Both nuclear_isotopes and waste_heat are allocated each time power_up_rat_thing is invoked. Just before power_up_rat_thing returns, these variables are deallocated.

Because you cannot access these variables outside of power_up_rat_thing, automatic variables are also called *local variables*.

Static Storage Duration

A *static object* is declared using the static or extern keyword. You declare static variables at the same level you declare functions—at global scope (or *namespace scope*). Static objects with global scope have *static storage duration* and are allocated when the program starts and deallocated when the program stops.

The program in Listing 4-2 powers up a Rat Thing with nuclear isotopes by calling the power_up_rat_thing function. When it does, the Rat Thing's power increases, and the variable rat_things_power keeps track of the power level between power-ups.

```
#include <cstdio>

static int rat_things_power = 200; ❶

void power_up_rat_thing(int nuclear_isotopes) {
  rat_things_power = rat_things_power + nuclear_isotopes; ❷
  const auto waste_heat = rat_things_power * 20; ❸
  if (waste_heat > 10000) { ❹
    printf("Warning! Hot doggie!\n"); ❺
  }
}

int main() {
  printf("Rat-thing power: %d\n", rat_things_power); ❻
  power_up_rat_thing(100); ❼
  printf("Rat-thing power: %d\n", rat_things_power);
  power_up_rat_thing(500);
  printf("Rat-thing power: %d\n", rat_things_power);
}
--------------------------------------------------------------------------------
Rat-thing power: 200
Rat-thing power: 300
Warning! Hot doggie! ❽
Rat-thing power: 800
```

Listing 4-2: A program with a static variable and several automatic variables

The variable rat_things_power ❶ is a static variable because it's declared at global scope with the static keyword. Another feature of being declared at global scope is that power_up_rat_thing can be accessed from any function in the translation unit. (Recall from Chapter 1 that a translation unit is what a preprocessor produces after acting on a single source file.) At ❷, you see power_up_rat_thing increasing rat_things_power by the number of nuclear_isotopes. Because rat_things_power is a static variable—and hence its

lifetime is the program's lifetime—each time you call `power_up_rat_thing`, the value of `rat_things_power` carries over into the next call.

Next, you calculate how much waste heat is produced given the new value of `rat_things_power`, and you store the result in the automatic variable `waste_heat` ❸. Its storage duration begins when `power_up_rat_thing` is called and ends when `power_up_rat_thing` returns, so its values aren't saved between function calls. Finally, you check whether `waste_heat` is over a threshold value of `10000` ❹. If it is, you print a warning message ❺.

Within `main`, you alternate between printing the value of `rat_things_power` ❻ and calling `power_up_rat_thing` ❼.

Once you've increased the Rat Thing's power from `300` to `800`, you get the warning message in the output ❽. The effects of modifying `rat_things_power` last for the lifetime of the program due to its static storage duration.

When you use the `static` keyword, you specify *internal linkage*. Internal linkage means that a variable is inaccessible to other translation units. You can alternately specify *external linkage*, which makes a variable accessible to other translation units. For external linkage, you use the `extern` keyword instead of `static`.

You could modify Listing 4-2 in the following way to achieve external linkage:

```
#include <cstdio>

extern int rat_things_power = 200; // External linkage
--snip--
```

With `extern` rather than `static`, you can access `rat_things_power` from other translation units.

Local Static Variables

A *local static variable* is a special kind of static variable that is a local—rather than global—variable. Local static variables are declared at function scope, just like automatic variables. But their lifetimes begin upon the first invocation of the enclosing function and end when the program exits.

For example, you could refactor Listing 4-2 to make `rat_things_power` a local static variable, as demonstrated in Listing 4-3.

```
#include <cstdio>

void power_up_rat_thing(int nuclear_isotopes) {
  static int rat_things_power = 200;
  rat_things_power = rat_things_power + nuclear_isotopes;
  const auto waste_heat = rat_things_power * 20;
  if (waste_heat > 10000) {
    printf("Warning! Hot doggie!\n");
  }
  printf("Rat-thing power: %d\n", rat_things_power);
}
```

```
int main() {
  power_up_rat_thing(100);
  power_up_rat_thing(500);
}
```

Listing 4-3: A refactor of Listing 4-2 using a local static variable.

Unlike in Listing 4-2, you cannot refer to `rat_things_power` from outside of the `power_up_rat_thing` function due to its local scope. This is an example of a programming pattern called *encapsulation*, which is the bundling of data with a function that operates on that data. It helps to protect against unintended modification.

Static Members

Static members are members of a class that aren't associated with a particular instance of the class. Normal class members have lifetimes nested within the class's lifetime, but static members have static storage duration.

These members are essentially similar to static variables and functions declared at global scope; however, you must refer to them by the containing class's name, using the scope resolution operator `::`. In fact, you must initialize static members at global scope. You cannot initialize a static member within a containing class definition.

NOTE *There is an exception to the static member initialization rule: you can declare and define integral types within a class definition as long as they're also* `const`.

Like other static variables, static members have only a single instance. All instances of a class with static members share the same member, so if you modify a static member, *all* class instances will observe the modification. To illustrate, you could convert `power_up_rat_thing` and `rat_things_power` in Listing 4-2 to static members of a `RatThing` class, as shown in Listing 4-4.

```
#include <cstdio>

struct RatThing {
  static int rat_things_power; ❶
  static❷ void power_up_rat_thing(int nuclear_isotopes) {
    rat_things_power❸ = rat_things_power + nuclear_isotopes;
    const auto waste_heat = rat_things_power * 20;
    if (waste_heat > 10000) {
      printf("Warning! Hot doggie!\n");
    }
    printf("Rat-thing power: %d\n", rat_things_power);
  }
};

int RatThing::rat_things_power = 200; ❹

int main() {
```

```
  RatThing::power_up_rat_thing(100); ❺
  RatThing::power_up_rat_thing(500);
}
```

Listing 4-4: A refactor of Listing 4-2 using static members

The `RatThing` class contains `rat_things_power` as a static member variable ❶ and `power_up_rat_thing` as a static method ❷. Because `rat_things_power` is a member of `RatThing`, you don't need the scope resolution operator ❸; you access it like any other member.

You see the scope resolution operator in action where `rat_things_power` is initialized ❹ and where you invoke the static method `power_up_rat_thing` ❺.

Thread-Local Storage Duration

One of the fundamental concepts in concurrent programs is the *thread*. Each program has one or more threads that can perform independent operations. The sequence of instructions that a thread executes is called its *thread of execution*.

Programmers must take extra precautions when using more than one thread of execution. Code that multiple threads can execute safely is called *thread-safe code*. Mutable global variables are the source of many thread safety issues. Sometimes, you can avoid these issues by giving each thread its own copy of a variable. You can do this by specifying that an object has *thread storage duration*.

You can modify any variable with static storage duration to have thread-local storage duration by adding the `thread_local` keyword to the `static` or `extern` keyword. If only `thread_local` is specified, `static` is assumed. The variable's linkage is unchanged.

Listing 4-3 is not thread safe. Depending on the order of reads and writes, `rat_things_power` could become corrupted. You could make Listing 4-3 thread safe by specifying `rat_things_power` as `thread_local`, as demonstrated here:

```
#include <cstdio>

void power_up_rat_thing(int nuclear_isotopes) {
  static thread_local int rat_things_power = 200; ❶
  --snip--
}
```

Now each thread would represent its own Rat Thing; if one thread modifies its `rat_things_power`, the modification will not affect the other threads. Each copy of `rat_things_power` is initialized to 200 ❶.

