algorithms reads an input sequence and writes to an output sequence. They require `dest` to be an *output iterator*, and the iterators denoting the input range must be *input iterators*.

[Click here to view code image](#)

```
copy(beg, end, dest)
copy_if(beg, end, dest, unaryPred)
copy_n(beg, n, dest)
```

Copies from the input range to the sequence denoted by `dest`. `copy` copies all elements, `copy_if` copies those for which `unaryPred` succeeds, and `copy_n` copies the first `n` elements. The input sequence must have at least `n` elements.

```
move(beg, end, dest)
```

Calls `std::move` (§ 13.6.1, p. 533) on each element in the input sequence to move that element to the sequence beginning at iterator `dest`.

[Click here to view code image](#)

```
transform(beg, end, dest, unaryOp)
transform(beg, end, beg2, dest, binaryOp)
```

Calls the given operation and writes the result of that operation to `dest`. The first version applies a unary operation to each element in the input range. The second applies a binary operation to elements from the two input sequences.

[Click here to view code image](#)

```
replace_copy(beg, end, dest, old_val, new_val)
replace_copy_if(beg, end, dest, unaryPred, new_val )
```

Copies each element to `dest`, replacing the specified elements with `new_val`. The first version replaces those elements that are `== old_val`. The second version replaces those elements for which `unaryPred` succeeds.

[Click here to view code image](#)

```
merge(beg1, end1, beg2, end2, dest)
merge(beg1, end1, beg2, end2, dest, comp )
```

Both input sequences must be sorted. Writes a merged sequence to `dest`. The first version compares elements using the `<` operator; the second version uses the given comparison operation.

### Write Algorithms with Forward Iterators

These algorithms require *forward iterators* because they write to elements in their input sequence. The iterators must give write access to the elements.

[Click here to view code image](#)

```
iter_swap(iter1, iter2)
swap_ranges(beg1, end1, beg2)
```

Swaps the element denoted by `iter1` with the one denoted by `iter2`; or swaps all of the elements in the input range with those in the second sequence beginning at `beg2`. The ranges must not overlap. `iter_swap` returns `void`; `swap_ranges` returns `beg2` incremented to denote the element just after the last one swapped.

[Click here to view code image](#)

```
replace(beg, end, old_val, new_val)
replace_if(beg, end, unaryPred, new_val )
```

Replaces each matching element with `new_val`. The first version uses `==` to compare elements with `old_val`; the second version replaces those elements for which `unaryPred` succeeds.

### Write Algorithms with Bidirectional Iterators

These algorithms require the ability to go backward in the sequence, so they require *bidirectional iterators*.

[Click here to view code image](#)

```
copy_backward(beg, end, dest)
```

**move\_backward(beg, end, dest )**

Copies or moves elements from the input range to the given destination. Unlike other algorithms, `dest` is the off-the-end iterator for the output sequence (i.e., the destination sequence will end immediately before `dest`). The last element in the input range is copied or moved to the last element in the destination, then the second-to-last element is copied/moved, and so on. Elements in the destination have the same order as those in the input range. If the range is empty, the return value is `dest`; otherwise, the return denotes the element that was copied or moved from `*beg`.

[Click here to view code image](#)

```
inplace_merge(beg, mid, end)
inplace_merge(beg, mid, end, comp )
```

Merges two sorted subsequences from the same sequence into a single, ordered sequence. The subsequences from `beg` to `mid` and from `mid` to `end` are merged and written back into the original sequence. The first version uses `<` to compare elements; the second version uses a given comparison operation. Returns `void`.

### A.2.5. Partitioning and Sorting Algorithms

The sorting and partitioning algorithms provide various strategies for ordering the elements of a sequence.

Each of the sorting and partitioning algorithms provides stable and unstable versions (§ 10.3.1, p. 387). A stable algorithm maintains the relative order of equal elements. The stable algorithms do more work and so may run more slowly and use more memory than the unstable counterparts.

#### Partitioning Algorithms

A `partition` divides elements in the input range into two groups. The first group consists of those elements that satisfy the specified predicate; the second, those that do not. For example, we can partition elements in a sequence based on whether the elements are odd, or on whether a word begins with a capital letter, and so forth. These algorithms require *bidirectional iterators*.

[Click here to view code image](#)

```
is_partitioned(beg, end, unaryPred )
```

Returns `true` if all the elements for which `unaryPred` succeeds precede those for which `unaryPred` is `false`. Also returns `true` if the sequence is empty.

[Click here to view code image](#)

```
partition_copy(beg, end, dest1, dest2, unaryPred )
```

Copies elements for which unaryPred succeeds to dest1 and copies those for which unaryPred fails to dest2. Returns a pair (§ 11.2.3, p. 426) of iterators. The first member denotes the end of the elements copied to dest1, and the second denotes the end of the elements copied to dest2. The input sequence may not overlap either of the destination sequences.

[Click here to view code image](#)

**partition\_point(beg, end, unaryPred )**

The input sequence must be partitioned by unaryPred. Returns an iterator one past the subrange for which unaryPred succeeds. If the returned iterator is not end, then unaryPred must be false for the returned iterator and for all elements that follow that point.

[Click here to view code image](#)

**stable\_partition(beg, end, unaryPred)**

**partition(beg, end, unaryPred )**

Uses unaryPred to partition the input sequence. Elements for which unaryPred succeeds are put at the beginning of the sequence; those for which the predicate is false are at the end. Returns an iterator just past the last element for which unaryPred succeeds, or beg if there are no such elements.

## Sorting Algorithms

These algorithms require *random-access iterators*. Each of the sorting algorithms provides two overloaded versions. One version uses the element's operator < to compare elements; the other takes an extra parameter that specifies an ordering relation (§ 11.2.2, p. 425). `partial_sort_copy` returns an iterator into the destination; the other sorting algorithms return `void`.

The `partial_sort` and `nth_element` algorithms do only part of the job of sorting the sequence. They are often used to solve problems that might otherwise be handled by sorting the entire sequence. Because these algorithms do less work, they typically are faster than sorting the entire input range.

[Click here to view code image](#)

**sort(beg, end)**

**stable\_sort(beg, end)**

**sort(beg, end, comp)**

**stable\_sort(beg, end, comp )**

Sorts the entire range.

[Click here to view code image](#)

**is\_sorted(beg, end)**

```
is_sorted(beg, end, comp)
is_sorted_until(beg, end)
is_sorted_until(beg, end, comp)
```

`is_sorted` returns a `bool` indicating whether the entire input sequence is sorted. `is_sorted_until` finds the longest initial sorted subsequence in the input and returns an iterator just after the last element of that subsequence.

[Click here to view code image](#)

```
partial_sort(beg, mid, end)
partial_sort(beg, mid, end, comp)
```

Sorts a number of elements equal to `mid - beg`. That is, if `mid - beg` is equal to 42, then this function puts the lowest-valued elements in sorted order in the first 42 positions in the sequence. After `partial_sort` completes, the elements in the range from `beg` up to but not including `mid` are sorted. No element in the sorted range is larger than any element in the range after `mid`. The order among the unsorted elements is unspecified.

[Click here to view code image](#)

```
partial_sort_copy(beg, end, destBeg, destEnd)
partial_sort_copy(beg, end, destBeg, destEnd, comp)
```

Sorts elements from the input range and puts as much of the sorted sequence as fits into the sequence denoted by the iterators `destBeg` and `destEnd`. If the destination range is the same size or has more elements than the input range, then the entire input range is sorted and stored starting at `destBeg`. If the destination size is smaller, then only as many sorted elements as will fit are copied.

Returns an iterator into the destination that refers just past the last element that was sorted. The returned iterator will be `destEnd` if that destination sequence is smaller than or equal in size to the input range.

[Click here to view code image](#)

```
nth_element(beg, nth, end)
nth_element(beg, nth, end, comp)
```

The argument `nth` must be an iterator positioned on an element in the input sequence. After `nth_element`, the element denoted by that iterator has the value that would be there if the entire sequence were sorted. The elements in the sequence are partitioned around `nth`: Those before `nth` are all smaller than or equal to the value denoted by `nth`, and the ones after it are greater than or equal to it.

## A.2.6. General Reordering Operations

Several algorithms reorder the elements of the input sequence. The first two, `remove` and `unique`, reorder the sequence so that the elements in the first part of the

sequence meet some criteria. They return an iterator marking the end of this subsequence. Others, such as `reverse`, `rotate`, and `random_shuffle`, rearrange the entire sequence.

The base versions of these algorithms operate “in place”; they rearrange the elements in the input sequence itself. Three of the reordering algorithms offer “copying” versions. These `_copy` versions perform the same reordering but write the reordered elements to a specified destination sequence rather than changing the input sequence. These algorithms require *output iterator* for the destination.

### **Reordering Algorithms Using Forward Iterators**

These algorithms reorder the input sequence. They require that the iterators be at least *forward iterators*.

[Click here to view code image](#)

```
remove(beg, end, val)
remove_if(beg, end, unaryPred)
remove_copy(beg, end, dest, val)
remove_copy_if(beg, end, dest, unaryPred )
```

“Removes” elements from the sequence by overwriting them with elements that are to be kept. The removed elements are those that are == `val` or for which `unaryPred` succeeds. Returns an iterator just past the last element that was not removed.

[Click here to view code image](#)

```
unique(beg, end)
unique(beg, end, binaryPred)
unique_copy(beg, end, dest)
unique_copy_if(beg, end, dest, binaryPred )
```

Reorders the sequence so that adjacent duplicate elements are “removed” by overwriting them. Returns an iterator just past the last unique element. The first version uses == to determine whether two elements are the same; the second version uses the predicate to test adjacent elements.

[Click here to view code image](#)

```
rotate(beg, mid, end)
rotate_copy(beg, mid, end, dest )
```

Rotates the elements around the element denoted by `mid`. The element at `mid` becomes the first element; elements from `mid + 1` up to but not including `end` come next, followed by the range from `beg` up to but not including `mid`. Returns an iterator denoting the element that was originally at `beg`.

### **Reordering Algorithms Using Bidirectional Iterators**

Because these algorithms process the input sequence backward, they require *bidirectional iterators*.

### [Click here to view code image](#)

```
reverse(beg, end)
reverse_copy(beg, end, dest )
```

Reverses the elements in the sequence. `reverse` returns `void`; `reverse_copy` returns an iterator just past the element copied to the destination.

### Reordering Algorithms Using Random-Access Iterators

Because these algorithms rearrange the elements in a random order, they require *random-access iterators*.

### [Click here to view code image](#)

```
random_shuffle(beg, end)
random_shuffle(beg, end, rand)
shuffle(beg, end, Uniform_rand )
```

Shuffles the elements in the input sequence. The second version takes a callable that must take a positive integer value and produce a uniformly distributed random integer in the exclusive range from 0 to the given value. The third argument to `shuffle` must meet the requirements of a uniform random number generator (§ 17.4, p. 745). All three versions return `void`.

### A.2.7. Permutation Algorithms

The permutation algorithms generate lexicographical permutations of a sequence. These algorithms reorder a permutation to produce the (lexicographically) next or previous permutation of the given sequence. They return a `bool` that indicates whether there was a next or previous permutation.

To understand what is meant by next or previous permutation, consider the following sequence of three characters: `abc`. There are six possible permutations on this sequence: `abc`, `acb`, `bac`, `bca`, `cab`, and `cba`. These permutations are listed in lexicographical order based on the less-than operator. That is, `abc` is the first permutation because its first element is less than or equal to the first element in every other permutation, and its second element is smaller than any permutation sharing the same first element. Similarly, `acb` is the next permutation because it begins with `a`, which is smaller than the first element in any remaining permutation. Permutations that begin with `b` come before those that begin with `c`.

For any given permutation, we can say which permutation comes before it and which after it, assuming a particular ordering between individual elements. Given the

permutation `bca`, we can say that its previous permutation is `bac` and that its next permutation is `cab`. There is no previous permutation of the sequence `abc`, nor is there a next permutation of `cba`.

These algorithms assume that the elements in the sequence are unique. That is, the algorithms assume that no two elements in the sequence have the same value.

To produce the permutation, the sequence must be processed both forward and backward, thus requiring *bidirectional iterators*.

[Click here to view code image](#)

```
is_permutation(beg1, end1, beg2)
is_permutation(beg1, end1, beg2, binaryPred )
```

Returns `true` if there is a permutation of the second sequence with the same number of elements as are in the first sequence and for which the elements in the permutation and in the input sequence are equal. The first version compares elements using `==`; the second uses the given `binaryPred`.

[Click here to view code image](#)

```
next_permutation(beg, end)
next_permutation(beg, end, comp )
```

If the sequence is already in its last permutation, then `next_permutation` reorders the sequence to be the lowest permutation and returns `false`. Otherwise, it transforms the input sequence into the lexicographically next ordered sequence, and returns `true`. The first version uses the element's `<` operator to compare elements; the second version uses the given comparison operation.

[Click here to view code image](#)

```
prev_permutation(beg, end)
prev_permutation(beg, end, comp )
```

Like `next_permutation`, but transforms the sequence to form the previous permutation. If this is the smallest permutation, then it reorders the sequence to be the largest permutation and returns `false`.

### A.2.8. Set Algorithms for Sorted Sequences

The set algorithms implement general set operations on a sequence that is in sorted order. These algorithms are distinct from the library `set` container and should not be confused with operations on `sets`. Instead, these algorithms provide setlike behavior on an ordinary sequential container (`vector`, `list`, etc.) or other sequence, such as an input stream.

These algorithms process elements sequentially, requiring *input iterators*. With the exception of `includes`, they also take an *output iterator* denoting a destination.

These algorithms return their `dest` iterator incremented to denote the element just after the last one that was written to `dest`.

Each algorithm is overloaded. The first version uses the `<` operator for the element type. The second uses a given comparison operation.

[Click here to view code image](#)

```
includes(beg, end, beg2, end2)
includes(beg, end, beg2, end2, comp)
```

Returns `true` if every element in the second sequence is contained in the input sequence. Returns `false` otherwise.

[Click here to view code image](#)

```
set_union(beg, end, beg2, end2, dest)
set_union(beg, end, beg2, end2, dest, comp)
```

Creates a sorted sequence of the elements that are in either sequence. Elements that are in both sequences occur in the output sequence only once. Stores the sequence in `dest`.

[Click here to view code image](#)

```
set_intersection(beg, end, beg2, end2, dest)
set_intersection(beg, end, beg2, end2, dest, comp)
```

Creates a sorted sequence of elements present in both sequences. Stores the sequence in `dest`.

[Click here to view code image](#)

```
set_difference(beg, end, beg2, end2, dest)
set_difference(beg, end, beg2, end2, dest, comp)
```

Creates a sorted sequence of elements present in the first sequence but not in the second.

[Click here to view code image](#)

```
set_symmetric_difference(beg, end, beg2, end2, dest)
set_symmetric_difference(beg, end, beg2, end2, dest, comp)
```

Creates a sorted sequence of elements present in either sequence but not in both.

### A.2.9. Minimum and Maximum Values

These algorithms use either the `<` operator for the element type or the given comparison operation. The algorithms in the first group operate on values rather than sequences. The algorithms in the second set take a sequence that is denoted by *input iterators*.

```
min(val1, val2)
min(val1, val2, comp)
min(initializer_list)
min(initializer_list, comp )

max(val1, val2)
max(val1, val2, comp)
max(initializer_list)
max(initializer_list, comp )
```

Returns the minimum/maximum of `val1` and `val2` or the minimum/maximum value in the `initializer_list`. The arguments must have exactly the same type as each other. Arguments and the return type are both references to `const`, meaning that objects are not copied.

[Click here to view code image](#)

```
minmax(val1, val2)
minmax(val1, val2, comp)
minmax(initializer_list)
minmax(initializer_list, comp )
```

Returns a `pair` (§ 11.2.3, p. 426) where the `first` member is the smaller of the supplied values and the `second` is the larger. The `initializer_list` version returns a `pair` in which the `first` member is the smallest value in the list and the `second` member is the largest.

[Click here to view code image](#)

```
min_element(beg, end)
min_element(beg, end, comp)
max_element(beg, end)
max_element(beg, end, comp)
minmax_element(beg, end)
minmax_element(beg, end, comp )
```

`min_element` and `max_element` return iterators referring to the smallest and largest element in the input sequence, respectively. `minmax_element` returns a `pair` whose `first` member is the smallest element and whose `second` member is the largest.

### Lexicographical Comparison

This algorithm compares two sequences based on the first unequal pair of elements. Uses either the `<` operator for the element type or the given comparison operation. Both sequences are denoted by *input iterators*.

[Click here to view code image](#)

```
lexicographical_compare(beg1, end1, beg2, end2)
lexicographical_compare(beg1, end1, beg2, end2, comp )
```

Returns `true` if the first sequence is lexicographically less than the second. Otherwise, returns `false`. If one sequence is shorter than the other and all its elements match the corresponding elements in the longer sequence, then the shorter sequence is lexicographically smaller. If the sequences are the same size and the corresponding elements match, then neither is lexicographically less than the other.

### A.2.10. Numeric Algorithms

The numeric algorithms are defined in the `numeric` header. These algorithms require *input iterators*; if the algorithm writes output, it uses an *output iterator* for the destination.

[Click here to view code image](#)

```
accumulate(beg, end, init)
accumulate(beg, end, init, binaryOp )
```

Returns the sum of all the values in the input range. The summation starts with the initial value specified by `init`. The return type is the same type as the type of `init`. The first version applies the `+` operator for the element type; the second version applies the specified binary operation.

[Click here to view code image](#)

```
inner_product(beg1, end1, beg2, init)
inner_product(beg1, end1, beg2, init, binOp1, binOp2 )
```

Returns the sum of the elements generated as the product of two sequences. The two sequences are processed in tandem, and the elements from each sequence are multiplied. The product of that multiplication is summed. The initial value of the sum is specified by `init`. The type of `init` determines the return type.

The first version uses the element's multiplication (`*`) and addition (`+`) operators. The second version applies the specified binary operations, using the first operation in place of addition and the second in place of multiplication.

[Click here to view code image](#)

```
partial_sum(beg, end, dest)
partial_sum(beg, end, dest, binaryOp )
```

Writes a new sequence to `dest` in which the value of each new element represents the sum of all the previous elements up to and including its position within the input range. The first version uses the `+` operator for the element type; the second version applies the specified binary operation. Returns the `dest` iterator incremented to refer just past the last element written.

[Click here to view code image](#)

```
adjacent_difference(beg, end, dest)
adjacent_difference(beg, end, dest, binaryOp )
```

Writes a new sequence to `dest` in which the value of each new element other than the first represents the difference between the current and previous elements. The first version uses the element type's `-` operation; the second version applies the specified binary operation.

```
iota(beg, end, val)
```

Assigns `val` to the first element and increments `val`. Assigns the incremented value to the next element, and again increments `val`, and assigns the incremented value to the next element in the sequence. Continues incrementing `val` and assigning its new value to successive elements in the input sequence.

## A.3. Random Numbers

The library defines a collection of random number engine classes and adaptors that use differing mathematical approaches to generating pseudorandom numbers. The library also defines a collection of distribution templates that provide numbers according to various probability distributions. Both the engines and the distributions have names that correspond to their mathematical properties.

The specifics of how these classes generate numbers is well beyond the scope of this Primer. In this section, we'll list the engine and distribution types, but the reader will need to consult other resources to learn how to use these types.

### A.3.1. Random Number Distributions

With the exception of the `bernouilli_distribution`, which always generates type `bool`, the distribution types are templates. Each of these templates takes a single type parameter that names the result type that the distribution will generate.

The distribution classes differ from other class templates we've used in that the distribution types place restrictions on the types we can specify for the template type. Some distribution templates can be used to generate only floating-point numbers; others can be used to generate only integers.

In the following descriptions, we indicate whether a distribution generates floating-point numbers by specifying the type as `template_name <RealT>`. For these templates, we can use `float`, `double`, or `long double` in place of `RealT`. Similarly, `IntT` requires one of the built-in integral types, not including `bool` or any of the `char` types. The types that can be used in place of `IntT` are `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

The distribution templates define a default template type parameter (§ 17.4.2, p. 750). The default for the integral distributions is `int`; the default for the classes that generate floating-point numbers is `double`.

The constructors for each distribution has parameters that are specific to the kind of distribution. Some of these parameters specify the range of the distribution. These ranges are always *inclusive*, unlike iterator ranges.

## **Uniform Distributions**

[Click here to view code image](#)

```
uniform_int_distribution<IntT> u(m, n);
uniform_real_distribution<RealT> u(x, y);
```

Generates values of the specified type in the given inclusive range. `m` (or `x`) is the smallest number that can be returned; `n` (or `y`) is the largest. `m` defaults to 0; `n` defaults to the maximum value that can be represented in an object of type `IntT`. `x` defaults to 0.0 and `y` defaults to 1.0.

## **Bernoulli Distributions**

[Click here to view code image](#)

```
bernoulli_distribution b(p);
```

Yields `true` with given probability `p`; `p` defaults to 0.5.

[Click here to view code image](#)

```
binomial_distribution<IntT> b(t, p);
```

Distribution computed for a sample size that is the integral value `t`, with probability `p`; `t` defaults to 1 and `p` defaults to 0.5.

[Click here to view code image](#)

```
geometric_distribution<IntT> g(p);
```

Per-trial probability of success `p`; `p` defaults to 0.5.

[Click here to view code image](#)

```
negative_binomial_distribution<IntT> nb(k, p);
```

Integral value `k` trials with probability of success `p`; `k` defaults to 1 and `p` to 0.5.

## **Poisson Distributions**

[Click here to view code image](#)

**poisson\_distribution<IntT> p(x);**

Distribution around double mean  $x$ .

[Click here to view code image](#)

**exponential\_distribution<RealT> e(lam);**

Floating-point valued lambda  $\lambda$ ;  $\lambda$  defaults to 1.0.

[Click here to view code image](#)

**gamma\_distribution<RealT> g(a, b);**

With alpha (shape)  $a$  and beta (scale)  $b$ ; both default to 1.0.

[Click here to view code image](#)

**weibull\_distribution<RealT> w(a, b);**

With shape  $a$  and scale  $b$ ; both default to 1.0.

[Click here to view code image](#)

**extreme\_value\_distribution<RealT> e(a, b);**

$a$  defaults to 0.0 and  $b$  defaults to 1.0.

## Normal Distributions

[Click here to view code image](#)

**normal\_distribution<RealT> n(m, s);**

Mean  $m$  and standard deviation  $s$ ;  $m$  defaults to 0.0,  $s$  to 1.0.

[Click here to view code image](#)

**lognormal\_distribution<RealT> ln(m, s);**

Mean  $m$  and standard deviation  $s$ ;  $m$  defaults to 0.0,  $s$  to 1.0.

[Click here to view code image](#)

**chi\_squared\_distribution<RealT> c(x);**

$x$  degrees of freedom; defaults to 1.0.

[Click here to view code image](#)

**cauchy\_distribution<RealT> c(a, b);**

Location  $a$  and scale  $b$  default to 0.0 and 1.0, respectively.

[Click here to view code image](#)

```
fisher_f_distribution<RealT> f(m, n);
```

$m$  and  $n$  degrees of freedom; both default to 1.

[Click here to view code image](#)

```
student_t_distribution<RealT> s(n);
```

$n$  degrees of freedom;  $n$  defaults to 1.

## Sampling Distributions

[Click here to view code image](#)

```
discrete_distribution<IntT> d(i, j);
discrete_distribution<IntT> d{il};
```

$i$  and  $j$  are input iterators to a sequence of weights;  $il$  is a braced list of weights. The weights must be convertible to `double`.

[Click here to view code image](#)

```
piecewise_constant_distribution<RealT> pc(b, e, w);
```

$b$ ,  $e$ , and  $w$  are input iterators.

[Click here to view code image](#)

```
piecewise_linear_distribution<RealT> pl(b, e, w);
```

$b$ ,  $e$ , and  $w$  are input iterators.

## A.3.2 Random Number Engines

The library defines three classes that implement different algorithms for generating random numbers. The library also defines three adaptors that modify the sequences produced by a given engine. The engine and engine adaptor classes are templates. Unlike the parameters to the distributions, the parameters to these engines are complex and require detailed understanding of the math used by the particular engine. We list the engines here so that the reader is aware of their existence, but describing how to generate these types is beyond the scope of this Primer.

The library also defines several types that are built from the engines or adaptors. The `default_random_engine` type is a type alias for one of the engine types parameterized by variables designed to yield good performance for casual use. The library also defines several classes that are fully specialized versions of an engine or adaptor. The engines and the specializations defined by the library are:

[Click here to view code image](#)

**default\_random\_engine**

Type alias for one of the other engines intended to be used for most purposes.

[Click here to view code image](#)

**linear\_congruential\_engine**

`minstd_rand0` Has a multiplier of 16807, a modulus of 2147483647, and an increment of 0.

`minstd_rand` Has a multiplier of 48271, a modulus of 2147483647, and an increment of 0.

[Click here to view code image](#)

**mersenne\_twister\_engine**

`mt19937` 32-bit unsigned Mersenne twister generator.

`mt19937_64` 64-bit unsigned Mersenne twister generator.

[Click here to view code image](#)

**subtract\_with\_carry\_engine**

`ranlux24_base` 32-bit unsigned subtract with carry generator.

`ranlux48_base` 64-bit unsigned subtract with carry generator.

[Click here to view code image](#)

**discard\_block\_engine**

Engine adaptor that discards results from its underlying engine. Parameterized by the underlying engine to use the block size, and size of the used blocks.

`ranlux24` Uses the `ranlux24_base` engine with a block size of 223 and a used block size of 23.

`ranlux48` Uses the `ranlux48_base` engine with a block size of 389 and a used block size of 11.

[Click here to view code image](#)

**independent\_bits\_engine**

Engine adaptor that generates numbers with a specified number of bits. Parameterized by the underlying engine to use, the number of bits to generate in its results, and an unsigned integral type to use to hold the generated bits. The number of bits specified must be less than the number of digits that the specified unsigned type can hold.

[Click here to view code image](#)

**shuffle\_order\_engine**

Engine adaptor that returns the same numbers as its underlying engine but delivers them in a different sequence. Parameterized by the underlying engine to use and the number of elements to shuffle.

`knuth_b` Uses the `minstd_rand0` engine with a table size of 256.

# Index

**Bold face** numbers refer to the page on which the term was first defined. Numbers in *italic* refer to the “Defined Terms” section in which the term is defined.

## What's new in C++11

- = default, [265](#), [506](#)
- = delete, [507](#)
- allocator, construct forwards to any constructor, [482](#)
- array container, [327](#)
- auto, [68](#)
  - for type abbreviation, [88](#), [129](#)
  - not with dynamic array, [478](#)
  - with dynamic object, [459](#)
- begin function, [118](#)
- bind function, [397](#)
- bitset enhancements, [726](#)
- constexpr
  - constructor, [299](#)
  - function, [239](#)
  - variable, [66](#)
- container
  - `cbegin` and `cend`, [109](#), [334](#)
  - `emplace` members, [345](#)
  - insert return type, [344](#)
  - nonmember `swap`, [339](#)
  - of container, [97](#), [329](#)
  - `shrink_to_fit`, [357](#)
- decltype, [70](#)
  - function return type, [250](#)

delegating constructor, 291  
deleted copy-control, 624  
division rounding, 141  
end function, 118  
enumeration  
    controlling representation, 834  
    forward declaration, 834  
    scoped, 832  
explicit conversion operator, 582  
explicit instantiation, 675  
final class, 600  
format control for floating-point, 757  
forward function, 694  
forward\_list container, 327  
function interface to callable objects, 577  
in-class initializer, 73, 274  
inherited constructor, 628, 804  
initializer\_list, 220  
inline namespace, 790  
lambda expression, 388  
list initialization  
    = (assignment), 145  
    container, 336, 423  
    dynamic array, 478  
    dynamic object, 459  
    pair, 431  
    return value, 226, 427  
    variable, 43  
    vector, 98  
long long, 33  
mem\_fn function, 843  
move function, 533  
move avoids copies, 529  
move constructor, 534  
move iterator, 543  
move-enabled this pointer, 546

- noexcept
  - exception specification, [535](#), [779](#)
  - operator, [780](#)
- nullptr, [54](#)
- random-number library, [745](#)
- range for statement, [91](#), [187](#)
  - not with dynamic array, [477](#)
- regular expression-library, [728](#)
- rvalue reference, [532](#)
  - cast from lvalue, [691](#)
  - reference collapsing, [688](#)
- sizeof data member, [157](#)
- sizeof... operator, [700](#)
- smart pointer, [450](#)
  - shared\_ptr, [450](#)
  - unique\_ptr, [470](#)
  - weak\_ptr, [473](#)
- string
  - numeric conversions, [367](#)
  - parameter with IO types, [317](#)
- template
  - function template default template argument, [670](#)
  - type alias, [666](#)
  - type parameter as friend, [666](#)
  - variadic, [699](#)
  - varidatics and forwarding, [704](#)
- trailing return type, [229](#)
  - in function template, [684](#)
  - in lambda expression, [396](#)
- tuple, [718](#)
- type alias declaration, [68](#)
- union member of class type, [848](#)
- unordered containers, [443](#)
- virtual function
  - final, [606](#)
  - override, [596](#), [606](#)

# Symbols

. . . (ellipsis parameter), [222](#)  
/\* \*/ (block comment), [9](#), [26](#)  
// (single-line comment), [9](#), [26](#)  
= default, [265](#), [306](#)  
    copy-control members, [506](#)  
    default constructor, [265](#)  
= delete, [507](#)  
    copy control, [507–508](#)  
    default constructor, [507](#)  
    function matching, [508](#)  
    move operations, [538](#)  
\_ \_DATE\_ \_, [242](#)  
\_ \_FILE\_ \_, [242](#)  
\_ \_LINE\_ \_, [242](#)  
\_ \_TIME\_ \_, [242](#)  
\_ \_cplusplus, [860](#)  
\0 (null character), [39](#)  
\xnnn (hexadecimal escape sequence), [39](#)  
\n (newline character), [39](#)  
\t (tab character), [39](#)  
\nnn (octal escape sequence), [39](#)  
{ } (curlybrace), [2](#), [26](#)  
#include, [6](#), [28](#)  
    standard header, [6](#)  
    user-defined header, [21](#)  
#define, [77](#), [80](#)  
#endif, [77](#), [80](#)  
#ifdef, [77](#), [80](#)  
#ifndef, [77](#), [80](#)  
~classname, see [destructor](#)  
; (semicolon), [3](#)  
    class definition, [73](#)  
    null statement, [172](#)  
++ (increment), [12](#), [28](#), [147–149](#), [170](#)

iterator, 107, 132  
overloaded operator, 566–568  
pointer, 118  
precedence and associativity, 148  
reverse iterator, 407  
StrBlobPtr, 566  
-- (decrement), 13, 28, 147–149, 170  
    iterator, 107  
    overloaded operator, 566–568  
    pointer, 118  
    precedence and associativity, 148  
    reverse iterator, 407, 408  
    StrBlobPtr, 566  
\* (dereference), 53, 80, 448  
    iterator, 107  
    map iterators, 429  
    overloaded operator, 569  
    pointer, 53  
    precedence and associativity, 148  
    smart pointer, 451  
    StrBlobPtr, 569  
& (address-of), 52, 80  
    overloaded operator, 554  
-> (arrow operator), 110, 132, 150  
    overloaded operator, 569  
    StrBlobPtr, 569  
. (dot), 23, 28, 150  
->\* (pointer to member arrow), 837  
. \* (pointer to member dot), 837  
[ ] (subscript), 93  
    array, 116, 132  
    array, 347  
    bitset, 727  
    deque, 347  
    does not add elements, 104  
    map, and unordered\_map, 435, 448

adds element, 435  
multidimensional array, 127  
out-of-range index, 93  
overloaded operator, 564  
pointer, 121  
string, 93, 132, 347  
StrVec, 565  
subscript range, 95  
vector, 103, 132, 347

( ) (call operator), 23, 28, 202, 252  
absInt, 571  
const member function, 573  
execution flow, 203  
overloaded operator, 571  
PrintString, 571  
ShorterString, 573  
SizeComp, 573

:: (scope operator), 8, 28, 82  
base-class member, 607  
class type member, 88, 282  
container, type members, 333  
global namespace, 789, 818  
member function, definition, 259  
overrides name lookup, 286

= (assignment), 12, 28, 144–147  
see also copy assignment  
see also move assignment  
associativity, 145  
base from derived, 603  
container, 89, 103, 337  
conversion, 145, 159  
derived class, 626  
in condition, 146  
initializer\_list, 563  
list initialization, 145  
low precedence, 146

- multiple inheritance, 805
- overloaded operator, 500, 563
- pointer, 55
  - to signed, 35
  - to unsigned, 35
- vs. == (equality), 146
- vs. initialization, 42
- +=(compound assignment), 12, 28, 147
  - bitwise operators, 155
  - iterator, 111
  - overloaded operator, 555, 560
  - Sales\_data, 564
  - exception version, 784
  - string, 89
- +(addition), 6, 140
  - iterator, 111
  - pointer, 119
  - Sales\_data, 560
  - exception version, 784
  - Sales\_item, 22
  - SmallInt, 588
  - string, 89
- (subtraction), 140
  - iterator, 111
  - pointer, 119
- \*(multiplication), 140
- /(division), 140
  - rounding, 141
- % (modulus), 141
  - grading program, 176
- ==(equality), 18, 28
  - arithmetic conversion, 144
  - container, 88, 102, 340, 341
  - iterator, 106, 107
  - overloaded operator, 561, 562
  - pointer, 55, 120

Sales\_data, 561  
string, 88  
tuple, 720  
unordered container key\_type, 443  
used in algorithms, 377, 385, 413  
vs. = (assignment), 146  
!= (inequality), 28  
arithmetic conversion, 144  
container, 88, 102, 340, 341  
iterator, 106, 107  
overloaded operator, 562  
pointer, 55, 120  
Sales\_data, 561  
string, 88  
tuple, 720  
< (less-than), 28, 143  
container, 88, 340  
ordered container key\_type, 425  
overloaded operator, 562  
strict weak ordering, 562  
string, 88  
tuple, 720  
used in algorithms, 378, 385, 413  
<= (less-than-or-equal), 12, 28, 143  
container, 88, 340  
string, 88  
> (greater-than), 28, 143  
container, 88, 340  
string, 88  
>= (greater-than-or-equal), 28, 143  
container, 88, 340  
string, 88  
>> (input operator), 8, 28  
as condition, 15, 86, 312  
chained-input, 8  
istream, 8

istream\_iterator, 403  
overloaded operator, 558–559  
precedence and associativity, 155  
Sales\_data, 558  
Sales\_item, 21  
string, 85, 132  
<< (output operator), 7, 28  
    bitset, 727  
    chained output, 7  
    ostream, 7  
    ostream\_iterator, 405  
    overloaded operator, 557–558  
    precedence and associativity, 155  
Query, 641  
Sales\_data, 557  
Sales\_item, 21  
string, 85, 132  
>> (right-shift), 153, 170  
<< (left-shift), 153, 170  
&& (logical AND), 94, 132, 142, 169  
    order of evaluation, 138  
    overloaded operator, 554  
    short-circuit evaluation, 142  
|| (logical OR), 142  
    order of evaluation, 138  
    overloaded operator, 554  
    short-circuit evaluation, 142  
& (bitwise AND), 154, 169  
    Query, 638, 644  
! (logical NOT), 87, 132, 143, 170  
|| (logical OR), 132, 170  
| (bitwise OR), 154, 170  
    Query, 638, 644  
^ (bitwise XOR), 154, 170  
~ (bitwise NOT), 154, 170  
    Query, 638, 643

- , (comma operator), [157](#), [169](#)
  - order of evaluation, [138](#)
  - overloaded operator, [554](#)
- ? : (conditional operator), [151](#), [169](#)
  - order of evaluation, [138](#)
  - precedence and associativity, [151](#)
- + (unary plus), [140](#)
- (unary minus), [140](#)
- L'c' (wchar\_t literal), [38](#)
- ddd.dddL* or *ddd.ddd1* (long double literal), [41](#)
- numE**num* or *numenum* (double literal), [39](#)
- numF* or *numf* (float literal), [41](#)
- numL* or *numl* (long literal), [41](#)
- numLL* or *numll* (long long literal), [41](#)
- numU* or *numu* (unsigned literal), [41](#)
- class member:*constant expression*, see [bit-field](#)

## A

- absInt*, [571](#)
- ( ) (call operator), [571](#)
- abstract base class, [610](#), [649](#)
  - BinaryQuery*, [643](#)
  - Disc\_quote*, [610](#)
  - Query\_base*, [636](#)
- abstract data type, [254](#), [305](#)
- access control, [611](#)–[616](#)
  - class derivation list, [596](#)
  - default inheritance access, [616](#)
  - default member access, [268](#)
  - derived class, [613](#)
  - derived-to-base conversion, [613](#)
  - design, [614](#)
  - inherited members, [612](#)
  - local class, [853](#)
  - nested class, [844](#)

private, 268  
protected, 595, 611  
public, 268  
using declaration, 615  
access specifier, 268, 305  
accessible, 611, 649  
    derived-to-base conversion, 613  
Account, 301  
accumulate, 379, 882  
    bookstore program, 406  
Action, 839  
adaptor, 372  
    back\_inserter, 402  
    container, 368, 368–371  
    front\_inserter, 402  
    inserter, 402  
    make\_move\_iterator, 543  
add, Sales\_data, 261  
add\_item, Basket, 633  
add\_to\_Folder, Message, 522  
address, 33, 78  
adjacent\_difference, 882  
adjacent\_find, 871  
advice  
    always initialize a pointer, 54  
    avoid casts, 165  
    avoid undefined behavior, 36  
    choosing a built-in type, 34  
    define small utility functions, 277  
    define variables near first use, 48  
    don't create unnecessary regex objects, 733  
    forwarding parameter pattern, 706  
    keep lambda captures simple, 394  
    managing iterators, 331, 354  
    prefix vs. postfix operators, 148  
    rule of five, 541

- use move sparingly, [544](#)
- use constructor initializer lists, [289](#)
- when to use overloading, [233](#)
- writing compound expressions, [139](#)
- aggregate class, [298](#), [305](#)
  - initialization, [298](#)
- algorithm header, [376](#)
- algorithms, [376](#), [418](#)
  - see also [Appendix A](#)
  - architecture
    - `_copy` versions, [383](#), [414](#)
    - `_if` versions, [414](#)
    - naming convention, [413–414](#)
    - operate on iterators not containers, [378](#)
    - overloading pattern, [414](#)
    - parameter pattern, [412–413](#)
    - read-only, [379–380](#)
    - reorder elements, [383–385](#), [414](#)
    - write elements, [380–383](#)
    - associative container and, [430](#)
    - bind as argument, [397](#)
    - can't change container size, [385](#)
    - element type requirements, [377](#)
    - function object arguments, [572](#)
    - `istream_iterator`, [404](#)
    - iterator category, [410–412](#)
    - iterator range, [376](#)
    - lambda as argument, [391](#), [396](#)
    - library function object, [575](#)
    - `ostream_iterator`, [404](#)
    - sort comparison, requires strict weak ordering, [425](#)
    - supplying comparison operation, [386](#), [413](#)
    - function, [386](#)
    - lambda, [389](#), [390](#)
    - two input ranges, [413](#)
    - type independence, [377](#)

use element's == (equality), 385, 413  
use element's < (less-than), 385, 413  
accumulate, 379  
bookstore program, 406  
copy, 382  
count, 378  
equal\_range, 722  
equal, 380  
fill\_n, 381  
fill, 380  
find\_if, 388, 397, 414  
find, 376  
for\_each, 391  
replace\_copy, 383  
replace, 383  
set\_intersection, 647  
sort, 384  
stable\_sort, 387  
transform, 396  
unique, 384  
alias declaration  
    namespace, 792, 817  
    template type, 666  
    type, 68  
all\_of, 871  
alloc\_n\_copy, StrVec, 527  
allocate, allocator, 481  
allocator, 481, 481–483, 491, 524–531  
    allocate, 481, 527  
    compared to operator new, 823  
    construct, 482  
    forwards to constructor, 527  
    deallocate, 483, 528  
    compared to operator delete, 823  
    destroy, 482, 528  
alternative operator name, 46

alternative\_sum, program, 682  
ambiguous  
    conversion, 583–589  
    multiple inheritance, 806  
    function call, 234, 245, 251  
    multiple inheritance, 808  
    overloaded operator, 588  
AndQuery, 637  
    class definition, 644  
    eval function, 646  
anonymous union, 848, 862  
any, bitset, 726  
any\_of, 871  
app (file mode), 319  
append, string, 362  
argc, 219  
argument, 23, 26, 202, 251  
    array, 214–219  
    buffer overflow, 215  
    to pointer conversion, 214  
    C-style string, 216  
    conversion, function matching, 234  
    default, 236  
    forwarding, 704  
    initializes parameter, 203  
    iterator, 216  
    low-level const, 213  
    main function, 218  
    multidimensional array, 218  
    nonreference parameter, 209  
    pass by reference, 210, 252  
    pass by value, 209, 252  
    uses copy constructor, 498  
    uses move constructor, 539, 541  
    passing, 208–212  
    pointer, 214

reference parameter, [210](#), [214](#)  
reference to `const`, [211](#)  
top-level `const`, [212](#)

argument list, [202](#)

argument-dependent lookup, [797](#)  
move and forward, [798](#)

`argv`, [219](#)

arithmetic  
conversion, [35](#), [159](#), [168](#)  
in equality and relational operators, [144](#)  
integral promotion, [160](#), [169](#)  
signed to unsigned, [34](#)  
to `bool`, [162](#)  
operators, [139](#)  
compound assignment (e.g., `+=`), [147](#)  
function object, [574](#)  
overloaded, [560](#)  
type, [32](#), [78](#)  
machine-dependent, [32](#)

arithmetic (addition and subtraction)  
iterators, [111](#), [131](#)  
pointers, [119](#), [132](#)

array, [113](#)–[130](#)  
[ ] (subscript), [116](#), [132](#)  
argument and parameter, [214](#)–[219](#)  
argument conversion, [214](#)  
`auto` returns pointer, [117](#)  
`begin` function, [118](#)  
compound type, [113](#)  
conversion to pointer, [117](#), [161](#)  
function arguments, [214](#)  
template argument deduction, [679](#)  
`decltype` returns array type, [118](#)  
definition, [113](#)  
dimension, constant expression, [113](#)  
dynamically allocated, [476](#), [476](#)–[484](#)

allocator, 481  
can't use begin and end, 477  
can't use range for statement, 477  
delete[], 478  
empty array, 478  
new[], 477  
shared\_ptr, 480  
unique\_ptr, 479  
elements and destructor, 502  
end function, 118  
initialization, 114  
initializer of vector, 125  
multidimensional, 125–130  
no copy or assign, 114  
of char initialization, 114  
parameter  
buffer overflow, 215  
converted to pointer, 215  
function template, 654  
pointer to, 218  
reference to, 217  
return type, 204  
trailing, 229  
type alias, 229  
decltype, 230  
sizeof, 157  
subscript range, 116  
subscript type, 116  
understanding complicated declarations, 115

array  
see also container  
see also sequential container  
[] (subscript), 347  
= (assignment), 337  
assign, 338  
copy initialization, 337  
default initialization, 336

definition, 336  
header, 329  
initialization, 334–337  
list initialization, 337  
overview, 327  
random-access iterator, 412  
swap, 339  
assert preprocessor macro, 241, 251  
assign  
    array, 338  
    invalidates iterator, 338  
    sequential container, 338  
    string, 362  
assignment, vs. initialization, 42, 288  
assignment operators, 144–147  
associative array, see `map`  
associative container, 420, 447  
    and library algorithms, 430  
    initialization, 423, 424  
    key\_type requirements, 425, 445  
    members  
        begin, 430  
        count, 437, 438  
        emplace, 432  
        end, 430  
        equal\_range, 439  
        erase, 434  
        find, 437, 438  
        insert, 432  
        key\_type, 428, 447  
        mapped\_type, 428, 448  
        value\_type, 428, 448  
    override default comparison, 425  
    override default hash, 446  
    overview, 423  
associativity, 134, 136–137, 168

= (assignment), 145  
?: (conditional operator), 151  
dot and dereference, 150  
increment and dereference, 148  
IO operator, 155  
overloaded operator, 553

at

deque, 348  
map, 435  
string, 348  
unordered\_map, 435  
vector, 348

ate (file mode), 319

auto, 68, 78

cbegin, 109, 379  
cend, 109, 379  
for type abbreviation, 88, 129  
of array, 117  
of reference, 69  
pointer to function, 249  
with new, 459

auto\_ptr deprecated, 471

automatic object, 205, 251  
see also local variable  
see also parameter  
and destructor, 502

avg\_price, Sales\_data, 259

## B

back

queue, 371  
sequential container, 346  
StrBlob, 457

back\_inserter, 382, 402, 417  
requires push\_back, 382, 402

bad, 313

bad\_alloc, 197, 460  
bad\_cast, 197, 826  
bad\_typeid, 828  
badbit, 312  
base, reverse iterator, 409  
base class, 592, 649  
    see also [virtual function](#)  
abstract, 610, 649  
base-to-derived conversion, not automatic, 602  
can be a derived class, 600  
definition, 594  
derived-to-base conversion, 597  
accessibility, 613  
key concepts, 604  
multiple inheritance, 805  
final, 600  
friendship not inherited, 614  
initialized or assigned from derived, 603  
member hidden by derived, 619  
member new and delete, 822  
multiple, see [multiple inheritance](#)  
must be complete type, 600  
protected member, 611  
scope, 617  
inheritance, 617–621  
multiple inheritance, 807  
virtual function, 620  
static members, 599  
user of, 614  
virtual, see [virtual base class](#)  
virtual destructor, 622  
Basket, 631  
    add\_item, 633  
    total, 632  
Bear, 803  
    virtual base class, 812

before\_begin, forward\_list, 351  
begin  
    associative container, 430  
    container, 106, 131, 333, 372  
    function, 118, 131  
    not with dynamic array, 477  
    multidimensional array, 129  
    StrBlob, 475  
    StrVec, 526  
bernoulli\_distribution, 752  
best match, 234, 251  
    *see also* function matching  
bidirectional iterator, 412, 417  
biggies program, 391  
binary (file mode), 319  
binary operators, 134, 168  
    overloaded operator, 552  
binary predicate, 386, 417  
binary\_function deprecated, 579  
binary\_search, 873  
BinaryQuery, 637  
    abstract base class, 643  
bind, 397, 417  
    check\_size, 398  
    generates callable object, 397  
    from pointer to member, 843  
    placeholders, 399  
    reference parameter, 400  
bind1st deprecated, 401  
bind2nd deprecated, 401  
binops desk calculator, 577  
bit-field, 854, 862  
    access to, 855  
    constant expression, 854  
bitset, 723, 723–728, 769  
    [ ] (subscript), 727

<< (output operator), [727](#)  
any, [726](#)  
count, [727](#)  
flip, [727](#)  
grading program, [728](#)  
header, [723](#)  
initialization, [723–725](#)  
from string, [724](#)  
from unsigned, [723](#)  
none, [726](#)  
reset, [727](#)  
set, [727](#)  
test, [727](#)  
to\_ulong, [727](#)  
bitwise, bitset, operators, [725](#)  
bitwise operators, [152–156](#)  
    += (compound assignment), [155](#)  
    compound assignment (e.g., +=), [147](#)  
    grading program, [154](#)  
    operand requirements, [152](#)  
blob  
    class template, [659](#)  
    constructor, [662](#)  
    initializer\_list, [662](#)  
    iterator parameters, [673](#)  
    instantiation, [660](#)  
    member functions, [661–662](#)  
block, [2](#), [12](#), [26](#), [173](#), [199](#)  
    function, [204](#)  
    scope, [48](#), [80](#), [173](#)  
    try, [193](#), [194](#), [200](#), [818](#)  
block /\* \*/, comment, [9](#), [26](#)  
book from author program, [438–440](#)  
bookstore program  
    Sales\_data, [255](#)  
    using algorithms, [406](#)  
    Sales\_item, [24](#)

bool, [32](#)  
conversion, [35](#)  
literal, [41](#)  
in condition, [143](#)

boolalpha, manipulator, [754](#)

brace, curly, [2](#), [26](#)

braced list, see [list initialization](#)

break statement, [190](#), [199](#)  
in switch, [179–181](#)

bucket management, unordered container, [444](#)

buffer, [7](#), [26](#)  
flushing, [314](#)

buffer overflow, [105](#), [116](#), [131](#)  
array parameter, [215](#)  
C-style string, [123](#)

buildMap program, [442](#)

built-in type, [2](#), [26](#), [32–34](#)  
default initialization, [43](#)

Bulk\_quote  
class definition, [596](#)  
constructor, [598](#), [610](#)  
derived from Disc\_quote, [610](#)  
design, [592](#)  
synthesized copy control, [623](#)

byte, [33](#), [78](#)

## C

.c file, [4](#)  
.cc file, [4](#)  
.cpp file, [4](#)  
.cp file, [4](#)

C library header, [91](#)

C-style cast, [164](#)

C-style string, [114](#), [122](#), [122–123](#), [131](#)  
buffer overflow, [123](#)  
initialization, [122](#)

parameter, [216](#)  
string, [124](#)  
`c_str`, [124](#)  
call by reference, [208](#), [210](#), [251](#)  
call by value, [209](#), [251](#)  
    uses copy constructor, [498](#)  
    uses move constructor, [539](#)  
call signature, [576](#), [590](#)  
callable object, [388](#), [417](#), [571–572](#)  
    `absInt`, [571](#)  
    `bind`, [397](#)  
    call signature, [576](#)  
    function and function pointers, [388](#)  
    function objects, [572](#)  
    pointer to member  
    and `bind`, [843](#)  
    and `function`, [842](#)  
    and `mem_fn`, [843](#)  
    not callable, [842](#)  
    `PrintString`, [571](#)  
    `ShorterString`, [573](#)  
    `SizeComp`, [573](#)  
    with `function`, [576–579](#)  
    with algorithms, [390](#)  
candidate function, [243](#), [251](#)  
    see also `function matching`  
function template, [695](#)  
namespace, [800](#)  
overloaded operator, [587](#)  
capacity  
    string, [356](#)  
    `StrVec`, [526](#)  
    vector, [356](#)  
capture list, see `lambda expression`  
case label, [179](#), [179–182](#), [199](#)  
    default, [181](#)

constant expression, 179  
case sensitive, string, 365  
cassert header, 241  
cast, see also [named cast](#), 168  
    checked, see [dynamic\\_cast](#)  
    old-style, 164  
    to rvalue reference, 691  
catch, 193, 195, 199, 775, 816  
    catch(...), 777, 816  
    exception declaration, 195, 200, 775, 816  
    exception object, 775  
    matching, 776  
    ordering of, 776  
    runtime\_error, 195  
catch all (catch(...)), 777, 816  
caution  
    ambiguous conversion operator, 581  
    conversions to unsigned, 37  
    dynamic memory pitfalls, 462  
    exception safety, 196  
    IO buffers, 315  
    overflow, 140  
    overloaded operator misuse, 555  
    overloaded operators and conversion operators, 586  
    smart pointer, pitfalls, 469  
    uninitialized variables, 45  
    using directives cause pollution, 795  
cbegin  
    auto, 109, 379  
    decltype, 109, 379  
    container, 109, 333, 334, 372  
cctype  
    functions, 91–93  
    header, 91  
cend  
    auto, 109, 379  
    decltype, 109, 379

container, [109](#), [333](#), [334](#), [372](#)  
`cerr`, [6](#), [26](#)  
chained input, [8](#)  
chained output, [7](#)  
`char`, [32](#)  
    signed, [34](#)  
    unsigned, [34](#)  
    array initialization, [114](#)  
    literal, [39](#)  
    representation, [34](#)  
`char16_t`, [33](#)  
`char32_t`, [33](#)  
character  
    newline (`\n`), [39](#)  
    nonprintable, [39](#), [79](#)  
    null (`\0`), [39](#)  
    tab (`\t`), [39](#)  
character string literal, see `string` literal  
check  
    `StrBlob`, [457](#)  
    `StrBlobPtr`, [474](#)  
`check_size`, [398](#)  
    bind, [398](#)  
checked cast, see `dynamic_cast`  
children's story program, [383–391](#)  
`chk_n_alloc`, `StrVec`, [526](#)  
`cin`, [6](#), [26](#)  
    tied to `cout`, [315](#)  
`cl`, [5](#)  
class, [19](#), [26](#), [72](#), [305](#)  
    see also `constructor`  
    see also `destructor`  
    see also `member function`  
    see also `static member`  
    access specifier, [268](#)  
    default, [268](#)

private, [268](#), [306](#)  
public, [268](#), [306](#)  
aggregate, [298](#), [305](#)  
assignment operator  
see [copy assignment](#)  
see [move assignment](#)  
base, see [base class](#), [649](#)  
data member, [73](#), [78](#)  
const vs. mutable, [274](#)  
const, initialization, [289](#)  
in-class initializer, [274](#)  
initialization, [263](#), [274](#)  
must be complete type, [279](#)  
mutable, [274](#), [306](#)  
order of destruction, [502](#)  
order of initialization, [289](#)  
pointer, not deleted, [503](#)  
reference, initialization, [289](#)  
sizeof, [157](#)  
declaration, [278](#), [305](#)  
default inheritance specifier, [616](#)  
definition, [72](#), [256–267](#)  
ends with semicolon, [73](#)  
derived, see [derived class](#), [649](#)  
exception, [193](#), [200](#)  
final specifier, [600](#)  
forward declaration, [279](#), [306](#)  
friend, [269](#), [280](#)  
class, [280](#)  
function, [269](#)  
member function, [280](#)  
overloaded function, [281](#)  
scope, [270](#), [281](#)  
template class or function, [664](#)  
implementation, [254](#)  
interface, [254](#)

literal, 299  
local, see [local class](#)  
member, **73**, 78  
member access, 282  
member new and delete, 822  
member : *constant expression*, see [bit-field](#)  
multiple base classes, see [multiple inheritance](#)  
name lookup, 284  
nested, see [nested class](#)  
pointer to member, see [pointer to member](#)  
preventing copies, 507  
scope, **73**, **282**, 282–287, 305  
synthesized, copy control, 267, 497, 500, 503, 537  
template member, see [member template](#)  
type member, 271  
:: (scope operator), 282  
user of, 255  
valuelike, 512  
without move constructor, 540

class  
compared to typename, 654  
default access specifier, 268  
default inheritance specifier, 616  
template parameter, 654

class derivation list, **596**  
access control, 612  
default access specifier, 616  
direct base class, 600  
indirect base class, 600  
multiple inheritance, 803  
virtual base class, 812

class template, **96**, 131, **658**, **659**, 658–667, 713  
see also [template parameter](#)  
see also [instantiation](#)  
blob, 659  
declaration, 669  
default template argument, 671

definition, [659](#)  
error detection, [657](#)  
explicit instantiation, [\*\*675\*\*](#), [675–676](#)  
explicit template argument, [660](#)  
friend, [664](#)  
all instantiations, [665](#)  
declaration dependencies, [665](#)  
same instantiation, [664](#)  
specific instantiation, [665](#)  
instantiation, [660](#)  
member function  
defined outside class body, [661](#)  
instantiation, [663](#)  
member template, see [member template](#)  
specialization, [707](#), [709–712](#), [714](#)  
`hash<key_type>`, [709](#), [788](#)  
member, [711](#)  
namespace, [788](#)  
partial, [\*\*711\*\*](#), [714](#)  
static member, [667](#)  
accessed through an instantiation, [667](#)  
definition, [667](#)  
template argument, [660](#)  
template parameter, used in definition, [660](#)  
type parameter as friend, [666](#)  
type-dependent code, [658](#)  
class type, [\*\*19\*\*](#), [26](#)  
conversion, [162](#), [305](#), [590](#)  
ambiguities, [587](#)  
conversion operator, [579](#)  
converting constructor, [294](#)  
impact on function matching, [584](#)  
overloaded function, [586](#)  
with standard conversion, [581](#)  
default initialization, [44](#)  
initialization, [73](#), [84](#), [262](#)

