Table 7-2 summarizes the arithmetic operators.

Table 7-2: Arithmetic Operators

Operator	Name	Examples	Result
`+x`	Unary plus	`+10`	`10`
`-x`	Unary minus	`-10`	`-10`
`x + y`	Binary addition	`1 + 2`	`3`
`x - y`	Binary subtraction	`1 - 2`	`-1`
`x * y`	Binary multiplication	`10 * 20`	`200`
`x / y`	Binary division	`300 / 15`	`20`
`x % y`	Binary modulo	`42 % 5`	`2`

Many of the binary operators in Tables 7-1 and 7-2 have corollary as *assignment operators* as well.

Assignment Operators

An assignment operator performs a given operation and then assigns the result to the first operand. For example, the *addition assignment* `x += y` computes the value `x + y` and assigns `x` equal to the result. You can achieve similar results with the expression `x = x + y`, but the *assignment operator* is more syntactically compact and at least as runtime efficient. Table 7-3 summarizes all of the available assignment operators.

Table 7-3: Assignment Operators

Operator	Name	Examples	Result (value of x)
`x = y`	Simple assignment	`x = 10`	`10`
`x += y`	Addition assignment	`x += 10`	`15`
`x -= y`	Subtraction assignment	`x -= 10`	`-5`
`x *= y`	Multiplication assignment	`x *= 10`	`50`
`x /= y`	Division assignment	`x /= 2`	`2`
`x %= y`	Modulo assignment	`x %= 2`	`1`
`x &= y`	Bitwise AND assignment	`x &= 0b1100`	`0b0100`
`x |= y`	Bitwise OR assignment	`x |= 0b1100`	`0b1101`
`x ^= y`	Bitwise XOR assignment	`x ^= 0b1100`	`0b1001`
`x <<= y`	Bitwise left-shift assignment	`x <<= 2`	`0b10100`
`x >>= y`	Bitwise right-shift assignment	`x >>= 2`	`0b0001`

NOTE *Promotion rules don't really apply when using assignment operators; the type of the assigned to operand won't change. For example, given* `int x = 5`*, the type of* `x` *after* `x /= 2.0f` *is still* `int`*.*

Increment and Decrement Operators

There are four (unary) *increment/decrement* operators, as outlined in Table 7-4.

Table 7-4: The Increment and Decrement Operators (values given for x=5)

Operator	Name	Value of x after evaluation	Value of expression
++x	Prefix increment	6	6
x++	Postfix increment	6	5
--x	Prefix decrement	4	4
x--	Postfix decrement	4	5

As Table 7-4 shows, increment operators increase the value of their operand by 1, whereas decrement operators decrease by 1. The value returned by the operator depends on whether it is prefix or postfix. A prefix operator will return the value of the operand after modification, whereas a postfix operator will return the value before modification.

Comparison Operators

Six comparison operators compare the given operands and evaluate to a `bool`, as outlined in Table 7-5. For arithmetic operands, the same type conversions (promotions) occur as with the arithmetic operators. The comparison operators also work with pointers, and they work approximately how you would expect them to.

NOTE *There are some nuances to pointer comparison. Interested readers should refer to* [expr.rel].

Table 7-5: The Comparison Operators

Operator	Name	Examples (all evaluate to `true`)
x == y	Equal-to operator	`100 == 100`
x != y	Not-equal-to operator	`100 != 101`
x < y	Less-than operator	`10 < 20`
x > y	Greater-than operator	`-10 > -20`
x <= y	Less-than-or-equal-to operator	`10 <= 10`
x >= y	Greater-than-or-equal-to operator	`20 >= 10`

Member Access Operators

You use *member access operators* to interact with pointers, arrays, and many of the classes you'll meet in Part II. The six such operators include *subscript* `[]`, *indirection* `*`, *address-of* `&`, *member-of-object* `.`, and *member-of-pointer* `->`. You met these operators in Chapter 3, but this section provides a brief summary.

NOTE *There are also* pointer-to-member-of-object `.*` *and* pointer-to-member-of-pointer `->*` *operators, but these are uncommon. Refer to [expr.mptr.oper].*

The subscript operator `x[y]` provides access to the `y`th element of the array pointed to by `x`, whereas the indirection operator `*x` provides access to the element pointed to by `x`. You can create a pointer to an element `x` using the address-of operator `&x`. This is essentially the inverse operation to the indirection operator. For elements `x` with a member `y`, you use the member-of-object operator `x.y`. You can also access members of a pointed-to object; given a pointer `x`, you use the member-of-pointer operator `x->y` to access an object pointed to by `x`.

Ternary Conditional Operator

The *ternary conditional operator* `x ? y : z` is a lump of syntactic sugar that takes three operands (hence "ternary"). It evaluates the first operand `x` as a Boolean expression and returns the second operand `y` or the third operand `z` depending on whether the Boolean is `true` or `false` (respectively). Consider the following step function that returns 1 if the parameter `input` is positive; otherwise, it returns zero:

```
int step(int input) {
  return input > 0 ? 1 : 0;
}
```

Using an equivalent if-then statement, you could also implement `step` the following way:

```
int step(int input) {
  if (input > 0) {
    return 1;
  } else {
    return 0;
  }
}
```

These two approaches are runtime equivalent, but the ternary conditional operator requires less typing and usually results in cleaner code. Use it generously.

NOTE *The conditional ternary operator has a more fashionable moniker: the* Elvis operator. *If you rotate the book 90 degrees clockwise and squint, you'll see why:* `?:`

The Comma Operator

The *comma operator*, on the other hand, doesn't usually promote cleaner code. It allows several expressions separated by commas to be evaluated within a larger expression. The expressions evaluate from left to right, and the rightmost expression is the return value, as Listing 7-1 illustrates.

```
#include <cstdio>

int confusing(int &x) {
  return x = 9, x++, x / 2;
}

int main() {
  int x{}; ❶
  auto y = confusing(x); ❷
  printf("x: %d\ny: %d", x, y);
}
```

```
x: 10
y: 5
```

Listing 7-1: A confusing function employing the comma operator

After invoking `confusing`, `x` equals `10` ❶ and `y` equals `5` ❷.

NOTE *A vestigial structure from C's wilder and altogether less-inhibited college days, the comma operator permits a particular kind of expression-oriented programming. Eschew the comma operator; its use is exceedingly uncommon and likely to sow confusion.*

Operator Overloading

For each fundamental type, some portion of the operators covered in this section will be available. For user-defined types, you can specify custom behavior for these operators by employing *operator overloading*. To specify behavior for an operator in a user-defined class, simply name the method with the word `operator` immediately followed by the operator; ensure that the return types and parameters match the types of the operands you want to deal with.

Listing 7-2 defines a `CheckedInteger`.

```
#include <stdexcept>

struct CheckedInteger {
  CheckedInteger(unsigned int value) : value{ value } ❶ { }

  CheckedInteger operator+(unsigned int other) const { ❷
    CheckedInteger result{ value + other }; ❸
    if (result.value < value) throw std::runtime_error{ "Overflow!" }; ❹
    return result;
  }

  const unsigned int value; ❺
};
```

Listing 7-2: A `CheckedInteger` class that detects overflow at runtime

In this class, you've defined a constructor that takes a single `unsigned int`. This argument is used ❶ to member initialize the private field `value` ❺. Because `value` is `const`, `CheckedInteger` is *immutable*—after construction, it's not possible to modify the state of a `CheckedInteger`. The method of interest here is `operator+` ❷, which allows you to add an ordinary `unsigned int` to a `CheckedInteger` to produce a new `CheckedInteger` with the correct `value`. The return value of `operator+` is constructed at ❸. Whenever addition results in the overflow of an `unsigned int`, the result will be less than the original values. You check for this condition at ❹. If an overflow is detected, you throw an exception.