NOTE *Concurrent programming is discussed in more detail in Chapter 19. Thread storage duration is presented here for completeness.*

Dynamic Storage Duration

Objects with *dynamic storage duration* are allocated and deallocated on request. You have manual control over when a *dynamic object*'s life begins and when it ends. Dynamic objects are also called *allocated objects* for this reason.

The primary way to allocate a dynamic object is with a *new expression*. A new expression begins with the `new` keyword followed by the desired type of the dynamic object. New expressions create objects of a given type and then return a pointer to the newly minted object.

Consider the following example where you create an `int` with dynamic storage duration and save it into a pointer called `my_int_ptr`:

```
int*❶ my_int_ptr = new❷ int❸;
```

You declare a pointer to `int` and initialize it with the result of the new expression on the right side of the equal sign ❶. The new expression is composed of the `new` keyword ❷ followed by the desired type `int` ❸. When the new expression executes, the C++ runtime allocates memory to store an `int` and then returns its pointer.

You can also initialize a dynamic object within a new expression, as shown here:

```
int* my_int_ptr = new int{ 42 }; // Initializes dynamic object to 42
```

After allocating storage for the `int`, the dynamic object will be initialized as usual. After initialization completes, the dynamic object's lifetime begins.

You deallocate dynamic objects using the *delete expression*, which is composed of the `delete` keyword followed by a pointer to the dynamic object. Delete expressions always return `void`.

To deallocate the object pointed to by `my_int_ptr`, you would use the following delete expression:

```
delete my_int_ptr;
```

The value contained in memory where the deleted object resided is undefined, meaning the compiler can produce code that leaves anything there. In practice, major compilers will try to be as efficient as possible, so typically the object's memory will remain untouched until the program reuses it for some other purposes. You would have to implement a custom destructor to, for example, zero out some sensitive contents.

NOTE *Because the compiler doesn't typically clean up memory after an object is deleted, a subtle and potentially serious type of bug called a* use after free *can occur. If you delete an object and accidentally reuse it, your program might appear to function correctly because the deallocated memory might still contain reasonable values. In some situations, the problems don't manifest until the program has been in production for a long time—or until a security researcher finds a way to exploit the bug and discloses it!*

Dynamic Arrays

Dynamic arrays are arrays with dynamic storage duration. You create dynamic arrays with *array new expressions*. Array new expressions have the following form:

```
new MyType[n_elements] { init-list }
```

`MyType` is the desired type of the array elements, `n_elements` is the length of the desired array, and the optional `init-list` is an initialization list to initialize the array. Array new expressions return a pointer to the first element of the newly allocated array.

In the following example, you allocate an `int` array of length 100 and save the result into a pointer called `my_int_array_ptr`:

```
int* my_int_array_ptr = new int[100❶];
```

The number of elements ❶ doesn't need to be constant: the size of the array can be determined at runtime, meaning the value between brackets ❶ could be a variable rather than a literal.

To deallocate a dynamic array, use the *array delete expression*. Unlike the array new expression, the array delete expression doesn't require a length:

```
delete[] my_int_ptr;
```

Like the delete expression, the array delete expression returns `void`.

Memory Leaks

With privilege comes responsibility, so you must make sure that dynamic objects you allocate are also deallocated. Failure to do so causes *memory leaks* in which memory that is no longer needed by your program isn't released. When you leak memory, you use up a resource in your environment that you'll never reclaim. This can cause performance problems or worse.

NOTE *In practice, your program's operating environment might clean up leaked resources for you. For example, if you've written user-mode code, modern operating systems will clean up the resources when the program exits. However, if you've written kernel code, those operating systems won't clean up the resources. You'll only reclaim them when the computer reboots.*

Tracing the Object Life Cycle

The object life cycle is as daunting to newcomers as it is powerful. Let's clarify with an example that explores each of the storage durations.

Consider the `Tracer` class in Listing 4-5, which prints a message whenever a `Tracer` object is constructed or destructed. You can use this class to

investigate object life cycles, because each Tracer clearly indicates when its life is beginning and ending.

```
#include <cstdio>

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

Listing 4-5: A `Tracer` class that announces construction and destruction

The constructor takes a single parameter ❶ and saves it into the member `name` ❷. It then prints a message containing `name` ❸. The destructor ❹ also prints a message with `name`.

Consider the program in Listing 4-6. Four different `Tracer` objects have different storage durations. By looking at the order of the program's `Tracer` output, you can verify what you've learned about storage durations.

```
#include <cstdio>

struct Tracer {
  --snip--
};

static Tracer t1{ "Static variable" }; ❶
thread_local Tracer t2{ "Thread-local variable" }; ❷

int main() {
  printf("A\n"); ❸
  Tracer t3{ "Automatic variable" }; ❹
  printf("B\n");
  const auto* t4 = new Tracer{ "Dynamic variable" }; ❺
  printf("C\n");
}
```

Listing 4-6: A program using the `Tracer` class in Listing 4-5 to illustrate storage duration

Listing 4-6 contains a `Tracer` with static duration ❶, thread local duration ❷, automatic duration ❹, and dynamic duration ❺. Between each line in `main`, you print the character `A`, `B`, or `C` for reference ❸.

Running the program yields Listing 4-7.

```
Static variable constructed.
Thread-local variable constructed.
A ❸
```

```
Automatic variable constructed.
B
Dynamic variable constructed.
C
Automatic variable destructed.
Thread-local variable destructed.
Static variable destructed.
```

Listing 4-7: Sample output from running Listing 4-6

Before the first line of `main` ❸, the static and thread local variables `t1` and `t2` have been initialized ❶ ❷. You can see this in Listing 4-7: both variables have printed their initialization messages before `A`. As an automatic variable, the scope of `t3` is bounded by the enclosing function `main`. Accordingly, `t3` is constructed where it is initialized just after `A`.

After `B`, you see the message corresponding to the initialization of `t4` ❺. Notice that there's no corresponding message generated by the dynamic destructor of `Tracer`. The reason is that you've (intentionally) leaked the object pointed to by `t4`. Because there's no command to `delete t4`, the destructor is never called.

Just before `main` returns, `C` prints. Because `t3` is an automatic variable whose scope is `main`, it's destroyed at this point because `main` is returning.

Finally, the static and thread-local variables `t1` and `t2` are destroyed just before program exit, yielding the final two messages in Listing 4-7.

Exceptions

Exceptions are types that communicate an error condition. When an error condition occurs, you *throw* an exception. After you throw an exception, it's *in flight*. When an exception is in flight, the program stops normal execution and searches for an *exception handler* that can manage the in-flight exception. Objects that fall out of scope during this process are destroyed.

In situations where there's no good way to handle an error locally, such as in a constructor, you generally use exceptions. Exceptions play a crucial role in managing object life cycles in such circumstances.

The other option for communicating error conditions is to return an error code as part of a function's prototype. These two approaches are complementary. In situations where an error occurs that can be dealt with locally or that is expected to occur during the normal course of a program's execution, you generally return an error code.

The throw Keyword

To throw an exception, use the `throw` keyword followed by a throwable object.

Most objects are throwable. But it's good practice to use one of the exceptions available in stdlib, such as `std::runtime_error` in the `<stdexcept>` header. The `runtime_error` constructor accepts a null-terminated `const char*` describing the nature of the error condition. You can retrieve this message via the `what` method, which takes no parameters.

The Groucho class in Listing 4-8 throws an exception whenever you invoke the forget method with an argument equal to 0xFACE.

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

struct Groucho {
  void forget(int x) {
    if (x == 0xFACE) {
      throw❶ std::runtime_error❷{ "I'd be glad to make an exception." };
    }
    printf("Forgot 0x%x\n", x);
  }
};
```

Listing 4-8: The Groucho class

To throw an exception, Listing 4-8 uses the throw keyword ❶ followed by a std::runtime_error object ❷.

