Programming and Data Structures

Release 0.3

Amir Kamil

Sep 08, 2024

CONTENTS

List of Lectures

Part I

Foundations

CHAPTER

ONE

INTRODUCTION

Welcome to EECS 280: Programming and Introductory Data Structures! This course covers several fundamental concepts in programming, including basic principles such as procedural and data abstraction, resource management, and basic data structures. The following is the official course description:

> Techniques and algorithm development and effective programming, top-down analysis, structured programming, testing, and program correctness. Program language syntax and static and runtime semantics. Scope, procedure instantiation, recursion, abstract data types, and parameter passing methods. Structured data types, pointers, linked data structures, stacks, queues, arrays, records, and trees.

This course, and Computer Science in general, is *not* about computers, as stated by Hal Abelson, author of the seminal textbook Structure and Interpretation of Computer Programs:

> [Computer science] is not really about computers – and it's not about computers in the same sense that physics is not really about particle accelerators, and biology is not about microscopes and Petri dishes…and geometry isn't really about using surveying instruments.

Instead, the purpose of this course is to examine the *generalizable concepts* in programming. To understand what a generalizable concept is, let's take a diversion from programming and consider a concept in baking. How many eggs do you put in a cake (assuming a single-layer, eight-inch cake)? Eggs serve several functions in a cake, including providing structure, influencing the texture and smoothness, providing moisture, and contributing to the taste of the cake. As an experiment, Summer Stone of The Cake Blog varied the number of eggs in a recipe between zero and four, finding the following:

- 0 eggs: The cake was short and dense, and it fell apart when cut. It also tasted very much like flour.
- 1 egg: The cake was dense and compact, but held together when cut and had a pleasant taste.
- 2 eggs: The cake had a greater height and lighter texture and also had a pleasant taste.
- 3 eggs: The cake was even taller and lighter, with a slightly spongy texture, and still had a pleasant taste.
- 4 eggs: The cake was short, dense, and rubbery and tasted very much like egg.

The generalizable concept here is that eggs provide structure to a cake, and that more eggs results in a lighter cake, up to a point. Thus, the structure and texture of a cake can be manipulated by varying the number of eggs, as long as there aren't too few or too many.

The topics that we will cover in this course concern the generalizable concepts in programming, including procedural abstraction, data abstraction, dynamic resource management, object orientation, and many others. We will use C++ as our vehicle for learning these concepts, and we will implement several large programming projects to develop our experience and understanding of the concepts. However, learning C++ and implementing projects aren't the end goal; rather, they are the mechanisms we use to explore the topics.

CHAPTER

TWO

C++ FUNDAMENTALS

C++ is a *programming language*, a language designed for expressing programs at a higher level than *machine code*, which is what directly runs on a computer's central processing unit (CPU). For example, Intel and AMD processors (typically used in Windows machines) implement the x86_64 *instruction set architecture (ISA)*, which results in machine code that looks like the following:

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], 3
mov     DWORD PTR [rbp-8], 4
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx
```

Apple CPUs, as well as most mobile processors, implement the AArch64 ISA, with code like:

```
sub     sp, sp, #16
mov     w0, 3
str     w0, [sp, 12]
mov     w0, 4
str     w0, [sp, 8]
ldr     w1, [sp, 12]
ldr     w0, [sp, 8]
add     w0, w1, w0
```

Machine code is extremely low level, so rather than writing programs in it directly, we prefer to use a higher-level programming language that allows us to write more readable code that can work on many different CPU architectures. We can then use a *compiler* to translate the program into the equivalent machine code to run directly on a CPU. Examples of compilers for C++ include `g++` (the GNU C++ compiler), Clang, and MSVC (Microsoft Visual C++).

Alternatively, we can run a higher-level program through an *interpreter*, which rather than translating the program into machine code, directly emulates the behavior of the program. Lobster is an example of an interpreter for (a subset of) C++.

To illustrate the compilation process, let's start with a small C++ program that prints to *standard output* (typically displaying in a console/terminal):

```
#include <iostream>

using namespace std;

int main() {
  // print a greeting
```

(continues on next page)

(continued from previous page)

```
  cout << "Hello World!" << endl;

  // Declare some variables
  int x = 10 + 5;
  int y = 0;
  y = x;
  x = 20;

  // Print out the result
  cout << "x = " << x << ", y = " << y << endl;
}
```

We will momentarily discuss what each part of the program means, but first let's compile and run the code. Assuming that the code is located in the file `hello.cpp`, we can compile it from the terminal (or *command line*) with `g++`:

```
g++ --std=c++17 hello.cpp -o hello.exe
```

The items following `g++` are *arguments* (or *command-line arguments* to be precise) to the compiler:

- `--std=c++17` tells the compiler to use the C++17 version of the language
- `hello.cpp` is our program
- `-o hello.exe` tells the compiler to name the resulting machine-code file, or *executable*, as `hello.exe`

After running the above command, we can now run the resulting executable from the command line:

```
./hello.exe
```

Here, `./` tells the terminal that the executable is in the current *directory* (or *folder*). The terminal runs the program and displays the printed output:

```
Hello World!
x = 20, y = 15
```

Returning to the program source code, the first line (`#include <iostream>`) imports functionality from the C++ standard library, allowing us to write output to the screen and read input from the keyboard. In particular, it defines `cout`, which we can use with the *insertion operator* `<<` to print values to standard output. The `cout` variable is actually defined inside the `std` (short for "standard") *namespace*, which is a way to organize names to avoid conflicts with other libraries. There are several ways we can use a name defined within a namespace:

- with a *qualified name* that includes the namespace, as in `std::cout`:

  ```
  #include <iostream>

  int main() {
    // print a greeting
    std::cout << "Hello World!" << std::endl;
  }
  ```

- by first importing each name we want to use directly via *using declarations*, after which we can use the name without qualification:

  ```
  #include <iostream>
  ```

(continues on next page)

(continued from previous page)

```
using std::cout;
using std::endl;

int main() {
  // print a greeting
  cout << "Hello World!" << endl;
}
```

- by importing all the names from a namespace at once:

```
#include <iostream>

using namespace std;

int main() {
  // print a greeting
  cout << "Hello World!" << endl;
}
```

Be careful with the last option, as there can be many names defined within a namespace, and importing them all increases the likelihood of there being a conflict with names defined outside of the namespace. For simplicity, we will use it here.

After the initial two lines, we have a definition of a `main()` function, which is the entry point of a C++ program. The function returns an integer, represented by the `int` type, and that comes before the function name itself. For now, we define `main()` to not take any arguments. (If the function takes arguments, there would be *function parameters* between the parentheses; we will return to this later.) Then we have the body of the function, enclosed by a pair of matching curly braces. Within the body, we have a sequence of *statements*, which are executed in order. The first line of the body is actually a *comment* rather than a statement:

```
// print a statement
```

This does not do anything, but it serves as documentation for someone reading the code. We can write a comment by starting it with double slashes, in which case the rest of the line comprises the comment. Alternatively, we can use the sequence `/*` to open a comment and `*/` to close it, which allows a comment to span multiple lines or part of a line:

```
int main() /* a comment here */ {
  /* here is a
     multiline
     comment */
}
```

After the initial comment, we have a statement that prints to standard output:

```
cout << "Hello World!" << endl;
```

We use the insertion operator `<<` to insert the *string* `"Hello World!"` – a string is a sequence of characters, and we can write one directly in our program by enclosing it in double quotes. (Be aware that unlike some other languages, C++ strings cannot span multiple lines in the source code.) After inserting the string, we *chain* another insertion of `endl`, which is defined by `<iostream>` in the `std` namespace. Inserting `endl` writes a newline, and it also *flushes* the output, meaning that it forces the output to appear immediately. (Without flushing, the output can be *buffered* until a later time, which can result in better performance at the expense of delaying output.) Finally, we end the statement with a semicolon, which is required at the end of simple statements in C++. Unlike Python and some other languages,

whitespace is not generally significant in C++ programs, so the end of a line does not automatically end a statement. We can see what happens if we forget a semicolon:

```
#include <iostream>

using namespace std;

int main() {
  // print a greeting
  cout << "Hello World!" << endl // oops
}
```

Attempting to compile the program results in the following:

```
hello.cpp:7:33: error: expected ';' after expression
  cout << "Hello World!" << endl // oops
                                ^
                                ;
1 error generated.
```

This is a *compile error*, and it is the compiler informing us that we have a mistake in our code. The compiler does not produce an executable in such a case, and we have to fix our error and recompile.

Continuing where we left off in our program, we have the following line:

```
int x = 10 + 5;
```

This is a *variable declaration*, which introduces a new variable, and in C++, it has three components:

- the type of the variable (`int` for an integer in this case)
- the name of the variable (`x` here)
- an optional *initialization expression* (`10 + 5` above), which specifies the initial value of the variable

Unlike languages like Python, C++ is *statically typed*, meaning that the type of a variable must be known at compile time, generally by explicitly specifying the type in the variable's declaration. And the type of the variable never changes – for as long as it exists, it will have the type specified in its declaration.

A variable's initialization is an *expression*, which is a fragment of code that *evaluates* to some value. (An expression may also have a *side effect*, which modifies the state of the program in some way, such as by printing to the screen or changing the value of an existing variable.) Expressions may be composed of the following elements:

- *literals*, which specify a value directly in source code, such as `42` or `"Hello World!"`
- variables
- functions calls, such as `sqrt(x)`
- operators such as + and <<

In the case of the variable declaration above, the initialization expression `10 + 5` evaluates to the value 15, which then becomes the initial value of the variable `x`.

We will see later when initialization expressions are required. For now, we note that for a variable of type `int`, without an initialization expression, the variable would have an **undefined value**, which would result in *undefined behavior* if we were to use its value. Undefined behavior means that we cannot rely on the outcome – the program might work, or it might crash, or it might empty your bank account. Thus, undefined behavior should be avoided at all costs.

Examining the following lines,

```
int y = 0;
y = x;
x = 20;
```

we see that the variable y is introduced with initial value 0. Then we have an *assignment*, which copies the value from the right-hand side into the left – in this case, the value of x, which is 15, is copied into y. The next line then assigns the value 20 to x. This, when the last line

```
cout << "x = " << x << ", y = " << y << endl;
```

executes, we see that x has the value 20 and y the value 15.

A variable can only be used when it is *in scope*. A *scope* is a region of source code where names are meaningful, and such a region often corresponds to a *block* of code delimited by curly braces. A variable is in scope starting at its declaration until the end of the scope region in which it is defined. For example, consider the following code:

```
int main() {
  int x = 5; // x is now in scope

  cout << y << endl; // COMPILE ERROR -- y is not in scope

  int y = -3; // y is now in scope

  if (x > y) {
    int a = x - y; // a is now in scope
    cout << a << endl;
  } // a is no longer in scope

  cout << a << endl; // COMPILE ERROR -- a is not in scope
}
```

The variable y is not in scope until its declaration, so attempting to use it in the second statement is an error. Below, the curly braces associated with the conditional define a new scope region, in which the variable a is introduced. Outside of the curly braces, a is no longer in scope, so using it in the last statement is erroneous. The compiler checks for us whether or not variables are in scope, reporting errors if we try to use one that is not.

We have seen a simple program, how to compile and run it, and some fundamental C++ elements. We proceed to discuss more complex C++ features, such as type conversions, standard-library types, functions, and control flow.

CHAPTER

THREE

TYPES

Previously, we saw that in C++, a variable is declared with its type, and the remains its type as long as the variable exists. C++ has several fundamental (also called *primitive* or *atomic*) types that are built-in to the language and always available, without any `#include` lines. The following are a few such types:

- `int` represents a *signed* (positive, negative, or zero) integer, and in most implementations, can represent integers between 2^{-31} and $2^{31} - 1$, inclusive

```
int x = 3;
```

- `double` represents a floating-point number[1]

```
double y = 2.5;
```

- `bool` represents a Boolean value, i.e. true or false

```
bool z = true;
```

- `char` represents a single character (as opposed to a string, which is a sequence of characters), and a character literal is written with single quotes

```
char c = 'w';
```

As with `int`, a variable of fundamental type must be explicitly initialized – otherwise its value is undefined.

```
int x;    // undefined value
double y; // undefined value
bool z;   // undefined value
char c;   // undefined value
```

The compiler's *type system* detects misuse of types. For instance, the following attempts to initialize a `double` with a string literal, which is not a compatible type:

```
double y = "llama"; // COMPILE ERROR
```

On the other hand, there are some combinations of types for which C++ performs an *implicit conversion*, allowing one to be used where the other is expected. For instance, an `int` can be converted to a `double`, and vice versa:

```
int x = 3.1;   // narrowing conversion -- truncated to 3
double y = 42; // widening conversion -- value unchanged
```

Similarly, numeric types can be converted to `bool`, with zero values converted to `false` and non-zero values to `true`.

[1] C++ also has a `float` type that represents floating-point numbers. A `double` typically uses twice the memory space of a `float` (hence the name), and it can represent a much larger range of values.

```
bool a = 3.1; // true
bool b = 0;   // false
```

Implicit conversions, while often useful, can also be a source of error. Consider the following code, which introduces a `max()` function that computes the maximum of two values. (We will discuss function definitions shortly.)

```
double max(double x, double y) {
  int result = x;
  if (y > x) {
    result = y;
  }
  return result;
}
```

Suppose we have the call `max(3.4, 3.1)`. The first line of the function implicitly converts the value of `x`, which is 3.4, to an `int`, so that `result` contains the value 3. Then it is the case that `y`, whose value is 3.1, is larger than `result`, so `result` is assigned the truncated value of `y`, which again results in 3. Thus, the function returns the value 3 instead of 3.4.

Rather than relying on implicit conversions, which can be hard to detect when reading code, we can also do an *explicit conversion* via a *cast*. Depending on the types involved, C++ may require a cast, while in other cases it is optional. The following is an example of casting a `double` value to an `int`:

```
double x = 4.2;
int y = x;                    // implicit conversion -- easy to miss
int z = static_cast<int>(x); // explicit conversion -- obvious in code
```

While there are other kinds of casts, `static_cast` is the most common and the one we recommend.

3.1 Arithmetic and Comparisons

C++ supports common arithmetic operations, such as addition (+), subtraction (-), multiplication (*), division (/), and modulo (%). The behavior of these operators depends on the types of the operand – for example, adding two `int` values together produces an `int`, while adding two `double` values produces a `double`. The same holds for division, where dividing two `int` values *truncates* the result to produce another `int` value:

```
cout << (1 / 2) << endl; // prints 0
```

We can obtain a `double` value by ensuring that one of the operands is a `double`:

```
cout << (1.0 / 2) << endl; // prints 0.5
cout << (1 / 2.) << endl; // prints 0.5
cout << (static_cast<double>(1) / 2) << endl; // prints 0.5
```

When the operand types differ as above, one operand is *promoted* to the type of the other, generally to the one that allows for more precision. In the cases above, the `int` operand is promoted to `double` as part of the division operation.

The modulo operator requires both operands to be integers. The `std::modf()` function in the <cmath> library can be used on floating-point operands. (Similarly, the `std::pow()` function in <cmath> does exponentiation – C++ does not have an operator for that.)

As an example of modulo, the following functions convert a total number of seconds to whole minutes and leftover seconds, respectively:

```
int minutes(int seconds) {
  return seconds / 60;
}

int remaining_seconds(int seconds) {
  return seconds % 60;
}

int main() {
  int total = 153;
  int min = minutes(total); // 2
  int sec = remaining_seconds(total); // 33
}
```

We rely on integer division to truncate the result in `minutes()`, and we use the modulo operator in `remaining_seconds()` to compute the remainder of the total number of seconds with 60.

C++ also has standard comparison operators: equality (==), inequality (!=), less (<), less or equal (<=), greater (>), and greater or equal (>=). However, avoid two pitfalls when using these operators:

- Be wary about comparing floating-point numbers that result from arithmetic operations – floating-point numbers cannot in general be represented exactly on a computer, so we can get unexpected results due to the loss of precision:

  ```
  cout << (0.1 + 0.2 == 0.3) << endl; // typically prints 0 (false)
  ```

  In such cases, values should be compared to within some margin of error rather than exactly:

  ```
  bool almost_equal(double x, double y, double epsilon) {
    return std::abs(x - y) < epsilon;
  }

  ...

  cout << almost_equal(0.1 + 0.2, 0.3, 0.00001) << endl; // prints 1 (true)
  ```

  Here, we use `std::abs()` to compute the absolute value of the difference between the two values, then compare against our margin of `epsilon`.

- C++ does not support Python-style chaining of comparisons:

  ```
  int x = 10;
  cout << (3 < x < 7) << endl; // prints 1 (true)
  ```

  In this example, `3 < x` produces true, which is then promoted to the integer value of 1 for the subsequent comparison with 7. Since `1 < 7`, the result is true.

  Rather than chaining comparisons, we need to use *Boolean logic* to combine the results of two separate comparisons:

  ```
  cout << (3 < x && x < 7) << endl; // prints 0 (false)
  ```

  More on logical operations below.

3.2 Library Types

Aside from the fundamental types that are always available to C++ programs, individual *headers* in the standard library define additional types. For instance, some commonly used types include `std::string` defined by the <string> header, `std::vector` defined by the <vector> header, and `std::pair` defined by the <utility> header.

The `std::string` type represents an ordered sequence of characters. Similar to the initial program we saw last time, we first need to import the relevant header to get access to the type. We can then either use the qualified name with the `std::` prefix, or include a `using` declaration to allow us to use `string` without qualification. The following is an example:

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string str1 = "make a ";
  string str2 = "wish";

  str1 < str2; // true, alphabetic comparison

  cout << str1.size() << endl; // 7 (includes spaces)

  string str3 = str1 + str2; // "make a wish"
  str3[0] = str3[7];
  str3[7] = 'f';
  cout << str3 << endl; // "wake a fish";
}
```

We start by declaring and initializing two `string` variables. We can compare them with < and other operators – the strings are compared *lexicographically*, meaning that the characters at each position are compared in order according to the underlying numerical value for the character. In the case above, since `'m'` comes before `'w'`, `str1` compares less than `str2`. We can obtain the length of a string by calling `.size()` on it (the dot here is necessary, and we'll see how this gets implemented later when we discuss classes and member functions). We can concatenate two strings with the + operator and use square brackets to read or write an individual character in a string. In the code above, we concatenate `str1` and `str2` to produce `"make a wish"`. Then we copy the character at index 7, which is `'w`, to position 0, resulting in `"wake a wish"`. Finally, we set the character at index 7 to `'f'`, which gives us the string `"wake a fish"`.

Aside from characters, we might want to keep track of a sequence of other elements, such as `int` values. A `std::vector` is a *generic* sequence type that allows us to specify the type of element it holds. For instance, the following creates both a vector of `double` values and one that holds `string` values:

```
#include <vector>

using namespace std;

vector<double> nums = {1, 5, 3.5, 6.5};
vector<string> pets = {"cat", "dog", "fish"};
```

Here, we initialize each vector with an explicit set of values. There are other ways to initialize a vector:

```
vector<int> v;        // initialize v to be empty (NOT undefined)
vector<int> v(5);     // initialize v to contain five zeros
vector<int> v(3, 42); // initialize v to contain three 42's
```

As with a string, we can use square brackets to index into a vector:

```
vector<int> v = {3, 5, 42, 28};
cout << v[0] << endl;
v[3] = 7;
v[4] = 100; // out of bounds... undefined behavior!
```

Be careful not to access an index that is out of bounds – such an access results in undefined behavior. Alternatively, we can use the `.at()` function, which checks whether we are within the bounds and throws an *exception* if we are not:

```
vector<int> v = {3, 5, 42, 28};
cout << v[0] << endl;
v.at(3) = 7;
v.at(4) = 100; // out of bounds... throws an exception
```

Assuming we don't handle the exception, this causes the program to crash, which is better than undefined behavior – the crash immediately tells us we did something wrong, and we can run the code through a debugger to get more details.

We can also modify the size of a vector by adding and removing elements. In particular, the `.push_back()` function adds an element to the end, and the `.pop_back()` function removes the last element. We can also remove all elements with `.clear()`:

```
vector<int> v;
// v contains {}

v.push_back(1);
v.push_back(2);
v.push_back(3);
// v contains {1, 2, 3}

v.pop_back();
// v contains {1, 2}

v.clear();
// v contains {}
```

A few other useful vector functions are the following:

- `.size()` returns the number of elements in the vector
- `.front()` returns a reference to the first element
- `.back()` returns a reference to the last element
- `.empty()` returns whether or not the vector is empty

For both strings and vectors, `.size()` returns the size as the `size_t` type, which is an *unsigned* integer, meaning that it cannot represent negative values. If we compare to a signed integer such as an `int`, the compiler may warn us that we are comparing a signed and unsigned integer, which might produce surprising results:

```
int x = -3;
size_t s = 5;
cout << (s > x) << endl;  // prints 0 (i.e. false)
```

The following is an example of a compiler warning for this:

```
foo.cpp:9:14: warning: comparison of integers of different signs:
'size_t' (aka 'unsigned long') and 'int' [-Wsign-compare]
  cout << (s > x) << endl;
            ~ ^ ~
1 warning generated.
```

To avoid this, we can use a cast to ensure that we are comparing values with the same "signedness":

```
cout << (static_cast<int>(s) > x) << endl;  // prints 1 (i.e. true)
```

One last library type we examine now is `std::pair`, which represents a pair of values rather than an arbitrary sequence. The following is an example of using pairs:

```
#include <utility>

int main() {
  std::pair<int, bool> p1;  // initialized to {0, false}
  std::pair<int, bool> p2 = {-3, true};
  p1.first = 7;
  p1.second = true;
  // p1 now is {7, true}

  std::pair<string, double> p3; // initialized to {"", 0.0}
  std::pair<string, double> p4 = {"hello", 73};
  p3 = p4; // copy the values from p4 to p3
  // p3 now is {"hello", 73}
}
```

CHAPTER

FOUR

CONTROL STRUCTURES

C++ is an *imperative* language, meaning that we specify computation as a sequence of statements. Like other languages, we often organize statements into functions, which we can then use as abstractions; we will return to this *shortly*. The syntax of a function definition is as follows:

```
<return type> <function name>(<optional parameters>) {
  <body statements>
}
```

The return type is specified first, and it corresponds to the type of value that the function returns. In some cases, a function does not return a value at all, in which case we use the special type `void` to specify that this is the case:

```
void print_value(int x) {
  cout << x << endl;
}
```

If a function has a non-void return type, we explicitly provide the return value using a *return statement*:

```
double square(double x) {
  return x * x;
}
```

We need to ensure that every path through a non-void function reaches a return statement that provides a value compatible with the return type. Otherwise, the behavior of the function is undefined, and the compiler may generate a warning or error.[2]

After the return type, we specify the name of the function, and if it takes arguments, the function parameters within the following parentheses. A parameter specifies both the type of the value it expects, as well as the name we use to refer to the corresponding argument within the function body. Finally, we have the body consisting of a sequence of statements enclosed by curly braces.

We can specify more complex control flow within a function through the use of compound statements such as conditionals and loops. The following demonstrates a conditional:

```
double abs(double x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

[2] The one exception in C++ is the `main()` function, which implicitly returns the value 0 if no return statement is reached. However, this is not the case for any other function, even if its return type is `int`.

We use the `if` keyword to introduce a conditional, followed by a test expression within parentheses. The "then" branch can be a single statement, or it can be a *block* of statements enclosed by curly braces. We recommend using blocks, as that makes it clear what is part of a branch and what isn't. For the "else" branch, we use the `else` keyword followed by a statement or block. We can also *chain* conditionals using `else if`:

```
double describe(int x) {
  if (x < 0) {
    cout << "negative" << endl;
  } else if (x > 0) {
    cout << "positive" << endl;
  } else {
    cout << "zero" << endl;
  }
}
```

At most one branch a conditional chain may execute.

A `for` statement can be used to express a loop that repeats until some condition is false. Its syntax is as follows:

```
for (<initialization>; <test>; <update>) {
  <body statements>
}
```

The initialization is run once at the beginning of the loop, and it can introduce new variables – the scope of such a variable is just the loop itself. The test is run before each iteration of the loop – if the test is true, the loop body runs, otherwise the loop exits. The update is run after each loop iteration completes.[3] As with a conditional, the body may be a single statement or a block – we recommend the latter.

As an example, let's compute the sum of the elements in a vector:

```
vector<double> values = /* fill with some values */;
double sum = 0;
for (size_t i = 0; i < values.size(); ++i) {
  sum += values[i];
}
```

We initialize a new variable `i` with value 0, representing the current index into the vector. We declare `i` to be of type `size_t`, so that we don't get a compiler warning by comparing it against `values.size()`, which produces a `size_t`. We execute the loop body as long as our index `i` is less than the size of the vector. After each iteration, `i` gets incremented, and the loop exits once its value is equal to `values.size()`.

In addition to for loops, C++ has while loops, which are a simplification of for loops:

```
while (<test>) {
  <body statements>
}
```

The loop will execute the body as long as the test is true, and it will exit when the test becomes false. We can translate our for loop above to a while loop as follows:

```
size_t i = 0;
while (i < values.size()) {
  sum += values[i];
```

(continues on next page)

[3] Any of the initialization, test, or update may be omitted, but the semicolons separating them must still be present. If the test is omitted, it is assumed to always be true.

(continued from previous page)

```
  ++i;
}
```

Depending on the computation, it may be more naturally expressed using one loop construct or the other. In this case, iterating over the elements of a vector arguably is more cleanly expressed with a for loop than a while loop.

In addition to the normal behavior of a loop, we further control execution through the loop with return, break, and continue statements. A return statement immediately exits the enclosing function. A *break statement* exits the (most inner) loop, proceeding past the loop:

```
// Print elements until the first 0
for (size_t i = 0; i < values.size(); ++i) {
  if (values[i] == 0) {
    break;
  }
  cout << values[i] << endl;
}
```

A *continue statement* skips the rest of the current loop iteration, but does not immediately exit the loop:

```
// Print the square root of non-negative numbers
for (size_t i = 0; i < v.size(); ++i) {
  if (v[i] < 0)) {
    continue;
  }
  cout << v[i] << endl;
}
```

Often, a loop can be restructured to avoid break or continue statements, but in other cases, a computation may be more easily expressed using them. For the second loop above, we can rewrite it as follows:

```
// Print the square root of non-negative numbers
for (size_t i = 0; i < v.size(); ++i) {
  if (v[i] >= 0)) {
    cout << v[i] << endl;
  }
}
```

For the first loop that uses a break statement, we can use logical operations instead:

```
// Print elements until the first 0
for (size_t i = 0; i < values.size() && values[i] != 0; ++i) {
  cout << values[i] << endl;
}
```

We can use either `&&` or `and` to specify a conjunction (an "and" of two conditions), either `||` or `or` for a disjunction (an "or" of two conditions), and either `!` or `not` to negate a truth value. Conjunction and disjunction are *short-circuiting*, meaning that the right-hand side will only be evaluated if necessary to compute the result. In our example here, `values[i] != 0` only gets evaluated when `i < values.size()` – good thing, because otherwise we could access an element past the end of the vector, producing the dreaded undefined behavior.

CHAPTER

FIVE

PROCEDURAL ABSTRACTION

Abstraction is the principle of separating *what* something is or does from *how* it does it. It is the primary tool for managing complexity in Computer Science and other fields. As a non-programming example, consider the task of making a sandwich. In order to make a sandwich, we don't need to know how bread is made, nor peanut butter nor jelly. All we need to know is how to put those ingredients together to construct a sandwich. Thus, we rely on the abstractions of bread, peanut butter, and jelly to make a sandwich. Similarly, the person who eats our sandwich doesn't need to know how it's made – all they need to know is that it is a delicious sandwich.

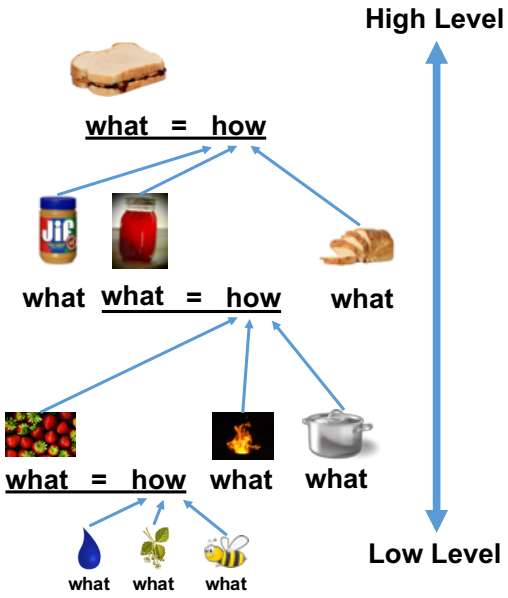

Figure 5.1: Abstraction layers for making a sandwich.

In a complex system, there are multiple layers of abstraction, each of which relies on the "what" knowledge of the lower layers without needing to know the "how." In the case of a sandwich, the top layer is the consumer of the sandwich, who only needs to know that it is a sandwich. The layer below is the maker of the sandwich, who needs to know what bread, peanut butter, and jelly are and how to combine them into a sandwich. They do not need to know how to make each of the ingredients. The jelly manufacturer needs to know how to combine fruit and other ingredients into a container and apply heat to make jelly. However, they do not need to know how to grow fruit. That's a farmer's job, who needs to know how to use plants, water, and pollinators in order to produce fruit.

Computer programs are similarly built using layers of abstraction. When it comes to computational procedures, *functions* are our mechanism for defining *procedural abstractions*. The user of a function need only know what the function does without caring about how the function accomplishes its task. Specifically, the user needs to know the *interface* of the function, including its actual code interface and the documentation of what the function does. The user does not

need to concern themselves with the *implementation* of the function, the actual details of how the function works.

5.1 Code Organization in C++

In general, we organize a program by decomposing it into independent *modules*, defined in separate files. In C++, a single module is further decomposed into a *header* file and a *source* file. In this course, we will use the `.hpp` extension for header files and the `.cpp` extension for source files[4]. The header contains the interface of the module, while the source file contains the actual implementation.

As an example, the following is a subset of `stats.hpp` from Project 1:

```
#include <vector>

//REQUIRES: v is not empty
//EFFECTS: returns the arithmetic mean of the numbers in v
double mean(std::vector<double> v);
```

Only a declaration of the `mean()` function appears in the header, along with its documentation. The actual definition goes in `stats.cpp`, the source file for the module:

```
#include "stats.hpp"
#include "p1_library.hpp"

using namespace std;

double mean(vector<double> v) {
  return sum(v) / count(v);
}
```

Other source files that use the functions in the stats module need only *include* the header file (`stats.hpp`) with the `#include` directive:

```
#include <iostream>
#include "stats.hpp"

using namespace std;

int main() {
  vector<double> data = { 1, 2, 3 };
  cout << mean(data) << endl;
}
```

A source file that uses the stats module only needs to know about the interface of the module. As long as the interface remains the same, the implementation of a module can change arbitrarily without affecting the behavior of a source file that uses it.

The `#include` directive actually pulls in the code from the target into the current file. So the end result is as if the declarations for the stats functions actually were written in this source file as well. We get access to those functions' declarations without having to manually repeat them.

Library modules such as `vector` and `iostream` are surrounded by angle brackets (e.g. `#include <vector>`) in a `#include` directive. Non-library headers that are located in the same directory are surrounded by double quotes (e.g.

[4] The extensions `.hxx` and `.h` are other common conventions for header files, and `.cxx` and `.cc` are also common for source files.

`#include "stats.hpp"`). For the purposes of this course, we never `#include` anything other than header files and standard libraries – we never `#include` a `.cpp` source file.

The `using namespace std;` directive allows us to refer to standard-library entities without prefixing them with `std::`. An alternative is to avoid the prefix for specific entities with individual `using` declarations:

```
#include <iostream>
#include "stats.hpp"

using std::cout;  // allow cout to be used without std::
using std::endl;  // allow endl to be used without std::

int main() {
  // must prefix vector with std::
  std::vector<double> data = { 1, 2, 3 };
  // can elide std:: with cout and endl
  cout << mean(data) << endl;
}
```

It is considered bad practice among C++ programmers to include a `using namespace std;` directive, or generally other `using` declarations, in a header file – this forces those directives and declarations on anyone who `#includes` the header. On the other hand, they are fine to place in a source file, since source files are generally not `#included`.

The overall file structure for Project 1 is shown in Figure 5.2, excluding testing modules.

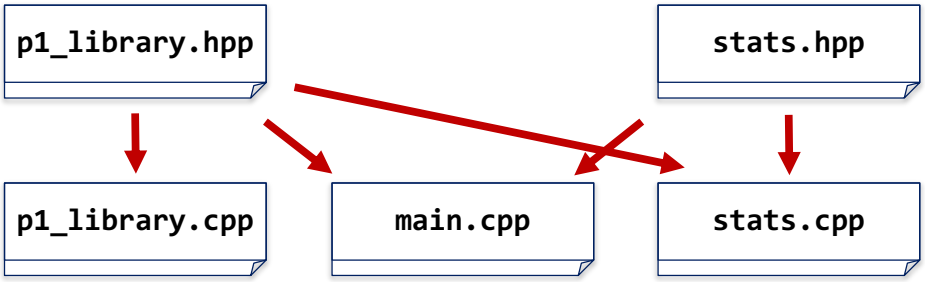

Figure 5.2: File structure for Project 1.

Arrows are shown between header files and the source files that `#include` them.

When compiling the project, only source files are passed to the compiler. The header files are folding into the source files through the `#include` directives:

```
$ g++ --std=c++17 -pedantic -g -Wall -Werror p1_library.cpp stats.cpp main.cpp -o main.
↪exe
```

Any number of source files can be passed to the compiler, but only one of those may contain a `main()` function.

5.2 The Compilation Process

Consider the compilation command above. The elements of the command are:

- `g++` is the C++ compiler we are invoking.
- The `--std=c++17` argument tells it to compile according to the C++17 language standard.
- The `-pedantic` argument tells the compiler to adhere strictly to the C++ standard. Without this flag, compilers often provide extensions or allow behavior that is not permitted by the standard.
- The `-g` argument tells the compiler to produce an executable that facilitates debugging.
- The `-Wall` argument asks the compiler to generate warnings about possible programming errors.
- The `-Werror` argument configures the compiler to treat warnings as errors, so that it does not compile code that has warnings.
- The arguments `-o main.exe` tell the compiler to produce the output file `main.exe`.
- The remaining three arguments are the source files for our program – `p1_library.cpp`, `stats.cpp`, and `main.cpp`.

For the source files, the compiler will compile each of them separately, producing temporary *object files*. It will then attempt to *link* the object files together into the output executable. The linking step can fail if:

- A function is declared and used in the source files, but no definition is found. For example, if the definition for `percentile()` is missing, a linker error such as the following results:

```
Undefined symbols for architecture x86_64:
  "percentile(std::__1::vector<double, std::__1::allocator<double> >, double)",␣
↪referenced from:
      _main in main-dc223c.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

- Multiple definitions of the same function are found. For example, if we try to compile and link multiple source files that define a `main()` function, we get a linker error like the following:

```
duplicate symbol _main in:
    /var/folders/gc/0lqwygqx381fmx9hhvj0373h0000gp/T/main-9eba7c.o
    /var/folders/gc/0lqwygqx381fmx9hhvj0373h0000gp/T/stats_tests-b74225.o
ld: 1 duplicate symbol for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Upon success, the result of the linking step is the final program executable, `main.exe` for the compilation command above.

5.3 Specification Comments (RMEs)

The interface of a function includes its *signature*, which consists of the function's name and parameter types. The return type of the function[5] is also part of its interface.

Another part of a function's interface is documentation about what it does, as in the following:

[5] Technically, the return type of a regular (i.e. non-template) function is not part of its signature as defined by the C++ standard (see [defns.signature] in the standard). It is, however, part of the function's interface.

```
//REQUIRES: v is not empty
//EFFECTS: returns the arithmetic mean of the numbers in v
double mean(std::vector<double> v);
```

This documentation describes the *what* of the function, so it is an integral part of what a user of the function needs to know.

The documentation format we use in this course is an *RME*, which consists of a *REQUIRES* clause, a *MODIFIES* clause, and an *EFFECTS* clause.

5.3.1 REQUIRES Clause

The *REQUIRES* clause lists what the function requires in order to accomplish its task. If the requirements are not met, then the function provides no guarantees – the behavior is *undefined*, and anything the function does (e.g. crashing your computer, stealing your credit-card info, etc.) is valid. Thus, a user should never call a function with arguments or in a state that violates the REQUIRES clause.

Within the function definition, the implementation is allowed to assume that the REQUIRES clause is met – again, a user should never call the function if they are violated. A good practice is to *assert* that the REQUIRES clause is met, if it is possible to do so:

```
#include <cassert>
#include "stats.hpp"
#include "p1_library.hpp"

using namespace std;

double mean(vector<double> v) {
  assert(!v.empty());
  return sum(v) / count(v);
}
```

In order to use `assert`, the `<cassert>` header must be included. Then a boolean expression can be passed to `assert`. If the expression evaluates to a true value, execution proceeds normally. However, if it evaluates to a false value, then the program crashes with a meaningful error message:

```
Assertion failed: (!v.empty()), function mean, file stats.cpp, line 8.
```

This is much more desirable than computing a wrong answer (or stealing your credit-card info!), as it tells the user they did something wrong and where the requirements were violated.

If a function doesn't have any requirements, the REQUIRES clause may be elided. Such a function is called *complete*, while one that has requirements is called *partial*.

5.3.2 MODIFIES Clause

The *MODIFIES* clause specifies the entities outside the function that might be modified by it. This includes pass-by-reference parameters, global variables (**not** in this course – only global constants are permitted), and input and output streams (e.g. `cout`, `cin`, a file stream, etc.):

```
//MODIFIES: v
//EFFECTS: sorts v in ascending order
void sort(std::vector<double> &v);
```

The MODIFIES clause only specifies what entities may be modified, leaving out any details about what those modifications actually might be. The latter is the job of the EFFECTS clause. Instead, the purpose of the MODIFIES clause is for the user to quickly tell what items might be modified.

If no non-local entities are modified, the MODIFIES clause may be elided.

5.3.3 EFFECTS Clause

The *EFFECTS* clause specifies what the function actually does. This includes details about what modifications are made, if any, as well as what the return value means, if there is one. All functions should have an EFFECTS clause – if a function does nothing, there is no point to writing the function.

The EFFECTS clause should generally only indicate *what* the function does without getting into implementation details (the *how*). It is part of the interface of a function, so it should not be affected if the implementation were to change.

5.4 Properties of Procedural Abstraction

As mentioned previously, the implementation of an abstraction should be able to change without affecting the way the abstraction is used. More formally, abstractions are *substitutable* – we should be able to swap out an implementation of an abstraction for a different one. As long as the interface remains the same, code that uses the abstraction will still work.

Abstractions should also be *local*, meaning that it should be possible to understand the implementation of one abstraction without knowing anything about the implementation of other abstractions. This implies that the implementation of one abstraction should not rely on implementation details of another – instead, the former should only work with the latter through the **interface** of the latter.

CHAPTER

SIX

TESTING

Testing is the primary mechanism we have for determining whether or not code is correct. Most programs are too complex to formally prove correct with reasonable effort. Instead, thorough testing provides a measure of confidence that the code behaves correctly.

Insufficient testing can lead to code that breaks when deployed, with potentially disastrous consequences. A canonical example is the Therac-25, a radiation-therapy machine from the 1980s. A combination of poor design and software bugs led to several cases where patients received massive radiation overdoses. Three patients died as a result. A more recent example of software bugs leading to catastrophic failure is Knight Capital Group, a financial-services firm that engaged in high-frequency trading. Lack of regression testing, poor software maintenance, and a mistake in deployment caused a new version of their software to execute millions of trades that bought shares at high prices and sold them at lower ones – all in a span of just 45 minutes. The end result was a pre-tax loss of $440 million, and the company was forced to sell itself to a competitor.

In this course, the stakes aren't quite as high as in the cases above. However, it is often the case that the difference between doing poorly on a project and doing well is how well you test your code. Testing is the mechanism for determining whether or not code is buggy. Once a bug has been detected, *debugging* is the process used to identify and fix the source of the bug.

There are two main categories of test cases:

- *Unit tests* test once piece of code at a time, often at the granularity of individual functions or small groups of functions. This helps to find bugs early as well as make them easier to debug, since a failed unit test identifies exactly which function is broken.
- *System tests* test an entire module or program as a whole. This can identify bugs that occur when integrating multiple units together – it's possible that two units appear to work individually, but one unit makes an incorrect assumption about the other that only manifests when they are tested together. System tests can also be closer to the real use case of a program, providing a measure of confidence that the program works as a whole.

In software development, it is important to maintain a set of *regression tests*, or tests that are run every time a code change is made. That way, a breaking change is identified immediately and can be fixed before the software is deployed. Regression tests generally include both unit and system tests.

Test-driven development is a common programming practice that integrates writing tests with developing the implementation of a program. Once the interface of a unit has been determined, tests can be written even before the implementation – that way, once the implementation is written, there are tests that can be run immediately. Furthermore, the process of writing tests can help inform the right strategy to use in implementation, reducing development time. The implementation in turn inspires new test cases, based on an understanding of what mistakes could have been made in the code. The result is an iterative process: writing tests for a unit, implementing the unit, writing more tests, fixing bugs, and so on until the code has been thoroughly vetted.

Only valid test cases should be written. Those that don't compile are useless, as well as those that result in undefined behavior (e.g. by violating the REQUIRES clause of a function). There are several types of valid test cases:

- Simple test cases are for the "average" case. For instance, in testing a `mode()` function in Project 1, an example of a simple test case is one that tests a vector that contains exactly one mode (e.g. `{ 1, 2, 3, 2 }`).
- *Edge* cases are those that test special cases in a unit's behavior. For instance, the `mode()` function requires that its argument vector be non-empty. Thus, the smallest vector that can be passed to `mode()` is one with a single element – this is an edge case. Another special case is when the input has two modes – in this case, the documentation of the function specifies that it returns the smaller one. So we should test that the function indeed behaves correctly in this case.
- *Stress* tests are intensive tests that ensure that a system will work under a heavy load. For example, an autograder for a 1000-person course should be able to withstand hundreds of students trying to submit right before the deadline. Thus, it is important to test the autograder with large numbers of concurrent requests. Stress tests will not be used in this course, since the projects we will focus on are not performance-critical.

Test cases should focus on cases that realistically could be erroneous. For example, a malicious programmer could insert a bug that only occurs for a specific, arbitrary input (e.g. if the number 42 is in the input). However, we generally can't test every possible input, so we have to concentrate on those cases where a reasonable programmer could make a mistake. (Other mechanisms, such as code reviews, are used to guard against malicious programmers.)

The *small scope hypothesis* states that thorough testing with "small" test cases is sufficient to catch most bugs in a system[6]. Thus, our test cases need not be large – in general, they should be small enough where we can compute the expected results by hand. Similarly, having more test cases is not necessarily better. Fewer test cases that are meaningfully different is more effective than having many, redundant test cases. As an example, the data sets `{ 1, 1, 2, 2, 2 }` and `{ 1, 1, 2, 2, 2, 2 }` are not meaningfully different for the `mode()` function – there is no good reason why it would behave differently in the two cases. On the other hand, the data set `{ 1, 1, 2, 2 }` is meaningfully different, since it contains two modes.

[6] See *Evaluating the "Small Scope Hypothesis"* by Andoni et al. and *On the Small-Scope Hypothesis for Testing Answer-Set Programs* by Oetsch et al. for empirical evaluations of the hypothesis.

CHAPTER

SEVEN

MACHINE MODEL I

A computer program consists of source code that determines what the program does. The program itself is run on a machine, and the program directs the machine on what computation should be preferred. In order to understand a program, it is important to have a *machine model* that helps us reason about how the source code relates to what happens at runtime.

As an example, consider the following C++ program:

```
int main() {
  int x = 3;
  double y = 4.1;
  int z = x;
  x = 5;
}
```

When the program runs, execution starts at `main()`. Each variable in the program is a name that refers to some *object* in memory, a region of memory that holds the data value for the variable. The variable's type determines how much memory is required, and how the data are represented – generally, data are stored in *bytes*, which themselves are sequences of eight *bits*, each of which is a binary digit that is zero or one. For instance, in typical C++ implementations, a variable of type `int` requires four bytes (32 bits) of storage, while one of type `double` requires eight bytes (64 bits).

It is often useful to think of memory as a large *array*, with data values stored at different *indices* into the array, as shown in Figure 7.2.

Here, the program is using memory index 6 for `x`, index 2 for `y`, and index 4 for `z`. (Later, we will see that a program uses a more systematic *method for locating the local variables of a function*.) The contents of memory illustrated above are after the initialization of `x` and `y` but before the initialization of `z`. The location for `x` has a representation of the `int` value 3, the location for `y` has a representation of the `double` value 4.1, and the location for `z` has some indeterminate value.

When the program proceeds to initialize `z`, it copies the value 3 from the memory for `x` into the memory for `z`, as demonstrated in Figure 7.3.

Finally, the assignment `x = 5` modifies the value of `x` to be 5, as Figure 7.4 illustrates.

In order to discuss the conceptual spaces for source code and runtime in more detail, we need some terminology. In source code, a *name* refers to some entity such as a variable, function, or type. As mentioned above, a *variable* is a name that refers to an *object* in memory. A name has a *scope*, which determines what region of code can use that name to refer to an entity. For example, the scope of `y` in the code above begins at the declaration of `y` and ends at the end of the function definition for `main()`. Attempting to use `y` outside this region will result in a compiler error. A *declaration* is what introduces a name into the program and begins its scope.

At runtime, an *object* is a piece of data in memory, and it is located at some *address* in memory (corresponding to the index in our basic machine model above). An object has a *lifetime* during which it is legal to use that object. More specifically, an object is created at some point in time, and at some later point in time it is destroyed. The *storage duration* of an object determines its lifetime. There are three options that we will see in C++:

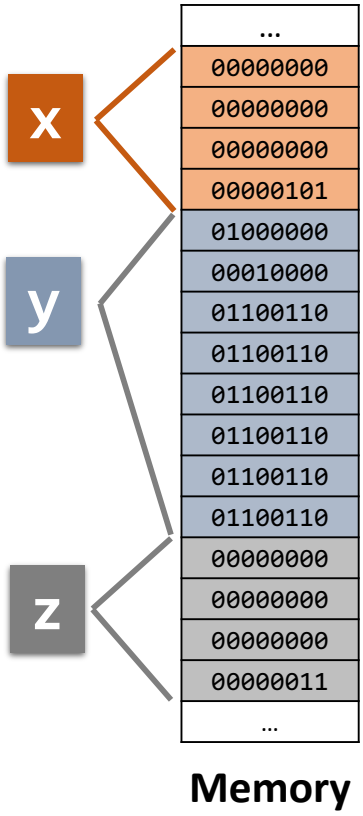

Figure 7.1: Variables correspond to objects in memory, each of which stores a data value represented in binary.

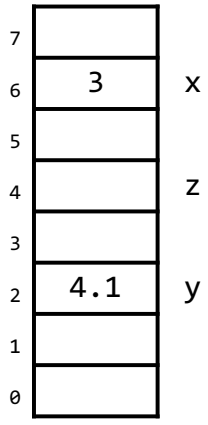

Figure 7.2: Simplified machine model, with memory as a linear (array) structure with slots for different objects.

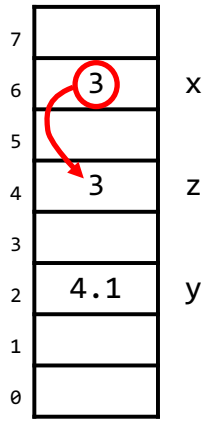

Figure 7.3: Initializing an object as a copy of another.

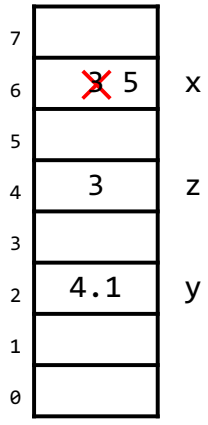

Figure 7.4: Modifying the value of an object.

- *static*: the lifetime is essentially the whole program
- *automatic* (also called *local*): the lifetime is tied to a particular scope, such as a block of code
- *dynamic*: the object is explicitly created and destroyed by the programmer

The former two durations are controlled by the compiler, while the latter is specified by the programmer. We will restrict ourselves to static and automatic storage duration until *later in the course*.

A variable is not the same thing as a memory object: a variable is a concept associated with source code, while an object is associated with the runtime. The same variable can refer to different objects at different times, such as a local variable in a function that is called more than once. An object that has dynamic storage duration is not associated with a variable at all.

An important consideration in the design of a language is the semantics of an initialization or assignment of the form `x = y`. Does this change which object `x` is referring to, or does it modify the value of the object that `x` is referring to? The first option is known as *reference semantics*, while the second is *value semantics*.

In C++, the default is value semantics. Consider the following program:

```
int x = 42;    // initialize value of x to 42
int y = 99;    // initialize value of y to 99
x = y;         // assign value of y to value of x
```

The assignment in the last line copies the value stored in the memory object associated with `y` into the memory object for `x`, as shown in Figure 7.5.

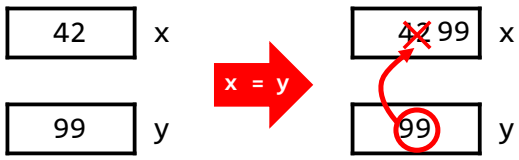

Figure 7.5: Assignment copies a value from the right-hand side into the left-hand-side object.

C++ supports reference semantics only when **initializing** a new variable. Placing an ampersand (&) to the left of the new variable name causes the name to be associated with an existing object. The following is an example with a local variable:

```
int x = 42;  // initialize value of x to 42
int z = 3;   // initialize value of z to 3
int &y = x;  // y and x are now names for the same object
x = 24;      // assigns 24 to object named x/y
y = z;       // Does NOT re-bind y to a different object
             // Value semantics used here.
```

The declaration `int &y = x;` introduces `y` as a new name for the object associated with `x`. Any subsequent modification to this object is reflected through both names, regardless of the name used to perform the modification. Figure 7.6 shows the effects of the assignments in the code above.

Since C++ only supports reference semantics in initialization, **the association between a variable and a memory object can never be broken**, except when the variable goes out of scope.

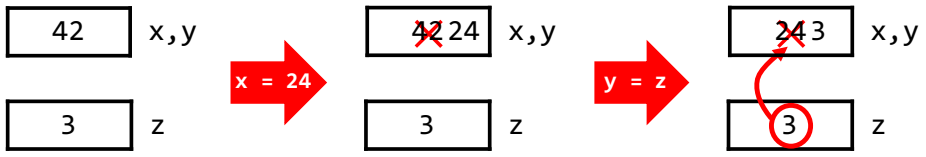

Figure 7.6: A reference is another name for an existing object.

7.1 Pointers

Recall that in C++, an *object* is a piece of data in memory, and that it is located at some *address* in memory. The compiler and runtime determine the location of an object when it is created; aside from deciding whether an object is in the global segment, on the stack, or in the *heap segment* (the segment used for dynamic memory), the programmer generally does not control the exact location where an object is placed[1]. Given the same program and the same inputs to that program, different systems will often end up placing the same objects at different memory locations. In fact, in many implementations, running the same program twice on the same system will result in different addresses for the objects.

Though the programmer does not have control over the address at which an object is located, the programmer does have the ability to query the address on an object once it has been created. In C++, the & (usually pronounced "address-of") operator can be applied to an object to determine its address:

```
int main() {
  int x = 3;
  double y = 5.5;
  cout << &x << endl; // sample output: 0x7ffee0659a2c
  cout << &y << endl; // sample output: 0x7ffee0659a20
}
```

Addresses are usually written in hexadecimal (base-16) notation, with a leading `0x` followed by digits in the range `0`-`9` and `a`-`f`, with `a` representing the value 10, `b` the value 11, and so on. Most modern machines use 64 bits for an address; since each digit in a hexadecimal number represents four bits ($2^4 = 16$ values), a 64-bit address requires up to 16 hexadecimal digits. The examples above use only 12 digits, implying that the leading four digits are zeros. Most examples in this text use fewer digits for conciseness.

In addition to being printed, addresses can also be stored in a category of objects called *pointers*[2]. A pointer variable can be declared by placing a * symbol to the left of the variable name in its declaration:

```
int x = 3;
int *ptr = &x;
cout << ptr << endlt; // sample output: 0x7ffee0659a2c
```

A pointer type consists of two elements:

- the type of the objects whose addresses the pointer can hold as a value. For example, an `int *` pointer can hold the address of an `int`, but not that of any other data type.
- the * symbol, which indicates that the type is a pointer

Each data type in C++ has a corresponding pointer type. For instance, `int *` is the pointer type corresponding to `int`,

[1] C++ has placement new, which allows a programmer to initialize a new object in a given memory location. However, even with placement new, the original memory must have been allocated by the programmer, and the programmer does not control the exact address produced by that allocation.

[2] The terms *address* and *pointer* are often used interchangeably. Technically, a pointer is an object that holds an address as its value, while the address is the value itself.

`double *` is the pointer type corresponding to `double`, and `double **` is the pointer type corresponding to `double *`.

A pointer object can be *dereferenced* to obtain the object whose address the pointer is holding by applying the * (usually pronounced "star" or "dereference") operator:

```
int x = 3;
int y = 4;
int *ptr = &x;
cout << *ptr << endl; // prints 3
ptr = &y;
cout << *ptr << endl; // prints 4
```

We often say that a pointer "points to" an object, and that the * operator "follows" the pointer to the object it is pointing at. In keeping with this terminology, a pointer is often pictured as an arrow from the pointer to the object it is pointing at, as shown in Figure 7.7.

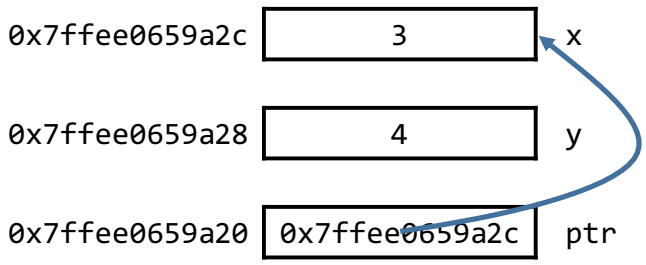

Figure 7.7: An arrow indicates the object whose address a pointer holds.

Usually, we are not concerned with actual address values – they are implementation-dependent and can vary between program runs. Instead, we only concern ourselves with which object each pointer is referring to. Thus, we often draw just an arrow to illustrate which object a pointer is pointing to, as in Figure 7.8, without the actual address value.

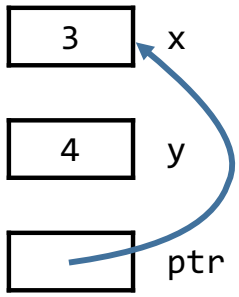

Figure 7.8: Pointers are often illustrated by just an arrow, without the actual address value.

The following is another example of working with a pointer:

```
int main() {
  int foo = 1;
  int *bar = &foo;
  foo += 1;
  *bar += 1;

  cout << foo << endl;   // prints 3
```

(continues on next page)

(continued from previous page)

```
  cout << bar << endl;   // prints address of foo (e.g. 0x1000)
  cout << *bar << endl;  // prints 3
}
```

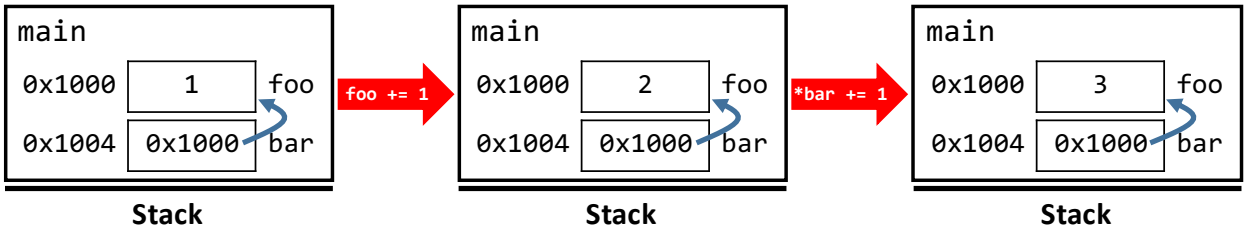

Figure 7.9: Example of modifying an object through a pointer.

The code initializes the pointer `bar` with the address of `foo`. It proceeds to increment `foo` directly, then dereferences `bar` to increment the object it is pointing at. Since it is pointing at the object associated with `foo`, the result is that `foo` has value 3. The state of memory at each point is shown in Figure 7.9.

7.1.1 Pointers and References

The * and & symbols mean different things when they are used as part of a type and when they are used in an expression:

- When used in a type, * means that the type is a pointer type, while & means the type is a reference type.
- When used as a unary prefix operator in an expression, * is used to dereference a pointer, while & is used to obtain the address of an object.
- When used as a binary infix operator, * is multiplication, while & is a *bitwise and* operation (which is outside the scope of this course).

The following illustrates some examples:

```
int x = 3;
int *ptr = &x;
int &ref = *ptr;
```

In the second line, * is used in a type, so `ptr` is a pointer to an `int`. The initialization is an expression, so the & obtains the address of the object corresponding to `x`. In the third line, & is used as part of the type, so `ref` is a reference variable. Its initialization is an expression, so the * dereferences `ptr` to get to the object associated with `x`. The result is that `ref` is an alias for the same object as `x`.

Pointers allow us to work indirectly with objects. The following is an implementation of a swap function that uses pointers to refer to the objects whose values to swap:

```
void swap_pointed(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main() {
  int a = 3;
  int b = 5;
```

(continues on next page)

(continued from previous page)

```
  swap_pointed(&a, &b);
  cout << a << endl;      // prints 5
  cout << b << endl;      // prints 3
}
```

Figure 7.10 demonstrates the execution of this code.

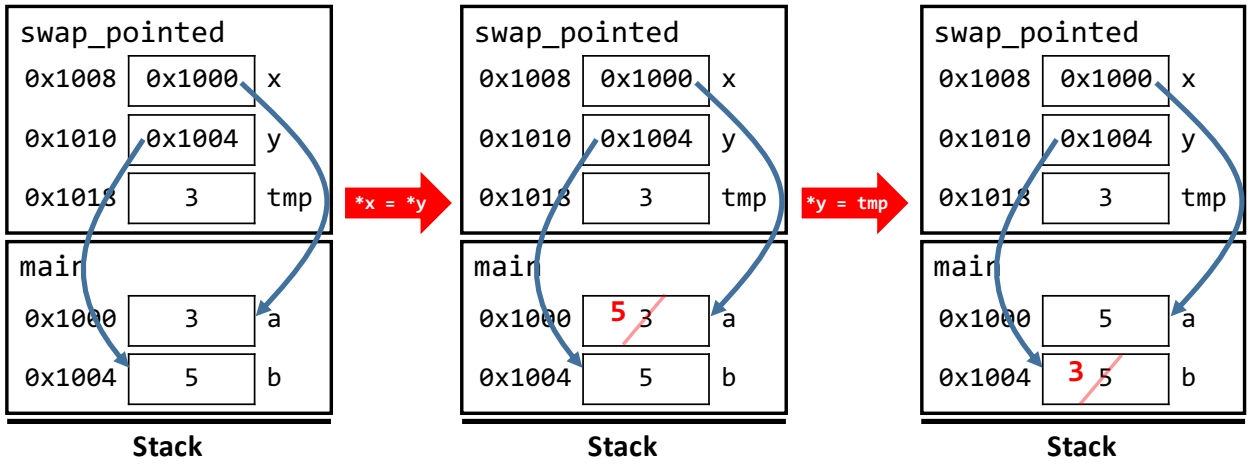

Figure 7.10: Pointers enable redirection, allowing a function to modify objects in a different scope.

References can also be used to indirectly refer to other objects. However, a pointer actually holds the address of another object, while a reference just acts as another name for an existing object. A reference must be initialized to refer to another object, and the association between the reference and its object cannot be broken. On the other hand, a pointer need not be initialized to point to an object, and the address value it holds can be later changed to be the address of a different object. A pointer can be null or have an undefined value, so it must be used carefully to avoid undefined behavior. On the other hand, while it is possible to construct an invalid reference, it usually does not occur in the normal course of programming in C++.

Both pointers and references allow objects to be used across scopes. They both enable *subtype polymorphism*, a concept we will see later in the course. Both can also be used to refer to objects in *dynamic memory*, as we will also see later.

Pointers are strongly connected to arrays. Indexing into an array actually works through *pointer arithmetic*, as we will see *later*.

CHAPTER

EIGHT

MACHINE MODEL II

8.1 Pointer Errors

A pointer is an *atomic* type, since it cannot be subdivided into smaller objects. As with other atomic types, a variable of pointer type that isn't explicitly initialized is *default initialized* to an undefined value:

```
int x = 3;
int *ptr;  // undefined value
ptr = &x;  // well-defined value -- the address of x
```

Dereferencing a default-initialized pointer results in undefined behavior – the program may crash, or it may not; reading the dereferenced value can result in zero, or some other random value. Undefined behavior is notoriously difficult to debug, as the behavior can be different on different machines or in different runs of the program on the same machine. Tools like Valgrind or an address sanitizer can help detect undefined behavior.

The following is another example of default initializing pointers:

```
int main() {
  int *x;
  int *y;
  int a = -1;
  x = &a;
  cout << *x << endl;  // prints -1
  *x = 42;
  cout << a << endl;   // prints 42
  *y = 13;             // UNDEFINED BEHAVIOR
}
```

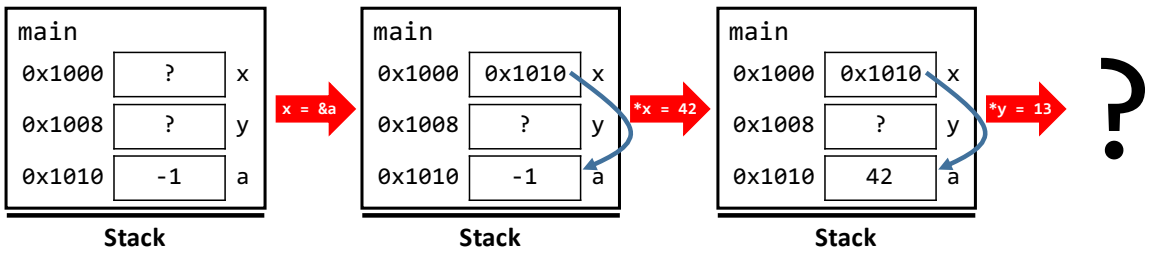

Figure 8.1: Dereferencing an uninitialized pointer results in undefined behavior.

Figure 8.1 illustrates the execution of this code. Both `x` and `y` are default initialized to indeterminate values. The code

proceeds to assign the address of `a` into `x`, so `x` now has a well-defined value, and it is valid to dereference `x`. Assigning to the object it is pointing at changes the value of that object, the one named by `a`. Dereferencing `y`, on the other hand, produces undefined behavior, since its value is a junk address. The program may crash, or it may not. It may overwrite a memory location that is in use by something else, or some location that is not in use. With undefined behavior, anything is possible.

A *null pointer* is a pointer that holds an address value of `0x0`. No object can be located at that address, making the null value a useful value for a pointer that does not point to a valid object. In C++, the `nullptr` literal represents the null value, and it can be used with any pointer type:

```
int *ptr1 = nullptr;
double *ptr2 = nullptr;
cout << (ptr1 == nullptr) << endl;  // prints 1 (i.e. true)
```

Dereferencing a null pointer also results in undefined behavior. However, in most implementations, doing so will crash the program, which is generally easier to debug than other forms of undefined behavior.

A null pointer is sometimes used to indicate the lack of a value, as in the following:

```
// EFFECTS: Returns a pointer to the first string in vec of the given
//          length, or a null pointer if no such string is in vec.
string * find_by_length(vector<string> &vec, int length) {
  for (size_t i = 0; i < vec.size(); ++i) {
    if (vec[i].size() == length) {
      return &vec[i];
    }
  }
  return nullptr; // no string of given length
}

int main() {
  vector<string> v = { "hello", "world", "from",
                       "EECS", "280" };
  string *found = find_by_length(v, 3);
  if (found) { // null pointer has false value
    cout << "found string: " << *found << endl;
  } else {
    cout << "no such string" << endl;
  }
}
```

In this example, the `find_by_length()` function returns a pointer to a string of the given length. However, if no such string exists, the function returns a null pointer to indicate this. The caller must check whether the return value is null before attempting to dereference it.

Since a pointer is an object in its own right, it can live past the lifetime of the object it is pointing at. The following is an example:

```
int * get_address(int x) {
  return &x;
}

void print(int val) {
  cout << val << endl;
}
```

(continues on next page)

(continued from previous page)

```
int main() {
  int a = 3;
  int *ptr = get_address(a);
  print(42);
  cout << *ptr << endl;       // UNDEFINED BEHAVIOR
}
```

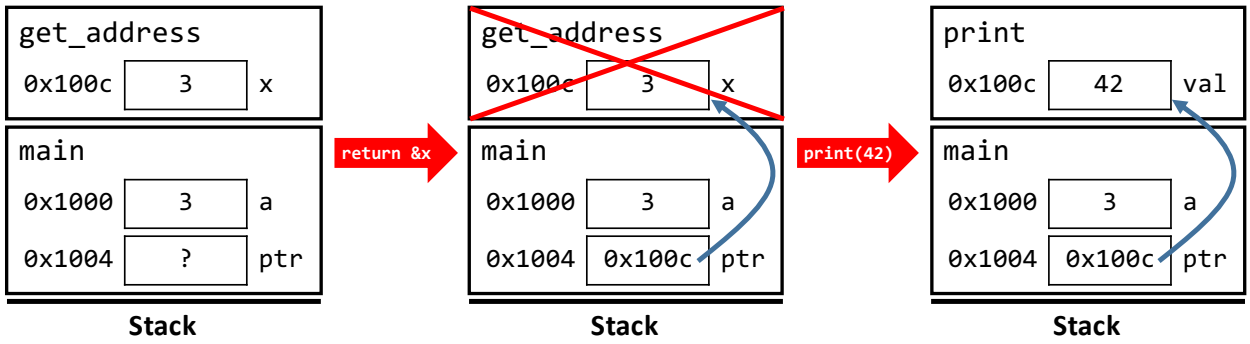

Figure 8.2: A pointer may refer to a dead object, in which case dereferencing it produces undefined behavior.

In this code, the parameter of the `get_address()` function is passed by value. So the `x` parameter is a new object in the activation record for `get_address()`, and it is initialized as a copy of `a`. The function returns the address of `x`, and that value is placed in the `ptr` object in `main()`. However, `x` dies along with the activation record of `get_address()`, so `ptr` is now pointing at a dead object. The code then calls `print()`, and it so happens that its activation record is placed in the same location previously used by `get_address()`, as shown in Figure 8.2. At this point, `ptr` happens to point at the `val` object, whose value is 42. When the `print()` function returns, its activation record is also reclaimed. Proceeding to dereference `ptr` produces undefined behavior. It so happens in this implementation that 42 is printed, but other implementations may have different behavior.

We can fix the code above by passing the parameter to `get_address()` by reference:

```
int * get_address(int &x) {
  return &x;
}

void print(int val) {
  cout << val << endl;
}

int main() {
  int a = 3;
  int *ptr = get_address(a);
  print(42);
  cout << *ptr << endl;       // prints 3
}
```

Now `x` aliases the object `a` in `main()`, as shown in Figure 8.3. Thus, `get_address()` returns the address of `a`, which is still alive when `*ptr` is printed.

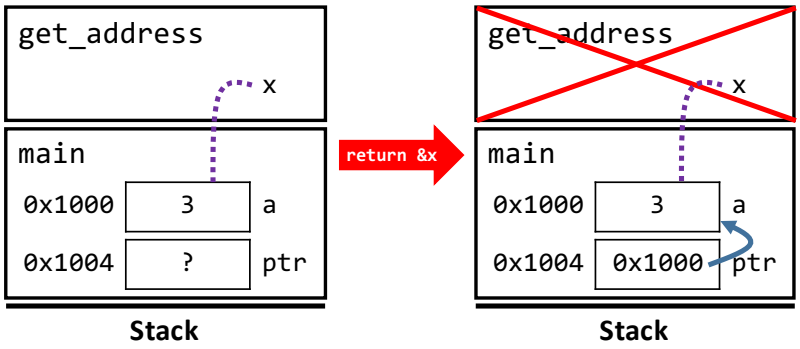

Figure 8.3: Example of taking the address of a reference parameter.

8.2 Function Calls and the Call Stack

Previously, we saw a basic machine model in which the program places each object at a different location in memory. We now examine a more structured model, stack-based memory management, that is used by many language implementations.

In most implementations, the data for a function call are collectively stored within an *activation record*, which contains space for each of the function's parameters and local variables, temporary objects, the return address, and other items that are needed by the function. In this course, we will generally only consider the parameters and local variables in an activation record, ignoring the other pieces of data there.

In stack-based memory management, activation records are stored in a data structure called a *stack*. A stack works just like a stack of pancakes: when a new pancake is made, it is placed on top of the stack, and when a pancake is removed from the stack, it is the top pancake that is taken off. Thus, the last pancake to be made is the first to be eaten, resulting in *last-in, first-out (LIFO)* behavior. Activation records are similarly stored: when an activation record is created, it is placed on top of the stack, and the first activation record to be destroyed is the last one that was created. This gives rise to an equivalent term *stack frame* for an activation record.

As an example, consider the following program:

```
void bar() {
}

void foo() {
  bar();
}

int main() {
  foo();
}
```

When the program is run, the `main()` function is called, so an activation record is created and added to the top of the stack. Then `main()` calls `foo()`, which places an activation record for `foo()` on the top of the stack. Then `bar()` is called, so its activation record is put on the stack. When `bar()` returns, its activation record is removed from the stack. Then `foo()` completes, removing its activation record. Finally, the activation record for `main()` is destroyed when the function returns. Figure 8.4 shows the state of the stack after each call and return.

In many implementations, the stack is actually stored upside down in memory, so that it grows downward rather than upward as shown in Figure 8.5. However, it still has the same LIFO behavior as a right-side-up stack.

As a more complex example, consider the following program:

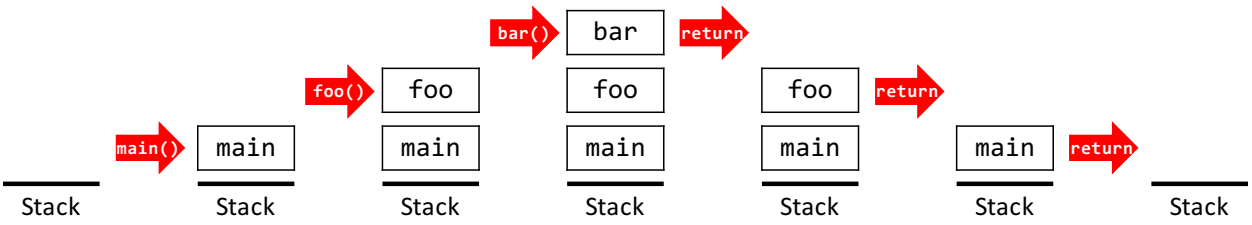

Figure 8.4: A stack that stores activation records.

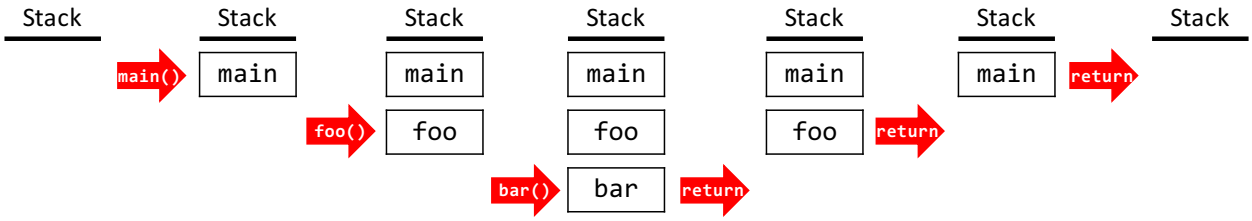

Figure 8.5: A stack that grows downward rather than upward.

```
int plus_one(int x) {
  return x + 1;
}

int plus_two(int x) {
  return plus_one(x + 1);
}

int main() {
  int result = 0;
  result = plus_one(0);
  result = plus_two(result);
  cout << result;             // prints 3
}
```

At program startup, the `main()` function is called, creating an activation record that holds the single local variable `result`. The declaration of `result` initializes its value to 0, and the program proceeds to call `plus_one(0)`. This creates an activation record for `plus_one()` that holds the parameter `x`. The program initializes the value of `x` to the argument value 0 and runs the body of `plus_one()`. The body computes the value of `x + 1` by obtaining the value of `x` and adding 1 to it, resulting in a value of 1, which the function then returns. The return value replaces the original call to `plus_one(0)`, and the activation record for `plus_one` is discarded before `main()` proceeds. The code then assigns the return value of 1 to `result`. Figure 8.6 illustrates the activation records up to this point.

The program then proceeds to call `plus_two(result)`. First, `result` is evaluated to obtain the value 1. Then an activation record is created for `plus_two()`, with space for its parameter `x`. Observe that this activation record is located in memory where the previous activation record for `plus_one()` was – the latter is no longer in use, so the memory can be reused. After the new activation record is created, the parameter `x` is initialized with the argument value 1. Then the program runs the body of `plus_two()`.

The body of `plus_two()` in turn calls `plus_one(x + 1)`. This evaluates `x + 1` to obtain the value 2, creates an activation record for `plus_one()`, initializes the value of `x` in the new activation record to be 2, and then runs the body of `plus_one()`. The state of memory at this point is shown in Figure 8.7.

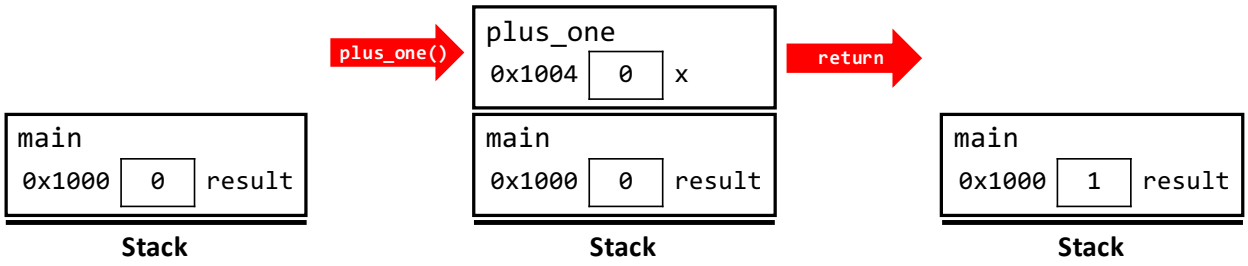

Figure 8.6: Activation record for `plus_one()`.

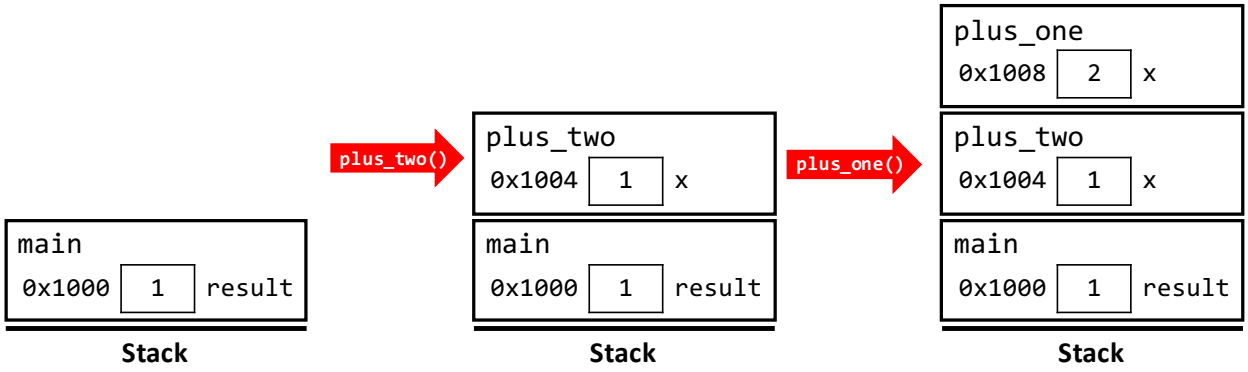

Figure 8.7: State of stack in second call to `plus_one()`.

Observe that the new activation record for `plus_one()` is distinct from the previous one – each invocation of a function gets its own activation record. In addition, there are now two variables `x` in the program. Within the scope of `plus_one()`, `x` refers to the object located in the activation record for `plus_one()`, and its value is 2. Within `plus_two()`, `x` refers to the object in the activation record for `plus_two()`, and its value is 1.

The invocation of `plus_one()` computes 3 as its return value, so that value replaces the call to `plus_one()`, and the activation record for `plus_one()` is discarded. `plus_two()` returns that same value 3, so the value 3 replaces the call to `plus_two()` in `main()`, and the activation record for `plus_two()` is discarded. Then `main()` proceeds to assign the value 3 to `result` and print it out. Finally, when `main()` returns, its activation record too is discarded.

8.2.1 Function-Call Process

To summarize, the following steps occur in a function call:

1. For pass-by-value parameters, the argument expressions are evaluated to obtain their values.

   For a pass-by-reference parameter, the corresponding argument expression is evaluated to obtain an object[1] rather than its value.

   The order in which arguments are evaluated is unspecified in C++.

2. A new and unique activation record is created for the call, with space for the function's parameters, local variables, and metadata. The activation record is pushed onto the stack.

3. The parameters are passed, using the corresponding arguments to initialize the parameters. For a pass-by-value parameter, the corresponding argument value is copied into the parameter. For a pass-by-reference parameter,

[1] C++ allows references to `const` to bind to values (i.e. rvalues in programming-language terms) rather than objects (lvalues). So a reference of type `const int &` can bind to just the value 3, as in `const int &ref = 3;`.

the parameter is initialized as an alias of the argument object.

4. The body of the called function is run. This transfer of control is often called *active flow*, since the code actively tells the computer which function to run.
5. When the called function returns, if the function returns a value, that value replaces the function call in the caller.
6. The activation record for the called function is destroyed. In simple cases, implementations will generally just leave in memory what is already there, simply marking the memory as no longer in use.
7. Execution proceeds from the point of the function call in the caller. This transfer of control is often called *passive flow*, since the code does not explicitly tell the computer which function to run.

The following program is an example of pass by reference:

```
void swap(int &x, int &y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main() {
  int a = 3;
  int b = 7;
  cout << a << ", " << b << endl;  // prints 3, 7
  swap(a, b);
  cout << a << ", " << b << endl;  // prints 7, 3
}
```

The program starts by creating an activation record for `main()`, with space for the local variables `a` and `b`. It initializes `a` to 3 and `b` to 7 and then prints out their values. The program then calls `swap(a, b)`, which evaluates the expressions `a` and `b` to obtain their objects, creates an activation record for `swap()`, and initializes the parameters from the argument objects. Since the two parameters are references, the activation record does not contain user-accessible memory for the parameters. (The metadata for the function, however, may include information about the parameters.) The activation record does contain explicit space for the local variable `tmp`, since it is not a reference. Figure 8.8 illustrates the activation record.

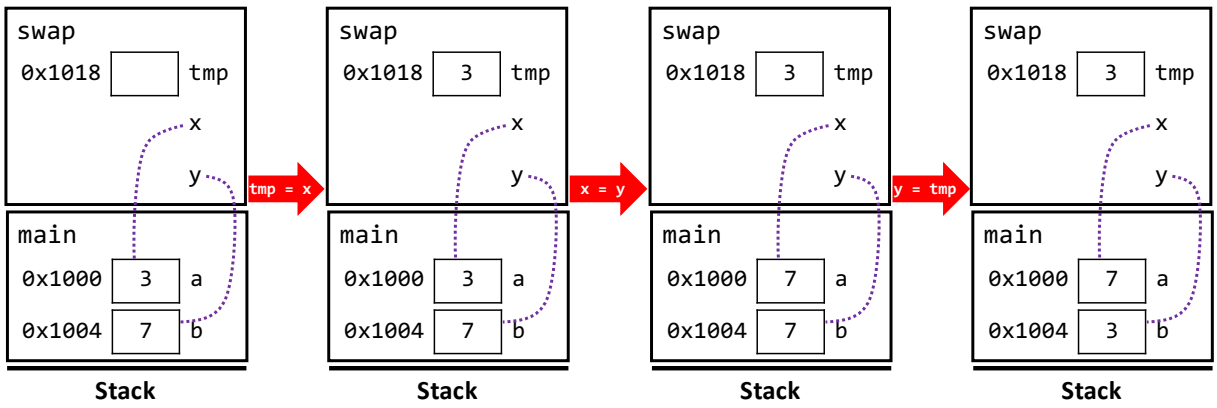

Figure 8.8: Activation record for a function that uses pass by reference.

Here, the reference parameters are depicted with a dotted line between the names and the objects they reference.

The program proceeds to run the body of `swap()`. First, `tmp` is initialized to the value of `x` – since `x` refers to the same object as `a` in `main()`, the value 3 of that object is copied into the memory for `tmp`. Then the assignment `x = y` copies

the value 7 from the object associated with `y` (`b` in `main()`) to the object associated with `x` (`a` in `main()`). Finally, the assignment `y = tmp` copies the value 3 from `tmp` into the object associated with `y` (`b` in `main()`). When `swap()` returns, its activation record is destroyed, and execution continues in `main()`. The latter prints out the new values of 7 and 3 for `a` and `b`, respectively.

A function can also return an object by reference, though we need to ensure that the *lifetime* of the object extends beyond the function invocation. The following is an example:

```
int & larger(int &x, int &y) {
  if (y > x) {
    return y;
  }
  return x;
}

int main() {
  int a = 3;
  int b = 7;
  cout << a << ", " << b << endl;  // prints 3, 7
  larger(a, b) = -1;
  cout << a << ", " << b << endl;  // prints 3, -1
}
```

The call `larger(a, b)` returns the object `a` or `b` whose value is larger, and we then assign the new value -1 to this object. If `larger()` were to return by value instead, attempting to assign to the resulting value would result in a compiler error.

Returning by reference is most common in data structures, where we wish to access an element by reference. For instance, this is how the indexing (or *subscript*) operator works on a vector:

```
std::vector<int> data = /* ... */;
data[0] = -1;
```

This operator is implemented using *operator overloading*, which we will discuss later.

Part II

Data Abstraction

CHAPTER

NINE

THE CONST KEYWORD

The `const` keyword is a *type qualifier* in C++ that allows us to prevent an object from being modified. As an example, consider the following erroneous definition of `strcpy()`:

```
void strcpy(char *dst, const char *src) {
  while (*src != '\0') {
    *src = *dst;
    ++src;
    ++dst;
  }
  *src = *dst;
}
```

In this definition, the assignments are backwards – the code is attempting to modify the source string rather than the destination. However, because the `src` parameter was declared with type `const char *`, the compiler will detect this error:

```
$ g++ --std=c++17 strcpy.cpp
strcpy.cpp:3:10: error: read-only variable is not assignable
    *src = *dst;
    ~~~~ ^
strcpy.cpp:7:8: error: read-only variable is not assignable
  *src = *dst;
  ~~~~ ^
2 errors generated.
```

A variable declared as `const` can be initialized, but its value cannot be later modified through an assignment. For a simple type, the `const` keyword may appear on either side of the type:

```
const int x = 3;  // initialization is OK
int const y = 4;  // const can go on the right as well
x = y;            // ERROR: attempt to assign to const object
```

Only types with values may be declared as `const`. The following types do not have values:

- References; they alias objects that may have values but do not have values of their own.
- Arrays; we will see arrays later in the course.
- Functions; we will see function types later in the course.

9.1 References and const

We saw previously that the binding between a reference and the object it aliases is established at initialization, and it cannot be broken as long as the reference exists. Thus, it is not meaningful for a reference itself to be const.