- union member of, [848](#)
- variable vs. function declaration, [294](#)
- clear
  - sequential container, [350](#)
  - stream, [313](#)
- clog, [6](#), [26](#)
- close, file stream, [318](#)
- cmatch, [733](#)
- cmath header, [751](#), [757](#)
- collapsing rule, reference, [688](#)
- combine, Sales\_data, [259](#)
- comma (,) operator, [157](#)
- comment, [9](#), [26](#)
  - block /\* \*/, [9](#), [26](#)
  - single-line (//), [9](#), [26](#)
- compare
  - default template argument, [670](#)
  - function template, [652](#)
  - default template argument, [670](#)
  - explicit template argument, [683](#)
  - specialization, [706](#)
  - string literal version, [654](#)
  - template argument deduction, [680](#)
  - string, [366](#)
- compareIsbn
  - and associative container, [426](#)
  - Sales\_data, [387](#)
- compilation
  - common errors, [16](#)
  - compiler options, [207](#)
  - conditional, [240](#)
  - declaration vs. definition, [44](#)
  - mixing C and C++, [860](#)
  - needed when class changes, [270](#)
  - templates, [656](#)
  - error detection, [657](#)
  - explicit instantiation, [675–676](#)

compiler  
extension, [114](#), [131](#)  
GNU, [5](#)  
Microsoft, [5](#)  
options for separate compilation, [207](#)

composition vs. inheritance, [637](#)

compound assignment (e.g., `+=`)  
    arithmetic operators, [147](#)  
    bitwise operators, [147](#)

compound expression, see [expression](#)

compound statement, [173](#), [199](#)

compound type, [50](#), [50–58](#), [78](#)  
    array, [113](#)  
    declaration style, [57](#)  
    understanding complicated declarations, [115](#)

concatenation  
    string, [89](#)  
    string literal, [39](#)

condition, [12](#), [26](#)  
    = (assignment) in, [146](#)  
    conversion, [159](#)  
    do while statement, [189](#)  
    for statement, [13](#), [185](#)  
    if statement, [18](#), [175](#)  
    in IO expression, [156](#)  
    logical operators, [141](#)  
    smart pointer as, [451](#)  
    stream type as, [15](#), [162](#), [312](#)  
    while statement, [12](#), [183](#)

condition state, IO classes, [312](#), [324](#)

conditional compilation, [240](#)

conditional operator (`? :`), [151](#)

connection, [468](#)

console window, [6](#)

const, [59](#), [78](#)  
    and `typedef`, [68](#)

conversion, [162](#)  
template argument deduction, [679](#)  
dynamically allocated  
destruction, [461](#)  
initialization, [460](#)  
initialization, [59](#)  
class type object, [262](#)  
low-level `const`, [64](#)  
argument and parameter, [213](#)  
conversion from, [163](#)  
conversion to, [162](#)  
overloaded function, [232](#)  
template argument deduction, [693](#)  
member function, [258](#), [305](#)  
`( )` (call operator), [573](#)  
not constructors, [262](#)  
overloaded function, [276](#)  
reference return, [276](#)  
parameter, [212](#)  
function matching, [246](#)  
overloaded function, [232](#)  
pointer, [63](#), [78](#)  
pointer to, [62](#), [79](#)  
conversion from `nonconst`, [162](#)  
initialization from `nonconst`, [62](#)  
overloaded parameter, [232](#)  
reference, see `reference to const`  
top-level `const`, [64](#)  
and `auto`, [69](#)  
argument and parameter, [212](#)  
`decltype`, [71](#)  
parameter, [232](#)  
template argument deduction, [679](#)  
variable, [59](#)  
declared in header files, [76](#)  
`extern`, [60](#)

local to file, 60  
const\_cast, 163, **163**  
const\_iterator, container, 108, 332  
const\_reference, container, 333  
const\_reverse\_iterator, container, 332, 407  
constant expression, 65, 78  
    array dimension, 113  
    bit-field, 854  
    case label, 179  
    enumerator, 833  
    integral, 65  
    nontype template parameter, 655  
    sizeof, 156  
    static data member, 303  
constexpr, 66, 78  
    constructor, 299  
    declared in header files, 76  
    function, 239, 251  
    nonconstant return value, 239  
    function template, 655  
    pointer, 67  
    variable, 66  
construct  
    allocator, 482  
    forwards to constructor, 527  
constructor, 262, **264**, 262–266, 305  
    see also default constructor  
    see also copy constructor  
    see also move constructor  
    calls to virtual function, 627  
    constexpr, 299  
    converting, 294, 305  
    function matching, 585  
    Sales\_data, 295  
    with standard conversion, 580  
    default argument, 290

delegating, [291](#), [306](#)  
derived class, [598](#)  
initializes direct base class, [610](#)  
initializes virtual base, [813](#)  
*explicit*, [296](#), [306](#)  
function *try* block, [778](#), [817](#)  
inherited, [628](#)  
initializer list, [265](#), [288–292](#), [305](#)  
class member initialization, [274](#)  
compared to assignment, [288](#)  
derived class, [598](#)  
function *try* block, [778](#), [817](#)  
sometimes required, [288](#)  
virtual base class, [814](#)  
*initializer\_list* parameter, [662](#)  
*not const*, [262](#)  
order of initialization, [289](#)  
derived class object, [598](#), [623](#)  
multiple inheritance, [804](#)  
virtual base classes, [814](#)  
overloaded, [262](#)  
*StrBlob*, [456](#)  
*StrBlobPtr*, [474](#)  
*TextQuery*, [488](#)  
*Blob*, [662](#)  
*initializer\_list*, [662](#)  
iterator parameters, [673](#)  
*Bulk\_quote*, [598](#), [610](#)  
*Disc\_quote*, [609](#)  
*Sales\_data*, [264–266](#)  
container, [96](#), [131](#), [326](#), [372](#)  
    see also *sequential container*  
    see also *associative container*  
adaptor, [368](#), [368–371](#)  
equality and relational operators, [370](#)  
initialization, [369](#)

requirements on container, 369  
and inheritance, 630  
as element type, 97, 329  
associative, 420, 447  
copy initialization, 334  
element type constraints, 329, 341  
elements and destructor, 502  
elements are copies, 342  
initialization from iterator range, 335  
list initialization, 336  
members  
see also [iterator](#)  
= (assignment), 337  
== (equality), 341  
!= (inequality), 341  
begin, 106, 333, 372  
cbegin, 109, 333, 334, 372  
cend, 109, 333, 334, 372  
const\_iterator, 108, 332  
const\_reference, 333  
const\_reverse\_iterator, 332, 407  
crbegin, 333  
crend, 333  
difference\_type, 131, 332  
empty, 87, 102, 131, 340  
end, 106, 131, 333, 373  
equality and relational operators, 88, 102, 340  
iterator, 108, 332  
rbegin, 333, 407  
reference, 333  
relational operators, 341  
rend, 333, 407  
reverse\_iterator, 332, 407  
size, 88, 102, 132, 340  
size\_type, 88, 102, 132, 332  
swap, 339

move operations, [529](#)  
moved-from object is valid but unspecified, [537](#)  
nonmember `swap`, [339](#)  
of container, [97](#), [329](#)  
overview, [328](#)  
sequential, [326](#), [373](#)  
type members, `::` (scope operator), [333](#)

continue statement, [191](#), [199](#)

control, flow of, [11](#), [172](#), [200](#)

conversion, [78](#), [159](#), [168](#)  
= (assignment), [145](#), [159](#)  
ambiguous, [583–589](#)  
argument, [203](#)  
arithmetic, [35](#), [159](#), [168](#)  
array to pointer, [117](#)  
argument, [214](#)  
exception object, [774](#)  
multidimensional array, [128](#)  
template argument deduction, [679](#)  
base-to-derived, not automatic, [602](#)  
`bool`, [35](#)  
class type, [162](#), [294](#), [305](#), [590](#)  
ambiguities, [587](#)  
conversion operator, [579](#)  
function matching, [584](#), [586](#)  
with standard conversion, [581](#)  
condition, [159](#)  
derived-to-base, [597](#), [649](#)  
accessibility, [613](#)  
key concepts, [604](#)  
`shared_ptr`, [630](#)  
floating-point, [35](#)  
function to pointer, [248](#)  
exception object, [774](#)  
template argument deduction, [679](#)  
integral promotion, [160](#), [169](#)

istream, 162  
multiple inheritance, 805  
ambiguous, 806  
narrowing, 43  
operand, 159  
pointer to bool, 162  
rank, 245  
return value, 223  
Sales\_data, 295  
signed type, 160  
signed to unsigned, 34  
to const, 162  
from pointer to nonconst, 62  
from reference to nonconst, 61  
template argument deduction, 679  
unscoped enumeration to integer, 834  
unsigned, 36  
virtual base class, 812  
conversion operator, 580, 580–587, 590  
    design, 581  
    explicit, 582, 590  
    bool, 583  
    function matching, 585, 586  
    SmallInt, 580  
    used implicitly, 580  
    with standard conversion, 580  
converting constructor, 294, 305  
    function matching, 585  
    with standard conversion, 580  
\_copy algorithms, 383, 414  
copy, 382, 874  
copy and swap assignment, 518  
    move assignment, 540  
    self-assignment, 519  
copy assignment, 500–501, 549  
    = default, 506

= delete, [507](#)  
base from derived, [603](#)  
copy and swap, [518](#), [549](#)  
derived class, [626](#)  
HasPtr  
reference counted, [516](#)  
valuelike, [512](#)  
memberwise, [500](#)  
Message, [523](#)  
preventing copies, [507](#)  
private, [509](#)  
reference count, [514](#)  
rule of three/five, [505](#)  
virtual destructor exception, [622](#)  
self-assignment, [512](#)  
StrVec, [528](#)  
synthesized, [500](#), [550](#)  
deleted function, [508](#), [624](#)  
derived class, [623](#)  
multiple inheritance, [805](#)  
union with class type member, [852](#)  
valuelike class, [512](#)  
copy constructor, [496](#), [496–499](#), [549](#)  
= default, [506](#)  
= delete, [507](#)  
base from derived, [603](#)  
derived class, [626](#)  
HasPtr  
reference counted, [515](#)  
valuelike, [512](#)  
memberwise, [497](#)  
Message, [522](#)  
parameter, [496](#)  
preventing copies, [507](#)  
private, [509](#)  
reference count, [514](#)  
rule of three/five, [505](#)

virtual destructor exception, 622  
StrVec, 528  
synthesized, 497, 550  
deleted function, 508, 624  
derived class, 623  
multiple inheritance, 805  
union with class type member, 851  
used for copy-initialization, 498  
copy control, 267, 496, 549  
    = delete, 507–508  
    inheritance, 623–629  
    memberwise, 267, 550  
    copy assignment, 500  
    copy constructor, 497  
    move assignment, 538  
    move constructor, 538  
    multiple inheritance, 805  
    synthesized, 267  
    as deleted function, 508  
    as deleted in derived class, 624  
    move operations as deleted function, 538  
    unions, 849  
    virtual base class, synthesized, 815  
copy initialization, 84, 131, 497, 497–499, 549  
    array, 337  
    container, 334  
    container elements, 342  
    explicit constructor, 498  
    invalid for arrays, 114  
    move vs. copy, 539  
    parameter and return value, 498  
    uses copy constructor, 497  
    uses move constructor, 541  
copy\_backward, 875  
copy\_if, 874  
copy\_n, 874

copyUnion, Token, 851  
count, reference, 550  
count  
    algorithm, 378, 871  
    associative container, 437, 438  
    bitset, 727  
count\_calls, program, 206  
count\_if, 871  
cout, 6, 26  
    tied to cin, 315  
cplusplus\_primer, namespace, 787  
crbegin, container, 333  
cref, binds reference parameter, 400, 417  
cregex\_iterator, 733, 769  
crend, container, 333  
cstddef header, 116, 120  
cstdlib header, 54, 227, 778, 823  
cstring  
    functions, 122–123  
    header, 122  
csub\_match, 733, 769  
ctime header, 749  
curly brace, 2, 26

## D

dangling else, 177, 199  
dangling pointer, 225, 463, 491  
    undefined behavior, 463  
data abstraction, 254, 306  
data hiding, 270  
data member, see class data member  
data structure, 19, 26  
deallocate, allocator, 483, 528  
debug\_rep program  
    additional nontemplate versions, 698  
    general template version, 695

nontemplate version, [697](#)  
pointer template version, [696](#)  
`DebugDelete`, member template, [673](#)  
`dec`, manipulator, [754](#)  
decimal, literal, [38](#)  
declaration, [45](#), [78](#)  
    class, [278](#), [305](#)  
    class template, [669](#)  
    class type, variable, [294](#)  
    compound type, [57](#)  
    dependencies  
    member function as friend, [281](#)  
    overloaded templates, [698](#)  
    template friends, [665](#)  
    template instantiation, [657](#)  
    template specializations, [708](#)  
    variadic templates, [702](#)  
    derived class, [600](#)  
    explicit instantiation, [675](#)  
    friend, [269](#)  
    function template, [669](#)  
    instantiation, [713](#)  
    member template, [673](#)  
    template, [669](#)  
    template specialization, [708](#)  
    type alias, [68](#)  
    using, [82](#), [132](#)  
    access control, [615](#)  
    overloaded inherited functions, [621](#)  
    variable, [45](#)  
    const, [60](#)  
declarator, [50](#), [79](#)  
`decltype`, [70](#), [79](#)  
    array return type, [230](#)  
    `cbegin`, [109](#), [379](#)  
    `cend`, [109](#), [379](#)

depends on form, 71  
for type abbreviation, 88, 106, 129  
of array, 118  
of function, 250  
pointer to function, 249  
top-level const, 71  
yields lvalue, 71, 135  
decrement operators, 147–149  
default argument, 236, 251  
    adding default arguments, 237  
    and header file, 238  
    constructor, 290  
    default constructor, 291  
    function call, 236  
    function matching, 243  
    initializer, 238  
    static member, 304  
    virtual function, 607  
default case label, 181, 199  
default constructor, 263, 306  
    = default, 265  
    = delete, 507  
    default argument, 291  
    Sales\_data, 262  
    StrVec, 526  
    synthesized, 263, 306  
deleted function, 508, 624  
derived class, 623  
Token, 850  
used implicitly  
default initialization, 293  
value initialization, 293  
default initialization, 43  
    array, 336  
    built-in type, 43  
    class type, 44

string, 44, 84  
uses default constructor, 293  
vector, 97  
default template argument, **670**  
    class template, 671  
    compare, 670  
    function template, 670  
    template<>, 671  
default\_random\_engine, 745, 769  
defaultfloat manipulator, 757  
definition, 79  
    array, 113  
    associative container, 423  
    base class, 594  
    class, 72, 256–267  
    class template, 659  
    member function, 661  
    static member, 667  
    class template partial specialization, 711  
    derived class, 596  
    dynamically allocated object, 459  
    explicit instantiation, **675**  
        function, 577  
        in if condition, 175  
        in while condition, 183  
        instantiation, 713  
        member function, 256–260  
        multidimensional array, 126  
        namespace, 785  
        can be discontiguous, 786  
        member, 788  
        overloaded operator, 500, 552  
        pair, 426  
        pointer, 52  
        pointer to function, 247  
        pointer to member, 836

reference, 51  
sequential container, 334  
`shared_ptr`, 450  
static member, 302  
`string`, 84  
template specialization, 706–712  
`unique_ptr`, 470, 472  
variable, 41, 45  
`const`, 60  
variable after case label, 182  
`vector`, 97  
`weak_ptr`, 473  
delegating constructor, 291, 306  
`delete`, 460, 460–463, 491  
    const object, 461  
    execution flow, 820  
    memory leak, 462  
    null pointer, 461  
    pointer, 460  
    runs destructor, 502  
`delete[]`, dynamically allocated array, 478  
deleted function, 507, 549  
deleter, 469, 491  
    `shared_ptr`, 469, 480, 491  
    `unique_ptr`, 472, 491  
deprecated, 401  
    `auto_ptr`, 471  
    `binary_function`, 579  
    `bind1st`, 401  
    `bind2nd`, 401  
    generalized exception specification, 780  
    `ptr_fun`, 401  
    `unary_function`, 579  
`deque`, 372  
    see also `container`, container member  
    see also `sequential container`

[ ] (subscript), [347](#)  
at, [348](#)  
header, [329](#)  
initialization, [334–337](#)  
list initialization, [336](#)  
overview, [327](#)  
push\_back, invalidates iterator, [354](#)  
push\_front, invalidates iterator, [354](#)  
random-access iterator, [412](#)  
value initialization, [336](#)  
deref, StrBlobPtr, [475](#)  
derived class, **592**, [649](#)  
    *see also* [virtual function](#)  
    :: (scope operator) to access base-class member, [607](#)  
    = (assignment), [626](#)  
    access control, [613](#)  
    as base class, [600](#)  
    assigned or copied to base object, [603](#)  
    base-to-derived conversion, not automatic, [602](#)  
    constructor, [598](#)  
    initializer list, [598](#)  
    initializes direct base class, [610](#)  
    initializes virtual base, [813](#)  
    copy assignment, [626](#)  
    copy constructor, [626](#)  
    declaration, [600](#)  
    default derivation specifier, [616](#)  
    definition, [596](#)  
    derivation list, **596**, [649](#)  
    access control, [612](#)  
    derived object  
    contains base part, [597](#)  
    multiple inheritance, [803](#)  
    derived-to-base conversion, [597](#)  
    accessibility, [613](#)  
    key concepts, [604](#)

multiple inheritance, 805  
destructor, 627  
direct base class, 600, 649  
final, 600  
friendship not inherited, 615  
indirect base class, 600, 650  
is user of base class, 614  
member new and delete, 822  
move assignment, 626  
move constructor, 626  
multiple inheritance, 803  
name lookup, 617  
order of destruction, 627  
multiple inheritance, 805  
order of initialization, 598, 623  
multiple inheritance, 804  
virtual base classes, 814  
scope, 617  
hidden base members, 619  
inheritance, 617–621  
multiple inheritance, 807  
name lookup, 618  
virtual function, 620  
static members, 599  
synthesized  
copy control members, 623  
deleted copy control members, 624  
using declaration  
access control, 615  
overloaded inherited functions, 621  
virtual function, 596  
derived-to-base conversion, 597, 649  
accessible, 613  
key concepts, 604  
multiple inheritance, 805  
not base-to-derived, 602

shared\_ptr, 630  
design  
    access control, 614  
    Bulk\_quote, 592  
    conversion operator, 581  
    Disc\_quote, 608  
    equality and relational operators, 562  
    generic programs, 655  
    inheritance, 637  
    Message class, 520  
    namespace, 786  
    overloaded operator, 554–556  
    Query classes, 636–639  
    Quote, 592  
    reference count, 514  
    StrVec, 525  
destination sequence, 381, 413  
destroy, allocator, 482, 528  
destructor, 452, 491, 501, 501–503, 549  
    = default, 506  
    called during exception handling, 773  
    calls to virtual function, 627  
    container elements, 502  
    derived class, 627  
    doesn't delete pointer members, 503  
    explicit call to, 824  
    HasPtr  
        reference counted, 515  
        valuelike, 512  
    local variables, 502  
    Message, 522  
    not deleted function, 508  
    not private, 509  
    order of destruction, 502  
    derived class, 627  
    multiple inheritance, 805

virtual base classes, 815  
reference count, 514  
rule of three/five, 505  
virtual destructor, exception, 622  
run by delete, 502  
shared\_ptr, 453  
should not throw exception, 774  
StrVec, 528  
synthesized, 503, 550  
deleted function, 508, 624  
derived class, 623  
multiple inheritance, 805  
Token, 850  
valuelike class, 512  
virtual function, 622  
virtual in base class, 622  
development environment, integrated, 3  
**difference\_type**, 112  
    vector, 112  
    container, 131, 332  
    string, 112  
direct base class, 600  
direct initialization, 84, 131  
    emplace members use, 345  
Disc\_quote  
    abstract base class, 610  
    class definition, 609  
    constructor, 609  
    design, 608  
discriminant, 849, 862  
    Token, 850  
distribution types  
    bernoulli\_distribution, 752  
    default template argument, 750  
    normal\_distribution, 751  
    random-number library, 745  
    uniform\_int\_distribution, 746

uniform\_real\_distribution, 750  
divides<T>, 575  
division rounding, 141  
do while statement, 189, 200  
domain\_error, 197  
double, 33  
    literal (*numEnum* or *numenum*), 38  
    output format, 755  
    output notation, 757  
dynamic binding, 593, 650  
    requirements for, 603  
    static vs. dynamic type, 605  
dynamic type, 601, 650  
dynamic\_cast, 163, 825, 825, 862  
    bad\_cast, 826  
    to pointer, 825  
    to reference, 826  
dynamically allocated, 450, 491  
    array, 476, 476–484  
    allocator, 481  
    can't use begin and end, 477  
    can't use range for statement, 477  
    delete[], 478  
    empty array, 478  
    new[], 477  
    returns pointer to an element, 477  
    shared\_ptr, 480  
    unique\_ptr, 479  
    delete runs destructor, 502  
    lifetime, 450  
    new runs constructor, 458  
    object, 458–463  
    const object, 460  
    delete, 460  
    factory program, 461  
    initialization, 459

make\_shared, 451  
new, 458  
shared objects, 455, 486  
shared\_ptr, 464  
unique\_ptr, 470

## E

echo command, 4  
ECMAScript, 730, 739  
    regular expression library, 730  
edit-compile-debug, 16, 26  
    errors at link time, 657  
element type constraints, container, 329, 341  
elimDups program, 383–391  
ellipsis, parameter, 222  
else, see `if` statement  
emplace  
    associative container, 432  
    priority\_queue, 371  
    queue, 371  
    sequential container, 345  
    stack, 371  
emplace\_back  
    sequential container, 345  
    StrVec, 704  
emplace\_front, sequential container, 345  
empty  
    container, 87, 102, 131, 340  
    priority\_queue, 371  
    queue, 371  
    stack, 371  
encapsulation, 254, 306  
    benefits of, 270  
end  
    associative container, 430  
    container, 106, 131, 333, 373

function, [118](#), [131](#)  
multidimensional array, [129](#)  
`StrBlob`, [475](#)  
`StrVec`, [526](#)  
end-of-file, [15](#), [26](#), [762](#)  
    character, [15](#)  
Endangered, [803](#)  
`endl`, [7](#)  
    manipulator, [314](#)  
`ends`, manipulator, [315](#)  
engine, random-number library, [745](#), [770](#)  
    `default_random_engine`, [745](#)  
    `max`, `min`, [747](#)  
    retain state, [747](#)  
    seed, [748](#), [770](#)  
enum, unscoped enumeration, [832](#)  
enum class, scoped enumeration, [832](#)  
enumeration, [832](#), [863](#)  
    as union discriminant, [850](#)  
    function matching, [835](#)  
    scoped, [832](#), [864](#)  
    unscoped, [832](#), [864](#)  
    conversion to integer, [834](#)  
    unnamed, [832](#)  
    enumerator, [832](#), [863](#)  
    constant expression, [833](#)  
    conversion to integer, [834](#)  
`eof`, [313](#)  
`eofbit`, [312](#)  
`equal`, [380](#), [872](#)  
`equal` virtual function, [829](#)  
`equal_range`  
    algorithm, [722](#), [873](#)  
    associative container, [439](#)  
`equal_to<T>`, [575](#)  
equality operators, [141](#)

- arithmetic conversion, 144
- container adaptor, 370
- container member, 340
- iterator, 106
- overloaded operator, 561
- pointer, 120
- Sales\_data, 561
- string, 88
- vector, 102

erase

- associative container, 434
- changes container size, 385
- invalidates iterator, 349
- sequential container, 349
- string, 362

error, standard, 6

error\_type, 732

error\_msg program, 221

ERRORLEVEL, 4

escape sequence, 39, 79

- hexadecimal (\xnnn), 39
- octal (\nnn), 39

eval function

- AndQuery, 646
- NotQuery, 647
- OrQuery, 645

exception

- class, 193, 200
- class hierarchy, 783
- deriving from, 782
- Sales\_data, 783
- header, 197
- initialization, 197
- what, 195, 782

exception handling, 193–198, 772, 817

- see also `throw`
- see also `catch`

exception declaration, 195, 775, 816  
and inheritance, 775  
must be complete type, 775  
exception in destructor, 773  
exception object, 774, 817  
finding a catch, 776  
function try block, 778, 817  
handler, see `catch`  
local variables destroyed, 773  
noexcept specification, 535, 779, 817  
nonthrowing function, 779, 818  
safe resource allocation, 467  
stack unwinding, 773, 818  
terminate function, 196, 200  
try block, 194, 773  
uncaught exception, 773  
unhandled exception, 196  
exception object, 774, 817  
    `catch`, 775  
    conversion to pointer, 774  
    initializes `catch` parameter, 775  
    pointer to local object, 774  
    `rethrow`, 777  
exception safety, 196, 200  
    smart pointers, 467  
exception specification argument, 780  
    generalized, deprecated, 780  
    `noexcept`, 779  
    nonthrowing, 779  
    pointer to function, 779, 781  
    `throw()`, 780  
    violation, 779  
    virtual function, 781  
executable file, 5, 251  
execution flow  
    ( ) (call operator), 203

- delete, 820
- for statement, 186
- new, 820
- switch statement, 180
- throw, 196, 773
- EXIT\_FAILURE**, 227
- EXIT\_SUCCESS**, 227
- expansion
  - forward, 705
  - parameter pack, 702, 702–704, 714
    - function parameter pack, 703
    - template parameter pack, 703
    - pattern, 702
- explicit
  - constructor, 296, 306
  - copy initialization, 498
  - conversion operator, 582, 590
  - conversion to bool, 583
- explicit call to
  - destructor, 824
  - overloaded operator, 553
  - postfix operators, 568
- explicit instantiation, 675, 713
- explicit template argument, 660, 713
  - class template, 660
  - forward, 694
  - function template, 682
  - function pointer, 686
  - template argument deduction, 682
- exporting C++ to C, 860
- expression, 7, 27, 134, 168
  - callable, see [callable object](#)
  - constant, 65, 78
  - lambda, see [lambda expression](#)
  - operand conversion, 159
  - order of evaluation, 137

- parenthesized, 136
- precedence and associativity, 136–137
- regular, see [regular expression](#)
- expression statement, 172, 200
- extension, compiler, 114, 131
- extern
  - and `const` variables, 60
  - explicit instantiation, 675
  - variable declaration, 45
- extern 'C', see [linkage directive](#)

## F

- fact program, 202
- factorial program, 227
- factory program
  - `new`, 461
  - `shared_ptr`, 453
- fail, 313
- failbit, 312
- failure, `new`, 460
- file, source, 4
- file extension, program, 730
  - version 2, 738
- file marker, stream, 765
- file mode, 319, 324
- file redirection, 22
- file static, 792, 817
- file stream, see [fstream](#)
- fill, 380, 874
- fill\_n, 381, 874
- final specifier, 600
  - class, 600
  - virtual function, 607
- find
  - algorithm, 376, 871
  - associative container, 437, 438

string, 364  
find last word program, 408  
find\_char program, 211  
find\_first\_of, 872  
find\_first\_not\_of, string, 365  
find\_first\_of, 872  
    string, 365  
find\_if, 388, 397, 414, 871  
find\_if\_not, 871  
find\_if\_not\_of, 871  
find\_last\_not\_of, string, 366  
find\_last\_of, string, 366  
findBook, program, 721  
fixed manipulator, 757  
flip  
    bitset, 727  
    program, 694  
flip1, program, 692  
flip2, program, 693  
float, 33  
    literal (*numF* or *numf*), 41  
floating-point, 32  
    conversion, 35  
    literal, 38  
    output format, 755  
    output notation, 757  
flow of control, 11, 172, 200  
flush, manipulator, 315  
Folder, see Message  
for statement, 13, 27, 185, 185–187, 200  
    condition, 13  
    execution flow, 186  
    for header, 185  
    range, 91, 187, 187–189, 200  
    can't add elements, 101, 188  
    multidimensional array, 128

for\_each, 391, 872  
format state, stream, 753  
formatted IO, 761, 769  
forward, 694  
    argument-dependent lookup, 798  
    explicit template argument, 694  
    pack expansion, 705  
    passes argument type unchanged, 694, 705  
    usage pattern, 706  
forward declaration, class, 279, 306  
forward iterator, 411, 417  
forward\_list  
    see also container  
    see also sequential container  
    before\_begin, 351  
    forward iterator, 411  
    header, 329  
    initialization, 334–337  
    list initialization, 336  
    merge, 415  
    overview, 327  
    remove, 415  
    remove\_if, 415  
    reverse, 415  
    splice\_after, 416  
    unique, 415  
    value initialization, 336  
forwarding, 692–694  
    passes argument type unchanged, 694  
    preserving type information, 692  
    rvalue reference parameters, 693, 705  
    typical implementation, 706  
    variadic template, 704  
free, StrVec, 528  
free library function, 823, 863  
free store, 450, 491  
friend, 269, 306

class, 280  
class template type parameter, 666  
declaration, 269  
declaration dependencies  
member function as friend, 281  
template friends, 665  
function, 269  
inheritance, 614  
member function, 280, 281  
overloaded function, 281  
scope, 270, 281  
namespace, 799  
template as, 664

front  
queue, 371  
sequential container, 346  
StrBlob, 457

front\_inserter, 402, 417  
compared to inserter, 402  
requires push\_front, 402

fstream, 316–320  
close, 318  
file marker, 765  
file mode, 319  
header, 310, 316  
initialization, 317  
off\_type, 766  
open, 318  
pos\_type, 766  
random access, 765  
random IO program, 766  
seek and tell, 763–768

function, 2, 27, 202, 251  
see also return type  
see also return value block, 204  
body, 2, 27, 202, 251

callable object, 388  
candidate, 251  
candidate function, 243  
`constexpr`, 239, 251  
nonconstant return value, 239  
declaration, 206  
declaration and header file, 207  
`decltype` returns function type, 250  
default argument, 236, 251  
adding default arguments, 237  
and header file, 238  
initializer, 238  
deleted, 507, 549  
function matching, 508  
exception specification  
`noexcept`, 779  
`throw()`, 780  
friend, 269  
function to pointer conversion, 248  
`inline`, 238, 252  
and header, 240  
linkage directive, 859  
member, see member function  
name, 2, 27  
nonthrowing, 779, 818  
overloaded  
compared to redeclaration, 231  
friend declaration, 281  
scope, 234  
parameter, see parameter  
parameter list, 2, 27, 202, 204  
prototype, 207, 251  
recursive, 227  
variadic template, 701  
scope, 204  
viable, 252

viable function, [243](#)  
virtual, see [virtual function](#)  
function, [577](#), [576–579](#), [590](#)  
    and pointer to member, [842](#)  
    definition, [577](#)  
    desk calculator, [577](#)  
function call  
    ambiguous, [234](#), [245](#), [251](#)  
    default argument, [236](#)  
    execution flow, [203](#)  
    overhead, [238](#)  
    through pointer to function, [248](#)  
    through pointer to member, [839](#)  
    to overloaded operator, [553](#)  
    to overloaded postfix operator, [568](#)  
function matching, [233](#), [251](#)  
    = delete, [508](#)  
    argument, conversion, [234](#)  
    candidate function, [243](#)  
    overloaded operator, [587](#)  
    const arguments, [246](#)  
    conversion, class type, [583–587](#)  
    conversion operator, [585](#), [586](#)  
    conversion rank, [245](#)  
    class type conversions, [586](#)  
    default argument, [243](#)  
    enumeration, [835](#)  
    function template, [694–699](#)  
    specialization, [708](#)  
    integral promotions, [246](#)  
    member function, [273](#)  
    multiple parameters, [244](#)  
    namespace, [800](#)  
    overloaded operator, [587–589](#)  
    prefers more specialized function, [695](#)  
    rvalue reference, [539](#)

variadic template, [702](#)  
viable function, [243](#)  
function object, [571](#), [590](#)  
    argument to algorithms, [572](#)  
    arithmetic operators, [574](#)  
    is callable object, [571](#)  
function parameter, see [parameter](#)  
function parameter pack, [700](#)  
    expansion, [703](#)  
    pattern, [704](#)  
function pointer, [247–250](#)  
    callable object, [388](#)  
    definition, [247](#)  
    exception specification, [779](#), [781](#)  
    function template instantiation, [686](#)  
    overloaded function, [248](#)  
    parameter, [249](#)  
    return type, [204](#), [249](#)  
    using `decltype`, [250](#)  
    template argument deduction, [686](#)  
    type alias declaration, [249](#)  
    typedef, [249](#)  
function table, [577](#), [577](#), [590](#), [840](#)  
function template, [652](#), [713](#)  
    see also [template parameter](#)  
    see also [template argument deduction](#)  
    see also [instantiation argument conversion](#), [680](#)  
    array function parameters, [654](#)  
    candidate function, [695](#)  
    compare, [652](#)  
    string literal version, [654](#)  
    `constexpr`, [655](#)  
    declaration, [669](#)  
    default template argument, [670](#)  
    error detection, [657](#)  
    explicit instantiation, [675](#), [675–676](#)

explicit template argument, [682](#)  
compare, [683](#)  
function matching, [694–699](#)  
inline function, [655](#)  
nontype parameter, [654](#)  
overloaded function, [694–699](#)  
parameter pack, [713](#)  
specialization, [707](#), [714](#)  
compare, [706](#)  
function matching, [708](#)  
is an instantiation, [708](#)  
namespace, [788](#)  
scope, [708](#)  
vs. overloading, [708](#)  
trailing return type, [684](#)  
type-dependent code, [658](#)  
function try block, [778](#), [817](#)  
functional header, [397](#), [399](#), [400](#), [575](#), [577](#), [843](#)

## G

g++, [5](#)  
gcount, istream, [763](#)  
generate, [874](#)  
generate\_n, [874](#)  
generic algorithms, see [algorithms](#)  
generic programming, [108](#)  
    type-independent code, [655](#)  
get  
    istream, [761](#)  
    multi-byte version, istream, [762](#)  
    returns int, istream, [762](#), [764](#)  
get<n>, [719](#), [770](#)  
getline, [87](#), [131](#)  
    istream, [762](#)  
    istringstream, [321](#)  
TextQuery constructor, [488](#)

**global function****operator delete**, 863**operator new**, 863**global namespace**, 789, 817**:: (scope operator)**, 789, 818**global scope**, 48, 80**global variable, lifetime**, 204**GNU compiler**, 5**good**, 313**goto statement**, 192, 200**grade clusters program**, 103**greater<T>**, 575**greater\_equal<T>**, 575**H****.h file header**, 19**handler**, see **catch****has-a relationship**, 637**hash<key\_type>**, 445, 447**override**, 446**specialization**, 709, 788**compatible with == (equality)**, 710**hash function**, 443, 447**HasPtr****reference counted**, 514–516**copy assignment**, 516**destructor**, 515**valuelike**, 512**copy assignment**, 512**move assignment**, 540**move constructor**, 540**swap**, 516**header**, 6, 27**iostream**, 27**C library**, 91**const and constexpr**, 76

default argument, 238  
function declaration, 207  
.h file, 19  
#include, 6, 21  
inline function, 240  
inline member function definition, 273  
namespace members, 786  
standard, 6  
table of library names, 866  
template definition, 656  
template specialization, 708  
user-defined, 21, 76–77, 207, 240  
using declaration, 83  
Sales\_data.h, 76  
Sales\_item.h, 19  
algorithm, 376  
array, 329  
bitset, 723  
cassert, 241  
ctype, 91  
cmath, 751, 757  
cstddef, 116, 120  
cstdlib, 54, 227, 778, 823  
cstring, 122  
ctime, 749  
deque, 329  
exception, 197  
forward\_list, 329  
fstream, 310, 316  
functional, 397, 399, 400, 575, 577, 843

memory, 450, 451, 481, 483  
new, 197, 460, 478, 821  
numeric, 376, 881  
queue, 371  
random, 745  
regex, 728  
set, 420  
sstream, 310, 321  
stack, 370  
stdexcept, 194, 197  
string, 74, 76, 84  
tuple, 718  
type\_info, 197  
type\_traits, 684  
typeinfo, 826, 827, 831  
unordered\_map, 420  
unordered\_set, 420  
utility, 426, 530, 533, 694  
vector, 96, 329  
header guard, 77, 79  
preprocessor, 77  
heap, 450, 491  
hex, manipulator, 754  
hexadecimal  
    escape sequence (`\xnnn`), 39  
    literal (`0xnum` or `0xnum`), 38  
hexfloat manipulator, 757  
high-order bits, 723, 770

## I

i before e, program, 729  
    version 2, 734  
IDE, 3  
identifier, 46, 79  
    reserved, 46

\_if algorithms, [414](#)  
if statement, [17](#), [27](#), [175](#), [175–178](#), [200](#)  
    compared to switch, [178](#)  
    condition, [18](#), [175](#)  
    dangling else, [177](#)  
    else branch, [18](#), [175](#), [200](#)  
ifstream, [311](#), [316–320](#), [324](#)  
    see also [istream](#)  
    close, [318](#)  
    file marker, [765](#)  
    file mode, [319](#)  
    initialization, [317](#)  
    off\_type, [766](#)  
    open, [318](#)  
    pos\_type, [766](#)  
    random access, [765](#)  
    random IO program, [766](#)  
    seek and tell, [763–768](#)  
ignore, istream, [763](#)  
implementation, [254](#), [254](#), [306](#)  
in (file mode), [319](#)  
in scope, [49](#), [79](#)  
in-class initializer, [73](#), [73](#), [79](#), [263](#), [265](#), [274](#)  
#include  
    standard header, [6](#), [21](#)  
    user-defined header, [21](#)  
includes, [880](#)  
incomplete type, [279](#), [306](#)  
    can't be base class, [600](#)  
    not in exception declaration, [775](#)  
    restrictions on use, [279](#)  
incr, StrBlobPtr, [475](#)  
increment operators, [147–149](#)  
indentation, [19](#), [177](#)  
index, [94](#), [131](#)  
    see also [\[ \] \(subscript\)](#)  
indirect base class, [600](#), [650](#)

inferred return type, lambda expression, 396

inheritance, 650

    and container, 630

    conversions, 604

    copy control, 623–629

    friend, 614

    hierarchy, 592, 600

    interface class, 637

    IO classes, 311, 324

    name collisions, 618

    private, 612, 650

    protected, 612, 650

    public, 612, 650

    vs. composition, 637

inherited, constructor, 628

initialization

    aggregate class, 298

    array, 114

    associative container, 423, 424

    bitset, 723–725

    C-style string, 122

    class type objects, 73, 262

    const

    static data member, 302

    class type object, 262

    data member, 289

    object, 59

    copy, 84, 131, 497, 497–499, 549

    default, 43, 293

    direct, 84, 131

    dynamically allocated object, 459

    exception, 197

    istream\_iterator, 405

    list, see [list initialization](#)

    lvalue reference, 532

    multidimensional array, 126

new[ ], 477  
ostream\_iterator, 405  
pair, 426  
parameter, 203, 208  
pointer, 52–54  
to const, 62  
queue, 369  
reference, 51  
data member, 289  
to const, 61  
return value, 224  
rvalue reference, 532  
sequential container, 334–337  
shared\_ptr, 464  
stack, 369  
string, 84–85, 360–361  
string streams, 321  
tuple, 718  
unique\_ptr, 470  
value, 98, 132, 293  
variable, 42, 43, 79  
vector, 97–101  
vs. assignment, 42, 288  
weak\_ptr, 473  
initializer\_list, 220, 220–222, 252  
= (assignment), 563  
constructor, 662  
header, 220  
inline function, 238, 252  
and header, 240  
function template, 655  
member function, 257, 273  
and header, 273  
inline namespace, 790, 817  
inner scope, 48, 79  
inner\_product, 882

inplace\_merge, 875  
input, standard, 6  
input iterator, 411, 418  
insert  
    associative container, 432  
    multiple key container, 433  
    sequential container, 343  
    string, 362  
insert iterator, 382, 401, 402, 418  
    back\_inserter, 402  
    front\_inserter, 402  
    inserter, 402  
inserter, 402, 418  
    compared to front\_inserter, 402  
instantiation, 96, 131, 653, 656, 713  
    Blob, 660  
    class template, 660  
    member function, 663  
    declaration, 713  
    definition, 713  
    error detection, 657  
    explicit, 675–676  
    function template from function pointer, 686  
    member template, 674  
    static member, 667  
int, 33  
    literal, 38  
integral  
    constant expression, 65  
    promotion, 134, 160, 169  
    function matching, 246  
    type, 32, 79  
integrated development environment, 3  
interface, 254, 306  
internal, manipulator, 759  
interval, left-inclusive, 373  
invalid pointer, 52

invalid\_argument, 197  
invalidated iterator  
    and container operations, 354  
    undefined behavior, 353  
invalidates iterator  
    assign, 338  
    erase, 349  
    resize, 352  
IO  
    formatted, 761, 769  
    unformatted, 761, 770  
IO classes  
    condition state, 312, 324  
    inheritance, 324  
IO stream, see [stream](#)  
[iomanip](#) header, 756  
iostate, 312  
    machine-dependent, 313  
[iostream](#), 5  
    file marker, 765  
    header, 6, 27, 310, 762  
    off\_type, 766  
    pos\_type, 766  
    random access, 765  
    random IO program, 766  
    seek and tell, 763–768  
    virtual base class, 810  
iota, 882  
is-a relationship, 637  
is\_partitioned, 876  
is\_permutation, 879  
is\_sorted, 877  
is\_sorted\_until, 877  
isalnum, 92  
isalpha, 92  
isbn

Sales\_data, 257  
Sales\_item, 23  
**ISBN, 2**  
isbn\_mismatch, 783  
iscntrl, 92  
isdigit, 92  
isgraph, 92  
islower, 92  
isprint, 92  
ispunct, 92  
isShorter program, 211  
isspace, 92  
**istream, 5, 27, 311**  
    see also [manipulator](#)  
    >> (input operator), 8 precedence and associativity, 155  
    as condition, 15  
    chained input, 8  
    condition state, 312  
    conversion, 162  
        explicit conversion to bool, 583  
    file marker, 765  
    flushing input buffer, 314  
    format state, 753  
    gcount, 763  
    get, 761  
        multi-byte version, 762  
        returns int, 762, 764  
    getline, 87, 321, 762  
    ignore, 763  
    no copy or assign, 311  
    off\_type, 766  
    peek, 761  
    pos\_type, 766  
    put, 761  
    putback, 761  
    random access, 765

random IO program, 766  
read, 763  
seek and tell, 763–768  
unformatted IO, 761  
multi-byte, 763  
single-byte, 761  
unget, 761  
`istream_iterator`, 403, 418  
    `>>` (input operator), 403  
algorithms, 404  
initialization, 405  
off-the-end iterator, 403  
operations, 404  
type requirements, 406  
`istringstream`, 311, 321, 321–323  
    see also `istream`  
word per line processing, 442  
file marker, 765  
`getline`, 321  
initialization, 321  
`off_type`, 766  
phone number program, 321  
`pos_type`, 766  
random access, 765  
random IO program, 766  
seek and tell, 763–768  
TextQuery constructor, 488  
`isupper`, 92  
`isxdigit`, 92  
`iter_swap`, 875  
`iterator`, 106, 106–112, 131  
    `++` (increment), 107, 132  
    `--` (decrement), 107  
    `*` (dereference), 107  
    `+=` (compound assignment), 111  
    `+` (addition), 111

- (subtraction), [111](#)
- $\text{==}$  (equality), [106](#), [107](#)
- $\text{!=}$  (inequality), [106](#), [107](#)
- algorithm type independence, [377](#)
- arithmetic, [111](#), [131](#)
- compared to reverse iterator, [409](#)
- destination, [413](#)
- insert, [401](#), [418](#)
- move, [401](#), [418](#), [543](#)
- uninitialized\_copy, [543](#)
- off-the-beginning
  - before\_begin, [351](#)
  - forward\_list, [351](#)
  - off-the-end, [106](#), [132](#), [373](#)
  - istream\_iterator, [403](#)
  - parameter, [216](#)
  - regex, [734](#)
  - relational operators, [111](#)
  - reverse, [401](#), [407–409](#), [418](#)
  - stream, [401](#), [403–406](#), [418](#)
  - used as destination, [382](#)
- iterator
  - compared to reverse\_iterator, [408](#)
  - container, [108](#), [332](#)
  - header, [119](#), [382](#), [401](#)
  - set iterators are const, [429](#)
- iterator category, [410](#), [410–412](#), [418](#)
  - bidirectional iterator, [412](#), [417](#)
  - forward iterator, [411](#), [417](#)
  - input iterator, [411](#), [418](#)
  - output iterator, [411](#), [418](#)
  - random-access iterator, [412](#), [418](#)
- iterator range, [331](#), [331–332](#), [373](#)
  - algorithms, [376](#)
  - as initializer of container, [335](#)
  - container erase member, [349](#)
  - container insert member, [344](#)

left-inclusive, [331](#)

off-the-end, [331](#)

## K

key concept

algorithms

and containers, [378](#)

iterator arguments, [381](#)

class user, [255](#)

classes define behavior, [20](#)

defining an assignment operator, [512](#)

dynamic binding in C++, [605](#)

elements are copies, [342](#)

encapsulation, [270](#)

headers for template code, [657](#)

indentation, [19](#)

inheritance and conversions, [604](#)

is A and has A relationships, [637](#)

name lookup and inheritance, [619](#)

protected members, [614](#)

refactoring, [611](#)

respecting base class interface, [599](#)

specialization declarations, [708](#)

type checking, [46](#)

types define behavior, [3](#)

use concise expressions, [149](#)

`key_type`

associative container, [428, 447](#)

requirements

ordered container, [425](#)

unordered container, [445](#)

keyword table, [47](#)

Koenig lookup, [797](#)

## L

L'c' (`wchar_t` literal), [38](#)  
label  
    case, [179](#), [199](#)  
    statement, [192](#)  
labeled statement, [192](#), [200](#)  
lambda expression, [388](#), [418](#)  
    arguments, [389](#)  
    biggies program, [391](#)  
    reference capture, [393](#)  
    capture list, [388](#), [417](#)  
    capture by reference, [393](#)  
    capture by value, [390](#), [392](#)  
    implicit capture, [394](#)  
    inferred return type, [389](#), [396](#)  
    mutable, [395](#)  
    parameters, [389](#)  
    passed to `find_if`, [390](#)  
    passed to `stable_sort`, [389](#)  
    synthesized class type, [572](#)–[574](#)  
    trailing return type, [396](#)  
left, manipulator, [758](#)  
left-inclusive interval, [331](#), [373](#)  
length\_error, [197](#)  
less<T>, [575](#)  
less\_equal<T>, [575](#)  
letter grade, program, [175](#)  
lexicographical\_compare, [881](#)  
library function objects, [574](#)  
    as arguments to algorithms, [575](#)  
library names to header table, [866](#)  
library type, [5](#), [27](#), [82](#)  
lifetime, [204](#), [252](#)  
    compared to scope, [204](#)  
    dynamically allocated objects, [450](#), [461](#)  
    global variable, [204](#)  
    local variable, [204](#)

parameter, 205  
linkage directive, 858, 863  
    C++ to C, 860  
    compound, 858  
    overloaded function, 860  
    parameter or return type, 859  
    pointer to function, 859  
    return type, 859  
    single, 858  
linker, 208, 252  
    template errors at link time, 657  
list, 373  
    *see also* container  
    *see also* sequential container  
    bidirectional iterator, 412  
    header, 329  
    initialization, 334–337  
    list initialization, 336  
    merge, 415  
    overview, 327  
    remove, 415  
    remove\_if, 415  
    reverse, 415  
    splice, 416  
    unique, 415  
    value initialization, 336  
list initialization, 43, 79  
    = (assignment), 145  
    array, 337  
    associative container, 423  
    container, 336  
    dynamically allocated, object, 459  
    pair, 427, 431, 527  
    preferred, 99  
    prevents narrowing, 43  
    return value, 226, 427, 527

sequential container, 336  
vector, 98

literal, 38, 38–41, 79  
bool, 41  
in condition, 143  
char, 39  
decimal, 38  
double (*numE**num* or *nume**num*), 38  
float (*numF* or *numf*), 41  
floating-point, 38  
hexadecimal (0x*num* or 0x*num*), 38  
int, 38  
long (*numL* or *numl*), 38  
long double (*ddd.dddL* or *ddd.ddd1*), 41  
long long (*numLL* or *numl1*), 38  
octal (0*num*), 38  
string, 7, 28, 39  
unsigned (*numU* or *numu*), 41  
wchar\_t, 40

literal type, 66  
class type, 299

local class, 852, 863  
access control, 853  
name lookup, 853  
nested class in, 854  
restrictions, 852

local scope, see block scope

local static object, 205, 252

local variable, 204, 252  
destroyed during exception handling, 467, 773  
destructor, 502  
lifetime, 204  
pointer, return value, 225  
reference, return value, 225  
return statement, 224

lock, weak\_ptr, 473

logic\_error, 197  
logical operators, 141, 142  
    condition, 141  
    function object, 574  
logical\_and<T>, 575  
logical\_not<T>, 575  
logical\_or<T>, 575  
long, 33  
    literal (*numL* or *numl*), 38  
long double, 33  
    literal (*ddd.dddL* or *ddd.dddl*), 41  
long long, 33  
    literal (*numLL* or *numll*), 38  
lookup, name, see name lookup  
low-level const, 64, 79  
    argument and parameter, 213  
    conversion from, 163  
    conversion to, 162  
    overloaded function, 232  
    template argument deduction, 693  
low-order bits, 723, 770  
lower\_bound  
    algorithm, 873  
    ordered container, 438  
lround, 751  
lvalue, 135, 169  
    cast to rvalue reference, 691  
    copy initialization, uses copy constructor, 539  
    decltype, 135  
    reference collapsing rule, 688  
    result  
        -> (arrow operator), 150  
        ++ (increment) prefix, 148  
        -- (decrement) prefix, 148  
        \* (dereference), 135  
        [ ] (subscript), 135

- = (assignment), [145](#)
- , (comma operator), [158](#)
- ? : (conditional operator), [151](#)
- cast, [163](#)
- decltype, [71](#)
- function reference return type, [226](#)
- variable, [533](#)

lvalue reference, see also [reference](#), [532](#), [549](#)

- collapsing rule, [688](#)
- compared to rvalue reference, [533](#)
- function matching, [539](#)
- initialization, [532](#)
- member function, [546](#)
- overloaded, [547](#)
- move, [533](#)
- template argument deduction, [687](#)

## M

- machine-dependent
  - bit-field layout, [854](#)
  - char representation, [34](#)
  - end-of-file character, [15](#)
  - enum representation, [835](#)
  - iostate, [313](#)
  - linkage directive language, [861](#)
  - nonzero return from `main`, [227](#)
  - random IO, [763](#)
  - `reinterpret_cast`, [164](#)
  - return from exception what, [198](#)
  - signed out-of-range value, [35](#)
  - signed types and bitwise operators, [153](#)
  - size of arithmetic types, [32](#)
  - terminate function, [196](#)
  - `type_info` members, [831](#)
  - `vector`, memory management, [355](#)
  - volatile implementation, [856](#)

main, [2](#), [27](#)  
    not recursive, [228](#)  
    parameters, [218](#)  
    return type, [2](#)  
    return value, [2–4](#), [227](#)  
make\_move\_iterator, [543](#)  
make\_pair, [428](#)  
make\_plural program, [224](#)  
make\_shared, [451](#)  
make\_tuple, [718](#)  
malloc library function, [823](#), [863](#)  
manipulator, [7](#), [27](#), [753](#), [770](#)  
    boolalpha, [754](#)  
    change format state, [753](#)  
    dec, [754](#)  
    defaultfloat, [757](#)  
    endl, [314](#)  
    ends, [315](#)  
    fixed, [757](#)  
    flush, [315](#)  
    hex, [754](#)  
    hexfloat, [757](#)  
    internal, [759](#)  
    left, [758](#)  
    noboolalpha, [754](#)  
    noshowbase, [755](#)  
    noshowpoint, [758](#)  
    noskipws, [760](#)  
    nouppercase, [755](#)  
    oct, [754](#)  
    right, [758](#)  
    scientific, [757](#)  
    setfill, [759](#)  
    setprecision, [756](#)  
    setw, [758](#)  
    showbase, [755](#)

showpoint, 758  
skipws, 760  
unitbuf, 315  
uppercase, 755  
**map**, 420, 447  
    see also *ordered container*  
    \* (dereference), 429  
    [ ] (subscript), 435, 448  
    adds element, 435  
    at, 435  
    definition, 423  
    header, 420  
    insert, 431  
    key\_type requirements, 425  
    list initialization, 423  
    lower\_bound, 438  
    map, initialization, 424  
    TextQuery class, 485  
    upper\_bound, 438  
    word\_count program, 421  
**mapped\_type**, associative container, 428, 448  
**match**  
    best, 251  
    no, 252  
**match\_flag\_type**, regex\_constants, 743  
**max**, 881  
**max\_element**, 881  
**mem\_fn**, 843, 863  
    generates callable, 843  
**member**, see class data member  
**member access operators**, 150  
**member function**, 23, 27, 306  
    as friend, 281  
    base member hidden by derived, 619  
    class template  
    defined outside class body, 661

instantiation, [663](#)  
`const`, [258](#), [305](#)  
( ) (call operator), [573](#)  
reference return, [276](#)  
declared but not defined, [509](#)  
defined outside class, [259](#)  
definition, [256–260](#)  
:: (scope operator), [259](#)  
name lookup, [285](#)  
parameter list, [282](#)  
return type, [283](#)  
explicitly `inline`, [273](#)  
function matching, [273](#)  
implicit `this` parameter, [257](#)  
implicitly `inline`, [257](#)  
`inline` and header, [273](#)  
move-enabled, [545](#)  
name lookup, [287](#)  
overloaded, [273](#)  
`on const`, [276](#)  
`on lvalue or rvalue reference`, [547](#)  
overloaded operator, [500](#), [552](#)  
reference qualified, [546](#), [550](#)  
returning `*this`, [260](#), [275](#)  
rvalue reference parameters, [544](#)  
scope, [282](#)  
template, see `member template`  
`member template`, [672](#), [714](#)  
    `Blob`, iterator constructor, [673](#)  
    `DebugDelete`, [673](#)  
    declaration, [673](#)  
    defined outside class body, [674](#)  
    instantiation, [674](#)  
    template parameters, [673](#), [674](#)  
`memberwise`  
    copy assignment, [500](#)

copy constructor, 497  
copy control, 267, 550  
destruction is implicit, 503  
move assignment, 538  
move constructor, 538

memory  
  see also *dynamically allocated*  
  exhaustion, 460  
  leak, 462

memory header, 450, 451, 481, 483

merge, 874  
  list and `forward_list`, 415

Message, 519–524  
  `add_to_Folder`, 522  
  class definition, 521  
  copy assignment, 523  
  copy constructor, 522  
  design, 520  
  destructor, 522  
  move assignment, 542  
  move constructor, 542  
  `move_Folders`, 542  
  `remove_from_Folders`, 523

method, see *member function*

Microsoft compiler, 5

`min`, 881  
`min_element`, 881  
`minmax`, 881  
`minus<T>`, 575  
`mismatch`, 872  
`mode`, *file*, 324  
`modulus<T>`, 575  
`move`, 530, 533, 874  
  argument-dependent lookup, 798  
  binds rvalue reference to lvalue, 533  
  explained, 690–692

inherently dangerous, 544  
Message, move operations, 541  
moved from object has unspecified value, 533  
reference collapsing rule, 691  
StrVec reallocate, 530  
remove\_reference, 691  
move assignment, **536**, 550  
    copy and swap, 540  
    derived class, 626  
    HasPtr, valuelike, 540  
    memberwise, 538  
    Message, 542  
    moved-from object destructible, 537  
    noexcept, 535  
    rule of three/five, virtual destructor exception, 622  
    self-assignment, 537  
    StrVec, 536  
    synthesized  
    deleted function, 538, 624  
    derived class, 623  
    multiple inheritance, 805  
    sometimes omitted, 538  
move constructor, **529**, **534**, 534–536, 550  
    and copy initialization, 541  
    derived class, 626  
    HasPtr, valuelike, 540  
    memberwise, 538  
    Message, 542  
    moved-from object destructible, 534, 537  
    noexcept, 535  
    rule of three/five, virtual destructor exception, 622  
    string, 529  
    StrVec, 535  
    synthesized  
    deleted function, 624  
    derived class, 623

- multiple inheritance, [805](#)
- sometimes omitted, [538](#)
- move iterator, [401](#), [418](#), **[543](#)**, [550](#)
  - `make_move_iterator`, [543](#)
  - `StrVec, reallocate`, [543](#)
  - `uninitialized_copy`, [543](#)
- move operations, [531](#)–[548](#)
  - function matching, [539](#)
  - `move`, [533](#)
  - `noexcept`, [535](#)
  - rvalue references, [532](#)
  - valid but unspecified, [537](#)
- `move_backward`, [875](#)
- `move_Folders`, `Message`, [542](#)
- multidimensional array, [125](#)–[130](#)
  - `[ ] (subscript)`, [127](#)
  - argument and parameter, [218](#)
  - `begin`, [129](#)
  - conversion to pointer, [128](#)
  - definition, [126](#)
  - `end`, [129](#)
  - initialization, [126](#)
  - pointer, [128](#)
  - range `for` statement and, [128](#)
- `multimap`, [448](#)
  - see also `ordered container`
  - `*` (dereference), [429](#)
  - definition, [423](#)
  - has no subscript operator, [435](#)
  - `insert`, [431](#), [433](#)
  - `key_type` requirements, [425](#)
  - list initialization, [423](#)
  - `lower_bound`, [438](#)
  - `map`, initialization, [424](#)
  - `upper_bound`, [438](#)
- multiple inheritance, **[802](#)**, [817](#)

see *also* [virtual base class](#)  
= (assignment), [805](#)  
ambiguous conversion, [806](#)  
ambiguous names, [808](#)  
avoiding ambiguities, [809](#)  
class derivation list, [803](#)  
conversion, [805](#)  
copy control, [805](#)  
name lookup, [807](#)  
object composition, [803](#)  
order of initialization, [804](#)  
scope, [807](#)  
virtual function, [807](#)

`multiplies<T>`, [575](#)

`multiset`, [448](#)  
see *also* [ordered container](#)  
`insert`, [433](#)  
`iterator`, [429](#)  
`key_type` requirements, [425](#)  
list initialization, [423](#)  
`lower_bound`, [438](#)  
override comparison  
Basket class, [631](#)  
using `compareIsbn`, [426](#)  
`upper_bound`, [438](#)  
used in Basket, [632](#)

`mutable`  
data member, [\*\*274\*\*](#)  
lambda expression, [395](#)

## N

`\n` (newline character), [39](#)  
name lookup, [\*\*283\*\*](#), [306](#)  
    `::` (scope operator), overrides, [286](#)  
    argument-dependent lookup, [797](#)  
    before type checking, [619](#)

multiple inheritance, 809  
block scope, 48  
class, 284  
class member  
declaration, 284  
definition, 285, 287  
function, 284  
depends on static type, 617, 619  
multiple inheritance, 806  
derived class, 617  
name collisions, 618  
local class, 853  
multiple inheritance, 807  
ambiguous names, 808  
namespace, 796  
nested class, 846  
overloaded virtual functions, 621  
templates, 657  
type checking, 235  
virtual base class, 812  
named cast, 162  
    const\_cast, 163, 163  
    dynamic\_cast, 163, 825  
    reinterpret\_cast, 163, 164  
    static\_cast, 163, 163  
namespace, 7, 27, 785, 817  
    alias, 792, 817  
    argument-dependent lookup, 797  
    candidate function, 800  
    cplusplus\_primer, 787  
    definition, 785  
    design, 786  
    discontiguous definition, 786  
    friend declaration scope, 799  
    function matching, 800  
    global, 789, 817

- inline, [790](#), [817](#)
- member, [786](#)
- member definition, [788](#)
- outside namespace, [788](#)
- name lookup, [796](#)
- nested, [789](#)
- overloaded function, [800](#)
- placeholders, [399](#)
- scope, [785–790](#)
- std, [7](#)
- template specialization, [709](#), [788](#)
- unnamed, [791](#), [818](#)
- local to file, [791](#)
- replace file static, [792](#)
- namespace pollution, [785](#), [817](#)
- narrowing conversion, [43](#)
- NDEBUG, [241](#)
- negate<T>, [575](#)
- nested class, [843](#), [863](#)
  - access control, [844](#)
  - class defined outside enclosing class, [845](#)
  - constructor, [QueryResult](#), [845](#)
  - in local class, [854](#)
  - member defined outside class body, [845](#)
  - name lookup, [846](#)
  - [QueryResult](#), [844](#)
  - relationship to enclosing class, [844](#), [846](#)
  - scope, [844](#)
  - static member, [845](#)
- nested namespace, [789](#)
- nested type, see nested class
- new, [458](#), [458–460](#), [491](#)
  - execution flow, [820](#)
  - failure, [460](#)
  - header, [197](#), [460](#), [478](#), [821](#)
  - initialization, [458](#)

placement, [460](#), [491](#), **824**, [863](#)  
union with class type member, [851](#)  
`shared_ptr`, [464](#)  
`unique_ptr`, [470](#)  
with `auto`, [459](#)

`new[ ]`, **477**, [477–478](#)  
    initialization, [477](#)  
    returns pointer to an element, [477](#)  
    value initialization, [478](#)

`newline (\n)`, character, [39](#)

`next_permutation`, [879](#)

no match, **234**, [252](#)  
    see also `function matching`

`noboolalpha`, manipulator, [754](#)

`NoDefault`, [293](#)

`noexcept`  
    exception specification, **779**, [817](#)  
    argument, [779–781](#)  
    violation, [779](#)  
    move operations, [535](#)  
    operator, **780**, [817](#)

`nonconst` reference, see `reference`

`none`, `bitset`, [726](#)

`none_of`, [871](#)

`nonportable`, [36](#), [863](#)

`nonprintable` character, [39](#), [79](#)

nonthrowing function, **779**, [818](#)

`nontype` parameter, **654**, [714](#)  
    `compare`, [654](#)  
    must be constant expression, [655](#)  
    type requirements, [655](#)

`normal_distribution`, [751](#)

`noshowbase`, manipulator, [755](#)

`noshowpoint`, manipulator, [758](#)

`noskipws`, manipulator, [760](#)

`not_equal_to<T>`, [575](#)

NotQuery, 637  
    class definition, 642  
    eval function, 647  
nouppercase, manipulator, 755  
nth\_element, 877  
NULL, 54  
null (\0), character, 39  
null pointer, 53, 79  
    delete of, 461  
null statement, 172, 200  
null-terminated character string, see C-style string  
nullptr, 54, 79  
numeric header, 376, 881  
numeric conversion, to and from string, 367  
numeric literal  
    float (numF or numf), 41  
    long (numL or numl), 41  
    long double (ddd.dddL or ddd.ddd1), 41  
    long long(numLL or numll), 41  
    unsigned (numU or numu), 41

## O

object, 42, 79  
    automatic, 205, 251  
    dynamically allocated, 458–463  
    const object, 460  
    delete, 460  
    factory program, 461  
    initialization, 459  
    lifetime, 450  
    new, 458  
    lifetime, 204, 252  
    local static, 205, 252  
    order of destruction  
    class type object, 502  
    derived class object, 627