Using try-catch Blocks

You use try-catch blocks to establish exception handlers for a block of code. Within the try block, you place code that might throw an exception. Within the catch block, you specify a handler for each exception type you can handle.

Listing 4-9 illustrates the use of a try-catch block to handle exceptions thrown by a Groucho object.

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

struct Groucho {
  --snip--
};

int main() {
  Groucho groucho;
  try { ❶
    groucho.forget(0xC0DE); ❷
    groucho.forget(0xFACE); ❸
    groucho.forget(0xC0FFEE); ❹
  } catch (const std::runtime_error& e❺) {
    printf("exception caught with message: %s\n", e.what()); ❻
  }
}
```

Listing 4-9: The use of try-catch to handle the exceptions of the Groucho class

In main, you construct a Groucho object and then establish a try-catch block ❶. Within the try portion, you invoke the groucho class's forget method with several different parameters: 0xC0DE ❷, 0xFACE ❸, and 0xC0FFEE ❹. Within the catch portion, you handle any std::runtime_error exceptions ❺ by printing the message to the console ❻.

When you run the program in Listing 4-9, you get the following output:

```
Forgot 0xc0de
exception caught with message: I'd be glad to make an exception.
```

When you invoked `forget` with the argument `0xC0DE` ❷, `groucho` printed `Forgot 0xc0de` and returned. When you invoked `forget` with the argument `0xFACE` ❸, `groucho` threw an exception. This exception stopped normal program execution, so `forget` is never invoked again ❹. Instead, the in-flight exception is caught ❺, and its message is printed ❻.

A CRASH COURSE IN INHERITANCE

Before introducing the stdlib exceptions, you need to understand simple C++ class inheritance at a very high level. Classes can have *subclasses* that inherit the functionality of their *superclasses*. The syntax in Listing 4-10 defines this relationship.

```
struct Superclass {
  int x;
};

struct Subclass : Superclass { ❶
  int y;
  int foo() {
    return x + y; ❷
  }
};
```

Listing 4-10: Defining superclasses and subclasses

There's nothing special about `Superclass`. But the declaration of `Subclass` ❶ is special. It defines the inheritance relationship using the `: Superclass` syntax. `Subclass` inherits members from `Superclass` that are not marked `private`. You can see this in action where `Subclass` uses the field `x` ❷. This is a field belonging to `Superclass`, but because `Subclass` inherits from `Superclass`, `x` is accessible.

Exceptions use these inheritance relationships to determine whether a handler catches an exception. Handlers will catch a given type *and* any of its parents' types.

stdlib Exception Classes

You can arrange classes into parent-child relationships using *inheritance*. Inheritance has a big impact on how the code handles exceptions. There is a nice, simple hierarchy of existing exception types available for use in the stdlib. You should try to use these types for simple programs. Why reinvent the wheel?

Standard Exception Classes

The stdlib provides you with the *standard exception classes* in the `<stdexcept>` header. These should be your first port of call when you're programming exceptions. The superclass for all the standard exception classes is the class `std::exception`. All the subclasses in `std::exception` can be partitioned into three groups: logic errors, runtime errors, and language support errors. While language support errors are not generally relevant to you as a programmer, you'll definitely encounter logic errors and runtime errors. Figure 4-1 summarizes their relationship.

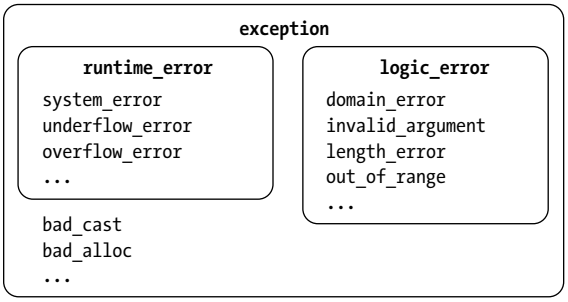

Figure 4-1: How stdlib exceptions are nested under `std::exception`

Logic Errors

Logic errors derive from the `logic_error` class. Generally, you could avoid these exceptions through more careful programming. A primary example is when a logical precondition of a class isn't satisfied, such as when a class invariant cannot be established. (Remember from Chapter 2 that a class invariant is a feature of a class that is always true.)

Since a class invariant is something that the programmer defines, neither the compiler nor the runtime environment can enforce it without help. You can use a class constructor to check for various conditions, and if you cannot establish a class invariant, you can throw an exception. If the failure is the result of, say, passing an incorrect parameter to the constructor, a `logic_error` is an appropriate exception to throw.

The `logic_error` has several subclasses that you should be aware of:

- The `domain_error` reports errors related to valid input range, especially for math functions. The square root, for example, only supports non-negative numbers (in the real case). If a negative argument is passed, a square root function could throw a `domain_error`.
- The `invalid_argument` exception reports generally unexpected arguments.
- The `length_error` exception reports that some action would violate a maximum size constraint.
- The `out_of_range` exception reports that some value isn't in an expected range. The canonical example is bounds-checked indexing into a data structure.

Runtime Errors

Runtime errors derive from the `runtime_error` class. These exceptions help you report error conditions that are outside the program's scope. Like `logic_error`, `runtime_error` has some subclasses that you might find useful:

- The `system_error` reports that the operating system encountered some error. You can get a lot of mileage out of this kind of exception. Inside of the `<system_error>` header, there's a large number of *error codes* and *error conditions*. When a `system_error` is constructed, information about the error is packed in so you can determine the nature of the error. The `.code()` method returns an `enum class` of type `std::errc` that has a large number of values, such as `bad_file_descriptor`, `timed_out`, and `permission_denied`.
- The `overflow_error` and `underflow_error` report arithmetic overflow and underflow, respectively.

Other errors inherit directly from `exception`. A common one is the `bad_alloc` exception, which reports that `new` failed to allocate the required memory for dynamic storage.

Language Support Errors

You won't use language support errors directly. They exist to indicate that some core language feature failed at runtime.

Handling Exceptions

The rules for exception handling are based on class inheritance. When an exception is thrown, a `catch` block handles the exception if the thrown exception's type matches the `catch` handler's exception type or if the thrown exception's type *inherits from* the `catch` handler's exception type.

For example, the following handler catches any exception that inherits from `std::exception`, including a `std::logic_error`:

```
try {
  throw std::logic_error{ "It's not about who wrong "
                          "it's not about who right" };
} catch (std::exception& ex) {
  // Handles std::logic_error as it inherits from std::exception
}
```

The following special handler catches *any* exception regardless of its type:

```
try {
  throw 'z'; // Don't do this.
} catch (...) {
  // Handles any exception, even a 'z'
}
```

Special handlers are typically used as a safety mechanism to log the program's catastrophic failure to catch an exception of a specific type.

You can handle different types of exceptions originating from the same `try` block by chaining together `catch` statements, as demonstrated here:

```
try {
  // Code that might throw an exception
  --snip--
} catch (const std::logic_error& ex) {
  // Log exception and terminate the program; there is a programming error!
  --snip--
} catch (const std::runtime_error& ex) {
  // Do our best to recover gracefully
  --snip--
} catch (const std::exception& ex) {
  // This will handle any exception that derives from std:exception
  // that is not a logic_error or a runtime_error.
  --snip--
} catch (...) {
  // Panic; an unforeseen exception type was thrown
  --snip--
}
```

It's common to see such code in a program's entry point.

RETHROWING AN EXCEPTION

In a `catch` block, you can use the `throw` keyword to resume searching for an appropriate exception handler. This is called *rethrowing an exception*. There are some unusual but important cases where you might want to further inspect an exception before deciding to handle it, as shown in Listing 4-11.