While a reference does not have a value and thus cannot be declared as `const` itself, it can refer to an object with `const` type. The `const` keyword can appear to the left of the `&` in a reference type, as in `const int &` – this is read "inside out" as "reference to a constant int". The object the reference is aliasing is not modifiable through the reference:

```
int x = 3;
const int &ref1  = x;  // reference to const int
int const &ref2  = x;  // const can go on the right of int as well
ref1 = 4;              // ERROR -- attempt to assign to const object
```

However, the original object is still modifiable if there is a non-const way to refer to it:

```
int x = 3;
int &ref1  = x;
const int &ref2 = x;
x = 4;                 // OK -- x not declared as const
ref1 = 5;              // OK -- ref1 a reference to a non-const int
ref2 = -6;             // ERROR -- ref2 a reference to a const int
```

The example above has three names that refer to the same object. However, the object is only modifiable through the names that do not include the `const` keyword.

In the two previous examples, we have created reference-to-const aliases for a non-const object. We will see shortly that non-const to const conversions are allowed, but the other direction is prohibited.

9.2 Pointers and const

Pointers do have a value, so they can be declared as `const`. To do so, the `const` keyword is placed to the right of the `*`:

```
int x = 3;
int y = 4;
int * const ptr = &x;
*ptr = -1;             // OK -- ptr is pointing to a non-const int
ptr = &y;              // ERROR -- attempt to assign to const object
```

Reading the declaration of `ptr` inside out, we get "`ptr` is a constant pointer to an int." Thus, we cannot modify the value of `ptr` itself. However, since the type that `ptr` is pointing to (what is to the left of the `*`) is not const, we can modify the object that `ptr` is pointing to.

Similar to a reference, we can declare that the object that a pointer is pointing to is const by placing the `const` keyword to the left of the `*`:

```
int x = 3;
int y = 4;
const int * ptr = &x;  // equivalent: int const *ptr = &x;
ptr = &y ;             // OK -- ptr is not const, so we can change its value
*ptr = -1;             // ERROR -- attempt to assign to const object
```

Finally, we can declare that both the pointer itself and the object it is pointing to are const:

```
int x = 3;
int y = 4;
const int * const ptr = &x;
ptr = &y ;              // ERROR -- attempt to assign to const object
*ptr = -1;              // ERROR -- attempt to assign to const object
```

9.3 const Conversions

We've seen examples above where we've constructed references-to-const and pointers-to-const objects from objects that were not declared as `const`. The general rule for converting between const and non-const is that the conversion must not enable modifications that are prohibited without the conversion.

A const object can appear on the right-hand side of an assignment, since an assignment has value semantics – it copies the value from the right-hand side to the left-hand side object. Thus, it does not permit the const object to be modified:

```
const int x = 3;
int y = x;          // OK -- copies value from x; does not enable x to be modified
```

On the other hand, if we initialize a reference-to-non-const with a const object, we would enable the object to be modified through the reference. Thus, such a conversion is prohibited:

```
const int x = 3;
int &ref = x;       // ERROR -- enables const object to be modified through ref
ref = 4;            // would modify the x object if the above were allowed
```

The same goes for placing the address of a const object in a pointer-to-non-const:

```
const int x = 3;
int *ptr = &x;      // ERROR -- enables const object to be modified through ptr
*ptr = 4;           // would modify the x object if the above were allowed
```

The other direction is allowed, however: creating a reference-to-const or pointer-to-const from a non-const object does not enable any new modifications:

```
int x = 3;
int const &ref = x;  // OK -- does not permit const object to be modified
const int *ptr = &x; // OK -- does not permit const object to be modified
```

The compiler only reasons about each conversion in isolation. This means that it does not allow a conversion from const to non-const even if we have some other means of modifying the underlying object:

```
int x = 3;
int *ptr1 = &x;          // OK -- does not permit const object to be modified
const int *ptr2 = ptr1;  // OK -- does not permit const object to be modified
ptr1 = ptr2;             // ERROR -- compiler sees that ptr2 is pointing to a
                         //          a const object, but ptr1 is not a pointer
                         //          to const
```

In the example above, even though `ptr2` was originally initialized from `ptr1`, we cannot later assign the value of `ptr2` to `ptr1`, since it would be converting a pointer-to-const to a pointer-to-non-const. This is actually useful for us as programmers: if we pass a pointer to another function, and the function guarantees that it won't modify the pointed-to

object by declaring its parameter as pointer-to-const, then we would be unhappy if the function could convert it back to a pointer-to-non-const and modify the pointed-to object.[1]

[1] C++ has a `const_cast` that enables `const` to be removed or added. However, it is only used in exceptional cases, none of which are relevant to this course.

CHAPTER

TEN

STRUCTS

C++ has several different categories of objects:

- *Atomic* types are built into the language, and these types are atomic because their objects cannot be subdivided into smaller objects. Atomic types are also known as *primitive* types. Examples of atomic types include basic numeric types such as `int`, `double`, `bool`, and `char`, as well as pointer types (e.g. `int *`, `string *`).
- *Arrays* are contiguous sequences of objects of the same type. They are therefore composed of *homogeneous* subobjects. We will discuss *arrays* later in the course.
- *Class-type* objects are objects composed of member subobjects, each which may be of a different type. Class-type objects are thus *heterogeneous*, and they are also often called *compound objects*.

In C++, a class type is introduced through the `struct` or `class` keyword. We will use the two keywords for different conventions when introducing our own data types. Later, we will see that the actual distinction between *how C++ treats the two keywords* is minimal.

In order to introduce a new data type, we use the `struct` keyword, followed by the name of the type we are introducing, followed by a body with *member declarations*:

```
struct Person {
  string name;
  int age;
  bool is_ninja;
};
```

Here, we are introducing a `Person` type. The body contains declarations for three *member variables*, each with their own type and name. The semicolon after the struct definition is mandatory, unlike in some other languages.

After the struct definition, we can create objects that have `Person` type. The following program creates two local variables of type `Person`:

```
int main() {
  Person elise;
  Person tali;
}
```

Each `Person` object has its own subobjects `name`, `age`, and `is_ninja` located within the memory for the `Person` object, as shown in Figure 10.1.

Since we did not explicitly initialize the `Person` objects, they are default initialized by in turn default initializing their member subobjects. The `age` and `is_ninja` members are of atomic type, so they are default initialized to indeterminate values. The `name` member is of class type, and it is default initialized to an empty string. In the future, we will see that *class types can specify how they are initialized*.

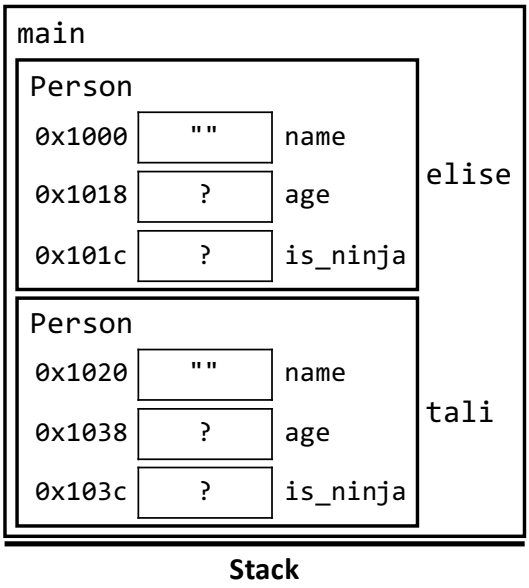

Figure 10.1: Memory for two objects of `Person` type.

We can explicitly initialize a `Person` object with an initializer list, similar to how we can initialize the elements of a vector:

```
Person elise = { "Elise", 22, true };
```

This initializes the struct member-by-member from the initializer list: the `name` member is initialized with `"Elise"`, the `age` member is initialized with `22`, and the `is_ninja` member is initialized with `true`. If fewer initializers are provided than members, the remaining members are implicitly initialized (with zeros for atomic types).

We can also copy structs when initializing a new struct or assigning to an existing one:

```
Person tali = elise;
```

By default, copying a struct copies the members one by one.[2] The result in memory is illustrated in Figure 10.2.

We can access individual members of a struct with the dot (`.`) operator. The struct object goes on the left-hand side, and the member name on the right:

```
tali.name = "Tali";
```

Here, we have assigned the string `"Tali"` to the `name` member of the `tali` object. Figure 10.3 shows the result in memory.

As the figure shows, `tali` and `elise` each have their own `name` members, so that modifying one does not affect the other.

We can use an initializer list for a struct in contexts other than initialization. The following uses an initializer list in an assignment (which is different from an initialization, since the target object already exists), as well as an argument to a function:

```
void Person_print_name(Person person) {
  cout << person.name << endl;
```

(continues on next page)

[2] We will see in the future that we can customize how class types are copied by overloading the *copy constructor* and *assignment operator*.

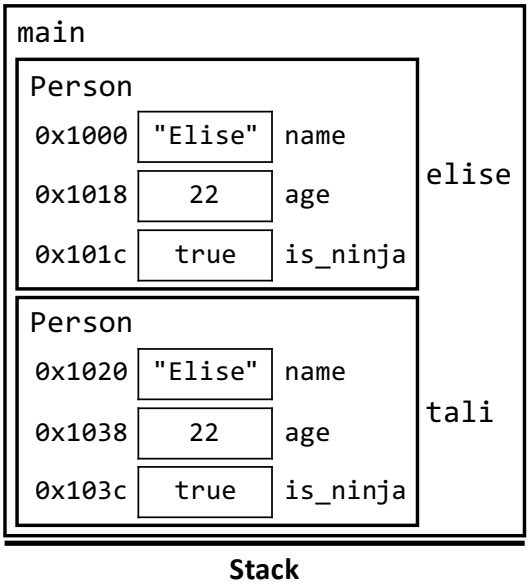

Figure 10.2: By default, copying a class-type object copies each of the member variables.

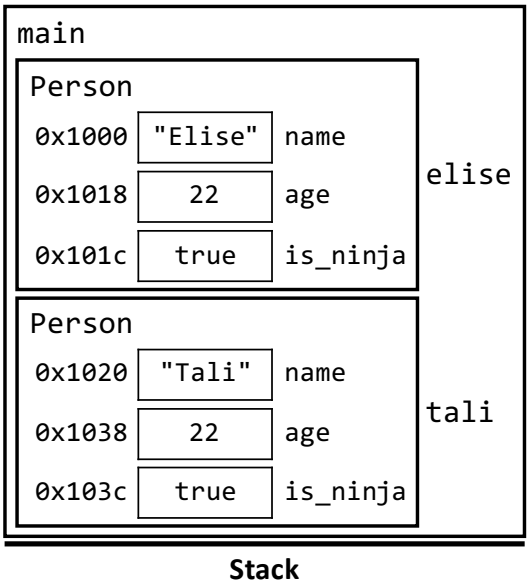

Figure 10.3: The result of modifying an individual member variable.

(continued from previous page)

```
}

int main() {
  Person tali;
  tali = { "Tali", 21, true };                // in an assignment
  Person_print_name({ "Elise", 22, true });  // as argument to function
}
```

When passing a struct to a function, we have our usual default of value semantics, meaning that a copy is made. For instance, the following erroneous definition of `Person_birthday()` does not modify the argument object, since it receives a copy of the argument:

```
// MODIFIES: person
// EFFECTS: Increases the person's age by one. If they are now older
//          than 70, they are no longer a ninja.
void Person_birthday(Person person) {
  ++person.age;
  if (person.age > 70) {
    person.is_ninja = false;
  }
}
```

Figure 10.4 illustrates what happens when we call `Person_birthday()` on a `Person` object:

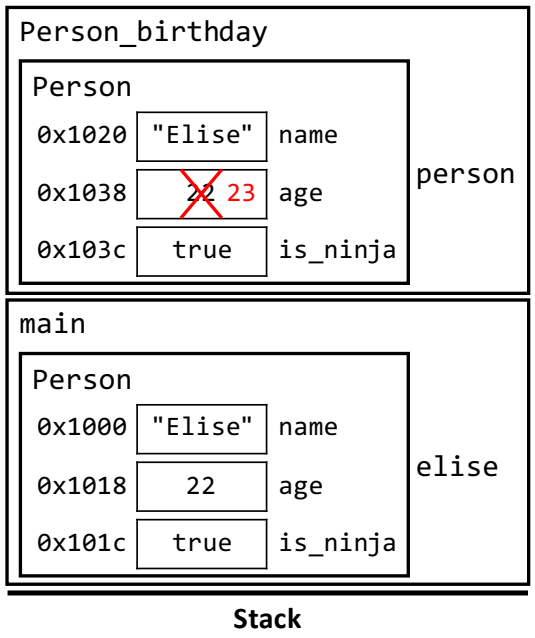

Figure 10.4: Passing a class-type object by value produces a copy that lives in the activation record of the callee.

The modification happens on a `Person` object that lives in the activation record for `Person_birthday()`. This copy will die when `Person_birthday()` returns, and the object that lives in `main()` is unchanged.

Instead, we need to pass the struct indirectly, either using a reference or a pointer. The following uses a pointer, and its execution is shown in Figure 10.5:

```
// REQUIRES: ptr points to a valid Person object
// MODIFIES: *ptr
// EFFECTS: Increases the person's age by one. If they are now older
//          than 70, they are no longer a ninja.
void Person_birthday(Person *ptr) {
  ++(*ptr).age;
  if ((*ptr).age > 70) {
    (*ptr).is_ninja = false;
  }
}
```

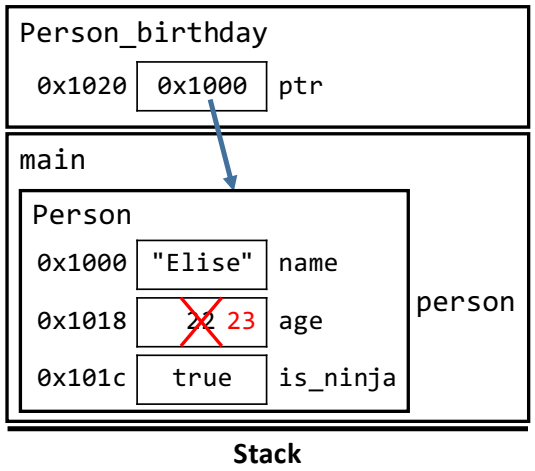

Figure 10.5: Passing a pointer to a class-type object avoids making a copy of that object.

The code uses the * operator to dereference the pointer, then uses the . operator to access a member of the resulting Person object. The parentheses around the dereference are required because the postfix . has higher precedence than the prefix *.

C++ provides the -> operator as a shorthand for dereference followed by member access. The following definition of Person_birthday() is equivalent to the one above:

```
void Person_birthday(Person *ptr) {
  ++ptr->age;
  if (ptr->age > 70) {
    ptr->is_ninja = false;
  }
}
```

Of course, this one is nicer to read and to write, so we should make use of the -> operator when possible.

The Person_birthday() function needs to modify the underlying Person object, so we cannot declare the Person as const. If a function does not need to modify the underlying object, then it should be declared as const. Declaring a struct as const prevents any of its members from being modified.

```
// REQUIRES: ptr points to a valid Person object
// MODIFIES: nothing
// EFFECTS: Prints a one-sentence description of the person
void Person_describe(const Person *ptr) {
```

(continues on next page)

(continued from previous page)

```
  cout << ptr->name << " is " << ptr->age << " years old and ";
  if (ptr->is_ninja) {
    cout << "is a ninja!" << endl;
  } else {
    cout << "is not a ninja." << endl;
  }
}
```

Except for very small structs, we generally do not pass structs by value, since creating a copy of a large struct can be expensive. Instead, we pass them by pointer or by reference. If the original object needs to be modified, we use a pointer or reference to non-const. Otherwise, we use a pointer or reference to const:

```
void func_ptr(const Person *p);   // const can go on either side of
void func_ref(Person const &p);   // Person, but must be to the left
                                  // of * or & for the Person itself
                                  // to be non-modifiable
```

10.1 Compound Objects and const

Since a class-type object has a value, it can be declared as `const`, which prevents any of its members from being modified. As an example, consider the following struct definition:

```
struct Foo {
  int num;
  int *ptr;
};
```

Like any `const` object, a `const Foo` must be initialized upon creation:

```
int main() {
  int x = 3;
  const Foo foo = { 4, &x };
  ...
}
```

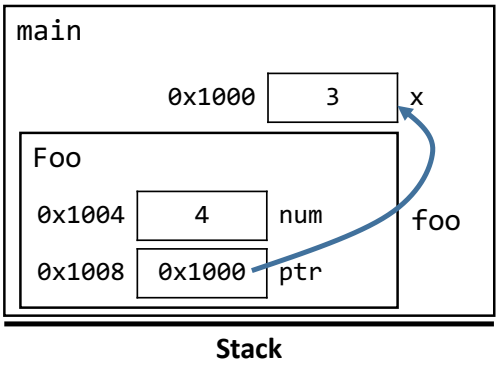

Figure 10.6: Contents of a `Foo` object. Declaring the object as `const` only prohibits modifications to the subobjects contained within the memory for the object.

With `foo` declared as `const`, attempting to modify any of its members results in a compile error:

```
foo.num = -1;      // ERROR
++foo.ptr;         // ERROR
```

Modifications cannot be made to any of the subobjects that live within the memory of an object declared `const`.[3]

On the other hand, it is possible to modify the object that `foo.ptr` points to:

```
*foo.ptr = -1;
cout << x;         // prints -1
```

Since `foo` is `const`, `foo.ptr` is a const pointer, and the expression has type `int * const`. This means it is not a pointer to const, so modifying the value of the object it is pointing at is allowed. Looking at it another way, the object `x` lives outside the memory for `foo`, so the fact that `foo` is `const` has no effect on whether or not `x` can be modified through a pointer that lives within `foo`.

[3] A member can be declared mutable, which allows it to be modified even if the object that contains it is const.

CHAPTER

ELEVEN

ABSTRACT DATA TYPES IN C

Recall that abstraction is the idea of separating *what* something is from *how* it works, by separating interface from implementation. Previously, we saw *procedural abstraction*, which applies abstraction to computational processes. With procedural abstraction, we use functions based on their signature and documentation without having to know details about their definition.

The concept of abstraction can be applied to data as well. An *abstract data type (ADT)* separates the interface of a data type from its implementation, and it encompasses both the data itself as well as functionality on the data. An example of an ADT is the `string` type in C++, used in the following code:

```
string str1 = "hello";
string str2 = "jello";
cout << str1 << endl;
if (str1.length() == str2.length()) {
  cout << "Same length!" << endl;
}
```

This code creates two strings and initializes them to represent different values, prints out one of them, and compares the lengths of both – all without needing to any details about the implementation of `string`. Rather, it relies solely on the interface provided by the `string` abstraction.

A `string` is an example of a full-featured C++ ADT, providing customized initialization, overloaded operations such as the stream-insertion operator, member functions, and so on. We will start with the simpler model of C ADTs, deferring C++ ADTs until next time.

The C language only has support for structs with data members (i.e. member variables). While this is sufficient to represent the data of an ADT, the functions that operate on the ADT must be defined separately from the struct. The following is the data definition of an ADT to represent triangles:

```
// A triangle ADT.
struct Triangle {
  double a;
  double b;
  double c;
};

int main() {
  Triangle t1 = { 3, 4, 5 };
  Triangle t2 = { 2, 2, 2 };
}
```

The `Triangle` struct contains three member variables, one for each side of the triangle, each represented by a `double`. The example in `main()` creates and initializes two `Triangle` structs, resulting in the memory layout in Figure 11.1.

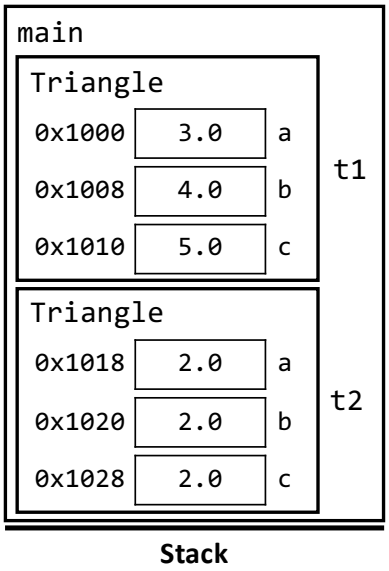

Figure 11.1: Two local `Triangle` objects.

An ADT also includes functions that operate on the data. We can define functions to compute the perimeter of a triangle or to modify it by scaling each of the sides by a given factor:

```
// REQUIRES: tri points to a valid Triangle
// EFFECTS:  Returns the perimeter of the given Triangle.
double Triangle_perimeter(const Triangle *tri) {
  return tri->a + tri->b + tri->c;
}

// REQUIRES: tri points to a valid Triangle; s > 0
// MODIFIES: *tri
// EFFECTS:  Scales the sides of the Triangle by the factor s.
void Triangle_scale(Triangle *tri, double s) {
  tri->a *= s;
  tri->b *= s;
  tri->c *= s;
}
```

Our naming convention for functions that are part of a C-style ADT is to prepend the function name with the name of the ADT, `Triangle` in this case. The first parameter is a pointer to the actual `Triangle` object the function works on. If the object need not be modified, we declare the pointer as a pointer to const.

The following demonstrates how to use the `Triangle` ADT functions:

```
int main() {
  Triangle t1 = { 3, 4, 5 };
  Triangle_scale(&t1, 2);                    // sides are now 6, 8, 10
  cout << Triangle_perimeter(&t1) << endl;  // prints 24
}
```

The code creates a `Triangle` as a local variable and initializes it with sides 3, 4, and 5. It then scales the sides by a factor of 2 by calling `Triangle_scale()`. Since that function takes a pointer to the actual triangle, we use the address-of operator to obtain and pass the address of `t1`, as shown in Figure 11.2.

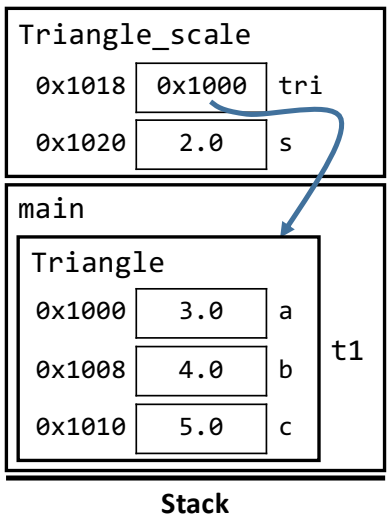

Figure 11.2: Passing a pointer to a `Triangle` object.

The function scales each side of the triangle, resulting in `t1` having sides of 6, 8, and 10. We then call `Triangle_perimeter()` on the address of `t1`, which computes the value 24.

In this example, the code in `main()` need not worry about the implementation of `Triangle_scale()` or `Triangle_perimeter()`. Instead, it relies on abstraction, using the functions for what they do rather than how they do it. However, in initializing `t1` itself, the code *is* relying on implementation details – specifically, that a `Triangle` is implemented as three `double` members that represent the lengths of the sides. If the implementation were to change to represent a triangle as two sides and the angle between them, for instance, then the behavior of the code in `main()` would change, and it would no longer print 24. Thus, we need to abstract the initialization of a `Triangle`, avoiding having to initialize each member directly. We do so by defining a `Triangle_init()` function:

```
// REQUIRES: tri points to a Triangle object
// MODIFIES: *tri
// EFFECTS:  Initializes the triangle with the given side lengths.
void Triangle_init(Triangle *tri, double a_in,
                   double b_in, double c_in) {
  tri->a = a_in;
  tri->b = b_in;
  tri->c = c_in;
}

int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5);
  Triangle_scale(&t1, 2);
  cout << Triangle_perimeter(&t1) << endl;
}
```

The user of the `Triangle` ADT creates an object without an explicit initialization and then calls `Triangle_init()` on its address to initialize it, providing the side lengths. After that call, the `Triangle` has been properly initialized and can be used with the other ADT functions. Now if the implementation of `Triangle` changes, as long as the interface remains the same, the code in `main()` will work as before. The code within the ADT, in the `Triangle_...` functions, will need to change, but outside code that uses the ADT will not. The following illustrates an implementation of `Triangle` that represents a triangle by two sides and an angle:

```
// A triangle ADT.
struct Triangle {
  double side1;
  double side2;
  double angle;
};

// REQUIRES: tri points to a Triangle object
// MODIFIES: *tri
// EFFECTS:  Initializes the triangle with the given side lengths.
void Triangle_init(Triangle *tri, double a_in,
                   double b_in, double c_in) {
  tri->side1 = a_in;
  tri->side2 = b_in;
  tri->angle = std::acos((std::pow(a_in, 2) + std::pow(b_in, 2) -
                          std::pow(c_in, 2)) /
                         (2 * a_in * b_in));
}

// REQUIRES: tri points to a valid Triangle
// EFFECTS:  Returns the first side of the given Triangle.
double Triangle_side1(const Triangle *tri) {
  return tri->side1;
}

// REQUIRES: tri points to a valid Triangle
// EFFECTS:  Returns the second side of the given Triangle.
double Triangle_side2(const Triangle *tri) {
  return tri->side2;
}

// REQUIRES: tri points to a valid Triangle
// EFFECTS:  Returns the third side of the given Triangle.
double Triangle_side3(const Triangle *tri) {
  return std::sqrt(std::pow(tri->side1, 2) +
                   std::pow(tri->side2, 2) -
                   2 * tri->side1 * tri->side2 * std::acos(tri->angle));
}

// REQUIRES: tri points to a valid Triangle
// EFFECTS:  Returns the perimeter of the given Triangle.
double Triangle_perimeter(const Triangle *tri) {
  return Triangle_side1(tri) + Triangle_side2(tri) + Triangle_side3(tri);
}

// REQUIRES: tri points to a valid Triangle; s > 0
// MODIFIES: *tri
// EFFECTS:  Scales the sides of the Triangle by the factor s.
void Triangle_scale(Triangle *tri, double s) {
  tri->side1 *= s;
  tri->side2 *= s;
}
```

Here, we have added *accessor* or *getter* functions for each of the sides, allowing a user to obtain the side lengths without needing to know implementation details. Even within the ADT itself, we have used `Triangle_side3()` from within `Triangle_perimeter()` to avoid code duplication.

The REQUIRES clauses of the ADT functions make a distiction between `Triangle` objects and *valid* `Triangle` objects. The former refers to an object that is of type `Triangle` but may not have been properly initialized, while the latter refers to a `Triangle` object that has been initialized by a call to `Triangle_init()`. Except for `Triangle_init()`, the ADT functions all work on valid `Triangle`s.

Now that we have a full definition of a C-style ADT, we adhere to the following convention for working with one: the user of a C-style ADT may only interact with the ADT through its interface, meaning the functions defined as part of the ADT's interface. The user is generally **prohibited** from accessing struct member variables directly, as those are implementation details of the ADT. **This convention also holds in testing an ADT**, since tests should only exercise the behavior of an ADT and not its implementation.

11.1 Representation Invariants

When designing an abstract data type, we must build a data representation on top of existing types. Usually, there will be cases where the underlying data representation permits combinations of values that do not make sense for our ADT. For example, not every combination of three `double`s represents a valid triangle – a `double` may have a negative value, but a triangle may not have a side with negative length. The space of values that represent valid instances of a triangle abstraction is a subset of the set of values that can be represented by three `double`s, as illustrated in Figure 11.3.

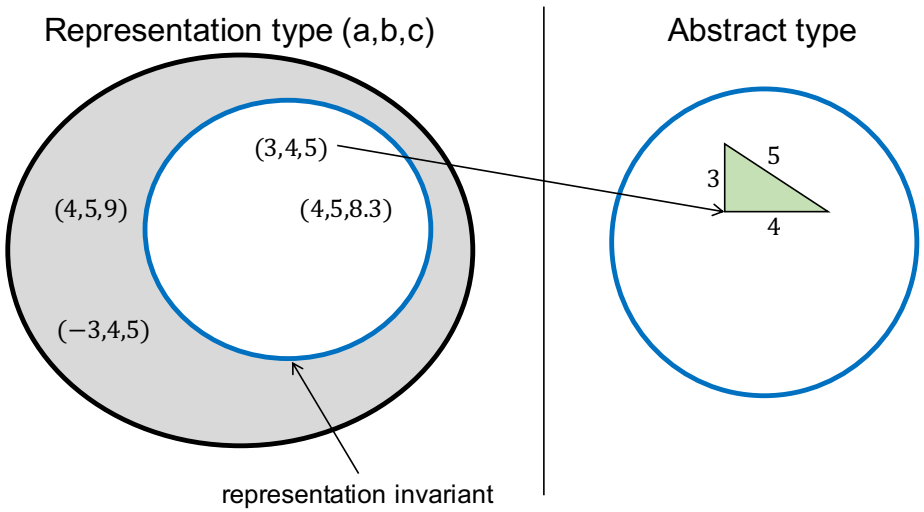

Figure 11.3: Representation invariants define the valid subset of the values allowed by the data representation of an ADT.

Thus, when designing an ADT, we need to determine the set of values that are valid for the ADT. We do so by specifying *representation invariants* for our ADT, which describe the conditions that must be met in order to make an object valid. For a triangle represented as a `double` for each side, the following representation invariants must hold:

- The length of each side must be positive.
- The *triangle inequality* must hold: the sum of any two sides must be strictly greater than the remaining side.

Often, we document the representation invariants as part of the ADT's data definition:

```
// A triangle ADT.
struct Triangle {
  double a;
  double b;
  double c;
  // INVARIANTS: a > 0 && b > 0 && c > 0 &&
  //             a + b > c && a + c > b && b + c > a
};
```

We then enforce the invariants when constructing or modifying an ADT object by encoding them into the REQUIRES clauses of our functions. We can use assertions to check for them as well, where possible:

```
// REQUIRES: tri points to a Triangle object;
//           each side length is positive (a > 0 && b > 0 && c > 0);
//           the sides meet the triangle inequality
//           (a + b > c && a + c > b && b + c > a)
// MODIFIES: *tri
// EFFECTS:  Initializes the triangle with the given side lengths.
void Triangle_init(Triangle *tri, double a, double b, double c) {
  assert(a > 0 && b > 0 && c > 0);              // positive lengths
  assert(a + b > c && a + c > b && b + c > a);  // triangle inequality
  tri->a = a;
  tri->b = b;
  tri->c = c;
}

// REQUIRES: tri points to a valid Triangle; s > 0
// MODIFIES: *tri
// EFFECTS:  Scales the sides of the Triangle by the factor s.
void Triangle_scale(Triangle *tri, double s) {
  assert(s > 0);  // positive lengths
  tri->a *= s;
  tri->b *= s;
  tri->c *= s;
}
```

11.2 Plain Old Data

As mentioned above, we adhere to the convention of only interacting with an ADT through its interface. Usually, this means that we do not access the data members of an ADT in outside code. However, occasionally we have the need for an ADT that provides no more functionality than grouping its members together. Such an ADT is just *plain old data (POD)*[4], without any functions that operate on that data, and we define its interface to be the same as its implementation.

The following is an example of a `Pixel` struct used as a POD:

```
// A pixel that represents red, green, and blue color values.
struct Pixel {
  int r; // red
```

(continues on next page)

[4] We use the term "plain old data" in the generic sense and not as the specific C++ term. C++ has a generalization of POD types called aggregates. Technically, the `Person` struct we saw last time is an aggregate but not a POD. What we mention here for POD types generally applies to aggregates as well.

(continued from previous page)

```
  int g; // green
  int b; // blue
};

int main() {
  Pixel p = { 255, 0, 0 };
  cout << p.r << " " << p.g << " " << p.b << endl;
}
```

The `Pixel` ADT consists of just a data representation with no further functionality. Since it is a POD, its interface and implementation are the same, so it is acceptable to access its members directly.

11.3 Abstraction Layers

As with procedural abstraction, data abstraction is also defined in layers, with each layer interacting solely with the interface of the layer below and not its implementation. For example, we can represent an image using three matrices, one for each color channel. Any code that uses an image relies on the image interface, without needing to know that it is implemented over three matrices. Each matrix in turn can be represented using a single-dimensional vector. Code that uses a matrix relies on the 2D abstraction provided by the interface without needing to know that it is implemented as a 1D vector under the hood.

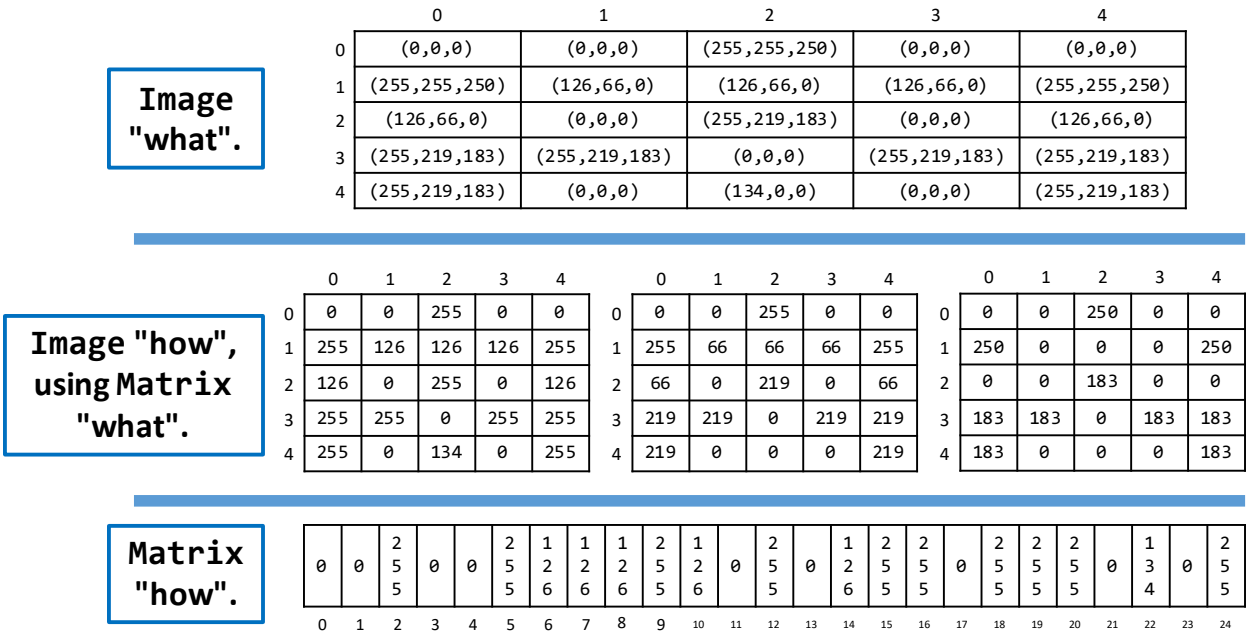

Figure 11.4: Abstraction layers for an image.

11.4 Testing an ADT

As mentioned previously, code outside of an ADT's implementation must interact with the ADT solely through its interface, including test code. Modifying an ADT's implementation should not require modifying its test code – we should be able to immediately run our regression tests in order to determine whether or not the ADT still works.

Adhering to the interface often means that we can't test each ADT function individually. For instance, we cannot test `Triangle_init()` in isolation; instead, we can test it in combination with the side accessors (e.g. `Triangle_side1()`) to determine whether or not the initialization works correctly. Instead of testing individual functions, we test individual *behaviors*, such as initialization.

As another example, let's proceed to design and test an ADT to represent a coordinate in two-dimensional space, using the principle of *test-driven development* that we saw previously. We will use polar coordinates, which represent a coordinate by the radius from the origin and angle from the horizontal axis, and we reflect this in the name of the ADT and its interface.

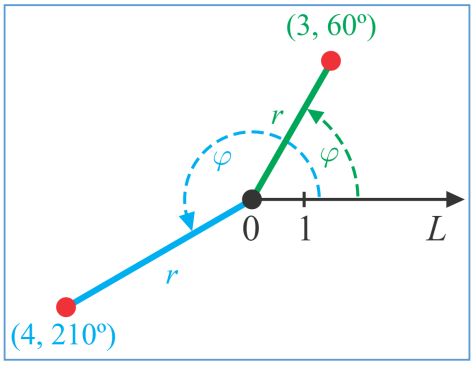

Figure 11.5: Polar representation of a point in two-dimensional space.

We start by determining the interface of the ADT:

```
// A set of polar coordinates in 2D space.
struct Polar;

// REQUIRES: p points to a Polar object
// MODIFIES: *p
// EFFECTS: Initializes the coordinate to have the given radius and
//          angle in degrees.
void Polar_init(Polar* p, double radius, double angle);

// REQUIRES: p points to a valid Polar object
// EFFECTS: Returns the radius portion of the coordinate as a
//          nonnegative value.
double Polar_radius(const Polar* p);

// REQUIRES: p points to a valid Polar object
// EFFECTS:  Returns the angle portion of the coordinate in degrees as
//           a value in [0, 360).
double Polar_angle(const Polar* p);
```

We then proceed to write some test cases, following the principles of test-driven development:

```
// Basic test of initializing a Polar object.
TEST(test_init_basic) {
  Polar p;
  Polar_init(&p, 5, 45);

  ASSERT_EQUAL(Polar_radius(&p), 5);
  ASSERT_EQUAL(Polar_angle(&p), 45);
}
```

We can then proceed to define a data representation. As part of this process, we should consider what representation invariants our ADT should have. For our `Polar` ADT, a reasonable set of invariants is that the radius is nonnegative, and the angle is in the range $[0, 360)$ (using degrees rather than radians)[5]:

```
struct Polar {
  double r;
  double phi;
  // INVARIANTS: r >= 0 && phi >= 0 && phi < 360
};
```

Now that we have a data representation, we can make an initial attempt at implementing the functions as well:

```
void Polar_init(Polar* p, double radius, double angle) {
  p->r = radius;
  p->phi = angle;
}

double Polar_radius(const Polar* p) {
  return p->r;
}

double Polar_angle(const Polar* p) {
  return p->phi;
}
```

We can run our existing test cases to get some confidence that our code is working. In addition, the process of coming up with a data representation, representation invariants, and function definitions often suggests new test cases. For instance, the following test cases check that the representation invariants are met when `Polar_init()` is passed values that don't directly meet the invariants:

```
// Tests initialization with a negative radius.
TEST(test_negative_radius) {
  Polar p;
  Polar_init(&p, -5, 225);
  ASSERT_EQUAL(Polar_radius(&p), 5);
  ASSERT_EQUAL(Polar_angle(&p), 45);
}

// Tests initialization with an angle >= 360.
TEST(test_big_angle) {
  Polar p;
  Polar_init(&p, 5, 405);
```

(continues on next page)

[5] A complete set of invariants would likely also specify a canonical representation of the origin. For example, it may specify that if the radius is 0, then so is the angle.

(continued from previous page)

```
  ASSERT_EQUAL(Polar_radius(&p), 5);
  ASSERT_EQUAL(Polar_angle(&p), 45);
}
```

Given our initial implementation, these test cases will fail. We can attempt to fix the problem as follows:

```
void Polar_init(Polar* p, double radius, double angle) {
  p->r = std::abs(radius);  // set radius to its absolute value
  p->phi = angle;
  if (radius < 0) {         // rotate angle by 180 degrees if radius
    p->phi = p->phi + 180;  // was negative
  }
}
```

Running our test cases again, we find that both `test_negative_radius` and `test_big_angle` still fail: the angle value returned by `Polar_angle()` is out of the expected range. We can fix this as follows:

```
void Polar_init(Polar* p, double radius, double angle) {
  p->r = std::abs(radius);  // set radius to its absolute value
  p->phi = angle;
  if (radius < 0) {         // rotate angle by 180 degrees if radius
    p->phi = p->phi + 180;  // was negative
  }
  p->phi = std::fmod(p->phi, 360);  // mod angle by 360
}
```

Now both test cases succeed. However, we may have thought of another test case through this process:

```
// Tests initialization with a negative angle.
TEST(test_negative_angle) {
  Polar p;
  Polar_init(&p, 5, -45);
  ASSERT_EQUAL(Polar_radius(&p), 5);
  ASSERT_EQUAL(Polar_angle(&p), 315);
}
```

Unfortunately, this test case fails. We can try another fix:

```
void Polar_init(Polar* p, double radius, double angle) {
  p->r = std::abs(radius);  // set radius to its absolute value
  p->phi = angle;
  if (radius < 0) {         // rotate angle by 180 degrees if radius
    p->phi = p->phi + 180;  // was negative
  }
  p->phi = std::fmod(p->phi, 360);  // mod angle by 360
  if (p->phi < 0) {         // rotate negative angle by 360
    p->phi += 360;
  }
}
```

Our test cases now all pass.

CHAPTER

TWELVE

COMMAND-LINE ARGUMENTS

So far, the programs we have considered have not worked with user input. More interesting programs, however, incorporate behavior that responds to user input. We will see two mechanisms for passing input to a program: command-line arguments and standard input.

Command-line arguments are arguments that are passed to a program when it is invoked from a shell or terminal. As an example, consider the following command:

```
$ g++ -Wall -O1 -std=c++17 -pedantic test.cpp -o test
```

Here, `g++` is the program we are invoking, and the arguments tell `g++` what to do. For instance, the `-Wall` argument tells the `g++` compiler to warn about any potential issues in the code, `-O1` tells the compiler to use optimization level 1, and so on.

Command-line arguments are passed to the program through arguments to `main()`. The `main()` function may have zero parameters, in which case the command-line arguments are discarded. It can also have two parameters[1], so the signature has the following form:

```
int main(int argc, char *argv[]);
```

The first argument is the number of command-line arguments passed to the program, and it is conventionally named `argc`. The second, conventionally named `argv`, contains each command-line argument as a *C-style string*. An array parameter is actually a pointer parameter, so the following signature is equivalent:

```
int main(int argc, char **argv);
```

Thus, the second parameter is a pointer to the first element of an array, each element of which is a pointer to the start of a C-style string, as shown in Figure 12.1.

The command-line arguments also include the name of the program as the first argument – this is often used in printing out error messages from the program.

We recommend converting command-line arguments to `std::string` before working with them, as it is less error-prone than working with C-style strings directly.

As an example, the following program takes an arbitrary number of integral arguments and computes their sum:

```
#include <iostream>
#include <string> // for string type and stoi() function

using namespace std;

int main(int argc, char *argv[]) {
```

(continues on next page)

[1] Implementations may also allow other signatures for `main()`.