Chapter 6 described `type_traits`, which allow you to determine features of your types at compile time. A related family of type support is available in the `<limits>` header, which allows you to query various properties of arithmetic types.

Within `<limits>`, the template class `numeric_limits` exposes a number of member constants that provide information about the template parameter. One such example is the `max()` method, which returns the highest finite value of a given type. You can use this method to kick the tires of the `CheckedInteger` class. Listing 7-3 illustrates the behavior of the `CheckedInteger`.

```
#include <limits>
#include <cstdio>
#include <stdexcept>

struct CheckedInteger {
  --snip--
};

int main() {
  CheckedInteger a{ 100 }; ❶
  auto b = a + 200; ❷
  printf("a + 200 = %u\n", b.value);
  try {
    auto c = a + std::numeric_limits<unsigned int>::max(); ❸
  } catch(const std::overflow_error& e) {
    printf("(a + max) Exception: %s\n", e.what());
  }
}
```

```
a + 200 = 300
(a + max) Exception: Overflow!
```

Listing 7-3: A program illustrating the use of `CheckedInteger`

After constructing a `CheckedInteger` ❶, you can add it to an `unsigned int` ❷. Because the resulting value, 300, is guaranteed to fit inside an `unsigned int`, this statement executes without throwing an exception. Next, you add the same `CheckedInteger a` to the maximum value of an `unsigned int` via `numeric_limits` ❸. This causes an overflow, which is detected by the `operator+` overload and results in a thrown `overflow_error`.

Overloading Operator new

Recall from Chapter 4 that you allocate objects with dynamic storage duration using operator `new`. By default, operator `new` will allocate memory on the free store to make space for your dynamic objects. The *free store*, also known as the *heap*, is an implementation-defined storage location. On desktop operating systems, the kernel usually manages the free store (see the `HeapAlloc` on Windows and `malloc` on Linux and macOS) and is generally vast.

Free Store Availability

In some environments, like the Windows kernel or embedded systems, there is no free store available to you by default. In other settings, such as game development or high-frequency trading, free store allocations simply involve too much latency, because you've delegated its management to the operating system.

You could try to avoid using the free store entirely, but this is severely limiting. One major limitation this would introduce is to preclude the use of stdlib containers, which after reading Part II you'll agree is a major loss. Rather than settling for these severe restrictions, you can overload the free store operations and take control over allocations. You do this by overloading operator `new`.

The <new> Header

In environments that support free store operations, the `<new>` header contains the following four operators:

- `void* operator new(size_t);`
- `void operator delete(void*);`
- `void* operator new[](size_t);`
- `void operator delete[](void*);`

Notice that the return type of operator `new` is `void*`. The free store operators deal in raw, uninitialized memory.

It's possible to provide your own versions of these four operators. All you do is define them once in your program. The compiler will use your versions rather than the defaults.

Free store management is a surprisingly complicated task. One of the major issues is *memory fragmentation*. Over time, large numbers of memory allocations and releases can leave free blocks of memory scattered throughout the region dedicated for the free store. It's possible to get into situations where there is plenty of free memory, but it's scattered across allocated memory. When this happens, large requests for memory will fail, even though there is technically enough free memory to provide to the requester. Figure 7-1 illustrates such a situation. There is plenty of memory for the desired allocation, but the available memory is noncontiguous.

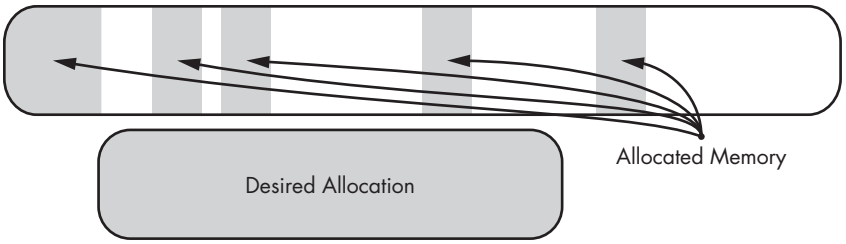


Figure 7-1: The memory fragmentation problem

Buckets

One approach is to chop allocated memory into so-called *buckets* of a fixed size. When you request memory, the environment allocates a whole bucket, even if you didn't request all the memory. For example, Windows provides two functions for allocating dynamic memory: `VirtualAllocEx` and `HeapAlloc`.

The `VirtualAllocEx` function is low level, which allows you to provide many options, such as which process to allocate memory into, the preferred memory address, the requested size, and permissions, like whether the memory should be readable, writable, and executable. This function will never allocate fewer than 4096 bytes (a so-called *page*).

On the other hand, `HeapAlloc` is a higher-level function that hands out less than a page of memory when it can; otherwise, it will invoke `VirtualAllocEx` on your behalf. At least with the Visual Studio compiler, `new` will call `HeapAlloc` by default.

This arrangement prevents memory fragmentation in exchange for some overhead associated with rounding up allocations to bucket size. Modern operating systems like Windows will have fairly complex schemes for allocating memory of different sizes. You don't see any of this complexity unless you want to take control.

Taking Control of the Free Store

Listing 7-4 demonstrates implementing very simple `Bucket` and `Heap` classes. These will facilitate taking control over dynamic memory allocation:

```
#include <cstddef>
#include <new>

struct Bucket { ❶
  const static size_t data_size{ 4096 };
  std::byte data[data_size];
};

struct Heap {
  void* allocate(size_t bytes) { ❷
    if (bytes > Bucket::data_size) throw std::bad_alloc{};
    for (size_t i{}; i < n_heap_buckets; i++) {
      if (!bucket_used[i]) {
        bucket_used[i] = true;
        return buckets[i].data;
```

```
      }
    }
    throw std::bad_alloc{};
  }

  void free(void* p) { ❸
    for (size_t i{}; i < n_heap_buckets; i++) {
      if (buckets[i].data == p) {
        bucket_used[i] = false;
        return;
      }
    }
  }
  static const size_t n_heap_buckets{ 10 };
  Bucket buckets[n_heap_buckets]{}; ❹
  bool bucket_used[n_heap_buckets]{}; ❺
};
```

Listing 7-4: `Heap` and `Bucket` classes

The `Bucket` class ❶ is responsible for taking up space in memory. As an homage to the Windows heap manager, the bucket size is hardcoded to 4096. All of the management logic goes into the `Heap` class.

Two important accounting members are in `Heap`: `buckets` ❹ and `bucket_used` ❺. The `buckets` member houses all the `Buckets`, neatly packed into a contiguous string. The `bucket_used` member is a relatively tiny array containing objects of type `bool` that keeps track of whether a `Bucket` in `buckets` with the same index has been loaned out yet. Both members are initialized to zero.

The `Heap` class has two methods: `allocate` ❷ and `free` ❸. The `allocate` method first checks whether the number of bytes requested is greater than the bucket size. If it is, it throws a `std::bad_alloc` exception. Once the size check passes, `Heap` iterates through the `buckets` looking for one that isn't marked `true` in `bucket_used`. If it finds one, it returns the `data` member pointer for the associated `Bucket`. If it can't find an unused `Bucket`, it throws a `std::bad_alloc` exception. The `free` method accepts a `void*` and iterates through all the `buckets` looking for a matching `data` member pointer. If it finds one, it sets `bucket_used` for the corresponding bucket to `false` and returns.