- multiple inheritance, [805](#)
- virtual base classes, [815](#)
- order of initialization
  - class type object, [289](#)
  - derived class object, [598, 623](#)
  - multiple inheritance, [804](#)
  - virtual base classes, [814](#)
- object code, [252](#)
- object file, [208, 252](#)
- object-oriented programming, [650](#)
- oct, manipulator, [754](#)
- octal, literal (`0num`), [38](#)
- octal escape sequence (`\nnn`), [39](#)
- off-the-beginning iterator, [351, 373](#)
  - before\_begin, [351](#)
  - forward\_list, [351](#)
- off-the-end
  - iterator, [106, 132, 373](#)
  - iterator range, [331](#)
  - pointer, [118](#)
- ofstream, [311, 316–320, 324](#)
  - see also `ostream`
  - close, [318](#)
  - file marker, [765](#)
  - file mode, [319](#)
  - initialization, [317](#)
  - off\_type, [766](#)
  - open, [318](#)
  - pos\_type, [766](#)
  - random access, [765](#)
  - random IO program, [766](#)
  - seek and tell, [763–768](#)
- old-style, cast, [164](#)
- open, file stream, [318](#)
- operand, [134, 169](#)
  - conversion, [159](#)

operator, [134](#), [169](#)  
operator alternative name, [46](#)  
operator delete  
    class member, [822](#)  
    global function, [820](#), [863](#)  
operator delete[]  
    class member, [822](#)  
    compared to deallocate, [823](#)  
    global function, [820](#)  
operator new  
    class member, [822](#)  
    global function, [820](#), [863](#)  
operator new[]  
    class member, [822](#)  
    compared to allocate, [823](#)  
    global function, [820](#)  
operator overloading, see [overloaded operator](#)  
operators  
    arithmetic, [139](#)  
    assignment, [12](#), [144–147](#)  
    binary, [134](#), [168](#)  
    bitwise, [152–156](#)  
    bitset, [725](#)  
    comma (,), [157](#)  
    compound assignment, [12](#)  
    conditional (? :), [151](#)  
    decrement, [147–149](#)  
    equality, [18](#), [141](#)  
    increment, [12](#), [147–149](#)  
    input, [8](#)  
    iterator  
        addition and subtraction, [111](#)  
        arrow, [110](#)  
        dereference, [107](#)  
        equality, [106](#), [108](#)  
        increment and decrement, [107](#)  
        relational, [111](#)

logical, 141  
member access, 150  
`noexcept`, 780  
output, 7  
overloaded, arithmetic, 560  
pointer  
addition and subtraction, 119  
equality, 120  
increment and decrement, 118  
relational, 120, 123  
subscript, 121  
relational, 12, 141, 143  
`Sales_data`  
`+=` (compound assignment), 564  
`+` (addition), 560  
`==` (equality), 561  
`!=` (inequality), 561  
`>>` (input operator), 558  
`<<` (output operator), 557  
`Sales_item`, 20  
scope, 82  
`sizeof`, 156  
`sizeof...`, 700  
string  
addition, 89  
equality and relational, 88  
IO, 85  
subscript, 93–95  
subscript, 116  
`typeid`, 826, 864  
unary, 134, 169  
vector  
equality and relational, 102  
subscript, 103–105  
options to `main`, 218  
order of destruction  
    class type object, 502

- derived class object, [627](#)
- multiple inheritance, [805](#)
- virtual base classes, [815](#)
- order of evaluation, [134](#), [169](#)
  - && (logical AND), [138](#)
  - || (logical OR), [138](#)
  - , (comma operator), [138](#)
  - ? : (conditional operator), [138](#)
  - expression, [137](#)
  - pitfalls, [149](#)
- order of initialization
  - class type object, [289](#)
  - derived class object, [598](#)
  - multiple base classes, [816](#)
  - multiple inheritance, [804](#)
  - virtual base classes, [814](#)
- ordered container
  - see also [container](#)
  - see also [associative container](#)
  - key\_type requirements, [425](#)
  - lower\_bound, [438](#)
  - override default comparison, [425](#)
  - upper\_bound, [438](#)
- ordering, strict weak, [425](#), [448](#)
- OrQuery, [637](#)
  - class definition, [644](#)
  - eval function, [645](#)
- ostream, [5](#), [27](#), [311](#)
  - see also [manipulator](#)
  - << (output operator), [7](#)
  - precedence and associativity, [155](#)
  - chained output, [7](#)
  - condition state, [312](#)
  - explicit conversion to bool, [583](#)
  - file marker, [765](#)
  - floating-point notation, [757](#)

flushing output buffer, 314  
format state, 753  
no copy or assign, 311  
not flushed if program crashes, 315  
`off_type`, 766  
output format, floating-point, 755  
`pos_type`, 766  
precision member, 756  
random access, 765  
random IO program, 766  
seek and tell, 763–768  
tie member, 315  
virtual base class, 810  
`write`, 763  
`ostream_iterator`, 403, 418  
  `<<` (output operator), 405  
algorithms, 404  
initialization, 405  
operations, 405  
type requirements, 406  
`ostringstream`, 311, 321, 321–323  
  see also `ostream`  
file marker, 765  
initialization, 321  
`off_type`, 766  
phone number program, 323  
`pos_type`, 766  
random access, 765  
random IO program, 766  
seek and tell, 763–768  
`str`, 323  
`out` (file mode), 319  
out-of-range value, signed, 35  
`out_of_range`, 197  
  at function, 348  
`out_of__stock`, 783

outer scope, [48](#), [79](#)  
output, standard, [6](#)  
output iterator, [411](#), [418](#)  
overflow, [140](#)  
`overflow_error`, [197](#)  
overhead, function call, [238](#)  
overload resolution, see [function matching](#)  
overloaded function, [230](#), [230–235](#), [252](#)  
    *see also* [function matching](#)  
    as friend, [281](#)  
    compared to redeclaration, [231](#)  
    compared to template specialization, [708](#)  
    const parameters, [232](#)  
    constructor, [262](#)  
    function template, [694–699](#)  
    linkage directive, [860](#)  
    member function, [273](#)  
    const, [276](#)  
    move-enabled, [545](#)  
    reference qualified, [547](#)  
    virtual, [621](#)  
    move-enabled, [545](#)  
    namespace, [800](#)  
    pointer to, [248](#)  
    scope, [234](#)  
    derived hides base, [619](#)  
    using declaration, [800](#)  
    in derived class, [621](#)  
    using directive, [801](#)  
overloaded operator, [135](#), [169](#), [500](#), [550](#), [552](#), [590](#)  
    `++` (increment), [566–568](#)  
    `--` (decrement), [566–568](#)  
    `*` (dereference), [569](#)  
    `StrBlobPtr`, [569](#)  
    `&` (address-of), [554](#)  
    `->` (arrow operator), [569](#)

StrBlobPtr, 569  
[ ] (subscript), 564  
StrVec, 565  
( ) (call operator), 571  
absInt, 571  
PrintString, 571  
= (assignment), 500, 563  
StrVec initializer\_list, 563  
+= (compound assignment), 555, 560  
Sales\_data, 564  
+ (addition), Sales\_data, 560  
== (equality), 561  
Sales\_data, 561  
!= (inequality), 562  
Sales\_data, 561  
< (less-than), strict weak ordering, 562  
>> (input operator), 558–559  
Sales\_data, 558  
<< (output operator), 557–558  
Sales\_data, 557  
&& (logical AND), 554  
|| (logical OR), 554  
& (bitwise AND), Query, 644  
| (bitwise OR), Query, 644  
~ (bitwise NOT), Query, 643  
, (comma operator), 554  
ambiguous, 588  
arithmetic operators, 560  
associativity, 553  
binary operators, 552  
candidate function, 587  
consistency between relational and equality operators, 562  
definition, 500, 552  
design, 554–556  
equality operators, 561  
explicit call to, 553

postfix operators, [568](#)  
function matching, [587–589](#)  
member function, [500](#), [552](#)  
member vs. nonmember function, [552](#), [555](#)  
precedence, [553](#)  
relational operators, [562](#)  
requires class-type parameter, [552](#)  
short-circuit evaluation lost, [553](#)  
unary operators, [552](#)  
override, virtual function, [595](#), [650](#)  
    override specifier, [593](#), [596](#), [606](#)

## P

`pair`, [426](#), [448](#)  
    default initialization, [427](#)  
    definition, [426](#)  
    initialization, [426](#)  
    list initialization, [427](#), [431](#), [527](#)  
    `make_pair`, [428](#)  
    `map`, \* (dereference), [429](#)  
    operations, [427](#)  
    public data members, [427](#)  
    return value, [527](#)  
`Panda`, [803](#)  
`parameter`, [202](#), [208](#), [252](#)  
    array, [214–219](#)  
    buffer overflow, [215](#)  
    to pointer conversion, [214](#)  
    C-style string, [216](#)  
    `const`, [212](#)  
    copy constructor, [496](#)  
    ellipsis, [222](#)  
    forwarding, [693](#)  
    function pointer, linkage directive, [859](#)  
    `implicit this`, [257](#)

initialization, [203](#), [208](#)  
iterator, [216](#)  
lifetime, [205](#)  
low-level const, [213](#)  
main function, [218](#)  
multidimensional array, [218](#)  
nonreference, [209](#)  
uses copy constructor, [498](#)  
uses move constructor, [539](#)  
pass by reference, [210](#), [252](#)  
pass by value, [209](#), [252](#)  
passing, [208–212](#)  
pointer, [209](#), [214](#)  
pointer to const, [246](#)  
pointer to array, [218](#)  
pointer to function, [249](#)  
linkage directive, [859](#)  
reference, [210–214](#)  
to const, [213](#), [246](#)  
to array, [217](#)  
reference to const, [211](#)  
template, see [template parameter](#)  
top-level const, [212](#)

parameter list  
    function, [2](#), [27](#), [202](#)  
    template, [653](#), [714](#)

parameter pack, [714](#)  
    expansion, [702](#), [702–704](#), [714](#)  
    function template, [713](#)  
    sizeof..., [700](#)  
    variadic template, [699](#)

parentheses, override precedence, [136](#)

partial\_sort, [877](#)

partial\_sort\_copy, [877](#)

partial\_sum, [882](#)

partition, [876](#)

partition\_copy, 876  
partition\_point, 876  
pass by reference, 208, 210, 252  
pass by value, 209, 252

- uses copy constructor, 498
- uses move constructor, 539

pattern, 702, 714

- function parameter pack, 704
- regular expression, phone number, 739
- template parameter pack, 703

peek, istream, 761

PersonInfo, 321

phone number, regular expression

- program, 738
- reformat program, 742
- valid, 740

pitfalls

- dynamic memory, 462
- order of evaluation, 149
- self-assignment, 512
- smart pointer, 469
- using directive, 795

placeholders, 399

placement new, 460, 491, 824, 863

- union, class type member, 851

plus<T>, 575

pointer, 52, 52–58, 79

- ++ (increment), 118
- (decrement), 118
- \* (dereference), 53
- [ ] (subscript), 121
- = (assignment), 55
- + (addition), 119
- (subtraction), 119
- == (equality), 55, 120
- != (inequality), 55, 120

and array, 117  
arithmetic, 119, 132  
const, 63, 78  
const pointer to const, 63  
constexpr, 67  
conversion  
from array, 161  
to bool, 162  
to const, 62, 162  
to void\*, 161  
dangling, 463, 491  
declaration style, 57  
definition, 52  
delete, 460  
derived-to-base conversion, 597  
under multiple inheritance, 805  
dynamic\_cast, 825  
implicit this, 257, 306  
initialization, 52–54  
invalid, 52  
multidimensional array, 128  
null, 53, 79  
off-the-end, 118  
parameter, 209, 214  
relational operators, 123  
return type, 204  
return value, local variable, 225  
smart, 450, 491  
to const, 62  
and typedef, 68  
to array  
parameter, 218  
return type, 204  
return type declaration, 229  
to const, 79  
overloaded parameter, 232, 246

to pointer, 58  
typeid operator, 828  
valid, 52  
volatile, 856  
pointer to function, 247–250  
    auto, 249  
    callable object, 388  
    decltype, 249  
    exception specification, 779, 781  
    explicit template argument, 686  
    function template instantiation, 686  
    linkage directive, 859  
    overloaded function, 248  
    parameter, 249  
    return type, 204, 249  
    using decltype, 250  
    template argument deduction, 686  
    trailing return type, 250  
    type alias, 249  
    typedef, 249  
pointer to member, 835, 863  
    arrow (`->*`), 837  
    definition, 836  
    dot (`.*`), 837  
    function, 838  
    and bind, 843  
    and function, 842  
    and mem\_fn, 843  
    not callable object, 842  
    function call, 839  
    function table, 840  
    precedence, 838  
polymorphism, 605, 650  
pop  
    priority\_queue, 371  
    queue, 371  
    stack, 371

pop\_back  
sequential container, 348  
StrBlob, 457

pop\_front, sequential container, 348

portable, 854

precedence, 134, 136–137, 169  
= (assignment), 146  
?: (conditional operator), 151  
assignment and relational operators, 146  
dot and dereference, 150  
increment and dereference, 148  
of IO operator, 156  
overloaded operator, 553  
parentheses overrides, 136  
pointer to member and call operator, 838

precedence table, 166

precision member, ostream, 756

predicate, 386, 418  
binary, 386, 417  
unary, 386, 418

prefix, smatch, 736

preprocessor, 76, 79  
#include, 7  
assert macro, 241, 251  
header guard, 77  
variable, 54, 79

prev\_permutation, 879

print, Sales\_data, 261

print program  
array parameter, 215  
array reference parameter, 217  
pointer and size parameters, 217  
pointer parameter, 216  
two pointer parameters, 216  
variadic template, 701

print\_total  
explained, 604

program, 593  
PrintString, 571  
() (call operator), 571  
priority\_queue, 371, 373  
    emplace, 371  
    empty, 371  
    equality and relational operators, 370  
    initialization, 369  
    pop, 371  
    push, 371  
    sequential container, 371  
    size, 371  
    swap, 371  
    top, 371  
private  
    access specifier, 268, 306  
    copy constructor and assignment, 509  
    inheritance, 612, 650  
program  
    addition  
    Sales\_data, 74  
    Sales\_item, 21, 23  
    alternative\_sum, 682  
    biggies, 391  
    binops desk calculator, 577  
    book from author version 1, 438  
    book from author version 2, 439  
    book from author version 3, 440  
    bookstore  
        Sales\_data, 255  
        Sales\_data using algorithms, 406  
        Sales\_item, 24  
        buildMap, 442  
        children's story, 383–391  
        compare, 652  
        count\_calls, 206

debug\_rep  
additional nontemplate versions, 698  
general template version, 695  
nontemplate version, 697  
pointer template version, 696

elimDups, 383–391

error\_msg, 221

fact, 202

factorial, 227

factory

- new, 461
- shared\_ptr, 453
- file extension, 730
- version 2, 738

find last word, 408

find\_char, 211

findBook, 721

flip, 694

flip1, 692

flip2, 693

grade clusters, 103

grading

- bitset, 728
- bitwise operators, 154
- i before e, 729
- version 2, 734
- isShorter, 211
- letter grade, 175
- make\_plural, 224
- message handling, 519
- phone number
  - istringstream, 321
  - ostringstream, 323
- reformat, 742
- regular expression version, 738
- valid, 740
- print

array parameter, 215  
array reference parameter, 217  
pointer and size parameters, 217  
pointer parameter, 216  
two pointer parameters, 216  
variadic template, 701  
`print_total`, 593  
`Query`, 635  
class design, 636–639  
random IO, 766  
`reset`  
pointer parameters, 209  
reference parameters, 210  
`restricted word_count`, 422  
`sum`, 682  
`swap`, 223  
`TextQuery`, 486  
design, 485  
`transform`, 442  
`valid`, 740  
vector capacity, 357  
vowel counting, 179  
`word_count`  
`map`, 421  
`unordered_map`, 444  
`word_transform`, 441  
`ZooAnimal`, 802  
promotion, see integral promotion  
`protected`  
    access specifier, 595, 611, 650  
    inheritance, 612, 650  
    member, 611  
`ptr_fun` deprecated, 401  
`ptrdiff_t`, 120, 132  
`public`  
    access specifier, 268, 306  
    inheritance, 612, 650

pure virtual function, [609](#), [650](#)  
Disc\_quote, [609](#)  
Query\_base, [636](#)  
push  
    priority\_queue, [371](#)  
    queue, [371](#)  
    stack, [371](#)  
push\_back  
    back\_inserter, [382](#), [402](#)  
    sequential container, [100](#), [132](#), [342](#)  
move-enabled, [545](#)  
StrVec, [526](#)  
move-enabled, [545](#)  
push\_front  
    front\_inserter, [402](#)  
    sequential container, [342](#)  
put, istream, [761](#)  
putback, istream, [761](#)

## Q

Query, [638](#)  
    << (output operator), [641](#)  
    & (bitwise AND), [638](#)  
    definition, [644](#)  
    | (bitwise OR), [638](#)  
    definition, [644](#)  
    ~ (bitwise NOT), [638](#)  
    definition, [643](#)  
    classes, [636–639](#)  
    definition, [640](#)  
    interface class, [637](#)  
    operations, [635](#)  
    program, [635](#)  
    recap, [640](#)  
Query\_base, [636](#)  
    abstract base class, [636](#)

- definition, 639
- member function, 636

QueryResult, 485

- class definition, 489
- nested class, 844
- constructor, 845
- print, 490

queue, 371, 373

- back, 371
- emplace, 371
- empty, 371
- equality and relational operators, 370
- front, 371
- header, 371
- initialization, 369
- pop, 371
- push, 371
- sequential container, 371
- size, 371
- swap, 371

Quote

- class definition, 594
- design, 592

## R

Raccoon, virtual base class, 812

raise exception, see `throw`

rand function, drawbacks, 745

random header, 745

random IO, 765

- machine-dependent, 763
- program, 766

random-access iterator, 412, 418

random-number library, 745

- compared to rand function, 745
- distribution types, 745, 770

engine, [745](#), [770](#)  
default\_random\_engine, [745](#)  
max, min, [747](#)  
retain state, [747](#)  
seed, [748](#), [770](#)  
generator, [746](#), [770](#)  
range, [747](#)  
random\_shuffle, [878](#)  
range for statement, [91](#), [132](#), [187](#), [187–189](#), [200](#)  
can't add elements, [101](#), [188](#)  
multidimensional array, [128](#)  
not with dynamic array, [477](#)  
range\_error, [197](#)  
rbegin, container, [333](#), [407](#)  
rdstate, stream, [313](#)  
read  
    istream, [763](#)  
    Sales\_data, [261](#)  
realloc, StrVec, [530](#)  
move iterator version, [543](#)  
recursion loop, [228](#), [252](#), [608](#)  
recursive function, [227](#), [252](#)  
    variadic template, [701](#)  
ref, binds reference parameter, [400](#), [418](#)  
refactoring, [611](#), [650](#)  
reference, [50](#), [79](#)  
    see also lvalue reference  
    see also rvalue reference  
    auto deduces referred to type, [69](#)  
    collapsing rule, [688](#)  
    forward, [694](#)  
    lvalue arguments, [688](#)  
    move, [691](#)  
    rvalue reference parameters, [693](#)  
    const, see [reference to const](#)  
    conversion

not from `const`, [61](#)  
to reference to `const`, [162](#)  
data member, initialization, [289](#)  
declaration style, [57](#)  
`decltype` yields reference type, [71](#)  
definition, [51](#)  
derived-to-base conversion, [597](#)  
under multiple inheritance, [805](#)  
`dynamic_cast` operator, [826](#)  
initialization, [51](#)  
member function, [546](#)  
parameter, [210–214](#)  
`bind`, [400](#)  
limitations, [214](#)  
template argument deduction, [687–689](#)  
`remove_reference`, [684](#)  
return type, [224](#)  
assignment operator, [500](#)  
is lvalue, [226](#)  
return value, local variable, [225](#)  
to array parameter, [217](#)  
`reference`, container, [333](#)  
reference count, [452](#), [491](#), [514](#), [550](#)  
    copy assignment, [514](#)  
    copy constructor, [514](#)  
    design, [514](#)  
    destructor, [514](#)  
    `HasPtr` class, [514–516](#)  
reference to `const`, [61](#), [80](#)  
    argument, [211](#)  
    initialization, [61](#)  
    parameter, [211](#), [213](#)  
    overloaded, [232](#), [246](#)  
    return type, [226](#)  
`regex`, [728](#), [770](#)  
    `error_type`, [732](#)

header, 728  
  regex\_error, 732, 770  
  syntax\_option\_type, 730  
regex\_constants, 743  
  match\_flag\_type, 743  
regex\_error, 732, 770  
regex\_match, 729, 770  
regex\_replace, 742, 770  
  format flags, 744  
  format string, 742  
regex\_search, 729, 730, 770  
regular expression library, 728, 770  
  case sensitive, 730  
  compiled at run time, 732  
  ECMAScript, 730  
  file extension program, 730  
  i before e program, 729  
  version 2, 734  
  match data, 735–737  
  pattern, 729  
  phone number, valid, 740  
  phone number pattern, 739  
  phone number program, 738  
  phone number reformat, program, 742  
  regex iterators, 734  
  search functions, 729  
  smatch, provides context for a match, 735  
  subexpression, 738  
  file extension program version 2, 738  
  types, 733  
  valid, program, 740  
reinterpret\_cast, 163, 164  
  machine-dependent, 164  
relational operators, 141, 143  
  arithmetic conversion, 144  
  container adaptor, 370

container member, 340  
function object, 574  
iterator, 111  
overloaded operator, 562  
pointer, 120, 123  
Sales\_data, 563  
string, 88  
tuple, 720  
vector, 102  
release, unique\_ptr, 470  
remove, 878  
    list and forward\_list, 415  
remove\_copy, 878  
remove\_copy\_if, 878  
remove\_from\_Folders, Message, 523  
remove\_if, 878  
    list and forward\_list, 415  
remove\_pointer, 685  
remove\_reference, 684  
    move, 691  
rend, container, 333, 407  
replace, 383, 875  
    string, 362  
replace\_copy, 383, 874  
replace\_copy\_if, 874  
replace\_if, 875  
reserve  
    string, 356  
    vector, 356  
reserved identifiers, 46  
reset  
    bitset, 727  
    shared\_ptr, 466  
    unique\_ptr, 470  
reset program  
    pointer parameters, 209  
    reference parameters, 210

resize  
invalidates iterator, 352  
sequential container, 352  
value initialization, 352

restricted word\_count program, 422

result, **134**, 169

- \* (dereference), lvalue, 135
- [ ] (subscript), lvalue, 135
- , (comma operator), lvalue, 158
- ? : (conditional operator), lvalue, 151
- cast, lvalue, 163

rethrow, **776**

- exception object, **777**
- throw, **776**, 818

return statement, **222**, 222–228

- from main, 227
- implicit return from main, 223
- local variable, 224, 225

return type, **2**, 27, **202**, 204, 252

- array, 204
- array using decltype, 230
- function, 204
- function pointer, 249
- using decltype, 250
- linkage directive, 859
- main, 2
- member function, 283
- nonreference, 224
- copy initialized, 498
- pointer, 204
- pointer to function, 204
- reference, 224
- reference to const, 226
- reference yields lvalue, 226
- trailing, **229**, 252, 396, 684
- virtual function, 606

void, 223  
return value  
    conversion, 223  
    copy initialized, 498  
    initialization, 224  
    list initialization, 226, 427, 527  
    local variable, pointer, 225  
    main, 2–4, 227  
    pair, 427, 527  
    reference, local variable, 225  
    \*this, 260, 275  
    tuple, 721  
    type checking, 223  
    unique\_ptr, 471  
reverse, 878  
    list and forward\_list, 415  
reverse iterator, 401, 407–409, 418  
    ++ (increment), 407  
    -- (decrement), 407, 408  
    base, 409  
    compared to iterator, 409  
reverse\_copy, 414, 878  
reverse\_copy\_if, 414  
reverse\_iterator  
    compared to iterator, 408  
    container, 332, 407  
rfind, string, 366  
right, manipulator, 758  
rotate, 878  
rotate\_copy, 878  
rule of three/five, 505, 541  
    virtual destructor exception, 622  
run-time type identification, 825–831, 864  
    compared to virtual functions, 829  
    dynamic\_cast, 825, 825  
    bad\_cast, 826  
    to pointer, 825

- to reference, 826
- type-sensitive equality, 829
- `typeid`, **826**, 827
  - returns `type_info`, 827
- runtime binding, **594**, 650
- `runtime_error`, **194**, 197
  - initialization from `string`, 196
- `rvalue`, **135**, 169
  - copy initialization, uses move constructor, 539
  - result
    - `++` (increment) postfix, 148
    - `--` (decrement) postfix, 148
  - function nonreference return type, 224
- `rvalue reference`, **532**, 550
  - cast from `lvalue`, 691
  - collapsing rule, 688
  - compared to `lvalue reference`, 533
  - function matching, 539
  - initialization, 532
  - member function, **546**
  - overloaded, 547
  - `move`, 533
  - parameter
    - forwarding, 693, 705
  - member function, 544
  - preserves argument type information, 693
  - template argument deduction, 687
  - variable, 533

## S

- `Sales_data`
  - `compareIsbn`, **387**
  - `+=` (compound assignment), **564**
  - `+` (addition), **560**
  - `==` (equality), **561**
  - `!=` (inequality), **561**

>> (input operator), 558  
<< (output operator), 557  
add, 261  
addition program, 74  
avg\_price, 259  
bookstore program, 255  
using algorithms, 406  
class definition, 72, 268  
combine, 259  
compareIsbn, 425  
with associative container, 426  
constructors, 264–266  
converting constructor, 295  
default constructor, 262  
exception classes, 783  
exception version  
+= (compound assignment), 784  
+ (addition), 784  
explicit constructor, 296  
isbn, 257  
operations, 254  
print, 261  
read, 261  
relational operators, 563  
`Sales_data.h` header, 76  
`Sales_item`, 20  
+ (addition), 22  
>> (input operator), 21  
<< (output operator), 21  
addition program, 21, 23  
bookstore program, 24  
isbn, 23  
operations, 20  
`Sales_item.h` header, 19  
scientific manipulator, 757  
scope, 48, 80

base class, 617  
block, 48, 80, 173  
class, 73, 282, 282–287, 305  
static member, 302  
compared to object lifetime, 204  
derived class, 617  
friend, 270, 281  
function, 204  
global, 48, 80  
inheritance, 617–621  
member function, 282  
parameters and return type, 283  
multiple inheritance, 807  
name collisions, using directive, 795  
namespace, 785–790  
nested class, 844  
overloaded function, 234  
statement, 174  
template parameter, 668  
template specialization, 708  
using directive, 794  
virtual function, 620  
scoped enumeration, 832, 864  
  enum class, 832  
Screen, 271  
  pos member, 272  
  concatenating operations, 275  
  do\_display, 276  
  friends, 279  
  get, 273, 282  
  get\_cursor, 283  
  Menu function table, 840  
  move, 841  
  move members, 275  
  set, 275  
search, 872

search\_n, 871  
seed, random-number engine, 748  
seekp, seekg, 763–768  
self-assignment  
    copy and swap assignment, 519  
    copy assignment, 512  
    explicit check, 542  
    HasPtr  
        reference counted, 515  
        valuelike, 512  
    Message, 523  
    move assignment, 537  
    pitfalls, 512  
    StrVec, 528  
semicolon (;), 3  
    class definition, 73  
    null statement, 172  
separate compilation, 44, 80, 252  
    compiler options, 207  
    declaration vs. definition, 44  
    templates, 656  
sequential container, 326, 373  
    array, 326  
    deque, 326  
    forward\_list, 326  
    initialization, 334–337  
    list, 326  
    list initialization, 336  
    members  
        assign, 338  
        back, 346  
        clear, 350  
        emplace, 345  
        emplace\_back, 345  
        emplace\_front, 345  
        erase, 349

front, 346  
insert, 343  
pop\_back, 348  
pop\_front, 348  
push\_back, 132  
push\_back, 100, 342, 545  
push\_front, 342  
resize, 352  
value\_type, 333  
performance characteristics, 327  
priority\_queue, 371  
queue, 371  
stack, 370  
value initialization, 336  
vector, 326  
set, 420, 448  
    see also ordered container  
    bitset, 727  
    header, 420  
    insert, 431  
    iterator, 429  
    key\_type requirements, 425  
    list initialization, 423  
    lower\_bound, 438  
    TextQuery class, 485  
    upper\_bound, 438  
    word\_count program, 422  
set\_difference, 880  
set\_intersection, 647, 880  
set\_symmetric\_difference, 880  
set\_union, 880  
setfill, manipulator, 759  
setprecision, manipulator, 756  
setstate, stream, 313  
setw, manipulator, 758  
shared\_ptr, 450, 450–457, 464–469, 491

- \* (dereference), [451](#)
- copy and assignment, [451](#)
- definition, [450](#)
- deleter, [469](#), [491](#)
- bound at run time, [677](#)
- derived-to-base conversion, [630](#)
- destructor, [453](#)
- dynamically allocated array, [480](#)
- exception safety, [467](#)
- factory program, [453](#)
- initialization, [464](#)
- `make_shared`, [451](#)
- pitfalls, [469](#)
- `reset`, [466](#)
- `StrBlob`, [455](#)
- `TextQuery` class, [485](#)
- `with new`, [464](#)
- `short`, [33](#)
- short-circuit evaluation, [142](#), [169](#)
  - `&&` (logical AND), [142](#)
  - `||` (logical OR), [142](#)
  - not in overloaded operator, [553](#)
- `ShorterString`, [573](#)
  - `()` (call operator), [573](#)
- `shorterString`, [224](#)
- `showbase`, manipulator, [755](#)
- `showpoint`, manipulator, [758](#)
- `shrink_to_fit`
  - `deque`, [357](#)
  - `string`, [357](#)
  - `vector`, [357](#)
- `shuffle`, [878](#)
- `signed`, [34](#), [80](#)
  - `char`, [34](#)
  - conversion to `unsigned`, [34](#), [160](#)
  - out-of-range value, [35](#)
- `signed type`, [34](#)

single-line (//), comment, [9](#), [26](#)  
size  
    container, [88](#), [102](#), [132](#), [340](#)  
    priority\_queue, [371](#)  
    queue, [371](#)  
    returns unsigned, [88](#)  
    stack, [371](#)  
    StrVec, [526](#)  
size\_t, [116](#), [132](#), [727](#)  
    array subscript, [116](#)  
size\_type, container, [88](#), [102](#), [132](#), [332](#)  
SizeComp, [573](#)  
    () (call operator), [573](#)  
sizeof, [156](#), [169](#)  
    array, [157](#)  
    data member, [157](#)  
sizeof..., parameter pack, [700](#)  
skipws, manipulator, [760](#)  
sliced, [603](#), [650](#)  
SmallInt  
    + (addition), [588](#)  
    conversion operator, [580](#)  
smart pointer, [450](#), [491](#)  
    exception safety, [467](#)  
    pitfalls, [469](#)  
smatch, [729](#), [733](#), [769](#), [770](#)  
    prefix, [736](#)  
    provide context for a match, [735](#)  
    suffix, [736](#)  
sort, [384](#), [876](#)  
source file, [4](#), [27](#)  
specialization, see [template specialization](#)  
splice, list, [416](#)  
splice\_after, forward\_list, [416](#)  
sregex\_iterator, [733](#), [770](#)  
    i before e program, [734](#)  
sstream

file marker, 765  
header, 310, 321  
off\_type, 766  
pos\_type, 766  
random access, 765  
random IO program, 766  
seek and tell, 763–768  
`ssub_match`, 733, 736, 770  
example, 740  
`stable_partition`, 876  
`stable_sort`, 387, 876  
stack, 370, 373  
    emplace, 371  
    empty, 371  
    equality and relational operators, 370  
    header, 370  
    initialization, 369  
    pop, 371  
    push, 371  
    sequential container, 370  
    size, 371  
    swap, 371  
    top, 371  
stack unwinding, exception handling, 773, 818  
standard error, 6, 27  
standard header, #include, 6, 21  
standard input, 6, 27  
standard library, 5, 27  
standard output, 6, 27  
statement, 2, 27  
    block, see `block`  
    `break`, 190, 199  
    compound, 173, 199  
    `continue`, 191, 199  
    `do while`, 189, 200  
    expression, 172, 200

for, **13**, **27**, **185**, 185–187, 200  
goto, **192**, 200  
if, **17**, **27**, **175**, 175–178, 200  
labeled, **192**, 200  
null, **172**, 200  
range for, **91**, **187**, 187–189, 200  
return, **222**, 222–228  
scope, 174  
switch, **178**, 178–182, 200  
while, **11**, **28**, **183**, 183–185, 200  
statement label, 192  
**static** (file static), **792**, 817  
static member  
    Account, 301  
    class template, 667  
    accessed through an instantiation, 667  
    definition, 667  
    const data member, initialization, 302  
    data member, 300  
    definition, 302  
    default argument, 304  
    definition, 302  
    inheritance, 599  
    instantiation, 667  
    member function, 301  
    nested class, 845  
    scope, 302  
**static object**, local, **205**, 252  
**static type**, **601**, 650  
    determines name lookup, **617**, 619  
    multiple inheritance, 806  
**static type checking**, **46**  
**static\_cast**, **163**, **163**  
    lvalue to rvalue, 691  
**std**, **7**, 28  
**std::forward**, see **forward**

std::move, see [move](#)  
stdexcept header, [194](#), [197](#)  
stod, [368](#)  
stof, [368](#)  
stoi, [368](#)  
stol, [368](#)  
stold, [368](#)  
stoll, [368](#)  
store, free, [450](#), [491](#)  
stoul, [368](#)  
stoull, [368](#)  
str, string streams, [323](#)  
StrBlob, [456](#)  
    back, [457](#)  
    begin, [475](#)  
    check, [457](#)  
    constructor, [456](#)  
    end, [475](#)  
    front, [457](#)  
    pop\_back, [457](#)  
    shared\_ptr, [455](#)  
StrBlobPtr, [474](#)  
    ++ (increment), [566](#)  
    -- (decrement), [566](#)  
    \* (dereference), [569](#)  
    -> (arrow operator), [569](#)  
    check, [474](#)  
    constructor, [474](#)  
    deref, [475](#)  
    incr, [475](#)  
    weak\_ptr, [474](#)  
strcat, [123](#)  
strcmp, [123](#)  
strcpy, [123](#)  
stream  
    as condition, [15](#), [162](#), [312](#)

clear, 313  
explicit conversion to `bool`, 583  
file marker, 765  
flushing buffer, 314  
format state, 753  
`istream_iterator`, 403  
`iterator`, 401, 403–406, 418  
type requirements, 406  
not flushed if program crashes, 315  
`ostream_iterator`, 403  
random IO, 765  
`rdstate`, 313  
`setstate`, 313  
strict weak ordering, 425, 448  
`string`, 80, 84–93, 132  
    see also `container`  
    see also `sequential container`  
    see also `iterator`  
    `[ ]` (subscript), 93, 132, 347  
    `+=` (compound assignment), 89  
    `+` (addition), 89  
    `>>` (input operator), 85, 132  
    `>>` (input operator) as condition, 86  
    `<<` (output operator), 85, 132  
    and string literal, 89–90  
    append, 362  
    assign, 362  
    at, 348  
    C-style string, 124  
    `c_str`, 124  
    capacity, 356  
    case sensitive, 365  
    compare, 366  
    concatenation, 89  
    default initialization, 44  
    difference\_type, 112

equality and relational operators, 88  
erase, 362  
find, 364  
find\_first\_not\_of, 365  
find\_last\_not\_of, 366  
find\_last\_of, 366  
getline, 87, 321  
header, 74, 76, 84  
initialization, 84–85, 360–361  
initialization from string literal, 84  
insert, 362  
move constructor, 529  
numeric conversions, 367  
random-access iterator, 412  
replace, 362  
reserve, 356  
rfind, 366  
subscript range, 95  
substr, 361  
TextQuery class, 485  
string literal, 7, 28, 39  
    see also C-style string  
    and string, 89–90  
    concatenation, 39  
stringstream, 321, 321–323, 324  
    initialization, 321  
strlen, 122  
struct  
    see also class  
    default access specifier, 268  
    default inheritance specifier, 616  
StrVec, 525  
    [] (subscript), 565  
    = (assignment), initializer\_list, 563  
    alloc\_n\_copy, 527  
    begin, 526

capacity, 526  
chk\_n\_alloc, 526  
copy assignment, 528  
copy constructor, 528  
default constructor, 526  
design, 525  
destructor, 528  
emplace\_back, 704  
end, 526  
free, 528  
memory allocation strategy, 525  
move assignment, 536  
move constructor, 535  
push\_back, 526  
move-enabled, 545  
reallocate, 530  
move iterator version, 543  
size, 526  
subexpression, 770  
subscript range, 93  
    array, 116  
    string, 95  
    validating, 104  
    vector, 105  
substr, string, 361  
suffix, smatch, 736  
sum, program, 682  
swap, 516  
    array, 339  
    container, 339  
    container nonmember version, 339  
    copy and swap assignment operator, 518  
    priority\_queue, 371  
    queue, 371  
    stack, 371  
typical implementation, 517–518

swap program, 223  
swap\_ranges, 875  
switch statement, 178, 178–182, 200  
    default case label, 181  
    break, 179–181, 190  
    compared to if, 178  
    execution flow, 180  
    variable definition, 182

syntax\_option\_type, regex, 730

synthesized

    copy assignment, 500, 550  
    copy constructor, 497, 550  
    copy control, 267  
    as deleted function, 508  
    as deleted in derived class, 624  
    Bulk\_quote, 623  
    multiple inheritance, 805  
    virtual base class, 815  
    virtual base classes, 815  
    volatile, 857  
    default constructor, 263, 306  
    derived class, 623  
    members of built-in type, 263  
    destructor, 503, 550  
    move operations  
    deleted function, 538  
    not always defined, 538

## T

\t (tab character), 39  
tellp, tellg, 763–768  
template  
    see also class template  
    see also function template  
    see also instantiation  
    declaration, 669

link time errors, [657](#)  
overview, [652](#)  
parameter, see [template parameter](#)  
parameter list, [714](#)  
template argument, [653](#), [714](#)  
explicit, [660](#), [713](#)  
template member, see [member template](#)  
type alias, [666](#)  
type transformation templates, [684](#), [714](#)  
type-dependencies, [658](#)  
variadic, see [variadic template](#)  
template argument deduction, [678](#), [714](#)  
    compare, [680](#)  
    explicit template argument, [682](#)  
    function pointer, [686](#)  
    limited conversions, [679](#)  
    low-level const, [693](#)  
    lvalue reference parameter, [687](#)  
    multiple function parameters, [680](#)  
    parameter with nontemplate type, [680](#)  
    reference parameters, [687](#)–[689](#)  
    rvalue reference parameter, [687](#)  
    top-level const, [679](#)  
template class, see [class template](#)  
template function, see [function template](#)  
template parameter, [653](#), [714](#)  
    default template argument, [670](#)  
    class template, [671](#)  
    function template, [671](#)  
    name, [668](#)  
    restrictions on use, [669](#)  
    nontype parameter, [654](#), [714](#)  
    must be constant expression, [655](#)  
    type requirements, [655](#)  
    scope, [668](#)  
    template argument deduction, [680](#)

type parameter, 654, **654**, 714  
as friend, 666  
used in template class, 660

template parameter pack, **699**, 714  
expansion, 703  
pattern, 703

template specialization, **707**, 706–712, 714  
class template, 709–712  
class template member, 711  
compare function template, 706  
compared to overloading, 708  
declaration dependencies, 708  
function template, 707  
`hash<key_type>`, 709, 788  
headers, 708  
of namespace member, 709, 788  
partial, class template, **711**, 714  
scope, 708  
`template<>`, 707

`template<>`  
default template argument, 671  
template specialization, 707

temporary, **62**, 80

terminate function, **773**, 818  
exception handling, **196**, 200  
machine-dependent, 196

terminology  
const reference, 61  
iterator, 109  
object, 42  
overloaded new and delete, 822

test, `bitset`, 727

`TextQuery`, 485  
class definition, 487  
constructor, 488  
main program, 486

program design, 485  
query, 489  
revisited, 635  
this pointer, 257, 306  
    static members, 301  
    as argument, 266  
    in return, 260  
    overloaded  
    on const, 276  
    on lvalue or rvalue reference, 546  
throw, 193, 193, 200, 772, 818  
    execution flow, 196, 773  
    pointer to local object, 774  
    rethrow, 776, 818  
    runtime\_error, 194  
throw(), exception specification, 780  
tie member, ostream, 315  
to\_string, 368  
Token, 849  
    assignment operators, 850  
    copy control, 851  
    copyUnion, 851  
    default constructor, 850  
    discriminant, 850  
tolower, 92  
top  
    priority\_queue, 371  
    stack, 371  
top-level const, 64, 80  
    and auto, 69  
    argument and parameter, 212  
    decltype, 71  
    parameter, 232  
    template argument deduction, 679  
toupper, 92  
ToyAnimal, virtual base class, 815  
trailing return type, 229, 252

- function template, [684](#)
- lambda expression, [396](#)
- pointer to array, [229](#)
- pointer to function, [250](#)
- transform
  - algorithm, [396](#), [874](#)
  - program, [442](#)
- translation unit, [4](#)
- trunc (file mode), [319](#)
- try block, [193](#), [194](#), [200](#), [773](#), [818](#)
- tuple, [718](#), [770](#)
  - findBook, program, [721](#)
  - equality and relational operators, [720](#)
  - header, [718](#)
  - initialization, [718](#)
  - make\_tuple, [718](#)
  - return value, [721](#)
  - value initialization, [718](#)
- type
  - alias, [67](#), [80](#)
  - template, [666](#)
  - alias declaration, [68](#)
  - arithmetic, [32](#), [78](#)
  - built-in, [2](#), [26](#), [32–34](#)
  - checking, [46](#), [80](#)
  - argument and parameter, [203](#)
  - array reference parameter, [217](#)
  - function return value, [223](#)
  - name lookup, [235](#)
  - class, [19](#), [26](#)
  - compound, [50](#), [50–58](#), [78](#)
  - conversion, see [conversion](#)
  - dynamic, [601](#), [650](#)
  - incomplete, [279](#), [306](#)
  - integral, [32](#), [79](#)
  - literal, [66](#)

class type, 299  
specifier, 41, 80  
static, 601, 650

type alias declaration, 68, 78, 80  
pointer, to array, 229  
pointer to function, 249  
pointer to member, 839  
template type, 666

type independence, algorithms, 377

type member, class, 271

type parameter, see template parameter

type transformation templates, 684, 714  
    type\_traits, 685

type\_info, 864  
    header, 197  
    name, 831  
    no copy or assign, 831  
    operations, 831  
    returned from typeid, 827

type\_traits  
    header, 684  
    remove\_pointer, 685  
    remove\_reference, 684  
    and move, 691  
    type transformation templates, 685

typedef, 67, 80  
    const, 68  
    and pointer, to const, 68  
    pointer, to array, 229  
    pointer to function, 249

typeid operator, 826, 827, 864  
    returns type\_info, 827

typeinfo header, 826, 827, 831

typename  
    compared to class, 654  
    required for type member, 670  
    template parameter, 654

# U

unary operators, [134](#), [169](#)

overloaded operator, [552](#)

unary predicate, [386](#), [418](#)

unary\_function deprecated, [579](#)

uncaught exception, [773](#)

undefined behavior, [35](#), [80](#)

base class destructor not virtual, [622](#)

bitwise operators and signed values, [153](#)

caching end( ) iterator, [355](#)

cstring functions, [122](#)

dangling pointer, [463](#)

default initialized members of built-in type, [263](#)

delete of invalid pointer, [460](#)

destination sequence too small, [382](#)

element access empty container, [346](#)

invalidated iterator, [107](#), [353](#)

missing return statement, [224](#)

misuse of smart pointer get, [466](#)

omitting [ ] when deleting array, [479](#)

operand order of evaluation, [138](#), [149](#)

out-of-range subscript, [93](#)

out-of-range value assigned to signed type, [35](#)

overflow and underflow, [140](#)

pointer casts, [163](#)

pointer comparisons, [123](#)

return reference or pointer to local variable, [225](#)

string invalid initializer, [361](#)

uninitialized

dynamic object, [458](#)

local variable, [205](#)

pointer, [54](#)

variable, [45](#)

using unconstructed memory, [482](#)

using unmatched match object, [737](#)  
writing to a `const` object, [163](#)  
wrong deleter with smart pointer, [480](#)

`underflow_error`, [197](#)

unformatted IO, [761](#), [770](#)  
    `istream`, [761](#)  
    multi-byte, `istream`, [763](#)  
    single-byte, `istream`, [761](#)

`unget`, `istream`, [761](#)

`uniform_int_distribution`, [746](#)

`uniform_real_distribution`, [750](#)

uninitialized, [8](#), [28](#), [44](#), [80](#)  
    pointer, undefined behavior, [54](#)  
    variable, undefined behavior, [45](#)

uninitialized\_copy, [483](#)  
    move iterator, [543](#)

uninitialized\_fill, [483](#)

union, [847](#), [864](#)  
    anonymous, [848](#), [862](#)  
    class type member, [848](#)  
    assignment operators, [850](#)  
    copy control, [851](#)  
    default constructor, [850](#)  
    deleted copy control, [849](#)  
    placement `new`, [851](#)  
    definition, [848](#)  
    discriminant, [850](#)  
    restrictions, [847](#)

unique, [384](#), [878](#)  
    list and `forward_list`, [415](#)

unique\_copy, [403](#), [878](#)

unique\_ptr, [450](#), [470–472](#), [491](#)  
    \* (dereference), [451](#)  
    copy and assignment, [470](#)  
    definition, [470](#), [472](#)  
    deleter, [472](#), [491](#)

bound at compile time, [678](#)  
dynamically allocated array, [479](#)  
initialization, [470](#)  
pitfalls, [469](#)  
release, [470](#)  
reset, [470](#)  
return value, [471](#)  
transfer ownership, [470](#)  
with `new`, [470](#)

`unitbuf`, manipulator, [315](#)

unnamed namespace, [791](#), [818](#)  
local to file, [791](#)  
replace file static, [792](#)

unordered container, [443](#), [448](#)  
*see also* `container`  
*see also* `associative container`  
bucket management, [444](#)  
`hash<key_type>` specialization, [709](#), [788](#)  
compatible with `==` (equality), [710](#)  
key\_type requirements, [445](#)  
override default hash, [446](#)

`unordered_map`, [448](#)  
*see also* `unordered container`  
`*` (dereference), [429](#)  
`[ ]` (subscript), [435](#), [448](#)  
adds element, [435](#)  
`at`, [435](#)  
definition, [423](#)  
header, [420](#)  
list initialization, [423](#)  
`word_count` program, [444](#)

`unordered_multimap`, [448](#)  
*see also* `unordered container`  
`*` (dereference), [429](#)  
definition, [423](#)  
has no subscript operator, [435](#)

insert, 433  
list initialization, 423  
unordered\_multiset, 448  
see also unordered container  
insert, 433  
iterator, 429  
list initialization, 423  
override default hash, 446  
unordered\_set, 448  
see also unordered container  
header, 420  
iterator, 429  
list initialization, 423  
unscoped enumeration, 832, 864  
as union discriminant, 850  
conversion to integer, 834  
enum, 832  
unsigned, 34, 80  
char, 34  
conversion, 36  
conversion from signed, 34  
conversion to signed, 160  
literal (`numU` or `numu`), 41  
size return type, 88  
unsigned type, 34  
unwinding, stack, 773, 818  
upper\_bound  
algorithm, 873  
ordered container, 438  
used in Basket, 632  
uppercase, manipulator, 755  
use count, see reference count  
user-defined conversion, see class type conversion  
user-defined header, 76–77  
const and constexpr, 76  
default argument, 238  
function declaration, 207

#include, 21  
inline function, 240  
inline member function definition, 273  
template definition, 656  
template specialization, 708  
using =, see type alias declaration  
using declaration, 82, 132, 793, 818  
    access control, 615  
    not in header files, 83  
    overloaded function, 800  
    overloaded inherited functions, 621  
    scope, 793  
using directive, 793, 818  
    overloaded function, 801  
    pitfalls, 795  
    scope, 793, 794  
    name collisions, 795  
utility header, 426, 530, 533, 694

## V

valid, program, 740  
valid but unspecified, 537  
valid pointer, 52  
value initialization, 98, 132  
    dynamically allocated, object, 459  
    map subscript operator, 435  
    new[], 478  
    resize, 352  
    sequential container, 336  
    tuple, 718  
    uses default constructor, 293  
    vector, 98  
value\_type  
    associative container, 428, 448  
    sequential container, 333  
valuelike class, copy control, 512

varargs, 222  
variable, 8, 28, 41, 41–49, 80  
    const, 59  
    constexpr, 66  
    declaration, 45  
    class type, 294  
    define before use, 46  
    defined after label, 182, 192  
    definition, 41, 45  
    extern, 45  
    extern and const, 60  
    initialization, 42, 43, 79  
    is lvalue, 533  
    lifetime, 204  
    local, 204, 252  
    preprocessor, 79  
variadic template, 699, 714  
    declaration dependencies, 702  
    forwarding, 704  
    usage pattern, 706  
    function matching, 702  
    pack expansion, 702–704  
    parameter pack, 699  
    print program, 701  
    recursive function, 701  
    sizeof..., 700  
vector, 96–105, 132, 373  
    see also container  
    see also sequential container  
    see also iterator  
    [] (subscript), 103, 132, 347  
    = (assignment), list initialization, 145  
    at, 348  
    capacity, 356  
    capacity program, 357  
    definition, 97

difference\_type, [112](#)  
erase, changes container size, [385](#)  
header, [96](#), [329](#)  
initialization, [97–101](#), [334–337](#)  
initialization from array, [125](#)  
list initialization, [98](#), [336](#)  
memory management, [355](#)  
overview, [326](#)  
push\_back, invalidates iterator, [354](#)  
random-access iterator, [412](#)  
reserve, [356](#)  
subscript range, [105](#)  
TextQuery class, [485](#)  
value initialization, [98](#), [336](#)  
viable function, [243](#), [252](#)  
    see also [function matching](#)  
virtual base class, [811](#), [818](#)  
    ambiguities, [812](#)  
    Bear, [812](#)  
    class derivation list, [812](#)  
    conversion, [812](#)  
    derived class constructor, [813](#)  
    iostream, [810](#)  
    name lookup, [812](#)  
    order of destruction, [815](#)  
    order of initialization, [814](#)  
    ostream, [810](#)  
    Raccoon, [812](#)  
    ToyAnimal, [815](#)  
    ZooAnimal, [811](#)  
virtual function, [592](#), [595](#), [603–610](#), [650](#)  
    compared to run-time type identification, [829](#)  
    default argument, [607](#)  
    derived class, [596](#)  
    destructor, [622](#)  
    exception specification, [781](#)

- final specifier, [607](#)
- in constructor, destructor, [627](#)
- multiple inheritance, [807](#)
- overloaded function, [621](#)
- override, [595](#), [650](#)
- override specifier, [593](#), [596](#), [606](#)
- overriding run-time binding, [607](#)
- overview, [595](#)
- pure, [609](#)
- resolved at run time, [604](#), [605](#)
- return type, [606](#)
- scope, [620](#)
- type-sensitive equality, [829](#)
- virtual inheritance, see [virtual base class](#)
- Visual Studio, [5](#)
- void, [32](#), [80](#)
  - return type, [223](#)
- void\*, [56](#), [80](#)
  - conversion from pointer, [161](#)
- volatile, [856](#), [864](#)
  - pointer, [856](#)
  - synthesized copy-control members, [857](#)
- vowel counting, program, [179](#)

## W

- wcerr, [311](#)
- wchar\_t, [33](#)
  - literal, [40](#)
- wchar\_t streams, [311](#)
- wcin, [311](#)
- wcout, [311](#)
- weak ordering, strict, [448](#)
- weak\_ptr, [450](#), [473–475](#), [491](#)
  - definition, [473](#)
  - initialization, [473](#)

lock, 473  
StrBlobPtr, 474  
wfstream, 311  
what, exception, 195, 782  
while statement, 11, 28, 183, 183–185, 200  
    condition, 12, 183  
wide character streams, 311  
wifstream, 311  
window, console, 6  
Window\_mngr, 279  
wiostream, 311  
wistream, 311  
wistringstream, 311  
wofstream, 311  
word, 33, 80  
word\_count program  
    map, 421  
    set, 422  
    unordered\_map, 444  
word\_transform program, 441  
WordQuery, 637, 642  
wostream, 311  
wostringstream, 311  
wregex, 733  
write, ostream, 763  
wstringstream, 311

## X

\xnnn (hexadecimal escape sequence), 39

## Z

ZooAnimal  
    program, 802  
    virtual base class, 811

## Take the Next Step to Mastering C++



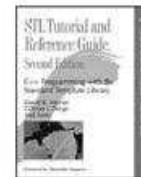
978-0-321-54372-1



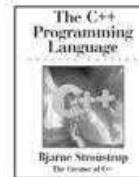
978-0-321-62321-8



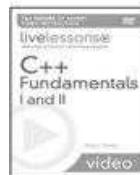
978-0-201-73484-3



978-0-321-70212-8



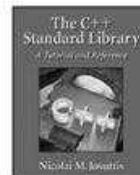
978-0-201-70073-2



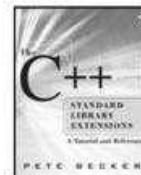
978-0-13-704483-2



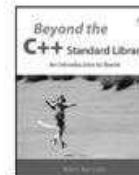
978-0-13-700130-9



978-0-201-37926-5



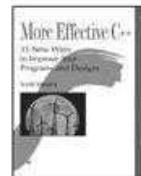
978-0-321-41299-7



978-0-321-13354-0



978-0-321-33487-9



978-0-201-63371-9



978-0-201-74962-5



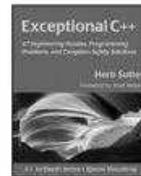
978-0-321-32192-3



978-0-321-11358-0



978-0-201-76042-2



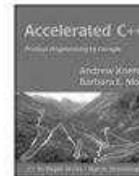
978-0-201-61562-3



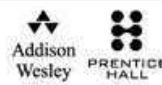
978-0-201-70434-1



978-0-201-70431-0



978-0-201-70353-5



For more information on these titles  
[visit informit.com](http://informit.com)





# REGISTER



## THIS PRODUCT

[informit.com/register](http://informit.com/register)

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to [informit.com/register](http://informit.com/register) to sign in or create an account.

You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

### About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall

Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

# informIT.com

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram  
IBM Press | Que | Prentice Hall | Sams  
SAFARI BOOKS ONLINE



THE TRUSTED TECHNOLOGY LEARNING SOURCE



**InformIT** is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

Addison-Wesley

Cisco Press

EXAM/CRAM

IBM

Press

QUE'

PRENTICE HALL

SAMS

Safari®

## LearnIT at **InformIT**

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters.  
Visit [informit.com/newsletters](http://informit.com/newsletters).
- Access FREE podcasts from experts at [informit.com/podcasts](http://informit.com/podcasts).
- Read the latest author articles and sample chapters at [informit.com/articles](http://informit.com/articles).
- Access thousands of books and videos in the Safari Books Online digital library at [safari.informit.com](http://safari.informit.com).
- Get tips from expert blogs at [informit.com/blogs](http://informit.com/blogs).

Visit [informit.com/learn](http://informit.com/learn) to discover all the ways you can access the hottest technology content.

### Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit [informit.com/socialconnect](http://informit.com/socialconnect).