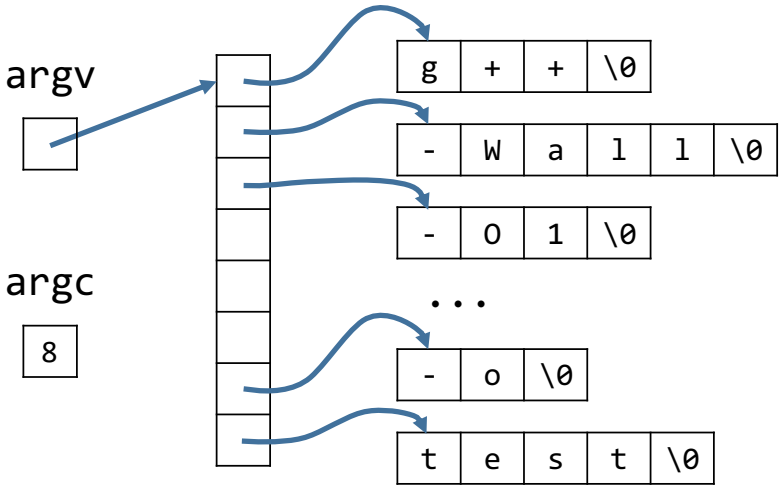

Figure 12.1: Representation of command-line arguments.

(continued from previous page)

```
  int sum = 0;
  for (int i = 1; i < argc; ++i) {
    string arg = argv[i];
    sum += stoi(argv[i]);
  }
  cout << "sum is " << sum << endl;
}
```

The first argument is skipped, since it is the program name. Each remaining argument is converted to an `int` by the `stoi()` function, which takes a string as the argument and returns the integer that it represents. For example, `stoi("123")` returns the number 123 as an `int`.

The following is an example of running the program:

```
$ ./sum.exe 2 4 6 8 10
sum is 30
```

CHAPTER

THIRTEEN

INPUT AND OUTPUT (I/O)

User input can also be obtained through *standard input*, which receives data that a user types into the console. In C++, the `cin` stream reads data from standard input. Data is extracted into an object using the *extraction operator* >>, and the extraction interprets the raw character data according to the target data type. For example, the following code extracts to `string`, which extracts individual words that are separated by whitespace:

```
string word;
while (cin >> word) {
  cout << "word = '" << word << "'" << endl;
}
```

The extraction operation evaluates to the `cin` stream, which has a truth value – if the extraction succeeds, it is true, but if the extraction fails, the truth value is false. Thus, the loop above will continue as long as extraction succeeds.

The following is an example of the program:

```
$ ./words.exe
hello world!
word = 'hello'
word = 'world!'
goodbye
word = 'goodbye'
```

The program only receives input after the user presses the enter key. The first line the user entered contains two words, each of which gets printed out. Then the program waits for more input. Another word is entered, so the program reads and prints it out. Finally, the user in this example inputs an *end-of-file* character – on Unix-based systems, the sequence Ctrl-d enters an end of file, while Ctrl-z does so on Windows systems. The end-of-file marker denotes the end of a stream, so extracting from `cin` fails at that point, ending the loop above.

The program above prints output to *standard output*, represented by the `cout` stream. The *insertion operator* << inserts the text representation of a value into an output stream.

13.1 I/O Redirection

Shells allow *input redirection*, which passes the data from a file to standard input rather than reading from the keyboard. For instance, if the file `words.in` contains the data:

```
hello world!
goodbye
```

Then using the < symbol before the filename redirects the file to standard input at the command line:

```
$ ./words.exe < words.in
word = 'hello'
word = 'world!'
word = 'goodbye'
```

A file has an implicit end of file at the end of its data, and the program terminates upon reaching the end of the file.

We can also do *output redirection*, where the shell writes the contents of standard output to a file. The symbol for output redirection is >:

```
$ ./words.exe > result.out
hello world!
goodbye
$ cat result.out
word = 'hello'
word = 'world!'
word = 'goodbye'
```

Here, we redirect the output to the file `result.out`. We then enter input from the keyboard, ending with the Ctrl-d sequence. When the program ends, we use the `cat` command to display the contents of `result.out`.

Input and output redirection can also be used together:

```
$ ./words.exe < words.in > result.out
$ cat result.out
word = 'hello'
word = 'world!'
word = 'goodbye'
```

13.2 Example: Adding Integers

Using standard input, we can write a program that adds up integers entered by a user. The program will terminate either upon reaching an end of file or if the user types in the word done:

```
#include <iostream>
#include <string>   // for stoi()

using namespace std;

int main() {
  int sum = 0;
  cout << "Enter some numbers to sum." << endl
  string word;
  while (cin >> word && word != "done") {
    sum += stoi(word);
  }
  cout << "sum is " << sum << endl;
}
```

The code extracts to a string so that it can be compared to the string "done". (The latter is a C-style string, but C++ strings can be compared with C-style strings using the built-in comparison operators.)

The following is an example of running the program:

```
$ ./sum
Enter some numbers to sum.
2
4
6
done
sum is 12
```

An alternate version of the program extracts directly to an `int`. However, it can only be terminated by an end of file or other failed extraction:

```
#include <iostream>

using namespace std;

int main() {
  int sum = 0;
  cout << "Enter some numbers to sum." << endl
  int number;
  while (cin >> number) {
    sum += number;
  }
  cout << "sum is " << sum << endl;
}
```

13.3 File I/O

A program can also read and write files directly using file streams. It must include the `<fstream>` header, and it can then use an `ifstream` to read from a file and an `ofstream` to write to a file. The former supports the same interface as `cin`, while the latter has the same interface as `cout`.

An `ifstream` object can be created from a file name:

```
string filename = "words.in";
ifstream fin(filename);
```

Alternatively, the `ifstream` object can be created without a file name, and then its `open()` function can be given the name of the file to open:

```
string filename = "words.in";
ifstream fin;
fin.open(filename);
```

In general, a program should check if the file was successfully opened, regardless of the mechanism used to create the `ifstream`:

```
if (!fin.is_open()) {
  cout << "open failed" << endl;
  return 1;
}
```

Once we've determined the file is open, we can read from it like `cin`. The following program reads individual words from the file `words.in` and prints them:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {
  string filename = "words.in";
  ifstream fin;
  fin.open(filename);
  if (!fin.is_open()) {
    cout << "open failed" << endl;
    return 1;
  }
  string word;
  while (fin >> word) {
    cout << "word = '" << word << "'" << endl;
  }
  fin.close();  // optional
}
```

The program closes the file before exiting. Doing so explicitly is optional – it will happen automatically at the end of the `ifstream` object's lifetime (e.g. when it goes out of scope if it is a local variable).

Best practice is to extract from an input stream, whether it is `cin` or an `ifstream`, in the test of a loop or conditional. That way, the test will evaluate to false if the extraction fails. The following examples all print the last word twice because they do not check for failure between extracting and printing a word:

```
while (!fin.fail()) {
  fin >> word;
  cout << word;
}

while (fin.good()) {
  fin >> word;
  cout << word;
}

while (!fin.eof()) {
  fin >> word;
  cout << word;
}

while (fin) {
  fin >> word;
  cout << word;
}
```

The following is printed when using any of the loops above:

```
$ ./main.exe
hello
world!
goodbye
```

(continues on next page)

(continued from previous page)

```
goodbye
```

Multiple extractions can be placed in the test of a loop by chaining them. The test evaluates to true when all extractions succeed. For example, the following reads two words at a time:

```
string word1, word2;
while (fin >> word1 >> word2) {
  cout << "word1 = '" << word1 << "'" << endl;
  cout << "word2 = '" << word2 << "'" << endl;
}
```

For `words.in`, only the first two words are printed, since the test will fail in the second iteration when it tries to read a fourth word:

```
$ ./main.exe
word1 = 'hello'
word2 = 'world!'
```

An entire line can be read using the `getline()` function, which takes in an input stream and a target string (by reference) and returns whether or not reading the line succeeded. If so, the target string will contain the full line read:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {
  string filename = "hello.txt";
  ifstream fin;
  fin.open(filename);
  if (!fin.is_open()) {
    cout << "open failed" << endl;
    return 1;
  }
  string line;
  while (getline(fin, line)) {
    cout << "line = '" << line << "'" << endl;
  }
}
```

For `words.in`, this will result in:

```
$ ./main.exe
line = 'hello world!'
line = 'goodbye'
```

An `ofstream` works similarly to an `ifstream`, except that it is used for printing output to a file. The following program prints data to the file `output.txt`:

```
#include <iostream>
#include <fstream>
#include <string>
```

(continues on next page)

(continued from previous page)

```
using namespace std;

int main() {
  const int SIZE = 4;
  int data[SIZE] = { 1, 2, 3, 4 };
  string filename = "output.txt";
  ofstream fout;
  fout.open(filename);
  if (!fout.is_open()) {
    cout << "open failed" << endl;
    return 1;
  }
  for (int i = 0; i < 4; ++i) {
    fout << "data[" << i << "] = " << data[i] << endl;
  }
  fout.close();  // optional
}
```

The following shows the resulting data in `output.txt`:

```
$ cat output.txt
data[0] = 1
data[1] = 2
data[2] = 3
data[3] = 4
```

CHAPTER

FOURTEEN

MORE ON STREAMS

Previously, we learned about the *standard input and output streams*, as well as *file streams*. We examine the relationship between streams more closely now, as well as how to write unit tests using string streams.

A *stream* is an abstraction over a *source* of input, from which we can read data, or a *sink* of output, to which we can write data. Streams support the abstraction of character-based input and output over many underlying resources, including the console, files, the network, strings, and so on.

In C++, input streams generally derive from `istream`[2]. We will see what this means specifically when we look at *inheritance* and *polymorphism* in the future. For our purposes right now, this means that we can pass different kinds of input-stream objects to a function that takes in a reference to an `istream`. Similarly, output streams generally derive from `ostream`, and we can pass different kinds of output-stream objects to a function that takes in a reference to an `ostream`.

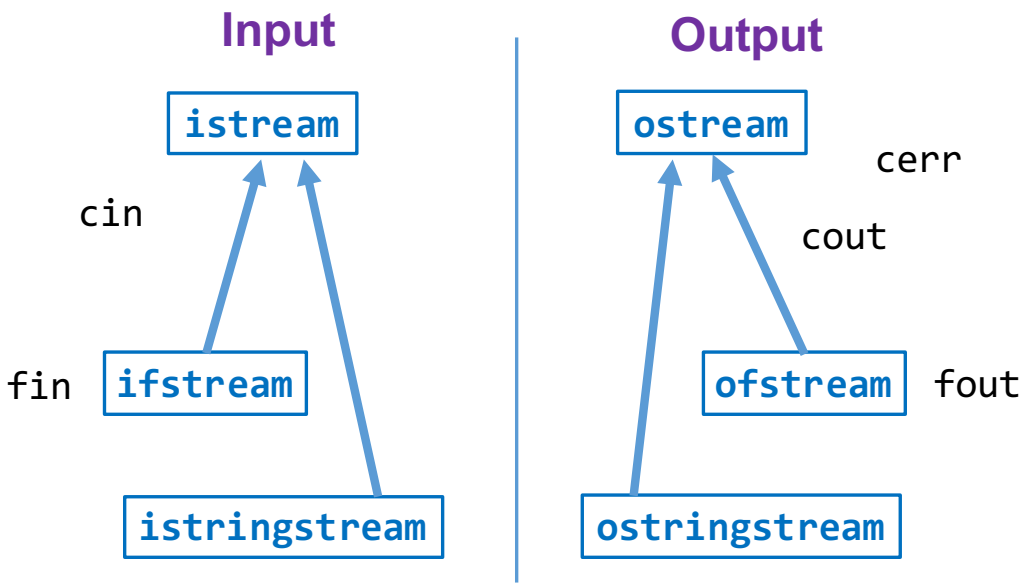

Figure 14.1: Relationships between different kinds of input and output streams.

To write data into an output stream, we use the *insertion operator* <<. The actual data written out depends on both the value itself as well as its type. For instance, if we use a `string` as the right-hand-side operand, the insertion operation will write the characters from the string into the stream:

[2] The `istream` type is actually an alias for `basic_istream<char>`, which is an input stream that supports input using the `char` type. The same goes for `ostream` and `basic_ostream<char>`.

```
int i = 123;
cout << i;           // writes the characters 123
double d = 12.3;
cout << d;           // writes the characters 12.3
char c = 'c';
cout << c;           // writes the character c
string s = "Hello";
cout << s;           // writes the characters Hello
```

Expressions that apply an operator generally evaluate to a value. In the case of stream insertion, the result is the actual stream object itself. This allows us to chain insertion operations:

```
cout << i << d << endl;
// equivalent to ((cout << i) << d) << endl;
//                ^^^^^^^^^
//       evaluates back to the cout object
```

To read data from an input stream, we use the *extraction operator* >>, with an object on the right-hand side. The characters are interpreted according to the type of the object. For built-in types, whitespace is generally skipped when extracting.

```
char c;
cin >> c;   // reads a single character; does not skip whitespace
string s;
cout >> s;  // reads in one "word", delimited by whitespace
int i;
cin >> i;   // attempts to parse the next characters as an integer value
double d;
cin >> d;   // attempts to parse the next characters as a floating-point value
```

As with the insertion operator, an expression that applies the extraction operator evaluates back to the stream itself, allowing extraction operations to be chained:

```
cin >> c >> s >> i >> d;
```

14.1 String Streams

When writing unit tests, we often want the tests to be standalone without requiring access to external data. For tests that work with streams, we can use string streams rather than standard input/output or file streams. To use a string stream, we `#include <sstream>`. We can then use an `istringstream` as an input stream, and an `ostringstream` as an output stream.

The following is an example of using an `istringstream` to represent input data for testing a function that takes in an input stream:

```
TEST(test_image_basic) {
  // A hardcoded PPM image
  string input = "P3\n2 2\n255\n255 0 0 0 255 0 \n";
  input += "0 0 255 255 255 255 \n";

  // Use istringstream for simulated input
  istringstream ss_input(input);
```

(continues on next page)

(continued from previous page)

```
  Image *img = new Image;
  Image_init(img, ss_input);

  ASSERT_EQUAL(Image_width(img), 2);
  Pixel red = { 255, 0, 0 };
  ASSERT_TRUE(Pixel_equal(Image_get_pixel(img, 0, 0), red));
  delete img;
}
```

We start with a `string` that contains the actual input data and then construct an `istringstream` from that. We can then pass that `istringstream` object to a function that has a parameter of type `istream &`. When that function extracts data, the result will be the data from the string we used to construct the `istringstream`.

We can similarly use an `ostringstream` to test a function that takes an output stream:

```
TEST(test_matrix_basic) {
  Matrix *mat = new Matrix;
  Matrix_init(mat, 3, 3);
  Matrix_fill(mat, 0);
  Matrix_fill_border(mat, 1);

  // Hardcoded correct output
  string output_correct = "3 3\n1 1 1 \n1 0 1 \n1 1 1 \n";

  // Capture output in ostringstream
  ostringstream ss_output;
  Matrix_print(mat, ss_output);
  ASSERT_EQUAL(ss_output.str(), output_correct);
  delete mat;
}
```

We default construct an `ostringstream` object and pass it to a function with a parameter of type `ostream &`. The `ostringstream` will internally capture the data that the function inserts into it. We can then call `.str()` on the `ostringstream` to obtain a `string` that contains that data, which we can then compare to another `string` that contains the expected output.

CHAPTER

FIFTEEN

PROGRAM DESIGN

Apologies – this section has not been written yet.

CHAPTER

SIXTEEN

ABSTRACT DATA TYPES IN C++

Now that we've seen the concept of abstract data types (ADTs), we proceed to examine the mechanisms C++ provides for defining an ADT. Unlike C, C++ allows the data and functions of an ADT to be defined together. It also enables an ADT to prevent access to internal implementation details, as well as to guarantee that an object is appropriately initialized when it is created.

Our convention in this course is to use the word *struct* to refer to C-style ADTs, as well as to use the `struct` keyword to define them. We use the word *class* to refer to C++ ADTs and use the `class` keyword to define them. We will discuss the technical difference between the two keywords momentarily.

A C++ class includes both member variables, which define the data representation, as well as *member functions* that operate on the data. The following is a `Triangle` class in the C++ style:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in);

  double perimeter() const {
    return this->a + this->b + this->c;
  }

  void scale(double s) {
    this->a *= s;
    this->b *= s;
    this->c *= s;
  }
};
```

The class has member variables for the length of each side, defining the data representation. We defer discussion of the `public:` and `Triangle(...)` lines for now. Below those lines are member functions for computing the perimeter of a triangle and scaling it by a factor.

The following is an example of creating and using a `Triangle` object:

```
int main() {
  Triangle t1(3, 4, 5);
  t1.scale(2);
  cout << t1.perimeter() << endl;
}
```

We initialize a triangle by passing in the side lengths as part of its declaration. We can then scale a triangle by using the same dot syntax we saw for accessing a member variable: `<object>.<function>(<arguments>)`.

Before we discuss the details of what the code is doing, let us compare elements of the C-style definition and use of the triangle ADT with the C++ version. The following contrasts the definition of an ADT function between the two styles:

C-Style Struct	C++ Class

`void Triangle_scale(Triangle *tri,`
`  double s) {`
`  tri->a *= s;`
`  tri->b *= s;`
`  tri->c *= s;`
`}`
	`class Triangle {`
	`  void scale(double s) {`
	`  this->a *= s;`
	`  this->b *= s;`
	`  this->c *= s;`
	`  }`
	`};`

The following compares how objects are created and manipulated:

C-Style Struct	C++ Class

`Triangle t1;`
`Triangle_init(&t1, 3, 4, 5);`
`Triangle_scale(&t1, 2);`
	`Triangle t1(3, 4, 5);`
	
	`t1.scale(2);`

With the C-style struct, we defined a top-level `Triangle_scale()` function whose first argument is a pointer to the `Triangle` object we want to scale. With a C++ class, on the other hand, we define a `scale()` member function within the `Triangle` class itself. There is no need to prepend `Triangle_`, since it is clear that `scale()` is a member of the `Triangle` class. The member function also does not explicitly declare a pointer parameter – instead, the C++ language adds an *implicit* `this` parameter that is a pointer to the `Triangle` object we are working on. We can then use the `this` pointer in the same way we used the explicit `tri` pointer in the C style.

As for using a `Triangle` object, in the C style, we had to separately create the `Triangle` object and then initialize it with a call to `Triangle_init()`. In the C++ style, object creation and initialization are combined – we will see how later. When invoking an ADT function, in the C case we have to explicitly pass the address of the object we are working on. With the C++ syntax, the object is part of the syntax – it appears on the left-hand side of the dot, so the compiler automatically passes its address as the `this` pointer of the `scale()` member function.

The following contrasts the definitions of a function that treats the ADT object as const:

C-Style Struct	C++ Class

`double Triangle_perimeter(`
`  const Triangle *tri) {`
`  return tri->a +`
`  tri->b +`
`  tri->c;`
`}`
	`class Triangle {`
	`  double perimeter() const {`
	`  return this->a +`
	`  this->b +`
	`  this->c;`
	`  }`
	`};`

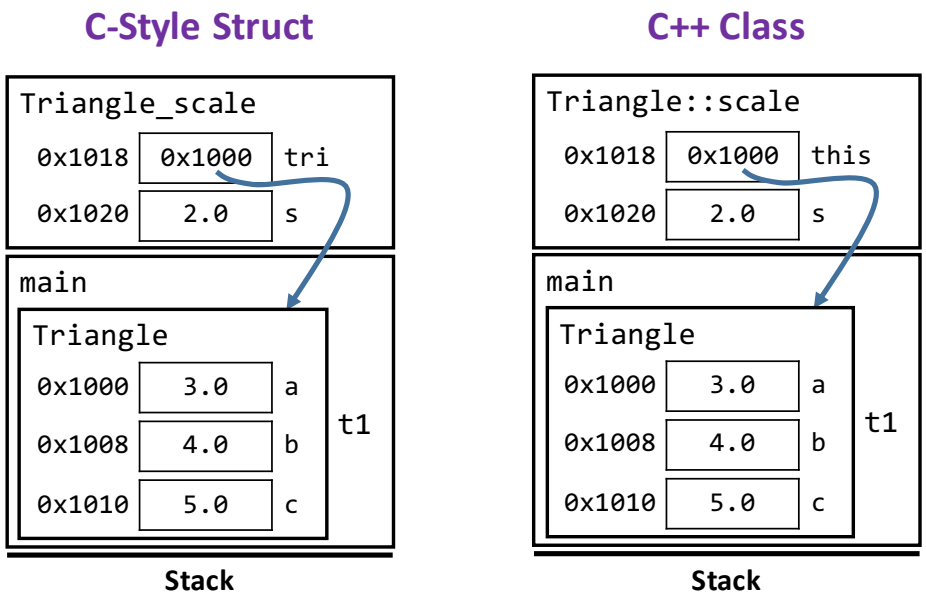

Figure 16.1: Memory layout when scaling a triangle in the C and C++ styles.

In the C style, we add the `const` keyword to the left of the `*` when declaring the explicit pointer parameter, resulting in `tri` being a pointer to const. In the C++ style, we don't have an explicit parameter where we can add the `const` keyword. Instead, we place the keyword after the parameter list for the function. The compiler will then make the implicit `this` parameter a pointer to const, as if it were declared with the type `const Triangle *`. This allows us to call the member function on a const `Triangle`:

```
const Triangle t1(3, 4, 5);
cout << t1.perimeter() << endl;  // OK: this pointer is a pointer to const
t1.scale(2);                     // ERROR: conversion from const to non-const
```

As with accessing member variables, we can use the arrow operator to invoke a member function through a pointer:

```
Triangle t1(3, 4, 5);
const Triangle *ptr = &t1;
cout << ptr->perimeter() << endl;  // OK: this pointer is a pointer to const
ptr->scale(2);                     // ERROR: conversion from const to non-const
```

16.1 Implicit `this->`

Since member variables and member functions are both located within the scope of a class, C++ allows us to refer to members from within a member function without the explicit `this->` syntax. The compiler automatically inserts the member dereference for us:

```
class Triangle {
  double a;
  double b;
  double c;
```

(continues on next page)

(continued from previous page)

```
  ...

  double perimeter() const {
    return a + b + c; // Equivalent to: this->a + this->b + this->c
  }
};
```

This is also the case for invoking other member functions. For instance, the following defines and uses *functions to get* each side length:

```
class Triangle {
  double a;
  double b;
  double c;

  ...

  double side1() const {
    return a;
  }

  double side2() const {
    return b;
  }

  double side3() const {
    return c;
  }

  double perimeter() const {
    return side1() + side2() + side3();
    // Equivalent to: this->side1() + this->side2() + this->side3()
  }
};
```

In both cases, the compiler can tell that we are referring to members of the class and therefore inserts the `this->`. However, if there are names in a closer scope that conflict with the member names, we must use `this->` ourselves. The following is an example:

```
class Triangle {
  double a;

  ...

  double set_side1(double a) {
    this->a = a;
  }
};
```

Here, the *unqualified* `a` refers to the parameter `a`, since it is declared in a narrower scope than the member variable. We can still refer to the member `a` by *qualifying* its name with `this->`.

In general, we should avoid declaring variables in a local scope that *hide* names in an outer scope. Doing so in a constructor or set function is often considered acceptable, but it should be avoided elsewhere.

16.2 Member Accessibility

The data representation of an ADT is usually an implementation detail (*plain old data* being an exception). With C-style structs, however, we have to rely on programmers to respect convention and avoid accessing member variables directly. With C++ classes, the language provides us a mechanism for enforcing this convention: declaring members as *private* prevents access from outside the class, while declaring them as *public* allows outside access.[1] We give a set of members a particular access level by placing `private:` or `public:` before the members – that access level applies to subsequent members until a new access specifier is encountered, and any number of specifiers may appear in a class. The following is an example:

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in);

  double perimeter() const {
    return a + b + c;
  }

  void scale(double s) {
    a *= s;
    b *= s;
    c *= s;
  }
};
```

In this example, the members `a`, `b`, and `c` are declared as private, while `Triangle()`, `perimeter()`, and `scale()` are declared as public. Private members, whether variables or functions, can be accessed from within the class, even if they are members of a different object of that class. They cannot be accessed from outside the class:

```
int main() {
  Triangle t1(3, 4, 5);           // OK: Triangle() is public
  t1.scale(2);                    // OK: scale() is public
  cout << t1.perimeter() << endl; // OK: perimeter() is public

  // Die triangle! DIE!
  t1.a = -1;                      // ERROR: a is private
}
```

With the `class` keyword, the default access level is private. Thus, the `private:` at the beginning of the `Triangle` definition is redundant, and the following is equivalent:

```
class Triangle {
  double a;
  double b;
  double c;

public:
```

(continues on next page)

[1] Later, we will discuss *protected* members that are accessible to derived classes. This access level, however, is rarely used.

(continued from previous page)

```
  Triangle(double a_in, double b_in, double c_in);

  ...
};
```

We have seen previously that members declared within a `struct` are accessible from outside the struct. In fact, the **only** difference between the `struct` and `class` keywords when defining a class type is the default access level: public for `struct` but private for `class`.[2] However, we use the two keywords for different conventions in this course.

16.3 Constructors

A *constructor* is similar to a member function, except that its purpose is to initialize a class-type object. In most cases, C++ guarantees that a constructor is called when creating an object of class type.[3] The following examples all call a constructor:

```
Triangle t1;                    // calls zero-argument (default) constructor
Triangle t2(3, 4, 5);           // calls three-argument constructor
Triangle t3 = Triangle(3, 4, 5);  // calls three-argument constructor
// examples with "uniform initialization syntax":
Triangle t4{3, 4, 5};           // calls three-argument constructor
Triangle t5 = {3, 4, 5};        // calls three-argument constructor
Triangle t6 = Triangle{3, 4, 5};  // calls three-argument constructor
```

As can be seen above, there are many forms of syntax for initializing a `Triangle` object, all of which call a constructor. When no arguments are provided, the zero-argument, or *default*, constructor is called. We will discuss this constructor in more detail later.

The following does *not* call a constructor:

```
Triangle t7();   // declares a function called t7 that returns a Triangle
```

In fact, it doesn't create an object at all. Instead, it declares a function named `t7` that takes no arguments and returns a `Triangle`. A function declaration can appear at local scope, so this is interpreted as a function declaration regardless of whether it is at local or global scope.

So far, we declared a single constructor for `Triangle` as follows:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in);
};
```

[2] This also applies to inheritance. We will see next time that private inheritance is the default for a `class`, and we will need to use the `public` keyword to override this. The default for a `struct` is public inheritance.

[3] The exception is aggregate initialization, where an initializer list is used to directly initialize the members of a class-type object. This is only possible for *aggregates*, which are class types that have a restricted set of features. Our pattern of C-style structs obey the rules for an aggregate, though we define init functions instead of using aggregate initialization. Our convention of C++ classes results in class types that are not aggregates, so objects of such types can only be initialized through a constructor. We can still use initializer-list syntax for a non-aggregate – it will call a constructor with the values in the initializer list as arguments.

The syntax is similar to declaring a member function, except:

- There is no return type.
- The name of the constructor is the same as the name of the class.

Like a member function, the constructor has an implicit `this` parameter that points to the object being initialized. As in a member function, we can leave out `this->` to access a member, as long as there is no local name that hides the member. The following is a definition of the `Triangle` constructor:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  // poor constructor implementation
  Triangle(double a_in, double b_in, double c_in) {
    a = a_in;
    b = b_in;
    c = c_in;
  }
};
```

However, there is a problem with the definition above: the statements in the body of the constructor perform *assignment*, not *initialization*. Thus, the member variables are actually default initialized and then assigned new values. In this case, it is a minor issue, but it can be more significant in other cases. In particular there are several kinds of variables that allow initialization but not assignment:

- arrays
- references
- `const` variables
- class-type variables that disable assignment (e.g. streams)

Another case where initialization followed by assignment is problematic is for types where both initialization and assignment perform nontrivial operations – we lose efficiency by doing both when we only need initialization.

C++ provides two mechanisms for initializing a member variable:

- directly in the declaration of the variable, similar to initializing a non-member variable
- through a member-initializer list

A *member-initializer list* is syntax specific to a constructor. It is a list of initializations that appear between a colon symbol and the constructor body:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  // good constructor implementation
  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {}
};
```

An individual initialization consists of a member-variable name, followed by an initialization expression enclosed by parentheses (or curly braces). The constructor above initializes the member `a` to the value of `a_in`, `b` to the value of `b_in`, and `c` to the value of `c_in`. The constructor body is empty, since it has no further work to do.

If a member is initialized in both its declaration and a member-initializer list, the latter takes precedence, so that the initialization in the member declaration is ignored.

16.4 Default Initialization and Default Constructors

Every object in C++ is initialized upon creation, whether the object is of class type or not. If no explicit initialization is provided, it undergoes *default initialization*. Default initialization does the following:

- Objects of atomic type (e.g. `int`, `double`, pointers) are default initialized by doing nothing. This means they retain whatever value was already there in memory. Put another way, atomic objects have undefined values when they are default initialized.
- An array is default initialized by in turn default initializing its elements. Thus, an array of atomic objects is default initialized by doing nothing, resulting in undefined element values.
- A class-type object is default initialized by calling the *default constructor*, which is the constructor that takes no arguments. If no such constructor exists, or if it is inaccessible (e.g. it is private), a compile-time error results.

  An array of class-type objects is default initialized by calling the default constructor on each element. Thus, the element type must have an accessible default constructor in order to create an array of that type.

Within a class, if a member variable is neither initialized at declaration nor in the member-initializer list of a constructor, it is default initialized.

The default constructor is so named because it is invoked in default initialization, and it takes no arguments. We can define a default constructor for `Triangle` as follows, making the design decision to initialize the object as a 1x1x1 equilateral triangle:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  // default constructor
  Triangle()
    : a(1), b(1), c(1) {}

  // non-default constructor
  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {}
};
```

A class can have multiple constructors. This is a form of *function overloading*, which we will return to in the future. The compiler determines which compiler to invoke based on the arguments that are provided when creating a `Triangle` object:

```
Triangle t1;            // 1x1x1 -- calls zero-argument (default) constructor
Triangle t2(3, 4, 5);   // 3x4x5 -- calls three-argument constructor
```

16.4.1 Implicit Default Constructor

If a class declares no constructors at all, the compiler provides an *implicit default constructor*. The behavior of this constructor is as if it were empty, so that it default initializes each member variable:

```
struct Person {
  string name;
  int age;
  bool is_ninja;
  // implicit default constructor
  // Person() {}  // default initializes each member variable
};

int main() {
  Person elise;           // calls implicit default constructor
  cout << elise.name; // prints nothing: default ctor for string makes it empty
  cout << elise.age;      // prints undefined value
  cout << elise.is_ninja; // prints undefined value
};
```

If a class declares *any* constructors whatsoever, no implicit default constructor is provided:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in);

  double perimeter() const {
    return a + b + c;
  }

  void scale(double s) {
    a *= s;
    b *= s;
    c *= s;
  }
};

int main() {
  Triangle t1;  // ERROR: no implicit or explicit default constructor
}
```

In this case, if we want our type to have a default constructor, we have to explicitly write one:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  // explicit default constructor
```

(continues on next page)

(continued from previous page)

```
  Triangle()
    : a(1), b(1), c(1) {}

  // non-default constructor
  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {}

  double perimeter() const {
    return a + b + c;
  }

  void scale(double s) {
    a *= s;
    b *= s;
    c *= s;
  }
};

int main() {
  Triangle t1;  // OK: explicit default constructor
}
```

16.5 Get and Set Functions

With C++ classes, member variables are usually declared private, since they are implementation details. However, many C++ ADTs provide a means of accessing the abstract data through *get* and *set functions* (also called *getters* and *setters* or *accessor functions*). These are provided as part of the interface as an abstraction over the underlying data. The following are examples for `Triangle`:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  // EFFECTS: Returns side a of the triangle.
  double get_a() const {
    return a;
  }

  // REQUIRES: a_in > 0 && a_in < get_b() + get_c()
  // MODIFIES: *this
  // EFFECTS:  Sets side a of the triangle to a_in.
  void set_a(double a_in) {
    a = a_in;
  }
};
```

If the implementation changes, the interface can remain the same, so that outside code is unaffected:

```
class Triangle {
  double side1;  // new names
  double side2;
  double side3;

public:
  // EFFECTS: Returns side a of the triangle.
  double get_a() const {  // same interface
    return side1;         // different implementation
  }

  // REQUIRES: a_in > 0 && a_in < get_b() + get_c()
  // MODIFIES: *this
  // EFFECTS:  Sets side a of the triangle to a_in.
  void set_a(double a_in) {
    side1 = a_in;
  }
};
```

With a set function, we've introduced a new location from which the representation can be modified. We need to ensure that the representation invariants are still met. We can do so by writing and using a private function to check the invariants:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {
    check_invariants();
  }

  void set_a(double a_in) {
    a = a_in;
    check_invariants();
  }

private:
  void check_invariants() {
    assert(0 < a && 0 < b && 0 < c);
    assert(a + b > c  && a + c > b && b + c > a);
  }
}
```

It is good practice to check the invariants anywhere the representation can be modified. Here, we have done so in both the constructor and in the set function.

16.6 Information Hiding

Good abstraction design uses *encapsulation*, which groups together both the data and functions of an ADT. With a class, we get encapsulation by defining both member variables and member functions.

A proper abstraction also provides *information hiding*, which separates interface from implementation. Access specifiers such as `private` allow us to prevent the outside world from accessing implementation details.

We can further hide information from the sight of the users of an ADT by physically separating the code for the interface from the code for the implementation. The standard mechanism to do so in C++ is to place declarations in header files and definitions in source files. With a class, we place a class definition that only contains member declarations in the header file:

```
// Triangle.hpp

// A class that represents a triangle ADT.
class Triangle {
public:
  // EFFECTS: Initializes this to a 1x1x1 triangle.
  Triangle();

  // EFFECTS: Initializes this with the given side lengths.
  Triangle(double a_in, double b_in, double c_in);

  // EFFECTS: Returns the perimeter of this triangle.
  double perimeter() const;

  // REQUIRES: s > 0
  // MODIFIES: *this
  // EFFECTS:  Scales the sides of this triangle by the factor s.
  void scale(double s);

private:
  double a;
  double b;
  double c;
  // INVARIANTS:
  // positive side lengths: a > 0 && b > 0 && c > 0
  // triangle inequality: a + b > c && a + c > b && b + c > a
};
```

It is also generally preferable to declare the public members of the class before private members, so that users do not have to skip over implementation details to find the public interface.

We then define the constructors and member functions outside of the class definition, in the corresponding source file. In order to define a member function outside of a class, we need two things:

1) A declaration of the function within the class, so that the compiler (and other programmers) can tell that the member exists.

2) Syntax in the definition that tells the compiler that the function is a member of the associated class and not a top-level function.

The latter is accomplished by prefixing the member name with the class name, followed by the *scope-resolution operator*:

```cpp
// Triangle.cpp
#include "Triangle.hpp"

Triangle::Triangle()
  : a(1), b(1), c(1) {}

Triangle::Triangle(double a_in, double b_in, double c_in)
  : a(a_in), b(b_in), c(c_in) {}

double Triangle::perimeter() const {
  return a + b + c;
}

void Triangle::scale(double s) {
  a *= s;
  b *= s;
  c *= s;
}
```

This tells the compiler that the two constructors, as well as the `perimeter()` and `scale()` functions, are members of the `Triangle` class.

16.7 Testing a C++ ADT

We test a C++ ADT by writing test cases that live outside of the ADT itself. C++ forces us to respect the interface, since the implementation details are private:

```cpp
// Triangle_tests.cpp
#include "Triangle.hpp"
#include "unit_test_framework.hpp"

TEST(test_triangle_basic) {
  Triangle t(3, 4, 5);
  ASSERT_EQUAL(t.area(), 6);
  ASSERT_EQUAL(t.get_a(), 3);  // must use get and set functions
  t.set_a(4);
  ASSERT_EQUAL(t.get_a(), 4);
}

TEST_MAIN()
```

16.8 Member-Initialization Order

Member variables are always initialized in the order in which they are declared in the class. This is the case regardless if some members are initialized at the declaration point and others are not, or if a constructor's member-initializer list is out of order:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in)
    : b(b_in), c(c_in), a(a_in) {    // this ordering is ignored
  }
};
```

Here, `a` is initialized first to `a_in`, then `b` to `b_in`, then `c` to `c_in`; the ordering in the member-initializer list is ignored. Some compilers will generate a warning if the order differs between the member declarations and the member-initializer list:

```
$ g++ --std=c++17 -Wall Triangle.cpp
Triangle.cpp:8:16: warning: field 'c' will be initialized after field 'a'
      [-Wreorder]
    : b(b_in), c(c_in), a(a_in) {    // this ordering is ignored
```

16.9 Delegating Constructors

When a class has multiple constructors, it can be useful to invoke one constructor from another. This allows us to avoid code duplication, and it also makes our code more maintainable by reducing the number of places where we hardcode implementation details.

In order to delegate to another constructor, we must do so in the member-initializer list. The member-initializer list must consist solely of the call to the other constructor:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  // EFFECTS: Initializes this to be an equilateral triangle with
  //          the given side length.
  Triangle(double side_in)
    : Triangle(side_in, side_in, side_in) {} // delegate to 3-argument constructor

  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {}
};
```

The delegation must be in the member-initializer list. If we invoke a different constructor from within the body, it does not do delegation; rather, it creates a new, temporary object and then throws it away:

```
Triangle(double side_in) {   // default initializes members
  Triangle(side_in, side_in, side_in);  // creates a new Triangle object that
                                        // lives in the activation record for
                                        // this constructor
}
```

CHAPTER

SEVENTEEN

DERIVED CLASSES AND INHERITANCE

In addition to encapsulation and information hiding, C++ classes provide two features that are fundamental to *object-oriented programming*:

- *Inheritance*: the ability for a class to reuse the interface or functionality of another class.
- *Subtype polymorphism*: the ability to use an object of a more specific type where an object of a more general type is expected.

We will discuss inheritance today, deferring *subtype polymorphism* until next time.

To motivate the concept of inheritance, consider the following definitions of `Chicken` and `Duck` ADTs:

```
class Chicken {
public:
  Chicken(const string &name_in)
    : age(0), name(name_in),
      roads_crossed(0) {
    cout << "Chicken ctor" << endl;
  }

  string get_name() const {
    return name;
  }

  int get_age() const {
    return age;
  }

  void cross_road() {
    ++roads_crossed;
  }

  void talk() const {
    cout << "bawwk" << endl;
  }

private:
  int age;
  string name;
  int roads_crossed;
};
```

```
class Duck {
public:
  Duck(const string &name_in)
    : age(0), name(name_in),
      num_ducklings(0) {
    cout << "Duck ctor" << endl;
  }

  string get_name const {
    return name;
  }

  int get_age() const {
    return age;
  }

  void have_babies() {
    num_ducklings += 7;
  }

  void talk() const {
    cout << "quack" << endl;
  }

private:
  int age;
  string name;
  int num_ducklings;
};
```

The two ADTs are nearly identical – both have `age` and `name` member variables, with their corresponding get functions, and both have a `talk()` member function that makes the appropriate chicken or duck sound. In terms of differences, chickens tend to cross roads (since they don't fly very well), so we keep track of how often they do that. On the other hand, ducks are often accompanied by their ducklings, so we keep track of how many they have.

Intuitively, it makes sense for `Chicken` and `Duck` to share a lot of functionality, since chickens and ducks are both types of birds.

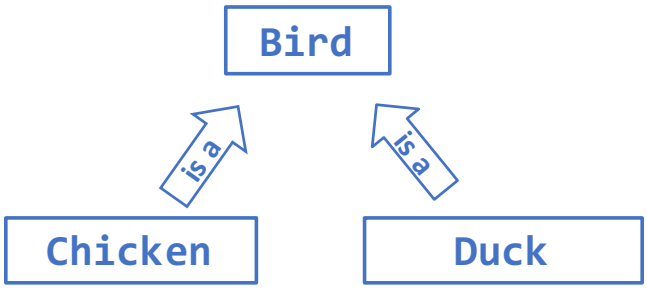

Figure 17.1: Relationship between different kinds of birds.

This "is-a" relationship, where a `Chicken` is a `Bird` and a `Duck` is also a `Bird`, can be encoded with *inheritance*. We write `Bird` as a *base class*, then write `Chicken` and `Duck` as classes that *inherit* or *derive* from `Bird`. We place the common functionality in `Bird`, which then gets inherited by the derived classes.

```
class Bird {
public:
  Bird(const string &name_in)
    : age(0), name(name_in) {
    cout << "Bird ctor" << endl;
  }

  string get_name() const {
    return name;
  }

  int get_age() const {
    return age;
  }

  void have_birthday() {
    ++age;
  }

  void talk() const {
    cout << "tweet" << endl;
  }

private:
  int age;
  string name;
};
```

Here, all birds have a name and an age, and the generic sound a bird makes is "tweet".

For more a more specific kind of bird, we obtain the functionality of the base class by deriving from it:

```
class Chicken : public Bird {
  ...
};
```

The syntax for deriving from a base class is to put a colon after the name of the derived class, then the `public` keyword, then the name of the base class. This results in *public inheritance*, where it is part of the interface of `Chicken` that it derives from `Bird`. Without the `public` keyword, it would be *private inheritance*[1], where it is an implementation detail and not part of the interface that `Chicken` derives from `Bird`.

Now that `Chicken` derives from `Bird`, it inherits the functionality of the `Bird` class, and the public interface of `Bird` is also supported by `Chicken`:

```
Chicken c("Myrtle");
c.have_birthday();
cout << c.get_name() << " " << c.get_age();  // prints Myrtle 1
```

Functionality that is specific to a particular kind of bird goes in the class for that bird:

```
class Chicken : public Bird {
public:
  ...

  void cross_road() {
    ++roads_crossed;
  }

  void talk() const {
    cout << "bawwk" << endl;
  }

private:
  int roads_crossed;
};
```

Here, we have the additional data member `roads_crossed`; `name` and `age` are inherited from `Bird`. (They are not directly accessible from the `Chicken` class, however, since they are private. We will come back to this later.) Figure 17.2 illustrates the layout of `Bird` and `Chicken` in memory.

```
int main() {
  Bird big_bird("Big Bird");
  Chicken myrtle("Myrtle");
}
```

The memory of `Chicken` actually consists of a `Bird` piece, plus the additional `Chicken` members. Thus, the data members defined by `Bird` are also included in a `Chicken`.