Using Our Heap

One way to allocate a `Heap` is to declare it at namespace scope so it has static storage duration. Because its lifetime begins when the program starts, you can use it inside the `operator new` and `operator delete` overrides, as shown in Listing 7-5.

```
Heap heap; ❶

void* operator new(size_t n_bytes) {
  return heap.allocate(n_bytes); ❷
}
```

```
void operator delete(void* p) {
  return heap.free(p); ❸
}
```

Listing 7-5: Overriding the `new` and `delete` operators to use the `Heap` class from Listing 7-4

Listing 7-5 declares a `Heap` ❶ and uses it inside the operator `new` overload ❷ and the operator `delete` overload ❸. Now if you use `new` and `delete`, dynamic memory management will use `heap` instead of the default free store offered by the environment. Listing 7-6 kicks the tires of the overloaded dynamic memory management.

```
#include <cstdio>
--snip--
int main() {
  printf("Buckets:   %p\n", heap.buckets); ❶
  auto breakfast = new unsigned int{ 0xC0FFEE };
  auto dinner = new unsigned int { 0xDEADBEEF };
  printf("Breakfast: %p 0x%x\n", breakfast, *breakfast); ❷
  printf("Dinner:    %p 0x%x\n", dinner, *dinner); ❸
  delete breakfast;
  delete dinner;
  try {
    while (true) {
      new char;
      printf("Allocated a char.\n"); ❹
    }
  } catch (const std::bad_alloc&) {
    printf("std::bad_alloc caught.\n"); ❺
  }
}
```

```
Buckets:   00007FF792EE3320 ❶
Breakfast: 00007FF792EE3320 0xc0ffee ❷
Dinner:    00007FF792EE4320 0xdeadbeef ❸
Allocated a char. ❹
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
Allocated a char.
std::bad_alloc caught. ❺
```

Listing 7-6: A program illustrating the use of `Heap` to manage dynamic allocations

You've printed the memory address of the first `buckets` element of the `heap` ❶. This is the memory location loaned out to the first `new` invocation. You verify that this is the case by printing the memory address and value pointed to by `breakfast` ❷. Notice that the memory address matches the memory address of the first `Bucket` in `heap`. You've done the same for

the memory pointed to by `dinner` ❸. Notice that the memory address is exactly `0x1000` greater than that of `breakfast`. This coincides exactly with the 4096-byte length of a `Bucket`, as defined in the `const static` member `Bucket::data_size`.

After printing ❷❸, you delete `breakfast` and `dinner`. Then, you allocate `char` objects with reckless abandon until a `std::bad_alloc` is thrown when `heap` runs out of memory. Each time you make an allocation, you print `Allocated a char.` starting at ❹. There are 10 lines before you see a `std::bad_alloc` exception ❺. Notice that this is exactly the number of `buckets` you've set in `Heap::n_heap_buckets`. This means that, for each `char` you've allocated, you've taken up 4096 bytes of memory!

Placement Operators

Sometimes, you don't want to override *all* free store allocations. In such situations, you can use the placement operators, which perform the appropriate initialization on preallocated memory:

- `void* operator new(size_t, void*);`
- `void operator delete(size_t, void*);`
- `void* operator new[](void*, void*);`
- `void operator delete[](void*, void*);`

Using placement operators, you can manually construct objects in arbitrary memory. This has the advantage of enabling you to manually manipulate an object's lifetime. However, you cannot use `delete` to release the resulting dynamic objects. You must call the object's destructor directly (and exactly once!), as demonstrated in Listing 7-7.

```
#include <cstdio>
#include <cstddef>
#include <new>

struct Point {
  Point() : x{}, y{}, z{} {
    printf("Point at %p constructed.\n", this); ❶
  }
  ~Point() {
    printf("Point at %p destructed.\n", this); ❷
  }
  double x, y, z;
};

int main() {
  const auto point_size = sizeof(Point);
  std::byte data[3 * point_size];
  printf("Data starts at %p.\n", data); ❸
  auto point1 = new(&data[0 * point_size]) Point{}; ❹
  auto point2 = new(&data[1 * point_size]) Point{}; ❺
  auto point3 = new(&data[2 * point_size]) Point{}; ❻
  point1->~Point(); ❼
```

```
  point2->~Point(); ❽
  point3->~Point(); ❾
}
```

```
Data starts at 0000004D290FF8E8. ❸
Point at 0000004D290FF8E8 constructed. ❹
Point at 0000004D290FF900 constructed. ❺
Point at 0000004D290FF918 constructed. ❻
Point at 0000004D290FF8E8 destructed. ❼
Point at 0000004D290FF900 destructed. ❽
Point at 0000004D290FF918 destructed. ❾
```

Listing 7-7: Using placement `new` to initialize dynamic objects

The constructor ❶ prints a message indicating that a `Point` at a particular address was constructed, and the destructor ❷ prints a corresponding message indicating that the `Point` is getting destructed. You've printed the address of `data`, which is the first address where placement `new` initializes a `Point` ❸.

Observe that each placement `new` has allocated the `Point` within the memory occupied by your `data` array ❹❺❻. You must invoke each destructor individually ❼❽❾.

Operator Precedence and Associativity

When more than one operator appears in an expression, *operator precedence* and *operator associativity* decide how the expression parses. Operators with higher precedence are bound tighter to their arguments than operators with lower precedence. If two operators have the same precedence, their associativity breaks the tie to decide how arguments bind. Associativity is either *left to right* or *right to left*.

Table 7-6 contains every C++ operator sorted by its precedence and annotated with its associativity. Each row contains one or more operators with the same precedence along with a description and its associativity. Higher rows have higher precedence.

Table 7-6: Operator Precedence and Associativity

Operator	Description	Associativity
`a::b`	Scope resolution	Left to right
`a++`	Postfix increment	Left to right
`a--`	Postfix decrement	
`fn()`	Function call	
`a[b]`	Subscript	
`a->b`	Member of pointer	
`a.b`	Member of object	
`Type(a)`	Functional cast	
`Type{ a }`	Functional cast	

Operator	Description	Associativity
`++a`	Prefix increment	Right to left
`--a`	Prefix decrement	
`+a`	Unary plus	
`-a`	Unary minus	
`!a`	Logical NOT	
`~a`	Bitwise complement	
`(Type)a`	C-style cast	
`*a`	Dereference	
`&a`	Address of	
`sizeof(Type)`	Size of	
`new Type`	Dynamic allocation	
`new Type[]`	Dynamic allocation (array)	
`delete a`	Dynamic deallocation	
`delete[] a`	Dynamic deallocation (array)	
`.*`	Pointer-to-member-of-pointer	Left to right
`->*`	Pointer-to-member-of-object	
`a * b`	Multiplication	Left to right
`a / b`	Division	
`a % b`	Modulo division	
`a + b`	Addition	Left to right
`a - b`	Subtraction	
`a << b`	Bitwise left shift	Left to right
`a >> b`	Bitwise right shift	
`a < b`	Less than	Left to right
`a > b`	Greater than	
`a <= b`	Less than or equal to	
`a >= b`	Greater than or equal to	
`a == b`	Equals	Left to right
`a != b`	Not equals	
`a & b`	Bitwise AND	Left to right
`a ^ b`	Bitwise AND	Left to right
`a | b`	Bitwise OR	Left to right
`a && b`	Logical AND	Left to right
`a || b`	Logical OR	Left to right
`a ? b : c`	Ternary	Right to left
`throw a`	Throw	
`a = b`	Assignment	
`a += b`	Sum assignment	
`a -= b`	Difference assignment	
`a *= b`	Product assignment	
`a /= b`	Quotient assignment	
`a %= b`	Remainder assignment	
`a <<= b`	Bitwise-left-shift assignment	
`a >>= b`	Bitwise-right-shift assignment	
`a &= b`	Bitwise AND assignment	
`a ^= b`	Bitwise XOR assignment	
`a |= b`	Bitwise OR assignment	
`a, b`	Comma	Left to right

NOTE *You haven't yet met the scope resolution operator (it first appears in Chapter 8), but Table 7-6 includes it for completeness.*

Because C++ has many operators, the operator precedence and associativity rules can be hard to keep track of. For the mental health of those reading your code, try to make expressions as clear as possible.

Consider the following expression:

```
*a++ + b * c
```

Because postfix addition has higher precedence than the dereference operator *, it binds first to the argument `a`, meaning the result of `a++` is the argument to the dereference operator. Multiplication * has higher precedence than addition `+`, so the multiplication operator * binds to `b` and `c`, and the addition operator `+` binds to the results of `*a++` and `b * c`.