THE TRUSTED TECHNOLOGY LEARNING SOURCE



Addison-Wesley

Cisco Press

EXAM/CRAM

IBM

Press

QUE'

PRENTICE HALL

SAMS

Safari®

```
$ g++ -o prog1 prog1.cc
C:\Users\me\Programs> cl /EHsc prog1.cpp
```

```

#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "The sum of " << v1 << " and " << v2
        << " is " << v1 + v2 << std::endl;
    return 0;
}
std::cout << "Enter two numbers:" << std::endl;
(std::cout << "Enter two numbers:") << std::endl;
std::cout << "Enter two numbers:";
    std::cout << std::endl;
std::cout << "The sum of " << v1 << " and " << v2
        << " is " << v1 + v2 << std::endl;
    std::cout << "The sum of " << v1;
        << " and " << v2;
        << " is " << v1 + v2 << std::endl;
#include <iostream>
/*
 * Simple main function:
 * Read two numbers and write their sum
 */
int main()
{
    // prompt user to enter two numbers
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0; // variables to hold the input we read
    std::cin >> v1 >> v2; // read input
    std::cout << "The sum of " << v1 << " and " << v2
        << " is " << v1 + v2 << std::endl;
    return 0;
}
/*
 * comment pairs /* */ cannot nest.
 * "cannot nest" is considered source code,
 * as is the rest of the program
 */
int main()
{
    return 0;
}
// /*
// * everything inside a single-line comment is ignored
// * including nested comment pairs
// */
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/ */;
std::cout << /* "*/ /* "*/ */;
```

```

#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // keep executing the while as long as val is less than or equal to 10
    while (val <= 10) {
        sum += val; // assigns sum + val to sum
        ++val;       // add 1 to val
    }
    std::cout << "Sum of 1 to 10 inclusive is "
               << sum << std::endl;
    return 0;
}
// keep executing the while as long as val is less than or equal to 10
while (val <= 10) {
    sum += val; // assigns sum + val to sum
    ++val;       // add 1 to val
}
{
    sum += val; // assigns sum + val to sum
    ++val;       // add 1 to val
}
sum = sum + val; // assign sum + val to sum
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 through 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val; // equivalent to sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
               << sum << std::endl;
    return 0;
}
for (int val = 1; val <= 10; ++val)
    sum += val;
sum += val; // equivalent to sum = sum + val
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
#include <iostream>
int main()
{
    int sum = 0, value = 0;
    // read until end-of-file, calculating a running total of all values read
    while (std::cin >> value)
        sum += value; // equivalent to sum = sum + value
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}

```

```

// error: missing ) in parameter list for main
int main ( {
    // error: used colon, not a semicolon, after endl
    std::cout << "Read each file." << std::endl;
    // error: missing quotes around string literal
    std::cout << Update master. << std::endl;
    // error: second output operator is missing
    std::cout << "Write new master." std::endl;
    // error: missing ; on return statement
    return 0
}
#include <iostream>
int main()
{
    int v1 = 0, v2 = 0;
    std::cin >> v >> v2; // error: uses "v" not "v1"
    // error: cout not defined; should be std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}
#include <iostream>
int main()
{
    // currVal is the number we're counting; we'll read new values into val
    int currVal = 0, val = 0;
    // read first number and ensure that we have data to process
    if (std::cin >> currVal) {
        int cnt = 1; // store the count for the current value we're processing
        while (std::cin >> val) { // read the remaining numbers
            if (val == currVal) // if the values are the same
                ++cnt; // add 1 to cnt
            else { // otherwise, print the count for the previous value
                std::cout << currVal << " occurs "
                    << cnt << " times" << std::endl;
                currVal = val; // remember the new value
                cnt = 1; // reset the counter
            }
        } // while loop ends here
        // remember to print the count for the last value in the file
        std::cout << currVal << " occurs "
            << cnt << " times" << std::endl;
    } // outermost if statement ends here
    return 0;
}
42 42 42 42 42 55 55 62 100 100 100
if (std::cin >> currVal) {
    // ...
} // outermost if statement ends here

```

```
        if (val == currVal)    // if the values are the same
            ++cnt;           // add 1 to cnt
        else { // otherwise, print the count for the previous value
            std::cout << currVal << " occurs "
                << cnt << " times" << std::endl;
            currVal = val;    // remember the new value
            cnt = 1;          // reset the counter
        }
    }
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    // read ISBN, number of copies sold, and sales price
    std::cin >> book;
    // write ISBN, number of copies sold, total revenue, and average price
    std::cout << book << std::endl;
    return 0;
}
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;    // read a pair of transactions
    std::cout << item1 + item2 << std::endl; // print their sum
    return 0;
}
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;
    // first check that item1 and item2 represent the same book
    if (item1.isbn() == item2.isbn()) {
        std::cout << item1 + item2 << std::endl;
        return 0; // indicate success
    } else {
        std::cerr << "Data must refer to same ISBN"
            << std::endl;
        return -1; // indicate failure
    }
}
```

```

#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item total; // variable to hold data for the next transaction
    // read the first transaction and ensure that there are data to process
    if (std::cin >> total) {
        Sales_item trans; // variable to hold the running sum
        // read and process the remaining transactions
        while (std::cin >> trans) {
            // if we're still processing the same book
            if (total.isbn() == trans.isbn())
                total += trans; // update the running total
            else {
                // print results for the previous book
                std::cout << total << std::endl;
                total = trans; // total now refers to the next book
            }
        }
        std::cout << total << std::endl; // print the last transaction
    } else {
        // no input! warn the user
        std::cerr << "No data?!" << std::endl;
        return -1; // indicate failure
    }
    return 0;
}
000110110111000010110010000111011 ...
bool b = 42; // b is true
int i = b; // i has value 1
i = 3.14; // i has value 3
double pi = i; // pi has value 3.0
unsigned char c = -1; // assuming 8-bit chars, c has value 255
signed char c2 = 256; // assuming 8-bit chars, the value of c2 is undefined
int i = 42;
if (i) // condition will evaluate as true
    i = 0;
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl; // prints -84
std::cout << u + i << std::endl; // if 32-bit ints, prints 4294967264
unsigned u1 = 42, u2 = 10;
std::cout << u1 - u2 << std::endl; // ok: result is 32
std::cout << u2 - u1 << std::endl; // ok: but the result will wrap around
for (int i = 10; i >= 0; --i)
    std::cout << i << std::endl;
// WRONG: u can never be less than 0; the condition will always succeed
for (unsigned u = 10; u >= 0; --u)
    std::cout << u << std::endl;

```

```

unsigned u = 11; // start the loop one past the first element we want to print
while (u > 0) {
    --u; // decrement first, so that the last iteration will print 0
    std::cout << u << std::endl;
}
    unsigned u = 10, u2 = 42;
    std::cout << u2 - u << std::endl;
    std::cout << u - u2 << std::endl;
    int i = 10, i2 = 42;
    std::cout << i2 - i << std::endl;
    std::cout << i - i2 << std::endl;
    std::cout << i - u << std::endl;
    std::cout << u - i << std::endl;
20 /* decimal */ 024 /* octal */ 0x14 /* hexadecimal */
3.14159 3.14159E0 0.0e0 .001
    'a' // character literal
    "Hello World!" // string literal
// multiline string literal
std::cout << "a really, really long string literal "
            "that spans two lines" << std::endl;
std::cout << '\n'; // prints a newline
std::cout << "\tHi!\n"; // prints a tab followed by "Hi!" and a newline
    \7 (bell) \12 (newline) \40 (blank)
    \0 (null) \115 ('M') \x4d ('M')
std::cout << "Hi \x4d0\115!\n"; // prints Hi MOM! followed by a newline
std::cout << '\115' << '\n'; // prints M followed by a newline
L'a' // wide character literal, type is wchar_t
u8"hi!" // utf-8 string literal (utf-8 encodes a Unicode character in 8 bits)
42ULL // unsigned integer literal, type is unsigned long long
1E-3F // single-precision floating-point literal, type is float
3.14159L // extended-precision floating-point literal, type is long double
int sum = 0, value, // sum, value, and units_sold have type int
    units_sold = 0; // sum and units_sold have initial value 0
Sales_item item; // item has type Sales_item (see § 1.5.1 (p. 20))
// string is a library type, representing a variable-length sequence of characters
std::string book("0-201-78345-X"); // book initialized from string literal
// ok: price is defined and initialized before it is used to initialize discount
double price = 109.99, discount = price * 0.16;
// ok: call applyDiscount and use the return value to initialize salePrice
double salePrice = applyDiscount(price, discount);
    long double ld = 3.1415926536;
        int a{ld}, b = {ld}; // error: narrowing conversion required
        int c(ld), d = ld; // ok: but value will be truncated
    std::string empty; // empty implicitly initialized to the empty string
    Sales_item item; // default-initialized Sales_item object
        std::string global_str;
        int global_int;
    int main()
    {
        int local_int;
        std::string local_str;
    }

```

```

        extern int i;    // declares but does not define i
        int j;          // declares and defines j
        extern double pi = 3.1416; // definition
        // defines four different int variables
        int somename, someName, SomeName, SOMENAME;
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 through 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val; // equivalent to sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
        << sum << std::endl;
    return 0;
}
#include <iostream>
// Program for illustration purposes only: It is bad style for a function
// to use a global variable and also define a local variable with the same name
int reused = 42; // reused has global scope
int main()
{
    int unique = 0; // unique has block scope
    // output #1: uses global reused; prints 42 0
    std::cout << reused << " " << unique << std::endl;
    int reused = 0; // new, local object named reused hides global reused
    // output #2: uses local reused; prints 0 0
    std::cout << reused << " " << unique << std::endl;
    // output #3: explicitly requests the global reused; prints 42 0
    std::cout << ::reused << " " << unique << std::endl;
    return 0;
}
        int i = 100, sum = 0;
        for (int i = 0; i != 10; ++i)
            sum += i;
        std::cout << i << " " << sum << std::endl;
int ival = 1024;
int &refVal = ival; // refVal refers to (is another name for) ival
int &refVal2; // error: a reference must be initialized
refVal = 2; // assigns 2 to the object to which refVal refers, i.e., to ival
int ii = refVal; // same as ii = ival
// ok: refVal3 is bound to the object to which refVal is bound, i.e., to ival
int &refVal3 = refVal;
// initializes i from the value in the object to which refVal is bound
int i = refVal; // ok: initializes i to the same value as ival
int i = 1024, i2 = 2048; // i and i2 are both ints
int &r = i, r2 = i2; // r is a reference bound to i; r2 is an int
int i3 = 1024, &ri = i3; // i3 is an int; ri is a reference bound to i3
int &r3 = i3, &r4 = i2; // both r3 and r4 are references

```

```

int &refVal4 = 10;    // error: initializer must be an object
double dval = 3.14;
int &refVal5 = dval; // error: initializer must be an int object
    int i = 0, &r1 = i; double d = 0, &r2 = d;
    int i, &ri = i;
    i = 5; ri = 10;
    std::cout << i << " " << ri << std::endl;
int *ip1, *ip2; // both ip1 and ip2 are pointers to int
double dp, *dp2; // dp2 is a pointer to double; dp is a double
int ival = 42;
    int *p = &ival; // p holds the address of ival; p is a pointer to ival
double dval;
double *pd = &dval; // ok: initializer is the address of a double
double *pd2 = pd; // ok: initializer is a pointer to double
int *pi = pd; // error: types of pi and pd differ
pi = &dval; // error: assigning the address of a double to a pointer to int
    int ival = 42;
    int *p = &ival; // p holds the address of ival; p is a pointer to ival
    cout << *p; // * yields the object to which p points; prints 42
    *p = 0; // * yields the object; we assign a new value to ival through p
    cout << *p; // prints 0
int i = 42;
int &r = i; // & follows a type and is part of a declaration; r is a reference
int *p; // * follows a type and is part of a declaration; p is a pointer
p = &i; // & is used in an expression as the address-of operator
*p = i; // * is used in an expression as the dereference operator
int &r2 = *p; // & is part of the declaration; * is the dereference operator
    int *p1 = nullptr; // equivalent to int *p1 = 0;
    int *p2 = 0; // directly initializes p2 from the literal constant 0
// must #include cstdlib
int *p3 = NULL; // equivalent to int *p3 = 0;
    int zero = 0;
    pi = zero; // error: cannot assign an int to a pointer
int i = 42;
    int *pi = 0; // pi is initialized but addresses no object
    int *pi2 = &i; // pi2 initialized to hold the address of i
    int *pi3; // if pi3 is defined inside a block, pi3 is uninitialized
    pi3 = pi2; // pi3 and pi2 address the same object, e.g., i
    pi2 = 0; // pi2 now addresses no object
    pi = &ival; // value in pi is changed; pi now points to ival
    *pi = 0; // value in ival is changed; pi is unchanged
int ival = 1024;
int *pi = 0; // pi is a valid, null pointer
int *pi2 = &ival; // pi2 is a valid pointer that holds the address of ival
if (pi) // pi has value 0, so condition evaluates as false
    // ...
if (pi2) // pi2 points to ival, so it is not 0; the condition evaluates as true
    // ...
    double obj = 3.14, *pd = &obj;
    // ok: void* can hold the address value of any data pointer type
    void *pv = &obj; // obj can be an object of any type
    pv = pd; // pv can hold a pointer to any type
    int i = 42; void *p = &i; long *lp = &i;

```

```

// i is an int; p is a pointer to int; r is a reference to int
int i = 1024, *p = &i, &r = i;
    int* p; // legal but might be misleading
int* p1, p2; // p1 is a pointer to int; p2 is an int
int *p1, *p2; // both p1 and p2 are pointers to int
    int* p1; // p1 is a pointer to int
    int* p2; // p2 is a pointer to int
int ival = 1024;
int *pi = &ival; // pi points to an int
int **ppi = &pi; // ppi points to a pointer to an int
cout << "The value of ival\n"
    << "direct value: " << ival << "\n"
    << "indirect value: " << *pi << "\n"
    << "doubly indirect value: " << **ppi
    << endl;
int i = 42;
int *p; // p is a pointer to int
int *&r = p; // r is a reference to the pointer p
r = &i; // r refers to a pointer; assigning &i to r makes p point to i
*r = 0; // dereferencing r yields i, the object to which p points; changes i to 0
    const int bufSize = 512; // input buffer size
bufSize = 512; // error: attempt to write to const object
const int i = get_size(); // ok: initialized at run time
const int j = 42; // ok: initialized at compile time
const int k; // error: k is uninitialized const
int i = 42;
    const int ci = i; // ok: the value in i is copied into ci
    int j = ci; // ok: the value in ci is copied into j
    const int bufSize = 512; // input buffer size
// file_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();

// file_1.h
extern const int bufSize; // same bufSize as defined in file_1.cc
const int ci = 1024;
const int &r1 = ci; // ok: both reference and underlying object are const
r1 = 42; // error: r1 is a reference to const
int &r2 = ci; // error: nonconst reference to a const object
int i = 42;
const int &r1 = i; // we can bind a const int& to a plain int object
const int &r2 = 42; // ok: r1 is a reference to const
const int &r3 = r1 * 2; // ok: r3 is a reference to const
int &r4 = r * 2; // error: r4 is a plain, nonconst reference
const int temp = dval; // create a temporary const int from the double
const int &ri = temp; // bind ri to that temporary
int i = 42;
int &r1 = i; // r1 bound to i
const int &r2 = i; // r2 also bound to i; but cannot be used to change i
r1 = 0; // r1 is not const; i is now 0
r2 = 0; // error: r2 is a reference to const
const double pi = 3.14; // pi is const; its value may not be changed
double *ptr = &pi; // error: ptr is a plain pointer
const double *cptr = &pi; // ok: cptr may point to a double that is const
*cptr = 42; // error: cannot assign to *cptr

```

```

double dval = 3.14;           // dval is a double; its value can be changed
cptr = &dval;                // ok: but can't change dval through cptr
int errNumb = 0;
int *const curErr = &errNumb; // curErr will always point to errNumb
const double pi = 3.14159;
const double *const pip = &pi; // pip is a const pointer to a const object
*pip = 2.72;                // error: pip is a pointer to const
// if the object to which curErr points (i.e., errNumb) is nonzero
if (*curErr) {
    errorHandler();
    *curErr = 0; // ok: reset the value of the object to which curErr is bound
}
int i = 0;
int *const p1 = &i; // we can't change the value of p1; const is top-level
const int ci = 42; // we cannot change ci; const is top-level
const int *p2 = &ci; // we can change p2; const is low-level
const int *const p3 = p2; // right-most const is top-level, left-most is not
const int &r = ci; // const in reference types is always low-level
i = ci; // ok: copying the value of ci; top-level const in ci is ignored
p2 = p3; // ok: pointed-to type matches; top-level const in p3 is ignored
int *p = p3; // error: p3 has a low-level const but p doesn't
p2 = p3; // ok: p2 has the same low-level const qualification as p3
p2 = &i; // ok: we can convert int* to const int*
int &r = ci; // error: can't bind an ordinary int& to a const int object
const int &r2 = i; // ok: can bind const int& to plain int
const int v2 = 0; int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;
r1 = v2;
p1 = p2; p2 = p1;
p1 = p3; p2 = p3;
const int max_files = 20; // max_files is a constant expression
const int limit = max_files + 1; // limit is a constant expression
int staff_size = 27; // staff_size is not a constant expression
const int sz = get_size(); // sz is not a constant expression
constexpr int mf = 20; // 20 is a constant expression
constexpr int limit = mf + 1; // mf + 1 is a constant expression
constexpr int sz = size(); // ok only if size is a constexpr function
const int *p = nullptr; // p is a pointer to a const int
constexpr int *q = nullptr; // q is a const pointer to int
constexpr int *np = nullptr; // np is a constant pointer to int that is null
int j = 0;
constexpr int i = 42; // type of i is const int
// i and j must be defined outside any function
constexpr const int *p = &i; // p is a constant pointer to the const int i
constexpr int *p1 = &j; // p1 is a constant pointer to the int j
int null = 0, *p = null;
typedef double wages; // wages is a synonym for double
typedef wages base, *p; // base is a synonym for double, p for double*
using SI = Sales_item; // SI is a synonym for Sales_item
wages hourly, weekly; // same as double hourly, weekly;
SI item; // same as Sales_item item

```

```

typedef char *pstring;
const pstring cstr = 0; // cstr is a constant pointer to char
const pstring *ps; // ps is a pointer to a constant pointer to char
const char *cstr = 0; // wrong interpretation of const pstring cstr
// the type of item is deduced from the type of the result of adding val1 and val2
auto item = val1 + val2; // item initialized to the result of val1 + val2
    auto i = 0, *p = &i; // ok: i is int and p is a pointer to int
    auto sz = 0, pi = 3.14; // error: inconsistent types for sz and pi
        int i = 0, &r = i;
            auto a = r; // a is an int (r is an alias for i, which has type int)
const int ci = i, &cr = ci;
auto b = ci; // b is an int (top-level const in ci is dropped)
auto c = cr; // c is an int (cr is an alias for ci whose const is top-level)
auto d = &i; // d is an int*& (&of an int object is int*)
auto e = &ci; // e is const int*& (&of a const object is low-level const)
const auto f = ci; // deduced type of ci is int; f has type const int
    auto &g = ci; // g is a const int& that is bound to ci
    auto &h = 42; // error: we can't bind a plain reference to a literal
    const auto &j = 42; // ok: we can bind a const reference to a literal
auto k = ci, &l = i; // k is int; l is int&
auto &m = ci, *p = &ci; // m is a const int&; p is a pointer to const int
// error: type deduced from i is int; type deduced from &ci is const int
auto &n = i, *p2 = &ci;
    a = 42; b = 42; c = 42;
    d = 42; e = 42; g = 42;
const int i = 42;
auto j = i; const auto &k = i; auto *p = &i;
const auto j2 = i, &k2 = i;
decltype(f()) sum = x; // sum has whatever type f returns
const int ci = 0, &cj = ci;
decltype(ci) x = 0; // x has type const int
decltype(cj) y = x; // y has type const int& and is bound to x
decltype(cj) z; // error: z is a reference and must be initialized
// decltype of an expression can be a reference type
int i = 42, *p = &i, &r = i;
decltype(r + 0) b; // ok: addition yields an int; b is an (uninitialized) int
decltype(*p) c; // error: c is int& and must be initialized
// decltype of a parenthesized variable is always a reference
decltype((i)) d; // error: d is int& and must be initialized
decltype(i) e; // ok: e is an (uninitialized) int
    struct Sales_data {
        std::string bookNo;
        unsigned units_sold = 0;
        double revenue = 0.0;
    };
    struct Sales_data { /* ... */ } accum, trans, *salesptr;
// equivalent, but better way to define these objects
    struct Sales_data { /* ... */ };
    Sales_data accum, trans, *salesptr;
    struct Foo { /* empty */ } // Note: no semicolon
    int main()
    {
        return 0;
    }
}

```

```
#include <iostream>
#include <string>
#include "Sales_data.h"
int main()
{
    Sales_data data1, data2;
    // code to read into data1 and data2
    // code to check whether data1 and data2 have the same ISBN
    // and if so print the sum of data1 and data2
}
double price = 0; // price per book, used to calculate total revenue
// read the first transaction: ISBN, number of books sold, price per book
std::cin >> data1.bookNo >> data1.units_sold >> price;
// calculate total revenue from price and units_sold
data1.revenue = data1.units_sold * price;
std::cin >> data1.bookNo >> data1.units_sold >> price;
// read the second transaction
std::cin >> data2.bookNo >> data2.units_sold >> price;
data2.revenue = data2.units_sold * price;
if (data1.bookNo == data2.bookNo) {
    unsigned totalCnt = data1.units_sold + data2.units_sold;
    double totalRevenue = data1.revenue + data2.revenue;
    // print: ISBN, total sold, total revenue, average price per book
    std::cout << data1.bookNo << " " << totalCnt
        << " " << totalRevenue << " ";
    if (totalCnt != 0)
        std::cout << totalRevenue/totalCnt << std::endl;
    else
        std::cout << "(no sales)" << std::endl;
    return 0; // indicate success
} else { // transactions weren't for the same ISBN
    std::cerr << "Data must refer to the same ISBN"
        << std::endl;
    return -1; // indicate failure
}
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

```

#include <iostream>
// using declaration; when we use the name cin, we get the one from the namespace std
using std::cin;
int main()
{
    int i;
    cin >> i;           // ok: cin is a synonym for std::cin
    cout << i;          // error: no using declaration; we must use the full name
    std::cout << i;    // ok: explicitly use cout from namespace std
    return 0;
}

#include <iostream>
// using declarations for names from the standard library
using std::cin;
using std::cout; using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
        << " is " << v1 + v2 << endl;
    return 0;
}
string s1;           // default initialization; s1 is the empty string
string s2 = s1;      // s2 is a copy of s1
string s3 = "hiya";  // s3 is a copy of the string literal
string s4(10, 'c');  // s4 is cccccccccc
string s5 = "hiya";  // copy initialization
string s6("hiya");  // direct initialization
string s7(10, 'c');  // direct initialization; s7 is cccccccccc
string s8 = string(10, 'c'); // copy initialization; s8 is cccccccccc
                           string temp(10, 'c'); // temp is cccccccccc
                           string s8 = temp;   // copy temp into s8
// Note: #include and using declarations must be added to compile this code
int main()
{
    string s;           // empty string
    cin >> s;          // read a whitespace-separated string into s
    cout << s << endl; // write s to the output
    return 0;
}
string s1, s2;
cin >> s1 >> s2; // read first input into s1, second into s2
cout << s1 << s2 << endl; // write both strings
int main()
{
    string word;
    while (cin >> word)      // read until end-of-file
        cout << word << endl; // write each word followed by a new line
    return 0;
}

```

```

int main()
{
    string line;
    // read input a line at a time until end-of-file
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
// read input a line at a time and discard blank lines
while (getline(cin, line))
    if (!line.empty())
        cout << line << endl;
string line;
// read input a line at a time and print lines that are longer than 80 characters
while (getline(cin, line))
    if (line.size() > 80)
        cout << line << endl;
auto len = line.size(); // len has type string::size_type
string str = "Hello";
string phrase = "Hello World";
string slang = "Hiya";
string st1(10, 'c'), st2; // st1 is cccccccccc; st2 is an empty string
st1 = st2; // assignment: replace contents of st1 with a copy of st2
            // both st1 and st2 are now the empty string
string s1 = "hello, ", s2 = "world\n";
string s3 = s1 + s2; // s3 is hello, world\n
s1 += s2; // equivalent to s1 = s1 + s2
string s1 = "hello", s2 = "world"; // no punctuation in s1 or s2
string s3 = s1 + ", " + s2 + '\n';
string s4 = s1 + ", "; // ok: adding a string and a literal
string s5 = "hello" + ", "; // error: no string operand
string s6 = s1 + ", " + "world"; // ok: each + has a string operand
string s7 = "hello" + ", " + s2; // error: can't add string literals
        string s6 = (s1 + ", ") + "world";
        string tmp = s1 + ", "; // ok: + has a string operand
        s6 = tmp + "world"; // ok: + has a string operand
string s7 = ("hello" + ", ") + s2; // error: can't add string literals
string str("some string");

// print the characters in str one character to a line
for (auto c : str) // for every char in str
    cout << c << endl; // print the current character followed by a newline
string s("Hello World!!!");
// punct_cnt has the same type that s.size returns; see § 2.5.3 (p. 70)
decltype(s.size()) punct_cnt = 0;
// count the number of punctuation characters in s
for (auto c : s) // for every char in s
    if (ispunct(c)) // if the character is punctuation
        ++punct_cnt; // increment the punctuation counter
cout << punct_cnt
    << " punctuation characters in " << s << endl;
3 punctuation characters in Hello World!!!

```

```

string s("Hello World!!!");
// convert s to uppercase
for (auto &c : s) // for every char in s (note: c is a reference)
    c = toupper(c); // c is a reference, so the assignment changes the char in s
cout << s << endl;
c = toupper(c); // c is a reference, so the assignment changes the char in s
if (!s.empty()) // make sure there's a character to print
    cout << s[0] << endl; // print the first character in s
string s("some string");
if (!s.empty()) // make sure there's a character in s[0]
    s[0] = toupper(s[0]); // assign a new value to the first character in s
// process characters in s until we run out of characters or we hit a whitespace
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
    s[index] = toupper(s[index]); // capitalize the current character
const string hexdigits = "0123456789ABCDEF"; // possible hex digits
cout << "Enter a series of numbers between 0 and 15"
    << " separated by spaces. Hit ENTER when finished: "
    << endl;
string result; // will hold the resulting hexify'd string
string::size_type n; // hold numbers from the input
while (cin >> n)
    if (n < hexdigits.size()) // ignore invalid input
        result += hexdigits[n]; // fetch the indicated hex digit
cout << "Your hex number is: " << result << endl;
const string s = "Keep out!";
for (auto &c : s) { /* ... */ }
vector<int> ivec; // ivec holds objects of type int
vector<Sales_item> Sales_vec; // holds Sales_items
vector<vector<string>> file; // vector whose elements are vectors
    vector<string> svec; // default initialization; svec has no elements
vector<int> ivec; // initially empty
// give ivec some values
vector<int> ivec2(ivec); // copy elements of ivec into ivec2
vector<int> ivec3 = ivec; // copy elements of ivec into ivec3
vector<string> svec(ivec2); // error: svec holds strings, not ints
    vector<string> articles = {"a", "an", "the"};
    vector<string> v1{"a", "an", "the"}; // list initialization
    vector<string> v2("a", "an", "the"); // error
vector<int> ivec(10, -1); // ten int elements, each initialized to -1
vector<string> svec(10, "hi!"); // ten strings; each element is "hi!"
    vector<int> ivec(10); // ten elements, each initialized to 0
    vector<string> svec(10); // ten elements, each an empty string
vector<int> vi = 10; // error: must use direct initialization to supply a size
    vector<int> v1(10); // v1 has ten elements with value 0
    vector<int> v2{10}; // v2 has one element with value 10
    vector<int> v3(10, 1); // v3 has ten elements with value 1
    vector<int> v4{10, 1}; // v4 has two elements with values 10 and 1
vector<string> v5{"hi"}; // list initialization: v5 has one element
vector<string> v6("hi"); // error: can't construct a vector from a string literal
vector<string> v7{10}; // v7 has ten default-initialized elements
vector<string> v8{10, "hi"}; // v8 has ten elements with value "hi"

```

```

vector<int> v2;           // empty vector
for (int i = 0; i != 100; ++i)
    v2.push_back(i);     // append sequential integers to v2
// at end of loop v2 has 100 elements, values 0...99
// read words from the standard input and store them as elements in a vector
string word;
vector<string> text;      // empty vector
while (cin >> word) {
    text.push_back(word); // append word to text
}
vector<int> v{1,2,3,4,5,6,7,8,9};
for (auto &i : v)          // for each element in v (note: i is a reference)
    i *= i;                // square the element value
for (auto i : v)          // for each element in v
    cout << i << " "; // print the element
cout << endl;
    vector<int>::size_type // ok
    vector::size_type      // error
// count the number of grades by clusters of ten: 0--9, 10--19, ... 90--99, 100
vector<unsigned> scores(11, 0); // 11 buckets, all initially 0
unsigned grade;
while (cin >> grade) {      // read the grades
    if (grade <= 100)        // handle only valid grades
        ++scores[grade/10]; // increment the counter for the current cluster
}
++scores[grade/10]; // increment the counter for the current cluster
auto ind = grade/10; // get the bucket index
scores[ind] = scores[ind] + 1; // increment the count
vector<int> ivec; // empty vector
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec[ix] = ix; // disaster: ivec has no elements
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec.push_back(ix); // ok: adds a new element with value ix
vector<int> ivec; // empty vector
cout << ivec[0]; // error: ivec has no elements!

vector<int> ivec2(10); // vector with ten elements
cout << ivec2[10]; // error: ivec2 has elements 0...9
// the compiler determines the type of b and e; see § 2.5.2 (p. 68)
// b denotes the first element and e denotes one past the last element in v
auto b = v.begin(), e = v.end(); // b and e have the same type
string s("some string");
if (s.begin() != s.end()) { // make sure s is not empty
    auto it = s.begin(); // it denotes the first character in s
    *it = toupper(*it); // make that character uppercase
}
// process characters in s until we run out of characters or we hit a whitespace
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); // capitalize the current character
vector<int>::iterator it; // it can read and write vector<int> elements
string::iterator it2; // it2 can read and write characters in a string
vector<int>::const_iterator it3; // it3 can read but not write elements
string::const_iterator it4; // it4 can read but not write characters

```

```

vector<int> v;
const vector<int> cv;
auto it1 = v.begin(); // it1 has type vector<int>::iterator
auto it2 = cv.begin(); // it2 has type vector<int>::const_iterator
auto it3 = v.cbegin(); // it3 has type vector<int>::const_iterator
(*it).empty() // dereferences it and calls the member empty on the resulting object
*it.empty() // error: attempts to fetch the member named empty from it
                // but it is an iterator and has no member named empty
// print each line in text up to the first blank line
for (auto it = text.cbegin();
     it != text.cend() && !it->empty(); ++it)
    cout << *it << endl;
// compute an iterator to the element closest to the midpoint of vi
auto mid = vi.begin() + vi.size() / 2;
if (it < mid)
    // process elements in the first half of vi
// text must be sorted
// beg and end will denote the range we're searching
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg)/2; // original midpoint
// while there are still elements to look at and we haven't yet found sought
while (mid != end && *mid != sought) {
    if (sought < *mid) // is the element we want in the first half?
        end = mid; // if so, adjust the range to ignore the second half
    else // the element we want is in the second half
        beg = mid + 1; // start looking with the element just after mid
    mid = beg + (end - beg)/2; // new midpoint
}
unsigned cnt = 42; // not a constant expression
constexpr unsigned sz = 42; // constant expression
                                // constexpr see § 2.4.4 (p. 66)
int arr[10]; // array of ten ints
int *parr[sz]; // array of 42 pointers to int
string bad[cnt]; // error: cnt is not a constant expression
string strs[get_size()]; // ok if get_size is constexpr, error otherwise
const unsigned sz = 3;
int ia1[sz] = {0,1,2}; // array of three ints with values 0, 1, 2
int a2[] = {0, 1, 2}; // an array of dimension 3
int a3[5] = {0, 1, 2}; // equivalent to a3 [] = {0, 1, 2, 0, 0}
string a4[3] = {"hi", "bye"}; // same as a4 [] = {"hi", "bye", ""}
int a5[2] = {0,1,2}; // error: too many initializers
char a1[] = {'C', '+', '+'}; // list initialization, no null
char a2[] = {'C', '+', '+', '\0'}; // list initialization, explicit null
char a3[] = "C++"; // null terminator added automatically
const char a4[6] = "Daniel"; // error: no space for the null!
int a[] = {0, 1, 2}; // array of three ints
int a2[] = a; // error: cannot initialize one array with another
a2 = a; // error: cannot assign one array to another
int *ptrs[10]; // ptrs is an array of ten pointers to int
int &refs[10] = /* ? */; // error: no arrays of references
int (*Parray)[10] = &arr; // Parray points to an array of ten ints
int (&arrRef)[10] = arr; // arrRef refers to an array of ten ints
int *(&arry)[10] = ptrs; // arry is a reference to an array of ten pointers

```

```

// count the number of grades by clusters of ten: 0--9, 10--19, ... 90--99, 100
unsigned scores[11] = {}; // 11 buckets, all value initialized to 0
unsigned grade;
while (cin >> grade) {
    if (grade <= 100)
        ++scores[grade/10]; // increment the counter for the current cluster
}
for (auto i : scores) // for each counter in scores
    cout << i << " "; // print the value of that counter
cout << endl;
constexpr size_t array_size = 10;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
    ia[ix] = ix;
string nums[] = {"one", "two", "three"}; // array of strings
string *p = &nums[0]; // p points to the first element in nums
string *p2 = nums; // equivalent to p2 = &nums[0]
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
auto ia2(ia); // ia2 is an int* that points to the first element in ia
ia2 = 42; // error: ia2 is a pointer, and we can't assign an int to a pointer
auto ia2(&ia[0]); // now it's clear that ia2 has type int*
// ia3 is an array of ten ints
decltype(ia) ia3 = {0,1,2,3,4,5,6,7,8,9};
ia3 = p; // error: can't assign an int* to an array
ia3[4] = i; // ok: assigns the value of i to an element in ia3
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; // p points to the first element in arr
++p; // p points to arr[1]
int *e = &arr[10]; // pointer just past the last element in arr
for (int *b = arr; b != e; ++b)
    cout << *b << endl; // print the elements in arr
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
int *beg = begin(ia); // pointer to the first element in ia
int *last = end(ia); // pointer one past the last element in ia
// pbeg points to the first and pend points just past the last element in arr
int *pbeg = begin(arr), *pend = end(arr);
// find the first negative element, stopping if we've seen all the elements
while (pbeg != pend && *pbeg >= 0)
    ++pbeg;
constexpr size_t sz = 5;
int arr[sz] = {1,2,3,4,5};
int *ip = arr; // equivalent to int *ip = &arr[0]
int *ip2 = ip + 4; // ip2 points to arr[4], the last element in arr
// ok: arr is converted to a pointer to its first element; p points one past the end of arr
int *p = arr + sz; // use caution -- do not dereference!
int *p2 = arr + 10; // error: arr has only 5 elements; p2 has undefined value
auto n = end(arr) - begin(arr); // n is 5, the number of elements in arr
int *b = arr, *e = arr + sz;
while (b < e) {
    // use *b
    ++b;
}

```

```

int i = 0, sz = 42;
int *p = &i, *e = &sz;
// undefined: p and e are unrelated; comparison is meaningless!
while (p < e)
int ia[] = {0,2,4,6,8}; // array with 5 elements of type int
int last = *(ia + 4); // ok: initializes last to 8, the value of ia[4]
last = *ia + 4; // ok: last = 4, equivalent to ia[0] + 4
int ia[] = {0,2,4,6,8}; // array with 5 elements of type int
int i = ia[2]; // ia is converted to a pointer to the first element in ia
                // ia[2] fetches the element to which (ia + 2) points
int *p = ia; // p points to the first element in ia
i = *(p + 2); // equivalent to i = ia[2]
int *p = &ia[2]; // p points to the element indexed by 2
int j = p[1]; // p[1] is equivalent to *(p + 1),
                // p[1] is the same element as ia[3]
int k = p[-2]; // p[-2] is the same element as ia[0]
char ca[] = {'C', '+', '+'}; // not null terminated
cout << strlen(ca) << endl; // disaster: ca isn't null terminated
string s1 = "A string example";
string s2 = "A different string";
if (s1 < s2) // false: s2 is less than s1
const char ca1[] = "A string example";
const char ca2[] = "A different string";
if (ca1 < ca2) // undefined: compares two unrelated addresses
if (strcmp(ca1, ca2) < 0) // same effect as string comparison s1 < s2
    // initialize largeStr as a concatenation of s1, a space, and s2
    string largeStr = s1 + " " + s2;
// disastrous if we miscalculated the size of largeStr
strcpy(largeStr, ca1); // copies ca1 into largeStr
strcat(largeStr, " "); // adds a space at the end of largeStr
strcat(largeStr, ca2); // concatenates ca2 onto largeStr
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
string s("Hello World"); // s holds Hello World
char *str = s; // error: can't initialize a char* from a string
const char *str = s.c_str(); // ok
int int_arr[] = {0, 1, 2, 3, 4, 5};
// ivec has six elements; each is a copy of the corresponding element in int_arr
vector<int> ivec(begin(int_arr), end(int_arr));
// copies three elements: int_arr[1], int_arr[2], int_arr[3]
vector<int> subVec(int_arr + 1, int_arr + 4);
int ia[3][4]; // array of size 3; each element is an array of ints of size 4
// array of size 10; each element is a 20-element array whose elements are arrays of 30 ints
int arr[10][20][30] = {0}; // initialize all elements to 0
int ia[3][4] = { // three elements; each element is an array of size 4
    {0, 1, 2, 3}, // initializers for the row indexed by 0
    {4, 5, 6, 7}, // initializers for the row indexed by 1
    {8, 9, 10, 11} // initializers for the row indexed by 2
};

```

```

// equivalent initialization without the optional nested braces for each row
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
    // explicitly initialize only element 0 in each row
    int ia[3][4] = {{ 0 }, { 4 }, { 8 }};
    // explicitly initialize row 0; the remaining elements are value initialized
    int ix[3][4] = {0, 3, 6, 9};
// assigns the first element of arr to the last element in the last row of ia
ia[2][3] = arr[0][0][0];

int (&row)[4] = ia[1]; // binds row to the second four-element array in ia
constexpr size_t rowCnt = 3, colCnt = 4;
    int ia[rowCnt][colCnt]; // 12 uninitialized elements
    // for each row
    for (size_t i = 0; i != rowCnt; ++i) {
        // for each column within the row
        for (size_t j = 0; j != colCnt; ++j) {
            // assign the element's positional index as its value
            ia[i][j] = i * colCnt + j;
        }
    }
    size_t cnt = 0;
    for (auto &row : ia) // for every element in the outer array
        for (auto &col : row) { // for every element in the inner array
            col = cnt; // give this element the next value
            ++cnt; // increment cnt
        }
    for (const auto &row : ia) // for every element in the outer array
        for (auto col : row) // for every element in the inner array
            cout << col << endl;
int ia[3][4]; // array of size 3; each element is an array of ints of size 4
int (*p)[4] = ia; // p points to an array of four ints
p = &ia[2]; // p now points to the last element in ia
    int *ip[4]; // array of pointers to int
    int (*ip)[4]; // pointer to an array of four ints
// print the value of each element in ia, with each inner array on its own line
// p points to an array of four ints
for (auto p = ia; p != ia + 3; ++p) {
    // q points to the first element of an array of four ints; that is, q points to an int
    for (auto q = *p; q != *p + 4; ++q)
        cout << *q << ' ';
    cout << endl;
}
// p points to the first array in ia
for (auto p = begin(ia); p != end(ia); ++p) {
    // q points to the first element in an inner array
    for (auto q = begin(*p); q != end(*p); ++q)
        cout << *q << ' '; // prints the int value to which q points
    cout << endl;
}

```

```

using int_array = int[4]; // new style type alias declaration; see § 2.5.1 (p. 68)
typedef int int_array[4]; // equivalent typedef declaration; § 2.5.1 (p. 67)
// print the value of each element in ia, with each inner array on its own line
for (int_array *p = ia; p != ia + 3; ++p) {
    for (int *q = *p; q != *p + 4; ++q)
        cout << *q << ' ';
    cout << endl;
}
// parentheses in this expression match default precedence and associativity
((6 + ((3 * 4) / 2)) + 2)
// parentheses result in alternative groupings
cout << (6 + 3) * (4 / 2 + 2) << endl; // prints 36
cout << ((6 + 3) * 4) / 2 + 2 << endl; // prints 20
cout << 6 + 3 * 4 / (2 + 2) << endl; // prints 9
int ia[] = {0,2,4,6,8}; // array with five elements of type int
int last = *(ia + 4); // initializes last to 8, the value of ia[4]
last = *ia + 4; // last = 4, equivalent to ia[0] + 4
    cin >> v1 >> v2; // read into v1 and then into v2
    int i = 0;
    cout << i << " " << ++i << endl; // undefined
short short_value = 32767; // max value if shorts are 16 bits
short_value += 1; // this calculation overflows
cout << "short_value: " << short_value << endl;
int ival1 = 21/6; // ival1 is 3; result is truncated; remainder is discarded
int ival2 = 21/7; // ival2 is 3; no remainder; result is an integral value
    int ival = 42;
    double dval = 3.14;
        ival % 12; // ok: result is 6
        ival % dval; // error: floating-point operand
    21 % 6; /* result is 3 */ 21 / 6; /* result is 3 */
    21 % 7; /* result is 0 */ 21 / 7; /* result is 3 */
    -21 % -8; /* result is -5 */ -21 / -8; /* result is 2 */
    21 % -5; /* result is 1 */ 21 / -5; /* result is -4 */
        12 / 3 * 4 + 5 * 15 + 24 % 4 / 2
    index != s.size() && !isspace(s[index])
// note s as a reference to const; the elements aren't copied and can't be changed
for (const auto &s : text) { // for each element in text
    cout << s; // print the current element
    // blank lines and those that end with a period get a newline
    if (s.empty() || s[s.size() - 1] == '.')
        cout << endl;
    else
        cout << " "; // otherwise just separate with a space
}
// print the first element in vec if there is one
if (!vec.empty())
    cout << vec[0];
// oops! this condition compares k to the bool result of i < j
if (i < j < k) // true if k is greater than 1!
// ok: condition is true if i is smaller than j and j is smaller than k
if (i < j && j < k) { /* ... */ }
if (val) { /* ... */ } // true if val is any nonzero value
if (!val) { /* ... */ } // true if val is zero
if (val == true) { /* ... */ } // true only if val is equal to 1!

```

```

        const char *cp = "Hello World";
        if (cp && *cp)
int i = 0, j = 0, k = 0; // initializations, not assignment
const int ci = i;           // initialization, not assignment
1024 = k;                 // error: literals are rvalues
i + j = k;                // error: arithmetic expressions are rvalues
ci = k;                   // error: ci is a const (nonmodifiable) lvalue
    k = 0;                  // result: type int, value 0
    k = 3.14159;            // result: type int, value 3
k = {3.14};                // error: narrowing conversion
vector<int> vi;           // initially empty
vi = {0,1,2,3,4,5,6,7,8,9}; // vi now has ten elements, values 0 through 9
    int ival, jval;
    ival = jval = 0; // ok: each assigned 0
int ival, *pval; // ival is an int; pval is a pointer to int
    ival = pval = 0; // error: cannot assign the value of a pointer to an int
string s1, s2;
s1 = s2 = "OK"; // string literal "OK" converted to string
// a verbose and therefore more error-prone way to write this loop
    int i = get_value(); // get the first value
    while (i != 42) {
        // do something...
        i = get_value(); // get remaining values
    }
int i;
// a better way to write our loop--what the condition does is now clearer
while ((i = get_value()) != 42) {
    // do something...
}
    int sum = 0;
// sum values from 1 through 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val; // equivalent to sum = sum + val
+= -= *= /= %= // arithmetic operators
<<= >>= &= ^= |= // bitwise operators; see § 4.8 (p. 152)
    double dval; int ival; int *pi;
    dval = ival = pi = 0;
int i = 0, j;
j = ++i; // j = 1, i = 1: prefix yields the incremented value
j = i++; // j = 1, i = 2: postfix yields the unincremented value
auto pbeg = v.begin();
// print elements up to the first negative value
while (pbeg != v.end() && *beg >= 0)
    cout << *pbeg++ << endl; // print the current value and advance pbeg
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); // capitalize the current character
// the behavior of the following loop is undefined!
while (beg != s.end() && !isspace(*beg))
    *beg = toupper(*beg++); // error: this assignment is undefined
*beg = toupper(*beg);           // execution if left-hand side is evaluated first
*(beg + 1) = toupper(*beg);   // execution if right-hand side is evaluated first

```

```

        string s1 = "a string", *p = &s1;
        auto n = s1.size(); // run the size member of the string s1
        n = (*p).size(); // run size on the object to which p points
        n = p->size(); // equivalent to (*p).size()
        // run the size member of p, then dereference the result!
        *p.size(); // error: p is a pointer and has no member named size
        string finalgrade = (grade < 60) ? "fail" : "pass";
        finalgrade = (grade > 90) ? "high pass"
                                : (grade < 60) ? "fail" : "pass";
        cout << ((grade < 60) ? "fail" : "pass"); // prints pass or fail
        cout << (grade < 60) ? "fail" : "pass"; // prints 1 or 0!
        cout << grade < 60 ? "fail" : "pass"; // error: compares cout to 60
        cout << (grade < 60); // prints 1 or 0
        cout ? "fail" : "pass"; // test cout and then yield one of the two literals
                                // depending on whether cout is true or false
        cout << grade; // less-than has lower precedence than shift, so print grade first
        cout < 60 ? "fail" : "pass"; // then compare cout to 60!
        string s = "word";
        string pl = s + s[s.size() - 1] == 's' ? "" : "s";
        unsigned long quiz1 = 0; // we'll use this value as a collection of bits
                                1UL << 27 // generate a value with only bit number 27 set
        quiz1 |= 1UL << 27; // indicate student number 27 passed
        quiz1 = quiz1 | 1UL << 27; // equivalent to quiz1 |= 1UL << 27;
        quiz1 &= ~ (1UL << 27); // student number 27 failed
        bool status = quiz1 & (1UL << 27); // how did student number 27 do?
        cout << "hi" << " there" << endl;
        ( (cout << "hi") << " there" ) << endl;
        cout << 42 + 10; // ok: + has higher precedence, so the sum is printed
        cout << (10 < 42); // ok: parentheses force intended grouping; prints 1
        cout << 10 < 42; // error: attempt to compare cout to 42!
Sales_data data, *p;
sizeof(Sales_data); // size required to hold an object of type Sales_data
sizeof data; // size of data's type, i.e., sizeof(Sales_data)
sizeof p; // size of a pointer
sizeof *p; // size of the type to which p points, i.e., sizeof(Sales_data)
sizeof data.revenue; // size of the type of Sales_data's revenue member
sizeof Sales_data::revenue; // alternative way to get the size of revenue
// sizeof(ia)/sizeof(*ia) returns the number of elements in ia
constexpr size_t sz = sizeof(ia)/sizeof(*ia);
int arr2[sz]; // ok sizeof returns a constant expression § 2.4.4 (p. 65)
                int x[10]; int *p = x;
                cout << sizeof(x)/sizeof(*x) << endl;
                cout << sizeof(p)/sizeof(*p) << endl;
vector<int>::size_type cnt = ivec.size();
// assign values from size...1 to the elements in ivec
for(vector<int>::size_type ix = 0;
    ix != ivec.size(); ++ix, --cnt)
    ivec[ix] = cnt;
constexpr int size = 5;
int ia[size] = {1,2,3,4,5};
for (int *ptr = ia, ix = 0;
     ix != size && ptr != ia+size;
     ++ix, ++ptr) { /* ... */ }
someValue ? ++x, ++y : --x, --y
int ival = 3.541 + 3; // the compiler might warn about loss of precision

```

```

bool      flag;          char           cval;
short     sval;          unsigned short usval;
int       ival;          unsigned int   uival;
long      lval;          unsigned long  ulval;
float     fval;          double         dval;

3.14159L + 'a'; // 'a' promoted to int, then that int converted to long double
dval + ival;    // ival converted to double
dval + fval;    // fval converted to double
ival = dval;    // dval converted (by truncation) to int
flag = dval;    // if dval is 0, then flag is false, otherwise true
cval + fval;    // cval promoted to int, then that int converted to float
sval + cval;    // sval and cval promoted to int
cval + lval;    // cval converted to long
ival + ulval;  // ival converted to unsigned long
usval + ival;  // promotion depends on the size of unsigned short and int
uival + lval;  // conversion depends on the size of unsigned int and long
char cval;      int ival;      unsigned int ui;
float fval;     double dval;
int ia[10];     // array of ten ints
int* ip = ia;  // convert ia to a pointer to the first element
char *cp = get_string();
if (cp) /* ... */ // true if the pointer cp is not zero
while (*cp) /* ... */ // true if *cp is not the null character
int i;
const int &j = i; // convert a nonconst to a reference to const int
const int *p = &i; // convert address of a nonconst to the address of a const
int &r = j, *q = p; // error: conversion from const to nonconst not allowed
string s, t = "a value"; // character string literal converted to type string
while (cin >> s) // while condition converts cin to bool
// cast used to force floating-point division
double slope = static_cast<double>(j) / i;
void* p = &d; // ok: address of any nonconst object can be stored in a void*
// ok: converts void* back to the original pointer type
double *dp = static_cast<double*>(p);
const char *pc;
char *p = const_cast<char*>(pc); // ok: but writing through p is undefined
const char *cp;
// error: static_cast can't cast away const
char *q = static_cast<char*>(cp);
static_cast<string>(cp); // ok: converts string literal to string
const_cast<string>(cp); // error: const_cast only changes constness
int *ip;
char *pc = reinterpret_cast<char*>(ip);
type (expr); // function-style cast notation
(type) expr; // C-language-style cast notation
char *pc = (char*) ip; // ip is a pointer to int
int i; double d; const string *ps; char *pc; void *pv;
double slope = static_cast<double>(j/i);
ival + 5; // rather useless expression statement
cout << ival; // useful expression statement
// read until we hit end-of-file or find an input equal to sought
while (cin >> s && s != sought)
; // null statement
ival = v1 + v2;; // ok: second semicolon is a superfluous null statement

```

```

// disaster: extra semicolon: loop body is this null statement
while (iter != svec.end()) ; // the while body is the empty statement
    ++iter; // increment is not part of the loop
        while (val <= 10) {
            sum += val; // assigns sum + val to sum
            ++val; // add 1 to val
        }
        while (cin >> s && s != sought)
            {} // empty block
while (int i = get_num()) // i is created and initialized on each iteration
    cout << i << endl;
i = 0; // error: i is not accessible outside the loop
// find the first negative element
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // we know that all elements in v are greater than or equal to zero
vector<string> scores = {"F", "D", "C", "B", "A", "A++"};
// if grade is less than 60 it's an F, otherwise compute a subscript
string lettergrade;
if (grade < 60)
    lettergrade = scores[0];
else
    lettergrade = scores[(grade - 50)/10];
if (grade % 10 > 7)
    lettergrade += '+'; // grades ending in 8 or 9 get a +
else if (grade % 10 < 3)
    lettergrade += '-'; // those ending in 0, 1, or 2 get a -
// if failing grade, no need to check for a plus or minus
if (grade < 60)
    lettergrade = scores[0];
else {
    lettergrade = scores[(grade - 50)/10]; // fetch the letter grade
    if (grade != 100) // add plus or minus only if not already an A++
        if (grade % 10 > 7)
            lettergrade += '+'; // grades ending in 8 or 9 get a +
        else if (grade % 10 < 3)
            lettergrade += '-'; // grades ending in 0, 1, or 2 get a -
}
if (grade < 60)
    lettergrade = scores[0];
else // WRONG: missing curly
    lettergrade = scores[(grade - 50)/10];
// despite appearances, without the curly brace, this code is always executed
// failing grades will incorrectly get a - or a +
if (grade != 100)
    if (grade % 10 > 7)
        lettergrade += '+'; // grades ending in 8 or 9 get a +
    else if (grade % 10 < 3)
        lettergrade += '-'; // grades ending in 0, 1, or 2 get a -

```

```

// WRONG: execution does NOT match indentation; the else goes with the inner if
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // grades ending in 8 or 9 get a +
else
    lettergrade += '-'; // grades ending in 3, 4, 5, 6 will get a minus!
// indentation matches the execution path, not the programmer's intent
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // grades ending in 8 or 9 get a +
else
    lettergrade += '-'; // grades ending in 3, 4, 5, 6 will get a minus!
// add a plus for grades that end in 8 or 9 and a minus for those ending in 0, 1, or 2
if (grade % 10 >= 3) {
    if (grade % 10 > 7)
        lettergrade += '+'; // grades ending in 8 or 9 get a +
} else // curly braces force the else to go with the outer if
    lettergrade += '-'; // grades ending in 0, 1, or 2 will get a minus
    if (int ival = get_value())
        cout << "ival = " << ival << endl;
    if (!ival)
        cout << "ival = 0\n";
// initialize counters for each vowel
unsigned aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
char ch;
while (cin >> ch) {
    // if ch is a vowel, increment the appropriate counter
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
// print results
cout << "Number of vowel a: \t" << aCnt << '\n'
    << "Number of vowel e: \t" << eCnt << '\n'
    << "Number of vowel i: \t" << iCnt << '\n'
    << "Number of vowel o: \t" << oCnt << '\n'
    << "Number of vowel u: \t" << uCnt << endl;

```

```
char ch = getVal();
int ival = 42;
switch(ch) {
    case 3.14: // error: noninteger as case label
    case ival: // error: nonconstant as case label
    // ...
    unsigned vowelCnt = 0;
    // ...
    switch (ch)
    {
        // any occurrence of a, e, i, o, or u increments vowelCnt
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            ++vowelCnt;
            break;
    }
    switch (ch)
    {
        // alternative legal syntax
        case 'a': case 'e': case 'i': case 'o': case 'u':
            ++vowelCnt;
            break;
    }
    // warning: deliberately incorrect!
    switch (ch) {
        case 'a':
            ++aCnt; // oops: should have a break statement
        case 'e':
            ++eCnt; // oops: should have a break statement
        case 'i':
            ++iCnt; // oops: should have a break statement
        case 'o':
            ++oCnt; // oops: should have a break statement
        case 'u':
            ++uCnt;
    }
    // if ch is a vowel, increment the appropriate counter
    switch (ch) {
        case 'a': case 'e': case 'i': case 'o': case 'u':
            ++vowelCnt;
            break;
        default:
            ++otherCnt;
            break;
    }
}
```

```
case true:
    // this switch statement is illegal because these initializations might be bypassed
    string file_name; // error: control bypasses an implicitly initialized variable
    int ival = 0;      // error: control bypasses an explicitly initialized variable
    int jval;          // ok: because jval is not initialized
    break;
case false:
    // ok: jval is in scope but is uninitialized
    jval = next_num(); // ok: assign a value to jval
    if (file_name.empty()) // file_name is in scope but wasn't initialized
        // ...
case true:
{
    // ok: declaration statement within a statement block
    string file_name = get_file_name();
    // ...
}
break;
case false:
    if (file_name.empty()) // error: file_name is not in scope
        unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
        char ch = next_text();
        switch (ch) {
            case 'a': aCnt++;
            case 'e': eCnt++;
            default: iouCnt++;
        }

        unsigned index = some_value();
        switch (index) {
            case 1:
                int ix = get_value();
                ivec[ ix ] = index;
                break;
            default:
                ix = ivec.size()-1;
                ivec[ ix ] = index;
        }
    unsigned evenCnt = 0, oddCnt = 0;
    int digit = get_num() % 10;
    switch (digit) {
        case 1, 3, 5, 7, 9:
            oddcnt++;
            break;
        case 2, 4, 6, 8, 10:
            evencnt++;
            break;
    }
}
```

```

        unsigned ival=512, jval=1024, kval=4096;
        unsigned bufsize;
        unsigned swt = get_bufCnt();
        switch(swt) {
            case ival:
                bufsize = ival * sizeof(int);
                break;
            case jval:
                bufsize = jval * sizeof(int);
                break;
            case kval:
                bufsize = kval * sizeof(int);
                break;
        }
    vector<int> v;
    int i;
    // read until end-of-file or other input failure
    while (cin >> i)
        v.push_back(i);
    // find the first negative element
    auto beg = v.begin();
    while (beg != v.end() && *beg >= 0)
        ++beg;
    if (beg == v.end())
        // we know that all elements in v are greater than or equal to zero
        for (init-statement condition; expression)
            statement
            for (initializer; condition; expression)
                statement
    // process characters in s until we run out of characters or we hit a whitespace
    for (decltype(s.size()) index = 0;
        index != s.size() && !isspace(s[index]); ++index)
        s[index] = toupper(s[index]); // capitalize the current character
    // remember the size of v and stop when we get to the original last element
    for (decltype(v.size()) i = 0, sz = v.size(); i != sz; ++i)
        v.push_back(v[i]);
    auto beg = v.begin();
    for ( /* null */; beg != v.end() && *beg >= 0; ++beg)
        ; // no work to do
        for (int i = 0; /* no condition */; ++i) {
            // process i; code inside the loop must stop the iteration!
        }
    vector<int> v;
    for (int i; cin >> i; /* no expression */)
        v.push_back(i);
    for (int ix = 0; ix != sz; ++ix) { /* ... */}
    if (ix != sz)
        // ...
        int ix;
        for (ix != sz; ++ix) { /* ... */}

```

```
vector<int> v = {0,1,2,3,4,5,6,7,8,9};  
    // range variable must be a reference so we can write to the elements  
    for (auto &r : v)      // for each element in v  
        r *= 2;             // double the value of each element in v  
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {  
    auto &r = *beg; // x must be a reference so we can change the element  
    r *= 2;           // double the value of each element in v  
}  
    // repeatedly ask the user for a pair of numbers to sum  
string rsp; // used in the condition; can't be defined inside the do  
do {  
    cout << "please enter two values: ";  
    int val1 = 0, val2 = 0;  
    cin >> val1 >> val2;  
    cout << "The sum of " << val1 << " and " << val2  
        << " = " << val1 + val2 << "\n\n";  
    cout << "More? Enter yes or no: ";  
    cin >> rsp;  
} while (!rsp.empty() && rsp[0] != 'n');  
do {  
    // ...  
    mumble(foo);  
} while (int foo = get_foo()); // error: declaration in a do condition  
do {  
    int v1, v2;  
    cout << "Please enter two numbers to sum: " ;  
    if (cin >> v1 >> v2)  
        cout << "Sum is: " << v1 + v2 << endl;  
    while (cin);  
    do {  
        // ...  
    } while (int ival = get_response());  
    do {  
        int ival = get_response();  
    } while (ival);
```

```

string buf;
while (cin >> buf && !buf.empty()) {
    switch(buf[0]) {
    case '-':
        // process up to the first blank
        for (auto it = buf.begin() + 1; it != buf.end(); ++it) {
            if (*it == ' ')
                break; // #1, leaves the for loop
        }
    }
    // break #1 transfers control here
    // remaining '-' processing:
    break; // #2, leaves the switch statement

    case '+':
        // ...
    }
}
// end of switch: break #2 transfers control here
} // end while
string buf;
while (cin >> buf && !buf.empty()) {
    if (buf[0] != '_')
        continue; // get another input
    // still here? the input starts with an underscore; process buf ...
}
end: return; // labeled statement; may be the target of a goto
// ...
goto end;

int ix = 10; // error: goto bypasses an initialized variable definition
end:
// error: code here could use ix but the goto bypassed its declaration
ix = 42;
// backward jump over an initialized variable definition is okay
begin:
    int sz = get_size();
    if (sz <= 0) {
        goto begin;
    }
Sales_item item1, item2;
cin >> item1 >> item2;
// first check that item1 and item2 represent the same book
if (item1.isbn() == item2.isbn()) {
    cout << item1 + item2 << endl;
    return 0; // indicate success
} else {
    cerr << "Data must refer to same ISBN"
        << endl;
    return -1; // indicate failure
}

```

```

// first check that the data are for the same item
if (item1.isbn() != item2.isbn())
    throw runtime_error("Data must refer to same ISBN");
// if we're still here, the ISBNs are the same
cout << item1 + item2 << endl;
try {
    program-statements
} catch (exception-declaration) {
    handler-statements
} catch (exception-declaration) {
    handler-statements
} // ...
while (cin >> item1 >> item2) {
    try {
        // execute code that will add the two Sales_items
        // if the addition fails, the code throws a runtime_error exception
    } catch (runtime_error err) {
        // remind the user that the ISBNs must match and prompt for another pair
        cout << err.what()
            << "\nTry Again? Enter y or n" << endl;
        char c;
        cin >> c;
        if (!cin || c == 'n')
            break; // break out of the while loop
    }
}
// factorial of val is val * (val - 1) * (val - 2) ... * ((val - (val - 1)) * 1)
int fact(int val)
{
    int ret = 1; // local variable to hold the result as we calculate it
    while (val > 1)
        ret *= val--;
    return ret; // return the result
}
int main()
{
    int j = fact(5); // j equals 120, i.e., the result of fact(5)
    cout << "5! is " << j << endl;
    return 0;
}
int val = 5; // initialize val from the literal 5
int ret = 1; // code from the body of fact
while (val > 1)
    ret *= val--;
int j = ret; // initialize j as a copy of ret
fact("hello"); // error: wrong argument type
fact(); // error: too few arguments
fact(42, 10, 0); // error: too many arguments
fact(3.14); // ok: argument is converted to int
void f1(){ /* ... */ } // implicit void parameter list
void f2(void){ /* ... */ } // explicit void parameter list

```

```

        int f3(int v1, v2) { /* ... */ }      // error
        int f4(int v1, int v2) { /* ... */ } // ok
size_t count_calls()
{
    static size_t ctr = 0; // value will persist across calls
    return ++ctr;
}
int main()
{
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;
    return 0;
}
// parameter names chosen to indicate that the iterators denote a range of values to print
void print(vector<int>::const_iterator beg,
           vector<int>::const_iterator end);
$ CC factMain.cc fact.cc    # generates factMain.exe or a.out
$ CC factMain.cc fact.cc -o main # generates main or main.exe
$ CC -c factMain.cc          # generates factMain.o
$ CC -c fact.cc              # generates fact.o
$ CC factMain.o fact.o     # generates factMain.exe or a.out
$ CC factMain.o fact.o -o main # generates main or main.exe
    int n = 0;                // ordinary variable of type int
    int i = n;                // i is a copy of the value in n
    i = 42;                  // value in i is changed; n is unchanged
    ret *= val--; // decrements the value of val
int n = 0, i = 42;
int *p = &n, *q = &i; // p points to n; q points to i
*p = 42;           // value in n is changed; p is unchanged
p = q;             // p now points to i; values in i and n are unchanged
// function that takes a pointer and sets the pointed-to value to zero
void reset(int *ip)
{
    *ip = 0; // changes the value of the object to which ip points
    ip = 0; // changes only the local copy of ip; the argument is unchanged
}
int i = 42;
reset(&i);           // changes i but not the address of i
cout << "i = " << i << endl; // prints i = 0
int n = 0, i = 42;
int &r = n;           // r is bound to n (i.e., r is another name for n)
r = 42;               // n is now 42
r = i;                // n now has the same value as i
i = r;                // i has the same value as n
// function that takes a reference to an int and sets the given object to zero
void reset(int &i) // i is just another name for the object passed to reset
{
    i = 0; // changes the value of the object to which i refers
}
int j = 42;
reset(j); // j is passed by reference; the value in j is changed
cout << "j = " << j << endl; // prints j = 0

```