In order to properly initialize a derived-class object, its constructor must ensure that its base-class subobject is also appropriately initialized. The base class may have private member variables, which cannot be accessed from the derived class, so the derived class does not initialize the inherited members directly. Instead, it invokes a base-class constructor in the member-initializer list of its own constructors:

[1] As with member accessibility, the default inheritance is public if the `struct` keyword is used to define a class type and private if the `class` keyword is used. With private inheritance, the (non-private) inherited members are private to the derived class, so that outside code may not access those members. There is also *protected inheritance*, where classes that derive from the derived class can access the (non-private) inherited members of the derived class, but outside code cannot.

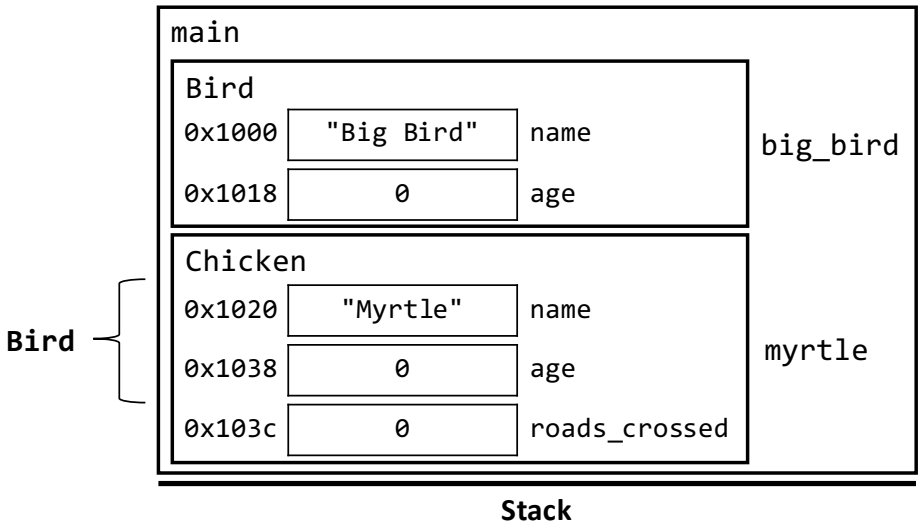

Figure 17.2: The layout of a derived-class object contains a subset that matches the layout of a base-class object.

```
class Chicken : public Bird {
public:
  Chicken(const string &name_in)
    : Bird(name_in), roads_crossed(0) {
    cout << "Chicken ctor" << endl;
  }

  ...

private:
  int roads_crossed;
};
```

In C++, a derived-class constructor **always** invokes a constructor for the base class. If an explicit invocation does not appear in the member-initializer list, there is an implicit call to the default constructor. If the base class has no default constructor, an error results:

```
class Chicken : public Bird {
public:
  Chicken(const string &name_in)
    : roads_crossed(0) {  // ERROR: implicit call to Bird(), which doesn't exist
    cout << "Chicken ctor" << endl;
  }

  ...
};
```

For completeness, the following is an implementation of `Duck` as a derived class of `Bird`:

```
class Duck : public Bird {
public:
  Duck(const string &name_in)
```

(continues on next page)

(continued from previous page)

```
    : Bird(name_in), num_ducklings(0) {
    cout << "Duck ctor" << endl;
  }

  void have_babies() {
    num_ducklings += 7;
  }

  void talk() const {
    cout << "quack" << endl;
  }

private:
  int num_ducklings;
};
```

By writing `Bird` and deriving from it in both `Chicken` and `Duck`, we have avoided duplication of the shared functionality.

17.1 Ordering of Constructors and Destructors

We have already seen that in most cases, a constructor is invoked when a class-type object is created. Similarly, a *destructor* is invoked when a class-type object's lifetime is over. For a local variable, this is when the variable goes out of scope. The following illustrates an example:

```
class Foo {
public:
  Foo() {                          // constructor
    cout << "Foo ctor" << endl;
  }

  ~Foo() {                         // destructor
    cout << "Foo dtor" << endl;
  }
};

void func() {
  Foo x;
}

int main() {
  cout << "before call" << endl;
  func();
  cout << "after call" << endl;
}
```

The class `Foo` has a custom destructor, written as `~Foo()`, which runs when a `Foo` object is dying. Here, we just have both the constructor and destructor print messages to standard out. The following is printed when the code is run:

```
before call
Foo ctor
```

(continues on next page)

(continued from previous page)

```
Foo dtor
after call
```

We will cover *destructors in more detail* later in the course. For now, we concern ourselves solely with the order in which constructors and destructors execute when we have derived classes.

When there are multiple objects that are constructed and destructed, C++ follows a "socks-and-shoes" ordering: when we put on socks and shoes in the morning, we put on socks first, then our shoes. In the evening, however, when we take them off, we do so in the reverse order: first our shoes, then our socks. In the case of a derived class, C++ will always construct the base-class subobject before initializing the derived-class pieces. Destruction is in the reverse order: first the derived-class destructor runs, then the base-class one. The following illustrates this order:

```
class Bird {
public:
  Bird(const string &name_in)
    : age(0), name(name_in) {
    cout << "Bird ctor" << endl;
  }

  ~Bird() {
    cout << "Bird dtor" << endl;
  }

  ...
};

class Chicken : public Bird {
public:
  Chicken(const string &name_in)
    : Bird(name_in), roads_crossed(0) {
    cout << "Chicken ctor" << endl;
  }

  ~Chicken() {
    cout << "Chicken dtor" << endl;
  }

  ...
};

int main() {
  cout << "construction:" << endl;
  Chicken myrtle("Myrtle");
  cout << "destruction:" << endl;
}
```

The following results from running the code:

```
construction:
Bird ctor
Chicken ctor
destruction:
Chicken dtor
```

(continues on next page)

(continued from previous page)

```
Bird dtor
```

When creating a `Chicken` object, the invocation of the `Bird` constructor is the first thing that happens in the `Chicken` constructor. This is true regardless of whether or not an explicit call to the `Bird` constructor appears, and regardless of ordering of the member-initializer list. Then the rest of the `Chicken` constructor runs. When a `Chicken` object is dying, first the code in the `Chicken` destructor runs. Then the code in the `Bird` destructor automatically runs. (It is generally erroneous to invoke the base-class destructor explicitly, since the compiler always does so implicitly.)

The following code creates both a `Chicken` and a `Duck` object:

```
int main() {
  cout << "construction:" << endl;
  Chicken myrtle("Myrtle");
  Duck donald("Donald");
  cout << "destruction:" << endl;
}
```

Assuming that `Duck` has a destructor that prints out `"Duck dtor"`, the code prints the following:

```
construction:
Bird ctor
Chicken ctor
Bird ctor
Duck ctor
destruction:
Duck dtor
Bird dtor
Chicken dtor
Bird dtor
```

We see the same ordering in `Duck` construction and destruction. Furthermore, we see that because `myrtle` is constructed before `donald`, it is destructed after `donald` – socks-and-shoes ordering here as well.

17.2 Name Lookup and Hiding

When a member access is applied to an object, the compiler follows a specific process to look up the given name:

- The compiler starts by looking for a member with that name in the compile-time or *static type* of the object. (We will discuss *static and dynamic types* next time.)
- If no member with that name is found, the compiler repeats the process on the base class of the given type. If the type has no base class, a compiler error is generated.
- If a member with the given name is found, the compiler then checks whether or not the member is accessible and whether the member can be used in the given context.[2] If not, a compiler error is generated – the lookup process does not proceed further.

As an example, consider the following member accesses:

```
Chicken myrtle("Myrtle");
myrtle.get_age();
```

(continues on next page)

[2] If name lookup finds a set of overloaded functions within the same class, the compiler performs overload resolution to determine which overload is the most appropriate. We will discuss *function overloading* in more detail next time.

(continued from previous page)

```
myrtle.talk();
myrtle.age;
myrtle.undefined;
```

- For `myrtle.get_age()`, the compiler first looks for a `get_age` member defined in the `Chicken` class. Since `Chicken` does not define such a member, the compiler looks for a `get_age` member in its base `Bird` class. There is indeed a `get_age` member in `Bird`, so the compiler then checks that the member is accessible and is a function that can be called with no arguments. These checks succeed, so the lookup process terminates successfully.
- For `myrtle.talk()`, the compiler looks for a `talk` member in `Chicken`. There is one, so it then checks to make sure it is accessible and is a function that can be called with no arguments. This succeeds, and at runtime, it is `Chicken::talk()` that is called. The function `Bird::talk()` is *hidden*, since the name lookup process never gets to it.
- For `myrtle.age`, the compiler looks for an `age` member in `Chicken`. There is none, so the compiler looks in `Bird`. There is such a member, so the compiler checks whether it is accessible from the given context. The member is private, so this check fails, and the compiler reports an error.
- For `myrtle.undefined`, the compiler looks for an `undefined` member in `Chicken`. There is none, so the compiler looks in `Bird`. There is no such member, and `Bird` has no base class, so the compiler reports an error.

As another example, consider the following code:

```
class Base {
public:
  int x;
  void foo(const string &s);
};

class Derived : public Base {
public:
  void x();
  void foo(int i);
};

int main() {
  Derived d;
  int a = d.x;
  d.foo("hello");
}
```

When looking up `d.x`, the compiler finds a member `x` in `Derived`. However, it is a member function, which cannot be assigned to an `int`. Thus, the compiler reports an error – it does not consider the hidden `x` that is defined in `Base`.

Similarly, when looking up `d.foo`, the compiler finds a member `foo` in `Derived`. Though it is a function, it cannot be called with a string literal, so the compiler reports an error. Again, the compiler does not consider the `foo` that is defined in `Base`; that member is hidden by the `foo` defined in `Derived`.

To summarize, C++ does not consider the context in which a member is used until after its finds a member of the given name. This is in contrast to some other languages, which consider context in the lookup process itself.

On occasion, we wish to access a hidden member rather than the member that hides it. The most common case is when a derived-class version of a function calls the base-class version as part of its functionality. The following illustrates this in `Chicken`:

```
class Chicken : public Bird {
public:
  ...

  void talk() const {
    if (age >= 1) {              // ERROR: age is private in Bird
      cout << "bawwk" << endl;
    } else {
      // baby chicks make more of a tweeting rather than clucking noise
      Bird::talk();              // call Bird's version of talk()
    }
  }
};
```

By using the scope-resolution operator, we are specifically asking for the `Bird` version of `talk()`, enabling access to it even though it is hidden.[3]

The code above does have a problem: it accesses the `age` member of `Bird`, which, though it is not hidden, is private and so not accessible to `Chicken`. There are two solutions to this problem:

- We can declare `age` to be *protected*, which allows derived classes of `Bird` to access the member but not the outside world.
- We can use the public `get_age()` function instead.

The former strategy is not desirable; since the member variable `age` is an implementation detail, making it protected exposes implementation details to the derived classes. Instead, we should use a get function to abstract the implementation detail. We could choose to make such a function protected, so that it is part of the interface that derived classes have access to but not the outside world. For `age`, we already have a public get function, so we can just use that:

```
class Chicken : public Bird {
public:
  ...

  void talk() const {
    if (get_age() >= 1) {
      cout << "bawwk" << endl;
    } else {
      // baby chicks make more of a tweeting rather than clucking noise
      Bird::talk();              // call Bird's version of talk()
    }
  }
};
```

[3] Interestingly, we can apply the scope-resolution operator as part of dot or arrow syntax in order to access the base-class version: `myrtle.Bird::talk()`. However, this is very uncommon in real code, and we will not do it at all in this course.

CHAPTER

EIGHTEEN

POLYMORPHISM

The word *polymorphism* literally means "many forms." In the context of programming, polymorphism refers to the ability of a piece of code to behave differently depending on the context in which it is used. Appropriately, there are several forms of polymorphism:

- *ad hoc polymorphism*, which refers to function overloading
- *parametric polymorphism* in the form of templates
- *subtype polymorphism*, which allows a derived-class object to be used where a base-class object is expected

The unqualified term "polymorphism" usually refers to subtype polymorphism.

We proceed to discuss ad hoc and subtype polymorphism, deferring *parametric polymorphism* until later.

18.1 Function Overloading

Ad hoc polymorphism refers to *function overloading*, which is the ability to use a single name to refer to many different functions in a single scope. C++ allows both top-level functions and member functions to be overloaded. The following is an example of overloaded member functions:

```
class Base {
public:
  void foo(int a);
  int foo(string b);
};

int main() {
  Base b;
  b.foo(42);
  b.foo("test");
}
```

When we invoke an overloaded function, the compiler resolves the function call by comparing the types of the arguments to the parameters of the candidate functions and finding the best match. The call `b.foo(42)` calls the member function `foo()` with parameter `int`, since 42 is an `int`. The call `b.foo("test")` calls the function with parameter `string` – `"test"` actually has type `const char *`, but a `string` parameter is a better match for a `const char *` than `int`.

In C++, functions can only be overloaded when defined within the same scope. If functions of the same name are defined in a different scope, then those that are defined in a closer scope hide the functions defined in a further scope:

```
class Derived : public Base {
public:
```

(continues on next page)

(continued from previous page)

```
  int foo(int a);
  double foo(double b);
};

int main() {
  Derived d;
  d.foo("test"); // ERROR
}
```

When handling the member access `d.foo`, under the *name-lookup process* we saw last time, the compiler finds the name `foo` in `Derived`. It then applies function-overload resolution; however, none of the functions with name `foo` can be invoked on a `const char *`, resulting in a compile error. The functions inherited from `Base` are not considered, since they were defined in a different scope.

Function overloading requires the signatures of the functions to differ, so that overload resolution can choose the overload with the most appropriate signature. Here, "signature" refers to the function name and parameter types – the return type is not part of the signature and is not considered in overload resolution.

```
class Person {
public:
  void greet();
  void greet(int x);            // OK
  void greet(string x);         // OK
  void greet(int x, string s);  // OK
  void greet(string s, int x);  // OK
  bool greet();                 // ERROR: signature the same as the first overload
  void greet() const;           // OK: implicit this parameter different
};
```

For member functions, the `const` keyword after the parameter list is part of the signature – it changes the implicit `this` parameter from being a pointer to non-const to a pointer to const. Thus, it is valid for two member-function overloads to differ solely in whether or not they are declared as `const`.

18.2 Subtype Polymorphism

Subtype polymorphism allows a derived-class object to be used where a base-class one is expected. In order for this to work, however, we need indirection. Consider what happens if we directly copy a `Chicken` object into a `Bird`:

```
int main() {
  Chicken chicken("Myrtle");
  // ...
  Bird bird = chicken;
}
```

While C++ allows this, the value of a `Chicken` does not necessarily fit into a `Bird` object, since a `Chicken` has more member variables than a `Bird`. The copy above results in object slicing – the members defined by `Bird` are copied, but the `Chicken` ones are not, as illustrated in Figure 18.1.

To avoid slicing, we need indirection through a reference or a pointer, so that we avoid making a copy:

```
Bird &bird_ref = chicken;
Bird *bird_ptr = &chicken;
```

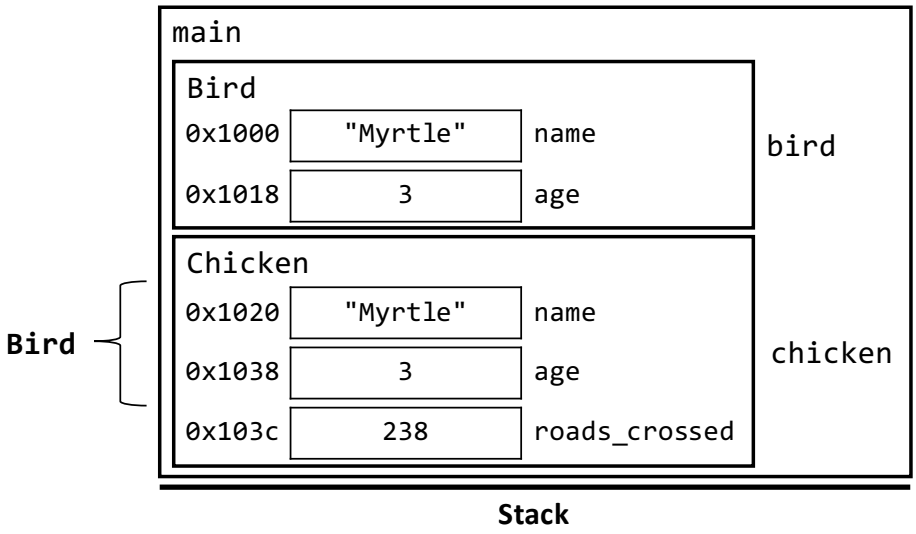

Figure 18.1: Object slicing copies only the members defined by the base class.

The above initializes `bird_ref` as an alias for the `chicken` object. Similarly, `bird_ptr` is initialized to hold the address of the `chicken` object. In either case, a copy is avoided.

C++ allows a reference or pointer of a base type to refer to an object of a derived type. It allows implicit *upcasts*, which are conversions that go upward in the inheritance hierarchy, such as from `Chicken` to `Bird`, as in the examples above. On the other hand, implicit *downcasts* are prohibited:

```
Chicken &chicken_ref = bird_ref;   // ERROR: implicit downcast
Chicken *chicken_ptr = bird_ptr;   // ERROR: implicit downcast
```

The implicit downcasts are prohibited by C++ even though `bird_ref` and `bird_ptr` actually refer to `Chicken` objects. In the general case, they can refer to objects that aren't of `Chicken` type, such as `Duck` or just plain `Bird` objects. Since the conversions may be unsafe, they are disallowed by the C++ standard.

While implicit downcasts are prohibited, we can do *explicit* downcasts with `static_cast`:

```
Chicken &chicken_ref = static_cast<Chicken &>(bird_ref);
Chicken *chicken_ptr = static_cast<Chicken *>(bird_ptr);
```

These conversions are unchecked at runtime, so we need to be certain from the code that the underlying object is a `Chicken`.

In order to be able to bind a base-class reference or pointer to a derived-class object, the inheritance relationship must be accessible. From outside the classes, this means that the derived class must publicly inherit from the derived class. Otherwise, the outside world is not allowed to take advantage of the inheritance relationship. Consider this example:

```
class A {
};

class B : A {     // default is private when using the class keyword
};

int main() {
  B b;
```

(continues on next page)

(continued from previous page)

```
  A *a_ptr = &b;  // ERROR: inheritance relationship is private
}
```

This results in a compiler error:

```
main.cpp:9:16: error: cannot cast 'B' to its private base class 'A'
    A *a_ptr = &b;  // ERROR: inheritance relationship is private
               ^
main.cpp:4:13: note: implicitly declared private here
  class B : A {     // default is private when using the class keyword
            ^
1 error generated.
```

18.3 Static and Dynamic Binding

Subtype polymorphism allows us to pass a derived-class object to a function that expects a base-class object:

```
void Image_init(Image* img, istream& is);

int main() {
  Image *image = /* ... */;
  istringstream input(/* ... */);
  Image_init(image, input);
}
```

Here, we have passed an `istringstream` object to a function that expects an `istream`. Extracting from the stream will use the functionality that `istringstream` defines for extraction.

Another common use case is to have a container of base-class pointers, each of which points to different derived-class objects:

```
void all_talk(Bird *birds[], int length) {
  for (int i = 0; i < length; ++i) {
    array[i]->talk();
  }
}

int main() {
  Chicken c1 = /* ... */;
  Duck d = /* ... */;
  Chicken c2 = /* ... */;
  Bird *array[] = { &c1, &d, &c2 };
  all_talk(array, 3);
}
```

Unfortunately, given the way we defined the `talk()` member function of `Bird` last time, this code will not use the derived-class versions of the function. Instead, all three calls to `talk()` will use the `Bird` version:

```
$ ./main.exe
tweet
tweet
tweet
```

In the invocation `array[i]->talk()`, the declared type of the *receiver*, the object that is receiving the member-function call, is different from the actual runtime type. The declared or *static type* is `Bird`, while the runtime or *dynamic type* is `Chicken` when `i == 0`. This disparity can only exist when we have indirection, either through a reference or a pointer.

For a particular member function, C++ gives us the option of either *static binding* where the compiler determines which function to call based on the static type of the receiver, or *dynamic binding*, where the program also takes the dynamic type into account. The default is static binding, since it is more efficient and can be done entirely at compile time.

In order to get dynamic binding instead, we need to declare the member function as *virtual* in the base class:

```
class Bird {
  ...
  virtual void talk() const {
    cout << "tweet" << endl;
  }
};
```

Now when we call the `all_talk()` function above, the compiler will use the dynamic type of the receiver in the invocation `array[i]->talk()`:

```
$ ./main.exe
bawwk
quack
bawwk
```

The `virtual` keyword is necessary in the base class, but optional in the derived classes. It can only be applied to the declaration within a class; if the function is subsequently defined outside of the class, the definition cannot include the `virtual` keyword:

```
class Bird {
  ...
  virtual void talk() const;
};

void Bird::talk() const {
  cout << "bawwk" << endl;
}
```

18.4 dynamic_cast

With dynamic binding, the only change we need to make to our code is to add the `virtual` keyword when declaring the base-class member function. No changes are required to the actual function calls (e.g. in `all_talk()`).

Consider an alternative to dynamic binding, where we manually check the runtime type of an object to call the appropriate function. In C++, a `dynamic_cast` conversion checks the dynamic type of the receiver object:

```
Chicken chicken("Myrtle");
Bird *b_ptr = &chicken;
Chicken *c_ptr = dynamic_cast<Chicken *>(b_ptr);
if (c_ptr) {  // check for null
  // do something chicken-specific
}
```

If the dynamic type is not actually a `Chicken`, the conversion results in a null pointer. Otherwise, it results in the address of the `Chicken` object. Thus, we can check for null after the conversion to determine if it succeeded.

There are two significant issues with `dynamic_cast`:

1. It generally results in messy and unmaintainable code. For instance, we would need to modify `all_talk()` as follows to use `dynamic_cast` rather than dynamic binding:

```
void all_talk(Bird * birds[], int length) {
  for (int i = 0; i < length; ++i) {
    Chicken *c_ptr = dynamic_cast<Chicken*>(birds[i]);
    if (c_ptr) {
      c_ptr->talk();
    }
    Duck *d_ptr = dynamic_cast<Duck*>(birds[i]);
    if (d_ptr) {
      d_ptr->talk();
    }
    Eagle *e_ptr = dynamic_cast<Eagle*>(birds[i]);
    if (e_ptr) {
      e_ptr->talk();
    }
    ...
  }
}
```

   We would need a branch for every derived type of `Bird`, and we would have to add a new branch every time we wrote a new derived class. The code also takes time that is linear in the number of derived classes.

2. In C++, `dynamic_cast` can only be applied to classes that are *polymorphic*, meaning that they define at least one virtual member function. Thus, we need to use `virtual` one way or another.

Code that uses `dynamic_cast` is usually considered to be poorly written. Almost universally, it can be rewritten to use dynamic binding instead.

18.5 Member Lookup Revisited

We have already seen that when a member is accessed on an object, the compiler first looks in the object's class for a member of that name before proceeding to its base class. With indirection, the following is the full lookup process:

1. The compiler looks up the member in the **static type** of the receiver object, using the *lookup process* we discussed before (starting in the class itself, then looking in the base class if necessary). It is an error if no member of the given name is found in the static type or its base types.
2. If the member found is an overloaded function, then the arguments of the function call are used to determine which overload is called.
3. If the member is a variable or non-virtual function (including *static member functions*, which we will see later), the access is statically bound at compile time.
4. If the member is a virtual function, the access uses dynamic binding. At runtime, the program will look for a **function of the same signature**, starting at the dynamic type of the receiver, then proceeding to its base type if necessary.

As indicated above, dynamic binding requires two conditions to be met to use the derived-class version of a function:

- The member function found at compile time using the static type must be virtual.

- The derived-class function must have the same signature as the function found at compile time.

When these conditions are met, the derived-class function *overrides* the base-class one – it will be used instead of the base-class function when the dynamic type of the receiver is the derived class. If these conditions are not met, the derived-class function *hides* the base-class one – it will only be used if the static type of the receiver is the derived class.

As an example, consider the following class hierarchy:

```
class Top {
public:
  int f1() const {
    return 1;
  }

  virtual int f2() const {
    return 2;
  }
};

class Middle : public Top {
public:
  int f1() const {
    return 3;
  }

  virtual int f2() const {
    return 4;
  }
};

class Bottom : public Middle {
public:
  int f1() const {
    return 5;
  }

  virtual int f2() const {
    return 6;
  }
};
```

Each class has a non-virtual `f1()` member function; since the function is non-virtual, the derived-class versions hide the ones in the base classes. The `f2()` function is virtual, so the derived-class ones override the base-class versions.

The following are some examples of invoking these functions:

```
int main() {
  Top top;
  Middle mid;
  Bottom bot;
  Top *top_ptr = &bot;
  Middle *mid_ptr = &mid;

  cout << top.f2() << endl;       // prints 2
```

(continues on next page)

(continued from previous page)

```
  cout << mid.f1() << endl;       // prints 3
  cout << top_ptr->f1() << endl;  // prints 1
  cout << top_ptr->f2() << endl;  // prints 6
  cout << mid_ptr->f2() << endl;  // prints 4
  mid_ptr = &bot;
  cout << mid_ptr->f1() << endl;  // prints 3
  cout << mid_ptr->f2() << endl;  // prints 6
}
```

We discuss each call in turn:

- There is no indirection in the calls `top.f1()` and `mid.f1()`, so there is no difference between the static and dynamic types of the receivers. The former calls the `Top` version of `f1()`, resulting in 2, while the latter calls the `Middle` version, producing 3.
- The static type of the receiver in `top_ptr->f1()` and `top_ptr->f2()` is `Top`, while the dynamic type is `Bottom`. Since `f1()` is non-virtual, static binding is used, resulting in 1. On the other hand, `f2()` is virtual, so dynamic binding uses the `Bottom` version, producing 6.
- In the first call to `mid_ptr->f2()`, both the static and dynamic type of the receiver is `Middle`, so `Middle`'s version is used regardless of whether `f2()` is virtual. The result is 4.
- The assignment `mid_ptr = &bot` changes the dynamic type of the receiver to `Bottom` in calls on `mid_ptr`. The static type remains `Middle`, so the call `mid_ptr->f1()` results in 3. The second call to `mid_ptr->f2()`, however, uses dynamic binding, so the `Bottom` version of `f2()` is called, resulting in 6.

18.6 The `override` Keyword

A common mistake when attempting to override a function is to inadvertently change the signature, so that the derived-class version hides rather than overrides the base-class one. The following is an example:

```
class Chicken : public Bird {
  ...
  virtual void talk() {
    cout << "bawwk" << endl;
  }
}

int main() {
  Chicken chicken("Myrtle");
  Bird *b_ptr = &chicken;
  b_ptr->talk();
}
```

This code compiles, but it prints `tweet` when run. Under the lookup process above, the program looks for an override of `Bird::talk()` at runtime. However, no such override exists – `Chicken::talk()` has a different signature, since it is not `const`. Thus, the dynamic lookup finds `Bird::talk()` and calls it instead.

Rather than having the code compile and then behave incorrectly, we can ask the compiler to detect bugs like this with the `override` keyword. Specifically, we can place the `override` keyword after the signature of a member function to let the compiler know we intended to override a base-class member function. If the derived-class function doesn't actually do so, the compiler will report this:

```
class Chicken : public Bird {
  ...
  void talk() override {
    cout << "bawwk" << endl;
  }
}
```

Here, we have removed the `virtual` keyword, since it is already implied by `override` – only a virtual function can be overridden, and the "virtualness" is inherited from the base class. Since we are missing the `const`, the compiler reports the following:

```
main.cpp:39:15: error: non-virtual member function marked 'override' hides
      virtual member function
  void talk() override {
              ^
main.cpp:20:16: note: hidden overloaded virtual function 'Bird::talk' declared
      here: different qualifiers (const vs none)
  virtual void talk() const {
               ^
1 error generated.
```

Adding in the `const` fixes the issue:

```
class Chicken : public Bird {
  ...
  void talk() const override {
    cout << "bawwk" << endl;
  }
}

int main() {
  Chicken chicken("Myrtle");
  Bird *b_ptr = &chicken;
  b_ptr->talk();
}
```

The code now prints `bawwk`.

18.7 Abstract Classes and Interfaces

In some cases, there isn't enough information in a base class to define a particular member function, but we still want that function to be part of the interface provided by all its derived classes. In the case of `Bird`, for example, we may want a `get_wingspan()` function that returns the average wingspan for a particular kind of bird. There isn't a default value that makes sense to put in the `Bird` class. Instead, we declare `get_wingspan()` as a *pure virtual function*, without any implementation in the base class:

```
class Bird {
  ...
  virtual int get_wingspan() const = 0;
};
```

The syntax for declaring a function as pure virtual is to put `= 0;` after its signature. This is just syntax – we aren't actually setting its value to 0.

Since `Bird` is now missing part of its implementation, we can no longer create objects of `Bird` type. The `Bird` class is said to be *abstract*. We can still declare `Bird` references and pointers, however, since that doesn't create a `Bird` object. We can then have such references and pointers refer to derived-class objects:

```
Bird bird("Big Bird");      // ERROR: Bird is abstract
Chicken chicken("Myrtle");  // OK, as long as Chicken is not abstract
Bird &bird_ref = chicken;   // OK
Bird *bird_ptr = &chicken;  // OK
```

In order for a derived class to not be abstract itself, it must provide implementations of the pure virtual functions in its base classes:

```
class Chicken : public Bird {
  ...
  int get_wingspan() const override {
    return 20;  // inches
  }
};
```

With a virtual function, a base class provides its derived classes with the option of overriding the function's behavior. With a pure virtual function, the base class **requires** its derived classes to override the function, since the base class does not provide an implementation itself. If a derived class fails to override the function, the derived class is itself abstract, and objects of that class cannot be created.

We can also define an *interface*, which is a class that consists only of pure virtual functions. Such a class provides no implementation; rather, it merely defines the interface that must be overridden by its derived classes. The following is an example:

```
class Shape {
public:
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  virtual void scale(double s) = 0;
};
```

With subtype polymorphism, we end up with two use cases for inheritance:

- *implementation inheritance*, where a derived class inherits functionality from a base class
- *interface inheritance*, where a derived class inherits the interface of its base class, but not necessarily any implementation

Deriving from a base class that isn't an interface results in both implementation and interface inheritance. Deriving from an interface results in just interface inheritance. The latter is useful to work with a hierarchy of types through a common interface, using a base-class reference or pointer, even if the derived types don't share any implementation.

Part III

Containers and Dynamic Memory

CHAPTER

NINETEEN

CONTAINERS AND ITERATORS

Apologies – this section has not been written yet.

19.1 Range-Based For Loops

A *range-based for loop* is a special syntax for iterating over sequences that support traversal by iterator. Rather than writing out each piece of the traversal, we can have the compiler generate it for us:

```
vector<int> vec = { 1, 2, 3, 4, 5 };
for (int item : vec) {
  cout << item << endl;
}
```

The syntax of a range-based for loop is:

```
for (<type> <variable> : <sequence>)
  <body>
```

In the example above, the variable is named `item` and has type `int`, and `vec` is the sequence over which the loop iterates. The compiler automatically converts this into a traversal by iterator:

```
for (auto it = vec.begin(); it != vec.end(); ++it) {
  int item = *it;
  cout << item << endl;
}
```

The variable `item` is initialized in each iteration from an element in the sequence. Then the code in the body of the range-based for loop follows.

The following loop attempts to set every element in the vector to the value 42:

```
for (int item : vec) {
  item = 42;              // attempt to set element to 42
}
```

However, this loop does not actually modify any elements. To understand why, let us take a look at its translation into a traversal by iterator:

```
for (auto it = vec.begin(); it != vec.end(); ++it) {
  int item = *it;
  item = 42;
}
```

The loop is actually modifying a **copy** of each element rather than an element itself. To avoid making a copy, we can declare the loop variable to be a reference:

```
for (int &item : vec) {
  item = 42;              // actually set element to 42
}
```

This translates to:

```
for (auto it = vec.begin(); it != vec.end(); ++it) {
  int &item = *it;
  item = 42;
}
```

This successfully modifies each element to have the value 42.

Range-based for loops are often used in combination with `auto`, as in the following:

```
vector<int> vec = { 1, 2, 3, 4, 5 };

for (auto &item : vec) {
  item = 42;
}

for (auto item : vec) {
  cout << item << endl;
}
```

The first loop declares `item` as a reference, so that it aliases an element in the sequence. The second loop does not declare `item` as a reference, so it produces a copy of an element in the sequence. The following is the translation of both loops:

```
vector<int> vec = { 1, 2, 3, 4, 5 };

for (auto it = vec.begin(); it != vec.end(); ++it) {
  int &item = *it;                                    // alias
  item = 42;
}

for (auto it = vec.begin(); it != vec.end(); ++it) {
  int item = *it;                                     // copy
  item = 42;
}
```

With a range-based for loop, we can simplify `print_all()`:

```
template <typename Sequence>
void print_all(const Sequence &sequence) {
  for (auto &item : sequence) {
    cout << item << endl;
  }
}
```

We can call it on any sequence[1] that supports traversal by iterator, as long as the element type can be inserted into an output stream:

[1] We can even call `print_all()` on an array, as long as it still is an array:

```
vector<string> vec = { "hello", "world!" };
print_all(vec);
```

19.1.1 Iterating over a Map

The standard-library map defines an iterator that produces key-value pairs upon dereference. The following is an example that constructs a map, counting how many times each unique word occurs in a vector. It then iterates over the map to print out the counts.

```
void print_word_counts(const std::vector<std::string> &words) {
  std::map<std::string, int> word_counts;

  // Each time a word is seen, add 1 to its entry in the map
  for (const auto &word : words) {
    word_counts[word] += 1;
  }

  // Print out the results by iterating through the map.
  for (const auto &key_value : word_counts) {     // key-value pairs
    const auto &word = key_value.first;
    const auto &count = key_value.second;
    cout << word << "occurred " << count << " times." << endl;
  }
}
```

When incrementing a word's count, we do not need to check whether the word is in the map; if not, the map automatically inserts the word into the map with a value-initialized count of zero, which we then increment to one.

We use range-based for loops to iterate over both the vector and the map, declaring a type-deduced reference to each element. A reference avoids making a copy, which is nontrivial for strings. The iteration over the map produces key-value pairs (`std::pair<std::string, int>`), and we access the `first` and `second` members to obtain the word and count, respectively.

```
string arr[] = { "hello", "world!" };
print_all(arr);
```

This is because a range-based for loop actually calls `std::begin()` and `std::end()` on the sequence, rather than directly calling the respective member functions. For arrays, `std::begin()` and `std::end()` return pointers, while for class types, they call the `begin()` and `end()` member functions, respectively.

CHAPTER

TWENTY

TIME COMPLEXITY

Apologies – this section has not been written yet.

CHAPTER

TWENTYONE

ARRAYS

As we mentioned previously, C++ has *several different categories of objects*, including atomic, array, and class-type objects. An *array* is simple collection of objects, built into C++ and many other languages. An array has the following properties:

- It has a fixed size, set when the array is created. This size never changes as long as the array is alive.
- An array holds *elements* that are of the same type.
- The elements of an array are stored in a specific order, with the index of the first element being 0.
- The elements are stored contiguously in memory, one after another.
- Accessing any element of an array takes constant time, regardless of whether the element is at the beginning, middle, or end of the array.

An array variable can be declared by placing square brackets to the right of the variable name, with a compile-time constant between the brackets, denoting the number of elements. For example, the following declares `array` to be an array of four `int` elements:

```
int array[4];
```

The following uses a named constant to declare `array2` to be an array of four `int`s:

```
const int SIZE = 4;
int array2[SIZE];
```

In both cases, we did not provide an explicit initialization. Thus, `array` and `array2` are default initialized by default initializing each of their elements. Since their elements are of atomic type `int`, they are default initialized to undefined values.

We can explicitly initialize an array with an *initializer list*, a list of values in curly braces:

```
int array[4] = { 1, 2, 3, 4 };
```

This initializes the element at index 0 to 1, the element at index 1 to 2, and so on.

If the initializer list contains fewer values than the size of the array, the remaining array elements are implicitly initialized. For atomic elements, these remaining elements are initialized to zero values[1]. Thus, the following initializes the first two elements of `array2` to 1 and 2, respectively, and the last two elements to 0:

```
int array2[4] = { 1, 2 };
```

The following results in every element in `array3` being initialized to 0:

[1] This is referred to as *value initialization*, which is distinct from default initialization.

```
int array3[4] = {};
```

Here, we have provided an empty initializer list, so that the first zero elements (i.e. none of them) are explicitly initialized while the remaining elements (i.e. all of them) are implicitly initialized to 0.

If the size of the array is the same as the size of the initializer list, we can elide the size of the array in its declaration:

```
int array[] = { 1, 2, 3, 4 };
```

Figure 21.1 illustrates the layout of `array` in memory.

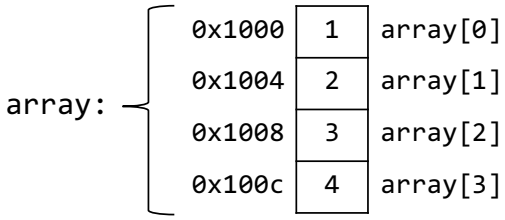

Figure 21.1: Layout of an array in memory.

This diagram assumes that an `int` takes up four bytes in memory, which is the case on most modern machines.

Individual array elements can be accessed with square brackets, with an index between the brackets. Indexing starts at 0, up through the size of the array minus one. For example, the following increments each element in `array` by one and prints out each resulting value:

```
for (int i = 0; i < 4; ++i) {
  ++array[i];
  cout << array[i] << endl;
}
```

Arrays can be composed with other kinds of objects, such as structs. The following is an array of three `Person` elements:

```
struct Person {
  string name;
  int age;
  bool is_ninja;
};

Person people[3];
```

Figure 21.2 shows the layout of this array in memory.

The following is a struct that contains an array as a member, and its layout is shown in Figure 21.3:

```
struct Matrix {
  int width;
  int height;
  int data[6];
};

int main() {
  Matrix matrix;
  ...
}
```

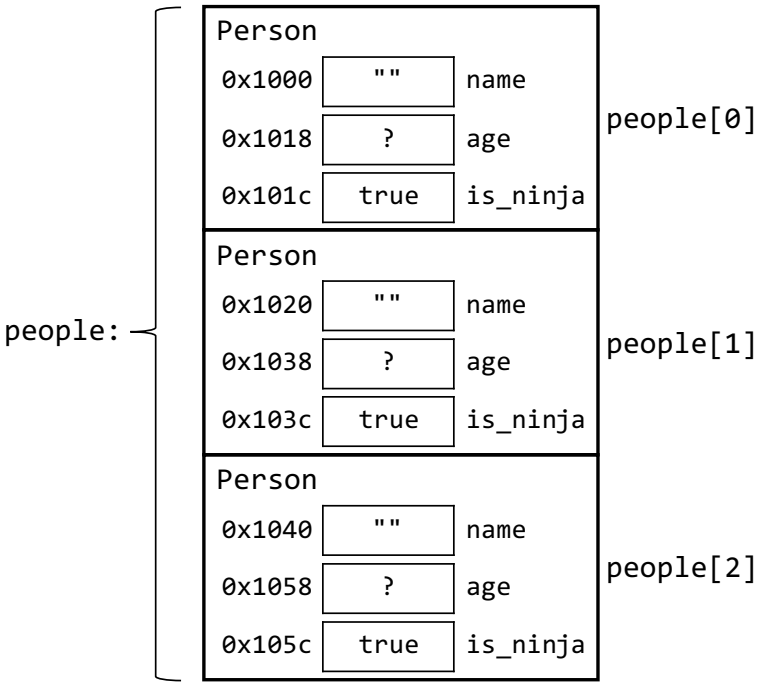

Figure 21.2: An array of class-type objects.

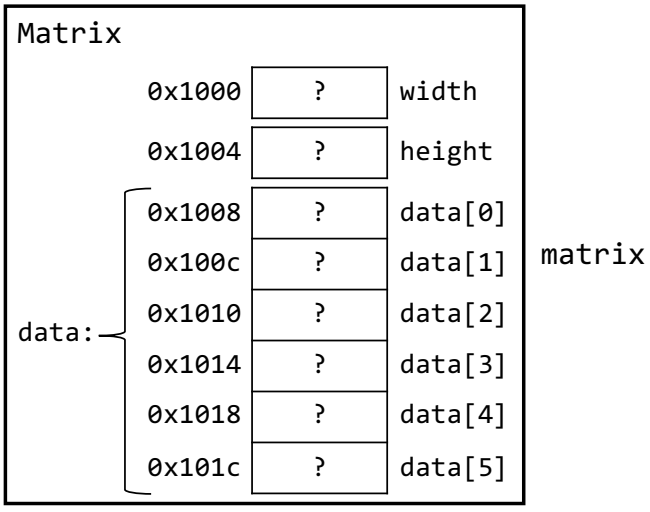

Figure 21.3: A class-type object with an array member.

21.1 Arrays and Pointers

Arrays in C++ are objects. However, in most contexts, there isn't a value associated with an array as a whole[2]. The individual elements (if they are not of array type), have values, but not the array as a whole. Instead, when we use an array in a context where a value is required, the compiler converts the array into a pointer to the first element in the array:

```
int array[] = { 1, 2, 3, 4 };
cout << &array[0] << endl; // prints 0x1000 assuming the figure above
cout << array << endl;     // prints 0x1000 assuming the figure above
*array = -1;
cout << array[0] << endl;  // prints -1
```

In this example, assuming the layout in Figure 21.1 where the first element is at address `0x1000`, printing `array` to standard output just prints out the address `0x1000` – it converts `array` to a pointer to its first element, and it is the pointer's value that is then printed. Similarly, dereferencing the array first turns it into a pointer to the first element, followed by the dereference that gives us the first element itself.