```
try {
  // Some code that might throw a system_error
  --snip--
} catch(const std::system_error& ex) {
  if(ex.code()!= std::errc::permission_denied){
    // Not a permission denied error
    throw; ❶
  }
  // Recover from a permission denied
  --snip--
}
```

Listing 4-11: Rethrowing an error

(continued)

In this example, some code that might throw a `system_error` is wrapped in a `try-catch` block. All `system_errors` are handled, but unless it's an `EACCES` (permission denied) error, you *rethrow* the exception ❶. There are some performance penalties to this approach, and the resulting code is often needlessly convoluted.

Rather than rethrowing, you can define a new exception type and create a separate `catch` handler for the `EACCES` error, as demonstrated in Listing 4-12.

```
try {
  // Throw a PermissionDenied instead
  --snip--
} catch(const PermissionDenied& ex) {
  // Recover from an EACCES error (Permission Denied) ❶
  --snip--
}
```

Listing 4-12: Catching a specific exception rather than rethrowing

If a `std::system_error` is thrown, the `PermissionDenied` handler ❶ won't catch it. (Of course, you could still keep the `std::system_error` handler to catch such exceptions if you wish.)

User-Defined Exceptions

You can define your own exceptions whenever you'd like; usually, these *user-defined exceptions* inherit from `std::exception`. All the classes from stdlib use exceptions that derive from `std::exception`. This makes it easy to catch all exceptions, whether from your code or from the stdlib, with a single `catch` block.

The noexcept Keyword

The keyword `noexcept` is another exception-related term you should know. You can, and should, mark any function that cannot possibly throw an exception `noexcept`, as in the following:

```
bool is_odd(int x) noexcept {
  return 1 == (x % 2);
}
```

Functions marked `noexcept` make a rigid contract. When you're using a function marked `noexcept`, you can rest assured that the function cannot throw an exception. In exchange, you must be extremely careful when you mark your own function `noexcept`, since the compiler won't check for you. If your code throws an exception inside a function marked `noexcept`,

it's bad juju. The C++ runtime will call the function `std::terminate`, a function that by default will exit the program via `abort`. Your program cannot recover:

```
void hari_kari() noexcept {
  throw std::runtime_error{ "Goodbye, cruel world." };
}
```

Marking a function `noexcept` enables some code optimizations that rely on the function's not being able to throw an exception. Essentially, the compiler is liberated to use move semantics, which may be faster (more about this in "Move Semantics" on page 122).

NOTE *Check out Item 16 of* Effective Modern C++ *by Scott Meyers for a thorough discussion of* `noexcept`. *The gist is that some move constructors and move assignment operators might throw an exception, for example, if they need to allocate memory and the system is out. Unless a move constructor or move assignment operator specifies otherwise, the compiler must assume that a move could cause an exception. This disables certain optimizations.*

Call Stacks and Exceptions

The *call stack* is a runtime structure that stores information about active functions. When a piece of code (the *caller*) invokes a function (the *callee*), the machine keeps track of who called whom by pushing information onto the call stack. This allows programs to have many function calls nested within each other. The callee could then, in turn, become the caller by invoking another function.

Stacks

A stack is a flexible data container that can hold a dynamic number of elements. There are two essential operations that all stacks support: *pushing* elements onto the top of the stack and *popping* those elements off. It is a last-in, first-out data structure, as illustrated in Figure 4-2.

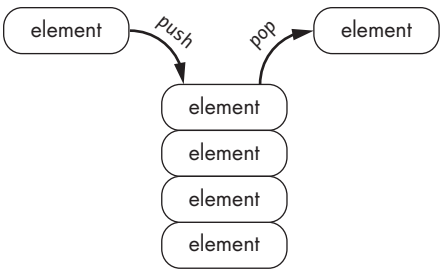

Figure 4-2: Elements being pushed onto and popped off of a stack

As its name suggests, the call stack is functionally similar to its namesake data container. Each time a function is invoked, information about the function invocation is arranged into a *stack frame* and pushed onto the call stack. Because a new stack frame is pushed onto the stack for every function call, a callee is free to call other functions, forming arbitrarily deep call chains. Whenever a function returns, its stack frame is popped off the top of the call stack, and execution control resumes as indicated by the previous stack frame.

Call Stacks and Exception Handling

The runtime seeks the closest exception handler to a thrown exception. If there is a matching exception handler in the current stack frame, it will handle the exception. If no matching handler is found, the runtime will unwind the call stack until it finds a suitable handler. Any objects whose lifetimes end are destroyed in the usual way.

Throwing in Destructors

If you throw an exception in a destructor, you are juggling with chainsaws. Such an exception absolutely must be caught within the destructor.

Suppose an exception is thrown, and during stack unwinding, another exception is thrown by a destructor during normal cleanup. Now you have *two* exceptions in flight. How should the C++ runtime handle such a situation?

You can have an opinion on the matter, but the runtime will call `terminate`. Consider Listing 4-13, which illustrates what can happen when you throw an exception from a destructor:

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

struct CyberdyneSeries800 {
  CyberdyneSeries800() {
    printf("I'm a friend of Sarah Connor."); ❶
  }
  ~CyberdyneSeries800() {
    throw std::runtime_error{ "I'll be back." }; ❷
  }
};

int main() {
  try {
    CyberdyneSeries800 t800; ❸
    throw std::runtime_error{ "Come with me if you want to live." }; ❹
  } catch(const std::exception& e) { ❺
    printf("Caught exception: %s\n", e.what()); ❻
  }
}
```

```
I'm a friend of Sarah Connor. ❶
```

Listing 4-13: A program illustrating the perils of throwing an exception within a destructor

NOTE *Listing 4-13 calls `std::terminate`, so depending on your environment, you might get a nasty pop-up indicating this.*

First, you declare the `CyberdyneSeries800` class, which has a simple constructor that prints a message ❶ and a thoroughly belligerent destructor that throws an uncaught exception ❷. Within `main`, you set up a `try` block where you initialize a `CyberdyneSeries800` called `t800` ❸ and throw a `runtime_error` ❹. Under better circumstances, the `catch` block ❺ would handle this exception, print its message ❻, and exit gracefully. Because `t800` is an automatic variable within the `try` block, it destructs during the normal process of finding a handler for the exception you've thrown ❹. And because `t800` throws an exception in its destructor ❷, your program invokes `std::terminate` and ends abruptly.

As a general rule, treat destructors as if they were `noexcept`.

