You can use a structured binding declaration to unpack a `TextFile` into its parts within your program, as in Listing 8-11.

```
#include <cstdio>

struct TextFile { ❶
  bool success;
  const char* data;
  size_t n_bytes;
};

TextFile read_text_file(const char* path) { ❷
  const static char contents[]{ "Sometimes the goat is you." };
  return TextFile{
    true,
    contents,
    sizeof(contents)
  };
}

int main() {
  const auto [success, contents, length]❸ = read_text_file("REAMDE.txt"); ❹
  if (success❺) {
    printf("Read %zd bytes: %s\n", length❻, contents❼);
  } else {
    printf("Failed to open REAMDE.txt.");
  }
}
```

```
Read 27 bytes: Sometimes the goat is you.
```

Listing 8-11: A program simulating the reading of a text file that returns a POD that you use in a structured binding

You've declared the `TextFile` ❶ and then provided a dummy definition for `read_text_file` ❷. (It doesn't actually read a file; more on that in Part II.)

Within `main`, you invoke `read_text_file` ❹ and use a structured binding declaration to unpack the results into three distinct variables: `success`, `contents`, and `length` ❸. After structured binding, you can use all these variables as though you had declared them individually ❺❻❼.

NOTE *The types within a structured binding declaration don't have to match.*

Attributes

Attributes apply implementation-defined features to an expression statement. You introduce attributes using double brackets `[[ ]]` containing a list of one or more comma-separated attribute elements.

Table 8-1 lists the standard attributes.

Table 8-1: The Standard Attributes

Attribute	Meaning
`[[noreturn]]`	Indicates that a function doesn't return.
`[[deprecated("`*reason*`")]]`	Indicates that this expression is deprecated; that is, its use is discouraged. The "*reason*" is optional and indicates the reason for deprecation.
`[[fallthrough]]`	Indicates that a switch case intends to fall through to the next switch case. This avoids compiler errors that will check for switch case fallthrough, because it's uncommon.
`[[nodiscard]]`	Indicates that the following function or type declaration should be used. If code using this element discards the value, the compiler should emit a warning.
`[[maybe_unused]]`	Indicates that the following element might be unused and that the compiler shouldn't warn about it.
`[[carries_dependency]]`	Used within the `<atomic>` header to help the compiler optimize certain memory operations. You're unlikely to encounter this directly.

Listing 8-12 demonstrates using the `[[noreturn]]` attribute by defining a function that never returns.

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

[[noreturn]] void pitcher() { ❶
  throw std::runtime_error{ "Knuckleball." }; ❷
}

int main() {
  try {
    pitcher(); ❸
  } catch(const std::exception& e) {
    printf("exception: %s\n", e.what()); ❹
  }
}
```

```
Exception: Knuckleball. ❹
```

Listing 8-12: A program illustrating the use of the `[[noreturn]]` attribute

First, you declare the `pitcher` function with the `[[noreturn]]` attribute ❶. Within this function, you throw an exception ❷. Because you always throw an exception, `pitcher` never returns (hence the `[[noreturn]]` attribute). Within `main`, you invoke `pitcher` ❸ and handle the caught exception ❹. Of course, this listing works without the `[[noreturn]]` attribute. But giving this information to the compiler allows it to reason more completely on your code (and potentially to optimize your program).

The situations in which you'll need to use an attribute are rare, but they convey useful information to the compiler nonetheless.

Selection Statements

Selection statements express conditional control flow. The two varieties of selection statements are the `if` statement and the `switch` statement.

if Statements

The `if` statement has the familiar form shown in Listing 8-13.

```
if (condition-1) {
  // Execute only if condition-1 is true ❶
} else if (condition-2) { // optional
  // Execute only if condition-2 is true ❷
}
// ... as many else ifs as desired
--snip--
} else { // optional
  // Execute only if none of the conditionals is true ❸
}
```

Listing 8-13: The syntax of the `if` statement

Upon encountering an `if` statement, you evaluate the *`condition-1`* expression first. If it's `true`, the block at ❶ is executed and the `if` statement stops executing (none of the `else if` or `else` statements are considered). If it's `false`, the `else if` statements' conditions evaluate in order. These are optional, and you can supply as many as you like.

If *`condition-2`* evaluates to `true`, for example, the block at ❷ will execute and none of the remaining `else if` or `else` statements are considered. Finally, the `else` block at ❸ executes if all of the preceding conditions evaluate to `false`. Like the `else if` blocks, the `else` block is optional.

The function template in Listing 8-14 converts an `else` argument into `Positive`, `Negative`, or `Zero`.

```
#include <cstdio>

template<typename T>
constexpr const char* sign(const T& x) {
  const char* result{};
  if (x == 0) { ❶
    result = "zero";
  } else if (x > 0) { ❷
    result = "positive";
  } else { ❸
    result = "negative";
  }
  return result;
}

int main() {
  printf("float 100 is %s\n", sign(100.0f));
  printf("int  -200 is %s\n", sign(-200));
```

```
  printf("char    0 is %s\n", sign(char{}));
}
```

```
float 100 is positive
int  -200 is negative
char    0 is zero
```

Listing 8-14: An example usage of the `if` statement

The `sign` function takes a single argument and determines if it's equal to 0 ❶, greater than 0 ❷, or less than 0 ❸. Depending on which condition matches, it sets the automatic variable `result` equal to one of three strings—`zero`, `positive`, or `negative`—and returns this value to the caller.

Initialization Statements and `if`

You can bind an object's scope to an `if` statement by adding an `init-statement` to `if` and `else if` declarations, as demonstrated in Listing 8-15.

```
if (init-statement; condition-1) {
  // Execute only if condition-1 is true
} else if (init-statement; condition-2) { // optional
  // Execute only if condition-2 is true
}
--snip--
```

Listing 8-15: An `if` statement with initializations

You can use this pattern with structured bindings to produce elegant error handling. Listing 8-16 refactors Listing 8-11 using the initialization statement to scope a `TextFile` to the `if` statement.

```
#include <cstdio>

struct TextFile {
  bool success;
  const char* data;
  size_t n_bytes;
};

TextFile read_text_file(const char* path) {
  --snip--
}

int main() {
  if(const auto [success, txt, len]❶ = read_text_file("REAMDE.txt"); success❷)
  {
    printf("Read %d bytes: %s\n", len, txt); ❸
  } else {
    printf("Failed to open REAMDE.txt."); ❹
  }
}
```

```
Read 27 bytes: Sometimes the goat is you. ❸
```

Listing 8-16: An extension of Listing 8-11 using structured binding and an `if` statement to handle errors

You've moved the structured binding declaration into the initialization statement portion of the `if` statement ❶. This scopes each of the unpacked objects—`success`, `txt`, and `len`—to the `if` block. You use `success` directly within the conditional expression of `if` to determine whether `read_text_file` was successful ❷. If it was, you print the contents of `REAMDE.txt` ❸. If it wasn't, you print an error message ❹.

constexpr if Statements

You can make an `if` statement `constexpr`; such statements are known as `constexpr if` statements. A `constexpr if` statement is evaluated at compile time. Code blocks that correspond to `true` conditions get emitted, and the rest is ignored.

Usage of the `constexpr if` follows usage for a regular `if` statement, as demonstrated in Listing 8-17.

```
if constexpr (condition-1) {
  // Compile only if condition-1 is true
} else if constexpr (condition-2) { // optional; can be multiple else ifs
  // Compile only if condition-2 is true
}
--snip--
} else { // optional
  // Compile only if none of the conditionals is true
}
```

Listing 8-17: Usage of the `constexpr if` statement

In combination with templates and the `<type_traits>` header, `constexpr if` statements are extremely powerful. A major use for `constexpr if` is to provide custom behavior in a function template depending on some attributes of type parameters.

The function template `value_of` in Listing 8-18 accepts pointers, references, and values. Depending on which kind of object the argument is, `value_of` returns either the pointed-to value or the value itself.

```
#include <cstdio>
#include <stdexcept>
#include <type_traits>

template <typename T>
auto value_of(T x❶) {
  if constexpr (std::is_pointer<T>::value) { ❷
    if (!x) throw std::runtime_error{ "Null pointer dereference." }; ❸
    return *x; ❹
```

```
  } else {
    return x; ❺
  }
}

int main() {
  unsigned long level{ 8998 };
  auto level_ptr = &level;
  auto &level_ref = level;
  printf("Power level = %lu\n", value_of(level_ptr)); ❻
  ++*level_ptr;
  printf("Power level = %lu\n", value_of(level_ref)); ❼
  ++level_ref;
  printf("It's over %lu!\n", value_of(level++)); ❽
  try {
    level_ptr = nullptr;
    value_of(level_ptr);
  } catch(const std::exception& e) {
    printf("Exception: %s\n", e.what()); ❾
  }
}
```

```
Power level = 8998 ❻
Power level = 8999 ❼
It's over 9000! ❽
Exception: Null pointer dereference. ❾
```

Listing 8-18: An example function template, `value_of`, employing a `constexpr if` statement

The `value_of` function template accepts a single argument `x` ❶. You determine whether the argument is a pointer type using the `std::is_pointer<T>` type trait as the conditional expression in a `constexpr if` statement ❷. In case `x` is a pointer type, you check for `nullptr` and throw an exception if one is encountered ❸. If `x` isn't a `nullptr`, you dereference it and return the result ❹. Otherwise, `x` is not a pointer type, so you return it (because it is therefore a value) ❺.

Within `main`, you instantiate `value_of` multiple times with an `unsigned long` pointer ❻, an `unsigned long` reference ❼, an `unsigned long` ❽, and a `nullptr` ❾ respectively.

At runtime, the `constexpr if` statement disappears; each instantiation of `value_of` contains one branch of the selection statement or the other. You might be wondering why such a facility is useful. After all, programs are meant to do something useful at runtime, not at compile time. Just flip back to Listing 7-17 (on page 206), and you'll see that compile time evaluation can substantially simplify your programs by eliminating magic values.

There are other examples where compile time evaluation is popular, especially when creating libraries for others to use. Because library writers usually cannot know all the ways their users will utilize their library, they need to write generic code. Often, they'll use techniques like those you learned in Chapter 6 so they can achieve compile-time polymorphism. Constructs like `constexpr` can help when writing this kind of code.

NOTE *If you have a C background, you'll immediately recognize the utility of compile time evaluation when considering that it almost entirely replaces the need for preprocessor macros.*

switch Statements

Chapter 2 first introduced the venerable `switch` statement. This section delves into the addition of the initialization statement into the `switch` declaration. The usage is as follows:

```
switch (init-expression❶; condition) {
  case (case-a): {
    // Handle case-a here
  } break;
  case (case-b): {
    // Handle case-b here
  } break;
    // Handle other conditions as desired
  default: {
    // Handle the default case here
  }
}
```

As with `if` statements, you can instantiate within `switch` statements ❶. Listing 8-19 employs an initialization statement within a `switch` statement.