```

// compare the length of two strings
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
// returns the index of the first occurrence of c in s
// the reference parameter occurs counts how often c occurs
string::size_type find_char(const string &s, char c,
                            string::size_type &occurs)
{
    auto ret = s.size(); // position of the first occurrence, if any
    occurs = 0; // set the occurrence count parameter
    for (decltype(ret) i = 0; i != s.size(); ++i) {
        if (s[i] == c) {
            if (ret == s.size())
                ret = i; // remember the first occurrence of c
            ++occurs; // increment the occurrence count
        }
    }
    return ret; // count is returned implicitly in occurs
}
const int ci = 42; // we cannot change ci; const is top-level
int i = ci; // ok: when we copy ci, its top-level const is ignored
int * const p = &i; // const is top-level; we can't assign to p
*p = 0; // ok: changes through p are allowed; i is now 0
void fcn(const int i) { /* fcn can read but not write to i */ }
void fcn(const int i) { /* fcn can read but not write to i */ }
void fcn(int i) { /* ... */ } // error: redefines fcn(int)
int i = 42;
const int *cp = &i; // ok: but cp can't change i (§ 2.4.2 (p. 62))
const int &r = i; // ok: but r can't change i (§ 2.4.1 (p. 61))
const int &r2 = 42; // ok: (§ 2.4.1 (p. 61))
int *p = cp; // error: types of p and cp don't match (§ 2.4.2 (p. 62))
int &r3 = r; // error: types of r3 and r don't match (§ 2.4.1 (p. 61))
int &r4 = 42; // error: can't initialize a plain reference from a literal (§ 2.3.1 (p. 50))
int i = 0;
const int ci = i;
string::size_type ctr = 0;
reset(&i); // calls the version of reset that has an int* parameter
reset(&ci); // error: can't initialize an int* from a pointer to a const int object
reset(i); // calls the version of reset that has an int& parameter
reset(ci); // error: can't bind a plain reference to the const object ci
reset(42); // error: can't bind a plain reference to a literal
reset(ctr); // error: types don't match; ctr has an unsigned type
// ok: find_char's first parameter is a reference to const
find_char("Hello World!", 'o', ctr);
// bad design: the first parameter should be a const string&
string::size_type find_char(string &s, char c,
                           string::size_type &occurs);
find_char("Hello World", 'o', ctr);

```

```

bool is_sentence(const string &s)
{
    // if there's a single period at the end of s, then s is a sentence
    string::size_type ctr = 0;
    return find_char(s, '.', ctr) == s.size() - 1 && ctr == 1;
}
bool is_empty(string& s) { return s.empty(); }
double calc(double);
int count(const string &, char);
int sum(vector<int>::iterator, vector<int>::iterator, int);
vector<int> vec(10);
// despite appearances, these three declarations of print are equivalent
// each function has a single parameter of type const int*
void print(const int*);
void print(const int[]); // shows the intent that the function takes an array
void print(const int[10]); // dimension for documentation purposes (at best)
    int i = 0, j[2] = {0, 1};
    print(&i); // ok: &i is int*
    print(j); // ok: j is converted to an int* that points to j [0]
void print(const char *cp)
{
    if (cp) // if cp is not a null pointer
        while (*cp) // so long as the character it points to is not a null character
            cout << *cp++; // print the character and advance the pointer
}
void print(const int *beg, const int *end)
{
    // print every element starting at beg up to but not including end
    while (beg != end)
        cout << *beg++ << endl; // print the current element
                                // and advance the pointer
}
int j[2] = {0, 1};
// j is converted to a pointer to the first element in j
// the second argument is a pointer to one past the end of j
print(begin(j), end(j)); // begin and end functions, see § 3.5.3 (p. 118)
// const int ia[] is equivalent to const int* ia
// size is passed explicitly and used to control access to elements of ia
void print(const int ia[], size_t size)
{
    for (size_t i = 0; i != size; ++i) {
        cout << ia[i] << endl;
    }
}
int j[] = { 0, 1 }; // int array of size 2
print(j, end(j) - begin(j));
// ok: parameter is a reference to an array; the dimension is part of the type
void print(int (&arr)[10])
{
    for (auto elem : arr)
        cout << elem << endl;
}

```

```

f(int &arr[10]) // error: declares arr as an array of references
f(int (&arr)[10]) // ok: arr is a reference to an array of ten ints
    int i = 0, j[2] = {0, 1};
    int k[10] = {0,1,2,3,4,5,6,7,8,9};
    print(&i); // error: argument is not an array of ten ints
    print(j); // error: argument is not an array of ten ints
    print(k); // ok: argument is an array of ten ints
// matrix points to the first element in an array whose elements are arrays of ten ints
void print(int (*matrix)[10], int rowSize) { /* ... */ }
    int *matrix[10]; // array of ten pointers
    int (*matrix)[10]; // pointer to an array of ten ints
// equivalent definition
void print(int matrix[][10], int rowSize) { /* ... */ }
    void print(const int ia[10])
    {
        for (size_t i = 0; i != 10; ++i)
            cout << ia[i] << endl;
    }
int main(int argc, char *argv[]) { ... }
int main(int argc, char **argv) { ... }
argv[0] = "prog"; // or argv[0] might point to an empty string
argv[1] = "-d";
argv[2] = "-o";
argv[3] = "ofile";
argv[4] = "data0";
argv[5] = 0;
initializer_list<string> ls; // initializer_list of strings
initializer_list<int> li; // initializer list of ints
void error_msg(initializer_list<string> il)
{
    for (auto beg = il.begin(); beg != il.end(); ++beg)
        cout << *beg << " ";
    cout << endl;
}
// expected, actual are strings
if (expected != actual)
    error_msg({"functionX", expected, actual});
else
    error msg({"functionX", "okay"});
void error_msg(ErrCode e, initializer_list<string> il)
{
    cout << e.msg() << ":" ;
    for (const auto &elem : il)
        cout << elem << " ";
    cout << endl;
}
if (expected != actual)
    error_msg(ErrCode(42), {"functionX", expected, actual});
else
    error msg(ErrCode(0), {"functionX", "okay"});

```

```

void swap(int &v1, int &v2)
{
    // if the values are already the same, no need to swap, just return
    if (v1 == v2)
        return;
    // if we're here, there's work to do
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    // no explicit return necessary
}
// incorrect return values, this code will not compile
bool str_subrange(const string &str1, const string &str2)
{
    // same sizes: return normal equality test
    if (str1.size() == str2.size())
        return str1 == str2; // ok: == returns bool
    // find the size of the smaller string; conditional operator, see § 4.7 (p. 151)
    auto size = (str1.size() < str2.size())
        ? str1.size() : str2.size();
    // look at each element up to the size of the smaller string
    for (decltype(size) i = 0; i != size; ++i) {
        if (str1[i] != str2[i])
            return; // error #1: no return value; compiler should detect this error
    }
    // error #2: control might flow off the end of the function without a return
    // the compiler might not detect this error
}
// return the plural version of word if ctr is greater than 1
string make_plural(size_t ctr, const string &word,
                    const string &ending)
{
    return (ctr > 1) ? word + ending : word;
}
// return a reference to the shorter of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
// disaster: this function returns a reference to a local object
const string &manip()
{
    string ret;
    // transform ret in some way
    if (!ret.empty())
        return ret; // WRONG: returning a reference to a local object!
    else
        return "Empty"; // WRONG: "Empty" is a local temporary string
}
// call the size member of the string returned by shorterString
auto sz = shorterString(s1, s2).size();

```

```

char &get_val(string &str, string::size_type ix)
{
    return str[ix]; // get_val assumes the given index is valid
}
int main()
{
    string s("a value");
    cout << s << endl; // prints a value
    get_val(s, 0) = 'A'; // changes s[0] to A
    cout << s << endl; // prints A value
    return 0;
}
shorterString("hi", "bye") = "X"; // error: return value is const
vector<string> process()
{
    // ...
    // expected and actual are strings
    if (expected.empty())
        return {}; // return an empty vector
    else if (expected == actual)
        return {"functionX", "okay"}; // return list-initialized vector
    else
        return {"functionX", expected, actual};
}
int main()
{
    if (some_failure)
        return EXIT_FAILURE; // defined in cstdlib
    else
        return EXIT_SUCCESS; // defined in cstdlib
}
// calculate val!, which is 1 * 2 * 3... * val
int factorial(int val)
{
    if (val > 1)
        return factorial(val-1) * val;
    return 1;
}
int &get(int *arry, int index) { return arry[index]; }
int main()
{
    int ia[10];
    for (int i = 0; i != 10; ++i)
        get(ia, i) = i;
}
typedef int arrT[10]; // arrT is a synonym for the type array of ten ints
using arrT = int[10];// equivalent declaration of arrT; see § 2.5.1 (p. 68)
arrT* func(int i); // func returns a pointer to an array of five ints
    int arr[10]; // arr is an array of ten ints
    int *p1[10]; // p1 is an array of ten pointers
    int (*p2)[10] = &arr; // p2 points to an array of ten ints
    Type (*function(parameter_list)) [dimension]
// fcn takes an int argument and returns a pointer to an array of ten ints
auto func(int i) -> int(*)[10];

```

```

int odd[] = {1,3,5,7,9};
int even[] = {0,2,4,6,8};
// returns a pointer to an array of five int elements
decltype(odd) *arrPtr(int i)
{
    return (i % 2) ? &odd : &even; // returns a pointer to the array
}
void print(const char *cp);
void print(const int *beg, const int *end);
void print(const int ia[], size_t size);
int j[2] = {0,1};
print("Hello World");           // calls print(const char*)
print(j, end(j) - begin(j));   // calls print(const int*, size_t)
print(begin(j), end(j));       // calls print(const int*, const int*)
Record lookup(const Account&); // find by Account
Record lookup(const Phone&);   // find by Phone
Record lookup(const Name&);    // find by Name

Account acct;
Phone phone;
Record r1 = lookup(acct);     // call version that takes an Account
Record r2 = lookup(phone);   // call version that takes a Phone
Record lookup(const Account&);

bool lookup(const Account&);  // error: only the return type is different
// each pair declares the same function
Record lookup(const Account &acct);
Record lookup(const Account&); // parameter names are ignored

typedef Phone Telno;
Record lookup(const Phone&);
Record lookup(const Telno&); // Telno and Phone are the same type
Record lookup(Phone);
Record lookup(const Phone);  // redeclares Record lookup(Phone)

Record lookup(Phone*);
Record lookup(Phone* const); // redeclares Record lookup(Phone*)
// functions taking const and nonconst references or pointers have different parameters
// declarations for four independent, overloaded functions
Record lookup(Account&);      // function that takes a reference to Account
Record lookup(const Account&); // new function that takes a const reference
Record lookup(Account*);        // new function, takes a pointer to Account
Record lookup(const Account*); // new function, takes a pointer to const
// return a reference to the shorter of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
Screen& moveHome();
Screen& moveAbs(int, int);
Screen& moveRel(int, int, string direction);
    Screen& move();
    Screen& move(int, int);
    Screen& move(int, int, string direction);

```

```

    // which is easier to understand?
    myScreen.moveHome(); // we think this one!
    myScreen.move();
string &shorterString(string &s1, string &s2)
{
    auto &r = shorterString(const_cast<const string&>(s1),
                           const_cast<const string&>(s2));
    return const_cast<string&>(r);
}
string read();
void print(const string &);
void print(double); // overloads the print function
void fooBar(int ival)
{
    bool read = false; // new scope: hides the outer declaration of read
    string s = read(); // error: read is a bool variable, not a function
    // bad practice: usually it's a bad idea to declare functions at local scope
    void print(int); // new scope: hides previous instances of print
    print("Value: "); // error: print(const string &) is hidden
    print(ival); // ok: print(int) is visible
    print(3.14); // ok: calls print(int); print(double) is hidden
}
void print(const string &);
void print(double); // overloads the print function
void print(int); // another overloaded instance
void fooBar2(int ival)
{
    print("Value: "); // calls print(const string &)
    print(ival); // calls print(int)
    print(3.14); // calls print(double)
}
typedef string::size_type sz; // typedef see § 2.5.1 (p. 67)
string screen(sz ht = 24, sz wid = 80, char backgrnd = ' ');
string window;
window = screen(); // equivalent to screen(24, 80, ' ')
window = screen(66); // equivalent to screen(66, 80, ' ')
window = screen(66, 256); // screen(66, 256, ' ')
window = screen(66, 256, '#'); // screen(66, 256, '#')
window = screen(, , '?'); // error: can omit only trailing arguments
window = screen('?''); // calls screen('?', 80, ' ')
// no default for the height or width parameters
string screen(sz, sz, char = ' ');
string screen(sz, sz, char = '*'); // error: redeclaration
string screen(sz = 24, sz = 80, char); // ok: adds default arguments
// the declarations of wd, def, and ht must appear outside a function
sz wd = 80;
char def = ' ';
sz ht();
string screen(sz = ht(), sz = wd, char = def);
string window = screen(); // calls screen(ht(), 80, ' ')

```

```

void f2()
{
    def = '*'; // changes the value of a default argument
    sz wd = 100; // hides the outer definition of wd but does not change the default
    window = screen(); // calls screen(ht(), 80, '*')
}
char *init(int ht, int wd = 80, char bckgrnd = ' ');
    cout << shorterString(s1, s2) << endl;
    cout << (s1.size() < s2.size() ? s1 : s2) << endl;
    // inline version: find the shorter of two strings
    inline const string &
    shorterString(const string &s1, const string &s2)
    {
        return s1.size() <= s2.size() ? s1 : s2;
    }
constexpr int new_sz() { return 42; }
constexpr int foo = new_sz(); // ok: foo is a constant expression
// scale(arg) is a constant expression if arg is a constant expression
constexpr size_t scale(size_t cnt) { return new_sz() * cnt; }
int arr[scale(2)]; // ok: scale(2) is a constant expression
int i = 2; // i is not a constant expression
int a2[scale(i)]; // error: scale(i) is not a constant expression
    assert(word.size() > threshold);
$ CC -D NDEBUG main.C # use /D with the Microsoft compiler
void print(const int ia[], size_t size)
{
#endif NDEBUG
// __func__ is a local static defined by the compiler that holds the function's name
cerr << __func__ << ": array size is " << size << endl;
#endif
// ...
if (word.size() < threshold)
    cerr << "Error: " << __FILE__
        << " : in function " << __func__
        << " at line " << __LINE__ << endl
        << "           Compiled on " << __DATE__
        << " at " << __TIME__ << endl
        << "           Word read was \\" << word
        << "\\": Length too short" << endl;
Error: wdebug.cc : in function main at line 27
    Compiled on Jul 11 2012 at 20:50:03
    Word read was "foo": Length too short
void f();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6); // calls void f(double, double)
string s;
while (cin >> s && s != sought) { } // empty body
assert(cin);
void ff(int);
void ff(short);
ff('a'); // char promotes to int; calls f(int)

```

```

        void manip(long);
        void manip(float);
        manip(3.14); // error: ambiguous call
Record lookup(Account&); // function that takes a reference to Account
Record lookup(const Account&); // new function that takes a const reference
const Account a;
Account b;
lookup(a); // calls lookup(const Account&)
lookup(b); // calls lookup(Account&)
// compares lengths of two strings
    bool lengthCompare(const string &, const string &);
// pf points to a function returning bool that takes two const string references
    bool (*pf)(const string &, const string &); // uninitialized
// declares a function named pf that returns a bool*
    bool *pf(const string &, const string &);
pf = lengthCompare; // pf now points to the function named lengthCompare
pf = &lengthCompare; // equivalent assignment: address-of operator is optional
    bool b1 = pf("hello", "goodbye"); // calls lengthCompare
    bool b2 = (*pf)("hello", "goodbye"); // equivalent call
    bool b3 = lengthCompare("hello", "goodbye"); // equivalent call
    string::size_type sumLength(const string&, const string&);
bool cstringCompare(const char*, const char*);
    pf = 0; // ok: pf points to no function
    pf = sumLength; // error: return type differs
    pf = cstringCompare; // error: parameter types differ
    pf = lengthCompare; // ok: function and pointer types match exactly
void ff(int*);
void ff(unsigned int);

void (*pf1)(unsigned int) = ff; // pf1 points to ff(unsigned)
void (*pf2)(int) = ff; // error: no ff with a matching parameter list
double (*pf3)(int*) = ff; // error: return type of ff and pf3 don't match
// third parameter is a function type and is automatically treated as a pointer to function
void useBigger(const string &s1, const string &s2,
               bool pf(const string &, const string &));
// equivalent declaration: explicitly define the parameter as a pointer to function
void useBigger(const string &s1, const string &s2,
               bool (*pf)(const string &, const string &));
// automatically converts the function lengthCompare to a pointer to function
useBigger(s1, s2, lengthCompare);
// Func and Func2 have function type
typedef bool Func(const string&, const string&);
typedef decltype(lengthCompare) Func2; // equivalent type
// FuncP and FuncP2 have pointer to function type
typedef bool(*FuncP)(const string&, const string&);
typedef decltype(lengthCompare) *FuncP2; // equivalent type
// equivalent declarations of useBigger using type aliases
void useBigger(const string&, const string&, Func);
void useBigger(const string&, const string&, FuncP2);
using F = int(int*, int); // F is a function type, not a pointer
using PF = int(*)(int*, int); // PF is a pointer type
PF f1(int); // ok: PF is a pointer to function; f1 returns a pointer to function
F f1(int); // error: F is a function type; f1 can't return a function
F *f1(int); // ok: explicitly specify that the return type is a pointer to function

```

```

        auto f1(int) -> int (*) (int*, int);
string::size_type sumLength(const string&, const string&);
string::size_type largerLength(const string&, const string&);

// depending on the value of its string parameter,
// getFcn returns a pointer to sumLength or to largerLength
decltype(sumLength) *getFcn(const string &);

Sales_data total; // variable to hold the running sum
if (read(cin, total)) { // read the first transaction
    Sales_data trans; // variable to hold data for the next transaction
    while(read(cin, trans)) { // read the remaining transactions
        if (total.isbn() == trans.isbn()) // check the ISBNs
            total.combine(trans); // update the running total
        else {
            print(cout, total) << endl; // print the results
            total = trans; // process the next book
        }
    }
    print(cout, total) << endl; // print the last transaction
} else {
    cerr << "No data?!" << endl; // there was no input
}
struct Sales_data {
    // new members: operations on Sales_data objects
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    // data members are unchanged from § 2.6.1 (p. 72)
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
// nonmember Sales_data interface functions
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&),
std::istream &read(std::istream&, Sales_data&),
    std::string isbn() const { return bookNo; }
// pseudo-code illustration of how a call to a member function is translated
Sales_data::isbn(&total)
    std::string isbn() const { return this->bookNo; }
// pseudo-code illustration of how the implicit this pointer is used
// this code is illegal: we may not explicitly define the this pointer ourselves
// note that this is a pointer to const because isbn is a const member
std::string Sales_data::isbn(const Sales_data *const this)
{ return this->isbn; }
    double Sales_data::avg_price() const {
        if (units_sold)
            return revenue/units_sold;
        else
            return 0;
}

```

```

Sales_data& Sales_data::combine(const Sales_data &rhs)
{
    units_sold += rhs.units_sold; // add the members of rhs into
    revenue += rhs.revenue; // the members of "this" object
    return *this; // return the object on which the function was called
}
    total.combine(trans); // update the running total
    units_sold += rhs.units_sold; // add the members of rhs into
    return *this; // return the object on which the function was called
// input transactions contain ISBN, number of copies sold, and sales price
istream &read(istream &is, Sales_data &item)
{
    double price = 0;
    is >> item.bookNo >> item.units_sold >> price;
    item.revenue = price * item.units_sold;
    return is;
}
ostream &print(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}
Sales_data add(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs; // copy data members from lhs into sum
    sum.combine(rhs); // add data members from rhs into sum
    return sum;
}
Sales_data total; // variable to hold the running sum
Sales_data trans; // variable to hold data for the next transaction
struct Sales_data {
    // constructors added
    Sales_data() = default;
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(std::istream &); // other members as before
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
Sales_data(const std::string &s): bookNo(s) { }
Sales_data(const std::string &s, unsigned n, double p):
    bookNo(s), units_sold(n), revenue(p*n) { }
    // has the same behavior as the original constructor defined above
Sales_data(const std::string &s):
    bookNo(s), units_sold(0), revenue(0) { }

```

```

Sales_data::Sales_data(std::istream &is)
{
    read(is, *this); // read will read a transaction from is into this object
}
    total = trans;           // process the next book
    // default assignment for Sales_data is equivalent to:
    total.bookNo = trans.bookNo;
    total.units_sold = trans.units_sold;
    total.revenue = trans.revenue;
class Sales_data {
public:           // access specifier added
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(std::istream&);
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);

private:          // access specifier added
    double avg_price() const
    { return units_sold ? revenue/units_sold : 0; }
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

class Sales_data {
// friend declarations for nonmember Sales_data operations added
friend Sales_data add(const Sales_data&, const Sales_data&);
friend std::istream &read(std::istream&, Sales_data&);
friend std::ostream &print(std::ostream&, const Sales_data&);

// other members and access specifiers as before
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(std::istream&);
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);

private:
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

// declarations for nonmember parts of the Sales_data interface
Sales_data add(const Sales_data&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);

```

```

        class Screen {
    public:
        typedef std::string::size_type pos;
private:
        pos cursor = 0;
        pos height = 0, width = 0;
        std::string contents;
    };
class Screen {
public:
    // alternative way to declare a type member using a type alias
    using pos = std::string::size_type;
    // other members as before
};
class Screen {
public:
    typedef std::string::size_type pos;
    Screen() = default; // needed because Screen has another constructor
    // cursor initialized to 0 by its in-class initializer
    Screen(pos ht, pos wd, char c): height(ht), width(wd),
                                    contents(ht * wd, c) {}
    char get() const // get the character at the cursor
    { return contents[cursor]; } // implicitly inline
    inline char get(pos ht, pos wd) const; // explicitly inline
    Screen &move(pos r, pos c); // can be made inline later
private:
    pos cursor = 0;
    pos height = 0, width = 0;
    std::string contents;
};
inline // we can specify inline on the definition
Screen &Screen::move(pos r, pos c)
{
    pos row = r * width; // compute the row location
    cursor = row + c; // move cursor to the column within that row
    return *this; // return this object as an lvalue
}
char Screen::get(pos r, pos c) const // declared as inline in the class
{
    pos row = r * width; // compute row location
    return contents[row + c]; // return character at the given column
}
Screen myscreen;
char ch = myscreen.get(); // calls Screen::get()
ch = myscreen.get(0,0); // calls Screen::get(pos, pos)

```

```

class Screen {
public:
    void some_member() const;
private:
    mutable size_t access_ctr; // may change even in a const object
    // other members as before
};
void Screen::some_member() const
{
    ++access_ctr; // keep a count of the calls to any member function
    // whatever other work this member needs to do
}
class Window_mgr {
private:
    // Screens this Window_mgr is tracking
    // by default, a Window_mgr has one standard sized blank Screen
    std::vector<Screen> screens{Screen(24, 80, ' ')};
};

class Screen {
public:
    Screen &set(char);
    Screen &set(pos, pos, char);
    // other members as before
};
inline Screen &Screen::set(char c)
{
    contents[cursor] = c; // set the new value at the current cursor location
    return *this;           // return this object as an lvalue
}
inline Screen &Screen::set(pos r, pos col, char ch)
{
    contents[r*width + col] = ch; // set specified location to given value
    return *this;               // return this object as an lvalue
}
// move the cursor to a given position, and set that character
myScreen.move(4,0).set('#');
// if move returns Screen not Screen&
Screen temp = myScreen.move(4,0); // the return value would be copied
temp.set('#'); // the contents inside myScreen would be unchanged
Screen myScreen;
// if display returns a const reference, the call to set is an error
myScreen.display(cout).set('*');

```

```
class Screen {
public:
    // display overloaded on whether the object is const or not
    Screen &display(std::ostream &os)
        { do_display(os); return *this; }
    const Screen &display(std::ostream &os) const
        { do_display(os); return *this; }
private:
    // function to do the work of displaying a Screen
    void do_display(std::ostream &os) const {os << contents;}
    // other members as before
};

Screen myScreen(5,3);
const Screen blank(5, 3);
myScreen.set('#').display(cout); // calls nonconst version
blank.display(cout);           // calls const version
Screen myScreen(5, 5, 'X');
myScreen.move(4,0).set('#').display(cout);
cout << "\n";
myScreen.display(cout);
cout << "\n";
struct First {
    int memi;
    int getMem();
};
struct Second {
    int memi;
    int getMem();
};
First obj1;
Second obj2 = obj1; // error: obj1 and obj2 have different types
Sales_data item1;      // default-initialized object of type Sales_data
class Sales_data item1; // equivalent declaration
    class Screen; // declaration of the Screen class
class Screen {
    // Window_mgr members can access the private parts of class Screen
    friend class Window_mgr;
    // ... rest of the Screen class
};
```

```

class Window_mgr {
public:
    // location ID for each screen on the window
    using ScreenIndex = std::vector<Screen>::size_type;
    // reset the Screen at the given position to all blanks
    void clear(ScreenIndex);
private:
    std::vector<Screen> screens{Screen(24, 80, ' ')};
};

void Window_mgr::clear(ScreenIndex i)
{
    // s is a reference to the Screen we want to clear
    Screen &s = screens[i];
    // reset the contents of that Screen to all blanks
    s.contents = string(s.height * s.width, ' ');
}

class Screen {
    // Window_mgr::clear must have been declared before class Screen
    friend void Window_mgr::clear(ScreenIndex);
    // ...rest of the Screen class
};

// overloaded storeOn functions
extern std::ostream& storeOn(std::ostream &, Screen &);
extern BitMap& storeOn(BitMap &, Screen &);

class Screen {
    // ostream version of storeOn may access the private parts of Screen objects
    friend std::ostream& storeOn(std::ostream &, Screen &);
    // ...
};

struct X {
    friend void f() { /* friend function can be defined in the class body */ }
    X() { f(); } // error: no declaration for f
    void g();
    void h();
};

void X::g() { return f(); } // error: f hasn't been declared
void f(); // declares the function defined inside X
void X::h() { return f(); } // ok: declaration for f is now in scope
Screen::pos ht = 24, wd = 80; // use the pos type defined by Screen
Screen scr(ht, wd, ' ');
Screen *p = &scr;
char c = scr.get(); // fetches the get member from the object scr
c = p->get(); // fetches the get member from the object to which p points
void Window_mgr::clear(ScreenIndex i)
{
    Screen &s = screens[i];
    s.contents = string(s.height * s.width, ' ');
}

```

```

class Window_mgr {
public:
    // add a Screen to the window and returns its index
    ScreenIndex addScreen(const Screen&);

    // other members as before
};

// return type is seen before we're in the scope of Window_mgr
Window_mgr::ScreenIndex
Window_mgr::addScreen(const Screen &s)
{
    screens.push_back(s);
    return screens.size() - 1;
}

pos Screen::size() const
{
    return height * width;
}

typedef double Money;
string bal;

class Account {
public:
    Money balance() { return bal; }

private:
    Money bal;
    // ...
};

typedef double Money;

class Account {
public:
    Money balance() { return bal; } // uses Money from the outer scope

private:
    typedef double Money; // error: cannot redefine Money
    Money bal;
    // ...
};

// note: this code is for illustration purposes only and reflects bad practice
// it is generally a bad idea to use the same name for a parameter and a member
int height; // defines a name subsequently used inside Screen

class Screen {
public:
    typedef std::string::size_type pos;
    void dummy_fcn(pos height) {
        cursor = width * height; // which height? the parameter
    }

private:
    pos cursor = 0;
    pos height = 0, width = 0;
};

```

```
// bad practice: names local to member functions shouldn't hide member names
void Screen::dummy_fcn(pos height) {
    cursor = width * this->height; // member height
    // alternative way to indicate the member
    cursor = width * Screen::height; // member height
}
// good practice: don't use a member name for a parameter or other local variable
void Screen::dummy_fcn(pos ht) {
    cursor = width * height; // member height
}
// bad practice: don't hide names that are needed from surrounding scopes
void Screen::dummy_fcn(pos height) {
    cursor = width * ::height; // which height? the global one
}
int height; // defines a name subsequently used inside Screen
class Screen {
public:
    typedef std::string::size_type pos;
    void setHeight(pos);
    pos height = 0; // hides the declaration of height in the outer scope
};
Screen::pos verify(Screen::pos);
void Screen::setHeight(pos var) {
    // var: refers to the parameter
    // height: refers to the class member
    // verify: refers to the global function
    height = verify(var);
}
typedef string Type;
Type initVal();
class Exercise {
public:
    typedef double Type;
    Type setVal(Type);
    Type initVal();
private:
    int val;
};
Type Exercise::setVal(Type parm) {
    val = parm + initVal();
    return val;
}
string foo = "Hello World!"; // define and initialize
string bar; // default initialized to the empty string
bar = "Hello World!"; // assign a new value to bar
```

```

// legal but sloppier way to write the Sales_data constructor: no constructor initializers
Sales_data::Sales_data(const string &s,
                      unsigned cnt, double price)
{
    bookNo = s;
    units_sold = cnt;
    revenue = cnt * price;
}
// error: ci and ri must be initialized
ConstRef::ConstRef(int ii)
{
    // assignments:
    i = ii; // ok
    ci = ii; // error: cannot assign to a const
    ri = i; // error: xi was never initialized
}
// ok: explicitly initialize reference and const members
ConstRef::ConstRef(int ii): i(ii), ci(ii), ri(i) { }
class X {
    int i;
    int j;
public:
    // undefined: i is initialized before j
    X(int val): j(val), i(j) { }
};
X(int val): i(val), j(val) { }
class Sales_data {
public:
    // defines the default constructor as well as one that takes a string argument
    Sales_data(std::string s = ""): bookNo(s) { }
    // remaining constructors unchanged
    Sales_data(std::string s, unsigned cnt, double rev):
        bookNo(s), units_sold(cnt), revenue(rev*cnt) { }
    Sales_data(std::istream &is) { read(is, *this); }
    // remaining members as before
};
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int rem, base;
};
Sales_data first_item(cin);

int main() {
    Sales_data next;
    Sales_data last("9-999-99999-9");
}

```

```

class Sales_data {
public:
    // nondelegating constructor initializes members from corresponding arguments
    Sales_data(std::string s, unsigned cnt, double price):
        bookNo(s), units_sold(cnt), revenue(cnt*price) { }
    // remaining constructors all delegate to another constructor
    Sales_data(): Sales_data("", 0, 0) {}
    Sales_data(std::string s): Sales_data(s, 0,0) {}
    Sales_data(std::istream &is): Sales_data()
        { read(is, *this); }

    // other members as before
};

class NoDefault {
public:
    NoDefault(const std::string&);

    // additional members follow, but no other constructors
};

struct A { // my_mem is public by default; see § 7.2 (p. 268)
    NoDefault my_mem;
};

A a;           // error: cannot synthesize a constructor for A

struct B {
    B() {} // error: no initializer for b_member
    NoDefault b_member;
};

Sales_data obj(); // ok: but defines a function, not an object
if (obj.isbn() == Primer_5th_ed::isbn()) // error: obj is a function
    // ok: obj is a default-initialized object
    Sales_data obj;

Sales_data obj(); // oops! declares a function, not an object
Sales_data obj2; // ok: obj2 is an object, not a function
string null_book = "9-999-99999-9";

// constructs a temporary Sales_data object
// with units_sold and revenue equal to 0 and bookNo equal to null_book
item.combine(null_book);
    // error: requires two user-defined conversions:
    // (1) convert "9-999-99999-9" to string
    // (2) convert that (temporary) string to Sales_data
    item.combine("9-999-99999-9");
// ok: explicit conversion to string, implicit conversion to Sales_data
item.combine(string("9-999-99999-9"));
// ok: implicit conversion to string, explicit conversion to Sales_data
item.combine(Sales_data("9-999-99999-9"));
    // uses the istream constructor to build an object to pass to combine
item.combine(cin);

```

```

class Sales_data {
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    explicit Sales_data(const std::string &s): bookNo(s) { }
    explicit Sales_data(std::istream&); // remaining members as before
};
item.combine(null_book); // error: string constructor is explicit
item.combine(cin); // error: istream constructor is explicit
// error: explicit allowed only on a constructor declaration in a class header
explicit Sales_data::Sales_data(istream& is)
{
    read(is, *this);
}
Sales_data item1(null_book); // ok: direct initialization
// error: cannot use the copy form of initialization with an explicit constructor
Sales_data item2 = null_book;
// ok: the argument is an explicitly constructed Sales_data object
item.combine(Sales_data(null_book));
// ok: static_cast can use an explicit constructor
item.combine(static_cast<Sales_data>(cin));
string null_isbn("9-999-99999-9");
Sales_data item1(null_isbn);
Sales_data item2("9-999-99999-9");
// val1.ival = 0; val1.s = string("Anna")
Data val1 = { 0, "Anna" };
// error: can't use "Anna" to initialize ival, or 1024 to initialize s
Data val2 = { "Anna", 1024 };
Sales_data item = {"978-0590353403", 25, 15.99};
class Debug {
public:
    constexpr Debug(bool b = true): hw(b), io(b), other(b) { }
    constexpr Debug(bool h, bool i, bool o):
        hw(h), io(i), other(o) { }
    constexpr bool any() { return hw || io || other; }
    void set_io(bool b) { io = b; }
    void set_hw(bool b) { hw = b; }
    void set_other(bool b) { other = b; }
private:
    bool hw; // hardware errors other than IO errors
    bool io; // IO errors
    bool other; // other errors
};
constexpr Debug io_sub(false, true, false); // debugging IO
if (io_sub.any()) // equivalent to if(true)
    cerr << "print appropriate error messages" << endl;
constexpr Debug prod(false); // no debugging during production
if (prod.any()) // equivalent to if(false)
    cerr << "print an error message" << endl;

```

```

class Account {
public:
    void calculate() { amount += amount * interestRate; }
    static double rate() { return interestRate; }
    static void rate(double);
private:
    std::string owner;
    double amount;
    static double interestRate;
    static double initRate();
};

double r;
r = Account::rate(); // access a static member using the scope operator
Account ac1;
Account *ac2 = &ac1;
// equivalent ways to call the static member rate function
r = ac1.rate();           // through an Account object or reference
r = ac2->rate();         // through a pointer to an Account object
class Account {
public:
    void calculate() { amount += amount * interestRate; }
private:
    static double interestRate;
    // remaining members as before
};
void Account::rate(double newRate)
{
    interestRate = newRate;
}
// define and initialize a static class member
double Account::interestRate = initRate();
class Account {
public:
    static double rate() { return interestRate; }
    static void rate(double);
private:
    static constexpr int period = 30; // period is a constant expression
    double daily_tbl[period];
};
// definition of a static member with no initializer
constexpr int Account::period; // initializer provided in the class definition
class Bar {
public:
    // ...
private:
    static Bar mem1; // ok: static member can have incomplete type
    Bar *mem2;       // ok: pointer member can have incomplete type
    Bar mem3;        // error: data members must have complete type
};

```

```

        class Screen {
    public:
        // bkground refers to the static member
        // declared later in the class definition
        Screen& clear(char = bkground);
    private:
        static const char bkground;
    };
// example.h
class Example {
public:
    static double rate = 6.5;
    static const int vecSize = 20;
    static vector<double> vec(vecSize);
};
// example.C
#include "example.h"
double Example::rate;
vector<double> Example::vec;
ofstream out1, out2;
out1 = out2;           // error: cannot assign stream objects
ofstream print(ofstream); // error: can't initialize the ostream parameter
out2 = print(out2);    // error: cannot copy stream objects
// remember the current state of cin
auto old_state = cin.rdstate(); // remember the current state of cin
cin.clear();             // make cin valid
process_input(cin);      // use cin
cin.setstate(old_state); // now reset cin to its old state
// turns off failbit and badbit but all other bits unchanged
cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);
cout << "hi!" << endl; // writes hi and a newline, then flushes the buffer
cout << "hi!" << flush; // writes hi, then flushes the buffer; adds no data
cout << "hi!" << ends; // writes hi and a null, then flushes the buffer
cout << unitbuf;        // all writes will be flushed immediately
// any output is flushed immediately, no buffering
cout << nounitbuf;      // returns to normal buffering
cin.tie(&cout);          // illustration only: the library ties cin and cout for us
// old_tie points to the stream (if any) currently tied to cin
ostream *old_tie = cin.tie(nullptr); // cin is no longer tied
// ties cin and cerr; not a good idea because cin should be tied to cout
cin.tie(&cerr);          // reading cin flushes cerr, not cout
cin.tie(old_tie);         // reestablish normal tie between cin and cout
ifstream in(ifile);       // construct an ifstream and open the given file
ofstream out;              // output file stream that is not associated with any file

```

```

ifstream input(argv[1]);    // open the file of sales transactions
ofstream output(argv[2]);  // open the output file
Sales_data total;          // variable to hold the running sum
if (read(input, total)) {   // read the first transaction
    Sales_data trans;      // variable to hold data for the next transaction
    while(read(input, trans)) { // read the remaining transactions
        if (total.isbn() == trans.isbn()) // check isbns
            total.combine(trans); // update the running total
        else {
            print(output, total) << endl; // print the results
            total = trans;           // process the next book
        }
    }
    print(output, total) << endl; // print the last transaction
} else
    cerr << "No data?!" << endl;
ifstream in(ifile); // construct an ifstream and open the given file
ofstream out;         // output file stream that is not associated with any file
out.open(ifile + ".copy"); // open the specified file
if (out)             // check that the open succeeded
    // the open succeeded, so we can use the file
    in.close();          // close the file
    in.open(ifile + "2"); // open another file
// for each file passed to the program
for (auto p = argv + 1; p != argv + argc; ++p) {
    ifstream input(*p); // create input and open the file
    if (input) {         // if the file is ok, "process" this file
        process(input);
    } else
        cerr << "couldn't open: " + string(*p);
} // input goes out of scope and is destroyed on each iteration
// file1 is truncated in each of these cases.
ofstream out("file1"); // out and trunc are implicit
ofstream out2("file1", ofstream::out); // trunc is implicit
ofstream out3("file1", ofstream::out | ofstream::trunc);
// to preserve the file's contents, we must explicitly specify app mode
ofstream app("file2", ofstream::app); // out is implicit
ofstream app2("file2", ofstream::out | ofstream::app);
ofstream out; // no file mode is set
out.open("scratchpad"); // mode implicitly out and trunc
out.close(); // close out so we can use it for a different file
out.open("precious", ofstream::app); // mode is out and app
out.close();
morgan 2015552368 8625550123
drew 9735550130
lee 6095550132 2015550175 8005550000
// members are public by default; see § 7.2 (p. 268)
struct PersonInfo {
    string name;
    vector<string> phones;
};

```

```

string line, word; // will hold a line and word from input, respectively
vector<PersonInfo> people; // will hold all the records from the input
// read the input a line at a time until cin hits end-of-file (or another error)
while (getline(cin, line)) {
    PersonInfo info; // create an object to hold this record's data
    istringstream record(line); // bind record to the line we just read
    record >> info.name; // read the name
    while (record >> word) // read the phone numbers
        info.phones.push_back(word); // and store them
    people.push_back(info); // append this record to people
}
for (const auto &entry : people) { // for each entry in people
    ostringstream formatted, badNums; // objects created on each loop
    for (const auto &nums : entry.phones) { // for each number
        if (!valid(nums)) {
            badNums << " " << nums; // string in badNums
        } else
            // "writes" to formatted's string
            formatted << " " << format(nums);
    }
    if (badNums.str().empty()) // there were no bad numbers
        os << entry.name << " " // print the name
        << formatted.str() << endl; // and reformatted numbers
    else // otherwise, print the name and bad numbers
        cerr << "input error: " << entry.name
        << " invalid number(s) " << badNums.str() << endl;
}
list<Sales_data> // list that holds Sales_data objects
deque<double> // deque that holds doubles
vector<vector<string>> lines; // vector of vectors
// assume noDefault is a type without a default constructor
vector<noDefault> v1(10, init); // ok: element initializer supplied
vector<noDefault> v2(10); // error: must supply an element initializer
while (begin != end) {
    *begin = val; // ok: range isn't empty so begin denotes an element
    ++begin; // advance the iterator to get the next element
}
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
               iter2 = lst1.end();
while (iter1 < iter2) /* ... */ // iter is the iterator type defined by list<string>
list<string>::iterator iter;
// count is the difference_type type defined by vector<int>
vector<int>::difference_type count;
list<string> a = {"Milton", "Shakespeare", "Austen"};
auto it1 = a.begin(); // list<string>::iterator
auto it2 = a.rbegin(); // list<string>::reverse_iterator
auto it3 = a.cbegin(); // list<string>::const_iterator
auto it4 = a.crbegin(); // list<string>::const_reverse_iterator

```

```

// type is explicitly specified
list<string>::iterator it5 = a.begin();
list<string>::const_iterator it6 = a.begin();
// iterator or const_iterator depending on a's type of a
auto it7 = a.begin(); // const_iterator only if a is const
auto it8 = a.cbegin(); // it8 is const_iterator
vector<int> v1;
const vector<int> v2;
auto it1 = v1.begin(), it2 = v2.begin();
auto it3 = v1.cbegin(), it4 = v2.cbegin();
// each container has three elements, initialized from the given initializers
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
list<string> list2(authors); // ok: types match
deque<string> authList(authors); // error: container types don't match
vector<string> words(articles); // error: element types must match
// ok: converts const char* elements to string
forward_list<string> words(articles.begin(), articles.end());
// copies up to but not including the element denoted by it
deque<string> authList(authors.begin(), it);
// each container has three elements, initialized from the given initializers
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
vector<int> ivec(10, -1); // ten int elements, each initialized to -1
list<string> svec(10, "hi!"); // ten strings; each element is "hi!"
forward_list<int> ivec(10); // ten elements, each initialized to 0
deque<string> svec(10); // ten elements, each an empty string
array<int, 42> // type is: array that holds 42 ints
array<string, 10> // type is: array that holds 10 strings
array<int, 10>::size_type i; // array type includes element type and size
array<int>::size_type j; // error: array<int> is not a type
array<int, 10> ia1; // ten default-initialized ints
array<int, 10> ia2 = {0,1,2,3,4,5,6,7,8,9}; // list initialization
array<int, 10> ia3 = {42}; // ia3[0] is 42, remaining elements are 0
int digs[10] = {0,1,2,3,4,5,6,7,8,9};
int cpy[10] = digs; // error: no copy or assignment for built-in arrays
array<int, 10> digits = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> copy = digits; // ok: so long as array types match
c1 = c2; // replace the contents of c1 with a copy of the elements in c2
c1 = {a,b,c}; // after the assignment c1 has size 3
array<int, 10> a1 = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> a2 = {0}; // elements all have value 0
a1 = a2; // replaces elements in a1
a2 = {0}; // error: cannot assign to an array from a braced list
list<string> names;
vector<const char*> oldstyle;
names = oldstyle; // error: container types don't match
// ok: can convert from const char* to string
names.assign(oldstyle.cbegin(), oldstyle.cend());
// equivalent to slist1.clear();
// followed by slist1.insert(slist1.begin(), 10, "Hiya!");
list<string> slist1(1); // one element, which is the empty string
slist1.assign(10, "Hiya!"); // ten elements; each one is Hiya!

```

```

        vector<string> svec1(10); // vector with ten elements
        vector<string> svec2(24); // vector with 24 elements
        swap(svec1, svec2);
vector<int> v1 = { 1, 3, 5, 7, 9, 12 };
vector<int> v2 = { 1, 3, 9 };
vector<int> v3 = { 1, 3, 5, 7 };
vector<int> v4 = { 1, 3, 5, 7, 9, 12 };
v1 < v2 // true; v1 and v2 differ at element [2]: v1[2] is less than v2[2]
v1 < v3 // false; all elements are equal, but v3 has fewer of them;
v1 == v4 // true; each element is equal and v1 and v4 have the same size()
v1 == v2 // false; v2 has fewer elements than v1
vector<Sales_data> storeA, storeB;
if (storeA < storeB) // error: Sales data has no less-than operator
    // read from standard input, putting each word onto the end of container
    string word;
    while (cin >> word)
        container.push_back(word);
void pluralize(size_t cnt, string &word)
{
    if (cnt > 1)
        word.push_back('s'); // same as word += 's'
}
list<int> ilist;
// add elements to the start of ilist
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
slist.insert(iter, "Hello!"); // insert "Hello!" just before iter
vector<string> svec;
list<string> slist;
// equivalent to calling slist.push_front("Hello!");
slist.insert(slist.begin(), "Hello!");
// no push_front on vector but we can insert before begin()
// warning: inserting anywhere but at the end of a vector might be slow
svec.insert(svec.begin(), "Hello!");
svec.insert(svec.end(), 10, "Anna");
vector<string> v = {"quasi", "simba", "frollo", "scar"};
// insert the last two elements of v at the beginning of slist
slist.insert(slist.begin(), v.end() - 2, v.end());
slist.insert(slist.end(), {"these", "words", "will",
    "go", "at", "the", "end"});
// run-time error: iterators denoting the range to copy from
// must not refer to the same container as the one we are changing
slist.insert(slist.begin(), slist.begin(), slist.end());
list<string> lst;
auto iter = lst.begin();
while (cin >> word)
    iter = lst.insert(iter, word); // same as calling push_front

```

```

// construct a Sales_data object at the end of c
// uses the three-argument Sales_data constructor
c.emplace_back("978-0590353403", 25, 15.99);
// error: there is no version of push_back that takes three arguments
c.push_back("978-0590353403", 25, 15.99);
// ok: we create a temporary Sales_data object to pass to push_back
c.push_back(Sales_data("978-0590353403", 25, 15.99));
// iter refers to an element in c, which holds Sales_data elements
c.emplace_back(); // uses the Sales_data default constructor
c.emplace(iter, "999-999999999"); // uses Sales_data(string)
// uses the Sales_data constructor that takes an ISBN, a count, and a price
c.emplace_front("978-0590353403", 25, 15.99);
vector<int>::iterator iter = iv.begin(),
                           mid = iv.begin() + iv.size()/2;
while (iter != mid)
    if (*iter == some_val)
        iv.insert(iter, 2 * some_val);
// check that there are elements before dereferencing an iterator or calling front or back
if (!c.empty()) {
    // val and val2 are copies of the value of the first element in c
    auto val = *c.begin(), val2 = c.front();
    // val3 and val4 are copies of the of the last element in c
    auto last = c.end();
    auto val3 = *(--last); // can't decrement forward_list iterators
    auto val4 = c.back(); // not supported by forward_list
}
if (!c.empty()) {
    c.front() = 42; // assigns 42 to the first element in c
    auto &v = c.back(); // get a reference to the last element
    v = 1024; // changes the element in c
    auto v2 = c.back(); // v2 is not a reference; it's a copy of c.back()
    v2 = 0; // no change to the element in c
}
vector<string> svec; // empty vector
cout << svec[0]; // run-time error: there are no elements in svec!
cout << svec.at(0); // throws an out_of_range exception
while (!ilist.empty()) {
    process(ilist.front()); // do something with the current top of ilist
    ilist.pop_front(); // done; remove the first element
}
list<int> lst = {0,1,2,3,4,5,6,7,8,9};
auto it = lst.begin();
while (it != lst.end())
    if (*it % 2) // if the element is odd
        it = lst.erase(it); // erase this element
    else
        ++it;
// delete the range of elements between two iterators
// returns an iterator to the element just after the last removed element
elem1 = slist.erase(elem1, elem2); // after the call elem1 == elem2
slist.clear(); // delete all the elements within the container
slist.erase(slist.begin(), slist.end()); // equivalent

```

```

        int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
forward_list<int> flst = {0,1,2,3,4,5,6,7,8,9};
auto prev = flst.before_begin(); // denotes element "off the start" of flst
auto curr = flst.begin(); // denotes the first element in flst
while (curr != flst.end()) { // while there are still elements to process
    if (*curr % 2) // if the element is odd
        curr = flst.erase_after(prev); // erase it and move curr
    else {
        prev = curr; // move the iterators to denote the next
        ++curr; // element and one before the next element
    }
}
list<int> ilist(10, 42); // ten ints: each has value 42
ilist.resize(15); // adds five elements of value 0 to the back of ilist
ilist.resize(25, -1); // adds ten elements of value -1 to the back of ilist
ilist.resize(5); // erases 20 elements from the back of ilist
// silly loop to remove even-valued elements and insert a duplicate of odd-valued elements
vector<int> vi = {0,1,2,3,4,5,6,7,8,9};
auto iter = vi.begin(); // call begin, not cbegin because we're changing vi
while (iter != vi.end()) {
    if (*iter % 2) {
        iter = vi.insert(iter, *iter); // duplicate the current element
        iter += 2; // advance past this element and the one inserted before it
    } else
        iter = vi.erase(iter); // remove even elements
    // don't advance the iterator; iter denotes the element after the one we erased
}
// disaster: the behavior of this loop is undefined
auto begin = v.begin(),
      end = v.end(); // bad idea, saving the value of the end iterator
while (begin != end) {
    // do some processing
    // insert the new value and reassign begin, which otherwise would be invalid
    ++begin; // advance begin because we want to insert after this element
    begin = v.insert(begin, 42); // insert the new value
    ++begin; // advance begin past the element we just added
}
// safer: recalculate end on each trip whenever the loop adds/erases elements
while (begin != v.end()) {
    // do some processing
    ++begin; // advance begin because we want to insert after this element
    begin = v.insert(begin, 42); // insert the new value
    ++begin; // advance begin past the element we just added
}
    iter = vi.insert(iter, *iter++);
    iter = vi.begin();
    while (iter != vi.end())
        if (*iter % 2)
            iter = vi.insert(iter, *iter);
        ++iter;
}

```

```

vector<int> ivec;
// size should be zero; capacity is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
// give ivec 24 elements
for (vector<int>::size_type ix = 0; ix != 24; ++ix)
    ivec.push_back(ix);

// size should be 24; capacity will be >= 24 and is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
ivec.reserve(50); // sets capacity to at least 50; might be more
// size should be 24; capacity will be >= 50 and is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
// add elements to use up the excess capacity
while (ivec.size() != ivec.capacity())
    ivec.push_back(0);

// capacity should be unchanged and size and capacity are now equal
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
ivec.push_back(42); // add one more element

// size should be 51; capacity will be >= 51 and is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
ivec.shrink_to_fit(); // ask for the memory to be returned

// size should be unchanged; capacity is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
vector<string> svec;
svec.reserve(1024);
string word;
while (cin >> word)
    svec.push_back(word);

svec.resize(svec.size() + svec.size() / 2);
const char *cp = "Hello World!!!"; // null-terminated array
char noNull[] = {'H', 'i'}; // not null terminated
string s1(cp); // copy up to the null in cp; s1 == "Hello World!!!"
string s2(noNull, 2); // copy two characters from no_null; s2 == "Hi"
string s3(noNull); // undefined: noNull not null terminated
string s4(cp + 6, 5); // copy 5 characters starting at cp[6]; s4 == "World"
string s5(s1, 6, 5); // copy 5 characters starting at s1[6]; s5 == "World"
string s6(s1, 6); // copy from s1[6] to end of s1; s6 == "World!!!"
string s7(s1, 6, 20); // ok, copies only to end of s1; s7 == "World!!!"
string s8(s1, 16); // throws an out_of_range exception
string s("hello world");
string s2 = s.substr(0, 5); // s2 = hello
string s3 = s.substr(6); // s3 = world
string s4 = s.substr(6, 11); // s3 = world
string s5 = s.substr(12); // throws an out_of_range exception

```

```

s.insert(s.size(), 5, '!'); // insert five exclamation points at the end of s
s.erase(s.size() - 5, 5); // erase the last five characters from s
const char *cp = "Stately, plump Buck";
s.assign(cp, 7); // s == "Stately"
s.insert(s.size(), cp + 7); // s == "Stately, plump Buck"
string s = "some string", s2 = "some other string";
s.insert(0, s2); // insert a copy of s2 before position 0 in s
// insert s2.size() characters from s2 starting at s2[0] before s[0]
s.insert(0, s2, 0, s2.size());
string s("C++ Primer"), s2 = s; // initialize s and s2 to "C++ Primer"
s.insert(s.size(), " 4th Ed."); // s == "C++ Primer 4th Ed."
s2.append(" 4th Ed."); // equivalent: appends " 4th Ed." to s2; s == s2
// equivalent way to replace "4th" by "5th"
s.erase(11, 3); // s == "C++ Primer Ed."
s.insert(11, "5th"); // s == "C++ Primer 5th Ed."
// starting at position 11, erase three characters and then insert "5th"
s2.replace(11, 3, "5th"); // equivalent: s == s2
s.replace(11, 3, "Fifth"); // s == "C++ Primer Fifth Ed."
string name("AnnaBelle");
auto pos1 = name.find("Anna"); // pos1 == 0
string lowercase("annabelle");
pos1 = lowercase.find("Anna"); // pos1 == npos
string numbers("0123456789"), name("r2d2");
// returns 1, i.e., the index of the first digit in name
auto pos = name.find_first_of(numbers);
string dept("03714p3");
// returns 5, which is the index to the character 'p'
auto pos = dept.find_first_not_of(numbers);
string::size_type pos = 0;
// each iteration finds the next number in name
while ((pos = name.find_first_of(numbers, pos))
       != string::npos) {
    cout << "found number at index: " << pos
       << " element is " << name[pos] << endl;
    ++pos; // move to the next character
}
string river("Mississippi");
auto first_pos = river.find("is"); // returns 1
auto last_pos = river.rfind("is"); // returns 4
int i = 42;
string s = to_string(i); // converts the int i to its character representation
double d = stod(s); // converts the string s to floating-point
string s2 = "pi = 3.14";
// convert the first substring in s that starts with a digit, d = 3.14
d = stod(s2.substr(s2.find_first_of("-+.0123456789")));
stack<int> stk(deq); // copies elements from deq into stk
// empty stack implemented on top of vector
stack<string, vector<string>> str_stk;
// str_stk2 is implemented on top of vector and initially holds a copy of svec
stack<string, vector<string>> str_stk2(svec);

```

```

stack<int> intStack; // empty stack
// fill up the stack
for (size_t ix = 0; ix != 10; ++ix)
    intStack.push(ix); // intStack holds 0...9 inclusive
while (!intStack.empty()) { // while there are still values in intStack
    int value = intStack.top();
    // code that uses value
    intStack.pop(); // pop the top element, and repeat
}
stack<int> intStack; // empty stack
intStack.push(ix); // intStack holds 0...9 inclusive
int val = 42; // value we'll look for
// result will denote the element we want if it's in vec, or vec.cend() if not
auto result = find(vec.cbegin(), vec.cend(), val);
// report the result
cout << "The value " << val
<< (result == vec.cend()
     ? " is not present" : " is present") << endl;
string val = "a value"; // value we'll look for
// this call to find looks through string elements in a list
auto result = find(1st.cbegin(), 1st.cend(), val);
int ia[] = {27, 210, 12, 47, 109, 83};
int val = 83;
int* result = find(begin(ia), end(ia), val);
// search the elements starting from ia[1] up to but not including ia[4]
auto result = find(ia + 1, ia + 4, val);
// sum the elements in vec starting the summation with the value 0
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
string sum = accumulate(v.cbegin(), v.cend(), string(""));
// error: no + on const char*
string sum = accumulate(v.cbegin(), v.cend(), "");
// roster2 should have at least as many elements as roster1
equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
fill(vec.begin(), vec.end(), 0); // reset each element to 0
// set a subsequence of the container to 10
fill(vec.begin(), vec.begin() + vec.size()/2, 10);
vector<int> vec; // empty vector
// use vec giving it various values
fill_n(vec.begin(), vec.size(), 0); // reset all the elements of vec to 0
vector<int> vec; // empty vector
// disaster: attempts to write to ten (nonexistent) elements in vec
fill_n(vec.begin(), 10, 0);
vector<int> vec; // empty vector
auto it = back_inserter(vec); // assigning through it adds elements to vec
*it = 42; // vec now has one element with value 42
vector<int> vec; // empty vector
// ok: back_inserter creates an insert iterator that adds elements to vec
fill_n(back_inserter(vec), 10, 0); // appends ten elements to vec
int a1[] = {0,1,2,3,4,5,6,7,8,9};
int a2[sizeof(a1)/sizeof(*a1)]; // a2 has the same size as a1
// ret points just past the last element copied into a2
auto ret = copy(begin(a1), end(a1), a2); // copy a1 into a2
// replace any element with the value 0 with 42
replace(ilist.begin(), ilist.end(), 0, 42);

```

```

    // use back_inserter to grow destination as needed
    replace_copy(ilist.cbegin(), ilist.cend(),
                 back_inserter(ivec), 0, 42);
the quick red fox jumps over the slow red turtle
vector<int> vec; list<int> lst; int i;
while (cin >> i)
    lst.push_back(i);
copy(lst.cbegin(), lst.cend(), vec.begin());
vector<int> vec;
vec.reserve(10); // reserve is covered in § 9.4 (p. 356)
fill_n(vec.begin(), 10, 0);
void elimDups(vector<string> &words)
{
    // sort words alphabetically so we can find the duplicates
    sort(words.begin(), words.end());
    // unique reorders the input range so that each word appears once in the
    // front portion of the range and returns an iterator one past the unique range
    auto end_unique = unique(words.begin(), words.end());
    // erase uses a vector operation to remove the nonunique elements
    words.erase(end_unique, words.end());
}
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
elimDups(words); // put words in alphabetical order and remove duplicates
// resort by length, maintaining alphabetical order among words of the same length
stable_sort(words.begin(), words.end(), isShorter);
for (const auto &s : words) // no need to copy the strings
    cout << s << " "; // print each element separated by a space
cout << endl;
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // put words in alphabetical order and remove duplicates
    // resort by length, maintaining alphabetical order among words of the same length
    stable_sort(words.begin(), words.end(), isShorter);
    // get an iterator to the first element whose size() is >= sz
    // compute the number of elements with size >= sz
    // print words of the given size or longer, each one followed by a space
}
[capture list] (parameter list) -> return type { function body }
    cout << f() << endl; // prints 42
    [] (const string &a, const string &b)
        { return a.size() < b.size(); }
// sort words by size, but maintain alphabetical order for words of the same size
stable_sort(words.begin(), words.end(),
            [] (const string &a, const string &b)
                { return a.size() < b.size(); });

```

```

[sz] (const string &a)
    { return a.size() >= sz; };
// error: sz not captured
[] (const string &a)
    { return a.size() >= sz; };
// get an iterator to the first element whose size() is >= sz
auto wc = find_if(words.begin(), words.end(),
    [sz] (const string &a)
        { return a.size() >= sz; });
// compute the number of elements with size >= sz
auto count = words.end() - wc;
cout << count << " " << make_plural(count, "word", "s")
    << " of length " << sz << " or longer" << endl;
// print words of the given size or longer, each one followed by a space
for_each(wc, words.end(),
    [] (const string &s){cout << s << " ;"});
cout << endl;
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // put words in alphabetical order and remove duplicates
    // sort words by size, but maintain alphabetical order for words of the same size
    stable_sort(words.begin(), words.end(),
        [] (const string &a, const string &b)
            { return a.size() < b.size(); });
    // get an iterator to the first element whose size() is >= sz
    auto wc = find_if(words.begin(), words.end(),
        [sz] (const string &a)
            { return a.size() >= sz; });
    // compute the number of elements with size >= sz
    auto count = words.end() - wc;
    cout << count << " " << make_plural(count, "word", "s")
        << " of length " << sz << " or longer" << endl;
    // print words of the given size or longer, each one followed by a space
    for_each(wc, words.end(),
        [] (const string &s){cout << s << " ;"});
    cout << endl;
}
void fcn1()
{
    size_t v1 = 42; // local variable
    // copies v1 into the callable object named f
    auto f = [v1] { return v1; };
    v1 = 0;
    auto j = f(); // j is 42; f stored a copy of v1 when we created it
}

```