A SimpleString Class

Using an extended example, let's explore how constructors, destructors, members, and exceptions gel together. The `SimpleString` class in Listing 4-14 allows you to add C-style strings together and print the result.

```
#include <stdexcept>

struct SimpleString {
  SimpleString(size_t max_size) ❶
    : max_size{ max_size }, ❷
      length{} { ❸
    if (max_size == 0) {
      throw std::runtime_error{ "Max size must be at least 1." }; ❹
    }
    buffer = new char[max_size]; ❺
    buffer[0] = 0; ❻
  }

  ~SimpleString() {
    delete[] buffer; ❼
  }
--snip--
private:
  size_t max_size;
  char* buffer;
  size_t length;
};
```

Listing 4-14: The constructor and destructor of a `SimpleString` class

The constructor ❶ takes a single `max_size` argument. This is the maximum length of your string, which includes a null terminator. The member initializer ❷ saves this length into the `max_size` member variable. This value is also used in the array new expression to allocate a buffer to store your string ❺. The resulting pointer is stored into `buffer`. You initialize length

to zero ❸ and ensure that there is at least enough size for a null byte ❹. Because the string is initially empty, you assign the first byte of the buffer to zero ❻.

NOTE *Because* `max_size` *is a* `size_t`*, it is unsigned and cannot be negative, so you don't need to check for this bogus condition.*

The `SimpleString` class owns a resource—the memory pointed to by `buffer`—which must be released when it's no longer needed. The destructor contains a single line ❼ that deallocates `buffer`. Because you've paired the allocation and deallocation of `buffer` with the constructor and destructor of `SimpleString`, you'll never leak the storage.

This pattern is called *resource acquisition is initialization (RAII)* or *constructor acquires, destructor releases (CADRe)*.

NOTE *The* `SimpleString` *class still has an implicitly defined* copy constructor. *Although it might never leak the storage, it will potentially double free if copied. You'll learn about copy constructors in "Copy Semantics" on page 115. Just be aware that Listing 4-14 is a teaching tool, not production-worthy code.*

Appending and Printing

The `SimpleString` class isn't of much use yet. Listing 4-15 adds the ability to print the string and append a line to the end of the string.

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

struct SimpleString {
  --snip--
  void print(const char* tag) const { ❶
    printf("%s: %s", tag, buffer);
  }

  bool append_line(const char* x) { ❷
    const auto x_len = strlen❸(x);
    if (x_len + length + 2 > max_size) return false; ❹
    std::strncpy❺(buffer + length, x, max_size - length);
    length += x_len;
    buffer[length++] = '\n';
    buffer[length] = 0;
    return true;
  }
--snip--
};
```

Listing 4-15: The `print` *and* `append_line` *methods of* `SimpleString`

The first method `print` ❶ prints your string. For convenience, you can provide a `tag` string so you can match an invocation of `print` with the result. This method is `const` because it doesn't need to modify the state of a `SimpleString`.

The `append_line` method ❷ takes a null-terminated string `x` and adds its contents—plus a newline character—to `buffer`. It returns `true` if `x` was successfully appended and `false` if there wasn't enough space. First, `append_line` must determine the length of `x`. For this, you employ the `strlen` function ❸ from the `<cstring>` header, which accepts a null-terminated string and returns its length:

```
size_t strlen(const char* str);
```

You use `strlen` to compute the length of `x` and initialize `x_len` with the result. This result is used to compute whether appending `x` (a newline character) and a null byte to the current string would result in a string with length greater than `max_size` ❹. If it would, `append_line` returns `false`.

If there is enough room to append `x`, you need to copy its bytes into the correct location in `buffer`. The `std::strncpy` function ❺ from the `<cstring>` header is one possible tool for this job. It accepts three arguments: the `destination` address, the `source` address, and the `num` of characters to copy:

```
char* std::strncpy(char* destination, const char* source, std::size_t num);
```

The `strncpy` function will copy up to `num` bytes from `source` into `destination`. Once complete, it will return `destination` (which you discard).

After adding the number of bytes `x_len` copied into `buffer` to `length`, you finish by adding a newline character `\n` and a null byte to the end of `buffer`. You return `true` to indicate that you've successfully appended the input `x` as a line to the end of `buffer`.

WARNING *Use `strncpy` very carefully. It's too easy to forget the null-terminator in the `source` string or not allocate enough space in the `destination` string. Both errors will cause undefined behavior. We'll cover a safer alternative in Part II of the book.*

Using SimpleString

Listing 4-16 illustrates an example use of `SimpleString` where you append several strings and print intermediate results to the console.

```
#include <cstdio>
#include <cstring>
#include <exception>

struct SimpleString {
  --snip--
}
```

```
int main() {
  SimpleString string{ 115 }; ❶
  string.append_line("Starbuck, whaddya hear?");
  string.append_line("Nothin' but the rain."); ❷
  string.print("A: "); ❸
  string.append_line("Grab your gun and bring the cat in.");
  string.append_line("Aye-aye sir, coming home."); ❹
  string.print("B: "); ❺
  if (!string.append_line("Galactica!")) { ❻
    printf("String was not big enough to append another message."); ❼
  }
}
```

Listing 4-16: The methods of `SimpleString`

First, you create a `SimpleString` with `max_length=115` ❶. You use the `append_line` method twice ❷ to add some data to `string` and then print the contents along with the tag `A` ❸. You then append more text ❹ and print the contents again, this time with the tag `B` ❺. When `append_line` determines that `SimpleString` has run out of space ❻, it returns `false` ❼. (It's your responsibility as a user of `string` to check for this condition.)

Listing 4-17 contains output from running this program.

```
A: Starbuck, whaddya hear? ❶
Nothin' but the rain.
B: Starbuck, whaddya hear? ❷
Nothin' but the rain.
Grab your gun and bring the cat in.
Aye-aye sir, coming home.
String was not big enough to append another message. ❸
```

Listing 4-17: Output from running the program in Listing 4-16

As expected, the string contains `Starbuck, whaddya hear?\nNothin' but the rain.\n` at `A` ❶. (Recall from Chapter 2 that `\n` is the newline special character.) After appending `Grab your gun and bring the cat in.` and `Aye-aye sir, coming home.`, you get the expected output at `B` ❷.

When Listing 4-17 tries to append `Galactica!` to `string`, `append_line` returns `false` because there is not enough space in `buffer`. This causes the message `String was not big enough to append another message` to print ❸.

Composing a SimpleString

Consider what happens when you define a class with a `SimpleString` member, as demonstrated in Listing 4-18.

```
#include <stdexcept>

struct SimpleStringOwner {
  SimpleStringOwner(const char* x)
    : string{ 10 } { ❶
    if (!string.append_line(x)) {
      throw std::runtime_error{ "Not enough memory!" };
```

```
    }
    string.print("Constructed: ");
  }
  ~SimpleStringOwner() {
    string.print("About to destroy: "); ❷
  }
private:
  SimpleString string;
};
```

Listing 4-18: The implementation of `SimpleStringOwner`

As suggested by the member initializer ❶, `string` is fully constructed, and its class invariants are established once the constructor of `SimpleStringOwner` executes. This illustrates the order of an object's members during construction: *members are constructed before the enclosing object's constructor.* This makes sense: how can you establish a class's invariants if you don't know about its members' invariants?

Destructors work the opposite way. Inside `~SimpleStringOwner()` ❷, you need the class invariants of `string` to hold so you can print its contents. *All members are destructed after the object's destructor is invoked.*

Listing 4-19 exercises a `SimpleStringOwner`.