```
#include <cstdio>

enum class Color { ❶
  Mauve,
  Pink,
  Russet
};

struct Result { ❷
  const char* name;
  Color color;
};

Result observe_shrub(const char* name) { ❸
  return Result{ name, Color::Russet };
}

int main() {
  const char* description;
  switch (const auto result❹ = observe_shrub("Zaphod"); result.color❺) {
  case Color::Mauve: {
    description = "mauvey shade of pinky russet";
    break;
  } case Color::Pink: {
    description = "pinky shade of mauvey russet";
    break;
  } case Color::Russet: {
```

```
    description = "russety shade of pinky mauve";
    break;
  } default: {
    description = "enigmatic shade of whitish black";
  }}
  printf("The other Shaltanac's joopleberry shrub is "
         "always a more %s.", description); ❻
}
```

```
The other Shaltanac's joopleberry shrub is always a more russety shade of
pinky mauve. ❻
```

Listing 8-19: Using an initialization expression in a `switch` statement

You declare the familiar `Color enum class` ❶ and join it with a `char*` member to form the POD type `Result` ❷. The function `observe_shrub` returns a `Result` ❸. Within `main`, you call `observe_shrub` within the initialization expression and store the result in the `result` variable ❹. Within the conditional expression of `switch`, you extract the `color` element of this `result` ❺. This element determines the case that executes (and sets the `description` pointer) ❻.

As with the `if`-statement-plus-initializer syntax, any object initialized in the initialization expression is bound to the scope of the `switch` statement.

Iteration Statements

Iteration statements execute a statement repeatedly. The four kinds of iteration statements are the `while` loop, the `do-while` loop, the `for` loop, and the range-based `for` loop.

while Loops

The `while` loop is the basic iteration mechanism. The usage is as follows:

```
while (condition) {
  // The statement in the body of the loop
  // executes upon each iteration
}
```

Before executing an iteration of the loop, the `while` loop evaluates the `condition` expression. If `true`, the loop continues. If `false`, the loop terminates, as demonstrated in Listing 8-20.

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

bool double_return_overflow(uint8_t& x) { ❶
  const auto original = x;
  x *= 2;
  return original > x;
}
```

```
int main() {
  uint8_t x{ 1 }; ❷
  printf("uint8_t:\n===\n");
  while (!double_return_overflow(x)❸) {
    printf("%u ", x); ❹
  }
}
--------------------------------------------------------------------------------
uint8_t:
===
2 4 8 16 32 64 128 ❹
```

Listing 8-20: A program that doubles a `uint8_t` and prints the new value on each iteration

You declare a `double_return_overflow` function taking an 8-bit, unsigned integer by reference ❶. This function doubles the argument and checks whether this causes an overflow. If it does, it returns `true`. If no overflow occurs, it returns `false`.

You initialize the variable `x` to 1 before entering the `while` loop ❷. The conditional expression in the `while` loop evaluates `double_return_overflow(x)` ❸. This has the side effect of doubling `x`, because you've passed it by reference. It also returns a value telling you whether the doubling caused `x` to overflow. The loop will execute when the conditional expression evaluates to `true`, but `double_return_overflow` is written so it returns `true` when the loop should stop. You fix this problem by prepending the logical negation operator (`!`). (Recall from Chapter 7 that this turns `true` to `false` and `false` to `true`.) So the `while` loop is actually asking, "If it's NOT true that `double_return_overflow` is true . . ."

The end result is that you print the values 2, then 4, then 8, and so on to 128 ❹.

Notice that the value 1 never prints, because evaluating the conditional expression doubles `x`. You can modify this behavior by putting the conditional statement at the end of a loop, which yields a `do-while` loop.

do-while Loops

A `do-while` loop is identical to a `while` loop, except the conditional statement evaluates after a loop completes rather than before. Its usage is as follows:

```
do {
  // The statement in the body of the loop
  // executes upon each iteration
} while (condition);
```

Because the condition evaluates at the end of a loop, you guarantee that the loop will execute at least once.

Listing 8-21 refactors Listing 8-20 into a `do-while` loop.

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

bool double_return_overflow(uint8_t& x) {
  --snip--
```

```
}

int main() {
  uint8_t x{ 1 };
  printf("uint8_t:\n===\n");
  do {
    printf("%u ", x); ❶
  } while (!double_return_overflow(x)❷);
}
```

```
uint8_t:
===
1 2 4 8 16 32 64 128 ❶
```

Listing 8-21: A program that doubles a `uint8_t` and prints the new value on each iteration

Notice that the output from Listing 8-21 now begins with 1 ❶. All you needed to do was reformat the `while` loop to put the condition at the end of the loop ❷.

In most situations involving iterations, you have three tasks:

1. Initialize some object.
2. Update the object before each iteration.
3. Inspect the object's value for some condition.

You can use a `while` or `do-while` loop to accomplish part of these tasks, but the `for` loop provides built-in facilities that make life easier.

for Loops

The `for` loop is an iteration statement containing three special expressions: *initialization*, *conditional*, and *iteration*, as described in the sections that follow.

The Initialization Expression

The initialization expression is like the initialization of `if`: it executes only once before the first iteration executes. Any objects declared within the initialization expression have lifetimes bound by the scope of the `for` loop.

The Conditional Expression

The `for` loop conditional expression evaluates just before each iteration of the loop. If the conditional evaluates to `true`, the loop continues to execute. If the conditional evaluates to `false`, the loop terminates (this behavior is exactly like the conditional of the `while` and `do-while` loops).

Like `if` and `switch` statements, `for` permits you to initialize objects with scope equal to the statement's.

The Iteration Expression

After each iteration of the `for` loop, the iteration expression evaluates. This happens before the conditional expression evaluates. Note that the iteration

expression evaluates after a successful iteration, so the iteration expression won't execute before the first iteration.

To clarify, the following list outlines the typical execution order in a `for` loop:

1. Initialization expression
2. Conditional expression
3. (Loop body)
4. Iteration expression
5. Conditional expression
6. (Loop body)

Steps 4 through 6 repeat until a conditional expression returns `false`.

Usage

Listing 8-22 demonstrates the use of a `for` loop.

```
for(initialization❶; conditional❷; iteration❸) {
  // The statement in the body of the loop
  // executes upon each iteration
}
```

Listing 8-22: Using a `for` loop

The initialization ❶, conditional ❷, and iteration ❸ expressions reside in parentheses preceding the body of the `for` loop.

Iterating with an Index

The `for` loops are excellent at iterating over an array-like object's constituent elements. You use an auxiliary *index* variable to iterate over the range of valid indices for the array-like object. You can use this index to interact with each array element in sequence. Listing 8-23 employs an index variable to print each element of an array along with its index.

```
#include <cstdio>

int main() {
  const int x[]{ 1, 1, 2, 3, 5, 8 }; ❶
  printf("i: x[i]\n"); ❷
  for (int i{}❸; i < 6❹; i++❺) {
    printf("%d: %d\n", i, x[i]);
  }
}
```

```
i: x[i] ❷
0: 1
1: 1
2: 2
3: 3
```

```
4: 5
5: 8
```

Listing 8-23: A program iterating over an array of Fibonacci numbers

You initialize an `int` array called `x` with the first six Fibonacci numbers ❶. After printing a header for the output ❷, you build a `for` loop containing your initialization ❸, conditional ❹, and iteration ❺ expressions. The initialization expression executes first, and it initializes the index variable `i` to zero.

Listing 8-23 shows a coding pattern that hasn't changed since the 1950s. You can eliminate a lot of boilerplate code by using the more modern range-based `for` loop.

Ranged-Based for Loops

The range-based `for` loop iterates over a *range* of values without needing an index variable. A range (or *range expression*) is an object that the range-based `for` loop knows how to iterate over. Many C++ objects are valid range expressions, including arrays. (All of the stdlib containers you'll learn about in Part II are also valid range expressions.)

Usage

Ranged-based `for` loop usage looks like this:

```
for(range-declaration : range-expression) {
  // The statement in the body of the loop
  // executes upon each iteration
}
```

A *range declaration* declares a named variable. This variable must have the same type as implied by the range expression (you can use `auto`).

Listing 8-24 refactors Listing 8-23 to use a range-based `for` loop.

```
#include <cstdio>

int main() {
  const int x[]{ 1, 1, 2, 3, 5, 8 }; ❶
  for (const auto element❷ : x❸) {
    printf("%d ", element❹);
  }
}
```

```
1 1 2 3 5 8
```

Listing 8-24: A range-based for loop iterating over the first six Fibonacci numbers

You still declare an array `x` containing six Fibonacci numbers ❶. The range-based `for` loop contains a range-declaration expression ❷ where you declare the `element` variable to hold each element of the range. It also contains the range expression `x` ❸, which contains the elements you want to iterate over to print ❹.

This code is a whole lot cleaner!

Range Expressions

You can define your own types that are also valid range expressions. But you'll need to specify several functions on your type.

Every range exposes a `begin` and an `end` method. These functions represent the common interface that a range-based `for` loop uses to interact with a range. Both methods return *iterators*. An iterator is an object that supports `operator!=`, `operator++`, and `operator*`.

Let's look at how all these pieces fit together. Under the hood, a range-based `for` loop looks just like the loop in Listing 8-25.

```
const auto e = range.end();❶
for(auto b = range.begin()❷; b != e❸; ++b❹) {
  const auto& element❺ = *b;
}
```

Listing 8-25: A for loop simulating a range-based for loop

The initialization expression stores two variables, `b` ❷ and `e` ❶, which you initialize to `range.begin()` and `range.end()` respectively. The conditional expression checks whether `b` equals `e`, in which case the loop has completed ❸ (this is by convention). The iteration expression increments `b` with the prefix operator ❹. Finally, the iterator supports the dereference operator *, so you can extract the pointed-to element ❺.

NOTE *The types returned by `begin` and `end` don't need to be the same. The requirement is that `operator!=` on `begin` accepts an `end` argument to support the comparison `begin != end`.*

A Fibonacci Range

You can implement a `FibonacciRange`, which will generate an arbitrarily long sequence of Fibonacci numbers. From the previous section, you know that this range must offer a `begin` and an `end` method that returns an iterator. This iterator, which is called `FibonacciIterator` in this example, must in turn offer `operator!=`, `operator++`, and `operator*`.

Listing 8-26 implements a `FibonacciIterator` and a `FibonacciRange`.

```
struct FibonacciIterator {
  bool operator!=(int x) const {
    return x >= current; ❶
  }

  FibonacciIterator& operator++() {
    const auto tmp = current; ❷
    current += last; ❸
    last = tmp; ❹
    return *this; ❺
  }

  int operator*() const {
    return current; ❻
```

```
  }
private:
  int current{ 1 }, last{ 1 };
};

struct FibonacciRange {
  explicit FibonacciRange(int max❼) : max{ max } { }
  FibonacciIterator begin() const { ❽
    return FibonacciIterator{};
  }
  int end() const { ❾
    return max;
  }
private:
  const int max;
};
```

Listing 8-26: An implementation of `FibonacciIterator` and `FibonacciRange`

The `FibonacciIterator` has two fields, `current` and `last`, which are initialized to 1. These keep track of two values in the Fibonacci sequence. Its `operator!=` checks whether the argument is greater than or equal to `current` ❶. Recall that this argument is used within the range-based `for` loop in the conditional expression. It should return `true` if elements remain in the range; otherwise, it returns `false`. The `operator++` appears in the iteration expression and is responsible for setting up the iterator for the next iteration. You first save `current` value into the temporary variable `tmp` ❷. Next, you increment current by `last`, yielding the next Fibonacci number ❸. (This follows from the definition of a Fibonacci sequence.) Then you set `last` equal to `tmp` ❹ and return a reference to `this` ❺. Finally, you implement `operator*`, which returns `current` ❻ directly.

`FibonacciRange` is much simpler. Its constructor takes a max argument that defines an upper limit for the range ❼. The `begin` method returns a fresh `FibonacciIterator` ❽, and the `end` method returns `max` ❾.

It should now be apparent why you need to implement `bool operator!=(int x)` on `FibonacciIterator` rather than, for example, `bool operator!=(const FibonacciIterator& x)`: a `FibonacciRange` returns an `int` from `end()`.

You can use the `FibonacciRange` in a ranged-based `for` loop, as demonstrated in Listing 8-27.