```

void fcn2()
{
    size_t v1 = 42; // local variable
    // the object f2 contains a reference to v1
    auto f2 = [&v1] { return v1; };
    v1 = 0;
    auto j = f2(); // j is 0; f2 refers to v1; it doesn't store it
}
void biggies(vector<string> &words,
             vector<string>::size_type sz,
             ostream &os = cout, char c = ' ')
{
    // code to reorder words as before
    // statement to print count revised to print to os
    for_each(words.begin(), words.end(),
              [&os, c](const string &s) { os << s << c; });
}
// sz implicitly captured by value
wc = find_if(words.begin(), words.end(),
              [=](const string &s)
              { return s.size() >= sz; });
void biggies(vector<string> &words,
             vector<string>::size_type sz,
             ostream &os = cout, char c = ' ')
{
    // other processing as before
    // os implicitly captured by reference; c explicitly captured by value
    for_each(words.begin(), words.end(),
              [&, c](const string &s) { os << s << c; });
    // os explicitly captured by reference; c implicitly captured by value
    for_each(words.begin(), words.end(),
              [=, &os](const string &s) { os << s << c; });
}
void fcn3()
{
    size_t v1 = 42; // local variable
    // f can change the value of the variables it captures
    auto f = [v1] () mutable { return ++v1; };
    v1 = 0;
    auto j = f(); // j is 43
}
void fcn4()
{
    size_t v1 = 42; // local variable
    // v1 is a reference to a non const variable
    // we can change that variable through the reference inside f2
    auto f2 = [&v1] { return ++v1; };
    v1 = 0;
    auto j = f2(); // j is 1
}
transform(vi.begin(), vi.end(), vi.begin(),
          [] (int i) { return i < 0 ? -i : i; });

```

```

// error: cannot deduce the return type for the lambda
transform(vi.begin(), vi.end(), vi.begin(),
           [] (int i) { if (i < 0) return -i; else return i; });
transform(vi.begin(), vi.end(), vi.begin(),
           [] (int i) -> int
               { if (i < 0) return -i; else return i; });
bool check_size(const string &s, string::size_type sz)
{
    return s.size() >= sz;
}
auto newCallable = bind(callable, arg_list);
// check6 is a callable object that takes one argument of type string
// and calls check_size on its given string and the value 6
auto check6 = bind(check_size, _1, 6);
string s = "hello";
bool b1 = check6(s); // check6(s) calls check_size(s, 6)
auto wc = find_if(words.begin(), words.end(),
                   [sz] (const string &a)
auto wc = find_if(words.begin(), words.end(),
                   bind(check_size, _1, sz));
using std::placeholders::_1;
using namespace namespace_name;
using namespace std::placeholders;
// g is a callable object that takes two arguments
auto g = bind(f, a, b, _2, c, _1);
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
// sort on word length, longest to shortest
sort(words.begin(), words.end(), bind(isShorter, _2, _1));
// os is a local variable referring to an output stream
// c is a local variable of type char
for_each(words.begin(), words.end(),
          [&os, c](const string &s) { os << s << c; });
ostream &print(ostream &os, const string &s, char c)
{
    return os << s << c;
}
// error: cannot copy os
for_each(words.begin(), words.end(), bind(print, os, _1, ' '));
for_each(words.begin(), words.end(),
          bind(print, ref(os), _1, ' '));
it = c.insert(it, val); // it points to the newly added element
++it; // increment it so that it denotes the same element as before
list<int> lst = {1,2,3,4};
list<int> lst2, lst3; // empty lists
// after copy completes, lst2 contains 4 3 2 1
copy(lst.cbegin(), lst.cend(), front_inserter(lst2));
// after copy completes, lst3 contains 1 2 3 4
copy(lst.cbegin(), lst.cend(), inserter(lst3, lst3.begin()));
istream_iterator<int> int_it(cin); // reads ints from cin
istream_iterator<int> int_eof; // end iterator value
ifstream in("afile");
istream_iterator<string> str_it(in); // reads strings from "afile"

```

```

istream_iterator<int> in_iter(cin); // read ints from cin
istream_iterator<int> eof; // istream "end" iterator
while (in_iter != eof) // while there's valid input to read
    // postfix increment reads the stream and returns the old value of the iterator
    // we dereference that iterator to get the previous value read from the stream
    vec.push_back(*in_iter++);
istream_iterator<int> in_iter(cin), eof; // read ints from cin
vector<int> vec(in_iter, eof); // construct vec from an iterator range
istream_iterator<int> in(cin), eof;
    cout << accumulate(in, eof, 0) << endl;
ostream_iterator<int> out_iter(cout, " ");
for (auto e : vec)
    *out_iter++ = e; // the assignment writes this element to cout
cout << endl;
for (auto e : vec)
    out_iter = e; // the assignment writes this element to cout
cout << endl;
copy(vec.begin(), vec.end(), out_iter);
cout << endl;
istream_iterator<Sales_item> item_iter(cin), eof;
ostream_iterator<Sales_item> out_iter(cout, "\n");
// store the first transaction in sum and read the next record
Sales_item sum = *item_iter++;
while (item_iter != eof) {
    // if the current transaction (which is stored in item_iter) has the same ISBN
    if (item_iter->isbn() == sum.isbn())
        sum += *item_iter++; // add it to sum and read the next transaction
    else {
        out_iter = sum; // write the current sum
        sum = *item_iter++; // read the next transaction
    }
}
out_iter = sum; // remember to print the last set of records
// store the first transaction in sum and read the next record
Sales_item sum = *item_iter++;
vector<int> vec = {0,1,2,3,4,5,6,7,8,9};
// reverse iterator of vector from back to front
for (auto r_iter = vec.crbegin(); // binds r_iter to the last element
     r_iter != vec.crend(); // crend refers 1 before 1st element
     ++r_iter) // decrements the iterator one element
    cout << *r_iter << endl; // prints 9, 8, 7, ... 0
sort(vec.begin(), vec.end()); // sorts vec in "normal" order
// sorts in reverse: puts the smallest element at the end of vec
sort(vec.rbegin(), vec.rend());
// find the first element in a comma-separated list
auto comma = find(line.cbegin(), line.cend(), ',');
cout << string(line.cbegin(), comma) << endl;
// find the last element in a comma-separated list
auto rcomma = find(line.crbegin(), line.crend(), ',');
// WRONG: will generate the word in reverse order
cout << string(line.crbegin(), rcomma) << endl;
// ok: get a forward iterator and read to the end of line
cout << string(rcomma.base(), line.cend()) << endl;

```



```

// define a vector with 20 elements, holding two copies of each number from 0 to 9
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i);
    ivec.push_back(i); // duplicate copies of each number
}
// iset holds unique elements from ivec; miset holds all 20 elements
set<int> iset(ivec.cbegin(), ivec.cend());
multiset<int> miset(ivec.cbegin(), ivec.cend());
cout << ivec.size() << endl; // prints 20
cout << iset.size() << endl; // prints 10
cout << miset.size() << endl; // prints 20
bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() < rhs.isbn();
}
// bookstore can have several transactions with the same ISBN
// elements in bookstore will be in ISBN order
multiset<Sales_data, decltype(compareIsbn)*>
bookstore(compareIsbn);
pair<string, string> anon; // holds two strings
pair<string, size_t> word_count; // holds a string and an size_t
pair<string, vector<int>> line; // holds string and vector<int>
pair<string, string> author{"James", "Joyce"};
// print the results
cout << w.first << " occurs " << w.second
    << ((w.second > 1) ? " times" : " time") << endl;
pair<string, int>
process(vector<string> &v)
{
    // process v
    if (!v.empty())
        return {v.back(), v.back().size()}; // list initialize
    else
        return pair<string, int>(); // explicitly constructed return value
}
if (!v.empty())
    return pair<string, int>(v.back(), v.back().size());
if (!v.empty())
    return make_pair(v.back(), v.back().size());
set<string>::value_type v1; // v1 is a string
set<string>::key_type v2; // v2 is a string
map<string, int>::value_type v3; // v3 is a pair<const string, int>
map<string, int>::key_type v4; // v4 is a string
map<string, int>::mapped_type v5; // v5 is an int
// get an iterator to an element in word_count
auto map_it = word_count.begin();
// *map_it is a reference to a pair<const string, size_t> object
cout << map_it->first; // prints the key for this element
cout << " " << map_it->second; // prints the value of the element
map_it->first = "new key"; // error: key is const
++map_it->second; // ok: we can change the value through an iterator

```

```

set<int> iset = {0,1,2,3,4,5,6,7,8,9};
set<int>::iterator set_it = iset.begin();
if (set_it != iset.end()) {
    *set_it = 42; // error: keys in a set are read-only
    cout << *set_it << endl; // ok: can read the key
}
// get an iterator positioned on the first element
auto map_it = word_count.cbegin();
// compare the current iterator to the off-the-end iterator
while (map_it != word_count.cend()) {
    // dereference the iterator to print the element key-value pairs
    cout << map_it->first << " occurs "
        << map_it->second << " times" << endl;
    ++map_it; // increment the iterator to denote the next element
}
copy(v.begin(), v.end(), inserter(c, c.end()));
copy(v.begin(), v.end(), back_inserter(c));
copy(c.begin(), c.end(), inserter(v, v.end()));
copy(c.begin(), c.end(), back_inserter(v));
vector<int> ivec = {2,4,6,8,2,4,6,8}; // ivec has eight elements
set<int> set2; // empty set
set2.insert(ivec.cbegin(), ivec.cend()); // set2 has four elements
set2.insert({1,3,5,7,1,3,5,7}); // set2 now has eight elements
// four ways to add word to word_count
word_count.insert({word, 1});
word_count.insert(make_pair(word, 1));
word_count.insert(pair<string, size_t>(word, 1));
word_count.insert(map<string, size_t>::value_type(word, 1));
    map<string, size_t>::value_type(s, 1)
// more verbose way to count number of times each word occurs in the input
map<string, size_t> word_count; // empty map from string to size_t
string word;
while (cin >> word) {
    // inserts an element with key equal to word and value 1;
    // if word is already in word_count, insert does nothing
    auto ret = word_count.insert({word, 1});
    if (!ret.second) // word was already in word_count
        ++ret.first->second; // increment the counter
}
    ++((ret.first)->second); // equivalent expression
pair<map<string, size_t>::iterator, bool> ret =
    word_count.insert(make_pair(word, 1));
multimap<string, string> authors;
// adds the first element with the key Barth, John
authors.insert({"Barth, John", "Sot-Weed Factor"});
// ok: adds the second element with the key Barth, John
authors.insert({"Barth, John", "Lost in the Funhouse"});
    while (cin >> word)
        ++word_count.insert({word, 0}).first->second;

```

```

// erase on a key returns the number of elements removed
if (word_count.erase(removal_word))
    cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";
    auto cnt = authors.erase("Barth, John");
map <string, size_t> word_count; // empty map

// insert a value-initialized element with key Anna; then assign 1 to its value
word_count["Anna"] = 1;
cout << word_count["Anna"]; // fetch the element indexed by Anna; prints 1
++word_count["Anna"]; // fetch the element and add 1 to it
cout << word_count["Anna"]; // fetch the element and print it; prints 2
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
iset.find(1); // returns an iterator that refers to the element with key == 1
iset.find(11); // returns the iterator == iset.end()
iset.count(1); // returns 1
iset.count(11); // returns 0
if (word_count.find("foobar") == word_count.end())
    cout << "foobar is not in the map" << endl;
string search_item("Alain de Botton"); // author we'll look for
auto entries = authors.count(search_item); // number of elements
auto iter = authors.find(search_item); // first entry for this author
// loop through the number of entries there are for this author
while(entries) {
    cout << iter->second << endl; // print each title
    ++iter; // advance to the next title
    --entries; // keep track of how many we've printed
}
// definitions of authors and search_items above
// beg and end denote the range of elements for this author
for (auto beg = authors.lower_bound(search_item),
     end = authors.upper_bound(search_item);
     beg != end; ++beg)
    cout << beg->second << endl; // print each title
// definitions of authors and search_items above
// pos holds iterators that denote the range of elements for this key
for (auto pos = authors.equal_range(search_item);
     pos.first != pos.second; ++pos.first)
    cout << pos.first->second << endl; // print each title

```

```

void word_transform(ifstream &map_file, ifstream &input)
{
    auto trans_map = buildMap(map_file); // store the transformations
    string text;                      // hold each line from the input
    while (getline(input, text)) {     // read a line of input
        istringstream stream(text);   // read each word
        string word;
        bool firstword = true;       // controls whether a space is printed
        while (stream >> word) {
            if (firstword)
                firstword = false;
            else
                cout << " "; // print a space between words
            // transform returns its first argument or its transformation
            cout << transform(word, trans_map); // print the output
        }
        cout << endl;               // done with this line of input
    }
}
map<string, string> buildMap(ifstream &map_file)
{
    map<string, string> trans_map; // holds the transformations
    string key;                  // a word to transform
    string value;                // phrase to use instead
    // read the first word into key and the rest of the line into value
    while (map_file >> key && getline(map_file, value))
        if (value.size() > 1) // check that there is a transformation
            trans_map[key] = value.substr(1); // skip leading space
        else
            throw runtime_error("no rule for " + key);
    return trans_map;
}
const string &
transform(const string &s, const map<string, string> &m)
{
    // the actual map work; this part is the heart of the program
    auto map_it = m.find(s);
    // if this word is in the transformation map
    if (map_it != m.cend())
        return map_it->second; // use the replacement word
    else
        return s;              // otherwise return the original unchanged
}
    trans_map[key] = value.substr(1);
    as trans_map.insert({key, value.substr(1)})?

```

```

// count occurrences, but the words won't be in alphabetical order
unordered_map<string, size_t> word_count;
string word;
while (cin >> word)
    ++word_count[word]; // fetch and increment the counter for word
for (const auto &w : word_count) // for each element in the map
    // print the results
    cout << w.first << " occurs " << w.second
        << ((w.second > 1) ? " times" : " time") << endl;
size_t hasher(const Sales_data &sd)
{
    return hash<string>()(sd.isbn());
}
bool eqOp(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn();
}
using SD_multiset = unordered_multiset<Sales_data,
                                         decltype(hasher)*, decltype(eqOp)*>;
// arguments are the bucket size and pointers to the hash function and equality operator
SD_multiset bookstore(42, hasher, eqOp);
// use FooHash to generate the hash code; Foo must have an == operator
unordered_set<Foo, decltype(FooHash)*> fooSet(10, FooHash);
shared_ptr<string> p1; // shared_ptr that can point at a string

shared_ptr<list<int>> p2; // shared_ptr that can point at a list of ints
// if p1 is not null, check whether it's the empty string
if (p1 && p1->empty())
    *p1 = "hi"; // if so, dereference p1 to assign a new value to that string
    // shared_ptr that points to an int with value 42
shared_ptr<int> p3 = make_shared<int>(42);
// p4 points to a string with value 9999999999
shared_ptr<string> p4 = make_shared<string>(10, '9');
// p5 points to an int that is value initialized (§ 3.3.1 (p. 98)) to 0
shared_ptr<int> p5 = make_shared<int>();
// p6 points to a dynamically allocated, empty vector<string>
auto p6 = make_shared<vector<string>>();
auto p = make_shared<int>(42); // object to which p points has one user
auto q(p); // p and q point to the same object
// object to which p and q point has two users
auto r = make_shared<int>(42); // int to which r points has one user
r = q; // assign to r, making it point to a different address
// increase the use count for the object to which q points
// reduce the use count of the object to which r had pointed
// the object r had pointed to has no users; that object is automatically freed
// factory returns a shared_ptr pointing to a dynamically allocated object
shared_ptr<Foo> factory(T arg)
{
    // process arg as appropriate
    // shared_ptr will take care of deleting this memory
    return make_shared<Foo>(arg);
}

```

```

void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
} // p goes out of scope; the memory to which p points is automatically freed
shared_ptr<Foo> use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
    return p; // reference count is incremented when we return p
} // p goes out of scope; the memory to which p points is not freed
vector<string> v1; // empty vector
{ // new scope
    vector<string> v2 = {"a", "an", "the"};
    v1 = v2; // copies the elements from v2 into v1
} // v2 is destroyed, which destroys the elements in v2
// v1 has three elements, which are copies of the ones originally in v2
Blob<string> b1; // empty Blob
{ // new scope
    Blob<string> b2 = {"a", "an", "the"};
    b1 = b2; // b1 and b2 share the same elements
} // b2 is destroyed, but the elements in b2 must not be destroyed
// b1 points to the elements originally created in b2
class StrBlob {
public:
    typedef std::vector<std::string>::size_type size_type;
    StrBlob();
    StrBlob(std::initializer_list<std::string> il);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // add and remove elements
    void push_back(const std::string &t) { data->push_back(t); }
    void pop_back();
    // element access
    std::string& front();
    std::string& back();
private:
    std::shared_ptr<std::vector<std::string>> data;
    // throws msg if data[i] isn't valid
    void check(size_type i, const std::string &msg) const;
};
StrBlob::StrBlob(): data(make_shared<vector<string>>()) { }
StrBlob::StrBlob(initializer_list<string> il):
    data(make_shared<vector<string>>(il)) { }
void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}

```

```

        string& StrBlob::front()
        {
            // if the vector is empty, check will throw
            check(0, "front on empty StrBlob");
            return data->front();
        }
        string& StrBlob::back()
        {
            check(0, "back on empty StrBlob");
            return data->back();
        }
        void StrBlob::pop_back()
        {
            check(0, "pop_back on empty StrBlob");
            data->pop_back();
        }
        StrBlob b1;
        {
            StrBlob b2 = {"a", "an", "the"};
            b1 = b2;
            b2.push_back("about");
        }
        int *pi = new int;           // pi points to a dynamically allocated,
                                    // unnamed, uninitialized int
        string *ps = new string;   // initialized to empty string
        int *pi = new int;           // pi points to an uninitialized int
        int *pi = new int(1024);    // object to which pi points has value 1024
        string *ps = new string(10, '9'); // *ps is "9999999999"
        // vector with ten elements with values from 0 to 9
        vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
        string *ps1 = new string; // default initialized to the empty string
        string *ps = new string(); // value initialized to the empty string
        int *pi1 = new int;         // default initialized; *pi1 is undefined
        int *pi2 = new int();       // value initialized to 0; *pi2 is 0
        auto p1 = new auto(obj);   // p points to an object of the type of obj
                                    // that object is initialized from obj
        auto p2 = new auto{a,b,c}; // error: must use parentheses for the initializer
                                    // allocate and initialize a const int
        const int *pci = new const int(1024);
                                    // allocate a default-initialized const empty string
        const string *pcs = new const string;
        // if allocation fails, new returns a null pointer
        int *p1 = new int; // if allocation fails, new throws std::bad_alloc
        int *p2 = new (nothrow) int; // if allocation fails, new returns a null pointer
        delete p;          // p must point to a dynamically allocated object or be null
        int i, *pi1 = &i, *pi2 = nullptr;
        double *pd = new double(33), *pd2 = pd;

        delete i; // error: i is not a pointer
        delete pi1; // undefined: pi1 refers to a local
        delete pd; // ok
        delete pd2; // undefined: the memory pointed to by pd2 was already freed
        delete pi2; // ok: it is always ok to delete a null pointer
    
```

```

        const int *pci = new const int(1024);
        delete pci; // ok: deletes a const object
// factory returns a pointer to a dynamically allocated object
Foo* factory(T arg)
{
    // process arg as appropriate
    return new Foo(arg); // caller is responsible for deleting this memory
}
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p but do not delete it
} // p goes out of scope, but the memory to which p points is not freed!
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p
    delete p; // remember to free the memory now that we no longer need it
}
Foo* use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p
    return p; // caller must delete the memory
}
int *p(new int(42)); // p points to dynamic memory
auto q = p; // p and q point to the same memory
delete p; // invalidates both p and q
p = nullptr; // indicates that p is no longer bound to an object
int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;
shared_ptr<double> p1; // shared_ptr that can point at a double
shared_ptr<int> p2(new int(42)); // p2 points to an int with value 42
shared_ptr<int> p1 = new int(1024); // error: must use direct initialization
shared_ptr<int> p2(new int(1024)); // ok: uses direct initialization
shared_ptr<int> clone(int p) {
    return new int(p); // error: implicit conversion to shared_ptr<int>
}
shared_ptr<int> clone(int p) {
    // ok: explicitly create a shared_ptr<int> from int*
    return shared_ptr<int>(new int(p));
}
// ptr is created and initialized when process is called
void process(shared_ptr<int> ptr)
{
    // use ptr
} // ptr goes out of scope and is destroyed
shared_ptr<int> p(new int(42)); // reference count is 1
process(p); // copying p increments its count; in process the reference count is 2
int i = *p; // ok: reference count is 1

```

```

int *x(new int(1024)); // dangerous: x is a plain pointer, not a smart pointer
process(x); // error: cannot convert int* to shared_ptr<int>
process(shared_ptr<int>(x)); // legal, but the memory will be deleted!
int j = *x; // undefined: x is a dangling pointer!
shared_ptr<int> p(new int(42)); // reference count is 1
int *q = p.get(); // ok: but don't use q in any way that might delete its pointer
{ // new block
// undefined: two independent shared_ptrs point to the same memory
shared_ptr<int>(q);
} // block ends, q is destroyed, and the memory to which q points is freed
int foo = *p; // undefined: the memory to which p points was freed
p = new int(1024); // error: cannot assign a pointer to a shared_ptr
p.reset(new int(1024)); // ok: p points to a new object
if (!p.unique())
    p.reset(new string(*p)); // we aren't alone; allocate a new copy
*p += newVal; // now that we know we're the only pointer, okay to change this object
    shared_ptr<int> p(new int(42));
    process(shared_ptr<int>(p));
    process(shared_ptr<int>(p.get()));
    auto sp = make_shared<int>();
    auto p = sp.get();
    delete p;
void f()
{
    shared_ptr<int> sp(new int(42)); // allocate a new object
    // code that throws an exception that is not caught inside f
}
// shared_ptr freed automatically when the function ends
void f()
{
    int *ip = new int(42); // dynamically allocate a new object
    // code that throws an exception that is not caught inside f
    delete ip; // free the memory before exiting
}
struct destination; // represents what we are connecting to
struct connection; // information needed to use the connection
connection connect(destination*); // open the connection
void disconnect(connection); // close the given connection
void f(destination &d /* other parameters */)
{
    // get a connection; must remember to close it when done
    connection c = connect(&d);
    // use the connection
    // if we forget to call disconnect before exiting f, there will be no way to close c
}
void end_connection(connection *p) { disconnect(*p); }
void f(destination &d /* other parameters */)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // use the connection
    // when f exits, even if by an exception, the connection will be properly closed
}

```

```

unique_ptr<double> p1; // unique_ptr that can point at a double
unique_ptr<int> p2(new int(42)); // p2 points to int with value 42
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1); // error: no copy for unique_ptr
unique_ptr<string> p3;
p3 = p2; // error: no assign for unique_ptr
// transfers ownership from p1 (which points to the string Stegosaurus) to p2
unique_ptr<string> p2(p1.release()); // release makes p1 null
unique_ptr<string> p3(new string("Trex"));
// transfers ownership from p3 to p2
p2.reset(p3.release()); // reset deletes the memory to which p2 had pointed
p2.release(); // WRONG: p2 won't free the memory and we've lost the pointer
auto p = p2.release(); // ok, but we must remember to delete(p)
unique_ptr<int> clone(int p) {
    // ok: explicitly create a unique_ptr<int> from int*
    return unique_ptr<int>(new int(p));
}
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int (p));
    // ...
    return ret;
}
// p points to an object of type objT and uses an object of type delT to free that object
// it will call an object named fcn of type delT
unique_ptr<objT, delT> p (new objT, fcn);
void f(destination &d /* other needed parameters */)
{
    connection c = connect(&d); // open the connection
    // when p is destroyed, the connection will be closed
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
    // use the connection
    // when f exits, even if by an exception, the connection will be properly closed
}
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
auto p = make_shared<int>(42);
weak_ptr<int> wp(p); // wp weakly shares with p; use count in p is unchanged
if (shared_ptr<int> np = wp.lock()) { // true if np is not null
    // inside the if, np shares its object with p
}

```

```

// StrBlobPtr throws an exception on attempts to access a nonexistent element
class StrBlobPtr {
public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0):
        wptr(a.data), curr(sz) { }
    std::string& deref() const;
    StrBlobPtr& incr();           // prefix version
private:
    // check returns a shared_ptr to the vector if the check succeeds
    std::shared_ptr<std::vector<std::string>>
        check(std::size_t, const std::string&) const;
    // store a weak_ptr, which means the underlying vector might be destroyed
    std::weak_ptr<std::vector<std::string>> wptr;
    std::size_t curr;            // current position within the array
};
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i, const std::string &msg) const
{
    auto ret = wptr.lock();     // is the vector still around?
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // otherwise, return a shared_ptr to the vector
}
std::string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr]; // (*p) is the vector to which this object points
}
// prefix: return a reference to the incremented object
StrBlobPtr& StrBlobPtr::incr()
{
    // if curr already points past the end of the container, can't increment it
    check(curr, "increment past end of StrBlobPtr");
    ++curr;             // advance the current state
    return *this;
}
// forward declaration needed for friend declaration in StrBlob
class StrBlobPtr;
class StrBlob {
    friend class StrBlobPtr;
    // other members as in § 12.1.1 (p. 456)
    // return StrBlobPtr to the first and one past the last elements
    StrBlobPtr begin() { return StrBlobPtr(*this); }
    StrBlobPtr end()
        { auto ret = StrBlobPtr(*this, data->size());
          return ret; }
};
std::string& deref() const
{ return (*check(curr, "dereference past end"))[curr]; }

```

```

// call get_size to determine how many ints to allocate
int *pia = new int[get_size()]; // pia points to the first of these ints
typedef int arrT[42]; // arrT names the type array of 42 ints
int *p = new arrT; // allocates an array of 42 ints; p points to the first one
int *pia = new int[10]; // block of ten uninitialized ints
int *pia2 = new int[10](); // block of ten ints value initialized to 0
string *psa = new string[10]; // block of ten empty strings
string *psa2 = new string[10](); // block of ten empty strings
// block of ten ints each initialized from the corresponding initializer
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
// block of ten strings; the first four are initialized from the given initializers
// remaining elements are value initialized
string *psa3 = new string[10>{"a", "an", "the", string(3,'x')};
size_t n = get_size(); // get_size returns the number of elements needed
int* p = new int[n]; // allocate an array to hold the elements
for (int* q = p; q != p + n; ++q)
    /* process the array */ ;
char arr[0]; // error: cannot define a zero-length array
char *cp = new char[0]; // ok: but cp can't be dereferenced
delete p; // p must point to a dynamically allocated object or be null
delete [] pa; // pa must point to a dynamically allocated array or be null
typedef int arrT[42]; // arrT names the type array of 42 ints
int *p = new arrT; // allocates an array of 42 ints; p points to the first one
delete [] p; // brackets are necessary because we allocated an array
// up points to an array of ten uninitialized ints
unique_ptr<int[]> up(new int[10]);
up.release(); // automatically uses delete [] to destroy its pointer
    for (size_t i = 0; i != 10; ++i)
        up[i] = i; // assign a new value to each of the elements
// to use a shared_ptr we must supply a deleter
shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });
sp.reset(); // uses the lambda we supplied that uses delete [] to free the array
// shared_ptrs don't have subscript operator and don't support pointer arithmetic
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i; // use get to get a built-in pointer
string *const p = new string[n]; // construct n empty strings
string s;
string *q = p; // q points to the first string
while (cin >> s && q != p + n)
    *q++ = s; // assign a new value to *q
const size_t size = q - p; // remember how many strings we read
// use the array
delete[] p; // p points to an array; must remember to use delete []
allocator<string> alloc; // object that can allocate strings
auto const p = alloc.allocate(n); // allocate n unconstructed strings
    auto q = p; // q will point to one past the last constructed element
    alloc.construct(q++); // *q is the empty string
    alloc.construct(q++, 10, 'c'); // *q is cccccccccc
    alloc.construct(q++, "hi"); // *q is hi!
cout << *p << endl; // ok: uses the string output operator
cout << *q << endl; // disaster: q points to unconstructed memory!
while (q != p)
    alloc.destroy(--q); // free the strings we actually allocated

```

```

// allocate twice as many elements as vi holds
auto p = alloc.allocate(vi.size() * 2);
// construct elements starting at p as copies of elements in vi
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
// initialize the remaining elements to 42
uninitialized_fill_n(q, vi.size(), 42);
element occurs 112 times
    (line 36) A set element contains only a key;
    (line 158) operator creates a new element
    (line 160) Regardless of whether the element
    (line 168) When we fetch an element from a map, we
    (line 214) If the element is not found, find returns
void runQueries(ifstream &infile)
{
    // infile is an ifstream that is the file we want to query
    TextQuery tq(infile); // store the file and build the query map
    // iterate with the user: prompt for a word to find and print results
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // stop if we hit end-of-file on the input or if a 'q' is entered
        if (!(cin >> s) || s == "q") break;
        // run the query and print the results
        print(cout, tq.query(s)) << endl;
    }
}
class QueryResult; // declaration needed for return type in the query function
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // input file
    // map of each word to the set of the lines in which that word appears
    std::map<std::string,
            std::shared_ptr<std::set<line_no>>> wm;
};
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
map<string, shared_ptr<set<line_no>>> wm;

```

```

// read the input file and build the map of lines to line numbers
TextQuery::TextQuery(ifstream &is) : file(new vector<string>)
{
    string text;
    while (getline(is, text)) {           // for each line in the file
        file->push_back(text);          // remember this line of text
        int n = file->size() - 1;        // the current line number
        istringstream line(text);         // separate the line into words
        string word;
        while (line >> word) {          // for each word in that line
            // if word isn't already in wm, subscripting adds a new entry
            auto &lines = wm[word]; // lines is a shared_ptr
            if (!lines) // that pointer is null the first time we see word
                lines.reset(new set<line_no>); // allocate a new set
            lines->insert(n);           // insert this line number
        }
    }
}
class QueryResult {
friend std::ostream& print(std::ostream&, const QueryResult&);
public:
    QueryResult(string s,
                std::shared_ptr<std::set<line_no>> p,
                std::shared_ptr<std::vector<std::string>> f) :
        sought(s), lines(p), file(f) { }
private:
    string sought; // word this query represents
    std::shared_ptr<std::set<line_no>> lines; // lines it's on
    std::shared_ptr<std::vector<std::string>> file; // input file
};
QueryResult
TextQuery::query(const string &sought) const
{
    // we'll return a pointer to this set if we don't find sought
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // use find and not a subscript to avoid adding words to wm!
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // not found
    else
        return QueryResult(sought, loc->second, file);
}

```

```

ostream &print(ostream & os, const QueryResult &qr)
{
    // if the word was found, print the count and all occurrences
    os << qr.sought << " occurs " << qr.lines->size() << " "
        << make_plural(qr.lines->size(), "time", "s") << endl;
    // print each line in which the word appeared
    for (auto num : *qr.lines) // for every element in the set
        // don't confound the user with text lines starting at 0
        os << "\t(line " << num + 1 << ") "
            << *(qr.file->begin() + num) << endl;
    return os;
}

class Foo {
public:
    Foo();                      // default constructor
    Foo(const Foo&);          // copy constructor
    // ...
};

class Sales_data {
public:
    // other members and constructors as before
    // declaration equivalent to the synthesized copy constructor
    Sales_data(const Sales_data&);

private:
    std::string bookNo;
    int units_sold = 0;
    double revenue = 0.0;
};

// equivalent to the copy constructor that would be synthesized for Sales_data
Sales_data::Sales_data(const Sales_data &orig):
    bookNo(orig.bookNo),           // uses the string copy constructor
    units_sold(orig.units_sold),   // copies orig.units_sold
    revenue(orig.revenue)         // copies orig.revenue
{                                // empty body
    string dots(10, '.');        // direct initialization
    string s(dots);              // direct initialization
    string s2 = dots;             // copy initialization
    string null_book = "9-999-99999-9"; // copy initialization
    string nines = string(100, '9'); // copy initialization
vector<int> v1(10); // ok: direct initialization
vector<int> v2 = 10; // error: constructor that takes a size is explicit
void f(vector<int>); // f's parameter is copy initialized
f(10); // error: can't use an explicit constructor to copy an argument
f(vector<int>(10)); // ok: directly construct a temporary vector from an int
    string null_book = "9-999-99999-9"; // copy initialization
    string null_book("9-999-99999-9"); // compiler omits the copy constructor
    Sales_data::Sales_data(Sales_data rhs);
}

```

```

    Point global;
    Point foo_bar(Point arg)
    {
        Point local = arg, *heap = new Point(global);
        *heap = local;
        Point pa[ 4 ] = { local, *heap };
        return *heap;
    }
    class HasPtr {
public:
    HasPtr(const std::string &s = std::string()): ps(new std::string(s)), i(0) { }
private:
    std::string *ps;
    int      i;
};
Sales_data trans, accum;
trans = accum; // uses the Sales_data copy-assignment operator
class Foo {
public:
    Foo& operator=(const Foo&); // assignment operator
    // ...
};
// equivalent to the synthesized copy-assignment operator
Sales_data&
Sales_data::operator=(const Sales_data &rhs)
{
    bookNo = rhs.bookNo;           // calls the string::operator=
    units_sold = rhs.units_sold;  // uses the built-in int assignment
    revenue = rhs.revenue;        // uses the built-in double assignment
    return *this;                 // return a reference to this object
}
// new scope
// p and p2 point to dynamically allocated objects
Sales_data *p = new Sales_data;           // p is a built-in pointer
auto p2 = make_shared<Sales_data>(); // p2 is a shared_ptr
Sales_data item(*p);                  // copy constructor copies *p into item
vector<Sales_data> vec;             // local object
vec.push_back(*p2);                  // copies the object to which p2 points
delete p;                           // destructor called on the object pointed to by p
} // exit local scope; destructor called on item, p2, and vec
// destroying p2 decrements its use count; if the count goes to 0, the object is freed
// destroying vec destroys the elements in vec
class Sales_data {
public:
    // no work to do other than destroying the members, which happens automatically
    ~Sales_data() { }
    // other members as before
};

```

```

        bool fcn(const Sales_data *trans, Sales_data accum)
    {
        Sales_data item1(*trans), item2(accum);
        return item1.isbn() != item2.isbn();
    }
    struct X {
        X() {std::cout << "X()" << std::endl;}
        X(const X&) {std::cout << "X(const X&)" << std::endl;}
    };
    class HasPtr {
public:
    HasPtr(const std::string &s = std::string()): ps(new std::string(s)), i(0) { }
    ~HasPtr() { delete ps; }
    // WRONG: HasPtr needs a copy constructor and copy-assignment operator
    // other members as before
};

    HasPtr f(HasPtr hp) // HasPtr passed by value, so it is copied
{
    HasPtr ret = hp; // copies the given HasPtr
    // process ret
    return ret; // ret and hp are destroyed
}
HasPtr p("some values");
f(p); // when f completes, the memory to which p.ps points is freed
HasPtr q(p); // now both p and q point to invalid memory!
void f (numbered s) { cout << s.mysn << endl; }
    numbered a, b = a, c = b;
    f(a); f(b); f(c);

class Sales_data {
public:
    // copy control; use defaults
    Sales_data() = default;
    Sales_data(const Sales_data&) = default;
    Sales_data& operator=(const Sales_data &);
    ~Sales_data() = default;
    // other members as before
};

Sales_data& Sales_data::operator=(const Sales_data&) = default;
struct NoCopy {
    NoCopy() = default; // use the synthesized default constructor
    NoCopy(const NoCopy&) = delete; // no copy
    NoCopy &operator=(const NoCopy&) = delete; // no assignment
    ~NoCopy() = default; // use the synthesized destructor
    // other members
};

    struct NoDtor {
        NoDtor() = default; // use the synthesized default constructor
        ~NoDtor() = delete; // we can't destroy objects of type NoDtor
    };
    NoDtor nd; // error: NoDtor destructor is deleted
    NoDtor *p = new NoDtor(); // ok: but we can't delete p
    delete p; // error: NoDtor destructor is deleted

```

```

class PrivateCopy {
    // no access specifier; following members are private by default; see § 7.2 (p. 268)
    // copy control is private and so is inaccessible to ordinary user code
    PrivateCopy(const PrivateCopy&);
    PrivateCopy &operator=(const PrivateCopy&);
    // other members
public:
    PrivateCopy() = default; // use the synthesized default constructor
    ~PrivateCopy(); // users can define objects of this type but not copy them
};

class HasPtr {
public:
    HasPtr(const std::string &s = std::string()): ps(new std::string(s)), i(0) { }
    // each HasPtr has its own copy of the string to which ps points
    HasPtr(const HasPtr &p):
        ps(new std::string(*p.ps)), i(p.i) { }
    HasPtr& operator=(const HasPtr &);

    ~HasPtr() { delete ps; }

private:
    std::string *ps;
    int i;
};

HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    auto newp = new std::string(*rhs.ps); // copy the underlying string
    delete ps; // free the old memory
    ps = newp; // copy data from rhs into this object
    i = rhs.i;
    return *this; // return this object
}
// WRONG way to write an assignment operator!
HasPtr&
HasPtr::operator=(const HasPtr &rhs)
{
    delete ps; // frees the string to which this object points
    // if rhs and *this are the same object, we're copying from deleted memory!
    ps = new std::string(*rhs.ps);
    i = rhs.i;
    return *this;
}

HasPtr p1("Hiya!");
HasPtr p2(p1); // p1 and p2 point to the same string
HasPtr p3(p1); // p1, p2, and p3 all point to the same string

```

```

class HasPtr {
public:
    // constructor allocates a new string and a new counter, which it sets to 1
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0), use(new std::size_t(1)) {}
    // copy constructor copies all three data members and increments the counter
    HasPtr(const HasPtr &p):
        ps(p.ps), i(p.i), use(p.use) { ++*use; }
    HasPtr& operator=(const HasPtr&);
    ~HasPtr();
private:
    std::string *ps;
    int i;
    std::size_t *use; // member to keep track of how many objects share *ps
};

HasPtr::~HasPtr()
{
    if (--*use == 0) { // if the reference count goes to 0
        delete ps; // delete the string
        delete use; // and the counter
    }
}

HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    ++*rhs.use; // increment the use count of the right-hand operand
    if (--*use == 0) { // then decrement this object's counter
        delete ps; // if no other users
        delete use; // free this object's allocated members
    }
    ps = rhs.ps; // copy data from rhs into this object
    i = rhs.i;
    use = rhs.use;
    return *this; // return this object
}

HasPtr temp = v1; // make a temporary copy of the value of v1
v1 = v2; // assign the value of v2 to v1
v2 = temp; // assign the saved value of v1 to v2
string *temp = v1.ps; // make a temporary copy of the pointer in v1.ps
v1.ps = v2.ps; // assign the pointer in v2.ps to v1.ps
v2.ps = temp; // assign the saved pointer in v1.ps to v2.ps
class HasPtr {
    friend void swap(HasPtr&, HasPtr&);
    // other members as in § 13.2.1 (p. 511)
};

inline
void swap(HasPtr &lhs, HasPtr &rhs)
{
    using std::swap;
    swap(lhs.ps, rhs.ps); // swap the pointers, not the string data
    swap(lhs.i, rhs.i); // swap the int members
}

```

```

void swap(Foo &lhs, Foo &rhs)
{
    // WRONG: this function uses the library version of swap, not the HasPtr version
    std::swap(lhs.h, rhs.h);
    // swap other members of type Foo
}
void swap(Foo &lhs, Foo &rhs)
{
    using std::swap;
    swap(lhs.h, rhs.h); // uses the HasPtr version of swap
    // swap other members of type Foo
}
// note rhs is passed by value, which means the HasPtr copy constructor
// copies the string in the right-hand operand into rhs
HasPtr& HasPtr::operator=(HasPtr rhs)
{
    // swap the contents of the left-hand operand with the local variable rhs
    swap(*this, rhs); // rhs now points to the memory this object had used
    return *this; // rhs is destroyed, which deletes the pointer in rhs
}
class Message {
    friend class Folder;
public:
    // folders is implicitly initialized to the empty set
    explicit Message(const std::string &str = "") :
        contents(str) { }

    // copy control to manage pointers to this Message
    Message(const Message&); // copy constructor
    Message& operator=(const Message&); // copy assignment
    ~Message(); // destructor

    // add/remove this Message from the specified Folder's set of messages
    void save(Folder&);
    void remove(Folder&);

private:
    std::string contents; // actual message text
    std::set<Folder*> folders; // Folders that have this Message

    // utility functions used by copy constructor, assignment, and destructor
    // add this Message to the Folders that point to the parameter
    void add_to_Folders(const Message&);
    // remove this Message from every Folder in folders
    void remove_from_Folders();
};


```

```

void Message::save(Folder &f)
{
    folders.insert(&f); // add the given Folder to our list of Folders
    f.addMsg(this); // add this Message to f's set of Messages
}

void Message::remove(Folder &f)
{
    folders.erase(&f); // take the given Folder out of our list of Folders
    f.remMsg(this); // remove this Message to f's set of Messages
}
// add this Message to Folders that point to m
void Message::add_to_Folders(const Message &m)
{
    for (auto f : m.folders) // for each Folder that holds m
        f->addMsg(this); // add a pointer to this Message to that Folder
}
Message::Message(const Message &m):
    contents(m.contents), folders(m.folders)
{
    add_to_Folders(m); // add this Message to the Folders that point to m
}
// remove this Message from the corresponding Folders
void Message::remove_from_Folders()
{
    for (auto f : folders) // for each pointer in folders
        f->remMsg(this); // remove this Message from that Folder
}
Message::~Message()
{
    remove_from_Folders();
}

Message& Message::operator=(const Message &rhs)
{
    // handle self-assignment by removing pointers before inserting them
    remove_from_Folders(); // update existing Folders
    contents = rhs.contents; // copy message contents from rhs
    folders = rhs.folders; // copy Folder pointers from rhs
    add_to_Folders(rhs); // add this Message to those Folders
    return *this;
}

```

```

void swap(Message &lhs, Message &rhs)
{
    using std::swap; // not strictly needed in this case, but good habit
    // remove pointers to each Message from their (original) respective Folders
    for (auto f: lhs.folders)
        f->remMsg(&lhs);
    for (auto f: rhs.folders)
        f->remMsg(&rhs);

    // swap the contents and Folder pointer sets
    swap(lhs.folders, rhs.folders); // uses swap(set&, set&)
    swap(lhs.contents, rhs.contents); // swap(string&, string&)

    // add pointers to each Message to their (new) respective Folders
    for (auto f: lhs.folders)
        f->addMsg(&lhs);
    for (auto f: rhs.folders)
        f->addMsg(&rhs);
}
// simplified implementation of the memory allocation strategy for a vector-like class
class StrVec {
public:
    StrVec() // the allocator member is default initialized
    elements(nullptr), first_free(nullptr), cap(nullptr) { }
    StrVec(const StrVec&); // copy constructor
    StrVec &operator=(const StrVec&); // copy assignment
    ~StrVec(); // destructor
    void push_back(const std::string&); // copy the element
    size_t size() const { return first_free - elements; }
    size_t capacity() const { return cap - elements; }
    std::string *begin() const { return elements; }
    std::string *end() const { return first_free; }
    // ...
private:
    std::allocator<std::string> alloc; // allocates the elements
    // used by the functions that add elements to the StrVec
    void chk_n_alloc()
    { if (size() == capacity()) reallocate(); }

    // utilities used by the copy constructor, assignment operator, and destructor
    std::pair<std::string*, std::string*> alloc_n_copy
    (const std::string*, const std::string*);
    void free(); // destroy the elements and free the space
    void reallocate(); // get more space and copy the existing elements
    std::string *elements; // pointer to the first element in the array
    std::string *first_free; // pointer to the first free element in the array
    std::string *cap; // pointer to one past the end of the array
};

void StrVec::push_back(const string& s)
{
    chk_n_alloc(); // ensure that there is room for another element
    // construct a copy of s in the element to which first_free points
    alloc.construct(first_free++, s);
}

```

```
pair<string*, string*>
StrVec::alloc_n_copy(const string *b, const string *e)
{
    // allocate space to hold as many elements as are in the range
    auto data = alloc.allocate(e - b);
    // initialize and return a pair constructed from data and
    // the value returned by uninitialized_copy
    return {data, uninitialized_copy(b, e, data)};
}
void StrVec::free()
{
    // may not pass deallocate a 0 pointer; if elements is 0, there's no work to do
    if (elements) {
        // destroy the old elements in reverse order
        for (auto p = first_free; p != elements; /* empty */)
            alloc.destroy(--p);
        alloc.deallocate(elements, cap - elements);
    }
}
StrVec::StrVec(const StrVec &s)
{
    // call alloc_n_copy to allocate exactly as many elements as in s
    auto newdata = alloc_n_copy(s.begin(), s.end());
    elements = newdata.first;
    first_free = cap = newdata.second;
}
StrVec::~StrVec() { free(); }
StrVec &StrVec::operator=(const StrVec &rhs)
{
    // call alloc_n_copy to allocate exactly as many elements as in rhs
    auto data = alloc_n_copy(rhs.begin(), rhs.end());
    free();
    elements = data.first;
    first_free = cap = data.second;
    return *this;
}
```

```

void StrVec::reallocate()
{
    // we'll allocate space for twice as many elements as the current size
    auto newcapacity = size() ? 2 * size() : 1;
    // allocate new memory
    auto newdata = alloc.allocate(newcapacity);
    // move the data from the old memory to the new
    auto dest = newdata; // points to the next free position in the new array
    auto elem = elements; // points to the next element in the old array
    for (size_t i = 0; i != size(); ++i)
        alloc.construct(dest++, std::move(*elem++));
    free(); // free the old space once we've moved the elements
    // update our data structure to point to the new elements
    elements = newdata;
    first_free = dest;
    cap = elements + newcapacity;
}
int i = 42;
int &r = i; // ok: r refers to i
int &&rr = i; // error: cannot bind an rvalue reference to an lvalue
int &r2 = i * 42; // error: i * 42 is an rvalue
const int &r3 = i * 42; // ok: we can bind a reference to const to an rvalue
int &&rr2 = i * 42; // ok: bind rr2 to the result of the multiplication
    int &&rr1 = 42; // ok: literals are rvalues
    int &&rr2 = rr1; // error: the expression rr1 is an lvalue!
    int &&rr3 = std::move(rr1); // ok
StrVec::StrVec(StrVec &s) noexcept // move won't throw any exceptions
    // member initializers take over the resources in s
    : elements(s.elements), first_free(s.first_free), cap(s.cap)
{
    // leave s in a state in which it is safe to run the destructor
    s.elements = s.first_free = s.cap = nullptr;
}
class StrVec {
public:
    StrVec(StrVec&&) noexcept; // move constructor
    // other members as before
};
StrVec::StrVec(StrVec &s) noexcept : /* member initializers */
{ /* constructor body */ }

```

```

StrVec &StrVec::operator=(StrVec &&rhs) noexcept
{
    // direct test for self-assignment
    if (this != &rhs) {
        free();                      // free existing elements
        elements = rhs.elements;    // take over resources from rhs
        first_free = rhs.first_free;
        cap = rhs.cap;
        // leave rhs in a destructible state
        rhs.elements = rhs.first_free = rhs.cap = nullptr;
    }
    return *this;
}
// the compiler will synthesize the move operations for X and hasX
struct X {
    int i;                      // built-in types can be moved
    std::string s; // string defines its own move operations
};
struct hasX {
    X mem; // X has synthesized move operations
};
X x, x2 = std::move(x);           // uses the synthesized move constructor
hasX hx, hx2 = std::move(hx); // uses the synthesized move constructor
// assume Y is a class that defines its own copy constructor but not a move constructor
struct hasY {
    hasY() = default;
    hasY(hasY&&) = default;
    Y mem; // hasY will have a deleted move constructor
};
hasY hy, hy2 = std::move(hy); // error: move constructor is deleted
StrVec v1, v2;
v1 = v2;                         // v2 is an lvalue; copy assignment
StrVec getVec(istream &); // getVec returns an rvalue
v2 = getVec(cin); // getVec(cin) is an rvalue; move assignment
class Foo {
public:
    Foo() = default;
    Foo(const Foo&); // copy constructor
    // other members, but Foo does not define a move constructor
};
Foo x;
Foo y(x); // copy constructor; x is an lvalue
Foo z(std::move(x)); // copy constructor, because there is no move constructor
class HasPtr {
public:
    // added move constructor
    HasPtr(HasPtr &&p) noexcept : ps(p.ps), i(p.i) { p.ps = 0; }
    // assignment operator is both the move- and copy-assignment operator
    HasPtr& operator=(HasPtr rhs)
        { swap(*this, rhs); return *this; }
    // other members as in § 13.2.1 (p. 511)
};

```

```

        hp = hp2; // hp2 is an lvalue; copy constructor used to copy hp2
        hp = std::move(hp2); // move constructor moves hp2
// move the Folder pointers from m to this Message
void Message::move_Folders(Message *m)
{
    folders = std::move(m->folders); // uses set move assignment
    for (auto f : folders) { // for each Folder
        f->remMsg(m); // remove the old Message from the Folder
        f->addMsg(this); // add this Message to that Folder
    }
    m->folders.clear(); // ensure that destroying m is harmless
}
Message::Message(Message &&m) : contents(std::move(m.contents))
{
    move_Folders(&m); // moves folders and updates the Folder pointers
}
Message& Message::operator=(Message &&rhs)
{
    if (this != &rhs) { // direct check for self-assignment
        remove_from_Folders();
        contents = std::move(rhs.contents); // move assignment
        move_Folders(&rhs); // reset the Folders to point to this Message
    }
    return *this;
}
void StrVec::reallocate()
{
    // allocate space for twice as many elements as the current size
    auto newcapacity = size() ? 2 * size() : 1;
    auto first = alloc.allocate(newcapacity);
    // move the elements
    auto last = uninitialized_copy(make_move_iterator(begin()),
                                   make_move_iterator(end()),
                                   first);
    free(); // free the old space
    elements = first; // update the pointers
    first_free = last;
    cap = elements + newcapacity;
}
void push_back(const X&); // copy: binds to any kind of X
void push_back(X&&); // move: binds only to modifiable rvalues of type X

```

```

class StrVec {
public:
    void push_back(const std::string&); // copy the element
    void push_back(std::string&);      // move the element
    // other members as before
};

// unchanged from the original version in § 13.5 (p. 527)
void StrVec::push_back(const string& s)
{
    chk_n_alloc(); // ensure that there is room for another element
    // construct a copy of s in the element to which first_free points
    alloc.construct(first_free++, s);
}

void StrVec::push_back(string &&s)
{
    chk_n_alloc(); // reallocates the StrVec if necessary
    alloc.construct(first_free++, std::move(s));
}

StrVec vec; // empty StrVec
string s = "some string or another";
vec.push_back(s); // calls push_back(const string&)
vec.push_back("done"); // calls push_back(string&&)
    string s1 = "a value", s2 = "another";
    auto n = (s1 + s2).find('a');

class Foo {
public:
    Foo &operator=(const Foo&) &; // may assign only to modifiable lvalues
    // other members of Foo
};

Foo &Foo::operator=(const Foo &rhs) &
{
    // do whatever is needed to assign rhs to this object
    return *this;
}

Foo &retFoo(); // returns a reference; a call to retFoo is an lvalue
Foo retVal(); // returns by value; a call to retVal is an rvalue
Foo i, j; // i and j are lvalues
i = j; // ok: i is an lvalue
retFoo() = j; // ok: retFoo() returns an lvalue
retVal() = j; // error: retVal() returns an rvalue
i = retVal(); // ok: we can pass an rvalue as the right-hand operand to assignment

class Foo {
public:
    Foo someMem() & const; // error: const qualifier must come first
    Foo anotherMem() const &; // ok: const qualifier comes first
};

```

```

class Foo {
public:
    Foo sorted() &&;           // may run on modifiable rvalues
    Foo sorted() const &;      // may run on any kind of Foo
    // other members of Foo
private:
    vector<int> data;
};

// this object is an rvalue, so we can sort in place
Foo Foo::sorted() &&
{
    sort(data.begin(), data.end());
    return *this;
}
// this object is either const or it is an lvalue; either way we can't sort in place
Foo Foo::sorted() const & {
    Foo ret(*this);           // make a copy
    sort(ret.data.begin(), ret.data.end()); // sort the copy
    return ret;               // return the copy
}
retVal().sorted(); // retVal() is an rvalue, calls Foo::sorted() &&
retFoo().sorted(); // retFoo() is an lvalue, calls Foo::sorted() const &

class Foo {
public:
    Foo sorted() &&;
    Foo sorted() const; // error: must have reference qualifier
    // Comp is type alias for the function type (see § 6.7 (p. 249))
    // that can be used to compare int values
    using Comp = bool(const int&, const int&);
    Foo sorted(Comp*);        // ok: different parameter list
    Foo sorted(Comp*) const; // ok: neither version is reference qualified
};

    Foo Foo::sorted() const & {
        Foo ret(*this);
        return ret.sorted();
    }

    Foo Foo::sorted() const & { return Foo(*this).sorted(); }
    cout << item1 + item2; // print the sum of two Sales_items
print(cout, add(data1, data2)); // print the sum of two Sales_datas
                                // error: cannot redefine the built-in operator for ints
    int operator+(int, int);
    // equivalent calls to a nonmember operator function
    data1 + data2;             // normal expression
    operator+(data1, data2);   // equivalent function call
data1 += data2;                // expression-based "call"
data1.operator+=(data2); // equivalent call to a member operator function
string s = "world";
string t = s + "!"; // ok: we can add a const char* to a string
string u = "hi" + s; // would be an error if + were a member of string

```

```

ostream &operator<<(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}
Sales_data data;
data << cout; // if operator<< is a member of Sales_data
istream &operator>>(istream &is, Sales_data &item)
{
    double price; // no need to initialize; we'll read into price before we use it
    is >> item.bookNo >> item.units_sold >> price;
    if (is) // check that the inputs succeeded
        item.revenue = item.units_sold * price;
    else
        item = Sales_data(); // input failed: give the object the default state
    return is;
}
if (is) // check that the inputs succeeded
    item.revenue = item.units_sold * price;
else
    item = Sales_data(); // input failed: give the object the default state
istream& operator>>(istream& in, Sales_data& s)
{
    double price;
    in >> s.bookNo >> s.units_sold >> price;
    s.revenue = s.units_sold * price;
    return in;
}
// assumes that both objects refer to the same book
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs; // copy data members from lhs into sum
    sum += rhs; // add rhs into sum
    return sum;
}
bool operator==(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn() &&
        lhs.units_sold == rhs.units_sold &&
        lhs.revenue == rhs.revenue;
}
bool operator!=(const Sales_data &lhs, const Sales_data &rhs)
{
    return !(lhs == rhs);
}
vector<string> v;
v = {"a", "an", "the"};

```

```

class StrVec {
public:
    StrVec &operator=(std::initializer_list<std::string>);
    // other members as in § 13.5 (p. 526)
};

StrVec &StrVec::operator=(initializer_list<string> il)
{
    // alloc_n_copy allocates space and copies elements from the given range
    auto data = alloc_n_copy(il.begin(), il.end());
    free(); // destroy the elements in this object and free the space
    elements = data.first; // update data members to point to the new space
    first_free = cap = data.second;
    return *this;
}

// member binary operator: left-hand operand is bound to the implicit this pointer
// assumes that both objects refer to the same book
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

class StrVec {
public:
    std::string& operator[](std::size_t n)
    { return elements[n]; }
    const std::string& operator[](std::size_t n) const
    { return elements[n]; }
    // other members as in § 13.5 (p. 526)
private:
    std::string *elements; // pointer to the first element in the array
};

// assume svec is a StrVec
const StrVec cvec = svec; // copy elements from svec into cvec
// if svec has any elements, run the string empty function on the first one
if (svec.size() && svec[0].empty()) {
    svec[0] = "zero"; // ok: subscript returns a reference to a string
    cvec[0] = "Zip"; // error: subscripting cvec returns a reference to const
}

class StrBlobPtr {
public:
    // increment and decrement
    StrBlobPtr& operator++(); // prefix operators
    StrBlobPtr& operator--();
    // other members as before
};

```

```

// prefix: return a reference to the incremented/decremented object
StrBlobPtr& StrBlobPtr::operator++()
{
    // if curr already points past the end of the container, can't increment it
    check(curr, "increment past end of StrBlobPtr");
    ++curr;           // advance the current state
    return *this;
}

StrBlobPtr& StrBlobPtr::operator--()
{
    // if curr is zero, decrementing it will yield an invalid subscript
    --curr;           // move the current state back one element
    check(-1, "decrement past begin of StrBlobPtr");
    return *this;
}

class StrBlobPtr {
public:
    // increment and decrement
    StrBlobPtr operator++(int);      // postfix operators
    StrBlobPtr operator--(int);
    // other members as before
};

// postfix: increment/decrement the object but return the unchanged value
StrBlobPtr StrBlobPtr::operator++(int)
{
    // no check needed here; the call to prefix increment will do the check
    StrBlobPtr ret = *this;          // save the current value
    +++this;           // advance one element; prefix ++ checks the increment
    return ret;         // return the saved state
}

StrBlobPtr StrBlobPtr::operator--(int)
{
    // no check needed here; the call to prefix decrement will do the check
    StrBlobPtr ret = *this;          // save the current value
    --*this;           // move backward one element; prefix -- checks the decrement
    return ret;         // return the saved state
}

    StrBlobPtr p(a1); // p points to the vector inside a1
    p.operator++(0); // call postfix operator++
    p.operator++();  // call prefix operator++

```

```

class StrBlobPtr {
public:
    std::string& operator*() const
    { auto p = check(curr, "dereference past end");
      return (*p)[curr]; // (*p) is the vector to which this object points
    }
    std::string* operator->() const
    { // delegate the real work to the dereference operator
      return & this->operator*();
    }
    // other members as before
};