You can impose precedence within an expression by adding parentheses, which have higher precedence than any operator. For example, you can rewrite the preceding expression using parentheses:

```
(*(a++)) + (b * c)
```

As a general rule, add parentheses wherever a reader could become confused about operator precedence. If the result is a bit ugly (as in this example), your expression is probably too complicated; you might consider breaking it up into multiple statements.

Evaluation Order

Evaluation order determines the execution sequence of operators in an expression. A common misconception is that precedence and evaluation order are equivalent: they are not. *Precedence* is a compile time concept that drives how operators bind to operands. *Evaluation order* is a runtime concept that drives the scheduling of operator execution.

In general, C++ has no clearly specified execution order for operands. Although operators bind to operands in the well-defined way explained in the preceding sections, those operands evaluate in an undefined order. The compiler can order operand evaluation however it likes.

You might be tempted to think that the parentheses in the following expression drive evaluation order for the functions `stop`, `drop`, and `roll`, or that some left-to-right associativity has some runtime effect:

```
(stop() + drop()) + roll()
```

They do not. The `roll` function might execute before, after, or between evaluations of `stop` and `drop`. If you require operations to execute in a specific

order, simply place them into separate statements in the desired sequence, as shown here:

```
auto result = stop();
result = result + drop();
result = result + roll();
```

If you aren't careful, you can even get undefined behavior. Consider the following expression:

```
b = ++a + a;
```

Because the ordering of the expressions `++a` and `a` is not specified, and because the value of `++a + a` depends on which expression evaluates first, the value of `b` cannot be well defined.

In some special situations, execution order is specified by the language. The most commonly encountered scenarios are as follows:

- The built-in logical AND operator `a && b` and built-in logical OR operator `a || b` guarantee that `a` executes before `b`.
- The ternary operator `a ? b : c` guarantees that `a` executes before `b` and `c`.
- The comma operator `a, b` guarantees that `a` executes before `b`.
- The constructor arguments in a `new` expression evaluate before the call to the allocator function.

You might be wondering why C++ doesn't enforce execution order, say from left to right, to avoid confusion. The answer is simply that by not arbitrarily constraining execution order, the language is allowing compiler writers to find clever optimization opportunities.

NOTE *For more information on execution order, see [expr].*

User-Defined Literals

Chapter 2 covered how to declare literals, constant values that you use directly in your programs. These help the compiler to turn embedded values into the desired types. Each fundamental type has its own syntax for literals. For example, a `char` literal is declared in single quotes like `'J'`, whereas a `wchar_t` is declared with an `L` prefix like `L'J'`. You can specify the precision of floating-point numbers using either the `F` or `L` suffix.

For convenience, you can also make your own *user-defined literals*. As with the baked-in literals, these provide you with some syntactical support for giving type information to the compiler. Although you'd rarely ever need to declare a user-defined literal, it's worth mentioning because you might find them in libraries. The stdlib `<chrono>` header uses literals extensively to give programmers a clean syntax for using time types—for

example, `700ms` denotes 700 milliseconds. Because user-defined literals are fairly rare, I won't cover them in any more detail here.

NOTE *For further reference, see Section 19.2.6 of* The C++ Programming Language, *4th Edition, by Bjarne Stroustrup.*

Type Conversions

You perform type conversions when you have one type but want to convert it to another type. Depending on the situation, type conversions can be explicit or implicit. This section treats both sorts of conversions while covering promotions, floating-point-to-integer conversions, integer-to-integer conversions, and floating-point-to-floating-point conversions.

Type conversions are fairly common. For example, you might need to compute the mean of some integers given a count and a sum. Because the count and sum are stored in variables of integral type (and you don't want to truncate fractional values), you'll want to compute the mean as a floating-point number. To do this, you'll need to use type conversion.

Implicit Type Conversions

Implicit type conversions can occur anywhere a particular type is called for but you provide a different type. These conversions occur in several different contexts.

"Binary Arithmetic Operators" on page 183 outlined so-called *promotion rules*. In fact, these are a form of implicit conversion. Whenever an arithmetic operation occurs, shorter integral types are promoted to `int` types. Integral types can also be promoted to floating-point types during arithmetic operation. All of this happens in the background. The result is that, in most situations, the type system simply gets out of your way so you can focus on programming logic.

Unfortunately, in some situations, C++ is a bit overzealous in silently converting types. Consider the following implicit conversion from a `double` to a `uint_8`:

```
#include <cstdint>

int main() {
  auto x = 2.718281828459045235360287471352 7L;
  uint8_t y = x; // Silent truncation
}
```

You should hope that the compiler will generate a warning here, but technically this is valid C++. Because this conversion loses information, it's a narrowing conversion that would be prevented by braced initialization {}:

```
#include <cstdint>

int main() {
```

```
  auto x = 2.71828182845904523536028747135 27L;
  uint8_t y{ x }; // Bang!
}
```

Recall that braced initialization doesn't permit narrowing conversions. Technically, the braced initializer is an explicit conversion, so I'll discuss that in "Explicit Type Conversion" on page 201.

Floating-Point-to-Integer Conversion

Floating-point and integral types can coexist peacefully within arithmetic expressions. The reason is implicit type conversion: when the compiler encounters mixed types, it performs the necessary promotions so arithmetic proceeds as expected.

Integer-to-Integer Conversion

Integers can be converted into other integer types. If the destination type is `signed`, all is well, as long as the value can be represented. If it cannot, the behavior is implementation defined. If the destination type is `unsigned`, the result is as many bits as can fit into the type. In other words, the high-order bits are lost.

Consider the example in Listing 7-8, which demonstrates how you can get undefined behavior resulting from signed conversion.

```
#include <cstdint>
#include <cstdio>

int main() {
  // 0b111111111 = 511
  uint8_t x = 0b111111111; ❶// 255
  int8_t y =  0b111111111; ❷// Implementation defined.
  printf("x: %u\ny: %d", x, y);
}
```

```
x: 255 ❶
y: -1 ❷
```

Listing 7-8: Undefined behavior resulting from signed conversion

Listing 7-8 implicitly casts an integer that is too big to fit in an 8-bit integer (511, or 9 bits of ones) into `x` and `y`, which are `unsigned` and `signed`. The value of `x` is guaranteed to be 255 ❶, whereas the value of `y` is implementation dependent. On a Windows 10 x64 machine, `y` equals `-1` ❷. The assignment of both `x` and `y` involve narrowing conversions that could be avoided using the braced initialization syntax.

Floating-Point-to-Floating-Point Conversions

Floating-point numbers can be implicitly cast to and from other floating-point numbers. As long as the destination value can fit the source value, all is well. When it cannot, you have undefined behavior. Again, braced

initialization can prevent potentially dangerous conversions. Consider the example in Listing 7-9, which demonstrates undefined behavior resulting from a narrowing conversion.

```
#include <limits>
#include <cstdio>

int main() {
  double x = std::numeric_limits<float>::max(); ❶
  long double y = std::numeric_limits<double>::max(); ❷
  float z = std::numeric_limits<long double>::max(); ❸  // Undefined Behavior
  printf("x: %g\ny: %Lg\nz: %g", x, y, z);
}
```

```
x: 3.40282e+38
y: 1.79769e+308
z: inf
```

Listing 7-9: Undefined behavior resulting from narrowing conversion

You have completely safe implicit conversions from `float` to `double` ❶ and `double` to `long double` ❷ respectively. Unfortunately, assigning the maximum value of a `long double` to a `float` results in undefined behavior ❸.

Conversion to bool

Pointers, integers, and floating-point numbers can all be implicitly converted to `bool` objects. If the value is nonzero, the result of implicit conversion is `true`. Otherwise, the result is `false`. For example, the value `int{ 1 }` converts to `true`, and the value `int{}` converts to `false`.