```
--snip--
int main() {
  SimpleStringOwner x{ "x" };
  printf("x is alive\n");
}
```

```
Constructed: x ❶
x is alive
About to destroy: x ❷
```

Listing 4-19: A program containing a `SimpleStringOwner`

As expected, the member `string` of `x` is created appropriately because *an object's member constructors are called before the object's constructor*, resulting in the message `Constructed: x` ❶. As an automatic variable, `x` is destroyed just before `main` returns, and you get `About to destroy: x` ❷. The member `string` is still valid at this point because member destructors are called after the enclosing object's destructor.

Call Stack Unwinding

Listing 4-20 demonstrates how exception handling and stack unwinding work together. You establish a `try-catch` block in `main` and then make a series of function calls. One of these calls causes an exception.

```
--snip--
void fn_c() {
  SimpleStringOwner c{ "ccccccccc" }; ❶
}
```

```
void fn_b() {
  SimpleStringOwner b{ "b" };
  fn_c(); ❷
}

int main() {
  try { ❸
    SimpleStringOwner a{ "a" };
    fn_b(); ❹
    SimpleStringOwner d{ "d" }; ❺
  } catch(const std::exception& e) { ❻
    printf("Exception: %s\n", e.what());
  }
}
```

Listing 4-20: A program illustrating the use of `SimpleStringOwner` and call stack unwinding

Listing 4-21 shows the results of running the program in Listing 4-20.

```
Constructed: a
Constructed: b
About to destroy: b
About to destroy: a
Exception: Not enough memory!
```

Listing 4-21: Output from running the program in Listing 4-20

You've set up a `try-catch` block ❸. The first `SimpleStringOwner`, `a`, gets constructed without incident, and you see `Constructed: a` printed to the console. Next, `fn_b` is called ❹. Notice that you're still in the `try-catch` block, so any `exception` that gets thrown *will* be handled. Inside `fn_b`, another `SimpleStringOwner`, `b`, gets constructed successfully, and `Constructed: b` is printed to the console. Next, there's a call into yet another function, `fn_c` ❷.

Let's pause for a moment to take an account of what the call stack looks like, what objects are alive, and what the exception-handling situation looks like. You have two `SimpleStringOwner` objects alive and valid: `a` and `b`. The call stack looks like `fn()` → `fn_b()` → `fn_c()`, and you have an exception handler set up inside `main` to handle any exceptions. Figure 4-3 summarizes this situation.

At ❶, you run into a little problem. Recall that `SimpleStringOwner` has a member `SimpleString` that is always initialized with a `max_size` of 10. When you try to construct `c`, the constructor of `SimpleStringOwner` throws an `exception` because you've tried to append `"ccccccccc"`, which has length 10 and is too big to fit alongside a newline and a null terminator.

Now you have an exception in flight. The stack will unwind until an appropriate handler is found, and all objects that fall out of scope as a result of this unwinding will be destructed. The handler is all the way up the stack ❻, so `fn_c` and `fn_b` unwind. Because `SimpleStringOwner b` is an automatic variable in `fn_b`, it gets destructed and you see `About to destroy: b` printed to the console. After `fn_b`, the automatic variables inside `try{}` are destroyed. This includes `SimpleStringOwner a`, so you see `About to destroy: a` printed to the console.

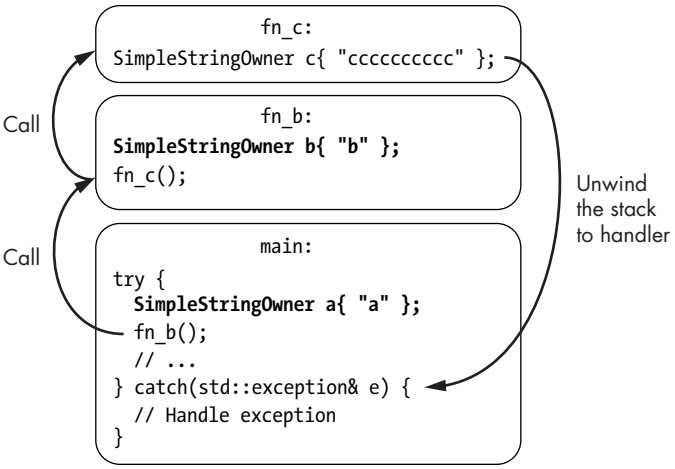

Figure 4-3: The call stack when `fn_c` calls the constructor of `SimpleStringOwner c`

Once an exception occurs in a `try{}` block, no further statements execute. As a result, `d` never initializes ❺, and you never see the constructor of `d` print to console. After the call stack is unwound, execution proceeds immediately to the `catch` block. In the end, you print the message `Exception: Not enough memory!` to the console ❻.

Exceptions and Performance

In your programs, you must handle errors; errors are unavoidable. When you use exceptions correctly and no errors occur, your code is faster than manually error-checked code. If an error does occur, exception handling can sometimes be slower, but you make huge gains in robustness and maintainability over the alternative. Kurt Guntheroth, the author of *Optimized C++*, puts it well: "use of exception handling leads to programs that are faster when they execute normally, and better behaved when they fail." When a C++ program executes normally (without exceptions being thrown), there is no runtime overhead associated with checking exceptions. It's only when an exception is thrown that you pay overhead.

Hopefully, you're convinced of the central role exceptions play in idiomatic C++ programs. Sometimes, unfortunately, you won't be able to use exceptions. One example is embedded development where real-time guarantees are required. Tools simply don't (yet) exist in this setting. With luck, this will change soon, but for now, you're stuck without exceptions in most embedded contexts. Another example is with some legacy code. Exceptions are elegant because of how they fit in with RAII objects. When destructors are responsible for cleaning up resources, stack unwinding is a direct and effective way to guarantee against resource leakages. In legacy code, you might find manual resource management and error handling instead of RAII objects. This makes using exceptions very dangerous, because stack unwinding is safe only with RAII objects. Without them, you could easily leak resources.

Alternatives to Exceptions

In situations where exceptions are not available, all is not lost. Although you'll need to keep track of errors manually, there are some helpful C++ features that you can employ to take the sting out a bit. First, you can manually enforce class invariants by exposing some method that communicates whether the class invariants could be established, as shown here:

```
struct HumptyDumpty {
   HumptyDumpty();
   bool is_together_again();
  --snip--
};
```

In idiomatic C++, you would just throw an exception in the constructor, but here you must remember to check and treat the situation as an error condition in your calling code:

```
bool send_kings_horses_and_men() {
  HumptyDumpty hd{};
  if (hd.is_together_again()) return false;
  // Class invariants of hd are now guaranteed.
  // Humpty Dumpty had a great fall.
  --snip--
  return true;
}
```

The second, complementary coping strategy is to return multiple values using *structured binding declaration*, a language feature that allows you to return multiple values from a function call. You can use this feature to return success flags alongside the usual return value, as demonstrated in Listing 4-22.