```
#include <cstdio>

struct FibonacciIterator {
  --snip--
};

struct FibonacciRange {
  --snip--;
};

int main() {
```

```
  for (const auto i : FibonacciRange{ 5000 }❶) {
    printf("%d ", i); ❷
  }
}
```

```
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 ❷
```

Listing 8-27: Using `FibonacciRange` in a program

It took a little work to implement `FibonacciIterator` and `FibonacciRange` in Listing 8-26, but the payoff is substantial. Within `main`, you simply construct a `FibonacciRange` with the desired upper limit ❶, and the range-based `for` loop takes care of everything else for you. You simply use the resulting elements within the `for` loop ❷.

Listing 8-27 is functionally equivalent to Listing 8-28, which converts the range-based `for` loop to a traditional `for` loop.

```
#include <cstdio>

struct FibonacciIterator {
  --snip--
};

struct FibonacciRange {
  --snip--;
};

int main() {
  FibonacciRange range{ 5000 };
  const auto end = range.end();❶
  for (const auto x = range.begin()❷; x != end ❸; ++x ❹) {
    const auto i = *x;
    printf("%d ", i);
  }
}
```

```
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Listing 8-28: A refactor of Listing 8-27 using a traditional `for` loop

Listing 8-28 demonstrates how all of the pieces fit together. Calling `range.begin()` ❷ yields a `FibonacciIterator`. When you call `range.end()` ❶, it yields an `int`. These types come straight from the method definitions of `begin()` and `end()` on `FibonacciRange`. The conditional statement ❸ uses `operator!=(int)` on `FibonacciIterator` to get the following behavior: if the iterator `x` has gone past the `int` argument to `operator!=`, the conditional evaluates to `false` and the loop ends. You've also implemented `operator++` on `FibonacciIterator` so `++x` ❹ increments the Fibonacci number within `FibonacciIterator`.

When you compare Listings 8-27 and 8-28, you can see just how much tedium range-based `for` loops hide.

NOTE *You might be thinking, "Sure, the range-based `for` loop looks a lot cleaner, but implementing `FibonacciIterator` and `FibonacciRange` is a lot of work." That's a great point, and for one-time-use code, you probably wouldn't refactor code in this way. Ranges are mainly useful if you're writing library code, writing code that you'll reuse often, or simply consuming ranges that someone else has written.*

Jump Statements

Jump statements, including the `break`, `continue`, and `goto` statements, transfer control flow. Unlike selection statements, jump statements are not conditional. You should avoid using them because they can almost always be replaced with higher-level control structures. They're discussed here because you might see them in older C++ code and they still play a central role in a lot of C code.

break Statements

The `break` statement terminates execution of the enclosing iteration or `switch` statement. Once `break` completes, execution transfers to the statement immediately following the `for`, range-based `for`, `while`, `do-while`, or `switch` statement.

You've already used `break` within `switch` statements; once a case completes, the `break` statement terminates the `switch`. Recall that, without a `break` statement, the `switch` statement would continue executing all of the following cases.

Listing 8-29 refactors Listing 8-27 to break out of a range-based `for` loop if the iterator `i` equals 21.

```
#include <cstdio>

struct FibonacciIterator {
  --snip--
};

struct FibonacciRange {
  --snip--;
};

int main() {
  for (auto i : FibonacciRange{ 5000 }) {
    if (i == 21) { ❶
      printf("*** "); ❷
      break; ❸
    }
    printf("%d ", i);
  }
}
```

```
1 2 3 5 8 13 *** ❷
```

Listing 8-29: A refactor of Listing 8-27 that breaks if the iterator equals 21

An `if` statement is added that checks whether `i` is 21 ❶. If it is, you print three asterisks *** ❷ and `break` ❸. Notice the output: rather than printing 21, the program prints three asterisks and the `for` loop terminates. Compare this to the output of Listing 8-27.

continue Statements

The `continue` statement skips the remainder of an enclosing iteration statement and continues with the next iteration. Listing 8-30 replaces the `break` in Listing 8-29 with a `continue`.

```
#include <cstdio>

struct FibonacciIterator {
  --snip--
};

struct FibonacciRange {
  --snip--;
};

int main() {
  for (auto i : FibonacciRange{ 5000 }) {
    if (i == 21) {
      printf("*** "); ❶
      continue; ❷
    }
    printf("%d ", i);
  }
}
```

```
1 2 3 5 8 13 *** ❶34 55 89 144 233 377 610 987 1597 2584 4181
```

Listing 8-30: A refactor of Listing 8-29 to use `continue` instead of `break`

You still print three asterisks ❶ when `i` is 21, but you use `continue` instead of `break` ❷. This causes 21 not to print, like Listing 8-29; however, unlike Listing 8-29, Listing 8-30 continues iterating. (Compare the output.)

goto Statements

The `goto` statement is an unconditional jump. The target of a `goto` statement is a label.

Labels

Labels are identifiers you can add to any statement. Labels give statements a name, and they have no direct impact on the program. To assign a label, prepend a statement with the desired name of the label followed by a semicolon.

Listing 8-31 adds the labels `luke` and `yoda` to a simple program.

```
#include <cstdio>

int main() {
luke: ❶
  printf("I'm not afraid.\n");
yoda: ❷
  printf("You will be.");
}
```

```
I'm not afraid.
You will be.
```

Listing 8-31: A simple program with labels

The labels ❶❷ do nothing on their own.

goto Usage

The `goto` statement's usage is as follows:

```
goto label;
```

For example, you can employ `goto` statements to needlessly obfuscate the simple program in Listing 8-32.

```
#include <cstdio>

int main() {
  goto silent_bob; ❶
luke:
  printf("I'm not afraid.\n");
  goto yoda; ❸
silent_bob:
  goto luke; ❷
yoda:
  printf("You will be.");
}
```

```
I'm not afraid.
You will be.
```

Listing 8-32: Spaghetti code showcasing the `goto` statement

Control flow in Listing 8-32 passes to `silent_bob` ❶, then to `luke` ❷, and then to `yoda` ❸.

The Role of goto in Modern C++ Programs

In modern C++, there is no good role for `goto` statements. Don't use them.

NOTE *In poorly written C++ (and in most C code), you might see `goto` used as a primitive error-handling mechanism. A lot of system programming entails acquiring resources, checking for error conditions, and cleaning up resources. The RAII paradigm neatly abstracts all of these details, but C doesn't have RAII available. See the Overture to C Programmers on page xxxvii for more information.*

Summary

In this chapter, you worked through different kinds of statements you can employ in your programs. They included declarations and initializations, selection statements, and iteration statements.

NOTE *Keep in mind that `try-catch` blocks are also statements, but they were already discussed in great detail in Chapter 4.*

EXERCISES

8-1. Refactor Listing 8-27 into separate translation units: one for `main` and another for `FibonacciRange` and `FibonacciIterator`. Use a header file to share definitions between the two translation units.

8-2. Implement a `PrimeNumberRange` class that can be used in a range exception to iterate over all prime numbers less than a given value. Again, use a separate header and source file.

8-3. Integrate `PrimeNumberRange` into Listing 8-27, adding another loop that generates all prime numbers less than 5,000.

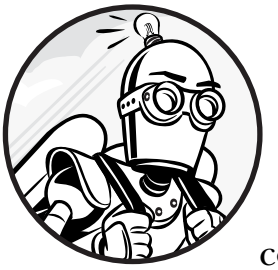
FURTHER READING

- *ISO International Standard ISO/IEC (2017) — Programming Language C++* (International Organization for Standardization; Geneva, Switzerland; *https://isocpp.org/std/the-standard/*)
- *Random Number Generation and Monte Carlo Methods*, 2nd Edition, by James E. Gentle (Springer-Verlag, 2003)
- *Random Number Generation and Quasi-Monte Carlo Methods* by Harald Niederreiter (SIAM Vol. 63, 1992)

9

FUNCTIONS

Functions should do one thing. They should do it well. They should do it only.
—Robert C. Martin, Clean Code



This chapter rounds out the ongoing discussion of functions, which encapsulate code into reusable components. Now that you're armed with a strong background in C++ fundamentals, this chapter first revisits functions with a far more in-depth treatment of modifiers, specifiers, and return types, which appear in function declarations and specialize the behavior of your functions.

Then you'll learn about overload resolution and accepting variable numbers of arguments before exploring function pointers, type aliases, function objects, and the venerable lambda expression. The chapter closes with an introduction to the `std::function` before revisiting the `main` function and accepting command line arguments.

Function Declarations

Function declarations have the following familiar form:

```
prefix-modifiers return-type func-name(arguments) suffix-modifiers;
```

You can provide a number of optional *modifiers* (or *specifiers*) to functions. Modifiers alter a function's behavior in some way. Some modifiers appear at the beginning in the function's declaration or definition (*prefix modifiers*), whereas others appear at the end (*suffix modifiers*). The prefix modifiers appear before the return type. The suffix modifiers appear after the argument list.

There isn't a clear language reason why certain modifiers appear as prefixes or suffixes: because C++ has a long history, these features evolved incrementally.

Prefix Modifiers

At this point, you already know several prefix modifiers:

- The prefix `static` indicates that a function that isn't a member of a class has internal linkage, meaning the function won't be used outside of this translation unit. Unfortunately, this keyword does double duty: if it modifies a method (that is, a function inside a class), it indicates that the function isn't associated with an instantiation of the class but rather with the class itself (see Chapter 4).
- The modifier `virtual` indicates that a method can be overridden by a child class. The `override` modifier indicates to the compiler that a child class intends to override a parent's virtual function (see Chapter 5).
- The modifier `constexpr` indicates that the function should be evaluated at compile time if possible (see Chapter 7).
- The modifier `[[noreturn]]` indicates that this function won't return (see Chapter 8). Recall that this attribute helps the compiler to optimize your code.

Another prefix modifier is `inline`, which plays a role in guiding the compiler when optimizing code.

On most platforms, a function call compiles into a series of instructions, such as the following:

1. Place arguments into registers and on the call stack.
2. Push a return address onto the call stack.
3. Jump to the called function.
4. After the function completes, jump to the return address.
5. Clean up the call stack.

These steps typically execute very quickly, and the payoff in reduced binary size can be substantial if you use a function in many places.

Inlining a function means copying and pasting the contents of the function directly into the execution path, eliminating the need for the five steps outlined. This means that as the processor executes your code, it will immediately execute your function's code rather than executing the (modest) ceremony required for function invocation. If you prefer this marginal increase in speed over the commensurate cost in increased binary size, you can use the `inline` keyword to indicate this to the compiler. The `inline` keyword hints to the compiler's optimizer to put a function directly inline rather than perform a function call.

Adding `inline` to a function doesn't change its behavior; it's purely an expression of preference to the compiler. You must ensure that if you define a function `inline`, you do so in all translation units. Also note that modern compilers will typically inline functions where it makes sense—especially if a function isn't used outside of a single translation unit.

Suffix Modifiers

At this point in the book, you already know two suffix modifiers:

- The modifier `noexcept` indicates that the function will *never* throw an exception. It enables certain optimizations (see Chapter 4).
- The modifier `const` indicates that the method won't modify an instance of its class, allowing `const` references types to invoke the method (see Chapter 4).