Pointer to void*

Pointers can always be implicitly converted to `void*`, as demonstrated in Listing 7-10.

```
#include <cstdio>

void print_addr(void* x) {
  printf("0x%p\n", x);
}

int main() {
  int x{};
  print_addr(&x); ❶
  print_addr(nullptr); ❷
}
```

```
0x000000F79DCFFB74 ❶
0x0000000000000000 ❷
```

Listing 7-10: Implicit pointer conversion to void. Output is from a Windows 10 x64 machine.*

Listing 7-10 compiles thanks to the pointers' implicit conversion to `void*`. The address refers to the address of `x` ❶ and prints `0` ❷.

Explicit Type Conversion

Explicit type conversions are also called *casts*. The first port of call for conducting an explicit type conversion is braced initialization {}. This approach has the major benefit of being fully type safe and non-narrowing. The use of braced initialization ensures at compile time that only safe, well-behaved, non-narrowing conversions are allowed. Listing 7-11 shows an example.

```
#include <cstdio>
#include <cstdint>

int main() {
  int32_t a = 100;
  int64_t b{ a }; ❶
  if (a == b) printf("Non-narrowing conversion!\n"); ❷
  //int32_t c{ b }; // Bang! ❸
}
```

```
Non-narrowing conversion! ❷
```

Listing 7-11: Explicit type conversion for 4- and 8-byte integers

This simple example uses braced initialization ❶ to build an `int64_t` from an `int32_t`. This is a well-behaved conversion because you're guaranteed not to have lost any information. You can always store 32 bits inside 64 bits. After a well-behaved conversion of a fundamental type, the original will always equal the result (according to `operator==`).

The example attempts a badly behaved (narrowing) conversion ❸. The compiler will generate an error. If you hadn't used the braced initializer {}, the compiler wouldn't have complained, as demonstrated in Listing 7-12.

```
#include <limits>
#include <cstdio>
#include <cstdint>

int main() {
  int64_t b = std::numeric_limits<int64_t>::max();
  int32_t c(b); ❶ // The compiler abides.
  if (c != b) printf("Narrowing conversion!\n"); ❷
}
```

```
Narrowing conversion! ❷
```

Listing 7-12: A refactor of Listing 7-11 without the braced initializer.

You make a narrowing conversion from a 64-bit integer to a 32-bit integer ❶. Because this narrows, the expression `c != b` evaluates to `true` ❷. This behavior is very dangerous, which is why Chapter 2 recommends using the braced initializer as much as possible.

C-Style Casts

Recall from Chapter 6 that the named conversion functions allow you to perform dangerous casts that braced initialization won't permit. You can also perform C-style casts, but this is done mainly to maintain some compatibility between the languages. Their usage is as follows:

```
(desired-type)object-to-cast
```

For each C-style cast, there exists some incantation of `static_casts`, `const_casts`, and `reinterpret_casts` that would achieve the desired type conversion. C-style casts are far more dangerous than the named casts (and this is saying quite a bit).

The syntax of the C++ explicit casts is intentionally ugly and verbose. This calls attention to a point in the code where the rigid rules of the type system are being bent or broken. The C-style cast doesn't do this. In addition, it's not clear from the cast what kind of conversion the programmer is intending. When you use finer instruments like the named casts, the compiler can at least enforce *some* constraints. For example, it's all too easy to forget `const` correctness when using a C-style cast when you only intended a `reinterpret_cast`.

Suppose you wanted to treat a `const char*` array as unsigned within the body of a function. It would be too easy to write code like that demonstrated in Listing 7-13.

```
#include <cstdio>

void trainwreck(const char* read_only) {
  auto as_unsigned = (unsigned char*)read_only;
  *as_unsigned = 'b'; ❶ // Crashes on Windows 10 x64
}

int main() {
  auto ezra = "Ezra";
  printf("Before trainwreck: %s\n", ezra);
  trainwreck(ezra);
  printf("After trainwreck: %s\n", ezra);
}
```

```
Before trainwreck: Ezra
```

Listing 7-13: A train wreck of a C-style cast that accidentally gets rid of the `const` qualifier on `read_only`. (This program has undefined behavior; output is from a Windows 10 x64 machine.)

Modern operating systems enforce memory access patterns. Listing 7-13 attempts to write into the memory storing the string literal `Ezra` ❶. On Windows 10 x64, this crashes the program with a memory access violation (it's read-only memory).

If you tried this with a `reinterpret_cast`, the compiler would generate an error, as Listing 7-14 demonstrates.

```
#include <cstdio>

void trainwreck(const char* read_only) {
  auto as_unsigned = reinterpret_cast<unsigned char*>(read_only); ❶
  *as_unsigned = 'b'; // Crashes on Windows 10 x64
}

int main() {
  auto ezra = "Ezra";
  printf("Before trainwreck: %s\n", ezra);
  trainwreck(ezra);
  printf("After trainwreck: %s\n", ezra);
}
```

Listing 7-14: A refactor of Listing 7-13 using a `static_cast`. (This code does not compile.)

If you really intended to throw away `const` correctness, you'd need to tack on a `const_cast` here ❶. The code would self-document these intentions and make such intentional rule breakages easy to find.

User-Defined Type Conversions

In user-defined types, you can provide user-defined conversion functions. These functions tell the compiler how your user-defined types behave during implicit and explicit type conversion. You can declare these conversion functions using the following usage pattern:

```
struct MyType {
  operator destination-type() const {
    // return a destination-type from here.
  --snip--
  }
}
```

For example, the `struct` in Listing 7-15 can be used like a read-only `int`.

```
struct ReadOnlyInt {
  ReadOnlyInt(int val) : val{ val } { }
  operator int() const { ❶
    return val;
  }
private:
  const int val;
};
```

Listing 7-15: A `ReadOnlyInt` class containing a user-defined type conversion to an `int`

The operator `int` method at ❶ defines the user-defined type conversion *from* a `ReadOnlyInt` *to* an `int`. You can now use `ReadOnlyInt` types just like regular `int` types thanks to implicit conversion:

```
struct ReadOnlyInt {
  --snip--
```

```
};
int main() {
  ReadOnlyInt the_answer{ 42 };
  auto ten_answers = the_answer * 10; // int with value 420
}
```

Sometimes, implicit conversions can cause surprising behavior. You should always try to use explicit conversions, especially with user-defined types. You can achieve explicit conversions with the `explicit` keyword. Explicit constructors instruct the compiler not to consider the constructor as a means for implicit conversion. You can provide the same guidelines for your user-defined conversion functions:

```
struct ReadOnlyInt {
  ReadOnlyInt(int val) : val{ val } { }
  explicit operator int() const {
    return val;
  }
private:
  const int val;
};
```

Now, you must explicitly cast a `ReadOnlyInt` to an `int` using `static_cast`:

```
struct ReadOnlyInt {
  --snip--
};
int main() {
  ReadOnlyInt the_answer{ 42 };
  auto ten_answers = static_cast<int>(the_answer) * 10;
}
```

Generally, this approach tends to promote less ambiguous code.

Constant Expressions

Constant expressions are expressions that can be evaluated at compile time. For performance and safety reasons, whenever a computation can be done at compile time rather than runtime, you should do it. Simple mathematical operations involving literals are an obvious example of expressions that can be evaluated at compile time.

You can extend the reach of the compiler by using the expression `constexpr`. Whenever all the information required to compute an expression is present at compile time, the compiler is *compelled to do so* if that expression is marked `constexpr`. This simple commitment can enable a surprisingly large impact on code readability and runtime performance.

Both `const` and `constexpr` are closely related. Whereas `constexpr` enforces that an expression is compile time evaluable, `const` enforces that a variable cannot change within some scope (at runtime). All `constexpr` expressions are `const` because they're always fixed at runtime.