StrBlob a1 = {"hi", "bye", "now"};
StrBlobPtr p(a1);           // p points to the vector inside a1
*p = "okay";               // assigns to the first element in a1
cout << p->size() << endl; // prints 4, the size of the first element in a1
cout << (*p).size() << endl; // equivalent to p->size()
(*point).mem;             // point is a built-in pointer type
point.operator()->mem; // point is an object of class type
struct absInt {
    int operator()(int val) const {
        return val < 0 ? -val : val;
    }
    int i = -42;
    absInt absObj; // object that has a function-call operator
    int ui = absObj(i); // passes i to absObj.operator()
};

class PrintString {
public:
    PrintString(ostream &o = cout, char c = ' '):
        os(o), sep(c) { }
    void operator()(const string &s) const { os << s << sep; }
private:
    ostream &os; // stream on which to write
    char sep; // character to print after each output
};
PrintString printer; // uses the defaults; prints to cout
printer(s); // prints s followed by a space on cout
PrintString errors(cerr, '\n');
errors(s); // prints s followed by a newline on cerr
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
// sort words by size, but maintain alphabetical order for words of the same size
stable_sort(words.begin(), words.end(),
            [] (const string &a, const string &b)
            { return a.size() < b.size(); });
class ShorterString {
public:
    bool operator()(const string &s1, const string &s2) const
    { return s1.size() < s2.size(); }
};

stable_sort(words.begin(), words.end(), ShorterString());

```

```

        // get an iterator to the first element whose size() is >= sz
        auto wc = find_if(words.begin(), words.end(),
                           [sz] (const string &a)
    class SizeComp {
        SizeComp(size_t n): sz(n) { } // parameter for each captured variable
        // call operator with the same return type, parameters, and body as the lambda
        bool operator()(const string &s) const
            { return s.size() >= sz; }
private:
    size_t sz; // a data member for each variable captured by value
}; // get an iterator to the first element whose size() is >= sz
auto wc = find_if(words.begin(), words.end(), SizeComp(sz));
plus<int> intAdd; // function object that can add two int values
negate<int> intNegate; // function object that can negate an int value
// uses intAdd::operator(int, int) to add 10 and 20
int sum = intAdd(10, 20); // equivalent to sum = 30
sum = intNegate(intAdd(10, 20)); // equivalent to sum = 30
// uses intNegate::operator(int) to generate -10 as the second parameter
// to intAdd::operator(int, int)
sum = intAdd(10, intNegate(10)); // sum = 0
// passes a temporary function object that applies the < operator to two strings
sort(svec.begin(), svec.end(), greater<string>());
vector<string *> nameTable; // vector of pointers
// error: the pointers in nameTable are unrelated, so < is undefined
sort(nameTable.begin(), nameTable.end(),
      [](string *a, string *b) { return a < b; });
// ok: library guarantees that less on pointer types is well defined
sort(nameTable.begin(), nameTable.end(), less<string*>());
// ordinary function
int add(int i, int j) { return i + j; }
// lambda, which generates an unnamed function-object class
auto mod = [](int i, int j) { return i % j; };
// function-object class
struct div {
    int operator()(int denominator, int divisor) {
        return denominator / divisor;
    }
};
// maps an operator to a pointer to a function taking two ints and returning an int
map<string, int(*)(int,int)> binops;
// ok: add is a pointer to function of the appropriate type
binops.insert({"+", add}); // {"+", add} is a pair § 11.2.3 (p. 426)
binops.insert({"%", mod}); // error: mod is not a pointer to function
function<int(int, int)> f1 = add; // function pointer
function<int(int, int)> f2 = div(); // object of a function-object class
function<int(int, int)> f3 = [](int i, int j) // lambda
    { return i * j; };
cout << f1(4,2) << endl; // prints 6
cout << f2(4,2) << endl; // prints 2
cout << f3(4,2) << endl; // prints 8

```

```

// table of callable objects corresponding to each binary operator
// all the callables must take two ints and return an int
// an element can be a function pointer, function object, or lambda
map<string, function<int(int, int)>> binops;
map<string, function<int(int, int)>> binops = {
    {"+", add}, // function pointer
    {"-", std::minus<int>()}, // library function object
    {"/", div()}, // user-defined function object
    {"*", [](int i, int j) { return i * j; }}, // unnamed lambda
    {"%", mod} // named lambda object
};
binops["+"](10, 5); // calls add(10, 5)
binops["-"](10, 5); // uses the call operator of the minus<int> object
binops["/"](10, 5); // uses the call operator of the div object
binops["*"](10, 5); // calls the lambda function object
binops[%](10, 5); // calls the lambda function object
int add(int i, int j) { return i + j; }
Sales_data add(const Sales_data&, const Sales_data&);
map<string, function<int(int, int)>> binops;
binops.insert( {"+", add}); // error: which add?
int (*fp)(int,int) = add; // pointer to the version of add that takes two ints
binops.insert( {"+", fp}); // ok: fp points to the right version of add
// ok: use a lambda to disambiguate which version of add we want to use
binops.insert( {"+", [](int a, int b) {return add(a, b);}} );
class SmallInt {
public:
    SmallInt(int i = 0): val(i)
    {
        if (i < 0 || i > 255)
            throw std::out_of_range("Bad SmallInt value");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
SmallInt si;
si = 4; // implicitly converts 4 to SmallInt then calls SmallInt::operator=
si + 3; // implicitly converts si to int followed by integer addition
// the double argument is converted to int using the built-in conversion
SmallInt si = 3.14; // calls the SmallInt(int) constructor
// the SmallInt conversion operator converts si to int;
si + 3.14; // that int is converted to double using the built-in conversion
class SmallInt;
operator int(SmallInt&); // error: nonmember
class SmallInt {
public:
    int operator int() const; // error: return type
    operator int(int = 0) const; // error: parameter list
    operator int*() const { return 42; } // error: 42 is not a pointer
};
int i = 42;
cin << i; // this code would be legal if the conversion to bool were not explicit!

```

```

class SmallInt {
public:
    // the compiler won't automatically apply this conversion
    explicit operator int() const { return val; }
    // other members as before
};

SmallInt si = 3; // ok: the SmallInt constructor is not explicit
si + 3; // error: implicit conversion required, but operator int is explicit
static cast<int>(si) + 3; // ok: explicitly request the conversion
    // usually a bad idea to have mutual conversions between two class types

struct B;
struct A {
    A() = default;
    A(const B&);           // converts a B to an A
    // other members
};
struct B {
    operator A() const; // also converts a B to an A
    // other members
};

A f(const A&);

B b;
A a = f(b); // error ambiguous: f(B::operator A())
    // or f(A::A(const B&))
A a1 = f(b.operator A()); // ok: use B's conversion operator
A a2 = f(A(b));          // ok: use A's constructor
struct A {
    A(int = 0); // usually a bad idea to have two
    A(double); // conversions from arithmetic types
    operator int() const; // usually a bad idea to have two
    operator double() const; // conversions to arithmetic types
    // other members
};

void f2(long double);
A a;
f2(a); // error ambiguous: f(A::operator int())
    // or f(A::operator double())

long lg;
A a2(lg); // error ambiguous: A::A(int) or A::A(double)
short s = 42;
// promoting short to int is better than converting short to double
A a3(s); // uses A::A(int)

```

```

struct C {
    C(int);
    // other members
};

struct D {
    D(int);
    // other members
};

void manip(const C&);

void manip(const D&);

manip(10); // error ambiguous: manip(C(10)) or manip(D(10))
            manip(C(10)); // ok: calls manip(const C&)

struct E {
    E(double);
    // other members
};

void manip2(const C&);

void manip2(const E&);

// error ambiguous: two different user-defined conversions could be used
manip2(10); // manip2(C(10)) or manip2(E(double(10)))
a.operatorsym(b); // a has operatorsym as a member function
operatorsym(a, b); // operatorsym is an ordinary function

struct LongDouble {
    LongDouble(double = 0.0);
    operator double();
    operator float();
};

LongDouble ldObj;
int ex1 = ldObj;
float ex2 = ldObj;
void calc(int);
void calc(LongDouble);
double dval;
calc(dval); // which calc?

class SmallInt {
    friend
    SmallInt operator+(const SmallInt&, const SmallInt&);

public:
    SmallInt(int = 0); // conversion from int
    operator int() const { return val; } // conversion to int

private:
    std::size_t val;
};

SmallInt s1, s2;
SmallInt s3 = s1 + s2; // uses overloaded operator+
int i = s3 + 0; // error: ambiguous

```

```

struct LongDouble {
    // member operator+ for illustration purposes; + is usually a nonmember
    LongDouble operator+(const SmallInt&);
    // other members as in § 14.9.2 (p. 587)
};

LongDouble operator+(LongDouble&, double);
SmallInt si;
LongDouble ld;
ld = si + ld;
ld = ld + si;
class Quote {
public:
    std::string isbn() const;
    virtual double net_price(std::size_t n) const;
};

class Bulk_quote : public Quote { // Bulk_quote inherits from Quote
public:
    double net_price(std::size_t) const override;
};

// calculate and print the price for the given number of copies, applying any discounts
double print_total(ostream &os,
                   const Quote &item, std::size_t n)
{
    // depending on the type of the object bound to the item parameter
    // calls either Quote::net_price or Bulk_quote::net_price
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn() // calls Quote::isbn
       << "# sold: " << n << " total due: " << ret << endl;
    return ret;
}

// basic has type Quote; bulk has type Bulk_quote
print_total(cout, basic, 20); // calls Quote version of net_price
print_total(cout, bulk, 20); // calls Bulk_quote version of net_price

class Quote {
public:
    Quote() = default; // = default see § 7.1.4 (p. 264)
    Quote(const std::string &book, double sales_price):
        bookNo(book), price(sales_price) { }
    std::string isbn() const { return bookNo; }
    // returns the total sales price for the specified number of items
    // derived classes will override and apply different discount algorithms
    virtual double net_price(std::size_t n) const
    {
        return n * price;
    }
    virtual ~Quote() = default; // dynamic binding for the destructor
private:
    std::string bookNo; // ISBN number of this item
protected:
    double price = 0.0; // normal, undiscounted price
};

```

```

class Bulk_quote : public Quote { // Bulk_quote inherits from Quote
    Bulk_quote() = default;
    Bulk_quote(const std::string&, double, std::size_t, double);
    // overrides the base version in order to implement the bulk purchase discount policy
    double net_price(std::size_t) const override;
private:
    std::size_t min_qty = 0; // minimum purchase for the discount to apply
    double discount = 0.0; // fractional discount to apply
};

    Quote item; // object of base type
    Bulk_quote bulk; // object of derived type
    Quote *p = &item; // p points to a Quote object
    p = &bulk; // p points to the Quote part of bulk
    Quote &r = bulk; // r bound to the Quote part of bulk
Bulk_quote(const std::string& book, double p,
           std::size_t qty, double disc) :
    Quote(book, p), min_qty(qty), discount(disc) { }
    // as before
};

// if the specified number of items are purchased, use the discounted price
double Bulk_quote::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}

    class Base {
public:
    static void statmem();
};

    class Derived : public Base {
        void f(const Derived&);

};

void Derived::f(const Derived &derived_obj)
{
    Base::statmem(); // ok: Base defines statmem
    Derived::statmem(); // ok: Derived inherits statmem
    // ok: derived objects can be used to access static from base
    derived_obj.statmem(); // accessed through a Derived object
    statmem(); // accessed through this object
}

class Bulk_quote : public Quote; // error: derivation list can't appear here
class Bulk_quote; // ok: right way to declare a derived class
    class Quote; // declared but not defined
    // error: Quote must be defined
    class Bulk_quote : public Quote { ... };
        class Base { /* ... */ };
        class D1: public Base { /* ... */ };
        class D2: public D1 { /* ... */ };

```

```

class NoDerived final { /* */ }; // NoDerived can't be a base class
class Base { /* */ };

// Last is final; we cannot inherit from Last
class Last final : Base { /* */ }; // Last can't be a base class

class Bad : NoDerived { /* */ }; // error: NoDerived is final
class Bad2 : Last { /* */ }; // error: Last is final
    Quote base;

    Bulk_quote* bulkP = &base; // error: can't convert base to derived
    Bulk_quote& bulkRef = base; // error: can't convert base to derived
    Bulk_quote bulk;

    Quote *itemP = &bulk; // ok: dynamic type is Bulk_quote
    Bulk_quote *bulkP = itemP; // error: can't convert base to derived
    Bulk_quote bulk; // object of derived type
    Quote item(bulk); // uses the Quote::Quote(const Quote&) constructor
    item = bulk; // calls Quote::operator=(const Quote&)
    Quote base("0-201-82470-1", 50);
    print_total(cout, base, 10); // calls Quote::net_price

    Bulk_quote derived("0-201-82470-1", 50, 5, .19);
    print_total(cout, derived, 10); // calls Bulk_quote::net_price
    base = derived; // copies the Quote part of derived into base
    base.net_price(20); // calls Quote::net_price

    struct B {
        virtual void f1(int) const;
        virtual void f2();
        void f3();
    };

    struct D1 : B {
        void f1(int) const override; // ok: f1 matches f1 in the base
        void f2(int) override; // error: B has no f2(int) function
        void f3() override; // error: f3 not virtual
        void f4() override; // error: B doesn't have a function named f4
    };
    struct D2 : B {
        // inherits f2() and f3() from B and overrides f1(int)
        void f1(int) const final; // subsequent classes can't override f1(int)
    };
    struct D3 : D2 {
        void f2(); // ok: overrides f2 inherited from the indirect base, B
        void f1(int) const; // error: D2 declared f2 as final
    };
    // calls the version from the base class regardless of the dynamic type of baseP
    double undiscounted = baseP->Quote::net_price(42);

```

```

class base {
public:
    string name() { return basename; }
    virtual void print(ostream &os) { os << basename; }
private:
    string basename;
};

class derived : public base {
public:
    void print(ostream &os) { print(os); os << " " << i; }
private:
    int i;
};

base bobj;      base *bp1 = &bobj;      base &br1 = bobj;
derived dobj;  base *bp2 = &dobj;      base &br2 = dobj;
// class to hold the discount rate and quantity
// derived classes will implement pricing strategies using these data
class Disc_quote : public Quote {
public:
    Disc_quote() = default;
    Disc_quote(const std::string& book, double price,
               std::size_t qty, double disc):
        Quote(book, price),
        quantity(qty), discount(disc) {}
    double net_price(std::size_t) const = 0;
protected:
    std::size_t quantity = 0; // purchase size for the discount to apply
    double discount = 0.0;   // fractional discount to apply
};
// Disc_quote declares pure virtual functions, which Bulk_quote will override
Disc_quote discounted; // error: can't define a Disc_quote object
Bulk_quote bulk;      // ok: Bulk_quote has no pure virtual functions
// the discount kicks in when a specified number of copies of the same book are sold
// the discount is expressed as a fraction to use to reduce the normal price
class Bulk_quote : public Disc_quote {
public:
    Bulk_quote() = default;
    Bulk_quote(const std::string& book, double price,
               std::size_t qty, double disc):
        Disc_quote(book, price, qty, disc) {}
    // overrides the base version to implement the bulk purchase discount policy
    double net_price(std::size_t) const override;
};

```

```

class Base {
protected:
    int prot_mem;      // protected member
};

class Sneaky : public Base {
    friend void clobber(Sneaky&); // can access Sneaky::prot_mem
    friend void clobber(Base&);   // can't access Base::prot_mem
    int j;                // j is private by default
};

// ok: clobber can access the private and protected members in Sneaky objects
void clobber(Sneaky &s) { s.j = s.prot_mem = 0; }

// error: clobber can't access the protected members in Base
void clobber(Base &b) { b.prot_mem = 0; }

class Base {
public:
    void pub_mem(); // public member
protected:
    int prot_mem; // protected member
private:
    char priv_mem; // private member
};

struct Pub_Derv : public Base {
    // ok: derived classes can access protected members
    int f() { return prot_mem; }
    // error: private members are inaccessible to derived classes
    char g() { return priv_mem; }
};

struct Priv_Derv : private Base {
    // private derivation doesn't affect access in the derived class
    int f1() const { return prot_mem; }
};

Pub_Derv d1; // members inherited from Base are public
Priv_Derv d2; // members inherited from Base are private
d1.pub_mem(); // ok: pub_mem is public in the derived class
d2.pub_mem(); // error: pub_mem is private in the derived class
struct Derived_from_Public : public Pub_Derv {
    // ok: Base::prot_mem remains protected in Pub_Derv
    int use_base() { return prot_mem; }
};

struct Derived_from_Private : public Priv_Derv {
    // error: Base::prot_mem is private in Priv_Derv
    int use_base() { return prot_mem; }
};

```

```

class Base {
    // added friend declaration; other members as before
    friend class Pal; // Pal has no access to classes derived from Base
};

class Pal {
public:
    int f(Base b) { return b.prot_mem; } // ok: Pal is a friend of Base
    int f2(Sneaky s) { return s.j; } // error: Pal not friend of Sneaky
    // access to a base class is controlled by the base class, even inside a derived object
    int f3(Sneaky s) { return s.prot_mem; } // ok: Pal is a friend
};

// D2 has no access to protected or private members in Base
class D2 : public Pal {
public:
    int mem(Base b)
        { return b.prot_mem; } // error: friendship doesn't inherit
};

class Base {
public:
    std::size_t size() const { return n; }
protected:
    std::size_t n;
};

class Derived : private Base { // note: private inheritance
public:
    // maintain access levels for members related to the size of the object
    using Base::size;
protected:
    using Base::n;
};

class Base {/* ... */};
struct D1 : Base {/* ... */}; // public inheritance by default
class D2 : Base {/* ... */}; // private inheritance by default
Base *p = &d1; // d1 has type Pub_Derv
p = &d2; // d2 has type Priv_Derv
p = &d3; // d3 has type Prot_Derv
p = &dd1; // dd1 has type Derived_from_Public
p = &dd2; // dd2 has type Derived_from_Private
p = &dd3; // dd3 has type Derived_from_Protected
class Disc_quote : public Quote {
public:
    std::pair<size_t, double> discount_policy() const
        { return {quantity, discount}; }
    // other members as before
};

Bulk_quote bulk;
Bulk_quote *bulkP = &bulk; // static and dynamic types are the same
Quote *itemP = &bulk; // static and dynamic types differ
bulkP->discount_policy(); // ok: bulkP has type Bulk_quote*
itemP->discount_policy(); // error: itemP has type Quote*

```

```

struct Base {
    Base(): mem(0) { }
protected:
    int mem;
};

struct Derived : Base {
    Derived(int i): mem(i) { } // initializes Derived::mem to i
                                // Base::mem is default initialized
    int get_mem() { return mem; } // returns Derived::mem
protected:
    int mem; // hides mem in the base
};
    Derived d(42);
    cout << d.get_mem() << endl; // prints 42
    struct Derived : Base {
        int get_base_mem() { return Base::mem; }
        // ...
    };
    struct Base {
        int memfcn();
    };
    struct Derived : Base {
        int memfcn(int); // hides memfcn in the base
    };
    Derived d; Base b;
    b.memfcn(); // calls Base::memfcn
    d.memfcn(10); // calls Derived::memfcn
    d.memfcn(); // error: memfcn with no arguments is hidden
    d.Base::memfcn(); // ok: calls Base::memfcn
class Base {
public:
    virtual int fcn();
};

class D1 : public Base {
public:
    // hides fcn in the base; this fcn is not virtual
    // D1 inherits the definition of Base::fcn()
    int fcn(int); // parameter list differs from fcn in Base
    virtual void f2(); // new virtual function that does not exist in Base
};

class D2 : public D1 {
public:
    int fcn(int); // nonvirtual function hides D1::fcn(int)
    int fcn(); // overrides virtual fcn from Base
    void f2(); // overrides virtual f2 from D1
};

```

```

Base bobj; D1 d1obj; D2 d2obj;
Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;
bp1->fcn(); // virtual call, will call Base::fcn at run time
bp2->fcn(); // virtual call, will call Base::fcn at run time
bp3->fcn(); // virtual call, will call D2::fcn at run time

D1 *d1p = &d1obj; D2 *d2p = &d2obj;
bp2->f2(); // error: Base has no member named f2
d1p->f2(); // virtual call, will call D1::f2() at run time
d2p->f2(); // virtual call, will call D2::f2() at run time
Base *p1 = &d2obj; D1 *p2 = &d2obj; D2 *p3 = &d2obj;
p1->fcn(42); // error: Base has no version of fcn that takes an int
p2->fcn(42); // statically bound, calls D1::fcn(int)
p3->fcn(42); // statically bound, calls D2::fcn(int)

class Quote {
public:
    // virtual destructor needed if a base pointer pointing to a derived object is deleted
    virtual ~Quote() = default; // dynamic binding for the destructor
};

Quote *itemP = new Quote; // same static and dynamic type
delete itemP; // destructor for Quote called
itemP = new Bulk_quote; // static and dynamic types differ
delete itemP; // destructor for Bulk_quote called

class B {
public:
    B();
    B(const B&) = delete;
    // other members, not including a move constructor
};

class D : public B {
    // no constructors
};

D d; // ok: D's synthesized default constructor uses B's default constructor
D d2(d); // error: D's synthesized copy constructor is deleted
D d3(std::move(d)); // error: implicitly uses D's deleted copy constructor

class Quote {
public:
    Quote() = default; // memberwise default initialize
    Quote(const Quote&) = default; // memberwise copy
    Quote(Quote&&) = default; // memberwise copy
    Quote& operator=(const Quote&) = default; // copy assign
    Quote& operator=(Quote&&) = default; // move assign
    virtual ~Quote() = default;
    // other members as before
};

```

```

class Base { /* ... */ };
class D: public Base {
public:
    // by default, the base class default constructor initializes the base part of an object
    // to use the copy or move constructor, we must explicitly call that
    // constructor in the constructor initializer list
    D(const D& d) : Base(d)           // copy the base members
                    /* initializers for members of D */ { /* ... */ }
    D(D&& d) : Base(std::move(d)) // move the base members
                    /* initializers for members of D */ { /* ... */ }
};

// probably incorrect definition of the D copy constructor
// base-class part is default initialized, not copied
D(const D& d) /* member initializers, but no base-class initializer */
{ /* ... */
    // Base::operator=(const Base&) is not invoked automatically
    D &D::operator=(const D &rhs)
{
    Base::operator=(rhs); // assigns the base part
    // assign the members in the derived class, as usual,
    // handling self-assignment and freeing existing resources as appropriate
    return *this;
}
class D: public Base {
public:
    // Base::~Base invoked automatically
    ~D() { /* do what it takes to clean up derived members */ }
};

class Bulk_quote : public Disc_quote {
public:
    using Disc_quote::Disc_quote; // inherit Disc_quote's constructors
    double net_price(std::size_t) const;
};

Bulk_quote(const std::string& book, double price,
           std::size_t qty, double disc):
    Disc_quote(book, price, qty, disc) { }
vector<Quote> basket;
basket.push_back(Quote("0-201-82470-1", 50));
// ok, but copies only the Quote part of the object into basket
basket.push_back(Bulk_quote("0-201-54848-8", 50, 10, .25));
// calls version defined by Quote, prints 750, i.e., 15 * $50
cout << basket.back().net_price(15) << endl;
vector<shared_ptr<Quote>> basket;

basket.push_back(make_shared<Quote>("0-201-82470-1", 50));
basket.push_back(
    make_shared<Bulk_quote>("0-201-54848-8", 50, 10, .25));
// calls the version defined by Quote; prints 562.5, i.e., 15 * $50 less the discount
cout << basket.back()->net_price(15) << endl;

```

```

class Basket {
public:
    // Basket uses synthesized default constructor and copy-control members
    void add_item(const std::shared_ptr<Quote> &sale)
        { items.insert(sale); }
    // prints the total price for each book and the overall total for all items in the basket
    double total_receipt(std::ostream&) const;
private:
    // function to compare shared_ptrs needed by the multiset member
    static bool compare(const std::shared_ptr<Quote> &lhs,
                        const std::shared_ptr<Quote> &rhs)
    { return lhs->isbn() < rhs->isbn(); }
    // multiset to hold multiple quotes, ordered by the compare member
    std::multiset<std::shared_ptr<Quote>, decltype(compare)>
        items{compare};
};

// multiset to hold multiple quotes, ordered by the compare member
std::multiset<std::shared_ptr<Quote>, decltype(compare)>
    items{compare};
double Basket::total_receipt(ostream &os) const
{
    double sum = 0.0;      // holds the running total
    // iter refers to the first element in a batch of elements with the same ISBN
    // upper_bound returns an iterator to the element just past the end of that batch
    for (auto iter = items.cbegin();
         iter != items.cend();
         iter = items.upper_bound(*iter)) {
        // we know there's at least one element with this key in the Basket
        // print the line item for this book
        sum += print_total(os, **iter, items.count(*iter));
    }
    os << "Total Sale: " << sum << endl; // print the final overall total
    return sum;
}
    sum += print_total(os, **iter, items.count(*iter));
Basket bsk;
bsk.add_item(make_shared<Quote>("123", 45));
bsk.add_item(make_shared<Bulk_quote>("345", 45, 3, .15));
void add_item(const Quote& sale); // copy the given object
void add_item(Quote&& sale);    // move the given object

```

```

class Quote {
public:
    // virtual function to return a dynamically allocated copy of itself
    // these members use reference qualifiers; see § 13.6.3 (p. 546)
    virtual Quote* clone() const & {return new Quote(*this);}
    virtual Quote* clone() &&
        {return new Quote(std::move(*this));}
    // other members as before
};

class Bulk_quote : public Quote {
    Bulk_quote* clone() const & {return new Bulk_quote(*this);}
    Bulk_quote* clone() &&
        {return new Bulk_quote(std::move(*this));}
    // other members as before
};

class Basket {
public:
    void add_item(const Quote& sale) // copy the given object
    { items.insert(std::shared_ptr<Quote>(sale.clone())); }
    void add_item(Quote&& sale) // move the given object
    { items.insert(
        std::shared_ptr<Quote>(std::move(sale).clone())); }
    // other members as before
};

Query q = Query("fiery") & Query("bird") | Query("wind");
// abstract class acts as a base class for concrete query types; all members are private
class Query_base {
    friend class Query;
protected:
    using line_no = TextQuery::line_no; // used in the eval functions
    virtual ~Query_base() = default;
private:
    // eval returns the QueryResult that matches this Query
    virtual QueryResult eval(const TextQuery&) const = 0;
    // rep is a string representation of the query
    virtual std::string rep() const = 0;
};

(a) Query(s1) | Query(s2) & ~Query(s3);
(b) Query(s1) | (Query(s2) & ~Query(s3));
(c) (Query(s1) & (Query(s2)) | (Query(s3) & Query(s4)));

```

```

// interface class to manage the Query_base inheritance hierarchy
class Query {
    // these operators need access to the shared_ptr constructor
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);

public:
    Query(const std::string&); // builds a new WordQuery
    // interface functions: call the corresponding Query_base operations
    QueryResult eval(const TextQuery &t) const
        { return q->eval(t); }
    std::string rep() const { return q->rep(); }

private:
    Query(std::shared_ptr<Query_base> query): q(query) { }
    std::shared_ptr<Query_base> q;
};

std::ostream &
operator<<(std::ostream &os, const Query &query)
{
    // Query::rep makes a virtual call through its Query_base pointer to rep()
    return os << query.rep();
}

Query andq = Query(sought1) & Query(sought2);
cout << andq << endl;

class WordQuery: public Query_base {
    friend class Query; // Query uses the WordQuery constructor
    WordQuery(const std::string &s): query_word(s) { }
    // concrete class: WordQuery defines all inherited pure virtual functions
    QueryResult eval(const TextQuery &t) const
        { return t.query(query_word); }
    std::string rep() const { return query_word; }
    std::string query_word; // word for which to search
};

inline
Query::Query(const std::string &s): q(new WordQuery(s)) { }

class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(const Query &q): query(q) { }
    // concrete class: NotQuery defines all inherited pure virtual functions
    std::string rep() const { return "~(" + query.rep() + ")"; }
    QueryResult eval(const TextQuery&) const;
    Query query;
};

inline Query operator~(const Query &operand)
{
    return std::shared_ptr<Query_base>(new NotQuery(operand));
}

// allocate a new NotQuery object
// bind the resulting NotQuery pointer to a shared_ptr<Query_base>
shared_ptr<Query_base> tmp(new NotQuery(expr));
return Query(tmp); // use the Query constructor that takes a shared_ptr

```

```
class BinaryQuery: public Query_base {
protected:
    BinaryQuery(const Query &l, const Query &r, std::string s):
        lhs(l), rhs(r), opSym(s) { }
    // abstract class: BinaryQuery doesn't define eval
    std::string rep() const { return "(" + lhs.rep() + " "
        + opSym + " "
        + rhs.rep() + ")"; }
    Query lhs, rhs;      // right- and left-hand operands
    std::string opSym; // name of the operator
};

class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery(const Query &left, const Query &right):
        BinaryQuery(left, right, "&") { }
    // concrete class: AndQuery inherits rep and defines the remaining pure virtual
    QueryResult eval(const TextQuery&) const;
};

inline Query operator&(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new AndQuery(lhs, rhs));
}

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery(const Query &left, const Query &right):
        BinaryQuery(left, right, "|") { }
    QueryResult eval(const TextQuery&) const;
};

inline Query operator|(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new OrQuery(lhs, rhs));
}
BinaryQuery a = Query("fiery") & Query("bird");
AndQuery b = Query("fiery") & Query("bird");
OrQuery c = Query("fiery") & Query("bird");
```

```
// returns the union of its operands' result sets
QueryResult
OrQuery::eval(const TextQuery& text) const
{
    // virtual calls through the Query members, lhs and rhs
    // the calls to eval return the QueryResult for each operand
    auto right = rhs.eval(text), left = lhs.eval(text);
    // copy the line numbers from the left-hand operand into the result set
    auto ret_lines =
        make_shared<set<line_no>>(left.begin(), left.end());
    // insert lines from the right-hand operand
    ret_lines->insert(right.begin(), right.end());
    // return the new QueryResult representing the union of lhs and rhs
    return QueryResult(rep(), ret_lines, left.get_file());
}
// returns the intersection of its operands' result sets
QueryResult
AndQuery::eval(const TextQuery& text) const
{
    // virtual calls through the Query operands to get result sets for the operands
    auto left = lhs.eval(text), right = rhs.eval(text);
    // set to hold the intersection of left and right
    auto ret_lines = make_shared<set<line_no>>();
    // writes the intersection of two ranges to a destination iterator
    // destination iterator in this call adds elements to ret
    set_intersection(left.begin(), left.end(),
                     right.begin(), right.end(),
                     inserter(*ret_lines, ret_lines->begin()));
    return QueryResult(rep(), ret_lines, left.get_file());
}
```

```

// returns the lines not in its operand's result set
QueryResult
NotQuery::eval(const TextQuery& text) const
{
    // virtual call to eval through the Query operand
    auto result = query.eval(text);
    // start out with an empty result set
    auto ret_lines = make_shared<set<line_no>>();
    // we have to iterate through the lines on which our operand appears
    auto beg = result.begin(), end = result.end();
    // for each line in the input file, if that line is not in result,
    // add that line number to ret_lines
    auto sz = result.get_file()->size();
    for (size_t n = 0; n != sz; ++n) {
        // if we haven't processed all the lines in result
        // check whether this line is present
        if (beg == end || *beg != n)
            ret_lines->insert(n); // if not in result, add this line
        else if (beg != end)
            ++beg; // otherwise get the next line number in result if there is one
    }
    return QueryResult(rep(), ret_lines, result.get_file());
}
// returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
int compare(const string &v1, const string &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int compare(const double &v1, const double &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
cout << compare(1, 0) << endl;      // T is int
// instantiates int compare(const int&, const int&)
cout << compare(1, 0) << endl;      // T is int
// instantiates int compare(const vector<int>&, const vector<int>&)
vector<int> vec1{1, 2, 3}, vec2{4, 5, 6};
cout << compare(vec1, vec2) << endl; // T is vector<int>

```

```

int compare(const int &v1, const int &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
// ok: same type used for the return type and parameter
template <typename T> T foo(T* p)
{
    T tmp = *p; // tmp will have the type to which p points
    // ...
    return tmp;
}
// error: must precede U with either typename or class
template <typename T, U> T calc(const T&, const U&);
// ok: no distinction between typename and class in a template parameter list
template <typename T, class U> calc (const T&, const U&);
template<unsigned N, unsigned M>
int compare(const char (&p1) [N], const char (&p2) [M])
{
    return strcmp(p1, p2);
}
int compare(const char (&p1) [3], const char (&p2) [4])
// ok: inline specifier follows the template parameter list
template <typename T> inline T min(const T&, const T&);
// error: incorrect placement of the inline specifier
inline template <typename T> T min(const T&, const T&);
// version of compare that will be correct even if used on pointers; see § 14.8.2 (p. 575)
template <typename T> int compare(const T &v1, const T &v2)
{
    if (less<T>()(v1, v2)) return -1;
    if (less<T>()(v2, v1)) return 1;
    return 0;
}
    if (v1 < v2) return -1; // requires < on objects of type T
    if (v2 < v1) return 1; // requires < on objects of type T
    return 0; // returns int; not dependent on T
Sales_data data1, data2;
cout << compare(data1, data2) << endl; // error: no < on Sales_data

```

```

template <typename T> class Blob {
public:
    typedef T value_type;
    typedef typename std::vector<T>::size_type size_type;
    // constructors
    Blob();
    Blob(std::initializer_list<T> il);
    // number of elements in the Blob
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // add and remove elements
    void push_back(const T &t) { data->push_back(t); }
    // move version; see § 13.6.3 (p. 548)
    void push_back(T &&t) { data->push_back(std::move(t)); }
    void pop_back();
    // element access
    T& back();
    T& operator[](size_type i); // defined in § 14.5 (p. 566)
private:
    std::shared_ptr<std::vector<T>> data;
    // throws msg if data[i] isn't valid
    void check(size_type i, const std::string &msg) const;
};

Blob<int> ia; // empty Blob<int>
Blob<int> ia2 = {0,1,2,3,4}; // Blob<int> with five elements
template <> class Blob<int> {
    typedef typename std::vector<int>::size_type size_type;
    Blob();
    Blob(std::initializer_list<int> il);
    // ...
    int& operator[](size_type i);
private:
    std::shared_ptr<std::vector<int>> data;
    void check(size_type i, const std::string &msg) const;
};

// these definitions instantiate two distinct Blob types
Blob<string> names; // Blob that holds strings
Blob<double> prices; // different element type
std::shared_ptr<std::vector<T>> data;
    shared_ptr<vector<string>>
        ret-type StrBlob::member-name (parm-list)
    template <typename T>
        ret-type Blob<T>::member-name (parm-list)
template <typename T>
void Blob<T>::check(size_type i, const std::string &msg) const
{
    if (i >= data->size())
        throw std::out_of_range(msg);
}

```

```

template <typename T>
T& Blob<T>::back()
{
    check(0, "back on empty Blob");
    return data->back();
}
template <typename T>
T& Blob<T>::operator[](size_type i)
{
    // if i is too big, check will throw, preventing access to a nonexistent element
    check(i, "subscript out of range");
    return (*data)[i];
}
template <typename T> void Blob<T>::pop_back()
{
    check(0, "pop_back on empty Blob");
    data->pop_back();
}
template <typename T>
Blob<T>::Blob(): data(std::make_shared<std::vector<T>>()) { }
template <typename T>
Blob<T>::Blob(initializer_list<T> il):
    data(std::make_shared<std::vector<T>>(il)) { }
    Blob<string> articles = {"a", "an", "the"};
// instantiates Blob<int> and the initializer_list<int> constructor
Blob<int> squares = {0,1,2,3,4,5,6,7,8,9};
// instantiates Blob<int>::size() const
for (size_t i = 0; i != squares.size(); ++i)
    squares[i] = i*i; // instantiates Blob<int>::operator[](size_t)
// BlobPtr throws an exception on attempts to access a nonexistent element
template <typename T> class BlobPtr
public:
    BlobPtr(): curr(0) { }
    BlobPtr(Blob<T> &a, size_t sz = 0):
        wptr(a.data), curr(sz) { }
    T& operator*() const
    { auto p = check(curr, "dereference past end");
        return (*p)[curr]; // (*p) is the vector to which this object points
    }
    // increment and decrement
    BlobPtr& operator++();           // prefix operators
    BlobPtr& operator--();
private:
    // check returns a shared_ptr to the vector if the check succeeds
    std::shared_ptr<std::vector<T>>
        check(std::size_t, const std::string&) const;
    // store a weak_ptr, which means the underlying vector might be destroyed
    std::weak_ptr<std::vector<T>> wptr;
    std::size_t curr;               // current position within the array
};

```

```

// postfix: increment/decrement the object but return the unchanged value
template <typename T>
BlobPtr<T> BlobPtr<T>::operator++(int)
{
    // no check needed here; the call to prefix increment will do the check
    BlobPtr ret = *this;      // save the current value
    ++*this;        // advance one element; prefix ++ checks the increment
    return ret;      // return the saved state
}
// forward declarations needed for friend declarations in Blob
template <typename> class BlobPtr;
template <typename> class Blob; // needed for parameters in operator==
template <typename T>
    bool operator==(const Blob<T>&, const Blob<T>&);

template <typename T> class Blob {
    // each instantiation of Blob grants access to the version of
    // BlobPtr and the equality operator instantiated with the same type
    friend class BlobPtr<T>;
    friend bool operator==(T>
        (const Blob<T>&, const Blob<T>&);
    // other members as in § 12.1.1 (p. 456)
};

Blob<char> ca; // BlobPtr<char> and operator==<char> are friends
Blob<int> ia; // BlobPtr<int> and operator==<int> are friends
// forward declaration necessary to befriend a specific instantiation of a template
template <typename T> class Pal;
class C { // C is an ordinary, nontemplate class
    friend class Pal<C>; // Pal instantiated with class C is a friend to C
    // all instances of Pal2 are friends to C;
    // no forward declaration required when we befriend all instantiations
    template <typename T> friend class Pal2;
};

template <typename T> class C2 { // C2 is itself a class template
    // each instantiation of C2 has the same instance of Pal as a friend
    friend class Pal<T>; // a template declaration for Pal must be in scope
    // all instances of Pal2 are friends of each instance of C2, prior declaration needed
    template <typename X> friend class Pal2;
    // Pal3 is a nontemplate class that is a friend of every instance of C2
    friend class Pal3; // prior declaration for Pal3 not needed
};

template <typename Type> class Bar {
    friend Type; // grants access to the type used to instantiate Bar
    // ...
};

template<typename T> using twin = pair<T, T>;
twin<string> authors; // authors is a pair<string, string>
twin<int> win_loss; // win_loss is a pair<int, int>
twin<double> area; // area is a pair<double, double>
template <typename T> using partNo = pair<T, unsigned>;
partNo<string> books; // books is a pair<string, unsigned>
partNo<Vehicle> cars; // cars is a pair<Vehicle, unsigned>
partNo<Student> kids; // kids is a pair<Student, unsigned>

```

```

template <typename T> class Foo {
public:
    static std::size_t count() { return ctr; }
    // other interface members
private:
    static std::size_t ctr;
    // other implementation members
};
// instantiates static members Foo<string>::ctr and Foo<string>::count
Foo<string> fs;
// all three objects share the same Foo<int>::ctr and Foo<int>::count members
Foo<int> fi, fi2, fi3;
template <typename T>
size_t Foo<T>::ctr = 0; // define and initialize ctr
Foo<int> fi;           // instantiates Foo<int> class
                        // and the static data member ctr
auto ct = Foo<int>::count(); // instantiates Foo<int>::count
ct = fi.count();          // uses Foo<int>::count
ct = Foo::count();         // error: which template instantiation?
template <typename elemType> class ListItem;
template <typename elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
    void insert(ListItem *ptr, elemType value);
private:
    ListItem *front, *end;
};
template <typename Foo> Foo calc(const Foo& a, const Foo& b)
{
    Foo tmp = a; // tmp has the same type as the parameters and return type
    // ...
    return tmp; // return type and parameters have the same type
}
typedef double A;
template <typename A, typename B> void f(A a, B b)
{
    A tmp = a; // tmp has same type as the template parameter A, not double
    double B; // error: redeclares template parameter B
}
// error: illegal reuse of template parameter name V
template <typename V, typename V> // ...
// declares but does not define compare and Blob
template <typename T> int compare(const T&, const T&);
template <typename T> class Blob;

```

```

// all three uses of calc refer to the same function template
template <typename T> T calc(const T&, const T&); // declaration
template <typename U> U calc(const U&, const U&); // declaration
// definition of the template
template <typename Type>
Type calc(const Type& a, const Type& b) { /* ... */ }
    template <typename T>
        typename T::value_type top(const T& c)
    {
        if (!c.empty())
            return c.back();
        else
            return typename T::value_type();
    }
// compare has a default template argument, less<T>
// and a default function argument, F()
template <typename T, typename F = less<T>>
int compare(const T &v1, const T &v2, F f = F())
{
    if (f(v1, v2)) return -1;
    if (f(v2, v1)) return 1;
    return 0;
}
bool i = compare(0, 42); // uses less; i is -1
// result depends on the ISBNs in item1 and item2
Sales_data item1(cin), item2(cin);
bool j = compare(item1, item2, compareIsbn);
template <class T = int> class Numbers { // by default T is int
public:
    Numbers(T v = 0): val(v) { }
    // various operations on numbers
private:
    T val;
};
Numbers<long double> lots_of_precision;
Numbers<> average_precision; // empty <> says we want the default type
(a) template <typename T, U, typename V> void f1(T, U, V);
(b) template <typename T> T f2(int &T);
(c) inline template <typename T> T foo(T, unsigned int*);
(d) template <typename T> f4(T, T);
(e) typedef char Ctype;
    template <typename Ctype> Ctype f5(Ctype a);

```

```

// function-object class that calls delete on a given pointer
class DebugDelete {
public:
    DebugDelete(std::ostream &s = std::cerr): os(s) { }
    // as with any function template, the type of T is deduced by the compiler
    template <typename T> void operator()(T *p) const
        { os << "deleting unique_ptr" << std::endl; delete p; }
private:
    std::ostream &os;
};

double* p = new double;
DebugDelete d;      // an object that can act like a delete expression
d(p); // calls DebugDelete::operator()(double*), which deletes p

int* ip = new int;
// calls operator()(int*) on a temporary DebugDelete object
DebugDelete()(ip);
// destroying the the object to which p points
// instantiates DebugDelete::operator()(int*)(int *)
unique_ptr<int, DebugDelete> p(new int, DebugDelete());
// destroying the the object to which sp points
// instantiates DebugDelete::operator()(string*)(string*)
unique_ptr<string, DebugDelete> sp(new string, DebugDelete());
// sample instantiations for member templates of DebugDelete
void DebugDelete::operator()(int *p) const { delete p; }
void DebugDelete::operator()(string *p) const { delete p; }

template <typename T> class Blob {
    template <typename It> Blob(It b, It e);
    // ...
};

template <typename T>      // type parameter for the class
template <typename It>      // type parameter for the constructor
    Blob<T>::Blob(It b, It e):
        data(std::make_shared<std::vector<T>>(b, e)) { }
int ia[] = {0,1,2,3,4,5,6,7,8,9};
vector<long> vi = {0,1,2,3,4,5,6,7,8,9};
list<const char*> w = {"now", "is", "the", "time"};
// instantiates the Blob<int> class
// and the Blob<int> constructor that has two int* parameters
Blob<int> a1(begin(ia), end(ia));
// instantiates the Blob<int> constructor that has
// two vector<long>::iterator parameters
Blob<int> a2(vi.begin(), vi.end());
// instantiates the Blob<string> class and the Blob<string>
// constructor that has two (list<const char*>::iterator) parameters
Blob<string> a3(w.begin(), w.end());
extern template declaration; // instantiation declaration
template declaration; // instantiation definition
// instantiation declaration and definition
extern template class Blob<string>; // declaration
template int compare(const int&, const int&); // definition

```

```

// Application.cc
// these template types must be instantiated elsewhere in the program
extern template class Blob<string>;
extern template int compare(const int&, const int&);

Blob<string> sa1, sa2; // instantiation will appear elsewhere
// Blob<int> and its initializer_list constructor instantiated in this file
Blob<int> a1 = {0,1,2,3,4,5,6,7,8,9};
Blob<int> a2(a1); // copy constructor instantiated in this file
int i = compare(a1[0], a2[0]); // instantiation will appear elsewhere
// templateBuild.cc
// instantiation file must provide a (nonextern) definition for every
// type and function that other files declare as extern
template int compare(const int&, const int&);
template class Blob<string>; // instantiates all members of the class template
    extern template class vector<string>;
        template class vector<Sales_data>;
template <typename T> class Stack { };
void f1(Stack<char>); // (a)
class Exercise {
    Stack<double> &rsd; // (b)
    Stack<int> si; // (c)
};
int main() {
    Stack<char> *sc; // (d)
    f1(*sc); // (e)
    int iObj = sizeof(Stack< string >); // (f)
}
// value of del known only at run time; call through a pointer
del ? del(p) : delete p; // del(p) requires run-time jump to del's location
    // del bound at compile time; direct call to the deleter is instantiated
    del(p); // no run-time overhead
template <typename T> T fobj(T, T); // arguments are copied
template <typename T> T fref(const T&, const T&); // references
string s1("a value");
const string s2("another value");
fobj(s1, s2); // calls fobj(string, string); const is ignored
fref(s1, s2); // calls fref(const string&, const string&)
    // uses permissible conversion to const on s1

int a[10], b[42];
fobj(a, b); // calls f(int*, int*)
fref(a, b); // error: array types don't match
long lmg;
compare(lmg, 1024); // error: cannot instantiate compare(long, int)
    // argument types can differ but must be compatible
template <typename A, typename B>
int flexibleCompare(const A& v1, const B& v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}

```

```

long lng;
flexibleCompare(lng, 1024); // ok: calls flexibleCompare(long, int)
template <typename T> ostream &print(ostream &os, const T &obj)
{
    return os << obj;
}
print(cout, 42); // instantiates print(ostream&, int)
ofstream f("output");
print(f, 10); // uses print(ostream&, int); converts f to ostream&
template <class T> int compare(const T&, const T&);
    template <typename T> T calc(T, int);
    template <typename T> T fcn(T, T);
    double d; float f; char c;
    template <typename T> f1(T, T);
    template <typename T1, typename T2> f2(T1, T2);
    int i = 0, j = 42, *p1 = &i, *p2 = &j;
    const int *cp1 = &i, *cp2 = &j;
    // T1 cannot be deduced: it doesn't appear in the function parameter list
    template <typename T1, typename T2, typename T3>
        T1 sum(T2, T3);
// T1 is explicitly specified; T2 and T3 are inferred from the argument types
auto val3 = sum<long long>(i, lng); // long long sum(int, long)
    // poor design: users must explicitly specify all three template parameters
    template <typename T1, typename T2, typename T3>
        T3 alternative sum(T2, T1);
// error: can't infer initial template parameters
auto val3 = alternative_sum<long long>(i, lng);
// ok: all three parameters are explicitly specified
auto val2 = alternative_sum<long long, int, long>(i, lng);
long lng;
compare(lng, 1024); // error: template parameters don't match
compare<long>(lng, 1024); // ok: instantiates compare(long, long)
compare<int>(lng, 1024); // ok: instantiates compare(int, int)
template <typename It>
    ??? &fcn(It beg, It end)
{
    // process the range
    return *beg; // return a reference to an element from the range
}
vector<int> vi = {1,2,3,4,5};
Blob<string> ca = { "hi", "bye" };
auto &i = fcn(vi.begin(), vi.end()); // fcn should return int&
auto &s = fcn(ca.begin(), ca.end()); // fcn should return string&
// a trailing return lets us declare the return type after the parameter list is seen
template <typename It>
auto fcn(It beg, It end) -> decltype(*beg)
{
    // process the range
    return *beg; // return a reference to an element from the range
}
remove_reference<decltype(*beg)>::type

```

```

// must use typename to use a type member of a template parameter; see § 16.1.3 (p. 670)
template <typename It>
auto fcn2(It beg, It end) ->
    typename remove_reference<decltype(*beg)>::type
{
    // process the range
    return *beg; // return a copy of an element from the range
}
template <typename It>
auto fcn3(It beg, It end) -> decltype(*beg + 0)
{
    // process the range
    return *beg; // return a copy of an element from the range
}
template <typename T> int compare(const T&, const T&);
// p1 points to the instantiation int compare(const int&, const int&)
int (*p1)(const int&, const int&) = compare;
// overloaded versions of func; each takes a different function pointer type
void func(int(*)(const string&, const string&));
void func(int(*)(const int&, const int&));

func(compare); // error: which instantiation of compare?
// ok: explicitly specify which version of compare to instantiate
func(compare<int>); // passing compare(const int&, const int&)
                     template <typename T> void f(T &p);
template <typename T> void f1(T&); // argument must be an lvalue
// calls to f1 use the referred-to type of the argument as the template parameter type
f1(i); // i is an int; template parameter T is int
f1(ci); // ci is a const int; template parameter T is const int
f1(5); // error: argument to a & parameter must be an lvalue
template <typename T> void f2(const T&); // can take an rvalue
// parameter in f2 is const &; const in the argument is irrelevant
// in each of these three calls, f2's function parameter is inferred as const int&
f2(i); // i is an int; template parameter T is int
f2(ci); // ci is a const int, but template parameter T is int
f2(5); // a const & parameter can be bound to an rvalue; T is int
template <typename T> void f3(T&&);
f3(42); // argument is an rvalue of type int; template parameter T is int
f3(i); // argument is an lvalue; template parameter T is int&
f3(ci); // argument is an lvalue; template parameter T is const int&
// invalid code, for illustration purposes only
void f3<int&>(int& &&); // when T is int&, function parameter is int& &&
void f3<int&>(int&); // when T is int&, function parameter collapses to int&
template <typename T> void f3(T&& val)
{
    T t = val; // copy or binding a reference?
    t = fcn(t); // does the assignment change only t or val and t?
    if (val == t) { /* ... */ } // always true if T is a reference type
}
template <typename T> void f(T&&); // binds to non const rvalues
template <typename T> void f(const T&); // lvalues and const rvalues
template <typename T> void g(T&& val);
    int i = 0; const int ci = i;
template <typename T> void g(T&& val) { vector<T> v; }

```

```

// for the use of typename in the return type and the cast see § 16.1.3 (p. 670)
// remove_reference is covered in § 16.2.3 (p. 684)
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
{
    // static_cast covered in § 4.11.3 (p. 163)
    return static_cast<typename remove_reference<T>::type&&>(t);
}
string s1("hi!"), s2;
s2 = std::move(string("bye!")); // ok: moving from an rvalue
s2 = std::move(s1); // ok: but after the assignment s1 has indeterminate value
for (size_t i = 0; i != size(); ++i)
    alloc.construct(dest++, std::move(*elem++));
// template that takes a callable and two parameters
// and calls the given callable with the parameters "flipped"
// flip1 is an incomplete implementation: top-level const and references are lost
template <typename F, typename T1, typename T2>
void flip1(F f, T1 t1, T2 t2)
{
    f(t2, t1);
}
void f(int v1, int &v2) // note v2 is a reference
{
    cout << v1 << " " << ++v2 << endl;
}
f(42, i); // f changes its argument i
flip1(f, j, 42); // f called through flip1 leaves j unchanged
void flip1(void(*fcn)(int, int&), int t1, int t2);
template <typename F, typename T1, typename T2>
void flip2(F f, T1 &&t1, T2 &&t2)
{
    f(t2, t1);
}
void g(int &i, int& j)
{
    cout << i << " " << j << endl;
}
flip2(g, i, 42); // error: can't initialize int&& from an lvalue
template <typename Type> intermediary(Type &&arg)
{
    finalFcn(std::forward<Type>(arg));
    // ...
}
template <typename F, typename T1, typename T2>
void flip(F f, T1 &&t1, T2 &&t2)
{
    f(std::forward<T2>(t2), std::forward<T1>(t1));
}

```

```

// print any type we don't otherwise handle
template <typename T> string debug_rep(const T &t)
{
    ostringstream ret; // see § 8.3 (p. 321)
    ret << t; // uses T's output operator to print a representation of t
    return ret.str(); // return a copy of the string to which ret is bound
}
// print pointers as their pointer value, followed by the object to which the pointer points
// NB: this function will not work properly with char*; see § 16.3 (p. 698)
template <typename T> string debug_rep(T *p)
{
    ostringstream ret;
    ret << "pointer: " << p; // print the pointer's own value
    if (p)
        ret << " " << debug_rep(*p); // print the value to which p points
    else
        ret << " null pointer"; // or indicate that the p is null
    return ret.str(); // return a copy of the string to which ret is bound
}
string s("hi");
cout << debug_rep(s) << endl;
cout << debug_rep(&s) << endl;
const string *sp = &s;
cout << debug_rep(sp) << endl;
// print strings inside double quotes
string debug_rep(const string &s)
{
    return '"' + s + '"';
}
string s("hi");
cout << debug_rep(s) << endl;
cout << debug_rep("hi world!") << endl; // calls debug_rep(T*)
// convert the character pointers to string and call the string version of debug_rep
string debug_rep(char *p)
{
    return debug_rep(string(p));
}
string debug_rep(const char *p)
{
    return debug_rep(string(p));
}
template <typename T> string debug_rep(const T &t);
template <typename T> string debug_rep(T *p);
// the following declaration must be in scope
// for the definition of debug_rep(char*) to do the right thing
string debug_rep(const string &);

string debug_rep(char *p)
{
    // if the declaration for the version that takes a const string& is not in scope
    // the return will call debug_rep(const T&) with T instantiated to string
    return debug_rep(string(p));
}

```

```

        template <typename T> void f(T);
        template <typename T> void f(const T*);
        template <typename T> void g(T);
        template <typename T> void g(T*);
        int i = 42, *p = &i;
        const int ci = 0, *p2 = &ci;
        g(42);   g(p);   g(ci);   g(p2);
        f(42);   f(p);   f(ci);   f(p2);
    //  Args is a template parameter pack; rest is a function parameter pack
    //  Args represents zero or more template type parameters
    //  rest represents zero or more function parameters
    template <typename T, typename... Args>
        void foo(const T &t, const Args& ... rest);
    int i = 0;  double d = 3.14; string s = "how now brown cow";
    foo(i, s, 42, d);      // three parameters in the pack
    foo(s, 42, "hi");     // two parameters in the pack
    foo(d, s);            // one parameter in the pack
    foo("hi");            // empty pack
    void foo(const int&, const string&, const int&, const double&);
    void foo(const string&, const int&, const char[3]&);
    void foo(const double&, const string&);
    void foo(const char[3]&);
    template<typename ... Args> void g(Args ... args) {
        cout << sizeof...(Args) << endl; // number of type parameters
        cout << sizeof...(args) << endl; // number of function parameters
    }
    // function to end the recursion and print the last element
    // this function must be declared before the variadic version of print is defined
    template<typename T>
    ostream &print(ostream &os, const T &t)
    {
        return os << t; // no separator after the last element in the pack
    }
    // this version of print will be called for all but the last element in the pack
    template <typename T, typename... Args>
    ostream &print(ostream &os, const T &t, const Args&... rest)
    {
        os << t << ", ";           // print the first argument
        return print(os, rest...); // recursive call; print the other arguments
    }
    return print(os, rest...); // recursive call; print the other arguments
    print(cout, i, s, 42); // two parameters in the pack
    template <typename T, typename... Args>
    ostream &
    print(ostream &os, const T &t, const Args&... rest)// expand Args
    {
        os << t << ", ";
        return print(os, rest...);                                // expand rest
    }
    print(cout, i, s, 42); // two parameters in the pack
    ostream&
    print(ostream&, const int&, const string&, const int&);

```

```

// call debug_rep on each argument in the call to print
template <typename... Args>
ostream &errorMsg(ostream &os, const Args&... rest)
{
    // print(os, debug_rep(a1), debug_rep(a2), ..., debug_rep(an))
    return print(os, debug_rep(rest)...);
}
errorMsg(cerr, fcnName, code.num(), otherData, "other", item);
print(cerr, debug_rep(fcnName), debug_rep(code.num()),
      debug_rep(otherData), debug_rep("otherData"),
      debug_rep(item));
// passes the pack to debug_rep; print(os, debug_rep(a1, a2, ..., an))
print(os, debug_rep(rest...)); // error: no matching function to call
    print(cerr, debug_rep(fcnName, code.num(),
                           otherData, "otherData", item));
class StrVec {
public:
    template <class... Args> void emplace_back(Args&&...);
        // remaining members as in § 13.5 (p. 526)
    };
template <class... Args>
inline
void StrVec::emplace_back(Args&&... args)
{
    chk_n_alloc(); // reallocates the StrVec if necessary
    alloc.construct(first_free++, std::forward<Args>(args)...);
}
svec.emplace_back(10, 'c'); // adds cccccccccc as a new last element
    std::forward<int>(10), std::forward<char>(c)
    svec.emplace_back(s1 + s2); // uses the move constructor
        std::forward<string>(string("the end"))
// fun has zero or more parameters each of which is
// an rvalue reference to a template parameter type
template<typename... Args>
void fun(Args&&... args) // expands Args as a list of rvalue references
{
    // the argument to work expands both Args and args
    work(std::forward<Args>(args)...);
}
// first version; can compare any two types
template <typename T> int compare(const T&, const T&);
// second version to handle string literals
template<size_t N, size_t M>
    int compare(const char (&) [N], const char (&) [M]);
const char *p1 = "hi", *p2 = "mom";
compare(p1, p2); // calls the first template
compare("hi", "mom"); // calls the template with two nontype parameters
// special version of compare to handle pointers to character arrays
template <>
int compare(const char* const &p1, const char* const &p2)
{
    return strcmp(p1, p2);
}
template <typename T> int compare(const T&, const T&);

```

```

    // open the std namespace so we can specialize std::hash
    namespace std {
        } // close the std namespace; note: no semicolon after the close curly
    // open the std namespace so we can specialize std::hash
    namespace std {
        template <> // we're defining a specialization with
        struct hash<Sales_data> // the template parameter of Sales_data
        {
            // the type used to hash an unordered container must define these types
            typedef size_t result_type;
            typedef Sales_data argument_type; // by default, this type needs ==
            size_t operator()(const Sales_data& s) const;
            // our class uses synthesized copy control and default constructor
        };
        size_t
        hash<Sales_data>::operator()(const Sales_data& s) const
        {
            return hash<string>()(s.bookNo) ^
                hash<unsigned>()(s.units_sold) ^
                hash<double>()(s.revenue);
        }
    } // close the std namespace; note: no semicolon after the close curly
    // uses hash<Sales_data> and Sales_data operator== from § 14.3.1 (p. 561)
    unordered_multiset<Sales_data> SDset;
    template <class T> class std::hash; // needed for the friend declaration
    class Sales_data {
        friend class std::hash<Sales_data>;
        // other members as before
    };
    // original, most general template
    template <class T> struct remove_reference {
        typedef T type;
    };
    // partial specializations that will be used for lvalue and rvalue references
    template <class T> struct remove_reference<T&> // lvalue references
    { typedef T type; };
    template <class T> struct remove_reference<T&&> // rvalue references
    { typedef T type; };
    int i;
    // decltype(42) is int, uses the original template
    remove_reference<decltype(42)>::type a;
    // decltype(i) is int&, uses first (T&) partial specialization
    remove_reference<decltype(i)>::type b;
    // decltype(std::move(i)) is int&&, uses second (i.e., T&&) partial specialization
    remove_reference<decltype(std::move(i))>::type c;

```

```

template <typename T> struct Foo {
    Foo(const T &t = T()): mem(t) { }
    void Bar() { /* ... */ }
    T mem;
    // other members of Foo
};

template<> // we're specializing a template
void Foo<int>::Bar() // we're specializing the Bar member of Foo<int>
{
    // do whatever specialized processing that applies to ints
}

Foo<string> fs; // instantiates Foo<string>::Foo()
fs.Bar(); // instantiates Foo<string>::Bar()

Foo<int> fi; // instantiates Foo<int>::Foo()
fi.Bar(); // uses our specialization of Foo<int>::Bar()
tuple<size_t, size_t, size_t> threeD; // all three members set to 0
tuple<string, vector<double>, int, list<int>>
    someVal("constants", {3.14, 2.718}, 42, {0,1,2,3,4,5});
tuple<size_t, size_t, size_t> threeD = {1,2,3}; // error
tuple<size_t, size_t, size_t> threeD{1,2,3}; // ok
// tuple that represents a bookstore transaction: ISBN, count, price per book
auto item = make_tuple("0-999-78345-X", 3, 20.00);
auto book = get<0>(item); // returns the first member of item
auto cnt = get<1>(item); // returns the second member of item
auto price = get<2>(item)/cnt; // returns the last member of item
get<2>(item) *= 0.8; // apply 20% discount
typedef decltype(item) trans; // trans is the type of item

// returns the number of members in object's of type trans
size_t sz = tuple_size<trans>::value; // returns 3
// cnt has the same type as the second member in item
tuple_element<1, trans>::type cnt = get<1>(item); // cnt is an int
tuple<string, string> duo("1", "2");
tuple<size_t, size_t> twoD(1, 2);
bool b = (duo == twoD); // error: can't compare a size_t and a string
tuple<size_t, size_t, size_t> threeD(1, 2, 3);
b = (twoD < threeD); // error: differing number of members
tuple<size_t, size_t> origin(0, 0);
b = (origin < twoD); // ok: b is true
// each element in files holds the transactions for a particular store
vector<vector<Sales_data>> files;

```