This section explores three more suffix modifiers: `final`, `override`, and `volatile`.

final and override

The `final` modifier indicates that a method cannot be overridden by a child class. It's effectively the opposite of `virtual`. Listing 9-1 attempts to override a `final` method and yields a compiler error.

```
#include <cstdio>

struct BostonCorbett {
  virtual void shoot() final❶ {
    printf("What a God we have...God avenged Abraham Lincoln");
  }
};

struct BostonCorbettJunior : BostonCorbett {
  void shoot() override❷ { } // Bang! shoot is final.
};

int main() {
  BostonCorbettJunior junior;
}
```

Listing 9-1: A class attempting to override a final method (This code doesn't compile.)

This listing marks the `shoot` method `final` ❶. Within `BostonCorbettJunior`, which inherits from `BostonCorbett`, you attempt to `override` the `shoot` method ❷. This causes a compiler error.

You can also apply the `final` keyword to an entire class, disallowing that class from becoming a parent entirely, as demonstrated in Listing 9-2.

```
#include <cstdio>

struct BostonCorbett final ❶ {
  void shoot()  {
    printf("What a God we have...God avenged Abraham Lincoln");
  }
};

struct BostonCorbettJunior : BostonCorbett ❷ { }; // Bang!

int main() {
  BostonCorbettJunior junior;
}
```

Listing 9-2: A program with a class attempting to inherit from a final class. (This code doesn't compile.)

The `BostonCorbett` class is marked as `final` ❶, and this causes a compiler error when you attempt to inherit from it in `BostonCorbettJunior` ❷.

NOTE *Neither `final` nor `override` is technically a language keyword; they are* identifiers. *Unlike keywords, identifiers gain special meaning only when used in a specific context. This means you can use `final` and `override` as symbol names elsewhere in your program, thereby leading to the insanity of constructions like `virtual void final() override`. Try not to do this.*

Whenever you're using interface inheritance, you should mark implementing classes `final` because the modifier can encourage the compiler to perform an optimization called *devirtualization*. When virtual calls are devirtualized, the compiler eliminates the runtime overhead associated with a virtual call.

volatile

Recall from Chapter 7 that a volatile object's value can change at any time, so the compiler must treat all accesses to volatile objects as visible side effects for optimization purposes. The `volatile` keyword indicates that a method can be invoked on volatile objects. This is analogous to how `const` methods can be applied to `const` objects. Together, these two keywords define a method's *const/volatile qualification* (or sometimes *cv qualification*), as demonstrated in Listing 9-3.

```
#include <cstdio>

struct Distillate {
```

```
  int apply() volatile ❶ {
    return ++applications;
  }
private:
  int applications{};
};

int main() {
  volatile ❷ Distillate ethanol;
  printf("%d Tequila\n", ethanol.apply()❸);
  printf("%d Tequila\n", ethanol.apply());
  printf("%d Tequila\n", ethanol.apply());
  printf("Floor!");
}
```

```
1 Tequila ❸
2 Tequila
3 Tequila
Floor!
```

Listing 9-3: Illustrating the use of a `volatile` method

In this listing, you declare the `apply` method on the `Distillate` class `volatile` ❶. You also create a `volatile Distillate` called `ethanol` within `main` ❷. Because the `apply` method is `volatile`, you can still invoke it ❸ (even though `ethanol` is `volatile`).

Had you not marked `apply volatile` ❶, the compiler would emit an error when you attempted to invoke it ❸. Just like you cannot invoke a non-`const` method on a `const` object, you cannot invoke a non-`volatile` method on a `volatile` object. Consider what would happen if you could perform such an operation: a non-`volatile` method is a candidate for all kinds of compiler optimizations for the reasons outlined in Chapter 7: many kinds of memory accesses can be optimized away without changing the observable side effects of your program.

How should the compiler treat a contradiction arising from you using a `volatile` object—which requires that all its memory accesses are treated as observable side effects—to invoke a non-`volatile` method? The compiler's answer is that it calls this contradiction an error.

auto Return Types

There are two ways to declare the return value of a function:

- (Primary) Lead a function declaration with its return type, as you've been doing all along.
- (Secondary) Have the compiler deduce the correct return type by using `auto`.

As with `auto` type deduction, the compiler deduces the return type, fixing the runtime type.

This feature should be used judiciously. Because function definitions are documentation, it's best to provide concrete return types when available.

auto and Function Templates

The primary use case for `auto` type deduction is with function templates, where a return type can depend (in potentially complicated ways) on the template parameters. Its usage is as follows:

```
auto my-function(arg1-type arg1, arg2-type arg2, ...) {
  // return any type and the
  // compiler will deduce what auto means
}
```

It's possible to extend the `auto`-return-type deduction syntax to provide the return type as a suffix with the arrow operator `->`. This way, you can append an expression that evaluates to the function's return type. Its usage is as follows:

```
auto my-function(arg1-type arg1, arg2-type arg2, ...) -> type-expression {
  // return an object with type matching
  // the type-expression above
}
```

Usually, you wouldn't use this pedantic form, but in certain situations it's helpful. For example, this form of `auto` type deduction is commonly paired with a `decltype` type expression. A `decltype` type expression yields another expression's resultant type. Its usage is as follows:

```
decltype(expression)
```

This expression resolves to the resulting type of the expression. For example, the following `decltype` expression yields `int`, because the integer literal 100 has that type:

```
decltype(100)
```

Outside of generic programming with templates, `decltype` is rare.

You can combine `auto`-return-type deduction and `decltype` to document the return types of function templates. Consider the `add` function in Listing 9-4, which defines a function template `add` that adds two arguments together.

```
#include <cstdio>

template <typename X, typename Y>
auto add(X x, Y y) -> decltype(x + y) { ❶
  return x + y;
}
```

```
int main() {
  auto my_double = add(100., -10);
  printf("decltype(double + int) = double; %f\n", my_double); ❷

  auto my_uint = add(100U, -20);
  printf("decltype(uint + int) = uint; %u\n", my_uint); ❸

  auto my_ulonglong = add(char{ 100 }, 54'999'900ull);
  printf("decltype(char + ulonglong) = ulonglong; %llu\n", my_ulonglong); ❹
}
--------------------------------------------------------------------------------
decltype(double + int) = double; 90.000000 ❷
decltype(uint + int) = uint; 80 ❸
decltype(char + ulonglong) = ulonglong; 55000000 ❹
```

Listing 9-4: Using `decltype` and auto-return-type deduction

The `add` function employs `auto` type deduction with the `decltype` type expression ❶. Each time you instantiate a template with two types `X` and `Y`, the compiler evaluates `decltype(X + Y)` and fixes the return type of `add`. Within `main`, you provide three instantiations. First, you add a `double` and an `int` ❷. The compiler determines that `decltype(double{ 100. } + int{ -10 })` is a `double`, which fixes the return type of this `add` instantiation. This, in turn, sets the type of `my_double` to `double` ❷. You have two other instantiations: one for an `unsigned int` and `int` (which results in an `unsigned int` ❸) and another for a `char` and an `unsigned long long` (which results in an `unsigned long long` ❹).

Overload Resolution

Overload resolution is the process that the compiler executes when matching a function invocation with its proper implementation.

Recall from Chapter 4 that function overloads allow you to specify functions with the same name but different types and possibly different arguments. The compiler selects among these function overloads by comparing the argument types within the function invocation with the types within each overload declaration. The compiler will choose the best among the possible options, and if it cannot select a best option, it will generate a compiler error.

Roughly, the matching process proceeds as follows:

1. The compiler will look for an exact type match.
2. The compiler will try using integral and floating-point promotions to get a suitable overload (for example, `int` to `long` or `float` to `double`).
3. The compiler will try to match using standard conversions like integral type to floating-point or casting a pointer-to-child into a pointer-to-parent.
4. The compiler will look for a user-defined conversion.
5. The compiler will look for a variadic function.

Variadic Functions

Variadic functions take a variable number of arguments. Typically, you specify the exact number of arguments a function takes by enumerating all of its parameters explicitly. With a variadic function, you can take any number of arguments. The variadic function `printf` is a canonical example: you provide a format specifier and an arbitrary number of parameters. Because `printf` is a variadic function, it accepts any number of parameters.

NOTE *The astute Pythonista will note an immediate conceptual relationship between variadic functions and `*args/**kwargs`.*

You declare variadic functions by placing `...` as the final parameter in the function's argument list. When a variadic function is invoked, the compiler matches arguments against declared arguments. Any leftovers pack into the variadic arguments represented by the `...` argument.

You cannot extract elements from the variadic arguments directly. Instead, you access individual arguments using the utility functions in the `<cstdarg>` header.

Table 9-1 lists these utility functions.

Table 9-1: Utility Functions in the `<cstdarg>` Header

Function	Description
`va_list`	Used to declare a local variable representing the variadic arguments
`va_start`	Enables access to the variadic arguments
`va_end`	Used to end iteration over the variadic arguments
`va_arg`	Used to iterate over each element in the variadic arguments
`va_copy`	Makes a copy of the variadic arguments

The utility functions' usage is a little convoluted and best presented in a cohesive example. Consider the variadic `sum` function in Listing 9-5, which contains a variadic argument.

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

int sum(size_t n, ...❶) {
  va_list args; ❷
  va_start(args, n); ❸
  int result{};
  while (n--) {
    auto next_element = va_arg(args, int); ❹
      result += next_element;
  }
  va_end(args); ❺
```

```
  return result;
}

int main() {
  printf("The answer is %d.", sum(6, 2, 4, 6, 8, 10, 12)); ❻
}
```

```
The answer is 42. ❻
```

Listing 9-5: A `sum` function with a variadic argument list

You declare `sum` as a variadic function ❶. All variadic functions must declare a `va_list`. You've named it `args` ❷. A `va_list` requires initialization with `va_start` ❸, which takes two arguments. The first argument is a `va_list`, and the second is the size of the variadic arguments. You iterate over each element in the variadic arguments using the `va_args` function. The first argument is the `va_list` argument, and the second is the argument type ❹. Once you've completed iterating, you call `va_list` with the `va_list` structure ❺.

You invoke `sum` with seven arguments: the first is the number of variadic arguments (six) followed by six numbers (2, 4, 6, 8, 10, 12) ❻.

Variadic functions are a holdover from C. Generally, variadic functions are unsafe and a common source of security vulnerabilities.

There are at least two major problems with variadic functions:

- Variadic arguments are not type-safe. (Notice that the second argument of `va_args` is a type.)
- The number of elements in the variadic arguments must be tracked separately.

The compiler cannot help you with either of these issues.

Fortunately, variadic templates provide a safer and more performant way to implement variadic functions.

Variadic Templates

The variadic template enables you to create function templates that accept variadic, same-typed arguments. They enable you to employ the considerable power of the template engine. To declare a variadic template, you add a special template parameter called a *template parameter pack*. Listing 9-6 demonstrates its usage.

```
template <typename...❶ Args>
return-type func-name(Args...❷ args) {
  // Use parameter pack semantics
  // within function body
}
```

Listing 9-6: A template function with a parameter pack

The template parameter pack is part of the template parameter list ❶. When you use `Args` within the function template ❷, it's called a *function parameter pack*. Some special operators are available for use with parameter packs:

- You can use `sizeof...(args)` to obtain the parameter pack's size.
- You can invoke a function (for example, `other_function`) with the special syntax `other_function(args...)`. This expands the parameter pack `args` and allows you to perform further processing on the arguments contained in the parameter pack.

Programming with Parameter Packs

Unfortunately, it's not possible to index into a parameter pack directly. You must invoke the function template from within itself—a process called *compile-time recursion*—to recursively iterate over the elements in a parameter pack.

Listing 9-7 demonstrates the pattern.