All constexpr expressions begin with one or more fundamental types (int, float, whchar_t, and so on). You can build on top of these types by using operators and constexpr functions. Constant expressions are used mainly to replace manually computed values in your code. This generally produces code that is more robust and easier to understand, because you can eliminate so-called *magic values*—manually calculated constants copy and pasted directly into source code.

A Colorful Example

Consider the following example where some library you're using for your project uses Color objects that are encoded using the hue-saturation-value (HSV) representation:

```
struct Color {
  float H, S, V;
};
```

Very roughly, hue corresponds with a family of colors like red, green, or orange. Saturation corresponds with colorfulness or intensity. Value corresponds with the color's brightness.

Suppose you want to instantiate Color objects using red-green-blue (RGB) representations. You could use a converter to calculate the RGB to HSV manually, but this is a prime example where you can use constexpr to eliminate magic values. Before you can write the conversion function, you need a few utility functions, namely min, max, and modulo. Listing 7-16 implements these functions.

```
#include <cstdint>
constexpr uint8_t max(uint8_t a, uint8_t b) { ❶
  return a > b ? a : b;
}
constexpr uint8_t max(uint8_t a, uint8_t b, uint8_t c) { ❷
  return max(max(a, b), max(a, c));
}
constexpr uint8_t min(uint8_t a, uint8_t b) { ❸
  return a < b ? a : b;
}
constexpr uint8_t min(uint8_t a, uint8_t b, uint8_t c) { ❹
  return min(min(a, b), min(a, c));
}
constexpr float modulo(float dividend, float divisor) { ❺
  const auto quotient = dividend / divisor; ❻
  return divisor * (quotient - static_cast<uint8_t>(quotient));
}
```

Listing 7-16: Several constexpr functions for manipulating uint8_t objects

Each function is marked constexpr, which tells the compiler that the function must be evaluable at compile time. The max function ❶ uses the ternary operator to return the value of the argument that is greatest. The

three-argument version of `max` ❷ uses the transitive property of comparison; by evaluating the two-argument `max` for the pairs `a, b` and `a, c`, you can find the max of this intermediate result to find the overall max. Because the two-argument version of max is `constexpr`, this is totally legal.

NOTE *You can't use `fmax` from the `<math.h>` header for the same reason: it's not `constexpr`.*

The `min` versions ❸ ❹ follow exactly with the obvious modification that the comparison is flipped. The `modulo` function ❺ is a quick-and-dirty, `constexpr` version of the C function `fmod`, which computes the floating-point remainder of dividing the first argument (`dividend`) by the second argument (`divisor`). Because `fmod` is *not* `constexpr`, you've hand-rolled your own. First, you obtain the quotient ❻. Next, you subtract the integral part of quotient using a `static_cast` and a subtraction. Multiplying the decimal portion of the quotient by `divisor` yields the result.

With a collection of `constexpr` utility functions in your arsenal, you can now implement your conversion function `rgb_to_hsv`, as demonstrated in Listing 7-17.

```
--snip--
constexpr Color rgb_to_hsv(uint8_t r, uint8_t g, uint8_t b) {
  Color c{}; ❶
  const auto c_max = max(r, g, b);
  c.V = c_max / 255.0f; ❷

  const auto c_min = min(r, g, b);
  const auto delta = c.V - c_min / 255.0f;
  c.S = c_max == 0 ? 0 : delta / c.V; ❸

  if (c_max == c_min) { ❹
    c.H = 0;
    return c;
  }
  if (c_max == r) {
    c.H = (g / 255.0f - b / 255.0f) / delta;
  } else if (c_max == g) {
    c.H = (b / 255.0f - r / 255.0f) / delta + 2.0f;
  } else if (c_max == b) {
    c.H = (r / 255.0f - g / 255.0f) / delta + 4.0f;
  }
  c.H *= 60.0f;
  c.H = c.H >= 0.0f ? c.H : c.H + 360.0f;
  c.H = modulo(c.H, 360.0f); ❺
  return c;
}
```

Listing 7-17: A `constexpr` conversion function from RGB to HSV

You've declared and initialized `Color c` ❶, which will eventually get returned by `rgb_to_hsv`. The value of the `Color`, `V`, is computed at ❷ by scaling the maximum value of `r, g`, and `b`. Next, the saturation `S` is calculated by computing the distance between the minimum and maximum RGB

values and scaling by `V` ❸. If you imagine the HSV values as existing inside a cylinder, *saturation* is the distance along the horizontal axis and *value* is the distance along the vertical axis. *Hue* is the angle. For brevity, I won't go into detail about how this angle is computed, but the calculation is implemented between ❹ and ❺. Essentially, it entails computing the angle as an offset from the dominant color component's angle. This is scaled and modulo-ed to fit on the 0- to 360-degree interval and stored into `H`. Finally, `c` is returned.

NOTE *For an explanation of the formula used to convert HSV to RGB, refer to* https://en.wikipedia.org/wiki/HSL_and_HSV#Color_conversion_formulae.

There's quite a bit going on here, but it's all computed at compile time. This means when you initialize colors, the compiler initializes a `Color` with all of the HSV field floats filled in:

```
--snip--
int main() {
  auto black   = rgb_to_hsv(0,     0,   0);
  auto white   = rgb_to_hsv(255, 255, 255);
  auto red     = rgb_to_hsv(255,   0,   0);
  auto green   = rgb_to_hsv(  0, 255,   0);
  auto blue    = rgb_to_hsv(  0,   0, 255);
  // TODO: Print these, output.
}
```

You've told the compiler that each of these color values is compile-time evaluable. Depending on how you use these values within the rest of the program, the compiler can decide whether or not to evaluate them at compile time or runtime. The upshot is that the compiler can usually emit instructions with hardcoded *magic numbers* corresponding to the correct HSV values for each `Color`.

The Case for constexpr

There are some restrictions on what sorts of functions can be `constexpr`, but these restrictions have been relaxed with each new C++ version.

In certain contexts, like embedded development, `constexpr` is indispensable. In general, if an expression can be declared `constexpr`, you should strongly consider doing so. Using `constexpr` rather than manually calculated literals can make your code more expressive. Often, it can also seriously boost performance and safety at runtime.

Volatile Expressions

The `volatile` keyword tells the compiler that every access made through this expression must be treated as a visible side effect. This means access cannot be optimized out or reordered with another visible side effect. This keyword is crucial in some settings, like embedded programming,

where reads and writes to some special portions of memory have effects on the underlying system. The `volatile` keyword keeps the compiler from optimizing such accesses away. Listing 7-18 illustrates why you might need the `volatile` keyword by containing instructions that the compiler would normally optimize away.

```
int foo(int& x) {
  x = 10; ❶
  x = 20; ❷
  auto y = x; ❸
  y = x; ❹
  return y;
}
```

Listing 7-18: A function containing a dead store and a redundant load

Because `x` is assigned ❶ but never used before getting reassigned ❷, it's called a *dead store* and is a straightforward candidate for getting optimized away. There's a similar story where `x` is used to set the value of `y` twice without any intervening instructions ❸❹. This is called a *redundant load* and is also a candidate for optimization.

You might expect any decent compiler to optimize the preceding function into something resembling Listing 7-19.

```
int foo(int& x) {
  x = 20;
  return x;
}
```

Listing 7-19: A plausible optimization of Listing 7-18

In some settings, the redundant reads and dead stores might have visible side effects on the system. By adding the `volatile` keyword to the argument of `foo`, you can avoid the optimizer getting rid of these important accesses, as demonstrated in Listing 7-20.

```
int foo(volatile int& x) {
  x = 10;
  x = 20;
  auto y = x;
  y = x;
  return y;
}
```

Listing 7-20: A `volatile` modification of Listing 7-18

Now the compiler will emit instructions to perform each of the reads and writes you've programmed.