The tendency of arrays to *decay* into pointers results in significant limitations when using an array. For instance, we cannot assign one array to another – the right-hand side of an assignment requires a value, which in the case of an array will become a pointer, which is then incompatible with the left-hand side array:

```
int arr1[4] = { 1, 2, 3, 4 };
int arr2[4] = { 5, 6, 7, 8 };
arr2 = arr1;  // error: LHS is an array, RHS is a pointer
```

As discussed before, by default, C++ passes parameters by value. This is also true if the parameter is an array. Since an array decays to a pointer when its value is required, this implies that an array is passed by value as a pointer to its first element. Thus, an array parameter to a function is actually equivalent to a pointer parameter, regardless of whether or not the parameter includes a size:

```
void func1(int arr[4]); // parameter equivalent to int *arr

void func2(int arr[5]); // parameter equivalent to int *arr

void func3(int arr[]);  // parameter equivalent to int *arr

void func4(int *arr);

int main() {
  int arr1[4] = { 1, 2, 3, 4 };
  int arr2[5];
  int x = -3;

  func1(arr1);  // OK: arr1 turns into pointer, as does parameter of func1
  func2(arr1);  // OK: arr1 turns into pointer, as does parameter of func2
                //     compiler ignores size in func2 parameter
  func3(arr1);  // OK: arr1 turns into pointer, as does parameter of func3
  func4(arr1);  // OK: arr1 turns into pointer, matches parameter of func4
```

(continues on next page)

[2] The system of values in C++ is very complicated and beyond the scope of this course. In the context of this course, we use the term *value* to mean something called an *rvalue* in programming-language terms. There are a handful of ways to construct an array rvalue in C++, but none that we will encounter in this course.

(continued from previous page)

```
  func1(arr2);  // OK: arr2 turns into pointer, as does parameter of func1
                //     compiler ignores size in func1 parameter
  func2(arr2);  // OK: arr2 turns into pointer, as does parameter of func2
  func3(arr2);  // OK: arr2 turns into pointer, as does parameter of func3
  func4(arr2);  // OK: arr2 turns into pointer, matches parameter of func4

  func1(&x);    // OK: parameter of func1 turns into pointer
  func2(&x);    // OK: parameter of func2 turns into pointer
  func3(&x);    // OK: parameter of func3 turns into pointer
  func4(&x);    // OK: matches parameter of func4
}
```

This means that a function that takes an array as a parameter cannot guarantee that the argument value corresponds to an array of matching size, or even that it is a pointer into an array. Instead, we need another mechanism for passing size information to a function; we will come back to this momentarily.

21.2 Pointer Arithmetic

C++ supports certain arithmetic operations on pointers:

- An integral value can be added to or subtracted from a pointer, resulting in a pointer that is offset from the original one.
- Two pointers can be subtracted, resulting in an integral value that is the distance between the pointers.

Pointer arithmetic is in terms of **number of elements** rather than **number of bytes**. For instance, if an `int` takes up four bytes of memory, then adding 2 to an `int *` results in a pointer that is two `int`s forward in memory, or a total of eight bytes:

```
int array[] = { 4, 3, 2, 1 };
int *ptr1 = array;      // pointer to first element
int *ptr2 = &array[2];  // pointer to third element
int *ptr3 = ptr1 + 2;   // pointer to third element
int *ptr4 = array + 2;  // pointer to third element
++ptr1;                 // move pointer to second element
```

In initializing `ptr4`, `array` is converted to a pointer to its first element, since the + operator requires a value, and the result is two `int`s forward in memory, producing a pointer to the third element. The last line increments `ptr1` to point to the next `int` in memory. The result is shown in Figure 21.4.

The following demonstrates subtracting pointers:

```
cout << ptr2 - ptr1 << endl;  // prints 1
```

Since `ptr2` is one `int` further in memory than `ptr`, the difference `ptr2 - ptr` is 1.

Pointer arithmetic is one reason why each C++ type has its own pointer type – in order to be able to do pointer arithmetic, the compiler needs to use the size of the pointed-to type, so it needs to know what that type is. For example, implementations generally represent `double` objects with eight bytes, so adding 2 to a `double *` moves 16 bytes forward in memory. In general, for a pointer of type `T *`, adding `N` to it moves `N * sizeof(T)` bytes forward in memory[3].

Pointers can also be compared with the comparison operators, as in the following using the pointers declared above:

[3] `sizeof` is an operator that can be applied to a type to obtain the number of bytes used to represent that type. When applied to a type, the parentheses are mandatory (e.g. `sizeof(int)`). The operator can also be applied to a value, in which case it results in the size of the compile-time type of that value. Parentheses are not required in this case (e.g. `sizeof 4` or `sizeof x`).

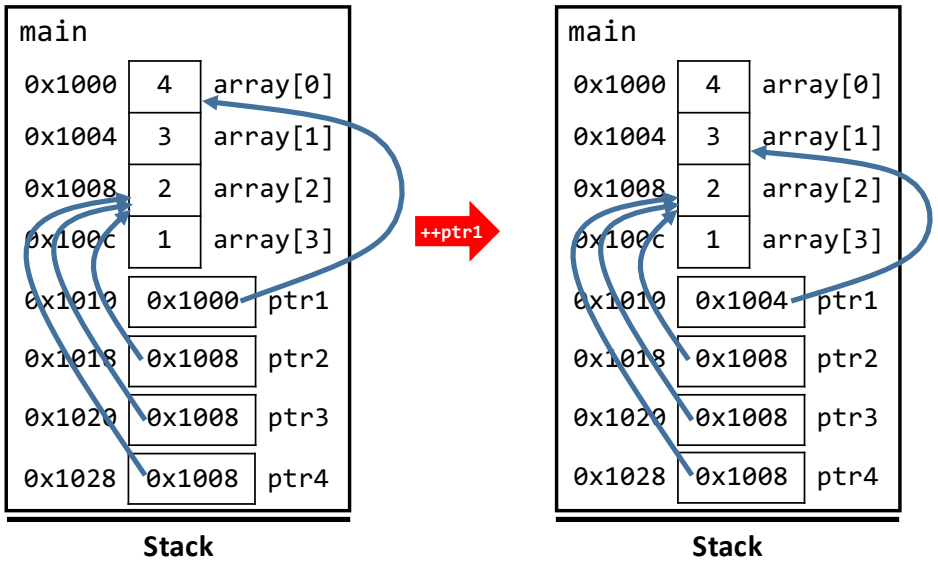

Figure 21.4: Pointer arithmetic is in terms of whole objects, not bytes.

```
cout << (ptr1 == ptr2) << endl;         // false (prints as 0)
cout << (ptr2 == ptr3) << endl;         // true (prints as 1)
cout << (ptr1 < ptr2) << endl;          // true
cout << (*ptr1 < *ptr2) << endl;        // false (compares element values)
++ptr1;
cout << (ptr1 == ptr2) << endl;         // true
cout << (array == &array[0]) << endl;   // true (LHS turns into pointer)
```

Arithmetic is generally useful only on pointers to array elements, since only array elements are guaranteed to be stored contiguously in memory. Similarly, comparisons are generally only well-defined on pointers into the same array or on pointers constructed from arithmetic operations on the same pointer.

21.3 Array Indexing

Array indexing in C++ is actually implemented using pointer arithmetic. If one of the operands to the subscript (`[]`) operator is an array and the other is integral, then the operation is equivalent to pointer arithmetic followed by a dereference:

```
int arr[4] = { 1, 2, 3, 4 };
cout << *(arr + 2) << endl;  // prints 3: arr+2 is pointer to 3rd element
cout << arr[2] << endl;      // prints 3: equivalent to *(arr + 2)
cout << 2[arr] << endl;      // prints 3: equivalent to *(2 + arr);
                             //           but don't do this!
```

Thus, if `arr` is an array and `i` is integral, then `arr[i]` is equivalent to `*(arr + i)`:

1. The subscript operation requires the value of `arr`, so it turns into a pointer to its first element.
2. Pointer arithmetic is done to produce a pointer `i` elements forward in memory.
3. The resulting pointer is dereferenced, resulting in the element at index `i`.

Because the subscript operation is equivalent to pointer arithmetic, it can be applied to a pointer equally as well:

```
int arr[4] = { 1, 2, 3, 4 };
int *ptr = arr + 1;      // pointer to second element
cout << ptr[2] << endl;  // prints 4: constructs a pointer that is 2
                         //           elements forward in memory, then
                         //           dereferences that
```

There are several implications of the equivalence between array indexing and pointer arithmetic. First, it is what makes array access a constant time operation – no matter the index, accessing an element turns into a single pointer addition followed by a single dereference. The equivalence is also what makes passing arrays by value work – the result is a pointer, which we can still subscript into since it just does pointer arithmetic followed by a dereference. Finally, it allows us to work with subsets of an array. For instance, the following code prints out just the middle elements of an array:

```
void print_array(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    cout << array[i] << " ";
  }
}

int main() {
  int array[4] = { 3, -1, 5, 2 };
  print_array(arr + 1, 2);      // prints out just -1 5
}
```

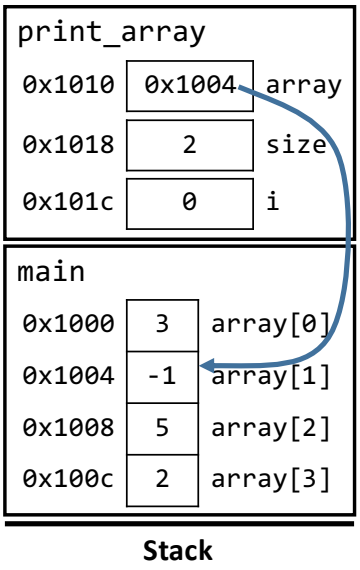

Figure 21.5: Passing a subset of an array to a function.

The `print_array()` function receives a pointer to the array's second element as well as a size of 2, as shown in Figure 21.5. Thus, it only prints out the second and third elements; as far as the function knows, it is working with an array of size 2 that starts at the address `0x1004`.

21.4 More on Array Decay

An array only decays into a pointer when its value is required. When an array object's value is not required, it does not decay into a pointer. For example, the address-of (&) operator requires an object but not its value – thus, applying & to an array produces a pointer to the whole array, not a pointer to an individual element nor a pointer to a pointer[4].

Another example is applying the `sizeof` operator to an array. The operator produces the size of the whole array in bytes[5], as opposed to applying it to a pointer, which just produces the size of a pointer (generally eight bytes on modern systems):

```
int x = 42;
int arr[5] = { 1, 2, 3, 4, 5 };
int *ptr = arr;
cout << sizeof x << endl;    // 4 (on most machines)
cout << sizeof arr << endl;  // 20
cout << sizeof ptr << endl;  // 8
```

Once an array has turned into a pointer, the resulting pointer loses all information about the size of the array, or even that it is a pointer into an array. Thus, we need another mechanism for keeping track of the size of an array, such as when we pass the array to a function (if it is passed by value, it turns into a pointer which retains no information about the array's size).

21.5 The End of an Array

If a program dereferences a pointer that goes past the bounds of the array, the result is undefined behavior[6]. If we are lucky, the program will crash, indicating we did something wrong and giving us an opportunity to debug it. In the worst case, the program may compute the right result when we run it on our own machine but misbehave when run on a different platform (e.g. the autograder).

There are two general strategies for keeping track of where an array ends.

1. Keep track of the length separately from the array. This can be done with either an integer size or by constructing a pointer that is just past the end of an array (by just adding the size of the array to a pointer to the array's first element).
2. Store a special *sentinel value* at the end of the array, which allows an algorithm to detect that it has reached the end.

The first strategy is what we used in defining the `print_array()` function above. As demonstrated there, the stored size may be smaller than the size of the array, resulting in the function operating on a subset of the array.

The second strategy requires there to be a special value that can be reserved to indicate the end of the array, and that we are assured will not occur as a real element. It is how built-in (C-style) strings (as opposed to C++ `std::string`) are implemented, though we do not cover the details here. Instead, we will return to the sentinel strategy when we implement *linked data structures*.

[4] A pointer to an array of 4 `int`s can be declared using the syntax `int (*ptr_to_arr)[4];`. The address value stored in a pointer to an array is generally the same address as that of the array's first element.

[5] Thus, the expression `sizeof array / sizeof *array` recovers the number of elements, as long as `array` is still an array.

[6] Constructing a pointer that is out of bounds is not a problem; we often construct pointers that are just past the end of an array, as we will see in a moment. It is **dereferencing** such a pointer that results in undefined behavior.

21.6 Array Traversal

The `print_array()` function above also demonstrates how to traverse through an array using an index that starts at 0 up to the size of the array, exclusive. The following is another example:

```
int const SIZE = 3;                    // constant to represent array size
int array[SIZE] = { -1, 7, 3 };
for (int i = 0; i < SIZE; ++i) {
  cout << *(array + i) << endl;                // using pointer arithmetic
  cout << array[i] << endl;                    // using subscript (better)
}
```

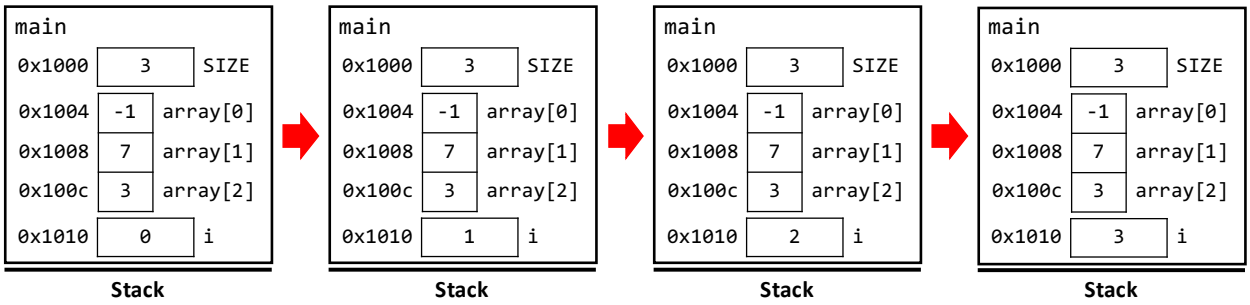

Figure 21.6: Traversal by index uses an index to walk through an array.

This pattern of accessing elements is called *traversal by index* – we use an integer index in the range $[0, SIZE)$ where $SIZE$ is the size of the array, and we use the index to obtain the corresponding element. We can use the subscript operator or do pointer arithmetic ourselves. (The former is generally considered better, since it is more familiar and clearer to most programmers. However, you will often see both `arr[0]` and `*arr` used to access the first element.) The actual syntax we use is irrelevant to the pattern – what makes this traversal by index is that we use an integer index to access the array, and we traverse through the elements by modifying the index.

Another pattern we can use is *traversal by pointer*, which walks a pointer across the elements of an array:

```
int const SIZE = 3;                    // constant to represent array size
int array[SIZE] = { -1, 7, 3 };
int *end = array + SIZE;               // pointer just past the end of arr
for (int *ptr = array; ptr < end; ++ptr) {     // walk pointer across arr
  cout << *ptr << endl;                   // dereference to obtain element
}
```

Here, we start by constructing a pointer that is just past the end of the array: the last element is at `arr + SIZE - 1`, so we need to end our traversal when the pointer we are using reaches `arr + SIZE`. We then use another pointer that starts at the first element, dereference it to obtain an element, and then increment it to move on to the next element. The syntax we use to dereference an element is irrelevant to the pattern (it can be `*ptr` or `ptr[0]`) – what makes this traversal by pointer is that we use a pointer to each element to access the array, and we traverse through the elements by modifying that pointer.

Traversal by index is the more common pattern when working with general arrays. However traversal by pointer is a special case of traversal by iterator, which we saw previously. We will shortly see that traversal by iterator/pointer allows us to write algorithms that work on both library containers and arrays. Thus, both the traversal-by-index and traversal-by-pointer patterns are important to programming in C++.

Aside from providing us insight about memory and how objects are stored, arrays are a fundamental abstraction that can be used to build more complex abstractions. We proceed to see how to use arrays to build data structures such as

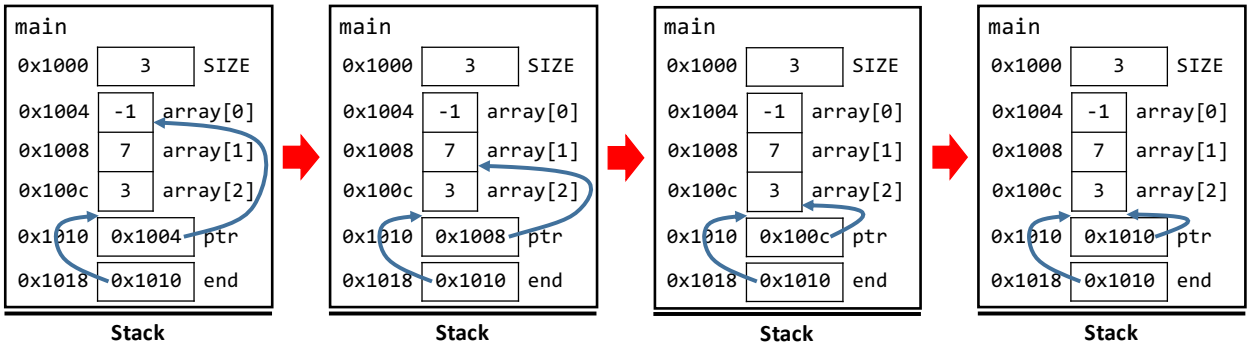

Figure 21.7: Traversal by pointer uses a pointer to walk through an array.

vectors and sets.

21.7 Arrays and `const`

Since an array does not have a value of its own, it cannot be assigned to as a whole – we saw previously that a compile error would result, since we cannot obtain an array value to place on the right-hand side of the assignment. Thus, it is also not meaningful for an array itself to be const either.

Similar to a reference, an array may not be const itself, but its elements may be:

```
const double arr[4] = { 1.1, 2.3, -4.5, 8 };
arr[2] = 3.1;          // ERROR -- attempt to assign to const object
```

The declaration `const double arr[4]` is read inside out as "`arr` is an array of four constant doubles." The elements can be initialized through an initializer list, but they may not be modified later through assignment.

If an array is a member of a class-type object, the array elements inherit the "constness" of the object itself. For example, consider the following:

```
struct Foo {
  int num;
  int *ptr;
  int arr[4];
};
```

Like any `const` object, a `const Foo` must be initialized upon creation:

```
int main() {
  int x = 3;
  const Foo foo = { 4, &x, { 1, 2, 3, 4 } };
  ...
}
```

The array member can be initialized using its own initializer list, which is the same syntax for initializing a local array variable.

As we saw *previously*, attempting to modify `foo.num` or `foo.ptr` results in a compiler error. The same is true for the elements of `foo.arr`:

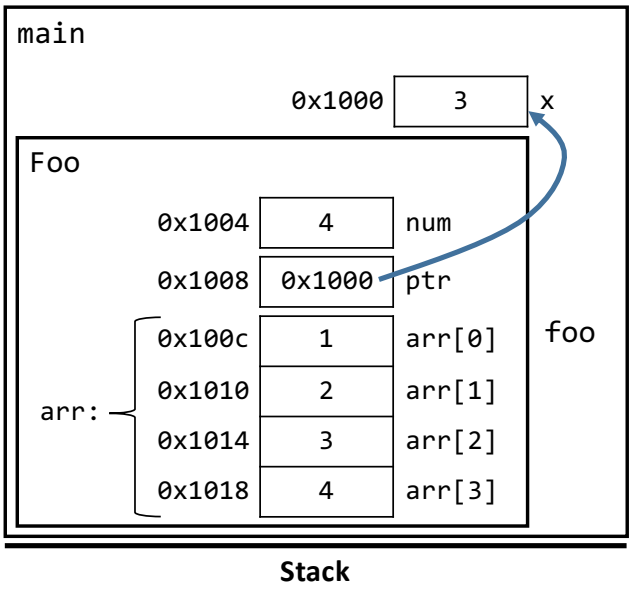

Figure 21.8: Contents of a `Foo` object. Declaring the object as `const` only prohibits modifications to the subobjects contained within the memory for the object.

```
foo.arr[0] = 2;    // ERROR
```

Sine the array element is a subobject of a const object, it cannot be modified.

CHAPTER

TWENTYTWO

ARRAY-BASED CONTAINERS

A *container* is an abstract data type whose purpose is to hold objects of some other type. We have already seen two examples of containers: built-in arrays and vectors from the standard library. Both are containers that hold elements in a particular order, indexed starting from 0. A built-in array has a fixed size, while a vector can grow and shrink.

Let's define our own container ADT, using built-in arrays as our building block. Rather than an ordered sequence, we will build a set abstraction, which represents an unordered collection of unique elements. We will start with a container that holds integers, generalizing it to arbitrary element types next time.

In designing our ADT, we will use the following process:

1. Determine which operations our ADT will support.
2. Write some code that uses our ADT. This will helps us figure out what the right interface should be, and it will serve as a test case. (Ideally, we would also write unit tests for each part of the public interface, following the pattern of test-driven development.)
3. Define the ADT's public interface, including both function signatures and documentation.
4. Come up with a data representation. Define member variables and determine the representation invariants.
5. Write definitions for the ADT's constructors and member functions, making sure that they adhere to the representation invariants.

22.1 Set Operations

The following are the operations our set will support:

- Creating an empty set.
- Inserting a value into a set. We will allow an existing item to be inserted – it will just do nothing.
- Removing a value from a set. We will allow a nonexistent item to be removed – it will just do nothing.
- Check if a value is contained in a set.
- Count the number of items in a set.
- Print a character representation of a set to an output stream.

22.2 Code Example

Here is a small program that uses a set:

```
int main() {
  IntSet set;
  set.insert(7);
  set.insert(32);
  set.insert(32);

  cout << "Size: " << set.size() << endl;                    // prints 2
  set.print(cout);              // prints { 7, 32 } in some arbitrary order

  set.insert(42);
  set.remove(32);
  set.remove(32);

  cout << "Contains 32? " << set.contains(32) << endl;  // prints 0 (false)
  cout << "Contains 42? " << set.contains(42) << endl;   // prints 1 (true)
}
```

22.3 Public Interface

The code example leads to the following public interface:

```
class IntSet {
public:
  // Maximum size of a set.
  static const int MAX_SIZE = 10;

  // EFFECTS:  Initializes this set to be empty.
  IntSet();

  // REQUIRES: size() < MAX_SIZE
  // MODIFIES: *this
  // EFFECTS:  Adds value to the set, if it isn't already in the set.
  void insert(int value);

  // MODIFIES: *this
  // EFFECTS:  Removes value from the set, if it is in the set.
  void remove(int value);

  // EFFECTS:  Returns whether value is in the set.
  bool contains(int value) const;

  // EFFECTS:  Returns the number of elements.
  int size() const;

  // EFFECTS:  Prints out the set in an arbitrary order.
  void print(std::ostream &os) const;
};
```

The public interface includes a default constructor and member functions to insert or remove an item, check if an item is in the set, obtain its size, and print a representation of the set to a stream. The latter three functions do not modify the set, so they are declared as `const`, meaning that the `this` pointer will be a pointer to const. The compiler will then enforce that those functions do not modify any member variables.

22.4 `static` Data Members

The public interface also includes a constant, `MAX_SIZE`, that denotes the maximum size of a set. We define the constant inside the `IntSet` class. This distinguishes the constant from other constants of the same name; there may be some other ADT with a `MAX_SIZE`. It also makes use of encapsulation, which allows all of an ADT's data and functionality to be defined together within a class.

The constant is declared `static`, which has a specific meaning when used in a class – it indicates that the given member is not associated with an object of the class, but with the class as a whole. For a member variable, this means that there is a single copy of that variable in the program, located in static storage. Without the `static` keyword, each object of the class would have its own copy, located in the memory for that object. The `static` keyword also allows a member variable to be a compile-time constant, provided it is also declared as `const` (or `constexpr`) and initialized from an expression that can be computed at compile time.[1]

A static member can be accessed by name from within the class, the same as a non-static member. It can be accessed from outside the class with the scope-resolution operator:

```
int main() {
  IntSet set;
  for (int i = 0; i < IntSet::MAX_SIZE; ++i) {
    set.insert(i);
  }
}
```

We need a compile-time constant because we will use a member variable that is a built-in array as part of our data representation.[2] In C++, a local or member variable that is an array must have a size that is known to the compiler. In the case of a local variable, the compiler needs to know how big it is so that the program can create activation records of the correct size. For a member variable, the compiler needs to know its size so that it can determine the size of the object as a whole. Pointer arithmetic (including array accesses) requires the compiler to know the size of an object at compile time, since it is in terms of whole objects rather than with respect to individual bytes.

22.5 Data Representation

We rely on existing types to represent the data of our `IntSet`. For the elements themselves, we use a built-in array, with `MAX_SIZE` as its capacity. However, our set will start out empty, after which we can add and remove elements as we please. Thus, the size of the set will generally not be the same as the size of the array, and we need a separate member variable to keep track of the number of elements in the set. We will use an `int` for this member.

```
class IntSet {
  ...
private:
```

(continues on next page)

[1] Member variables that are `static` need not be compile-time constants. However, a member variable that isn't a compile-time constant cannot be initialized at the point of declaration (unless it is declared as `inline` in C++17 and onward). The variable must be defined outside of the class, generally in a source (`.cpp`) file. As with a member function, the scope-resolution operator is used to define a member variable outside the class (e.g. `int IntSet::foo = 3;`).

[2] In the future, we will see how to eliminate the fixed-size restriction by storing the elements indirectly using a dynamic array. We will see that this raises issues of *memory management* and learn how to resolve them.

(continued from previous page)

```
  int elements[MAX_SIZE];
  int num_elements;
};
```

Now that we've determined a data representation, we need to figure out what values correspond to valid sets. We use representation invariants to specify the set of valid values.

For `num_elements`, a negative value does not make sense, since a set cannot have a negative size. At the same time, our sets do not have enough space to store more than `MAX_SIZE` elements, so that places an upper bound on `num_elements`. The invariants for `num_elements` are thus `0 <= num_elements && num_elements <= MAX_SIZE`.

For `elements`, we need to know where in the array the set's elements are actually stored. We will store them at the beginning of the array, so one invariant is that the first `num_elements` items in the array are the ones in the set. The set abstraction prohibits duplicate elements, so another invariant is that there are no duplicates among the first `num_elements` items in `elements`.

We document the representation invariants where we define our representation:

```
class IntSet {
  ...
private:
  int elements[MAX_SIZE];
  int num_elements;
  // INVARIANTS:
  // 0 <= num_elements && num_elements <= MAX_SIZE
  // the first num_elements items in elements are the items in the set
  // the first num_elements items in elements contain no duplicates
};
```

22.6 size_t

The `size_t` type represents only nonnegative integers, and the C++ standard guarantees that it is large enough to hold the size of any object. Since it does not represent negative integers, we could have used it for `num_elements` rather than defining `0 <= num_elements` as a representation invariant. We would then also use `size_t` as the return type for `size()`, which is what the standard-library containers do.

C++ has integral types that are *unsigned*, which do not represent negative numbers, and `size_t` is just one example. The `int` type is *signed* (unless it is preceded by the `unsigned` keyword). Unsigned integers can lead to subtle programming errors. The following is an example that compares a signed and unsigned integer:

```
int main() {
  size_t s = 3;
  int i = -1;
  cout << (s < i) << endl;   // prints 1 (true)
}
```

Perhaps non-intuitively, the comparison `s < i` is true, even though `s` is positive but `i` is negative. This is because the compiler converts `i` to a `size_t` in doing the comparison, and the data that represents a negative signed integer actually represents a very large unsigned integer.

Another common error is incorrectly iterating backwards over a sequence:

```
int main() {
  vector<int> vec = { 1, 2, 3 };
  for (size_t i = vec.size() - 1; i >= 0; --i) {
    cout << vec[i] << endl;
  }
}
```

This code goes off the end of the vector, resulting in undefined behavior. The reason is that a `size_t` is always greater than or equal to 0, and when `i` is equal to 0 and decremented, its value wraps around to the largest possible `size_t` value. The following iteration correctly avoids this problem:

```
int main() {
  vector<int> vec = { 1, 2, 3 };
  for (size_t i = vec.size(); i > 0; --i) {
    cout << vec[i - 1] << endl;
  }
}
```

The loop iterates from `vec.size()` down to 1, offsetting by 1 when indexing into the vector. The following equivalent code makes use of *postfix decrement* to avoid the offset in the index expression:

```
int main() {
  vector<int> vec = { 1, 2, 3 };
  for (size_t i = vec.size(); i-- > 0;) {
    cout << vec[i] << endl;
  }
}
```

Given the pitfalls of unsigned integers, many C++ programmers avoid them, using signed integers instead. We too will generally stick to signed integers.

22.7 Implementation

We proceed to define the constructor and member functions of `IntSet`, defining them as they would appear in a source file, outside the class definition.

We require only a default constructor for `IntSet`. However, we cannot rely on a compiler-generated implicit one: it would default initialize `num_elements` to an undefined value, violating our representation invariants. Thus, we need to define our own default constructor.[3]

```
IntSet::IntSet() : num_elements(0) {}
```

We need not do anything with `elements`. It gets default initialized to contain undefined values, but that doesn't violate our representation invariants; since `num_elements` is 0, the invariants for `elements` are trivially satisfied.

We proceed to define `contains()`. The representation invariants tell us that the valid items are the first `num_elements` in `elements`. They can be stored in any order, so we need to iterate through them to see if the value is in the set.

```
bool IntSet::contains(int value) const {
  for (int i = 0; i < num_elements; ++i) {
```

(continues on next page)

[3] Alternatively, we can initialize `num_elements` to 0 at the point it is declared. The implicitly defined default constructor would then be sufficient; it would still have an empty member-initializer list, but the compiler uses the inline initialization of a member variable if it is not initialized in the member-initializer list.

(continued from previous page)

```
    if (elements[i] == value) {
      return true;
    }
  }
  return false;
}
```

When we find an item, we can return immediately without checking the rest of the elements. On the other hand, if we get through all the elements without finding the given value, it is not in the set so we return false.

For inserting into the set, we first assert the requires clause, which ensures that our invariant of `num_elements <= MAX_SIZE` is never violated. We also must check whether the value is in the set, since adding it again would violation the invariant of no duplicates. If the value is not in the set, the simplest way to insert it is to place it at index `num_elements` and increment that variable. This meets the invariant that the first `num_elements` items in the array are the values in the set.

```
void IntSet::insert(int value) {
  assert(size() < MAX_SIZE);
  if (!contains(value)) {
    elements[num_elements] = value;
    ++num_elements;   // we could use num_elements++ in the line above instead
  }
}
```

To remove an element, we first need to check whether the value is in the set. If so, we need to know its location in the array, which would require iterating through the elements to find it. Rather than repeating this algorithm in both `contains()` and `remove()`, we define a private helper function to do so and call it in both places.

```
class IntSet {
  ...
private:
  int indexOf(int value) const;
};

int IntSet::indexOf(int value) const {
  for (int i = 0; i < num_elements; ++i) {
    if (elements[i] == value) {
      return i;
    }
  }
  return -1;
}
```

Rather than returning whether or not the value is in the set, `indexOf()` returns its actual index if it is there. If not, it returns an invalid index; we use -1, since indices start at 0. We can then write `contains()` as follows:

```
bool IntSet::contains(int value) const {
  return indexOf(value) != -1;
}
```

We proceed to define `remove()`:

```
void IntSet::remove(int value) {
  int index = indexOf(value);
  if (index != -1) {
    elements[index] = elements[num_elements - 1];
    --num_elements;
  }
}
```

In order to actually remove the item, we need to place another item at its location. The simplest solution that meets the representation invariants is to copy over the last item and then decrement `num_elements`, which ensures that the first `num_elements` items are the values in the set.

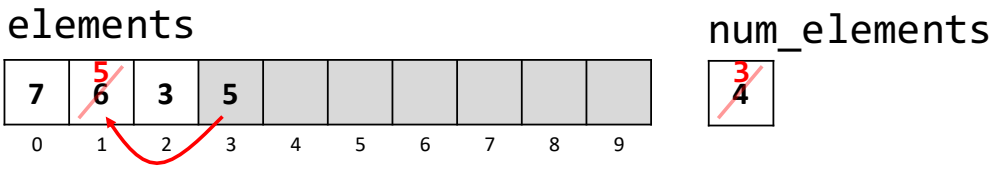

Figure 22.1: Removing an element from an unordered set can be done by copying the last element into the position vacated by the removed element.

The remaining member functions are defined as follows:

```
int IntSet::size() const {
  return num_elements;
}

void IntSet::print(std::ostream &os) const {
  os << "{ ";
  for (int i = 0; i < num_elements; ++i) {
    os << elements[i] << " ";
  }
  os << "}";
}
```

22.8 Sorted Representation

The representation of `IntSet` places no restrictions on the ordering of elements in the array. This simplifies the implementation of `insert()` and `remove()`, but it requires `contains()` (and `indexOf()`) to iterate over every element in the worst case.

An alternate representation would be to require that the set elements are stored in sorted, increasing order. The member variables remain the same – it is the representation invariants that change.

```
class SortedIntSet {
  ...
private:
  int elements[MAX_SIZE];
  int num_elements;
  // INVARIANTS:
```

(continues on next page)

(continued from previous page)

```
  // 0 <= num_elements && num_elements <= MAX_SIZE
  // the first num_elements items in elements are the items in the set
  // the first num_elements items in elements contain no duplicates
  // the first num_elements items are in sorted, increasing order
};
```

To insert an item, we can no longer just put it after the existing elements, since the new value may be smaller than existing values. Instead, we need to store it at its appropriate sorted position. This requires moving existing elements out of the way if necessary. Our algorithm will start after the last element and repeatedly:

- Compare the item to the left with the value we are inserting.
- If the item to the left is less than the new value, the new value is placed at the current position, and we are done.
- If the item is greater than the new value, then the old item is moved one position to the right, and we repeat the process one position over to the left.
- If no more items are to the left, we place the new value at the current position (i.e. the beginning of the array).

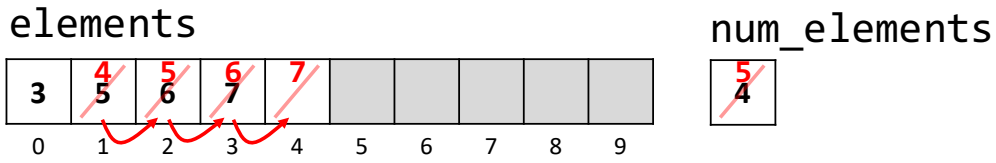

Figure 22.2: Inserting into an ordered set requires shifting existing elements to make room for the new element at the proper location.

```
void SortedIntSet::insert(int value) {
  assert(size() < MAX_SIZE);
  if (!contains(value)) {
    int index = num_elements;
    while (index > 0 && elements[index - 1] > value) {
      elements[index] = elements[index - 1];
      --index;
    }
    elements[index] = value;
    ++num_elements;
  }
}
```

Similarly, to remove an item, we must shift the remaining elements leftward in order to maintain our representation invariants.

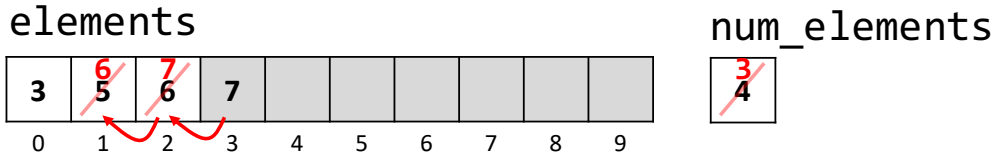

Figure 22.3: Removing from an ordered set requires shifting elements to preserve the representation invariants.

```
void SortedIntSet::remove(int value) {
  int index = indexOf(value);
  if (index != -1) {
    for (; index < num_elements - 1; ++index) {
      elements[index] = elements[index + 1];
    }
    --num_elements;
  }
}
```

The advantage of sorting is that we don't have to look through all the elements to determine the location of a value. Instead, we can use *binary search*, which eliminates half the search space in each step:

- Compare the value we are looking for to the middle element among the remaining items.
- If the value is equal to the middle element, we have found its index.
- If the value is less than the middle element, we know it must be to the left, if it is in the set. Thus, we need only repeat the search on the items in the first half.
- If the value is greater than the middle element, we know it must be to the right, if it is in the set. Thus, we need only repeat the search on the items in the second half.
- If we have run out of elements to search, the value is not in the set.

The following implements this algorithm:

```
int SortedIntSet::indexOf(int value) const {
  int start = 0;
  int end = num_elements;
  while (start < end) {
    int middle = start / 2 + end / 2;
    if (value == elements[middle]) {
      return middle;
    } else if (value < elements[middle]) {
      end = middle;
    } else {
      start = middle + 1;
    }
  }
  return -1;
}
```

Since half the search space is eliminated in each step, this algorithm takes time that is logarithmic in the size of the set. We denote this as $\mathrm{O}(\log n)$, where n is the number of elements in the set.

For a set of size n, the following compares the worst-case runtime of each operation on unsorted and sorted sets:

Operation	Unsorted Set	Sorted Set
`insert()`	$\mathrm{O}(n)$	$\mathrm{O}(n)$
`remove()`	$\mathrm{O}(n)$	$\mathrm{O}(n)$
`contains()`	$\mathrm{O}(n)$	$\mathrm{O}(\log n)$
`size()`	$\mathrm{O}(1)$	$\mathrm{O}(1)$
`constructor`	$\mathrm{O}(1)$	$\mathrm{O}(1)$

The notation $\mathrm{O}(1)$ means that the operation takes constant time: the time it takes is independent of the size of the set.

The `insert()` and `remove()` operations must first check whether the item is in the set, so they can be no faster than `contains()`. The latter is significantly faster on a sorted set than an unsorted set, demonstrating the advantage of sorting.

CHAPTER

TWENTYTHREE

CONTAINER ADTS AND POLYMORPHISM

We continue our discussion of container ADTs by examining several forms of polymorphism in the context of containers. We will start by looking at operator overloading, a form of ad hoc polymorphism, and then proceed to discuss parametric polymorphism.

23.1 Operator Overloading

C++ follows the philosophy that user-defined types should have the same access to language facilities as built-in types. Since operators can be used with built-in atomic types, C++ allows operators to be applied to class types through *operator overloading*.

Most operators in C++ can be overloaded. An operator overload requires at least one of the operands to be of class type[1] – the behavior of operators on atomic types cannot be changed.

When the compiler encounters an operator where at least one operand is of class type, it looks for a function whose name is `operator` followed by the symbol for the actual operator. For example, if + is applied to two `IntSet`s, the compiler looks for a function with name `operator+` that can be applied to two `IntSet` objects. For most operators, the function can either be a top-level function or a member of the type of the left-most operand, if it is of class type.

The following is a member function that defines the + operation to compute the union of two `IntSet`s:

```
class IntSet {
  ...
public:
  IntSet operator+(const IntSet &rhs) const;
};

IntSet IntSet::operator+(const IntSet &rhs) const {
  IntSet result = *this;
  for (int i = 0; i < rhs.num_elements; ++i) {
    result.insert(rhs.elements[i]);
  }
  return result;
}
```

The function first copies over the receiver object into a new local `IntSet`. It then iterates over the elements in the other `IntSet`, inserting them into the result. As we saw last time, the `insert()` function only inserts an element if the set does not already contain the item.

We can now call the member function directly, or we can apply the + operator instead. When we do so, the compiler finds the overloaded `operator+()` and calls it for us.

[1] The operand can also be of enumeration type, which is beyond the scope of this course.

```
int main() {
  IntSet set1;
  set1.insert(32);
  set1.insert(42);
  set1.insert(7);

  IntSet set2;
  set2.insert(12);
  set2.insert(-3);
  set2.insert(42);

  IntSet set3 = set1.operator+(set2);
  set3.print(cout);           // prints { 32, 42, 7, 12, -3 }

  IntSet set4 = set1 + set2;
  set4.print(cout);           // prints { 32, 42, 7, 12, -3 }
}
```

In theory, we could instead define the overloaded operator as a top-level function, declared as follows:

```
IntSet operator+(const IntSet &lhs, const IntSet &rhs);
```

However, we did not define an interface for iterating through the elements of an `IntSet` from outside the class, so we cannot actually implement this function.

Overloaded operators can take arguments by value or by reference, like any other function. In most cases, we pass the arguments by reference to avoid making a copy.

Though most operators can be overloaded either as top-level or member functions, there are some cases where we must use a top-level function:

- The first operand is of atomic type. Atomic types are not classes, so they do not have member functions.
- The first operand is of class type, but we do not have access to the class definition, so we cannot define a new member function.

An example of the latter is overloading the stream-insertion operator, where we do not have access to the definition of `ostream`. The following overloads insertion of an `IntSet`:

```
std::ostream & operator<<(std::ostream &os, const IntSet &set) {
  set.print(os);
  return os;
}
```

We saw previously that inserting to a stream evaluates back to the stream object. To support this properly, our overload returns the given stream object. It must return the object by reference:

- Streams cannot be copied, so the code would not compile if it returned a stream by value.
- Even if streams could be copied, we want to return the original stream object itself, not a copy.
- Even if a copy would work, we would end up with object slicing, since `os` actually will refer to an object of a class that derives from `ostream`.

The parameters are in the same order as the operands, from left to right. The function need only call the `print()` member function on the `IntSet` and then return the given `ostream` object. Then we can insert an `IntSet` directly into a stream:

```
int main() {
  IntSet set;
  set.insert(32);
  set.insert(42);
  cout << set << endl;  // prints { 32, 42 }
}
```

In other cases, we need to define an operator as a member function:

- If the overload needs access to private members, a member function would have access because it is part of the class.[2]
- Some operators can only be overloaded as member functions: the assignment operator (=), the function-call operator (()), the subscript operator ([]), and the arrow operator (->). (We will see examples of overloading the first two operators later.)