```
template <typename T, typename...Args>
void my_func(T x❶, Args...args) {
  // Use x, then recurse:
  my_func(args...); ❷
}
```

Listing 9-7: A template function illustrating compile-time recursion with parameter packs. Unlike other usage listings, the ellipses contained in this listing are literal.

The key is to add a regular template parameter before the parameter pack ❶. Each time you invoke `my_func`, `x` absorbs the first argument. The remainder packs into `args`. To invoke, you use the `args...` construct to expand the parameter pack ❷.

The recursion needs a stopping criteria, so you add a function template specialization without the parameter:

```
template <typename T>
void my_func(T x) {
  // Use x, but DON'T recurse
}
```

Revisiting the sum Function

Consider the (much improved) `sum` function implemented as a variadic template in Listing 9-8.

```
#include <cstdio>

template <typename T>
constexpr❶ T sum(T x) { ❷
    return x;
```

```
}

template <typename T, typename... Args>
constexpr❸ T sum(T x, Args... args) { ❹
    return x + sum(args...❺);
}

int main() {
  printf("The answer is %d.", sum(2, 4, 6, 8, 10, 12)); ❻
}
```

```
The answer is 42. ❻
```

Listing 9-8: A refactor of Listing 9-5 using a template parameter pack instead of va_args

The first function ❷ is the overload that handles the stopping condition; if the function has only a single argument, you simply return the argument `x`, because the sum of a single element is just the element. The variadic template ❹ follows the recursion pattern outlined in Listing 9-7. It peels a single argument `x` off the parameter pack `args` and then returns `x` plus the result of the recursive call to `sum` with the expanded parameter pack ❺. Because all of this generic programming can be computed at compile time, you mark these functions `constexpr` ❶❸. This compile-time computation is a *major* advantage over Listing 9-5, which has identical output but computes the result at runtime ❻. (Why pay runtime costs when you don't have to?)

When you just want to apply a single binary operator (like plus or minus) over a range of values (like Listing 9-5), you can use a fold expression instead of recursion.

Fold Expressions

A *fold expression* computes the result of using a binary operator over all the arguments of a parameter pack. Fold expressions are distinct from but related to variadic templates. Their usage is as follows:

```
(... binary-operator parameter-pack)
```

For example, you could employ the following fold expression to sum over all elements in a parameter pack called `pack`:

```
(... + args)
```

Listing 9-9 refactors 9-8 to use a fold expression instead of recursion.

```
#include <cstdio>

template <typename... T>
constexpr auto sum(T... args) {
  return (... + args); ❶
}
```

```
int main() {
  printf("The answer is %d.", sum(2, 4, 6, 8, 10, 12)); ❷
}
```

```
The answer is 42. ❷
```

Listing 9-9: A refactor of Listing 9-8 using a fold expression

You simplify the `sum` function by using a fold expression instead of the recursion approach ❶. The end result is identical ❷.

Function Pointers

Functional programming is a programming paradigm that emphasizes function evaluation and immutable data. One of the major concepts in functional programming is to pass a function as a parameter to another function.

One way you can achieve this is to pass a function pointer. Functions occupy memory, just like objects. You can refer to this memory address via usual pointer mechanisms. However, unlike objects, you cannot modify the pointed-to function. In this respect, functions are conceptually similar to `const` objects. You can take the address of functions and invoke them, and that's about it.

Declaring a Function Pointer

To declare a function pointer, use the following ugly syntax:

```
return-type (*pointer-name)(arg-type1, arg-type2, ...);
```

This has the same appearance as a function declaration where the function name is replaced (`*pointer-name`).

As usual, you can employ the address-of operator `&` to take the address of a function. This is optional, however; you can simply use the function name as a pointer.

Listing 9-10 illustrates how you can obtain and use function pointers.

```
#include <cstdio>

float add(float a, int b) {
  return a + b;
}

float subtract(float a, int b) {
  return a - b;
}

int main() {
  const float first{ 100 };
  const int second{ 20 };

  float(*operation)(float, int) {}; ❶
  printf("operation initialized to 0x%p\n", operation); ❷
```

```
  operation = &add; ❸
  printf("&add = 0x%p\n", operation); ❹
  printf("%g + %d = %g\n", first, second, operation(first, second)); ❺

  operation = subtract; ❻
  printf("&subtract = 0x%p\n", operation); ❼
  printf("%g - %d = %g\n", first, second, operation(first, second)); ❽
}
--------------------------------------------------------------------------------
operation initialized to 0x0000000000000000 ❷
&add = 0x00007FF6CDFE1070 ❹
100 + 20 = 120 ❺
&subtract = 0x00007FF6CDFE10A0 ❼
100 - 20 = 80 ❽
```

Listing 9-10: A program illustrating function pointers. (Due to address space layout randomization, the addresses ❹❼ will vary at runtime.)

This listing shows two functions with identical function signatures, `add` and `subtract`. Because the function signatures match, pointer types to these functions will also match. You initialize a function pointer `operation` accepting a `float` and an `int` as arguments and returning a `float` ❶. Next, you print the value of `operation`, which is `nullptr`, after initialization ❷.

You then assign the address of `add` to `operation` ❸ using the address-of operator and print its new address ❹. You invoke `operation` and print the result ❺.

To illustrate that you can reassign function pointers, you assign `operation` to `subtract` without using the address of operator ❻, print the new value of `operation` ❼, and finally print the result ❽.

Type Aliases and Function Pointers

Type aliases provide a neat way to program with function pointers. The usage is as follows:

```
using alias-name = return-type(*)(arg-type1, arg-type2, ...)
```

You could have defined an `operation_func` type alias in Listing 9-10, for example:

```
using operation_func = float(*)(float, int);
```

This is especially useful if you'll be using function pointers of the same type; it can really clean up the code.

The Function-Call Operator

You can make user-defined types callable or invocable by overloading the function-call operator `operator()()`. Such a type is called a *function type*, and instances of a function type are called *function objects*. The function-call

operator permits any combination of argument types, return types, and modifiers (except `static`).

The primary reason you might want to make a user-defined type callable is to interoperate with code that expects function objects to use the function-call operator. You'll find that many libraries, such as the stdlib, use the function-call operator as the interface for function-like objects. For example, in Chapter 19, you'll learn how to create an asynchronous task with the `std::async` function, which accepts an arbitrary function object that can execute on a separate thread. It uses the function-call operator as the interface. The committee that invented `std::async` could have required you to expose, say, a `run` method, but they chose the function-call operator because it allows generic code to use identical notation to invoke a function or a function object.

Listing 9-11 illustrates the function-call operator's usage.

```
struct type-name {
  return-type❶ operator()❷(arg-type1 arg1, arg-type2 arg2, ...❸) {
    // Body of function-call operator
  }
}
```

Listing 9-11: The function-call operator's usage

The function-call operator has the special `operator()` method name ❷. You declare an arbitrary number of arguments ❸, and you also decide the appropriate return type ❶.

When the compiler evaluates a function-call expression, it will invoke the function-call operator on the first operand, passing the remaining operands as arguments. The result of the function-call expression is the result of invoking the corresponding function-call operator.

A Counting Example

Consider the function type `CountIf` in Listing 9-12, which computes the frequency of a particular `char` in a null-terminated string.

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

struct CountIf {
  CountIf(char x) : x{ x } { }❶
  size_t operator()(const char* str❷) const {
    size_t index{}❸, result{};
    while (str[index]) {
      if (str[index] == x) result++; ❹
      index++;
    }
    return result;
  }
private:
  const char x;
```

```
};

int main() {
  CountIf s_counter{ 's' }; ❺
  auto sally = s_counter("Sally sells seashells by the seashore."); ❻
  printf("Sally: %zd\n", sally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
  auto buffalo = CountIf{ 'f' }("Buffalo buffalo Buffalo buffalo "
                                "buffalo buffalo Buffalo buffalo."); ❼
  printf("Buffalo: %zd\n", buffalo);
}
```

```
Sally: 7
Sailor: 3
Buffalo: 16
```

Listing 9-12: A function type that counts the number of characters appearing in a null-terminated string

You initialize `CountIf` objects using a constructor taking a `char` ❶. You can call the resulting function object as if it were a function taking a null-terminated string argument ❷, because you've implemented the function call operator. The function call operator iterates through each character in the argument `str` using an `index` variable ❸, incrementing the `result` variable whenever the character matches the `x` field ❹. Because calling the function doesn't modify the state of a `CountIf` object, you've marked it `const`.

Within `main`, you've initialized the `CountIf` function object `s_counter`, which will count the frequency of the letter `s` ❺. You can use `s_counter` as if it were a function ❻. You can even initialize a `CountIf` object and use the function operator directly as an rvalue object ❼. You might find this convenient to do in some settings where, for example, you might only need to invoke the object a single time.

You can employ function objects as partial applications. Listing 9-12 is conceptually similar to the `count_if` function in Listing 9-13.

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

size_t count_if(char x❶, const char* str) {
  size_t index{}, result{};
  while (str[index]) {
    if (str[index] == x) result++;
    index++;
  }
  return result;
}

int main() {
  auto sally = count_if('s', "Sally sells seashells by the seashore.");
  printf("Sally: %zd\n", sally);
  auto sailor = count_if('s', "Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
```

```
  auto buffalo = count_if('f', "Buffalo buffalo Buffalo buffalo "
                               "buffalo buffalo Buffalo buffalo.");
  printf("Buffalo: %zd\n", buffalo);
}
```

```
Sally: 7
Sailor: 3
Buffalo: 16
```

Listing 9-13: A free function emulating Listing 9-12

The `count_if` function has an extra argument `x` ❶, but otherwise it's almost identical to the function operator of `CountIf`.

NOTE *In functional programming parlance, the `CountIf` is the partial application of `x` to `count_if`. When you partially apply an argument to a function, you fix that argument's value. The product of such a partial application is another function taking one less argument.*

Declaring function types is verbose. You can often reduce the boilerplate substantially with lambda expressions.

Lambda Expressions

Lambda expressions construct unnamed function objects succinctly. The function object implies the function type, resulting in a quick way to declare a function object on the fly. Lambdas don't provide any additional functionality other than declaring function types the old-fashioned way. But they're extremely convenient when you need to initialize a function object in only a single context.

Usage

There are five components to a lambda expression:

- *`captures`*: The member variables of the function object (that is, the partially applied parameters)
- *`parameters`*: The arguments required to invoke the function object
- *`body`*: The function object's code
- *`specifiers`*: Elements like `constexpr`, `mutable`, `noexcept`, and `[[noreturn]]`
- *`return type`*: The type returned by the function object

Lambda expression usage is as follows:

```
[captures❶] (parameters❷) modifiers❺ -> return-type❹ { body❸ }
```

Only the captures and the body are required; everything else is optional. You'll learn about each of these components in depth in the next few sections.

Each lambda component has a direct analogue in a function object. To form a bridge between the function objects like `CountIf` and lambda expressions, look at Listing 9-14, which lists the `CountIf` function type from Listing 9-12 with annotations that correspond to the analogous portions of the lambda expression in the usage listing.