A common misconception is that `volatile` has to do with concurrent programming. It does not. Variables marked `volatile` are not generally thread safe. Part II discusses `std::atomic`, which guarantees certain thread safe primitives on types. Too often, `volatile` is confused with `atomic`!

Summary

This chapter covered the major features of operators, which are the fundamental units of work in a program. You explored several aspects of type conversions and took control of dynamic memory management from the environment. You were also introduced to `constexpr`/`volatile` expressions. With these tools in hand, you can perform almost any system-programming task.

EXERCISES

7-1. Create an `UnsignedBigInteger` class that can handle numbers bigger than a `long`. You can use a byte array as the internal representation (for example, `uint8_t[]` or `char[]`). Implement operator overloads for `operator+` and `operator-`. Perform runtime checks for overflow. For the intrepid, also implement `operator*`, `operator/`, and `operator%`. Make sure that your operator overloads work for both `int` types and `UnsignedBigInteger` types. Implement an `operator int` type conversion. Perform a runtime check if narrowing would occur.

7-2. Create a `LargeBucket` class that can store up to 1MB of data. Extend the `Heap` class so it gives out a `LargeBucket` for allocations greater than 4096 bytes. Make sure that you still throw `std::bad_alloc` whenever the `Heap` is unable to allocate an appropriately sized bucket.

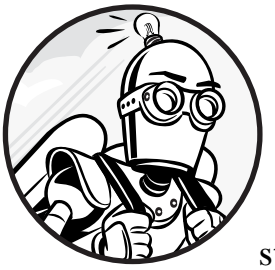
FURTHER READING

- *ISO International Standard ISO/IEC (2017) — Programming Language C++* (International Organization for Standardization; Geneva, Switzerland; *https://isocpp.org/std/the-standard/*)

8

STATEMENTS

Progress doesn't come from early risers—progress is made by lazy men looking for easier ways to do things.
—Robert A. Heinlein, Time Enough for Love



Each C++ function comprises a sequence of *statements*, which are programming constructs that specify the order of execution. This chapter uses an understanding of the object life cycle, templates, and expressions to explore the nuances of statements.

Expression Statements

An *expression statement* is an expression followed by a semicolon (`;`). Expression statements comprise most of the statements in a program. You can turn any expression into a statement, which you should do whenever you need to evaluate an expression but want to discard the result. Of course, this is only useful if evaluating that expression causes a side effect, like printing to the console or modifying the program's state.

Listing 8-1 contains several expression statements.

```
#include <cstdio>

int main() {
  int x{};
  ++x; ❶
  42; ❷
  printf("The %d True Morty\n", x); ❸
}
```

```
The 1 True Morty ❸
```

Listing 8-1: A simple program containing several expression statements

The expression statement at ❶ has a side effect (incrementing `x`), but the one at ❷ doesn't. Both are valid (although the one at ❷ isn't useful). The function call to `printf` ❸ is also an expression statement.

Compound Statements

Compound statements, also called *blocks*, are a sequence of statements enclosed by braces `{ }`. Blocks are useful in control structures like `if` statements, because you might want multiple statements to execute rather than one.

Each block declares a new scope, which is called a *block scope*. As you learned in Chapter 4, objects with automatic storage duration declared within a block scope have lifetimes bound by the block. Variables declared within a block get destroyed in a well-defined order: the reverse of the order in which they were declared.

Listing 8-2 uses the trusty `Tracer` class from Listing 4-5 (on page 97) to explore block scope.

```
#include <cstdio>

struct Tracer {
  Tracer(const char* name) : name{ name } {
    printf("%s constructed.\n", name);
  }
  ~Tracer() {
    printf("%s destructed.\n", name);
  }
private:
  const char* const name;
};

int main() {
  Tracer main{ "main" }; ❶
  {
    printf("Block a\n"); ❷
    Tracer a1{ "a1" }; ❸
    Tracer a2{ "a2" }; ❹
```

```
  }
  {
    printf("Block b\n"); ❺
    Tracer b1{ "b1" }; ❻
    Tracer b2{ "b2" }; ❼
  }
}
-----------------------------------------------------------------------------
main constructed. ❶
Block a ❷
a1 constructed. ❸
a2 constructed.  ❹
a2 destructed.
a1 destructed.
Block b ❺
b1 constructed. ❻
b2 constructed. ❼
b2 destructed.
b1 destructed.
main destructed.
```

Listing 8-2: A program exploring compound statements with the `Tracer` class

Listing 8-2 begins by initializing a `Tracer` called `main` ❶. Next, you generate two compound statements. The first compound statement begins with a left brace { followed by the block's first statement, which prints `Block a` ❷. You create two `Tracers`, `a1` ❸ and `a2` ❹, and then close the block with a right brace }. These two tracers get destructed once execution passes through `Block a`. Notice that these two tracers destruct in reverse order from their initialization: `a2` then `a1`.

Also notice another compound statement following `Block a`, where you print `Block b` ❺ and then construct two tracers, `b1` ❻ and `b2` ❼. Its behavior is identical: `b2` destructs followed by `b1`. Once execution passes through `Block b`, the scope of `main` ends and `Tracer main` finally destructs.

Declaration Statements

Declaration statements (or just *declarations*) introduce identifiers, such as functions, templates, and namespaces, into your programs. This section explores some new features of these familiar declarations, as well as type aliases, attributes, and structured bindings.

NOTE *The expression `static_assert`, which you learned about in Chapter 6, is also a declaration statement.*

Functions

A *function declaration*, also called the function's *signature* or *prototype*, specifies a function's inputs and outputs. The declaration doesn't need to include

parameter names, only their types. For example, the following line declares a function called `randomize` that takes a `uint32_t` reference and returns `void`:

```
void randomize(uint32_t&);
```

Functions that aren't member functions are called *non-member functions*, or sometimes *free functions*, and they're always declared outside of `main()` at namespace scope. A *function definition* includes the function declaration as well as the function's body. A function's declaration defines a function's interface, whereas a function's definition defines its implementation. For example, the following definition is one possible implementation of the `randomize` function:

```
void randomize(uint32_t& x) {
  x = 0x3FFFFFFF & (0x41C64E6D * x + 12345) % 0x80000000;
}
```

NOTE *This `randomize` implementation is a linear congruential generator, a primitive kind of random number generator. See "Further Reading" on page 241 for sources of more information on generating random numbers.*

As you've probably noticed, function declarations are optional. So why do they exist?

The answer is that you can use declared functions throughout your code as long as they're eventually defined somewhere. Your compiler tool chain can figure it out. (You'll learn how this works in Chapter 21.)

The program in Listing 8-3 determines how many iterations the random number generator takes to get from the number 0x4c4347 to the number 0x474343.

```
#include <cstdio>
#include <cstdint>

void randomize(uint32_t&); ❶

int main() {
  size_t iterations{}; ❷
  uint32_t number{ 0x4c4347 }; ❸
  while (number != 0x474343) { ❹
    randomize(number); ❺
    ++iterations; ❻
  }
  printf("%zd", iterations); ❼
}

void randomize(uint32_t& x) {
  x = 0x3FFFFFFF & (0x41C64E6D * x + 12345) % 0x80000000; ❽
}
```

```
927393188 ❼
```

Listing 8-3: A program that uses a function in `main` that isn't defined until later

First, you declare `randomize` ❶. Within `main`, you initialize an `iterations` counter variable to zero ❷ and a `number` variable to 0x4c4347 ❸. A `while` loop checks whether `number` equals the target 0x4c4347 ❹. If it doesn't, you invoke `randomize` ❺ and increment `iterations` ❻. Notice that you haven't yet defined `randomize`. Once `number` equals the target, you print the number of `iterations` ❼ before returning from `main`. Finally, you define `randomize` ❽. The program's output shows that it takes almost a billion iterations to randomly draw the target value.

Try to delete the definition of `randomize` and recompile. You should get an error stating that the definition of `randomize` couldn't be found.