```
struct Result { ❶
  HumptyDumpty hd;
  bool success;
};

Result make_humpty() { ❷
  HumptyDumpty hd{};
  bool is_valid;
  // Check that hd is valid and set is_valid appropriately
  return { hd, is_valid };
}

bool send_kings_horses_and_men() {
  auto [hd, success] = make_humpty(); ❸
  if(!success) return false;
  // Class invariants established
  --snip--
  return true;
}
```

Listing 4-22: A code segment illustrating structured binding declaration

First, you declare a POD that contains a `HumptyDumpty` and a `success` flag ❶. Next, you define the function `make_humpty` ❷, which builds and validates a `HumptyDumpty`. Such methods are called *factory methods*, because their purpose is to initialize objects. The `make_humpty` function packs this and the success flag into a `Result` when it returns. The syntax at the call site ❸ illustrates how you can unpack the `Result` into multiple, `auto`-type-deduced variables.

NOTE *You'll explore structured bindings in more detail in "Structured Bindings" on page 222.*

Copy Semantics

Copy semantics is "the meaning of copy." In practice, programmers use the term to mean the rules for making copies of objects: after `x` is *copied into* `y`, they're *equivalent* and *independent*. That is, `x == y` is true after a copy (equivalence), and a modification to `x` doesn't cause a modification of `y` (independence).

Copying is extremely common, especially when passing objects to functions by value, as demonstrated in Listing 4-23.

```
#include <cstdio>

int add_one_to(int x) {
  x++; ❶
  return x;
}

int main() {
  auto original = 1;
  auto result = add_one_to(original); ❷
  printf("Original: %d; Result: %d", original, result);
}
```

```
Original: 1; Result: 2
```

Listing 4-23: A program illustrating that passing by value generates a copy

Here, `add_one_to` takes its argument `x` by value. It then modifies the value of `x` ❶. This modification is isolated from the caller ❷; `original` is unaffected because `add_one_to` gets a copy.

For user-defined POD types, the story is similar. Passing by value causes each member value to be copied into the parameter (a *member-wise copy*), as demonstrated in Listing 4-24.

```
struct Point {
  int x, y;
};

Point make_transpose(Point p) {
  int tmp = p.x;
  p.x = p.y;
```

```
  p.y = tmp;
  return p;
}
```

Listing 4-24: The function `make_transpose` generates a copy of the POD type `Point`.

When `make_transpose` is invoked, it receives a copy `Point` in `p`, and the original is unaffected.

For fundamental and POD types, the story is straightforward. Copying these types is memberwise, which means each member gets copied into its corresponding destination. This is effectively a bitwise copy from one memory address to another.

Fully featured classes require some more thought. The default copy semantics for fully featured classes is also the memberwise copy, and this can be extremely dangerous. Consider again the `SimpleString` class. You would invite disaster if you allowed a user to make a memberwise copy of a live `SimpleString` class. Two `SimpleString` classes would point to the same `buffer`. With both of the copies appending to the same `buffer`, they'll clobber each other. Figure 4-4 summarizes the situation.

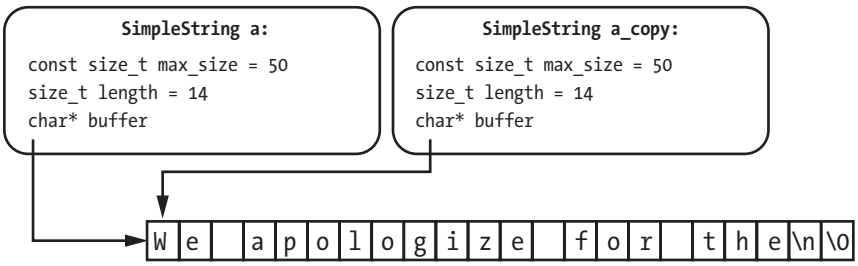

Figure 4-4: A depiction of default copy semantics on the `SimpleString` class

This result is bad, but even worse things happen when the `SimpleString` classes start destructing. When one of the `SimpleString` classes is destructed, `buffer` will be freed. When the remaining `SimpleString` class tries to write its `buffer`—bang!—you have undefined behavior. At some point, this remaining `SimpleString` class will be destructed and free `buffer` again, resulting in what is commonly called a *double free*.

NOTE *Like its nefarious cousin the use after free, the double free can result in subtle and hard-to-diagnose bugs that manifest only very infrequently. A double free occurs when you deallocate an object twice. Recall that once you've deallocated an object, its storage lifetime ends. This memory is now in an undefined state, and if you destruct an object that's already been destructed, you've got undefined behavior. In certain situations, this can cause serious security vulnerabilities.*

You can avoid this dumpster fire by taking control of copy semantics. You can specify copy constructors and copy assignment operators, as described in the following sections.

Copy Constructors

There are two ways to copy an object. One is to use *copy construction*, which creates a copy and assigns it to a brand-new object. The copy constructor looks like other constructors:

```
struct SimpleString {
  --snip--
  SimpleString(const SimpleString& other);
};
```

Notice that `other` is `const`. You're copying from some original `SimpleString`, and you have no reason to modify it. You use the copy constructor just like other constructors, using the uniform initialization syntax of braced initializers:

```
SimpleString a;
SimpleString a_copy{ a };
```

The second line invokes the copy constructor of `SimpleString` with `a` to yield `a_copy`.

Let's implement the copy constructor of `SimpleString`. You want what is known as a *deep copy* where you copy the data pointed to by the original `buffer` into a new `buffer`, as depicted in Figure 4-5.

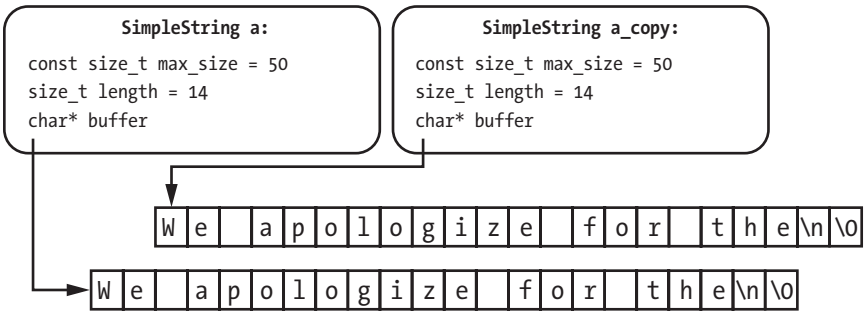

Figure 4-5: A depiction of a deep copy on the `SimpleString` class

Rather than copying the pointer `buffer`, you'll make a new allocation on the free store and then copy all the data pointed to by the original `buffer`. This gives you two independent `SimpleString` classes. Listing 4-25 implements the copy constructor of `SimpleString`:

```
SimpleString(const SimpleString& other)
  : max_size{ other.max_size }, ❶
    buffer{ new char[other.max_size] }, ❷
    length{ other.length } { ❸
    std::strncpy(buffer, other.buffer, max_size); ❹
}
```

Listing 4-25: `SimpleString` class's copy constructor

You use member initializers for `max_size` ❶, `buffer` ❷, and `length` ❸ and pass in the corresponding fields on `other`. You can use array new ❶ to initialize `buffer` because you know `other.max_size` is greater than 0. The copy constructor's body contains a single statement ❹ that copies the contents pointed to by `other.buffer` into the array pointed to by `buffer`.

Listing 4-26 uses this copy constructor by initializing a `SimpleString` with an existing `SimpleString`:

```
--snip--
int main() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the");
  SimpleString a_copy{ a }; ❶
  a.append_line("inconvenience."); ❷
  a_copy.append_line("incontinence."); ❸
  a.print("a");
  a_copy.print("a_copy");
}
```

```
a: We apologize for the
inconvenience.
a_copy: We apologize for the
incontinence.
```

Listing 4-26: A program using `SimpleString` class's copy constructor

In the program, `SimpleString a_copy` ❶ is copy constructed from `a`. It's equivalent to—and independent from—the original. You can append different messages to the end of `a` ❷ and `a_copy` ❸, and the changes are isolated.

The copy constructor is invoked when passing `SimpleString` into a function by value, as demonstrated in Listing 4-27.