```

// matches has three members: an index of a store and iterators into that store's vector
typedef tuple<vector<Sales_data>::size_type,
              vector<Sales_data>::const_iterator,
              vector<Sales_data>::const_iterator> matches;

// files holds the transactions for every store
// findBook returns a vector with an entry for each store that sold the given book
vector<matches>
findBook(const vector<vector<Sales_data>> &files,
         const string &book)
{
    vector<matches> ret; // initially empty
    // for each store find the range of matching books, if any
    for (auto it = files.cbegin(); it != files.cend(); ++it) {
        // find the range of Sales_data that have the same ISBN
        auto found = equal_range(it->cbegin(), it->cend(),
                                 book, compareIsbn);
        if (found.first != found.second) // this store had sales
            // remember the index of this store and the matching range
            ret.push_back(make_tuple(it - files.cbegin(),
                                      found.first, found.second));
    }
    return ret; // empty if no matches found
}
void reportResults(istream &in, ostream &os,
                   const vector<vector<Sales_data>> &files)
{
    string s; // book to look for
    while (in >> s) {
        auto trans = findBook(files, s); // stores that sold this book
        if (trans.empty()) {
            cout << s << " not found in any stores" << endl;
            continue; // get the next book to look for
        }
        for (const auto &store : trans) // for every store with a sale
            // get<n> returns the specified member from the tuple in store
            os << "store " << get<0>(store) << " sales: "
            << accumulate(get<1>(store), get<2>(store),
                           Sales_data(s))
            << endl;
    }
}

bitset<32> bitvec(1U); // 32 bits; low-order bit is 1, remaining bits are 0
// bitvec1 is smaller than the initializer; high-order bits from the initializer are discarded
bitset<13> bitvec1(0xbeef); // bits are 1111011101111
// bitvec2 is larger than the initializer; high-order bits in bitvec2 are set to zero
bitset<20> bitvec2(0xbeef); // bits are 00001011111011101111
// on machines with 64-bit long long OULL is 64 bits of 0, so ~OULL is 64 ones
bitset<128> bitvec3(~OULL); // bits 0...63 are one; 63...127 are zero
    bitset<32> bitvec4("1100"); // bits 2 and 3 are 1, all others are 0
string str("1111111000000011001101");
bitset<32> bitvec5(str, 5, 4); // four bits starting at str[5], 1100
bitset<32> bitvec6(str, str.size()-4); // use last four characters

```

```

bitset<32> bitvec(1U); // 32 bits; low-order bit is 1, remaining bits are 0
bool is_set = bitvec.any();           // true, one bit is set
bool is_not_set = bitvec.none();     // false, one bit is set
bool all_set = bitvec.all();         // false, only one bit is set
size_t onBits = bitvec.count();      // returns 1
size_t sz = bitvec.size();          // returns 32
bitvec.flip();                     // reverses the value of all the bits in bitvec
bitvec.reset();                    // sets all the bits to 0
bitvec.set();                      // sets all the bits to 1
    bitvec.flip(0);                // reverses the value of the first bit
    bitvec.set(bitvec.size() - 1); // turns on the last bit
    bitvec.set(0, 0);              // turns off the first bit
    bitvec.reset(i);              // turns off the ith bit
    bitvec.test(0);               // returns false because the first bit is off
bitvec[0] = 0;                      // turn off the bit at position 0
bitvec[31] = bitvec[0];             // set the last bit to the same value as the first bit
bitvec[0].flip();                  // flip the value of the bit at position 0
~bitvec[0];                        // equivalent operation; flips the bit at position 0
bool b = bitvec[0];                // convert the value of bitvec[0] to bool
    unsigned long ulong = bitvec3.to_ulong();
    cout << "ulong = " << ulong << endl;
bitset<16> bits;
cin >> bits; // read up to 16 1 or 0 characters from cin
cout << "bits: " << bits << endl; // print what we just read
bool status;
// version using bitwise operators
unsigned long quizA = 0;           // this value is used as a collection of bits
quizA |= 1UL << 27;               // indicate student number 27 passed
status = quizA & (1UL << 27); // check how student number 27 did
quizA &= ~(1UL << 27);           // student number 27 failed

// equivalent actions using the bitset library
bitset<30> quizB;                // allocate one bit per student; all bits initialized to 0
quizB.set(27);                   // indicate student number 27 passed
status = quizB[27];              // check how student number 27 did
quizB.reset(27);                 // student number 27 failed
// find the characters ei that follow a character other than c
string pattern("[^clei]");
// we want the whole word in which our pattern appears
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern); // construct a regex to find pattern
smatch results; // define an object to hold the results of a search
// define a string that has text that does and doesn't match pattern
string test_str = "receipt freind theif receive";
// use r to find a match to pattern in test_str
if (regex_search(test_str, results, r)) // if there is a match
    cout << results.str() << endl; // print the matching word

```

```

// one or more alphanumeric characters followed by a '.' followed by "cpp" or "cxx" or "cc"
regex r("[[:alnum:]]+\\".(cpp|cxx|cc)$", regex::icase);
smatch results;
string filename;
while (cin >> filename)
    if (regex_search(filename, results, r))
        cout << results.str() << endl; // print the current match
try {
    // error: missing close bracket after alnum; the constructor will throw
    regex r("[[:alnum:]]+\\".(cpp|cxx|cc)$", regex::icase);
} catch (regex_error e)
{
    cout << e.what() << "\nerror: " << e.code() << endl;
}
regex r("[[:alnum:]]+\\".(cpp|cxx|cc)$", regex::icase);
smatch results; // will match a string input sequence, but not char*
if (regex_search("myfile.cc", results, r)) // error: char* input
    cout << results.str() << endl;
cmatch results; // will match character array input sequences
if (regex_search("myfile.cc", results, r))
    cout << results.str() << endl; // print the current match
// find the characters ei that follow a character other than c
string pattern("[^c]ei");
// we want the whole word in which our pattern appears
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern, regex::icase); // we'll ignore case in doing the match
// it will repeatedly call regex_search to find all matches in file
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
     it != end_it; ++it)
    cout << it->str() << endl; // matched word
    // read or write according to the type
    >>> being <<<
    handled. The input operators ignore whi
// same for loop header as before
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
     it != end_it; ++it) {
    auto pos = it->prefix().length(); // size of the prefix
    pos = pos > 40 ? pos - 40 : 0; // we want up to 40 characters
    cout << it->prefix().str().substr(pos) // last part of the prefix
        << "\n\t\t>>> " << it->str() << " <<<\n" // matched word
        << it->suffix().str().substr(0, 40) // first part of the suffix
        << endl;
}
// r has two subexpressions: the first is the part of the file name before the period
// the second is the file extension
regex r("([[:alnum:]]+)\\".(cpp|cxx|cc)$", regex::icase);
if (regex_search(filename, results, r))
    cout << results.str(1) << endl; // print the first subexpression
// our overall expression has seven subexpressions: (ddd ) separator ddd separator dddd
// subexpressions 1, 3, 4, and 6 are optional; 2, 5, and 7 hold the number
"(\\"()?(\\d{3})\\\")?([-. ])?(\\d{3})([-. ]?)\\d{4})";

```

```

string phone =
    "(\\"(?) (\\"d{3}) (\\")) ? ([-. ]) ? (\\"d{3}) ([-. ]) ? (\\"d{4}) ";
regex r(phone); // a regex to find our pattern
smatch m;
string s;
// read each record from the input file
while (getline(cin, s)) {
    // for each matching phone number
    for (sregex_iterator it(s.begin(), s.end(), r), end_it;
         it != end_it; ++it)
        // check whether the number's formatting is valid
        if (valid(*it))
            cout << "valid: " << it->str() << endl;
        else
            cout << "not valid: " << it->str() << endl;
    }
bool valid(const smatch& m)
{
    // if there is an open parenthesis before the area code
    if (m[1].matched)
        // the area code must be followed by a close parenthesis
        // and followed immediately by the rest of the number or a space
        return m[3].matched
            && (m[4].matched == 0 || m[4].str() == " ");
    else
        // then there can't be a close after the area code
        // the delimiters between the other two components must match
        return !m[3].matched
            && m[4].str() == m[6].str();
}
string fmt = "$2.$5.$7"; // reformat numbers to ddd.ddd.dddd
regex r(phone); // a regex to find our pattern
string number = "(908) 555-1800";
cout << regex_replace(number, r, fmt) << endl;
morgan (201) 555-2368 862-555-0123
drew (973) 555.0130
lee (609) 555-0132 201.555.0175 800.555-0000
morgan 201.555.2368 862.555.0123
drew 973.555.0130
lee 609.555.0132 201.555.0175 800.555.0000
int main()
{
    string phone =
        "(\\"(?) (\\"d{3}) (\\")) ? ([-. ]) ? (\\"d{3}) ([-. ]) ? (\\"d{4}) ";
    regex r(phone); // a regex to find our pattern
    smatch m;
    string s;
    string fmt = "$2.$5.$7"; // reformat numbers to ddd.ddd.dddd
    // read each record from the input file
    while (getline(cin, s))
        cout << regex_replace(s, r, fmt) << endl;
    return 0;
}

```

```

        using std::regex_constants::format_no_copy;
        using namespace std::regex_constants;
// generate just the phone numbers: use a new format string
string fmt2 = "$2.$5.$7 "; // put space after the last number as a separator
// tell regex_replace to copy only the text that it replaces
cout << regex_replace(s, r, fmt2, format_no_copy) << endl;
201.555.2368 862.555.0123
973.555.0130
609.555.0132 201.555.0175 800.555.0000
default_random_engine e; // generates random unsigned integers
for (size_t i = 0; i < 10; ++i)
    // e() "calls" the object to produce the next random number
    cout << e() << " ";
16807 282475249 1622650073 984943658 1144108930 470211272 ...
// uniformly distributed from 0 to 9 inclusive
uniform_int_distribution<unsigned> u(0,9);
default_random_engine e; // generates unsigned random integers
for (size_t i = 0; i < 10; ++i)
    // u uses e as a source of numbers
    // each call returns a uniformly distributed value in the specified range
    cout << u(e) << " ";
cout << "min: " << e.min() << " max: " << e.max() << endl;
// almost surely the wrong way to generate a vector of random integers
// output from this function will be the same 100 numbers on every call!
vector<unsigned> bad_randVec()
{
    default_random_engine e;
    uniform_int_distribution<unsigned> u(0,9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
vector<unsigned> v1(bad_randVec());
vector<unsigned> v2(bad_randVec());
// will print equal
cout << (v1 == v2) ? "equal" : "not equal" << endl;
// returns a vector of 100 uniformly distributed random numbers
vector<unsigned> good_randVec()
{
    // because engines and distributions retain state, they usually should be
    // defined as static so that new numbers are generated on each call
    static default_random_engine e;
    static uniform_int_distribution<unsigned> u(0,9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}

```

```

default_random_engine e1; // uses the default seed
default_random_engine e2(2147483646); // use the given seed value
// e3 and e4 will generate the same sequence because they use the same seed
default_random_engine e3; // uses the default seed value
e3.seed(32767); // call seed to set a new seed value
default_random_engine e4(32767); // set the seed value to 32767
for (size_t i = 0; i != 100; ++i) {
    if (e1() == e2())
        cout << "unseeded match at iteration: " << i << endl;
    if (e3() != e4())
        cout << "seeded differs at iteration: " << i << endl;
}
default_random_engine e1(time(0)); // a somewhat random seed
default_random_engine e; // generates unsigned random integers
// uniformly distributed from 0 to 1 inclusive
uniform_real_distribution<double> u(0,1);
for (size_t i = 0; i < 10; ++i)
    cout << u(e) << " ";
0.131538 0.45865 0.218959 0.678865 0.934693 0.519416 ...
// empty <> signify we want to use the default result type
uniform_real_distribution<> u(0,1); // generates double by default
default_random_engine e; // generates random integers
normal_distribution<> n(4,1.5); // mean 4, standard deviation 1.5
vector<unsigned> vals(9); // nine elements each 0
for (size_t i = 0; i != 200; ++i) {
    unsigned v = lround(n(e)); // round to the nearest integer
    if (v < vals.size()) // if this result is in range
        ++vals[v]; // count how often each number appears
}
for (size_t j = 0; j != vals.size(); ++j)
    cout << j << ":" << string(vals[j], '*') << endl;
0: ***
1: *****
2: *********
3: *********
4: *********
5: *********
6: *********
7: *****
8: *
string resp;
default_random_engine e; // e has state, so it must be outside the loop!
bernoulli_distribution b; // 50/50 odds by default
do {
    bool first = b(e); // if true, the program will go first
    cout << (first ? "We go first"
              : "You get to go first") << endl;
    // play the game passing the indicator of who goes first
    cout << ((play(first)) ? "sorry, you lost"
              : "congrats, you won") << endl;
    cout << "play again? Enter 'yes' or 'no'" << endl;
} while (cin >> resp && resp[0] == 'y');
bernoulli_distribution b(.55); // give the house a slight edge

```

```

cout << "default bool values: " << true << " " << false
    << "\nalpha bool values: " << bootalpha
    << true << " " << false << endl;
        default bool values: 1 0
            alpha bool values: true false
bool bool_val = get_status();
cout << bootalpha      // sets the internal state of cout
    << bool_val
    << noboolalpha; // resets the internal state to default formatting
cout << "default: " << 20 << " " << 1024 << endl;
cout << "octal: " << oct << 20 << " " << 1024 << endl;
cout << "hex: " << hex << 20 << " " << 1024 << endl;
cout << "decimal: " << dec << 20 << " " << 1024 << endl;
cout << showbase; // show the base when printing integral values
cout << "default: " << 20 << " " << 1024 << endl;
cout << "in octal: " << oct << 20 << " " << 1024 << endl;
cout << "in hex: " << hex << 20 << " " << 1024 << endl;
cout << "in decimal: " << dec << 20 << " " << 1024 << endl;
cout << noshowbase; // reset the state of the stream
cout << uppercase << showbase << hex
    << "printed in hexadecimal: " << 20 << " " << 1024
    << nouppercase << noshowbase << dec << endl;
// cout.precision reports the current precision value
cout << "Precision: " << cout.precision()
    << ", Value: " << sqrt(2.0) << endl;
// cout.precision(12) asks that 12 digits of precision be printed
cout.precision(12);
cout << "Precision: " << cout.precision()
    << ", Value: " << sqrt(2.0) << endl;
// alternative way to set precision using the setprecision manipulator
cout << setprecision(3);
cout << "Precision: " << cout.precision()
    << ", Value: " << sqrt(2.0) << endl;
        Precision: 6, Value: 1.41421
        Precision: 12, Value: 1.41421356237
        Precision: 3, Value: 1.41
cout << "default format: " << 100 * sqrt(2.0) << '\n'
    << "scientific: " << scientific << 100 * sqrt(2.0) << '\n'
    << "fixed decimal: " << fixed << 100 * sqrt(2.0) << '\n'
    << "hexadecimal: " << hexfloat << 100 * sqrt(2.0) << '\n'
    << "use defaults: " << defaultfloat << 100 * sqrt(2.0)
    << "\n\n";
cout << 10.0 << endl;           // prints 10
cout << showpoint << 10.0      // prints 10.0000
    << noshowpoint << endl; // revert to default format for the decimal point

```

```

int i = -16;
double d = 3.14159;
// pad the first column to use a minimum of 12 positions in the output
cout << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';
// pad the first column and left-justify all columns
cout << left
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n'
    << right; // restore normal justification
// pad the first column and right-justify all columns
cout << right
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';
// pad the first column but put the padding internal to the field
cout << internal
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';
// pad the first column, using # as the pad character
cout << setfill('#')
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n'
    << setfill(' '); // restore the normal pad character
cin >> noskipws; // set cin so that it reads whitespace
while (cin >> ch)
    cout << ch;
cin >> skipws; // reset cin to the default state so that it discards whitespace
int ch; // use an int, not a char to hold the return from get()
// loop to read and write all the data in the input
while ((ch = cin.get()) != EOF)
    cout.put(ch);
char ch; // using a char here invites disaster!
// the return from cin.get is converted to char and then compared to an int
while ((ch = cin.get()) != EOF)
    cout.put(ch);
// set the marker to a fixed position
seekg(new_position); // set the read marker to the given pos_type location
seekp(new_position); // set the write marker to the given pos_type location

// offset some distance ahead of or behind the given starting point
seekg(offset, from); // set the read marker offset distance from from
seekp(offset, from); // offset has type off_type
// remember the current write position in mark
ostringstream writeStr; // output stringstream
ostringstream::pos_type mark = writeStr.tellp();
// ...
if (cancelEntry)
    // return to the remembered position
    writeStr.seekp(mark);

```

```

int main()
{
    // open for input and output and preposition file pointers to end-of-file
    // file mode argument see § 8.4 (p. 319)
    fstream inFile("copyOut",
                  fstream::ate | fstream::in | fstream::out);
    if (!inFile) {
        cerr << "Unable to open file!" << endl;
        return EXIT_FAILURE; // EXIT_FAILURE see § 6.3.2 (p. 227)
    }
    // inFile is opened in ate mode, so it starts out positioned at the end
    auto end_mark = inFile.tellg(); // remember original end-of-file position
    inFile.seekg(0, fstream::beg); // reposition to the start of the file
    size_t cnt = 0; // accumulator for the byte count
    string line; // hold each line of input
    // while we haven't hit an error and are still reading the original data
    while (inFile && inFile.tellg() != end_mark
          && getline(inFile, line)) { // and can get another line of input
        cnt += line.size() + 1; // add 1 to account for the newline
        auto mark = inFile.tellg(); // remember the read position
        inFile.seekp(0, fstream::end); // set the write marker to the end
        inFile << cnt; // write the accumulated length
        // print a separator if this is not the last line
        if (mark != end_mark) inFile << " ";
        inFile.seekg(mark); // restore the read position
    }
    inFile.seekp(0, fstream::end); // seek to the end
    inFile << "\n"; // write a newline at end-of-file
    return 0;
}

void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    int *p = new int[v.size()];
    ifstream in("ints");
    // exception occurs here
}
catch (my_error &eObj) { // specifier is a reference type
    eObj.status = errCodes::severeErr; // modifies the exception object
    throw; // the status member of the exception object is severeErr
} catch (other_error eObj) { // specifier is a nonreference type
    eObj.status = errCodes::badErr; // modifies the local copy only
    throw; // the status member of the exception object is unchanged
}

```

```

void manip() {
    try {
        // actions that cause an exception to be thrown
    }
    catch (...) {
        // work to partially handle the exception
        throw;
    }
}
try {
    // use of the C++ standard library
} catch(exception) {
    // ...
} catch(const runtime_error &re) {
    // ...
} catch(overflow_error eobj) { /* ... */ }
int main() {
    // use of the C++ standard library
}
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il) try :
    data(std::make_shared<std::vector<T>>(il)) {
    /* empty body */
} catch(const std::bad_alloc &e) { handle_out_of_memory(e); }
    void recoup(int) noexcept; // won't throw
    void alloc(int);          // might throw
// this function will compile, even though it clearly violates its exception specification
void f() noexcept           // promises not to throw any exception
{
    throw exception(); // violates the exception specification
}
    void recoup(int) noexcept; // recoup doesn't throw
    void recoup(int) throw(); // equivalent declaration
    void recoup(int) noexcept(true); // recoup won't throw
    void alloc(int) noexcept(false); // alloc can throw
noexcept(recoup(i)) // true if calling recoup can't throw, false otherwise
void f() noexcept(noexcept(g())); // f has same exception specifier as g
    // both recoup and pf1 promise not to throw
    void (*pf1)(int) noexcept = recoup;
    // ok: recoup won't throw; it doesn't matter that pf2 might
    void (*pf2)(int) = recoup;
    pf1 = alloc; // error: alloc might throw but pf1 said it wouldn't
    pf2 = alloc; // ok: both pf2 and alloc might throw

```

```

class Base {
public:
    virtual double f1(double) noexcept; // doesn't throw
    virtual int f2() noexcept(false); // can throw
    virtual void f3(); // can throw
};

class Derived : public Base {
public:
    double f1(double); // error: Base::f1 promises not to throw
    int f2() noexcept(false); // ok: same specification as Base::f2
    void f3() noexcept; // ok: Derived f3 is more restrictive
};

// hypothetical exception classes for a bookstore application
class out_of_stock: public std::runtime_error {
public:
    explicit out_of_stock(const std::string &s):
        std::runtime_error(s) { }
};

class isbn_mismatch: public std::logic_error {
public:
    explicit isbn_mismatch(const std::string &s):
        std::logic_error(s) { }
    isbn_mismatch(const std::string &s,
                  const std::string &lhs, const std::string &rhs):
        std::logic_error(s), left(lhs), right(rhs) { }
    const std::string left, right;
};

// throws an exception if both objects do not refer to the same book
Sales_data&
Sales_data::operator+=(const Sales_data& rhs)
{
    if (isbn() != rhs.isbn())
        throw isbn_mismatch("wrong ISBNs", isbn(), rhs.isbn());
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

// use the hypothetical bookstore exceptions
Sales_data item1, item2, sum;
while (cin >> item1 >> item2) { // read two transactions
    try {
        sum = item1 + item2; // calculate their sum
        // use sum
    } catch (const isbn_mismatch &e) {
        cerr << e.what() << ": left ISBN(" << e.left
            << ") right ISBN(" << e.right << ")" << endl;
    }
}

class cplusplus_primer_Query { ... };
string cplusplus_primer_make_plural(size_t, string&);

```

```

namespace cplusplus_primer {
    class Sales_data { /* ... */;
    Sales_data operator+(const Sales_data&,
                           const Sales_data&);
    class Query { /* ... */;
    class Query_base { /* ... */;
} // like blocks, namespaces do not end with a semicolon
cplusplus_primer::Query q =
    cplusplus_primer::Query("hello");
AddisonWesley::Query q = AddisonWesley::Query("hello");
// --- Sales_data.h ---
// #includes should appear before opening the namespace
#include <string>
namespace cplusplus_primer {
    class Sales_data { /* ... */;
    Sales_data operator+(const Sales_data&,
                           const Sales_data&);
    // declarations for the remaining functions in the Sales_data interface
}
// --- Sales_data.cc ---
// be sure any #includes appear before opening the namespace
#include "Sales_data.h"
namespace cplusplus_primer {
    // definitions for Sales_data members and overloaded operators
}
// --- user.cc ---
// names in the Sales_data.h header are in the cplusplus_primer namespace
#include "Sales_data.h"
int main()
{
    using cplusplus_primer::Sales_data;
    Sales_data transl, trans2;
    // ...
    return 0;
}
#include "Sales_data.h"
namespace cplusplus_primer { // reopen cplusplus_primer
    // members defined inside the namespace may use unqualified names
    std::istream&
operator>>(std::istream& in, Sales_data& s) { /* ... */}
}
// namespace members defined outside the namespace must use qualified names
cplusplus_primer::Sales_data
cplusplus_primer::operator+(const Sales_data& lhs,
                           const Sales_data& rhs)
{
    Sales_data ret(lhs);
    // ...
}

```

```

// we must declare the specialization as a member of std
namespace std {
    template <> struct hash<Sales_data>;
}

// having added the declaration for the specialization to std
// we can define the specialization outside the std namespace
template <> struct std::hash<Sales_data>
{
    size_t operator()(const Sales_data& s) const
    { return hash<string>()(s.bookNo) ^
           hash<unsigned>()(s.units_sold) ^
           hash<double>()(s.revenue); }
    // other members as before
};

namespace cplusplus_primer {
    // first nested namespace: defines the Query portion of the library
    namespace QueryLib {
        class Query { /* ... */ };
        Query operator&(const Query&, const Query&);
        // ...
    }

    // second nested namespace: defines the Sales_data portion of the library
    namespace Bookstore {
        class Quote { /* ... */ };
        class Disc_quote : public Quote { /* ... */ };
        // ...
    }

    cplusplus_primer::QueryLib::Query
    inline namespace FifthEd {
        // namespace for the code from the Primer Fifth Edition
    }

    namespace FifthEd { // implicitly inline
        class Query_base { /* ... */ };
        // other Query-related declarations
    }

    namespace FourthEd {
        class Item_base { /* ... */ };
        class Query_base { /* ... */ };
        // other code from the Fourth Edition
    }

    namespace cplusplus_primer {
        #include "FifthEd.h"
        #include "FourthEd.h"
    }
}

int i; // global declaration for i
namespace {
    int i;
}

// ambiguous: defined globally and in an unnested, unnamed namespace
i = 10;

```

```

namespace local {
    namespace {
        int i;
    }
}
// ok: i defined in a nested unnamed namespace is distinct from global i
local::i = 42;
namespace mathLib {
    namespace MatrixLib {
        class matrix { /* ... */ };
        matrix operator*
            (const matrix &, const matrix &);
        // ...
    }
}
namespace cplusplus_primer { /* ... */ };
namespace Qlib = cplusplus_primer::QueryLib;
Qlib::Query q;
// namespaces A and function f are defined at global scope
namespace A {
    int i, j;
}
void f()
{
    using namespace A;      // injects the names from A into the global scope
    cout << i * j << endl; // uses i and j from namespace A
    // ...
}
namespace blip {
    int i = 16, j = 15, k = 23;
    // other declarations
}
int j = 0; // ok: j inside blip is hidden inside a namespace
void manip()
{
    // using directive; the names in blip are "added" to the global scope
    using namespace blip; // clash between ::j and blip::j
    // detected only if j is used
    ++i;           // sets blip::i to 17
    ++j;           // error ambiguous: global j or blip::j?
    ++::j;         // ok: sets global j to 1
    ++blip::j;     // ok: sets blip::j to 16
    int k = 97; // local k hides blip::k
    ++k;           // sets local k to 98
}

```

```

namespace Exercise {
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}
int ivar = 0;
// position 1
void manip() {
    // position 2
    double dvar = 3.1416;
    int iobj = limit + 1;
    ++ivar;
    ++::ivar;
}
namespace A {
    int i;
    namespace B {
        int i;           // hides A::i within B
        int j;
        int f1()
        {
            int j;    // j is local to f1 and hides A::B::j
            return i; // returns B::i
        }
        // namespace B is closed and names in it are no longer visible
        int f2() {
            return j;      // error: j is not defined
        }
        int j = i;       // initialized from A::i
    }
    namespace A {
        int i;
        int k;
        class C1 {
    public:
        C1(): i(0), j(0) { }   // ok: initializes C1::i and C1::j
        int f1() { return k; } // returns A::k
        int f2() { return h; } // error: h is not defined
        int f3();
    private:
        int i;                // hides A::i within C1
        int j;
    };
    int h = i;             // initialized from A::i
}
// member f3 is defined outside class C1 and outside namespace A
int A::C1::f3() { return h; } // ok: returns A::h
using std::operator>>;          // needed to allow cin >> s
std::operator>>(std::cin, s); // ok: explicitly use std::>>

```

```

namespace A {
    class C {
        // two friends; neither is declared apart from a friend declaration
        // these functions implicitly are members of namespace A
        friend void f2();           // won't be found, unless otherwise declared
        friend void f(const C&);   // found by argument-dependent lookup
    };
}
int main()
{
    A::C cobj;
    f(cobj);      // ok: finds A::f through the friend declaration in A::C
    f2();         // error: A::f2 not declared
}
void swap(T v1, T v2)
{
    using std::swap;
    swap(v1.mem1, v2.mem1);
    // swap remaining members of type T
}
namespace NS {
    class Quote { /* ... */ };
    void display(const Quote&){ /* ... */ }
}
// Bulk_item's base class is declared in namespace NS
class Bulk_item : public NS::Quote { /* ... */ };
int main()
{
    Bulk_item book1;
    display(book1);
    return 0;
}
using NS::print(int); // error: cannot specify a parameter list
using NS::print;     // ok: using declarations specify names only
namespace libs_R_us {
    extern void print(int);
    extern void print(double);
}
// ordinary declaration
void print(const std::string &);

// this using directive adds names to the candidate set for calls to print;
using namespace libs_R_us;
// the candidates for calls to print at this point in the program are:
// print(int) from libs_R_us
// print(double) from libs_R_us
// print(const std::string &) declared explicitly
void fooBar(int ival)
{
    print("Value: "); // calls global print(const string &)
    print(ival);      // calls libs_R_us::print(int)
}

```

```
namespace AW {
    int print(int);
}
namespace Primer {
    double print(double);
}

// using directives create an overload set of functions from different namespaces
using namespace AW;
using namespace Primer;

long double print(long double);

int main() {
    print(1);    // calls AW::print(int)
    print(3.1); // calls Primer::print(double)
    return 0;
}

namespace primerLib {
    void compute();
    void compute(const void *);
}
using primerLib::compute;
void compute(int);
void compute(double, double = 3.4);
void compute(char*, char* = 0);
void f()
{
    compute(0);
}

class Bear : public ZooAnimal {
class Panda : public Bear, public Endangered { /* ... */ };

// explicitly initialize both base classes
Panda::Panda(std::string name, bool onExhibit)
    : Bear(name, onExhibit, "Panda"),
      Endangered(Endangered::critical) { }

// implicitly uses the Bear default constructor to initialize the Bear subobject
Panda::Panda()
    : Endangered(Endangered::critical) { }
```

```

struct Basel {
    Basel() = default;
    Basel(const std::string&);
    Basel(std::shared_ptr<int>);
};

struct Base2 {
    Base2() = default;
    Base2(const std::string&);
    Base2(int);
};

// error: D1 attempts to inherit D1::D1 (const string&) from both base classes
struct D1: public Basel, public Base2 {
    using Basel::Basel; // inherit constructors from Basel
    using Base2::Base2; // inherit constructors from Base2
};

// D2 must define its own constructor that takes a string
struct D2: public Basel, public Base2 {
    using Basel::Basel; // inherit constructors from Basel
    using Base2::Base2; // inherit constructors from Base2
    D2(const string &s): Basel(s), Base2(s) { }
    D2() = default; // needed once D2 defines its own constructor
};

Panda ying_yang("ying_yang");
Panda ling_ling = ying_yang; // uses the copy constructor
// operations that take references to base classes of type Panda
void print(const Bear&);

void highlight(const Endangered&);

ostream& operator<<(ostream&, const ZooAnimal&);

Panda ying_yang("ying_yang");

print(ying_yang); // passes Panda to a reference to Bear
highlight(ying_yang); // passes Panda to a reference to Endangered
cout << ying_yang << endl; // passes Panda to a reference to ZooAnimal
    class A { ... };
    class B : public A { ... };
    class C : public B { ... };
    class X { ... };
    class Y { ... };
    class Z : public X, public Y { ... };
    class MI : public C, public Z { ... };
    void print(const Bear&);
    void print(const Endangered&);

Panda ying_yang("ying_yang");
print(ying_yang); // error: ambiguous
Bear *pb = new Panda("ying_yang");

pb->print(); // ok: Panda::print()
pb->curl(); // error: not part of the Bear interface
pb->highlight(); // error: not part of the Bear interface
delete pb; // ok: Panda::~Panda()

```

```

Endangered *pe = new Panda("ying_yang");
pe->print();           // ok: Panda::print()
pe->toes();            // error: not part of the Endangered interface
pe->cuddle();          // error: not part of the Endangered interface
pe->highlight();       // ok: Panda::highlight()
delete pe;              // ok: Panda::~Panda()
class D1 : public Basel { /* ... */ };
class D2 : public Base2 { /* ... */ };
class MI : public D1, public D2 { /* ... */ };
    double d = ying_yang.max_weight();
double Panda::max_weight() const
{
    return std::max(ZooAnimal::max_weight(),
                     Endangered::max_weight());
}
struct Basel {
    void print(int) const;      // public by default
protected:
    int    ival;
    double dval;
    char   cval;
private:
    int    *id;
};
struct Base2 {
    void print(double) const;    // public by default
protected:
    double fval;
private:
    double dval;
};
struct Derived : public Basel {
    void print(std::string) const; // public by default
protected:
    std::string sval;
    double      dval;
};
struct MI : public Derived, public Base2 {
    void print(std::vector<double>); // public by default
protected:
    int             *ival;
    std::vector<double>  dvec;
};
// the order of the keywords public and virtual is not significant
class Raccoon : public virtual ZooAnimal { /* ... */ };
class Bear : virtual public ZooAnimal { /* ... */ };
    class Panda : public Bear,
                  public Raccoon, public Endangered {
};

```

```

void dance(const Bear&);
void rummage(const Raccoon&);
ostream& operator<<(ostream&, const ZooAnimal&);
Panda ying_yang;
dance(ying_yang); // ok: passes Panda object as a Bear
rummage(ying_yang); // ok: passes Panda object as a Raccoon
cout << ying_yang; // ok: passes Panda object as a ZooAnimal
struct Base {
    void bar(int); // public by default
protected:
    int ival;
};

struct Derived1 : virtual public Base {
    void bar(char); // public by default
    void foo(char);
protected:
    char cval;
};

struct Derived2 : virtual public Base {
    void foo(int); // public by default
protected:
    int ival;
    char cval;
};

class VMI : public Derived1, public Derived2 { };
Bear::Bear(std::string name, bool onExhibit):
    ZooAnimal(name, onExhibit, "Bear") {}
Raccoon::Raccoon(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Raccoon") {}
Panda::Panda(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Panda"),
      Bear(name, onExhibit),
      Raccoon(name, onExhibit),
      Endangered(Endangered::critical),
      sleeping_flag(false) {}

class Character {/* ... */};
class BookCharacter : public Character {/* ... */};
class ToyAnimal {/* ... */};

class TeddyBear : public BookCharacter,
                  public Bear, public virtual ToyAnimal
{ /* ... */};

ZooAnimal(); // Bear's virtual base class
ToyAnimal(); // direct virtual base class
Character(); // indirect base class of first nonvirtual base class
BookCharacter(); // first direct nonvirtual base class
Bear(); // second direct nonvirtual base class
TeddyBear(); // most derived class

```

```

        class Class { ... };
        class Base : public Class { ... };
        class D1 : virtual public Base { ... };
        class D2 : virtual public Base { ... };
        class MI : public D1, public D2 { ... };
        class Final : public MI, public Class { ... };

// newexpressions
string *sp = new string("a value"); // allocate and initialize a string
string *arr = new string[10]; // allocate ten default initialized strings
    delete sp; // destroy *sp and free the memory to which sp points
    delete [] arr; // destroy the elements in the array and free the memory
        // these versions might throw an exception
void *operator new(size_t); // allocate an object
void *operator new[](size_t); // allocate an array
void *operator delete(void*) noexcept; // free an object
void *operator delete[](void*) noexcept; // free an array

// versions that promise not to throw; see § 12.1.2 (p. 460)
void *operator new(size_t, nothrow_t&) noexcept;
void *operator new[](size_t, nothrow_t&) noexcept;
void *operator delete(void*, nothrow_t&) noexcept;
void *operator delete[](void*, nothrow_t&) noexcept;
void *operator new(size_t, void*); // this version may not be redefined
void *operator new(size_t size) {
    if (void *mem = malloc(size))
        return mem;
    else
        throw bad_alloc();
}
void operator delete(void *mem) noexcept { free(mem); }
    new (place_address) type
    new (place_address) type (initializers)
    new (place_address) type [size]
    new (place_address) type [size] { braced initializer list }

string *sp = new string("a value"); // allocate and initialize a string
sp->~string();
    if (Derived *dp = dynamic_cast<Derived*>(bp))
    {
        // use the Derived object to which dp points
    } else { // bp points at a Base object
        // use the Base object to which bp points
    }
void f(const Base &b)
{
    try {
        const Derived &d = dynamic_cast<const Derived&>(b);
        // use the Derived object to which b referred
    } catch (bad_cast) {
        // handle the fact that the cast failed
    }
}

```

```

        class A { /* ... */ };
        class B : public A { /* ... */ };
        class C : public B { /* ... */ };
        class D : public B, public A { /* ... */ };
        if (C *pc = dynamic_cast< C*>(pa))
            // use C's members
        } else {
            // use A's members
        }
    Derived *dp = new Derived;
    Base *bp = dp; // both pointers point to a Derived object
    // compare the type of two objects at run time
    if (typeid(*bp) == typeid(*dp)) {
        // bp and dp point to objects of the same type
    }
    // test whether the run-time type is a specific type
    if (typeid(*bp) == typeid(Derived)) {
        // bp actually points to a Derived
    }
    // test always fails: the type of bp is pointer to Base
    if (typeid(bp) == typeid(Derived)) {
        // code never executed
    }
    class Base {
        friend bool operator==(const Base&, const Base&);
    public:
        // interface members for Base
    protected:
        virtual bool equal(const Base&) const;
        // data and other implementation members of Base
    };
    class Derived: public Base {
    public:
        // other interface members for Derived
    protected:
        bool equal(const Base&) const;
        // data and other implementation members of Derived
    };
    bool operator==(const Base &lhs, const Base &rhs)
    {
        // returns false if typeids are different; otherwise makes a virtual call to equal
        return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
    }
    bool Derived::equal(const Base &rhs) const
    {
        // we know the types are equal, so the cast won't throw
        auto r = dynamic_cast<const Derived&>(rhs);
        // do the work to compare two Derived objects and return the result
    }
}

```

```

        bool Base::equal(const Base &rhs) const
    {
        // do whatever is required to compare to Base objects
    }
    int arr[10];
    Derived d;
    Base *p = &d;
    cout << typeid(42).name() << ", "
        << typeid(arr).name() << ", "
        << typeid(Sales_data).name() << ", "
        << typeid(std::string).name() << ", "
        << typeid(p).name() << ", "
        << typeid(*p).name() << endl;
i, A10_i, 10Sales_data, Ss, P4Base, 7Derived
    class A { /* ... */ };
    class B : public A { /* ... */ };
    class C : public B { /* ... */ };

    (a) A *pa = new C;
        cout << typeid(pa).name() << endl;
    (b) C cobj;
        A& ra = cobj;
        cout << typeid(&ra).name() << endl;
    (c) B *px = new B;
        A& ra = *px;
        cout << typeid(ra).name() << endl;
    enum class open_modes {input, output, append};
    enum color {red, yellow, green};           // unscoped enumeration
    // unnamed, unscoped enum
    enum {floatPrec = 6, doublePrec = 10, double_doublePrec = 10};
    enum color {red, yellow, green};           // unscoped enumeration
    enum stoplight {red, yellow, green};       // error: redefines enumerators
    enum class peppers {red, yellow, green};   // ok: enumerators are hidden
    color eyes = green; // ok: enumerators are in scope for an unscoped enumeration
    peppers p = green; // error: enumerators from peppers are not in scope
                    // color::green is in scope but has the wrong type
    color hair = color::red; // ok: we can explicitly access the enumerators
    peppers p2 = peppers::red; // ok: using red from peppers
    enum class intTypes {
        charTyp = 8, shortTyp = 16, intTyp = 16,
        longTyp = 32, long_longTyp = 64
    };
    constexpr intTypes charbits = intTypes::charTyp;
    open_modes om = 2;           // error: 2 is not of type open_modes
    om = open_modes::input; // ok: input is an enumerator of open_modes
    int i = color::red; // ok: unscoped enumerator implicitly converted to int
    int j = peppers::red; // error: scoped enumerations are not implicitly converted
    enum intValues : unsigned long long {
        charTyp = 255, shortTyp = 65535, intTyp = 65535,
        longTyp = 4294967295UL,
        long_longTyp = 18446744073709551615ULL
    };

```

```

// forward declaration of unscoped enum named intValues
enum intValues : unsigned long long; // unscoped, must specify a type
enum class open_modes; // scoped enums can use int by default
// error: declarations and definition must agree whether the enum is scoped or unscoped
enum class intValues;
enum intValues; // error: intValues previously declared as scoped enum
enum intValues : long; // error: intValues previously declared as int
// unscoped enumeration; the underlying type is machine dependent
enum Tokens {INLINE = 128, VIRTUAL = 129};
void ff(Tokens);
void ff(int);
int main() {
    Tokens curTok = INLINE;
    ff(128); // exactly matches ff(int)
    ff(INLINE); // exactly matches ff(Tokens)
    ff(curTok); // exactly matches ff(Tokens)
    return 0;
}
void newf(unsigned char);
void newf(int);
unsigned char uc = VIRTUAL;
newf(VIRTUAL); // calls newf(int)
newf(uc); // calls newf(unsigned char)
class Screen {
public:
    typedef std::string::size_type pos;
    char get_cursor() const { return contents[cursor]; }
    char get() const;
    char get(pos ht, pos wd) const;
private:
    std::string contents;
    pos cursor;
    pos height, width;
};
// pdata can point to a string member of a const (or nonconst) Screen object
const string Screen::*pdata;
Screen myScreen, *pScreen = &myScreen;
// .* dereferences pdata to fetch the contents member from the object myScreen
auto s = myScreen.*pdata;
// ->* dereferences pdata to fetch contents from the object to which pScreen points
s = pScreen->*pdata;
class Screen {
public:
    // data is a static member that returns a pointer to member
    static const std::string Screen::*data()
        { return &Screen::contents; }
    // other members as before
};
// data() returns a pointer to the contents member of class Screen
const string Screen::*pdata = Screen::data();
// fetch the contents of the object named myScreen
auto s = myScreen.*pdata;

```

```

// pmf is a pointer that can point to a Screen member function that is const
// that returns a char and takes no arguments
auto pmf = &Screen::get_cursor;
    char (Screen::*pmf2)(Screen::pos, Screen::pos) const;
    pmf2 = &Screen::get;
        // error: nonmember function p cannot have a const qualifier
        char Screen::*p(Screen::pos, Screen::pos) const;
// pmf points to a Screen member that takes no arguments and returns char
pmf = &Screen::get; // must explicitly use the address-of operator
pmf = Screen::get; // error: no conversion to pointer for member functions
Screen myScreen, *pScreen = &myScreen;
// call the function to which pmf points on the object to which pScreen points
char c1 = (pScreen->*pmf)();
// passes the arguments 0, 0 to the two-parameter version of get on the object myScreen
char c2 = (myScreen.*pmf2)(0, 0);
    // Action is a type that can point to a member function of Screen
    // that returns a char and takes two pos arguments
    using Action =
        char (Screen::*)(Screen::pos, Screen::pos) const;
Action get = &Screen::get; // get points to the get member of Screen
// action takes a reference to a Screen and a pointer to a Screen member function
Screen& action(Screen&, Action = &Screen::get);
    Screen myScreen;

// equivalent calls:
action(myScreen); // uses the default argument
action(myScreen, get); // uses the variable get that we previously defined
action(myScreen, &Screen::get); // passes the address explicitly
    class Screen {
public:
    // other interface and implementation members as before
    Screen& home(); // cursor movement functions
    Screen& forward();
    Screen& back();
    Screen& up();
    Screen& down();
};
class Screen {
public:
    // other interface and implementation members as before
    // Action is a pointer that can be assigned any of the cursor movement members
    using Action = Screen& (Screen::*());
    // specify which direction to move; enum see § 19.3 (p. 832)
    enum Directions { HOME, FORWARD, BACK, UP, DOWN };
    Screen& move(Directions);
private:
    static Action Menu[]; // function table
};
Screen& Screen::move(Directions cm)
{
    // run the element indexed by cm on this object
    return (this->*Menu[cm])(); // Menu[cm] points to a member function
}

```

```

Screen myScreen;
myScreen.move(Screen::HOME); // invokes myScreen.home
myScreen.move(Screen::DOWN); // invokes myScreen.down
Screen::Action Screen::Menu[] = { &Screen::home,
                                  &Screen::forward,
                                  &Screen::back,
                                  &Screen::up,
                                  &Screen::down,
                                };
auto pmf = &Screen::get_cursor;
pmf = &Screen::get;
auto fp = &string::empty; // fp points to the string empty function
// error: must use .* or ->* to call a pointer to member
find_if(svec.begin(), svec.end(), fp);
// check whether the given predicate applied to the current element yields true
if (fp(*it)) // error: must use ->* to call through a pointer to member
    function<bool (const string&)> fcn = &string::empty;
    find_if(svec.begin(), svec.end(), fcn);
// assuming it is the iterator inside find_if, so *it is an object in the given range
if (fcn(*it)) // assuming fcn is the name of the callable inside find_if
// assuming it is the iterator inside find_if, so *it is an object in the given range
if (((*it).*p)()) // assuming p is the pointer to member function inside fcn
    vector<string*> pvec;
    function<bool (const string*)> fp = &string::empty;
    // fp takes a pointer to string and uses the ->* to call empty
    find_if(pvec.begin(), pvec.end(), fp);
    find_if(svec.begin(), svec.end(), mem_fn(&string::empty));
auto f = mem_fn(&string::empty); // f takes a string or a string*
f(*svec.begin()); // ok: passes a string object; f uses .* to call empty
f(&svec[0]); // ok: passes a pointer to string; f uses .-> to call empty
// bind each string in the range to the implicit first argument to empty
auto it = find_if(svec.begin(), svec.end(),
                  bind(&string::empty, _1));
auto f = bind(&string::empty, _1);
f(*svec.begin()); // ok: argument is a string f will use .* to call empty
f(&svec[0]); // ok: argument is a pointer to string f will use .-> to call empty
class TextQuery {
public:
    class QueryResult; // nested class to be defined later
    // other members as in § 12.3.2 (p. 487)
};
// we're defining the QueryResult class that is a member of class TextQuery
class TextQuery::QueryResult {
    // in class scope, we don't have to qualify the name of the QueryResult parameters
    friend std::ostream&
        print(std::ostream&, const QueryResult&);
public:
    // no need to define QueryResult::line_no; a nested class can use a member
    // of its enclosing class without needing to qualify the member's name
    QueryResult(std::string,
                std::shared_ptr<std::set<line_no>>,
                std::shared_ptr<std::vector<std::string>>);
    // other members as in § 12.3.2 (p. 487)
};

```

```
// defining the member named QueryResult for the class named QueryResult
// that is nested inside the class TextQuery
TextQuery::QueryResult::QueryResult(string s,
                                     shared_ptr<set<line_no>> p,
                                     shared_ptr<vector<string>> f):
    sought(s), lines(p), file(f) { }
// defines an int static member of QueryResult
// which is a class nested inside TextQuery
int TextQuery::QueryResult::static_mem = 1024;
// return type must indicate that QueryResult is now a nested class
TextQuery::QueryResult
TextQuery::query(const string &sought) const
{
    // we'll return a pointer to this set if we don't find sought
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // use find and not a subscript to avoid adding words to wm!
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // not found
    else
        return QueryResult(sought, loc->second, file);
}
return QueryResult(sought, loc->second, file);
// objects of type Token have a single member, which could be of any of the listed types
union Token {
    // members are public by default
    char    cval;
    int     ival;
    double  dval;
};
Token first_token = {'a'}; // initializes the cval member
Token last_token;         // uninitialized Token object
Token *pt = new Token;    // pointer to an uninitialized Token object
union {                   // anonymous union
    char    cval;
    int     ival;
    double  dval;
}; // defines an unnamed object, whose members we can access directly
cval = 'c'; // assigns a new value to the unnamed, anonymous union object
ival = 42;   // that object now holds the value 42
```

```

class Token {
public:
    // copy control needed because our class has a union with a string member
    // defining the move constructor and move-assignment operator is left as an exercise
    Token(): tok(INT), ival{0} { }
    Token(const Token &t): tok(t.tok) { copyUnion(t); }
    Token &operator=(const Token&);
    // if the union holds a string, we must destroy it; see § 19.1.2 (p. 824)
    ~Token() { if (tok == STR) sval.~string(); }
    // assignment operators to set the differing members of the union
    Token &operator=(const std::string&);
    Token &operator=(char);
    Token &operator=(int);
    Token &operator=(double);
private:
    enum { INT, CHAR, DBL, STR } tok; // discriminant
    union { // anonymous union
        char cval;
        int ival;
        double dval;
        std::string sval;
    }; // each Token object has an unnamed member of this unnamed union type
    // check the discriminant and copy the union member as appropriate
    void copyUnion(const Token&);

};

Token &Token::operator=(int i)
{
    if (tok == STR) sval.~string(); // if we have a string, free it
    ival = i; // assign to the appropriate member
    tok = INT; // update the discriminant
    return *this;
}

Token &Token::operator=(const std::string &s)
{
    if (tok == STR) // if we already hold a string, just do an assignment
        sval = s;
    else
        new(&sval) string(s); // otherwise construct a string
    tok = STR; // update the discriminant
    return *this;
}

void Token::copyUnion(const Token &t)
{
    switch (t.tok) {
        case Token::INT: ival = t.ival; break;
        case Token::CHAR: cval = t.cval; break;
        case Token::DBL: dval = t.dval; break;
        // to copy a string, construct it using placement new; see (§ 19.1.2 (p. 824))
        case Token::STR: new(&sval) string(t.sval); break;
    }
}

```

```

Token &Token::operator=(const Token &t)
{
    // if this object holds a string and t doesn't, we have to free the old string
    if (tok == STR && t.tok != STR) sval.~string();
    if (tok == STR && t.tok == STR)
        sval = t.sval; // no need to construct a new string
    else
        copyUnion(t); // will construct a string if t.tok is STR
    tok = t.tok;
    return *this;
}
int a, val;
void foo(int val)
{
    static int si;
    enum Loc { a = 1024, b };
    // Bar is local to foo
    struct Bar {
        Loc locVal; // ok: uses a local type name
        int barVal;
        void fooBar(Loc l = a) // ok: default argument is Loc::a
        {
            barVal = val; // error: val is local to foo
            barVal = ::val; // ok: uses a global object
            barVal = si; // ok: uses a static local object
            locVal = b; // ok: uses an enumerator
        }
    };
    // ...
}
void foo()
{
    class Bar {
    public:
        // ...
        class Nested; // declares class Nested
    };
    // definition of Nested
    class Bar::Nested {
        // ...
    };
}

```

```

typedef unsigned int Bit;

class File {
    Bit mode: 2;           // mode has 2 bits
    Bit modified: 1;       // modified has 1 bit
    Bit prot_owner: 3;     // prot_owner has 3 bits
    Bit prot_group: 3;     // prot_group has 3 bits
    Bit prot_world: 3;     // prot_world has 3 bits
    // operations and data members of File

public:
    // file modes specified as octal literals; see § 2.1.3 (p. 38)
    enum modes { READ = 01, WRITE = 02, EXECUTE = 03 };
    File &open(modes);
    void close();
    void write();
    bool isRead() const;
    void setWrite();
};

void File::write()
{
    modified = 1;
    // ...
}

void File::close()
{
    if (modified)
        // ...save contents
}

File &File::open(File::modes m)
{
    mode |= READ;          // set the READ bit by default
    // other processing
    if (m & WRITE) // if opening READ and WRITE
        // processing to open the file in read/write mode
    return *this;
}

inline bool File::isRead() const { return mode & READ; }
inline void File::setWrite() { mode |= WRITE; }

volatile int display_register; // int value that might change
volatile Task *curr_task; // curr_task points to a volatile object
volatile int iax[max_size]; // each element in iax is volatile
volatile Screen bitmapBuf; // each member of bitmapBuf is volatile
volatile int v; // v is a volatile int
int *volatile vip; // vip is a volatile pointer to int
volatile int *ivp; // ivp is a pointer to volatile int
// vivp is a volatile pointer to volatile int
volatile int *volatile vivp;

int *ip = &v; // error: must use a pointer to volatile
*ivp = &v; // ok: ivp is a pointer to volatile
vivp = &v; // ok: vivp is a volatile pointer to volatile

```

```

class Foo {
public:
    Foo(const volatile Foo&); // copy from a volatile object
    // assign from a volatile object to a nonvolatile object
    Foo& operator=(volatile const Foo&);
    // assign from a volatile object to a volatile object
    Foo& operator=(volatile const Foo&) volatile;
    // remainder of class Foo
};

// illustrative linkage directives that might appear in the C++ header <cstring>
// single-statement linkage directive
extern "C" size_t strlen(const char *);

// compound-statement linkage directive
extern "C" {
    int strcmp(const char*, const char*);
    char *strcat(char*, const char*);
}
// compound-statement linkage directive
extern "C" {
#include <string.h>      // C functions that manipulate C-style strings
}

// pf points to a C function that returns void and takes an int
extern "C" void (*pf)(int);
void (*pf1)(int); // points to a C++ function
extern "C" void (*pf2)(int); // points to a C function
pf1 = pf2; // error: pf1 and pf2 have different types
            // f1 is a C function; its parameter is a pointer to a C function
extern "C" void f1(void(*)(int));
// FC is a pointer to a C function
extern "C" typedef void FC(int);

// f2 is a C++ function with a parameter that is a pointer to a C function
void f2(FC *);
// the calc function can be called from C programs
extern "C" double calc(double dparm) { /* ... */ }
#ifndef __cplusplus
// ok: we're compiling C++
extern "C"
#endif
    int strcmp(const char*, const char*);
    // error: two extern "C" functions with the same name
    extern "C" void print(const char*);
    extern "C" void print(int);
    class SmallInt { /* ... */ };
    class BigNum { /* ... */ };

// the C function can be called from C and C++ programs
// the C++ functions overload that function and are callable from C++
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);

```

```
    find(beg, end, val)
    find_if(beg, end, unaryPred)
    find_if_not(beg, end, unaryPred)
    count(beg, end, val)
    count_if(beg, end, unaryPred)
        all_of(beg, end, unaryPred)
        any_of(beg, end, unaryPred)
        none_of(beg, end, unaryPred)
    adjacent_find(beg, end)
    adjacent_find(beg, end, binaryPred)
    search_n(beg, end, count, val)
    search_n(beg, end, count, val, binaryPred)
    search(beg1, end1, beg2, end2)
    search(beg1, end1, beg2, end2, binaryPred)
    find_first_of(beg1, end1, beg2, end2)
    find_first_of(beg1, end1, beg2, end2, binaryPred)
        find_end(beg1, end1, beg2, end2)
        find_end(beg1, end1, beg2, end2, binaryPred)
            for each(beg, end, unaryOp)
                mismatch(beg1, end1, beg2)
                mismatch(beg1, end1, beg2, binaryPred)
                equal(beg1, end1, beg2)
                equal(beg1, end1, beg2, binaryPred)
                    lower_bound(beg, end, val)
                    lower_bound(beg, end, val, comp)
                    upper_bound(beg, end, val)
                    upper_bound(beg, end, val, comp)
                    equal_range(beg, end, val)
                    equal_range(beg, end, val, comp)
                binary_search(beg, end, val)
                binary_search(beg, end, val, comp)
                    fill(beg, end, val)
                    fill_n(dest, cnt, val)
                    generate(beg, end, Gen)
                    generate_n(dest, cnt, Gen)
                copy(beg, end, dest)
                copy_if(beg, end, dest, unaryPred)
                copy_n(beg, n, dest)
                transform(beg, end, dest, unaryOp)
                transform(beg, end, beg2, dest, binaryOp)
    replace_copy(beg, end, dest, old_val, new_val)
    replace_copy_if(beg, end, dest, unaryPred, new_val)
        merge(beg1, end1, beg2, end2, dest)
        merge(beg1, end1, beg2, end2, dest, comp)
            iter_swap(iter1, iter2)
            swap_ranges(beg1, end1, beg2)
        replace(beg, end, old_val, new_val)
        replace_if(beg, end, unaryPred, new_val)
            copy_backward(beg, end, dest)
            move_backward(beg, end, dest)
        inplace_merge(beg, mid, end)
        inplace_merge(beg, mid, end, comp)
```

```
    is_partitioned(beg, end, unaryPred)
partition_copy(beg, end, dest1, dest2, unaryPred)
    partition_point(beg, end, unaryPred)
stable_partition(beg, end, unaryPred)
partition(beg, end, unaryPred)
    sort(beg, end)
stable_sort(beg, end)
sort(beg, end, comp)
stable_sort(beg, end, comp)
is_sorted(beg, end)
is_sorted(beg, end, comp)
is_sorted_until(beg, end)
is_sorted_until(beg, end, comp)
partial_sort(beg, mid, end)
partial_sort(beg, mid, end, comp)
partial_sort_copy(beg, end, destBeg, destEnd)
partial_sort_copy(beg, end, destBeg, destEnd, comp)
    nth_element(beg, nth, end)
    nth_element(beg, nth, end, comp)
remove(beg, end, val)
remove_if(beg, end, unaryPred)
remove_copy(beg, end, dest, val)
remove_copy_if(beg, end, dest, unaryPred)
unique(beg, end)
unique(beg, end, binaryPred)
unique_copy(beg, end, dest)
unique_copy_if(beg, end, dest, binaryPred)
    rotate(beg, mid, end)
    rotate_copy(beg, mid, end, dest)
    reverse(beg, end)
    reverse_copy(beg, end, dest)
random_shuffle(beg, end)
random_shuffle(beg, end, rand)
shuffle(beg, end, Uniform_rand)
is_permutation(beg1, end1, beg2)
is_permutation(beg1, end1, beg2, binaryPred)
    next_permutation(beg, end)
    next_permutation(beg, end, comp)
    prev_permutation(beg, end)
    prev_permutation(beg, end, comp)
includes(beg, end, beg2, end2)
includes(beg, end, beg2, end2, comp)
set_union(beg, end, beg2, end2, dest)
set_union(beg, end, beg2, end2, dest, comp)
set_intersection(beg, end, beg2, end2, dest)
set_intersection(beg, end, beg2, end2, dest, comp)
    set_difference(beg, end, beg2, end2, dest)
    set_difference(beg, end, beg2, end2, dest, comp)
set_symmetric_difference(beg, end, beg2, end2, dest)
set_symmetric_difference(beg, end, beg2, end2, dest, comp)
```

```
    minmax(val1, val2)
    minmax(val1, val2, comp)
    minmax(init_list)
    minmax(init_list, comp)
min_element(beg, end)
min_element(beg, end, comp)
max_element(beg, end)
max_element(beg, end, comp)
minmax_element(beg, end)
minmax_element(beg, end, comp)
lexicographical_compare(beg1, end1, beg2, end2)
lexicographical_compare(beg1, end1, beg2, end2, comp)
    accumulate(beg, end, init)
    accumulate(beg, end, init, binaryOp)
inner_product(beg1, end1, beg2, init)
inner_product(beg1, end1, beg2, init, binOp1, binOp2)
    partial_sum(beg, end, dest)
    partial sum(beg, end, dest, binaryOp)
adjacent_difference(beg, end, dest)
adjacent_difference(beg, end, dest, binaryOp)
uniform_int_distribution<IntT> u(m, n);
uniform_real_distribution<RealT> u(x, y);
bernoulli_distribution b(p);
binomial_distribution<IntT> b(t, p);
geometric_distribution<IntT> g(p);
negative_binomial_distribution<IntT> nb(k, p);
poisson_distribution<IntT> p(x);
exponential_distribution<RealT> e(lam);
gamma_distribution<RealT> g(a, b);
weibull_distribution<RealT> w(a, b);
extreme_value_distribution<RealT> e(a, b);
normal_distribution<RealT> n(m, s);
lognormal_distribution<RealT> ln(m, s);
chi_squared_distribution<RealT> c(x);
cauchy_distribution<RealT> c(a, b);
fisher_f_distribution<RealT> f(m, n);
student_t_distribution<RealT> s(n);
discrete_distribution<IntT> d(i, j);
discrete_distribution<IntT> d{il};
piecewise_constant_distribution<RealT> pc(b, e, w);
piecewise_linear_distribution<RealT> pl(b, e, w);
    default_random_engine
    linear_congruential_engine
    mersenne_twister_engine
    subtract_with_carry_engine
    discard_block_engine
    independent_bits_engine
    shuffle_order_engine
```