As an example, let's overload the subscript operator to check whether the given value is in the set. The following does so:

```
class IntSet {
  ...
public:
  bool operator[](int value) const;
};

bool IntSet::operator[](int value) const {
  return contains(value);
}
```

The following is an example of applying the operator:

```
int main() {
  IntSet set;
  set.insert(32);
  cout << set[32] << endl;  // prints 1 (true)
  cout << set[42] << endl;  // prints 0 (false)
}
```

23.2 Parametric Polymorphism

The `IntSet` container only holds elements that are of type `int`. Suppose we wanted another container that holds `char` values. One solution is to copy and paste the `IntSet` code, then change `int` to `char` everywhere it refers to the element type:

```
class CharSet {
public:
  // Maximum size of a set.
  static const int MAX_SIZE = 10;

  // EFFECTS:  Initializes this set to be empty.
```

(continues on next page)

[2] In the future, we will see that we can use a *friend declaration* to give an outside class or function access to private members. Friend declarations are sometimes used with operator overloads.

(continued from previous page)

```
  CharSet();

  // REQUIRES: size() < MAX_SIZE
  // MODIFIES: *this
  // EFFECTS:  Adds value to the set, if it isn't already in the set.
  void insert(char value);

  // MODIFIES: *this
  // EFFECTS:  Removes value from the set, if it is in the set.
  void remove(char value);

  // EFFECTS:  Returns whether value is in the set.
  bool contains(char value) const;

  // EFFECTS:  Returns the number of elements.
  int size() const;

  // EFFECTS:  Prints out the set in an arbitrary order.
  void print(std::ostream &os) const;

private:
  char elements[MAX_SIZE];
  int num_elements;
  // INVARIANTS:
  // 0 <= num_elements && num_elements <= MAX_SIZE
  // the first num_elements items in elements are the items in the set
  // the first num_elements items in elements contain no duplicates
};
```

This is not a very satisfying solution. It leads to duplication of nearly identical code. Furthermore, if we then wanted a set of `doubles`, we would have to define an almost identical `DoubleSet`, and so on.

We already know how to avoid code duplication when we have a value that can be different: add a function parameter that allows the user to specify the value they care about. The problem here is that our entity that differs is not a value, but a type. While function parameters can represent different argument values, we need another mechanism for specifying *type arguments*. The mechanism that C++ provides is a *template*.

A template is a model for producing code. We write a generic version, parameterized by one or more *template parameters*. The compiler then *instantiates* a specific version of the code by substituting arguments for the parameters and compiling the resulting code. We specify a template and its parameters by placing a template header before the entity that we are defining.

```
template <typename T>
class UnsortedSet {
  ...
};
```

The template header can go on the same line or the previous one: whitespace generally does not matter in C++. The header begins with the `template` keyword, followed by a parameter list surrounded by angle brackets. Within the parameter list, we introduce a template parameter by specifying the kind of entity the parameter can represent. The `typename` keyword indicates that the parameter is a type parameter.[3] The parameter name then follows. Here, we have

[3] The `class` keyword can be used instead of `typename`; the two are equivalent in this context. With either keyword, a non-class type can still be used as the type argument.

chosen the name `T`, since we don't have further information about the type. (`Value_type` or `Element_type` are other common names to use with a container.)

A template may have more than one parameter, and it can also have a parameter that is of integral type. The following is how the `std::array` template is declared:

```
template <typename T, std::size_t N>
class array;
```

Since the entity that follows the template header is a class, it is a *class template*. It takes two arguments, one for the element type and the other for the size of the container, which must be a compile-time constant. We can then create a `std::array` of 10 `int`s as follows:

```
std::array<int, 10> items;
```

The syntax for using a class template is to follow the name of the template with an argument list enclosed by angle brackets. We can similarly create and use an unsorted set of `char`s:

```
UnsortedSet<char> char_set;
char_set.insert('e');
char_set.insert('a');
char_set.insert('e');
cout << char_set << endl;   // prints { e, a }
```

In order for this to work, we write the definition of `UnsortedSet` to use the template parameter `T` for the element type. The scope of a template parameter is the entire entity that follows; if it is a class, then the scope is the entire class definition.

```
template <typename T>
class UnsortedSet {
public:
  // Maximum size of a set.
  static const int MAX_SIZE = 10;

  // EFFECTS:  Initializes this set to be empty.
  UnsortedSet();

  // REQUIRES: size() < MAX_SIZE
  // MODIFIES: *this
  // EFFECTS:  Adds value to the set, if it isn't already in the set.
  void insert(const T &value);

  // MODIFIES: *this
  // EFFECTS:  Removes value from the set, if it is in the set.
  void remove(const T &value);

  // EFFECTS:  Returns whether value is in the set.
  bool contains(const T &value) const;

  // EFFECTS:  Returns the number of elements.
  int size() const;

  // EFFECTS:  Prints out the set in an arbitrary order.
  void print(std::ostream &os) const;
```

(continues on next page)

(continued from previous page)

```
private:
  T elements[MAX_SIZE];
  int num_elements;
  // INVARIANTS:
  // 0 <= num_elements && num_elements <= MAX_SIZE
  // the first num_elements items in elements are the items in the set
  // the first num_elements items in elements contain no duplicates
};
```

The actual type argument used to instantiate `UnsortedSet` may be something small like `int`, or it may be a large class type such as `string`. Thus, it is good practice to pass objects of a template parameter by reference rather than by value, avoiding copying potentially large objects.

When we use `UnsortedSet` with a particular type argument, the compiler actually plugs the argument in for `T`, generating code for that instantiation and compiling it with the rest of the program. For example, `UnsortedSet<string>` is instantiated as follows:

```
class UnsortedSet<string> {
public:
  // Maximum size of a set.
  static const int MAX_SIZE = 10;

  // EFFECTS:  Initializes this set to be empty.
  UnsortedSet();

  // REQUIRES: size() < MAX_SIZE
  // MODIFIES: *this
  // EFFECTS:  Adds value to the set, if it isn't already in the set.
  void insert(const string &value);

  // MODIFIES: *this
  // EFFECTS:  Removes value from the set, if it is in the set.
  void remove(const string &value);

  // EFFECTS:  Returns whether value is in the set.
  bool contains(const string &value) const;

  // EFFECTS:  Returns the number of elements.
  int size() const;

  // EFFECTS:  Prints out the set in an arbitrary order.
  void print(std::ostream &os) const;

private:
  string elements[MAX_SIZE];
  int num_elements;
  // INVARIANTS:
  // 0 <= num_elements && num_elements <= MAX_SIZE
  // the first num_elements items in elements are the items in the set
  // the first num_elements items in elements contain no duplicates
};
```

23.2.1 Function Templates

We can also define a function as a template, resulting in a *function template*. For example, the following computes the maximum of two items of the same type:

```
template <typename T>
T max(const T &value1, const T &value2) {
  return value2 > value1 ? value2 : value1;
}
```

As with a class template, we define a function template by preceding it with a template header, which introduces one or more template parameters. We can then use the parameter anywhere in the function definition. Here, our function template takes two arguments of the same type and compares them. The `?:` operator is a conditional. It evaluates its first operand, and if the result is true, it produces the second operand. Otherwise, it produces the third operand.

Similar to a class template, we can use a function template by following its name with an argument list enclosed by angle brackets:[4]

```
int main() {
  int i1 = 3;
  int i2 = -3;
  cout << max<int>(i1, i2) << endl;
  cout << max<double>(3.1, 7.5) << endl;
}
```

Unlike a class template, however, the compiler is able in most cases to *deduce* the template argument from the arguments to the function call. In the first call above, the arguments are both of type `int`, so the compiler can deduce that we want `int` as the template argument. Similarly, the arguments are both of type `double` in the second call, so the compiler can deduce we want `double`. Thus, we can leave off the explicit template arguments:

```
int main() {
  int i1 = 3;
  int i2 = -3;
  cout << max(i1, i2) << endl;    // deduced as max<int>(i1, i2)
  cout << max(3.1, 7.5) << endl;  // deduced as max<double>(3.1, 7.5)
}
```

The `max()` function template can only be applied to a type that supports the > operator. If we try to call it on objects that don't support the operator, the result is a compile error:

```
int main() {
  Duck d1("Donald");
  Duck d2("Scrooge");
  Duck best_duck = max(d1, d2);
  cout << best_duck.get_name() << " wins!" << endl;
}
```

This results in an error like the following:

```
main.cpp: In instantiation of 'T max(const T&, const T&) [with T = Duck]':
main.cpp:20:30:   required from here
main.cpp:14:17: error: no match for 'operator>' (operand types are 'const Duck' and
```

(continues on next page)

[4] Depending on the compiler and what headers are included, we may have to tell the compiler that we intend to use our `max()` function template rather than the one in the standard library. We can do so by using the scope-resolution operator with nothing to the left: `::max<double>(3.1, 7.5)`.

(continued from previous page)

```
↪'const Duck')
  return value2 > value1 ? value2 : value1;
         ~~~~~~~^~~~~~~~
```

The offending line of code is the one marked as "required from here" – it is the line that attempts to call `max()` on two `Duck` objects. The compiler actually instantiates `max<Duck>()`, but the resulting instantiation produces errors when the compiler tries to compile it.

23.2.2 Compiling Templates

We saw previously that the C++ compiler *only needs access to declarations* when compiling code that uses a class or a function defined in some other source file. However, this is not the case for class and function templates. The compiler must actually instantiate the definitions for each set of template arguments, so it must have access to the full definition of a template.

To provide the compiler with access to the full definition of a template wherever it is used, we must arrange for the header file to contain the full definition. We can just define the template directly in the header file itself; it is still good practice to separate the declarations from the definitions for the benefit of anyone using our code.

```
// max.hpp

// EFFECTS: Returns the maximum of value1 and value2.
template <typename T>
T max(const T &value1, const T &value2);

...

template <typename T>
T max(const T &value1, const T &value2) {
  return value2 > value1 ? value2 : value1;
}
```

A better organization is to separate the definitions into their own file; common convention is to use a suffix like `.tpp` for this file. We can then use a `#include` directive to pull the code into the header file:

```
// max.hpp

// EFFECTS: Returns the maximum of value1 and value2.
template <typename T>
T max(const T &value1, const T &value2);

#include "max.tpp"
```

```
// max.tpp

template <typename T>
T max(const T &value1, const T &value2) {
  return value2 > value1 ? value2 : value1;
}
```

Code that uses the `max` module would then just `#include "max.hpp"`, which would transitively include `max.tpp`.

23.2.3 Include Guards

In complex programs, it is often the case that a single file inadvertently `#include`s the same header file more than once. For instance, it may be that module A depends on module B and thus includes its header. Then module C depends on both A and B, so it includes both their headers. This results in module C including B's header twice, which can cause compiler errors that a function or a class is defined more than once.

To avoid problems with a header being included more than once, headers generally have *include guards* that arrange for the compiler to ignore the code if the header is included a second time. The following is an example of include guards:

```
#ifndef MAX_HPP
#define MAX_HPP

// max.hpp

// EFFECTS: Returns the maximum of value1 and value2.
template <typename T>
T max(const T &value1, const T &value2);

#include "max.tpp"

#endif /* MAX_HPP */
```

The `#ifndef` and `#define` directives are the opening of the include guard, and the `#endif` closes it. The code within is ignored if the header is included again.[5]

23.2.4 Member-Function Templates

Defining a member function within the definition for a class template is no different than a member function within a class. Defining a member function outside the definition of a class template differs from that of a regular class; we must inform the compiler that the function is a member of a class template.

```
template <typename T>
bool UnsortedSet<T>::contains(const T &value) const {
  return indexOf(value) != -1;
}

template <typename T>
void UnsortedSet<T>::insert(const T &value) {
  assert(size() < MAX_SIZE);
  if (!contains(value)) {
    elements[num_elements] = value;
    ++num_elements;
  }
}
```

Here, we must tell the compiler that `contains()` is a member of `UnsortedSet<T>`. However, before we can use the name `T`, we need to introduce it with a template header. The scope of a template header is the entity that immediately follows; thus, each member-function definition needs its own template header.

[5] The `#ifndef` checks whether the *macro* `MAX_HPP` is defined. If not, the compiler processes the code between the `#ifndef` and the `#endif`. The `#define` introduces a definition for `MAX_HPP`. The next time the header is included, `MAX_HPP` is defined, so the `#ifndef` becomes false, and the compiler ignores the code between the `#ifndef` and `#endif`.

A widely supported but nonstandard alternative is `#pragma once`, which need only be placed at the top of file.

23.2.5 Insertion-Operator Template

Now that we have an `UnorderedSet` class template, we can write an insertion operator that works on any instantiation by defining it as a function template:

```
template <typename T>
std::ostream & operator<<(std::ostream &os, const UnsortedSet<T> &set) {
  set.print(os);
  return os;
}
```

The function template is parameterized by the element type of an `UnorderedSet`, and we use the template parameter as part of the type for the second function parameter. The compiler is able to deduce the template argument when we insert a set into a stream:

```
UnsortedSet<char> char_set;
char_set.insert('e');
char_set.insert('a');
char_set.insert('e');
cout << char_set << endl;   // prints { e, a }
```

23.2.6 Another Example

Let us write a function template that copies elements from an array into a set. We would like to use it as follows:

```
int main() {
  UnsortedSet<int> set1;
  int arr1[4] = { 1, 2, 3, 2 };
  fill_from_array(set1, arr1, 4);
  cout << set1 << endl;              // prints { 1, 2, 3 }

  UnsortedSet<char> set2;
  char arr2[3] = { 'a', 'b', 'a' };  // prints { a, b }
  fill_from_array(set2, arr2, 3);
}
```

The `fill_from_array()` template must be parameterized by the element type, and we use that type for both the set and array function parameters:

```
template <typename T>
void fill_from_array(UnsortedSet<T> &set, const T *arr, int size) {
  for (int i = 0; i < size; ++i) {
    set.insert(arr[i]);
  }
}
```

The set is passed by reference, to avoid making a copy and to allow the function to modify the original set. The array decays into a pointer when it is passed, and we declare the pointer as a pointer to const, since we do not need to modify the array elements. The body of the function template just iterates over the array, inserting each element into the set; the set ignores insertion of duplicate values, so there is no need to check whether the set already contains a value before inserting it.

23.3 Static vs. Dynamic Polymorphism

The template mechanism gives us parametric polymorphism; a template type parameter can take on the form of any type. The compiler instantiates the code at compile time, so we also refer to this as *static polymorphism*. Compare this with subtype polymorphism, where the program resolves virtual function calls at runtime; we therefore use *dynamic polymorphism* to also refer to it.

In designing an ADT, we have the choice of static polymorphism, dynamic polymorphism, or both. For instance, `SortedSet` and `UnsortedSet` share the same interface, so we could define a common base interface and make use of dynamic binding. However, dynamic polymorphism comes with a runtime performance cost; a virtual function call generally requires two extra pointer dereferences compared to a non-virtual call. For a container, a program may make many calls to insert items, remove items, check whether an item is in the container, and so on. Thus, the general convention in C++ is to avoid dynamic polymorphism with containers.

Another reason why dynamic polymorphism is not a good fit for a container is that we usually make the decision of which container to use when we write our code, not when we run it. However, it is still good practice to write our code in such a way that we can easily change the container type we are using. Ideally, we would have a single location in our code that needs to change if we decide to use something else. We've seen this pattern already when it comes to a value that is used in many places: define a constant or variable, and use the associated name rather than the value directly. Thus, introducing a new name for an entity is a powerful form of abstraction, allowing us to avoid hardcoding the entity everywhere in our code. C++ gives us this capability for types with type aliases.

23.3.1 Type Aliases

A *type alias* is a new name for an existing type. We can introduce a type alias with a *using declaration*:

```
using UnsortedIntSet = UnsortedSet<int>;
```

The syntax for introducing a type alias with a using declaration is the `using` keyword, followed by the name we want to introduce, the = symbol, and the existing type that we want the name to alias. The example above introduces the name `UnsortedIntSet` as an alias for the type `UnsortedSet<int>`.

A simple alias can also be defined with the `typedef` keyword:

```
typedef UnsortedSet<int> UnsortedIntSet;
```

The syntax is in the reverse order of a using declaration: the existing type first, then the new alias.

A using declaration (but not a typedef) can also define an *alias template*, which introduces a new name that is a template for a set of types. The following is an example:

```
template <typename T>
using Set = UnsortedSet<T>;
```

Similar to any template, we introduce an alias template by placing a template header with the requisite parameters before the using declaration. Then we can use `Set` in the same way we would a class template:

```
template <typename T>
using Set = UnsortedSet<T>;

template <typename T>
void fill_from_array(Set<T> &set, const T *arr, int n);

int main() {
```

(continues on next page)

(continued from previous page)

```
  Set<int> set1;
  int arr1[4] = { 1, 2, 3, 2 };
  fill_from_array(set1, arr1, 4);

  Set<char> set2;
  char arr2[3] = { 'a', 'b', 'a' };
  fill_from_array(set2, arr2, 3);
}
```

The compiler treats `Set<int>` and `Set<char>` as `UnsortedSet<int>` and `UnsortedSet<char>`, since `Set` is an alias for `UnsortedSet`. If we decide to use sorted sets instead, there is just one place we need to change:

```
template <typename T>
using Set = SortedSet<T>;
```

Now the compiler will treat `Set<int>` and `Set<char>` as `SortedSet<int>` and `SortedSet<char>`.

23.3.2 The Factory Pattern

With dynamic polymorphism, we can defer the decision of which derived class to use until runtime. For example, the following code asks the user for what color bird they want, then calls a *factory* function to create the actual object:

```
int main() {
  string color;
  cin >> color;
  Bird *bird = Bird_factory(color, "Myrtle");
  cout << "Bird " << bird->get_name() << " is " << bird->get_age()
       << " and says: " << bird->talk() << endl;
  delete bird;

  cin >> color;
  bird = Bird_factory(color, "Heihei");
  cout << "Bird " << bird->get_name() << " is " << bird->get_age()
       << " and says: " << bird->talk() << endl;
  delete bird;
}
```

The factory function creates an object of the appropriate derived type and returns a pointer to it. It cannot create the object in local memory – a local object dies when the function that created it returns, so it would not live past the call to `Bird_factory()`. Instead, we create it in *dynamic memory* with the `new` operator:

```
Bird * Bird_factory(const string &color, const string &name) {
  if (color == "blue") {
    return new BlueBird(name);
  } else if (color == "black") {
    return new Raven(name);
  }
  ...
}
```

We then use the `delete` operator on the object's address when we no longer need it.

We will start looking at *dynamic memory* in more detail next time.

23.3.3 Liskov Substitution Principle

When designing an inheritance hierarchy, an important property is *substitutability* of a derived-class object for a base-class one. This notion is formalized by the *Liskov substitution principle*, which states that in order for type S to be a *subtype* of type T, the following requirement must be met:

> Any property of objects of type T should also be a property of objects of type S.
>
> —Barbara Liskov, MIT

This implies that in any code that depends on T's interface, an object of type S can be substituted without any undesirable effects. If the requirement above is not satisfied, S would be a derived type of T but not a subtype.

A classic example is a `ToyDuck` class that derives from `Duck`:

```
class ToyDuck : public Duck {
  ...
public:
  // EFFECTS: Prints out quack if battery level >= 10.
  void talk() const override {
    if (battery_level >= 10) {
      cout << "quack" << endl;
      --battery_level;
    }
  }
};
```

Code that uses a `Duck` may assume that a quacking noise gets printed out each time `talk()` is called. A `ToyDuck` would violate this expectation, since it does not print anything out if the `ToyDuck` has insufficient battery. Thus, `ToyDuck` is a derived type of `Duck`, but not a subtype.

Figure 23.1: Liskov substitution principle. Credit: Derick Bailey, retrieved from Stack Overflow

CHAPTER

TWENTYFOUR

MEMORY MODELS AND DYNAMIC MEMORY

Recall that an *object is a piece of data in memory*, located at some address in memory. An object also has a *storage duration* that determines its lifetime. In this course, we consider three storage durations:

- *static*: the lifetime is essentially the whole program
- *automatic* (also called *local*): the lifetime is tied to a particular scope, such as a block of code
- *dynamic*: the object is explicitly created and destroyed by the programmer

The first two durations are controlled by the compiler, and an object with static or automatic scope is associated with a variable. Objects with dynamic storage are managed by the programmer, and they are not directly associated with variables.

The following variables refer to objects with static storage duration:

- global variables[1]
- *static member variables*
- static local variables

The lifetime of these objects is essentially the whole program, so they can be used at any point in time in the program.[2]

24.1 Static Local Variables

We have already seen global and static member variables. A *static local variable* is a local variable declared with the `static` keyword. Rather than living in the activation record for a function call, it is located in the same region of memory as other variables with static storage duration, and there is one copy of each static local variable in the entire program. The following is an example:

```
int count_calls() {
  static int count = 0;
  return ++count;
}

int main() {
  cout << count_calls() << endl;    // prints 1
  cout << count_calls() << endl;    // prints 2
  cout << count_calls() << endl;    // prints 3
  cout << count_calls() << endl;    // prints 4
}
```

[1] Also those defined at *namespace* scope.

[2] Initialization in C++ is rather complicated, and the details are beyond the scope of this course. A relatively safe assumption is that variables with static storage duration can be initialized in any order before `main()` is run, but that they have all been initialized by the time `main()` is called.

The `count` static local variable is initialized to 0, so its value is 0 the first time `count_calls()` is called. The call increments `count` and returns the new value of 1. The next time `count_calls()` is called, `count` has value 1. It gets incremented again, and the call returns 2. And so on. Thus, all calls to `count_calls()` use the same `count` object, unlike a non-static local variable, where each call would get its own object.

24.2 Automatic Storage Duration

Function parameters and local variables have automatic storage duration, also called local storage duration. The lifetime of the associated object corresponds to the variable's *scope*, the region of code where it is valid to refer to the variable. The scope of a variable begins at its point of declaration and ends at the end of the scope region in which it is defined. A scope region can be the body of a function, a whole loop or conditional, the body of a loop or conditional, or just a plain *block* denoted by curly braces. For example, consider the following code:

```
void func(int x) {
  for (int i = 2; i < 10; ++i) {
    string s = " cats";
    cout << i << s << endl;
  } // end scope of i, s

  int y = 10;

  {
    int z = -3;
    cout << x << " " << y << " " << z << endl;
  } // end scope of z

} // end scope of x, y
```

The variables fall into the following categories;

- The scope of a parameter is the entire function body. Thus, the lifetime of the object associated with `x` begins when the function is called and ends when the function returns.
- The scope of a variable declared in a loop header is the entire loop. Thus, the lifetime of the object `i` begins when the loop starts and ends when the loop exits.
- The scope of a variable declared within a block is from its point of declaration to the end of the block. The lifetime of `s` begins when its initialization runs and ends at the end of a loop iteration – a new object associated with `s` is created in each iteration. The lifetime of `y` also begins when its initialization runs, and it ends when the function returns. Lastly, the lifetime of `z` starts when its declaration is run and ends when the code exits the block containing `z`'s declaration.

Scope generally prevents us from using a local variable beyond the lifetime of its associated object. However, indirection through a pointer or reference can circumvent this protection. The following is an example:

```
int *func1() {
  int x = 3;
  return &x;
}

void func2() {
  int y = 5;
}
```

(continues on next page)

(continued from previous page)

```
int main() {
  int *z = func1();
  func2();
  assert(*z == 3);  // ASSERTION FAILURE
}
```

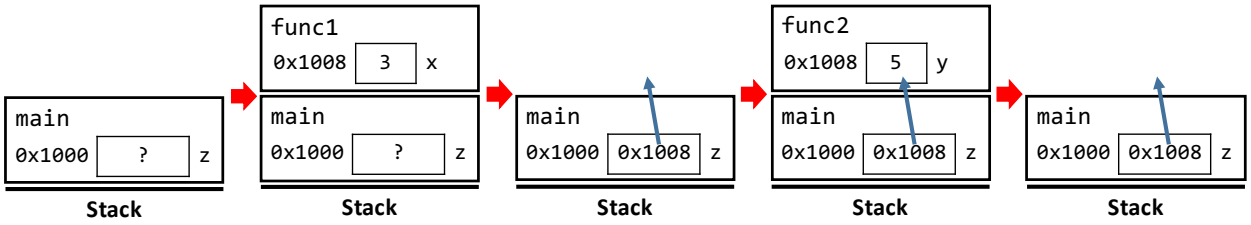

Figure 24.1: Returning the address of a local variable results in a pointer that refers to a dead object. Dereferencing the pointer produces undefined behavior.

When the program calls `func1()`, it places the activation record on the stack, with `x` inside of the activation record. The call returns the address of `x`, which gets stored in `z`. However, when the call returns, its activation record is destroyed, so `z` is actually pointing at a dead object. The memory for that activation record can then be reused for something else – in the case of Figure 24.1, the activation record for `func2()`, placing `y` at the same address used for `x`. Dereferencing `z` results in undefined behavior; in this case, since the memory was reused for `y`, dereferencing `z` results in the value 5, not 3.

There are cases, however, where we want an object's lifetime to last beyond the execution of the function that creates it. Previously, we saw the *factory pattern*, where a factory function creates an object and returns its address. We used the `new` operator to decouple the lifetime of the object from the duration of the function call:

```
Bird * Bird_factory(const string &color, const string &name) {
  if (color == "blue") {
    return new BlueBird(name);
  } else if (color == "black") {
    return new Raven(name);
  }
  ...
}
```

24.3 Address Space

The `new` operator creates an object not in some activation record, but in an independent region of memory called the *heap*. The stack and heap are two of the *segments* that make up a program's *address space*, which is the total memory associated with a running program. Figure 24.2 depicts a typical layout for an address space.

The *text* segment contains the actual machine instructions representing the program's code, and it is often placed at the bottom of a program's address space. (Nothing is located at address 0, since that is the representation used for a null pointer.) Space for objects in static storage generally is located above the text segment, followed by the heap. The latter can grow or shrink as objects in dynamic memory are created and destroyed; in most implementations the heap grows upward as needed, into the empty space between the heap and stack. The stack starts at the top of the address space, and it grows downward when a new activation record is created.

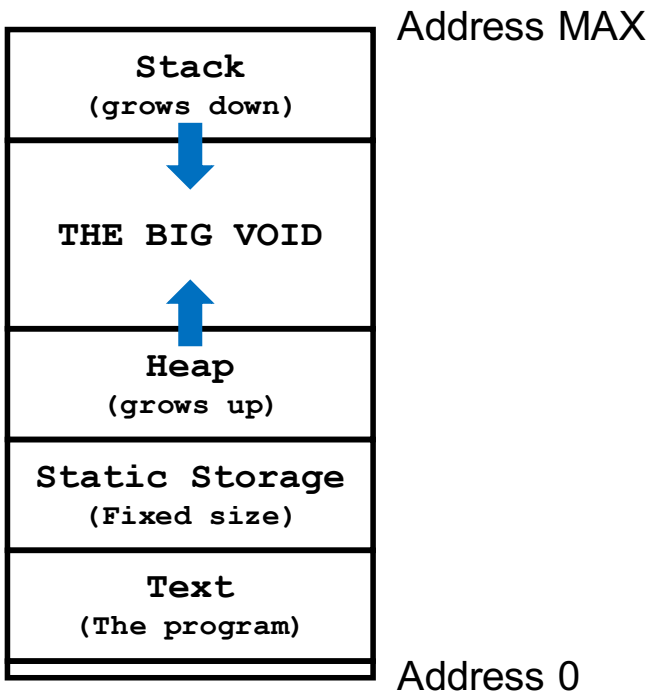

Figure 24.2: The address space of a program.

24.4 The `new` and `delete` Operators

The syntax of a `new` expression consists of the `new` keyword, followed by a type, followed by an optional initialization using parentheses or curly braces. The following are examples:

```
// default initialization
new int;
// value initialization
new int();
new int{};
// explicit initialization
new int(3);
new int{3};
```

If no initialization is provided, the newly created object is default initialized. For an atomic type, nothing is done, so the object's value is whatever is already there in memory. For a class type, the default constructor is called.

Empty parentheses or curly braces result in *value initialization*. For an atomic type, the object is initialized to a zero value. For a class type, the default constructor is called.

Explicit initialization can be done by supplying values in parentheses or curly braces. For atomic types, the value is used to initialize the object. For C-style ADTs, curly braces must be used, and the values within are used to initialize the member variables. For C++ ADTs, both parentheses and curly braces invoke a constructor with the values within as arguments.

A `new` expression does the following:

- Allocates space for an object of the given type on the heap.

- Initializes the object according to the given initialization expression.
- Evaluates to the address of the newly created object.

The address is how we keep track of the new object; it is not associated with a variable name, so we have to store the address somewhere to be able to use the object.

```
int main() {
  int *ptr = new int(3);
  cout << *ptr << endl;   // prints 3
  ...
}
```

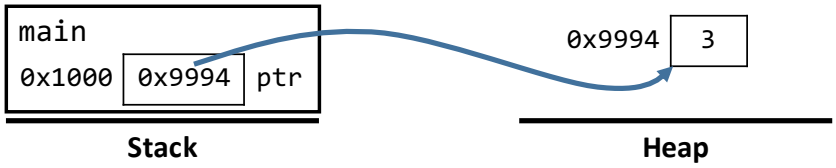

Figure 24.3: A `new` expression creates an object on the heap and evaluates to its address.

The newly created object's lifetime is not tied to a particular scope. Rather, it begins when the `new` expression is evaluated and ends when the `delete` operator is applied to the object's address.

```
delete ptr;
```

Here, the operand to `delete` evaluates to the address value contained in `ptr`. The `delete` operator follows this address and destroys the object at that address.

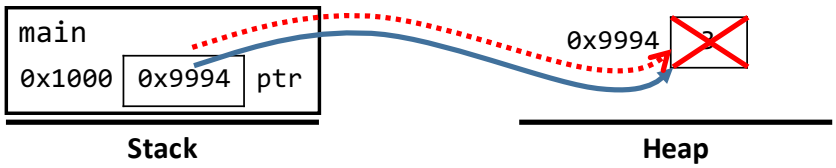

Figure 24.4: Applying `delete` to an address in dynamic memory kills the object that lives at that address.

The expression `delete ptr;` does **not** kill the `ptr` object – it is a local variable, and its lifetime ends when it goes out of scope. Rather, `delete` follows the pointer to the object it is pointing to and kills the latter. We can continue to use the pointer object itself:

```
int main() {
  int *ptr = new int(3);
  cout << *ptr << endl;   // prints 3
  delete ptr;             // kills *ptr, not ptr itself
  ptr = new int(-1);
  cout << *ptr << endl;   // prints -1
  delete ptr;             // kills *ptr, not ptr itself
}                         // ptr dies here, since it is going out of scope
```

24.5 Dynamic Arrays

We saw previously that local and member-variable arrays must have a size that is known at compile time, so the compiler can reason about the sizes of activation records and class-type objects. This restriction does not apply to dynamic arrays – since they are located on the heap rather than in an activation record or directly within a class-type object, the compiler does not need to know their size.

The syntax for creating a dynamic array is the `new` keyword, an element type, square brackets containing an integer expression, and an optional initializer list. The expression within the square brackets need not be a compile-time constant:

```
int main() {
  cout << "How many elements? ";
  int count;
  cin >> count;
  int *arr = new int[count];
  for (int i = 0; i < count; ++i) {
    arr[i] = i;
  }
  ...
}
```

A `new` array expression does the following:

- Allocates space for an array of the given number of elements on the heap.
- Initializes the array elements according to the given initialization expression. If no initialization is provided, the elements are default initialized.
- Evaluates to the address of the first element of the newly created array.

Previously, we saw that a local or member-variable array *decays to a pointer* to its first element when the array's value is required. A dynamic array immediately decays as part of the `new` array expression, so the expression evaluates to the address of the first element.

Figure 24.5 illustrates memory at the end of the code snippet above.

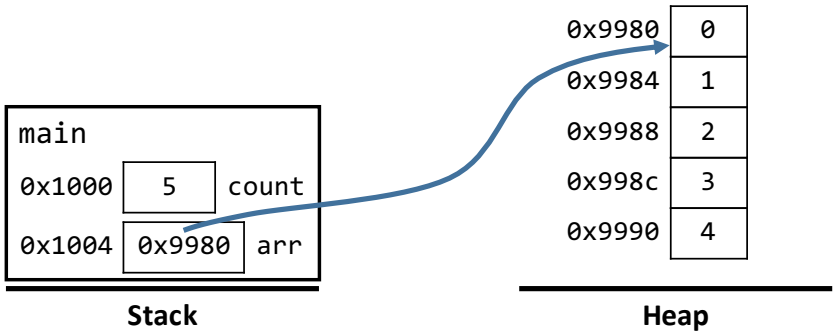

Figure 24.5: A `new` array expression creates an array on the heap and evaluates to the address of its first element.

As with any pointer into an array, we can use the subscript operator to index into the array:

```
arr[i] = i;
```

This is equivalent to:

```
*(arr + i) = i;
```

The lifetime of a dynamic array begins when it is created by a `new` array expression. It ends when the array-deletion operator `delete[]` is applied to the address of the array's first element:

```
delete[] arr;
```

Though the type of `arr` is `int *`, we cannot use the regular `delete` operator on it; instead, we must inform the program with `delete[]` that we intend to delete a dynamic array, not just a single object on its own.

Using the wrong deletion operator results in undefined behavior. It is also undefined behavior to apply `delete[]` to anything other than the address of the first element of a dynamic array.

We cannot delete an individual element of an array. The lifetime of an array's elements are tied to the lifetime of the array as a whole; they are born when the array as a whole is born and die when the array dies.

24.6 Lifetime of Class-Type Objects

When a class-type object is created, a constructor is run to initialize it.[3] When a class-type object dies, its destructor is run to clean up its data. For a local object, this is when the associated variable goes out of scope. For a dynamic object, it is when `delete` is applied to its address.

If a class-type object is an element of an array, it dies when the array dies, so its destructor runs when the array is dying.

The lifetime of member variables of a class-type object is tied to the lifetime of the object as a whole. When the object dies, its members die as well; if they are of class-type themselves, their destructors run. Specifically, the members of a class-type object are automatically destroyed after the destructor of the object is run, in reverse order of their declarations. The following is an example:

```
class M {
public:
  M(const string &s) : mstr(s) {
    cout << "M ctor: " << mstr << endl;
  }

  ~M() {
    cout << "M dtor: " << mstr << endl;
  }

private:
  string mstr;
};

class C {
public:
  C(const string &s) : cstr(s), m1(s + ".m1"), m2(s + ".m2") {
    cout << "C ctor: " << cstr << endl;
  }

  ~C() {
    cout << "C dtor: " << cstr << endl;
  }
```

(continues on next page)

[3] The exception is aggregates (C-style ADTs) that are initialized directly with an initializer list.

(continued from previous page)

```
private:
  string cstr;
  M m1;
  M m2;
};

int main() {
  C c1("c1");
  C *cptr = new C("(*cptr)");
  C c2("c2");
  delete cptr;
}
```

This prints the following when run:

```
M ctor: c1.m1
M ctor: c1.m2
C ctor: c1
M ctor: (*cptr).m1
M ctor: (*cptr).m2
C ctor: (*cptr)
M ctor: c2.m1
M ctor: c2.m2
C ctor: c2
C dtor: (*cptr)
M dtor: (*cptr).m2
M dtor: (*cptr).m1
C dtor: c2
M dtor: c2.m2
M dtor: c2.m1
C dtor: c1
M dtor: c1.m2
M dtor: c1.m1
```

When `main()` is run, the declaration of `c1` creates a `C` object and calls its constructor. The latter first initializes the members `m1` and `m2` in order before running the body of `C`'s constructor. Thus, we get the output:

```
M ctor: c1.m1
M ctor: c1.m2
C ctor: c1
```

The code then constructs a `C` object in dynamic memory, resulting in:

```
M ctor: (*cptr).m1
M ctor: (*cptr).m2
C ctor: (*cptr)
```

It then creates another local `C` object:

```
M ctor: c2.m1
M ctor: c2.m2
C ctor: c2
```

The program proceeds to apply `delete` to the address of the dynamic `C` object, which causes its destructor to run. The body of `~C()` runs first, and then the members are destroyed in reverse order:

```
C dtor: (*cptr)
M dtor: (*cptr).m2
M dtor: (*cptr).m1
```

The string `cstr` is also destroyed, since it is a class-type object, and it is destroyed after `m1`. However, the `string` destructor doesn't print anything out, so we do not see it in the output.

The order in which the bodies of the destructors are run is the reverse of the constructors – we get the same "socks-and-shoes" ordering that we saw with *base-class and derived-class constructors and destructors*.

When execution reaches the end of `main()`, both `c1` and `c2` go out of scope, so their associated objects are destroyed in reverse order of construction. Thus, `c2` dies first, followed by `c1`:

```
C dtor: c2
M dtor: c2.m2
M dtor: c2.m1
C dtor: c1
M dtor: c1.m2
M dtor: c1.m1
```

24.7 Dynamic-Memory Errors

The compiler manages objects with static and automatic storage duration, and the management of subobjects is directly tied to the objects that contain them. Dynamic objects, on the other hand, must be explicitly managed by the programmer. Improper management can result in many kinds of errors that are unique to dynamic memory.

The most common error with dynamic memory is a *memory leak*, where a program is no longer in need of a dynamic object but has failed to delete it. The usual cause of a memory leak is *orphaned memory*, when we lose track of a dynamic object, inevitably resulting in a leak. The following is an example:

```
void func1() {
  double *ptr = new double(3.0);
}

int main() {
  func1();
  ...
}
```

When `func1()` is called, it allocates a dynamic object and stores its address in the local variable `ptr`. When `func1()` returns, its activation record is destroyed and `ptr` dies. Thus, the program no longer knows the address of the dynamic object that `func1()` created – the object has been orphaned, and there is no way to get to it to delete it. Thus, the corresponding memory is leaked.

The solution is to apply `delete` to the address before we lose track of it:

```
void func1() {
  double *ptr = new double(3.0);
  delete ptr;
}
```

(continues on next page)

(continued from previous page)

```
int main() {
  func1();
  ...
}
```

Memory leaks result in a program using more memory than it needs. This is problematic on systems that run more than one program at once, which is the case for most machines – it reduces the amount of physical memory available to the other programs, potentially slowing them down. It is our responsibility as programmers to write well-behaved programs that avoid memory leaks.

Another memory error is a *double free*, also called a *double delete*, where a program deletes the same object more than once:

```
void func2() {
  int x = 0;
  int *ptr1 = new int(1);
  int *ptr2 = new int(2);
  ptr2 = ptr1;

  delete ptr1;
  delete ptr2;
}
```

Here, the second dynamic `int` is orphaned by the assignment `ptr2 = ptr1`. The program then applies `delete` to the same address twice; this is a double free, which results in undefined behavior.

Another category of errors is a *bad delete*, which is when a delete operator is applied to the wrong kind of address. This includes applying `delete` to the address of a non-dynamic object and applying `delete[]` to anything other than the address of the first element in a dynamic array. The result of a bad delete is undefined behavior.

Deleting a dynamic object is not enough to avoid a memory error. If we keep around its address, we have a *dangling pointer* – a pointer that points to a dead object. If we then dereference the pointer, we get undefined behavior. The following is an example:

```
void func3() {
  int *ptr = new int(42);
  cout << *ptr << endl;
  delete ptr;

  int *ptr2 = new int(3);
  cout << *ptr << endl;    // oops
  delete ptr2;
}
```

In this code, we have accidentally dereferenced `ptr` after applying `delete` to it. The object it is pointing at is dead, and the memory may have been reused for something else.

We can avoid dangling pointers by restricting the scope of a pointer to the region of code where it is expected to be used[4]. One way to do so is by moving that code into its own block, as in the following:

[4] An alternative strategy is to set a pointer to null after applying `delete` to it. Dereferencing a null pointer still results in undefined behavior, but implementations almost universally detect this, resulting in a *segmentation fault* or a *bad access*. However, there are two downsides to this strategy. First, it complicates the code, making it harder to read and maintain. Second, deleting a null pointer is valid in C++ – it has no effect. This means that if we set a pointer to null and then subsequently delete it a second time, we will not encounter an error. However, this may hide a logic bug – perhaps we expected the pointer to still be alive even though it was deleted earlier, in which case we would prefer a visible error from a double delete rather the code passing our test cases. For these reasons, it is better to rely on scoping to detect dangling pointers at compile time rather than relying on runtime checks.

```
void func3() {
  {
    int *ptr = new int(42);
    cout << *ptr << endl;
    delete ptr;
  }

  int *ptr2 = new int(3);
  cout << *ptr << endl;    // oops
  delete ptr2;
}
```

Now `ptr` is no longer in scope when we mistakenly access it, and the compiler will detect this and give us a meaningful error.

24.8 Uses for Dynamic Memory

Dynamic memory adds another use for indirection through a pointer or reference. Since a dynamic object is not directly associated with a variable, we are forced to interact with the object indirectly.

We have already seen two uses for dynamic memory:

- The factory pattern, where a factory function creates objects of specific derived types based on runtime information.
- Dynamic arrays, where the size can be determined at runtime.

The factory pattern is an example of decoupling the lifetime of an object from a particular scope. Next time, we will make use of dynamic arrays to build a container ADT without a fixed maximum capacity; we will do so by decoupling the storage for the container's elements from that of the container itself. However, we will see that this results in nontrivial memory-management issues, and we will learn how to address them.

CHAPTER

TWENTYFIVE

MANAGING DYNAMIC MEMORY

We saw previously that the compiler and runtime automatically manage the lifetime of objects associated with static, local, and member variables. For instance, if we declare an array as a local variable, it will die when the variable goes out of scope:

```
int main() {
  int arr[10];

  for (int i = 0; i < 10; ++i) {
    arr[i] = i;
  }

  ...
} // arr dies here
```

An automatically managed array must have a size that is known at compile time, so that the compiler can properly manage it. The compiler does not automatically manage dynamic arrays:

```
int main() {
  int size;
  cout << "Enter a size:" << endl;
  cin >> size;

  int *darr = new int[size];

  for (int i = 0; i < size; ++i) {
    darr[i] = i;
  }

  ...
} // the pointer darr dies here, but not the memory it is pointing to
```

The code above contains a memory leak, since we did not delete the memory we dynamically allocated.