You can similarly separate method declarations from their definitions. As with non-member functions, you can declare a method by omitting its body. For example, the following `RandomNumberGenerator` class replaces the `randomize` function with `next`:

```
struct RandomNumberGenerator {
  explicit RandomNumberGenerator(uint32_t seed) ❶
    : number{ seed } {} ❷
  uint32_t next(); ❸
private:
  uint32_t number;
};
```

You can construct a `RandomNumberGenerator` with a `seed` value ❶, which it uses to initialize the `number` member variable ❷. You've declared the `next` function using the same rules as non-member functions ❸. To provide the definition of `next`, you must use the scope resolution operator and the class name to identify which method you want to define. Otherwise, defining a method is the same as defining a non-member function:

```
uint32_t❶ RandomNumberGenerator::❷next() {
  number = 0x3FFFFFFF & (0x41C64E6D * number + 12345) % 0x80000000; ❸
  return number; ❹
}
```

This definition shares the same return type as the declaration ❶. The `RandomNumberGenerator::` construct specifies that you're defining a method ❷. The function details are essentially the same ❸, except you're returning a copy of the random number generator's state rather than writing into a parameter reference ❹.

Listing 8-4 illustrates how you can refactor Listing 8-3 to incorporate `RandomNumberGenerator`.

```
#include <cstdio>
#include <cstdint>
```

```
struct RandomNumberGenerator {
  explicit RandomNumberGenerator(uint32_t seed)
    : iterations{}❶, number { seed }❷ {}
  uint32_t next(); ❸
  size_t get_iterations() const; ❹
private:
  size_t iterations;
  uint32_t number;
};

int main() {
  RandomNumberGenerator rng{ 0x4c4347 }; ❺
  while (rng.next() != 0x474343) { ❻
    // Do nothing...
  }
  printf("%zd", rng.get_iterations()); ❼
}

uint32_t RandomNumberGenerator::next() { ❽
  ++iterations;
  number = 0x3FFFFFFF & (0x41C64E6D * number + 12345) % 0x80000000;
  return number;
}

size_t RandomNumberGenerator::get_iterations() const { ❾
  return iterations;
}
```

```
927393188 ❼
```

Listing 8-4: A refactor of Listing 8-3 using a `RandomNumberGenerator` class

As in Listing 8-3, you've separated declaration from definition. After declaring a constructor that initializes an `iterations` member to zero ❶ and sets its `number` member to a `seed` ❷, the `next` ❸ and `get_iterations` ❹ method declarations don't contain implementations. Within `main`, you initialize the `RandomNumberGenerator` class with your seed value of 0x4c4347 ❺ and invoke the `next` method to extract new random numbers ❻. The results are the same ❼. As before, the definitions of `next` and `get_iterations` follow their use in `main` ❽❾.

NOTE *The utility of separating definition and declaration might not be apparent because you've been dealing with single-source-file programs so far. Chapter 21 explores multiple-source-file programs where separating declaration and definition provides major benefits.*

Namespaces

Namespaces prevent naming conflicts. In large projects or when importing libraries, namespaces are essential for disambiguating exactly the symbols you're looking for.

Placing Symbols Within Namespaces

By default, all symbols you declare go into the *global namespace*. The global namespace contains all the symbols that you can access without adding any namespace qualifiers. Aside from several classes in the `std` namespace, you've been using objects living exclusively in the global namespace.

To place a symbol within a namespace other than the global namespace, you declare the symbol within a *namespace block*. A namespace block has the following form:

```
namespace BroopKidron13 {
  // All symbols declared within this block
  // belong to the BroopKidron13 namespace
}
```

Namespaces can be nested in one of two ways. First, you can simply nest namespace blocks:

```
namespace BroopKidron13 {
  namespace Shaltanac {
    // All symbols declared within this block
    // belong to the BroopKidron13::Shaltanac namespace
  }
}
```

Second, you can use the scope-resolution operator:

```
namespace BroopKidron13::Shaltanac {
  // All symbols declared within this block
  // belong to the BroopKidron13::Shaltanac namespace
}
```

The latter approach is more succinct.

Using Symbols in Namespaces

To use a symbol within a namespace, you can always use the scope-resolution operator to specify the fully qualified name of a symbol. This allows you to prevent naming conflicts in large projects or when you're using a third-party library. If you and another programmer use the same symbol, you can avoid ambiguity by placing the symbol within a namespace.

Listing 8-5 illustrates how you can use fully qualified symbol names to access a symbol within a namespace.

```
#include <cstdio>

namespace BroopKidron13::Shaltanac { ❶
  enum class Color { ❷
    Mauve,
    Pink,
    Russet
```

```
  };
}

int main() {
  const auto shaltanac_grass{ BroopKidron13::Shaltanac::Color::Russet❸ };
  if(shaltanac_grass == BroopKidron13::Shaltanac::Color::Russet) {
    printf("The other Shaltanac's joopleberry shrub is always "
           "a more mauvey shade of pinky russet.");
  }
}
```

```
The other Shaltanac's joopleberry shrub is always a more mauvey shade of pinky
russet.
```

Listing 8-5: Nested namespace blocks using the scope-resolution operator

Listing 8-5 uses nested namespaces ❶ and declares a `Color` type ❷. To use `Color`, you apply the scope-resolution operator to specify the full name of the symbol, `BroopKidron13::Shaltanac::Color`. Because `Color` is an `enum class`, you use the scope-resolution operator to access its values, as when you assign `shaltanac_grass` to `Russet` ❸.

Using Directives

You can employ a `using` *directive* to avoid a lot of typing. A `using` directive imports a symbol into a block or, if you declare a `using` directive at namespace scope, into the current namespace. Either way, you have to type the full namespace path only once. The usage has the following pattern:

```
using my-type;
```

The corresponding *my-type* gets imported into the current namespace or block, meaning you no longer have to use its full name. Listing 8-6 refactors Listing 8-5 with a using directive.

```
#include <cstdio>

namespace BroopKidron13::Shaltanac {
  enum class Color {
    Mauve,
    Pink,
    Russet
  };
}

int main() {
  using BroopKidron13::Shaltanac::Color; ❶
  const auto shaltanac_grass = Color::Russet❷;
  if(shaltanac_grass == Color::Russet❸) {
    printf("The other Shaltanac's joopleberry shrub is always "
           "a more mauvey shade of pinky russet.");
  }
}
```

```
The other Shaltanac's joopleberry shrub is always a more mauvey shade of pinky
russet.
```

Listing 8-6: A refactor of Listing 8-5 employing a using directive

With a `using` directive ❶ within `main`, you no longer have to type the namespace `BroopKidron13::Shaltanac` to use `Color` ❷❸.

If you're careful, you can introduce all the symbols from a given namespace into the global namespace with the `using namespace` directive.

Listing 8-7 elaborates Listing 8-6: the namespace `BroopKidron13::Shaltanac` contains multiple symbols, which you want to import into the global namespace to avoid a lot of typing.

```
#include <cstdio>

namespace BroopKidron13::Shaltanac {
  enum class Color {
    Mauve,
    Pink,
    Russet
  };

  struct JoopleberryShrub {
    const char* name;
    Color shade;
  };

  bool is_more_mauvey(const JoopleberryShrub& shrub) {
    return shrub.shade == Color::Mauve;
  }
}

using namespace BroopKidron13::Shaltanac; ❶

int main() {
  const JoopleberryShrub❷ yours{
    "The other Shaltanac",
    Color::Mauve❸
  };

  if (is_more_mauvey(yours)❹) {
    printf("%s's joopleberry shrub is always a more mauvey shade of pinky"
           "russet.", yours.name);
  }
}
```

```
The other Shaltanac's joopleberry shrub is always a more mauvey shade of pinky
russet.
```

Listing 8-7: A refactor of Listing 8-6 with multiple symbols imported into the global namespace