```
struct CountIf {
  CountIf(char x) : x{ x } { } ❶
  size_t❹ operator()(const char* str❷) const❺ {
    --snip--❸
  }
private:
  const char x; ❷
};
```

Listing 9-14: Comparing the `CountIf` type declaration with a lambda expression

The member variables you set in the constructor of `CountIf` are analogous to a lambda's capture ❶. The function-call operator's arguments ❷, body ❸, and return type ❹ are analogous to the lambda's parameters, body, and return type. Finally, modifiers can apply to the function-call operator ❺ and the lambda. (The numbers in the Lambda expession usage example and Listing 9-14 correspond.)

Lambda Parameters and Bodies

Lambda expressions produce function objects. As function objects, lambdas are callable. Most of the time, you'll want your function object to accept parameters upon invocation.

The lambda's body is just like a function body: all of the parameters have function scope.

You declare lambda parameters and bodies using essentially the same syntax that you use for functions.

For example, the following lambda expression yields a function object that will square its `int` argument:

```
[](int x) { return x*x; }
```

The lambda takes a single `int x` and uses it within the lambda's body to perform the squaring.

Listing 9-15 employs three different lambdas to transform the array `1, 2, 3`.

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

template <typename Fn>
void transform(Fn fn, const int* in, int* out, size_t length) { ❶
  for(size_t i{}; i<length; i++) {
    out[i] = fn(in[i]); ❷
  }
```

```
}

int main() {
  const size_t len{ 3 };
  int base[]{ 1, 2, 3 }, a[len], b[len], c[len];
  transform([](int x) { return 1; }❸, base, a, len);
  transform([](int x) { return x; }❹, base, b, len);
  transform([](int x) { return 10*x+5; }❺, base, c, len);
  for (size_t i{}; i < len; i++) {
    printf("Element %zd: %d %d %d\n", i, a[i], b[i], c[i]);
  }
}
```

```
Element 0: 1 1 15
Element 1: 1 2 25
Element 2: 1 3 35
```

Listing 9-15: Three lambdas and a transform function

The `transform` template function ❶ accepts four arguments: a function object `fn`, an `in` array and an `out` array, and the corresponding `length` of those arrays. Within transform, you invoke `fn` on each element of `in` and assign the result to the corresponding element of `out` ❷.

Within `main`, you declare a `base` array `1, 2, 3` that will be used as the `in` array. In the same line you also declare three uninitialized arrays `a`, `b`, and `c`, which will be used as the `out` arrays. The first call to `transform` passes a lambda (`[](int x) { return 1; }`) that always returns 1 ❸, and the result is stored into `a`. (Notice that the lambda didn't need a name!) The second call to `transform` (`[](int x) { return x; }`) simply returns its argument ❹, and the result is stored into `b`. The third call to `transform` multiplies the argument by 10 and adds 5 ❺. The result is stored in `c`. You then print the output into a matrix where each column illustrates the transform that was applied to the different lambdas in each case.

Notice that you declared `transform` as a template function, allowing you to reuse it with any function object.

Default Arguments

You can provide default arguments to a lambda. Default lambda parameters behave just like default function parameters. The caller can specify values for default parameters, in which case the lambda uses the caller-provided values. If the caller doesn't specify a value, the lambda uses the default.

Listing 9-16 illustrates the default argument behavior.

```
#include <cstdio>

int main() {
  auto increment = [](auto x, int y = 1❶) { return x + y; };
  printf("increment(10)    = %d\n", increment(10)); ❷
  printf("increment(10, 5) = %d\n", increment(10, 5)); ❸
}
```

```
increment(10)    = 11 ❷
increment(10, 5) = 15 ❸
```

Listing 9-16: Using default lambda parameters

The increment lambda has two parameters, `x` and `y`. But the `y` parameter is optional because it has the default argument 1 ❶. If you don't specify an argument for `y` when you call the function ❷, `increment` returns `1 + x`. If you do call the function with an argument for `y` ❸, that value is used instead.

Generic Lambdas

Generic lambdas are lambda expression templates. For one or more parameters, you specify `auto` rather than a concrete type. These `auto` types become template parameters, meaning the compiler will stamp out a custom instantiation of the lambda.

Listing 9-17 illustrates how to assign a generic lambda into a variable and then use the lambda in two different template instantiations.

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

template <typename Fn, typename T❶>
void transform(Fn fn, const T* in, T* out, size_t len) {
  for(size_t i{}; i<len; i++) {
    out[i] = fn(in[i]);
  }
}

int main() {
  constexpr size_t len{ 3 };
  int base_int[]{ 1, 2, 3 }, a[len]; ❷
  float base_float[]{ 10.f, 20.f, 30.f }, b[len]; ❸
  auto translate = [](auto x) { return 10 * x + 5; }; ❹
  transform(translate, base_int, a, l); ❺
  transform(translate, base_float, b, l); ❻

  for (size_t i{}; i < l; i++) {
    printf("Element %zd: %d %f\n", i, a[i], b[i]);
  }
}
```

```
Element 0: 15 105.000000
Element 1: 25 205.000000
Element 2: 35 305.000000
```

Listing 9-17: Using a generic lambda

You add a second template parameter to `transform` ❶, which you use as the pointed-to type of `in` and `out`. This allows you to apply transform to arrays of any type, not just of `int` types. To test out the upgraded transform template, you declare two arrays with different pointed-to types: `int` ❷ and

`float` ❸. (Recall from Chapter 3 that the `f` in `10.f` specifies a `float` literal.) Next, you assign a generic lambda expression to `translate` ❹. This allows you to use the same lambda for each instantiation of transform: when you instantiate with `base_int` ❺ and with `base_float` ❻.

Without a generic lambda, you'd have to declare the parameter types explicitly, like the following:

```
--snip--
  transform([](int x) { return 10 * x + 5; }, base_int, a, l); ❺
  transform([](double x) { return 10 * x + 5; }, base_float, b, l); ❻
```

So far, you've been leaning on the compiler to deduce the return types of your lambdas. This is especially useful for generic lambdas, because often the lambda's return type will depend on its parameter types. But you can explicitly state the return type if you want.

Lambda Return Types

The compiler deduces a lambda's return type for you. To take over from the compiler, you use the arrow `->` syntax, as in the following:

```
[](int x, double y) -> double { return x + y; }
```

This lambda expression accepts an `int` and a `double` and returns a `double`.

You can also use `decltype` expressions, which can be useful with generic lambdas. For example, consider the following lambda:

```
[](auto x, double y) -> decltype(x+y) { return x + y; }
```

Here you've explicitly declared that the return type of the lambda is whatever type results from adding an `x` to a `y`.

You'll rarely need to specify a lambda's return type explicitly.

A far more common requirement is that you must inject an object into a lambda before invocation. This is the role of lambda captures.

Lambda Captures

Lambda captures inject objects into the lambda. The injected objects help to modify the behavior of the lambda.

Declare a lambda's capture by specifying a capture list within brackets `[]`. The capture list goes before the parameter list, and it can contain any number of comma-separated arguments. You then use these arguments within the lambda's body.

A lambda can capture by reference or by value. By default, lambdas capture by value.

A lambda's capture list is analogous to a function type's constructor. Listing 9-18 reformulates `CountIf` from Listing 9-12 as the lambda `s_counter`.

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

int main() {
  char to_count{ 's' }; ❶
  auto s_counter = [to_count❷](const char* str) {
    size_t index{}, result{};
    while (str[index]) {
      if (str[index] == to_count❸) result++;
      index++;
    }
    return result;
  };
  auto sally = s_counter("Sally sells seashells by the seashore."❹);
  printf("Sally: %zd\n", sally);
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
}
```

```
Sally: 7
Sailor: 3
```

Listing 9-18: Reformulating `CountIf` from Listing 9-12 as a lambda

You initialize a `char` called `to_count` to the letter `s` ❶. Next, you capture `to_count` within the lambda expression assigned to `s_counter` ❷. This makes `to_count` available within the body of the lambda expression ❸.

To capture an element by reference rather than by value, prefix the captured object's name with an ampersand `&`. Listing 9-19 adds a capture reference to `s_counter` that keeps a running tally across lambda invocations.

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

int main() {
  char to_count{ 's' };
  size_t tally{};❶
  auto s_counter = [to_count, &tally❷](const char* str) {
    size_t index{}, result{};
    while (str[index]) {
      if (str[index] == to_count) result++;
      index++;
    }
    tally += result;❸
    return result;
  };
  printf("Tally: %zd\n", tally); ❹
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Sally: %zd\n", sally);
  printf("Tally: %zd\n", tally); ❺
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
```

```
  printf("Tally: %zd\n", tally); ❻
}
--------------------------------------------------------------------------------
Tally: 0 ❹
Sally: 7
Tally: 7 ❺
Sailor: 3
Tally: 10 ❻
```

Listing 9-19: Using a capture reference in a lambda

You initialize the counter variable `tally` to zero ❶, and then the `s_counter` lambda captures `tally` by reference (note the ampersand `&`) ❷. Within the lambda's body, you add a statement to increment `tally` by an invocation's `result` before returning ❸. The result is that `tally` will track the total count no matter how many times you invoke the lambda. Before the first `s_counter` invocation, you print the value of `tally` ❹ (which is still zero). After you invoke `s_counter` with `Sally sells seashells by the seashore.`, you have a `tally` of 7 ❺. The last invocation of `s_counter` with `Sailor went to sea to see what he could see.` returns 3, so the value of `tally` is 7 + 3 = 10 ❻.

Default Capture

So far, you've had to capture each element by name. Sometimes this style of capturing is called *named capture*. If you're lazy, you can capture all automatic variables used within a lambda using *default capture*. To specify a default capture by value within a capture list, use a lone equal sign `=`. To specify a default capture by reference, use a lone ampersand `&`.

For example, you could "simplify" the lambda expression in Listing 9-19 to perform a default capture by reference, as demonstrated in Listing 9-20.

```
--snip--
  auto s_counter = [&❶](const char* str) {
    size_t index{}, result{};
    while (str[index]) {
      if (str[index] == to_count❷) result++;
      index++;
    }
    tally❸ += result;
    return result;
  };
--snip--
```

Listing 9-20: Simplifying a lambda expression with a default capture by reference

You specify a default capture by reference ❶, which means any automatic variables in the body of the lambda expression get captured by reference. There are two: `to_count` ❷ and `tally` ❸.

If you compile and run the refactored listing, you'll obtain identical output. However, notice that `to_count` is now captured by reference. If you

accidentally modify it within the lambda expression's body, the change will occur across lambda invocations as well as within `main` (where `to_count` is an automatic variable).

What would happen if you performed a default capture by value instead? You would only need to change the `=` to an `&` in the capture list, as demonstrated in Listing 9-21.