```
--snip--
void foo(SimpleString x) {
  x.append_line("This change is lost.");
}

int main() {
  SimpleString a { 20 };
  foo(a); // Invokes copy constructor
  a.print("Still empty");
}
```

```
Still empty:
```

Listing 4-27: A program illustrating that copy constructors get invoked when passing an object by value

NOTE *You shouldn't pass by value to avoid modification. Use a `const` reference.*

The performance impact of copying can be substantial, especially in a situation where free store allocations and buffer copies are involved. For example, suppose you have a class that manages the life cycle of a gigabyte

of data. Each time you copy the object, you'll need to allocate and copy a gigabyte of data. This can take a lot of time, so you should be absolutely sure you need the copy. If you can get away with passing a `const` reference, strongly prefer it.

Copy Assignment

The other way to make a copy in C++ is with the *copy assignment operator*. You can create a copy of an object and assign it to another existing object, as demonstrated in Listing 4-28.

```
--snip--
void dont_do_this() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the");
  SimpleString b{ 50 };
  b.append_line("Last message");
  b = a; ❶
}
```

Listing 4-28: Using the default copy assignment operator to create a copy of an object and assign it to another existing object

NOTE *The code in Listing 4-28 causes undefined behavior because it doesn't have a user-defined copy assignment operator.*

The line at ❶ *copy assigns* `a` to `b`. The major difference between copy assignment and copy construction is that in copy assignment, `b` might already have a value. You must clean up `b`'s resources before copying `a`.

WARNING *The default copy assignment operator for simple types just copies the members from the source object to the destination object. In the case of `SimpleString`, this is very dangerous for two reasons. First, the original `SimpleString` class's buffer gets rewritten without freeing the dynamically allocated `char` array. Second, now two `SimpleString` classes own the same buffer, which can cause dangling pointers and double frees. You must implement a copy assignment operator that performs a clean hand-off.*

The copy assignment operator uses the `operator=` syntax, as demonstrated in Listing 4-29.

```
struct SimpleString {
  --snip--
  SimpleString& operator=(const SimpleString& other) {
    if (this == &other) return *this; ❶
    --snip--
    return *this; ❷
  }
}
```

Listing 4-29: A user-defined copy assignment operator for `SimpleString`

The copy assignment operator returns a reference to the result, which is always `*this` ❷. It's also generally good practice to check whether `other` refers to `this` ❶.

You can implement copy assignment for `SimpleString` by following these guidelines: free the current `buffer` of `this` and then copy `other` as you did in copy construction, as shown in Listing 4-30.

```
SimpleString& operator=(const SimpleString& other) {
  if (this == &other) return *this;
  const auto new_buffer = new char[other.max_size]; ❶
  delete[] buffer; ❷
  buffer = new_buffer; ❸
  length = other.length; ❹
  max_size = other.max_size; ❺
  strcpy_s(buffer, max_size, other.buffer); ❻
  return *this;
}
```

Listing 4-30: A copy assignment operator for `SimpleString`

The copy assignment operator starts by allocating a `new_buffer` with the appropriate size ❶. Next, you clean up `buffer` ❷. The rest is essentially identical to the copy constructor in Listing 4-25. You copy `buffer` ❸, `length` ❹, and `max_size` ❺ and then copy the contents from `other.buffer` into your own `buffer` ❻.

Listing 4-31 illustrates how `SimpleString` copy assignment works (as implemented in Listing 4-30).

```
--snip--
int main() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the"); ❶
  SimpleString b{ 50 };
  b.append_line("Last message"); ❷
  a.print("a"); ❸
  b.print("b"); ❹
  b = a; ❺
  a.print("a"); ❻
  b.print("b"); ❼
}
-----------------------------------------------------------------------------
a: We apologize for the ❸
b: Last message ❹
a: We apologize for the ❻
b: We apologize for the ❼
```

Listing 4-31: A program illustrating copy assignment with the `SimpleString` class

You begin by declaring two `SimpleString` classes with different messages: the string `a` contains `We apologize for the` ❶, and `b` contains `Last`

message ❷. You print these strings to verify that they contain the text you've specified ❸❹. Next, you copy assign b equal to a ❺. Now, a and b contain copies of the same message, We apologize for the ❻❼. But—and this is important—that message resides in two separate memory locations.

Default Copy

Often, the compiler will generate default implementations for copy construction and copy assignment. The default implementation is to invoke copy construction or copy assignment on each of a class's members.

Any time a class manages a resource, you must be extremely careful with default copy semantics; they're likely to be wrong (as you saw with SimpleString). Best practice dictates that you explicitly declare that default copy assignment and copy construction are acceptable for such classes using the default keyword. The Replicant class, for example, has default copy semantics, as demonstrated here:

```
struct Replicant {
  Replicant(const Replicant&) = default;
  Replicant& operator=(const Replicant&) = default;
  --snip--
};
```

Some classes simply cannot or should not be copied—for example, if your class manages a file or if it represents a mutual exclusion lock for concurrent programming. You can suppress the compiler from generating a copy constructor and a copy assignment operator using the delete keyword. The Highlander class, for example, cannot be copied:

```
struct Highlander {
  Highlander(const Highlander&) = delete;
  Highlander& operator=(const Highlander&) = delete;
  --snip--
};
```

Any attempt to copy a Highlander will result in a compiler error:

```
--snip--
int main() {
  Highlander a;
  Highlander b{ a }; // Bang! There can be only one.
}
```

I highly recommend that you explicitly define the copy assignment operator and copy constructor for *any* class that owns a resource (like a printer, a network connection, or a file). If custom behavior is not needed, use either default or delete. This will save you from a lot of nasty and difficult-to-debug errors.

Copy Guidelines

When you implement copy behavior, think about the following criteria:

Correctness You must ensure that class invariants are maintained. The `SimpleString` class demonstrated that the default copy constructor can violate invariants.

Independence After copy assignment or copy construction, the original object and the copy shouldn't change each other's state during modification. Had you simply copied `buffer` from one `SimpleString` to another, writing to one `buffer` could overwrite the data from the other.

Equivalence The original and the copy should be the *same*. The semantics of sameness depend on context. But generally, an operation applied to the original should yield the same result when applied to the copy.

Move Semantics

Copying can be quite time-consuming at runtime when a large amount of data is involved. Often, you just want to *transfer ownership* of resources from one object to another. You could make a copy and destroy the original, but this is often inefficient. Instead, you can *move*.

Move semantics is move's corollary to copy semantics, and it requires that after an object `y` is *moved into* an object `x`, `x` is equivalent to the former value of `y`. After the move, `y` is in a special state called the *moved-from* state. You can perform only two operations on moved-from objects: (re)assign them or destruct them. Note that moving an object `y` into an object `x` isn't just a renaming: these are separate objects with separate storage and potentially separate lifetimes.

Similar to how you specify copying behavior, you specify how objects move with *move constructors* and *move assignment operators*.

Copying Can Be Wasteful

Suppose you want to move a `SimpleString` into a `SimpleStringOwner` in the following way:

```
--snip--
void own_a_string() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the");
  a.append_line("inconvenience.");
  SimpleStringOwner b{ a };
  --snip--
}
```

You could add a constructor for `SimpleStringOwner` and then copy-construct its `SimpleString` member, as demonstrated in Listing 4-32.