```
--snip--
  auto s_counter = [=❶](const char* str) {
    size_t index{}, result{};
    while (str[index]) {
      if (str[index] == to_count❷) result++;
      index++;
    }
    tally❸ += result;
    return result;
  };
--snip--
```

Listing 9-21: Modifying Listing 9-20 to capture by value instead of by reference (This code doesn't compile.)

You change the default capture to be by value ❶. The `to_count` capture is unaffected ❷, but attempting to modify `tally` results in a compiler error ❸. You're not allowed to modify variables captured by value unless you add the `mutable` keyword to the lambda expression. The `mutable` keyword allows you to modify value-captured variables. This includes calling non-`const` methods on that object.

Listing 9-22 adds the `mutable` modifier and has a default capture by value.

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

int main() {
  char to_count{ 's' };
  size_t tally{};
  auto s_counter = [=❶](const char* str) mutable❷ {
    size_t index{}, result{};
    while (str[index]) {
      if (str[index] == to_count) result++;
      index++;
    }
    tally += result;
    return result;
  };
  auto sally = s_counter("Sally sells seashells by the seashore.");
  printf("Tally: %zd\n", tally); ❸
  printf("Sally: %zd\n", sally);
  printf("Tally: %zd\n", tally); ❹
  auto sailor = s_counter("Sailor went to sea to see what he could see.");
  printf("Sailor: %zd\n", sailor);
```

```
  printf("Tally: %zd\n", tally); ❺
}
```

```
Tally: 0
Sally: 7
Tally: 0
Sailor: 3
Tally: 0
```

Listing 9-22: A `mutable` lambda expression with a default capture by value

You declare a default capture by value ❶, and you make the lambda `s_counter mutable` ❷. Each of the three times you print `tally` ❸❹❺, you get a zero value. Why?

Because `tally` gets copied by value (via the default capture), the version in the lambda is, in essence, an entirely different variable that just happens to have the same name. Modifications to the lambda's copy of `tally` don't affect the automatic `tally` variable of `main`. The `tally` in `main()` is initialized to zero and never gets modified.

It's also possible to mix a default capture with a named capture. You could, for example, default capture by reference and copy `to_count` by value using the following formulation:

```
auto s_counter = [&❶,to_count❷](const char* str) {
  --snip--
};
```

This specifies a default capture by reference ❶ and `to_count` ❷ capture by value.

Although performing a default capture might seem like an easy shortcut, refrain from using it. It's far better to declare captures explicitly. If you catch yourself saying "I'll just use a default capture because there are too many variables to list out," you probably need to refactor your code.

Initializer Expressions in Capture Lists

Sometimes you want to initialize a whole new variable within a capture list. Maybe renaming a captured variable would make a lambda expression's intent clearer. Or perhaps you want to move an object into a lambda and therefore need to initialize a variable.

To use an initializer expression, just declare the new variable's name followed by an equal sign and the value you want to initialize your variable with, as Listing 9-23 demonstrates.

```
auto s_counter = [&tally❶,my_char=to_count❷](const char* str) {
  size_t index{}, result{};
  while (str[index]) {
    if (str[index] == my_char❸) result++;
  --snip--
};
```

Listing 9-23: Using an initializer expression within a lambda capture

The capture list contains a simple named capture where you have `tally` by reference ❶. The lambda also captures `to_count` by value, but you've elected to use the variable name `my_char` instead ❷. Of course, you'll need to use the name `my_char` instead of `to_count` inside the lambda ❸.

NOTE *An initializer expression in a capture list is also called an* init capture.

Capturing this

Sometimes lambda expressions have an enclosing class. You can capture an enclosing object (pointed-to by `this`) by value or by reference using either `[*this]` or `[this]`, respectively.

Listing 9-24 implements a `LambdaFactory` that generates counting lambdas and keeps track of a `tally`.

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

struct LambdaFactory {
  LambdaFactory(char in) : to_count{ in }, tally{} { }
  auto make_lambda() { ❶
    return [this❷](const char* str) {
      size_t index{}, result{};
      while (str[index]) {
        if (str[index] == to_count❸) result++;
        index++;
      }
      tally❹ += result;
      return result;
    };
  }
  const char to_count;
  size_t tally;
};

int main() {
  LambdaFactory factory{ 's' }; ❺
  auto lambda = factory.make_lambda(); ❻
  printf("Tally: %zd\n", factory.tally);
  printf("Sally: %zd\n", lambda("Sally sells seashells by the seashore."));
  printf("Tally: %zd\n", factory.tally);
  printf("Sailor: %zd\n", lambda("Sailor went to sea to see what he could
see."));
  printf("Tally: %zd\n", factory.tally);
}
```

```
Tally: 0
Sally: 7
Tally: 7
Sailor: 3
Tally: 10
```

Listing 9-24: A `LambdaFactory` illustrating the use of `this` capture

The `LambdaFactory` constructor takes a single character and initializes the `to_count` field with it. The `make_lambda` ❶ method illustrates how you can capture `this` by reference ❷ and use the `to_count` ❸ and `tally` ❹ member variables within the lambda expression.

Within `main`, you initialize a `factory` ❺ and make a `lambda` using the `make_lambda` method ❻. The output is identical to Listing 9-19, because you capture `this` by reference and state of `tally` persists across invocations of `lambda`.

Clarifying Examples

There are a lot of possibilities with capture lists, but once you have a command of the basics—capturing by value and by reference—there aren't many surprises. Table 9-2 provides short, clarifying examples that you can use for future reference.

Table 9-2: Clarifying Examples of Lambda Capture Lists

Capture list	Meaning
`[&]`	Default capture by reference
`[&,i]`	Default capture by reference; capture `i` by value
`[=]`	Default capture by value
`[=,&i]`	Default capture by value; capture `i` by reference
`[i]`	Capture `i` by value
`[&i]`	Capture `i` by reference
`[i,&j]`	Capture `i` by value; capture `j` by reference
`[i=j,&k]`	Capture `j` by value as `i`; capture `k` by reference
`[this]`	Capture `enclosing object` by reference
`[*this]`	Capture `enclosing object` by value
`[=,*this,i,&j]`	Default capture by value; capture `this` and `i` by value; capture `j` by reference

constexpr Lambda Expressions

All lambda expressions are `constexpr` as long as the lambda can be invoked at compile time. You can optionally make the `constexpr` declaration explicit, as in the following:

```
[] (int x) constexpr { return x * x; }
```

You should mark a lambda `constexpr` if you want to make sure that it meets all `constexpr` requirements. As of C++17, this means no dynamic memory allocations and no calling non-`constexpr` functions, among other restrictions. The standards committee plans to loosen these restrictions with each release, so if you write a lot of code using `constexpr`, be sure to brush up on the latest `constexpr` constraints.

std::function

Sometimes you just want a uniform container for storing callable objects. The `std::function` class template from the `<functional>` header is a polymorphic wrapper around a callable object. In other words, it's a generic function pointer. You can store a static function, a function object, or a lambda into a `std::function`.

NOTE *The `function` class is in the stdlib. We're presenting it a little ahead of schedule because it fits naturally.*

With `functions`, you can:

- Invoke without the caller knowing the function's implementation
- Assign, move, and copy
- Have an empty state, similar to a `nullptr`

Declaring a Function

To declare a `function`, you must provide a single template parameter containing the function prototype of the callable object:

```
std::function<return-type(arg-type-1, arg-type-2, etc.)>
```

The `std::function` class template has a number of constructors. The default constructor constructs a `std::function` in empty mode, meaning it contains no callable object.

Empty Functions

If you invoke a `std::function` with no contained object, `std::function` will throw a `std::bad_function_call` exception. Consider Listing 9-25.

```
#include <cstdio>
#include <functional>

int main() {
    std::function<void()> func; ❶
    try {
        func(); ❷
    } catch(const std::bad_function_call& e) {
        printf("Exception: %s", e.what()); ❸
    }
}
```

```
Exception: bad function call ❸
```

Listing 9-25: The default `std::function` constructor and the `std::bad_function_call` exception

You default-construct a `std::function` ❶. The template parameter `void()` denotes a function taking no arguments and returning `void`. Because you didn't fill `func` with a callable object, it's in an empty state. When you invoke `func` ❷, it throws a `std::bad_function_call`, which you catch and print ❸.

Assigning a Callable Object to a Function

To assign a callable object to a `function`, you can either use the constructor or assignment operator of `function`, as in Listing 9-26.

```
#include <cstdio>
#include <functional>

void static_func() { ❶
  printf("A static function.\n");
}

int main() {
  std::function<void()> func { [] { printf("A lambda.\n"); } }; ❷
  func(); ❸
  func = static_func; ❹
  func(); ❺
}
```

```
A lambda. ❸
A static function. ❺
```

Listing 9-26: Using the constructor and assignment operator of `function`

You declare the static function `static_func` that takes no arguments and returns `void` ❶. In `main`, you create a function called `func` ❷. The template parameter indicates that a callable object contained by `func` takes no arguments and returns `void`. You initialize `func` with a lambda that prints the message `A lambda`. You invoke `func` immediately afterward ❸, invoking the contained lambda and printing the expected message. Next, you assign `static_func` to `func`, which replaces the lambda you assigned upon construction ❹. You then invoke `func`, which invokes `static_func` rather than the lambda, so you see `A static function.` printed ❺.

An Extended Example

You can construct a `function` with callable objects, as long as that object supports the function semantics implied by the template parameter of `function`.

Listing 9-27 uses an array of `std::function` instances and fills it with a static function that counts spaces, a `CountIf` function object from Listing 9-12, and a lambda that computes string length.

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

struct CountIf {
```

```
  --snip--
};

size_t count_spaces(const char* str) {
  size_t index{}, result{};
  while (str[index]) {
    if (str[index] == ' ') result++;
    index++;
  }
  return result;
}

std::function❶<size_t(const char*)❷> funcs[]{
  count_spaces, ❸
  CountIf{ 'e' }, ❹
  [](const char* str) { ❺
    size_t index{};
    while (str[index]) index++;
    return index;
  }
};

auto text = "Sailor went to sea to see what he could see.";

int main() {
  size_t index{};
  for(const auto& func : funcs❻) {
    printf("func #%zd: %zd\n", index++, func(text)❼);
  }
}
--------------------------------------------------------------------------------
func #0: 9 ❸
func #1: 7 ❹
func #2: 44 ❺
```

Listing 9-27: Using a `std::function` array to iterate over a uniform collection of callable objects with varying underlying types

You declare a `std::function` array ❶ with static storage duration called `funcs`. The template argument is the function prototype for a function taking a `const char*` and returning a `size_t` ❷. In the `funcs` array, you pass in a static function pointer ❸, a function object ❹, and a lambda ❺. In `main`, you use a range-based `for` loop to iterate through each function in `funcs` ❻. You invoke each function `func` with the text `Sailor went to sea to see what he could see.` and print the result.

Notice that, from the perspective of `main`, all the elements in `funcs` are the same: you just invoke them with a null-terminated string and get back a `size_t` ❼.

NOTE *Using a `function` can incur runtime overhead. For technical reasons, `function` might need to make a dynamic allocation to store the callable object. The compiler also has difficulty optimizing away `function` invocations, so you'll often incur an indirect function call. Indirect function calls require additional pointer dereferences.*

The main Function and the Command Line

All C++ programs must contain a global function with the name `main`. This function is defined as the program's entry point, the function invoked at program startup. Programs can accept any number of environment-provided arguments called *command line parameters* upon startup.

Users pass command line parameters to programs to customize their behavior. You've probably used this feature when executing command line programs, as in the `copy` (on Linux: `cp`) command:

```
$ copy file_a.txt file_b.txt
```

When invoking this command, you instruct the program to copy `file_a.txt` into `file_b.txt` by passing these values as command line parameters. As with command line programs you might be used to, it's possible to pass values as command line parameters to your C++ programs.

You can choose whether your program handles command line parameters by how you declare `main`.

The Three main Overloads

You can access command line parameters within `main` by adding arguments to your `main` declaration.

There are three valid varieties of overload for `main`, as shown in Listing 9-28.

```
int main(); ❶
int main(int argc, char* argv[]); ❷
int main(int argc, char* argv[], impl-parameters); ❸
```

Listing 9-28: The valid overloads for `main`

The first overload ❶ takes no parameters, which is the way you've been using `main()` in this book so far. Use this form if you want to ignore any arguments provided to your program.

The second overload ❷ accepts two parameters, `argc` and `argv`. The first argument, `argc`, is a non-negative number corresponding to the number of elements in `argv`. The environment calculates this automatically: you don't have to provide the number of elements in `argc`. The second argument, `argv`, is an array of pointers to null-terminated strings that corresponds to an argument passed in from the execution environment.

The third overload ❸ is an extension of the second overload ❷: it accepts an arbitrary number of additional implementation parameters. This way, the target platform can offer some additional arguments to the program. Implementation parameters aren't common in modern desktop environments.

Usually, an operating system passes the full path to the program's executable as the first command line argument. This behavior depends on your operating environment. On macOS, Linux, and Windows, the

executable's path is the first argument. The format of this path depends on the operating system. (Chapter 17 discusses filesystems in depth.)

Exploring Program Parameters

Let's build a program to explore how the operating system passes parameters to your program. Listing 9-29 prints the number of command line arguments and then prints the index and value of the arguments on each line.

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

int main(int argc, char** argv) { ❶
  printf("Arguments: %d\n", argc); ❷
  for(size_t i{}; i<argc; i++) {
    printf("%zd: %s\n", i, argv[i]); ❸
  }
}
```

Listing 9-29: A program that prints the command line arguments. Compile this program as `list_929`.

You declare main with the `argc`/`argv` overload, which makes command line parameters available to your program ❶. First, you print the number of command line arguments via `argc` ❷. Then you loop through each argument, printing its index and its value ❸.

Let's look at some sample output (on Windows 10 x64). Here is one program invocation:

```
$ list_929 ❶
Arguments: 1 ❷
0: list_929.exe ❸
```

Here, you provide no additional command line arguments aside from the name of the program, `list_929` ❶. (Depending on how you compiled the listing, you should replace this with the name of your executable.) On a Windows 10 x64 machine, the result is that your program receives a single argument ❷, the name of the executable ❸.

And here is another invocation:

```
$ list_929 Violence is the last refuge of the incompetent. ❶
Arguments: 9
0: list_929.exe
1: Violence
2: is
3: the
4: last
5: refuge
6: of
7: the
8: incompetent.
```

Here, you provide additional program arguments: `Violence is the last refuge of the incompetent.` ❶. You can see from the output that Windows has split the command line by spaces, resulting in a total of nine arguments.

In major desktop operating systems, you can force the operating system to treat such a phrase as a single argument by enclosing it within quotes, as in the following:

```
$ list_929 "Violence is the last refuge of the incompetent."
Arguments: 2
0: list_929.exe
1: Violence is the last refuge of the incompetent.
```

A More Involved Example

Now that you know how to process command line input, let's consider a more involved example. A *histogram* is an illustration that shows a distribution's relative frequency. Let's build a program that computes a histogram of the letter distribution of the command line arguments.

Start with two helper functions that determine whether a given `char` is an uppercase letter or a lowercase letter:

```
constexpr char pos_A{ 65 }, pos_Z{ 90 }, pos_a{ 97 }, pos_z{ 122 };
constexpr bool within_AZ(char x) { return pos_A <= x && pos_Z >= x; } ❶
constexpr bool within_az(char x) { return pos_a <= x && pos_z >= x; } ❷
```

The `pos_A`, `pos_Z`, `pos_a`, and `pos_z` constants contain the ASCII values of the letters A, Z, a, and z respectively (refer to the ASCII chart in Table 2-4). The `within_AZ` function determines whether some `char x` is an uppercase letter by determining whether its value is between `pos_A` and `pos_Z` inclusive ❶. The `within_az` function does the same for lowercase letters ❷.

Now that you have some elements for processing ASCII data from the command line, let's build an `AlphaHistogram` class that can ingest command line elements and store character frequencies, as shown in Listing 9-30.

```
struct AlphaHistogram {
  void ingest(const char* x); ❶
  void print() const; ❷
private:
  size_t counts[26]{}; ❸
};
```

Listing 9-30: An `AlphaHistogram` that ingests command line elements

An `AlphaHistogram` will store the frequency of each letter in the `counts` array ❸. This array initializes to zero whenever an `AlphaHistogram` is constructed. The `ingest` method will take a null-terminated string and update `counts` appropriately ❶. Then the `print` method will display the histogram information stored in `counts` ❷.

First, consider the implementation of `ingest` in Listing 9-31.

```
void AlphaHistogram::ingest(const char* x) {
  size_t index{}; ❶
  while(const auto c = x[index]) { ❷
    if (within_AZ(c)) counts[c - pos_A]++; ❸
    else if (within_az(c)) counts[c - pos_a]++; ❹
    index++; ❺
  }
}
```

Listing 9-31: An implementation of the `ingest` method

Because `x` is a null-terminated string, you don't know its length ahead of time. So, you initialize an `index` variable ❶ and use a `while` loop to extract a single `char` `c` at a time ❷. This loop will terminate if `c` is null, which is the end of the string. Within the loop, you use the `within_AZ` helper function to determine whether `c` is an uppercase letter ❸. If it is, you subtract `pos_A` from `c`. This normalizes an uppercase letter to the interval 0 to 25 to correspond with `counts`. You do the same check for lowercase letters using the `within_az` helper function ❹, and you update `counts` in case `c` is lowercase. If `c` is neither lowercase nor uppercase, `counts` is unaffected. Finally, you increment `index` before continuing to loop ❺.

Now, consider how to `print` `counts`, as shown in Listing 9-32.

```
void AlphaHistogram::print() const {
  for(auto index{ pos_A }; index <= pos_Z; index++) {  ❶
    printf("%c: ", index); ❷
    auto n_asterisks = counts[index - pos_A]; ❸
    while (n_asterisks--) printf("*"); ❹
    printf("\n"); ❺
  }
}
```

Listing 9-32: An implementation of the `print` method

To print the histogram, you loop over each letter from A to Z ❶. Within the loop, you first print the `index` letter ❷, and then determine how many asterisks to print by extracting the correct letter out of `counts` ❸. You print the correct number of asterisks using a `while` loop ❹, and then you print a terminating newline ❺.

Listing 9-33 shows `AlphaHistogram` in action.

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

constexpr char pos_A{ 65 }, pos_Z{ 90 }, pos_a{ 97 }, pos_z{ 122 };
constexpr bool within_AZ(char x) { return pos_A <= x && pos_Z >= x; }
constexpr bool within_az(char x) { return pos_a <= x && pos_z >= x; }

struct AlphaHistogram {
  --snip--
```

```
};

int main(int argc, char** argv) {
  AlphaHistogram hist;
  for(size_t i{ 1 }; i<argc; i++) { ❶
    hist.ingest(argv[i]); ❷
  }
  hist.print(); ❸
}
```

```
$ list_933 The quick brown fox jumps over the lazy dog
A: *
B: *
C: *
D: *
E: ***
F: *
G: *
H: **
I: *
J: *
K: *
L: *
M: *
N: *
O: ****
P: *
Q: *
R: **
S: *
T: **
U: **
V: *
W: *
X: *
Y: *
Z: *
```

Listing 9-33: A program illustrating `AlphaHistogram`

You iterate over each command line argument after the program name ❶, passing each into the `ingest` method of your `AlphaHistogram` object ❷. Once you've ingested them all, you print the `histogram` ❸. Each line corresponds to a letter, and the asterisks show the absolute frequency of the corresponding letter. As you can see, the phrase `The quick brown fox jumps over the lazy dog` contains each letter in the English alphabet.

Exit Status

The `main` function can return an `int` corresponding to the exit status of the program. What the values represent is environment defined. On modern desktop systems, for example, a zero return value corresponds with a successful program execution. If no `return` statement is explicitly given, an implicit `return 0` is added by the compiler.

Summary

This chapter took a deeper look at functions, including how to declare and define them, how to use the myriad keywords available to you to modify function behavior, how to specify return types, how overload resolution works, and how to take a variable number of arguments. After a discussion of how you take pointers to functions, you explored lambda expressions and their relationship to function objects. Then you learned about the entry point for your programs, the `main` function, and how to take command line arguments.

EXERCISES

9-1. Implement a `fold` function template with the following prototype:

```
template <typename Fn, typename In, typename Out>
constexpr Out fold(Fn function, In* input, size_t length, Out initial);
```

For example, your implementation must support the following usage:

```
int main() {
  int data[]{ 100, 200, 300, 400, 500 };
  size_t data_len = 5;
  auto sum = fold([](auto x, auto y) { return x + y; }, data, data_len,
0);
  print("Sum: %d\n", sum);
}
```

The value of `sum` should be 1,500. Use `fold` to calculate the following quantities: the `maximum`, the `minimum`, and the number of elements greater than 200.

9-2. Implement a program that accepts an arbitrary number of command line arguments, counts the length in characters of each argument, and prints a histogram of the argument length distribution.

9-3. Implement an `all` function with the following prototype:

```
template <typename Fn, typename In, typename Out>
constexpr bool all(Fn function, In* input, size_t length);
```

The `Fn` function type is a `predicate` that supports `bool operator()(In)`. Your `all` function must test whether `function` returns `true` for every element of `input`. If it does, return `true`. Otherwise, return `false`.

For example, your implementation must support the following usage:

```
int main() {
  int data[]{ 100, 200, 300, 400, 500 };
  size_t data_len = 5;
  auto all_gt100 = all([](auto x) { return x > 100; }, data, data_len);
  if(all_gt100) printf("All elements greater than 100.\n");
}
```

FURTHER READING

- *Functional Programming in C++: How to Improve Your C++ Programs Using Functional Techniques* by Ivan Čukić (Manning, 2019)
- *Clean Code: A Handbook of Agile Software Craftsmanship* by Robert C. Martin (Pearson Education, 2009)

PART II

C++ LIBRARIES AND FRAMEWORKS

NEO: Why do my eyes hurt?
MORPHEUS: You've never used them before.
—The Matrix

Part II exposes you to the world of C++ libraries and frameworks, including the C++ Standard Library (stdlib) and the Boost Libraries (Boost). The latter is an open source volunteer project to produce much-needed C++ libraries.

In Chapter 10, you'll tour several testing and mocking frameworks. In a major departure from Part I, most listings in Part II are unit tests. These provide you with practice in testing code, and unit tests are often more succinct and expressive than `printf`-based example programs.

Chapter 11 takes a broad look at smart pointers, which manage dynamic objects and facilitate the most powerful resource management model in any programming language.

Chapter 12 explores the many utilities that implement common programming tasks.

Chapter 13 delves into the massive suite of containers that can hold and manipulate objects.

Chapter 14 explains iterators, the common interface that all containers provide.

Chapter 15 reviews strings and string operations, which store and manipulate human-language data.

Chapter 16 discusses streams, a modern way to perform input and output operations.

Chapter 17 illuminates the filesystem library, which provides facilities for interacting with filesystems.

Chapter 18 surveys the dizzying array of algorithms that query and manipulate iterators.

Chapter 19 outlines the major approaches to concurrency, which allows your programs to run simultaneous threads of execution.

Chapter 20 reviews Boost ASIO, a cross-platform library for network and low-level input/output programming using an asynchronous approach.

Chapter 21 provides several application frameworks that implement standard structures required in everyday application programming.

Part II will function well as a quick reference, but your first reading should be sequential.