We also have seen that when a class-type object dies, its destructor is run:

```
class C {
public:
  ~C() {
    cout << "C dtor" << endl;
  }
};
```

(continues on next page)

(continued from previous page)

```
int main() {
  C c;

  ...

  cout << "end of c's scope" << endl;
} // c dies here
```

This prints out the following when run:

```
end of c's scope
C dtor
```

Here, the compiler automatically manages the lifetime of the object associated with the local variable `c`, and since it is of class type, its destructor is run when the object dies.

25.1 RAII

We can leverage automatic memory management and destructors by wrapping up the management of a dynamic resource in a class. In particular, we will arrange for the constructor of a class-type object to allocate a resource and for the destructor to release that resource. Doing so ties the management of the resource to the lifetime of the class-type object. This strategy is called *resource acquisition is initialization (RAII)*, and it is also known as *scope-based resource management*.

The following is a class that manages a dynamic array:

```
class DynamicIntArray {
public:
  DynamicIntArray(int size_in)
    : elements(new int[size_in]), num_elements(size_in) {}

  ~DynamicIntArray() {
    delete[] elements;
  }

  int size() const {
    return num_elements;
  }

  int & operator[](int i) {
    return elements[i];
  }

  const int & operator[](int i) const {
    return elements[i];
  }

private:
  int *elements;
  int num_elements;
};
```

When a `DynamicIntArray` object is created, it allocates a dynamic array of the specified size. The subscript operator is overloaded to index into the array. There are two overloads, one that returns a modifiable element by reference if applied to a non-const `DynamicIntArray`, and another that returns a non-modifiable element by reference to const when applied to a const `DynamicIntArray`. The class also provides a `size()` member function to query the size of the array. When the `DynamicIntArray` object dies, it deletes the dynamic array that it had allocated, so that memory does not leak.

The following is an example that uses `DynamicIntArray`:

```
int main() {
  int size;
  cout << "Enter a size:" << endl;
  cin >> size;

  DynamicIntArray darr(size);      // internally allocates an array

  for (int i = 0; i < darr.size(); ++i) { // size() member function
    darr[i] = i;                          // overloaded subscript
  }

  ...
}     // darr dies here, causing its destructor to run and free the
      // array it allocated
```

When the object associated with `darr` is created, it allocates a dynamic array, storing the address of the first element in its `elements` member.

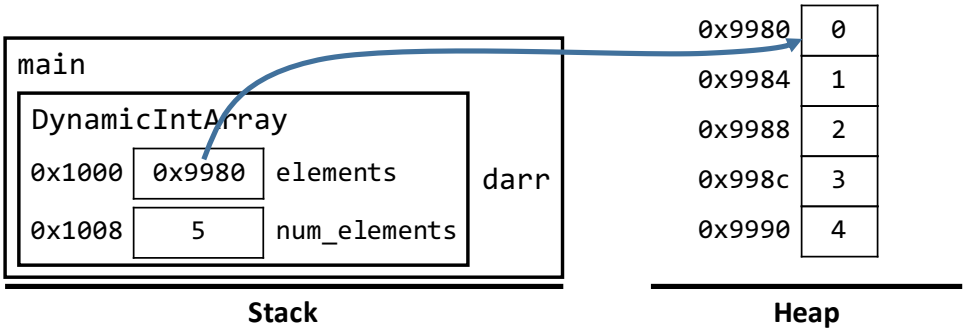

Figure 25.1: A `DynamicIntArray` object manages an array in dynamic memory.

When `darr` goes out of scope, its associated object dies, and the `DynamicIntArray` destructor is run. The destructor deletes the array, cleaning up the resource that it was using.

Thus, with the RAII pattern, we have leveraged automatic memory management and a class-type object to successfully manage a dynamic array.

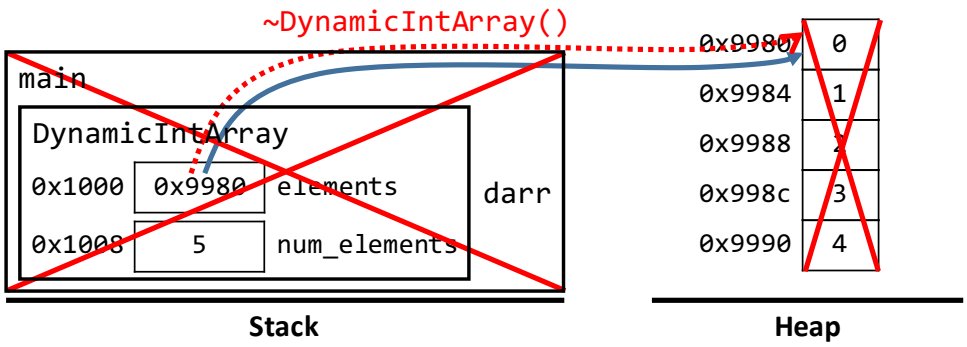

Figure 25.2: The destructor for `DynamicIntArray` deletes the array that it is managing.

25.2 Growable Set

Let's use a similar strategy to write a new version of `UnsortedSet` without a fixed-capacity limitation. We allow the set to have an arbitrary number of elements by decoupling the storage of the elements from that of the set object, using dynamic memory for the former.

We modify the data representation so that the member variable `elements` is now a pointer to the first element of a dynamic array. We also add a `capacity` member variable to keep track of the size of that dynamic array. The resulting class definition for `UnsortedSet` is as follows:

```
template <typename T>
class UnsortedSet {
public:
  // EFFECTS:  Initializes this set to be empty.
  UnsortedSet();

  // MODIFIES: *this
  // EFFECTS:  Adds value to the set, if it isn't already in the set.
  void insert(const T &value);

  // MODIFIES: *this
  // EFFECTS:  Removes value from the set, if it is in the set.
  void remove(const T &value);

  // EFFECTS:  Returns whether value is in the set.
  bool contains(const T &value) const;

  // EFFECTS:  Returns the number of elements.
  int size() const;

  // EFFECTS:  Prints out the set in an arbitrary order.
  void print(std::ostream &os) const;

private:
  // MODIFIES: *this
  // EFFECTS:  Doubles the capacity of this set.
  void grow();
```

(continues on next page)

(continued from previous page)

```
  // Initial size of a set.
  static const int INITIAL_SIZE = 5;

  T *elements;
  int capacity;
  int num_elements;
  // INVARIANTS:
  // elements points to the sole dynamic array associated with this set
  // capacity is the size of the array that elements points to
  // 0 <= num_elements && num_elements <= capacity
  // the first num_elements items in elements are the items in the set
  // the first num_elements items in elements contain no duplicates
};
```

We have added two representation invariants:

- `elements` points to the start of the sole dynamic array associated with the set, and each set has its own dynamic array
- `capacity` is the size of dynamic array that `elements` points to

We have also added two private members:

- `grow()` doubles the capacity of the set
- `INITIAL_SIZE` is a constant whose value is the initial size of a set

The following is the new constructor for `UnsortedSet`:

```
template <typename T>
UnsortedSet<T>::UnsortedSet()
  : elements(new T[INITIAL_SIZE]),
    capacity(INITIAL_SIZE),
    num_elements(0) {}
```

The constructor allocates a dynamic array of size `INITIAL_SIZE` and stores the address of its first element in `elements`. We satisfy the remaining representation invariants by setting `capacity` to `INITIAL_SIZE` and `num_elements` to 0.

We modify `insert()` so that if the set is out of space, it calls the `grow()` member function to increase the available capacity:

```
template <typename T>
void UnsortedSet<T>::insert(const T &value) {
  if (contains(value)) {
    return;
  }
  if (num_elements == capacity) {
    grow(); // double the capacity;
  }
  elements[num_elements] = value;
  ++num_elements;
}
```

The `grow()` member function doubles the capacity of the set. In C++, an array has a fixed size, even if it is dynamic, so we cannot change the size of the existing `elements` array. Instead, we allocate a new dynamic array, copy over the elements, and discard the old array, as shown in Figure 25.3.

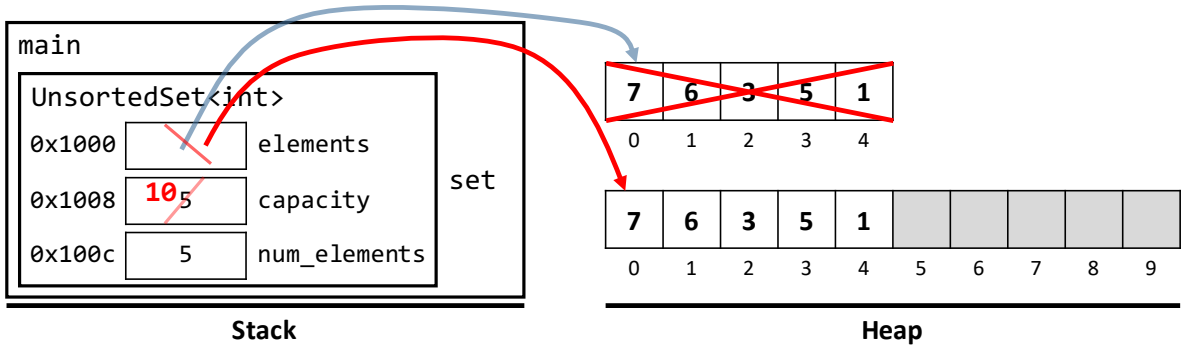

Figure 25.3: The `grow()` operation creates a new, larger array, copies the elements to the new array, and deletes the old array.

The following implements this:

```
template <typename T>
void UnsortedSet<T>::grow() {
  T *new_arr = new T[2 * capacity]; // allocate a new array that is twice as big
  for (int i = 0; i < num_elements; ++i) { // copy the elements to the new array
    new_arr[i] = elements[i];
  }
  capacity *= 2;                                          // update capacity
  delete[] elements;                                      // free the old array
  elements = new_arr; // set elements to point to first element of the new array
}
```

The function allocates a new array and deletes the old one, satisfying the invariant that there is exactly one dynamic array associated with the set. It sets `elements` to point to that array and updates `capacity` to be the array's size. Copying the elements to the new array satisfies the representation invariant that the first `num_elements` items in `elements` are the ones in the set.

The `grow()` operation demonstrates the power of indirection. By decoupling the storage for the elements from that of the `UnsortedSet` object, we have also decoupled their lifetimes. Thus, if the old storage no longer meets our needs, we can replace it with different storage that does, killing the old array and creating a new one. Compare this to storing the array directly within the memory for the set, which both limits the array to be of a fixed size and ties its lifetime to that of the set.

Since `UnsortedSet` now manages the resource of dynamic memory, we need to write a destructor that frees the resource when the `UnsortedSet` object dies.

Before we proceed to write the destructor, a clarification is in order. The destructor does not **cause** the object to die. The object dies when its lifetime ends:

- if it is associated with a local variable, when the variable goes out of scope
- if it is associated with a static variable, when the program ends
- if it is a member of a class-type object, when the latter object dies
- if it is an element of an array, when the array dies
- if it is a dynamic object, when `delete` is applied to its address

The destructor merely **runs** when the object dies – it gives the object a chance to put its affairs in order while it is on its deathbed. If the object is managing dynamic memory, it needs to release that memory.

```
template <typename T>
UnsortedSet<T>::~UnsortedSet() {
  delete[] elements;
}
```

The representation invariant that there is exactly one dynamic array associated with each `UnsortedSet` ensures that after the destructor runs, there is no longer a live array associated with the dying set.

With the destructor, we have ensured that in simple cases, `UnsortedSet` properly manages memory. However, there is one case that we missed: copying a set. Consider the following code:

```
int main() {
  UnsortedSet<int> s1;
  for (int i = 0; i < 5; ++i) {
    s1.insert(i);
  }

  UnsortedSet<int> s2 = s1;
  cout << s1 << endl;    // prints { 0, 1, 2, 3, 4 }
  cout << s2 << endl;    // prints { 0, 1, 2, 3, 4 }

  s2.insert(5);          // causes a grow

  cout << s2 << endl;    // prints { 0, 1, 2, 3, 4, 5 }
  cout << s1 << endl;    // UNDEFINED BEHAVIOR
}
```

The code creates a set `s1` and adds five items to the set, filling it to capacity. It then creates `s2` as a copy of `s1`; by default, this does a member-by-member copy, so that both `s1.elements` and `s2.elements` point to the same dynamic array. We then add an item into `s2`, causing it to grow and delete its old array. This is the same array that `s1.elements` is pointing to, so that when we proceed to print out `s1`, it accesses a dead object. The result is undefined behavior.

Had we not caused a grow, there would have been a double delete when `s2` and `s1` die, since they would both delete the same array. This still results in undefined behavior. The fundamental problem is that the copy violates the representation invariant that each set has its own array. We will see how to fix this next time.

CHAPTER

TWENTYSIX

THE BIG THREE

We saw last time that copying an `UnsortedSet` ultimately results in undefined behavior. Before we fix the issue, we need to understand how C++ copies class-type objects and what mechanisms it provides to control the behavior of a copy.

By default, a copy just copies over the members one by one, as we saw when we first discussed *class-type objects*:

```
Person elise = { "Elise", 22, true };
Person tali = elise;
```

The result is shown in Figure 26.1.

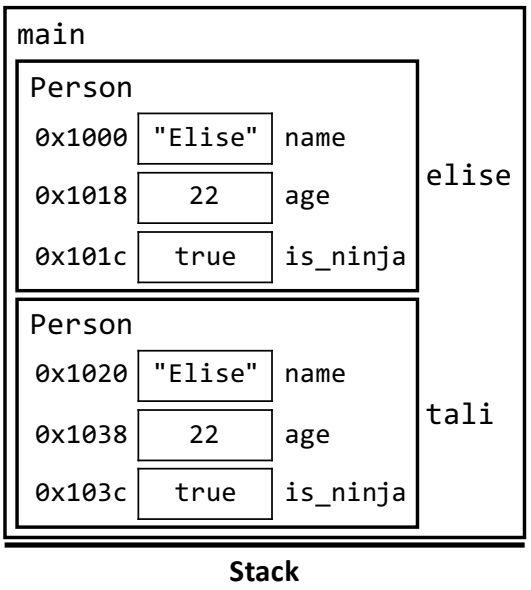

Figure 26.1: The default copy behavior for a class-type object is to copy each of its members.

The code above is an example of making a copy when initializing a new object. There are several forms of syntax we can use for initialization as a copy:

```
Person tali = elise;
Person tali(elise);
Person tali{elise};
Person tali = {elise};
```

Initialization as a copy also occurs when we pass an object by value:

```
void func(Person p);

int main() {
  Person elise = { "Elise", 22, true };
  func(elise);
}
```

The parameter `p` is associated with an object that lives in the activation record for `func()`, and the object is initialized when the function is called. It is initialized as a copy of the argument value `elise`.

Just like we can pass an object by value, we can also return an object by value, which generally makes a copy:

```
Person func2() {
  Person elise = { "Elise", 22, true };
  return elise;
}

int main() {
  Person tali = func2();
}
```

26.1 Copy Constructor

We also previously discussed that a constructor is always called when a class-type object is created (except for C-style ADTs when the members are initialized directly, like `elise` above). Copy initialization of a class-type object also invokes a constructor, specifically the *copy constructor* for the class. The following is an example of explicitly defining a copy constructor:

```
class Example {
public:
  Example(int x_in, double y_in)
    : x(x_in), y(y_in) {}

  Example(const Example &other)
    : x(other.x), y(other.y) {
    cout << "Example copy ctor: " << x << ", " << y << endl;
  }

  int get_x() const {
    return x;
  }

  double get_y() const {
    return y;
  }

private:
  int x;
  double y;
};
```

The second constructor above is the copy constructor, and it takes a reference to an existing object as the parameter.

The parameter must be passed by reference – otherwise, a copy will be done to initialize the parameter, which itself will invoke the copy constructor, which will call the copy constructor to initialize its parameter, and so on.

We have instrumented the copy constructor to print a message when it is called. Thus, the code

```
int main() {
  Example e1(2, -1.3);
  Example e2 = e1;
}
```

will print the following when run:

```
Example copy ctor: 2, -1.3
```

The program invokes the copy constructor to initialize `e2` from `e1`.

The C++ compiler provides an implicit copy constructor if a user-defined one is not present. The implicit copy constructor just does a member-by-member copy, so in the case of `Example`, it acts like the following:

```
Example(const Example &other)
  : x(other.x), y(other.y) {}
```

26.2 Assignment Operator

Assignment is another situation where an object is copied; unlike initialization, however, assignment copies into an existing object rather than a new one. The following is an example:

```
int main() {
  Example e1(2, -1.3);
  Example e2(3, 4.1);
  e2 = e1;
}
```

An assignment expression consists of a left-hand-side object, the = operator, and a right-hand-side object or value. The expression evaluates the right-hand side, copies it into the left-hand-side object, and then evaluates back to the left-hand-side object. We can then use the result in a larger expression:

```
cout << (e2 = e1).get_x() << endl;  // prints 2
```

Like most operators, the assignment operator can be overloaded; the overload must be a member function of the type of the left-hand side.

```
class Example {
public:
  Example & operator=(const Example &rhs);
  ...
};
```

The function takes in an `Example` by reference to const, corresponding to the right-hand operand of the assignment.[1] The return value will be the left-hand-side object itself, and it needs to be returned by reference rather than by value (the latter would make a copy rather than returning the object itself). Thus, the return type is `Example &`.

The following is a definition of the overloaded assignment operator that just does a member-by-member copy:

[1] In C++, a reference to const can bind to a temporary, allowing the right-hand operand of the assignment to be a value rather than an object.

```
Example & Example::operator=(const Example &rhs) {
  x = rhs.x;
  y = rhs.y;
  return *this;
}
```

The two members are individually copied from the right-hand side. The left-hand-side object must be returned; we need to dereference the `this` pointer to get to the object to which it is pointing.

Like the copy constructor, the compiler provides an implicit definition of the assignment operator if a user-defined one is not present. Like the implicitly defined copy constructor, the implicit assignment operator performs a member-by-member copy.

26.3 Shallow and Deep Copies

For most class types, a member-by-member copy is sufficient, and there is no need to write a custom copy constructor or assignment operator. However, for a type that manages a dynamic resource, a member-by-member copy generally results in incorrect sharing of a resource. For example, consider the following code that copies an `UnsortedSet`:

```
int main() {
  UnsortedSet<int> s1;
  for (int i = 0; i < 5; ++i) {
    s1.insert(i);
  }

  UnsortedSet<int> s2 = s1;
  cout << s1 << endl;    // prints { 0, 1, 2, 3, 4 }
  cout << s2 << endl;    // prints { 0, 1, 2, 3, 4 }

  s2.insert(5);          // causes a grow

  cout << s2 << endl;    // prints { 0, 1, 2, 3, 4, 5 }
  cout << s1 << endl;    // UNDEFINED BEHAVIOR
}
```

The initialization of `s2` calls the implicit copy constructor, which does a member-by-member copy, as if it were written as follows:

```
template <typename T>
UnsortedSet<T>::UnsortedSet(const UnsortedSet &other)
  : elements(other.elements), capacity(other.capacity),
    num_elements(other.num_elements) {}
```

The result is that `s1.elements` and `s2.elements` point to the same array, as depicted by Figure 26.2.

Inserting 5 into `s2` causes a grow operation, which creates a new array and deletes the old one. The result is shown in Figure 26.3.

Then printing out `s1` accesses the old, deleted array, resulting in undefined behavior.

The fundamental problem is that the implicitly defined member-by-member copy is a *shallow copy*: it copies the pointer to the array of elements, rather than following it and copying the array as a whole. This violates the representation invariant that each set has its own array. Instead, we need a *deep copy*, where we make a copy of the underlying

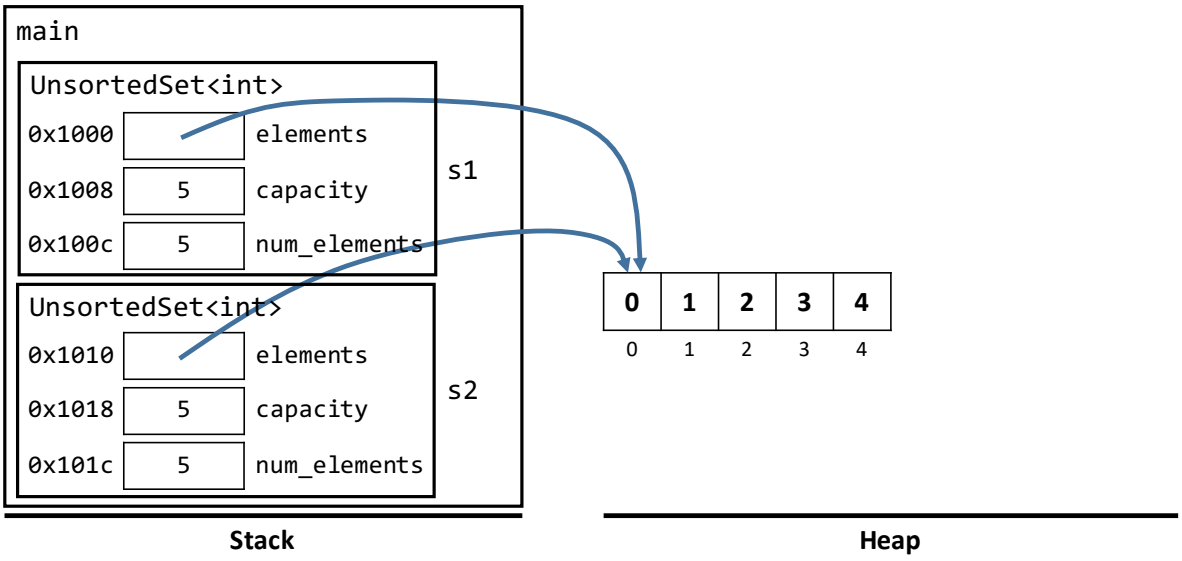

Figure 26.2: The implicit copy constructor copies each member one by one, resulting in a shallow copy.

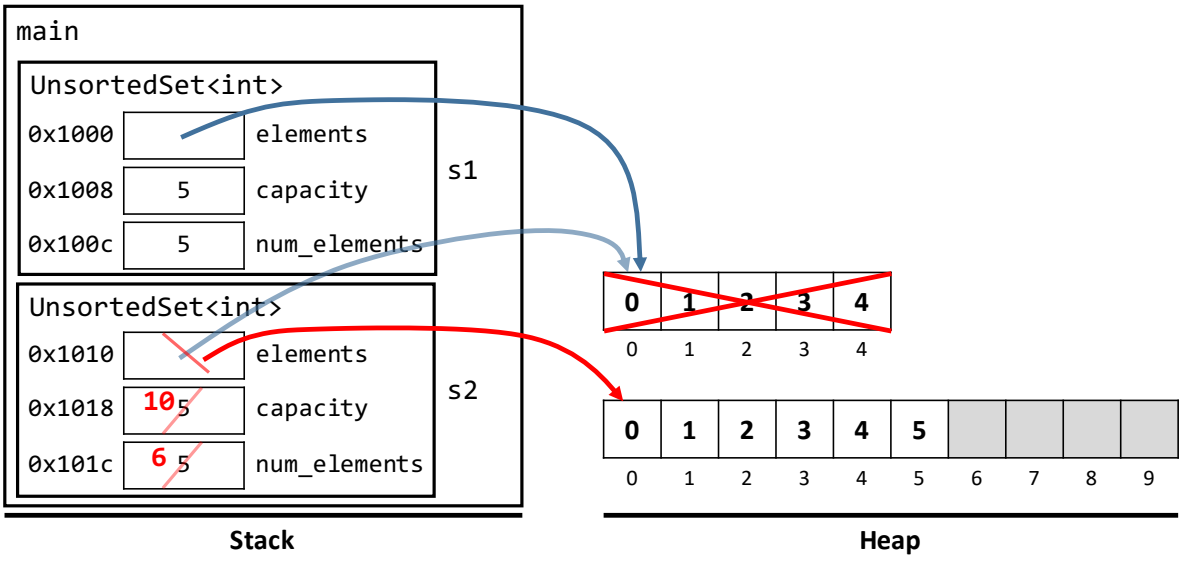

Figure 26.3: A subsequent `grow()` results in one of the sets using a dead array.

resource rather than having the two sets share the same resource. To obtain a deep copy, we need to provide a custom implementation of the copy constructor:

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet(const UnsortedSet &other);
  ...
};

template <typename T>
UnsortedSet<T>::UnsortedSet(const UnsortedSet &other)
  : elements(new T[other.capacity]),        // create new array
    capacity(other.capacity),               // shallow copy non-resources
    num_elements(other.num_elements) {
  for (int i = 0; i < num_elements; ++i) {  // copy over the elements
    elements[i] = other.elements[i];
  }
}
```

Rather than copying the `elements` pointer, we initialize the new set's member to point to the start of a dynamic array of the same capacity as `other`'s. The members that don't represent resources are just copied directly (`capacity` and `num_elements`). The body of the constructor copies each element from the old set to the new one. The result is that each set has its own, independent copy of the elements, as shown in Figure 26.4.

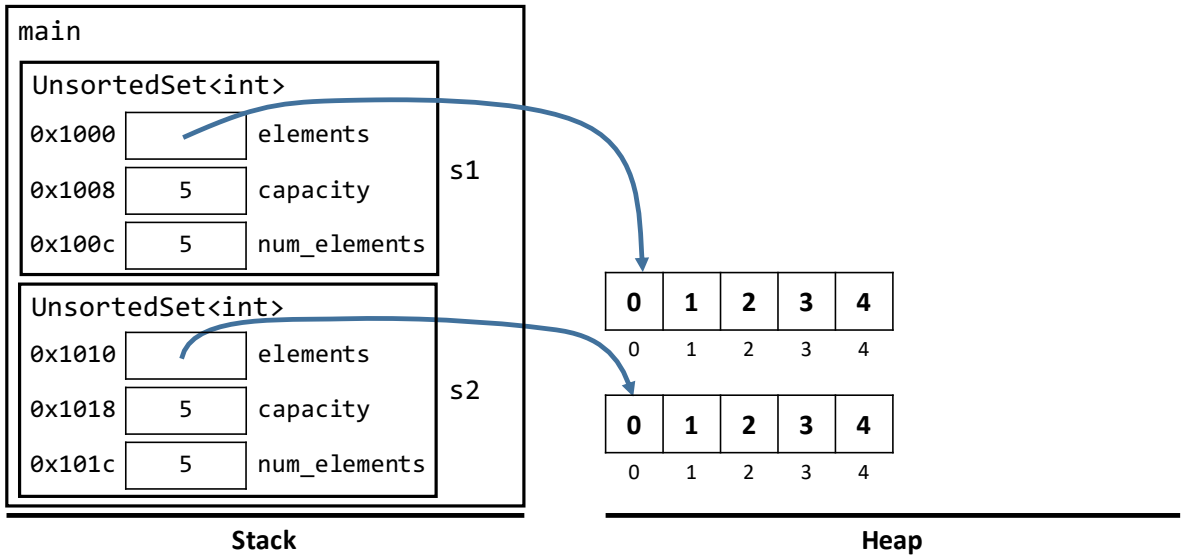

Figure 26.4: The custom copy constructor for `UnorderedSet` performs a deep copy, providing the new set with its own array.

The custom copy constructor provides a deep copy in the case of initializing a new set as a copy. We need a deep copy in assignment as well:

```
s2 = s1;
```

Thus, we need a custom assignment operator in addition to a custom copy constructor. The following is a first attempt at defining one:

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet & operator=(const UnsortedSet &rhs);
  ...
};

template <typename T>
UnsortedSet<T> & UnsortedSet<T>::operator=(const UnsortedSet &rhs) {
  delete[] elements;                          // delete old array

  elements = new T[rhs.capacity];             // make new array of the required size
  capacity = rhs.capacity;                    // shallow copy non-resources
  num_elements = rhs.num_elements;

  for (int i = 0; i < num_elements; ++i) {    // copy over the elements
    elements[i] = rhs.elements[i];
  }

  return *this;                               // return LHS object
}
```

The function first deletes the old array before creating a new one. It may be the case that the size of the `rhs` set is larger than the capacity of the set receiving the copy, in which case creating a new array is necessary. The function then makes a shallow copy of the non-resources, followed by copying over each of the elements. Finally, it returns the left-hand-side object.

While this assignment operator works in most cases, it fails in the case of *self assignment*. An expression such as `s2 = s2` will delete `s2.elements` before proceeding to access the elements in the subsequent loop. Instead, the assignment should have no effect when both operands are the same object, so we need to check for this case before doing any work. We do so as follows:

```
template <typename T>
UnsortedSet<T> & UnsortedSet<T>::operator=(const UnsortedSet &rhs) {
  if (this == &rhs) {                         // self-assignment check
    return *this;
  }

  delete[] elements;                          // delete old array

  elements = new T[rhs.capacity];             // make new array of the required size
  capacity = rhs.capacity;                    // shallow copy non-resources
  num_elements = rhs.num_elements;

  for (int i = 0; i < num_elements; ++i) {    // copy over the elements
    elements[i] = rhs.elements[i];
  }

  return *this;                               // return LHS object
}
```

The `this` pointer points to the left-hand-side operand, while the parameter `rhs` is an alias for the right-hand-side operand. We need to obtain the address of `rhs` to compare to the address stored in the `this` pointer. If the two addresses are the same, then the two operands are the same object, so we immediately return the left-hand-side object.

(We cannot return `rhs` because it is declared as a reference to const, while the return type is not.)

26.4 The Law of the Big Three

We have seen that the implicitly defined copy constructor and assignment operator both do a shallow copy, and that this behavior is incorrect for classes that manage a dynamic resource. Instead, we need a deep copy, which requires us to write our own custom versions of the two functions.

We also saw last time that resource management requires us to write our own custom destructor as well. It is not a coincidence that we needed to write custom versions of all three of the destructor, copy constructor, and assignment operator. The *Law of the Big Three* (also know as the *Rule of Three*) is a rule of thumb in C++ that if a custom version of any of the destructor, copy constructor, or assignment operator is required, almost certainly all three need to be custom. We refer to these three members as the *big three*.

By default, C++ provides implicit definitions of each of the big three:

- The implicitly defined destructor does no special cleanup; it is equivalent to a destructor with an empty body. Like other destructors, it does implicitly destroy the members as well as the base class, if there is one.
- The implicitly defined copy constructor does a member-by-member shallow copy.
- The implicitly defined assignment operator does a member-by-member shallow copy.

When a class manages a resource, however, the programmer must provide custom versions of the big three:

- The destructor should free the resources.
- The copy constructor should make a deep copy of the resources and shallow copy the non-resources.
- The assignment operator should:
  - Do a self-assignment check.
  - Free the old resources.
  - Make a deep copy of the right-hand-side object's resources.
  - Shallow copy the non-resources from the right-hand-side object.
  - Return the left-hand-side object with `*this`.

26.5 Example: Calls to the Big Three

To better understand when the big three are invoked, we instrument the big three for the following class to print a message:

```
class Foo {
public:
  Foo(const string &str_in) : str(str_in) {
    cout << "Foo ctor " << str << endl;
  }

  Foo(const Foo &other) : str(other.str) {
    cout << "Foo copy ctor " << str << endl;
  }

  Foo & operator=(const Foo &rhs) {
```

(continues on next page)

(continued from previous page)

```
    cout << "Foo assign " << rhs.str << " to " << str << endl;
    str = rhs.str;
    return *this;
  }

  ~Foo() {
    cout << "Foo dtor " << str << endl;
  }

private:
  string str;
};
```

The `Foo` class has a `str` member variable that we will use to distinguish between different `Foo` objects. The constructors, assignment operator, and destructor all print the value of `str`.

Let us take a look at a small program that uses `Foo`:

```
void func(const Foo &x, Foo y) {
  Foo z = x;
}

int main() {
  Foo a("apple");
  Foo b("banana");
  Foo c("craisin");

  func(a, b);

  Foo c2 = c;
  c2 = c;
}
```

This prints the following when run:

```
Foo ctor apple
Foo ctor banana
Foo ctor craisin
Foo copy ctor banana
Foo copy ctor apple
Foo dtor apple
Foo dtor banana
Foo copy ctor craisin
Foo assign craisin to be craisin
Foo dtor craisin
Foo dtor craisin
Foo dtor banana
Foo dtor apple
```

In `main()`, the variables `a`, `b`, and `c` are constructed in order, resulting in the `Foo ctor` outputs. Then `func()` is called. The first parameter is passed by reference, so no copy is made. The second parameter is passed by value, so the parameter object is initialized as a copy of the argument. This invokes the copy constructor, which prints `Foo copy ctor banana`. Within the body of `func()`, `z` is initialized as a copy of `x`, resulting in `Foo copy ctor apple`. When `func()` returns, the local objects are destructed in reverse order of construction: `z` is destructed first, producing `Foo`

`dtor apple`, then `y` is destructed, printing `Foo dtor banana`.

Continuing in `main()`, `c2` is initialized as a copy of `c`, invoking the copy constructor and printing `Foo copy ctor craisin`. In the next line, `c2` already exists, so the expression is an assignment and calls the assignment operator, which prints `Foo assign craisin to craisin`. At the end of `main()`, the local objects are destructed in reverse order: `c2`, then `c`, then `b`, and then `a`. The last four lines of output are the result.

26.6 Destructors and Polymorphism

We saw previously that applying the `delete` operator to a pointer follows the pointer and kills the object at the given address. If the object is of class type, its destructor is run. However, a subtle issue arises when the static and dynamic type of the object do not match: does the destructor use static or dynamic binding? Consider the following example:

```
class Base {
public:
  virtual void add(int x) = 0;

  ~Base() {
    cout << "Base dtor" << endl;
  }
};

class Derived : public Base {
public:
  void add(int x) override {
    items.push_back(x);
  }

  ~Derived() {
    cout << "Derived dtor" << endl;
  }

private:
  vector<int> items;
};

int main() {
  Base *bptr = new Derived;
  bptr->add(3);
  delete bptr;
}
```

Running this code results in:

```
$ ./a.out
Base dtor
```

The destructor is statically bound, so `~Base()` is invoked rather than `~Derived()`. This is problematic: even though `Derived` itself is not managing dynamic memory, its member variable `items` is. Since the destructor for `Derived` was not called, the members introduced by `Derived` were also not destructed. Running Valgrind on the program illustrates the issue:

```
$ valgrind --leak-check=full ./a.out
==13359== Memcheck, a memory error detector
==13359== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13359== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13359== Command: ./a.out
==13359==
Base dtor
==13359==
==13359== HEAP SUMMARY:
==13359==     in use at exit: 4 bytes in 1 blocks
==13359==   total heap usage: 4 allocs, 3 frees, 73,764 bytes allocated
==13359==
==13359== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==13359==    at 0x4C3017F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_
↪memcheck-amd64-linux.so)
==13359==    by 0x1098CD: __gnu_cxx::new_allocator<int>::allocate(unsigned long, void␣
↪const*) (in /home/akamil/tmp/a.out)
==13359==    by 0x109782: std::allocator_traits<std::allocator<int> >
↪::allocate(std::allocator<int>&, unsigned long) (in /home/akamil/tmp/a.out)
==13359==    by 0x1095BB: std::_Vector_base<int, std::allocator<int> >::_M_
↪allocate(unsigned long) (in /home/akamil/tmp/a.out)
==13359==    by 0x109163: void std::vector<int, std::allocator<int> >::_M_realloc_insert
↪<int const&>(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> >␣
↪>, int const&) (in /home/akamil/tmp/a.out)
==13359==    by 0x109035: std::vector<int, std::allocator<int> >::push_back(int const&)␣
↪(in /home/akamil/tmp/a.out)
==13359==    by 0x108F65: Derived::add(int) (in /home/akamil/tmp/a.out)
==13359==    by 0x108E63: main (in /home/akamil/tmp/a.out)
==13359==
==13359== LEAK SUMMARY:
==13359==    definitely lost: 4 bytes in 1 blocks
==13359==    indirectly lost: 0 bytes in 0 blocks
==13359==      possibly lost: 0 bytes in 0 blocks
==13359==    still reachable: 0 bytes in 0 blocks
==13359==         suppressed: 0 bytes in 0 blocks
==13359==
==13359== For counts of detected and suppressed errors, rerun with: -v
==13359== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

The error message indicates that the memory allocated by the `items` vector is never freed.

The solution is to use dynamic binding for the destructor instead by declaring it as virtual:

```
class Base {
public:
  virtual void add(int x) = 0;

  virtual ~Base() {
    cout << "Base dtor" << endl;
  }
};

class Derived : public Base {
```

(continues on next page)

(continued from previous page)

```
public:
  void add(int x) override {
    items.push_back(x);
  }

  ~Derived() {                          // virtual implicitly inherited
    cout << "Derived dtor" << endl;
  }
private:
  vector<int> items;
};

int main() {
  Base *bptr = new Derived;
  bptr->add(3);
  delete bptr;
}
```

Now the following is printed when the code is run:

```
$ ./a.out
Derived dtor
Base dtor
```

Valgrind also no longer reports an error.

In general, **a base class should always declare the destructor to be virtual if the class will be used polymorphically** (meaning that a base-class pointer or reference may be bound to a derived-class object). If the base class destructor has no work to do, it may be defaulted. The derived class destructor need not be explicitly defined, since "virtualness" is inherited even if the destructor is implicitly defined by the compiler.

```
class Base {
public:
  virtual void add(int x) = 0;

  virtual ~Base() = default;   // use implicitly defined dtor, but make it virtual
};

class Derived : public Base {
public:
  void add(int x) override {
    items.push_back(x);
  }
                               // implicitly defined dtor inherits virtual
private:
  vector<int> items;
};

int main() {
  Base *bptr = new Derived;
  bptr->add(3);
  delete bptr;
}
```

Valgrind does not show a memory leak:

```
$ valgrind --leak-check=full ./a.out
==13479== Memcheck, a memory error detector
==13479== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13479== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13479== Command: ./a.out
==13479==
==13479==
==13479== HEAP SUMMARY:
==13479==     in use at exit: 0 bytes in 0 blocks
==13479==   total heap usage: 3 allocs, 3 frees, 72,740 bytes allocated
==13479==
==13479== All heap blocks were freed -- no leaks are possible
==13479==
==13479== For counts of detected and suppressed errors, rerun with: -v
==13479== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

CHAPTER

TWENTYSEVEN

LINKED LISTS

A *sequential container* is a container that holds elements and allows them to be accessed in sequential order. The simplest sequential container is a built-in array, which stores its elements contiguously in memory:

```
const int SIZE = 5;
int array[SIZE] = { 1, 2, 3, 4, 5 };
// iterate over elements in order with traversal by pointer
for (int *ptr = array; ptr != array + SIZE; ++ptr) {
  cout << *ptr << endl;
}
```

We can also traverse over arrays by index, using *random access* to get to an element:

```
const int SIZE = 5;
int array[SIZE] = { 1, 2, 3, 4, 5 };
// iterate over elements in order with traversal by index
for (int i = 0; i < SIZE; ++i) {
  cout << array[i] << endl;
}
```

The fact that the elements are stored contiguously makes random access efficient: it translates directly to pointer arithmetic, so that it takes the same amount of time to access an element in the middle of the array as one at the beginning.

```
cout << *(array + i) << endl;
```

However, arrays have significant limitations: their size is fixed upon creation, and for non-dynamic arrays, the size must be known at compile time.

A *vector* is a class template that provides an abstraction over a dynamically allocated array. Like `UnsortedSet`, it grows as needed by swapping out an old array for a new one. A vector also provides an interface for iterating over the elements in order:

```
vector<int> data;
// add elements to the vector
for (int i = 1; i <= 5; ++i) {
  data.push_back(i);
}
// iterate over elements in order with traversal by iterator
for (vector<int>::iterator it = data.begin(); it != vec.end(); ++it) {
  cout << *it << endl;
}
```

We will cover *iterators and traversal by iterator* next time.

Sequence abstractions based on contiguous memory have the drawback of inefficient insertion at the beginning or middle of the sequence. For instance, to insert an item at a particular position, we need to first move the elements at that position and later out of the way, as shown in Figure 27.1.

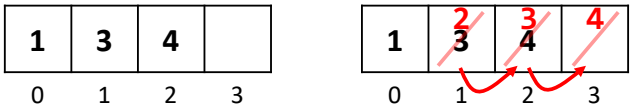

Figure 27.1: Inserting in the middle of a contiguous data structure requires elements to be shifted out of the way.

Thus, insertion at the beginning or middle is a linear-time operation.

On the other hand, if we give up the abstraction of contiguous memory, we can place a new item anywhere in memory. Since the elements are no longer stored contiguously, we cannot move to the next element by just incrementing an address. Instead, we have to explicitly keep track of where the next element is through a pointer.

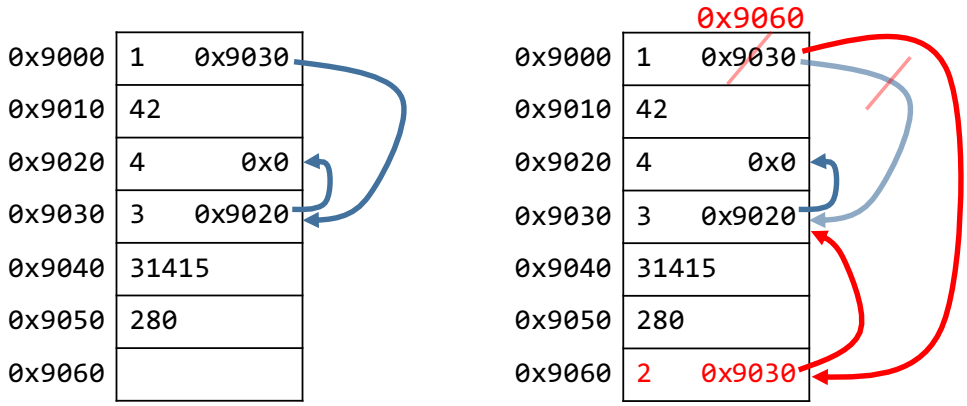

Figure 27.2: A noncontiguous data structure can use pointers to keep track of each element's location. Inserting in the middle can be done by just modifying a pointer.

As illustrated in Figure 27.2, inserting in the middle now just involves changing the value of a pointer.

For each element in our sequence, we need to keep track of both the datum itself as well as a pointer to the next piece of the sequence. This is heterogeneous data, so we use a *node* class-type object to keep track of the two values:

```
struct Node {
  int datum;
  Node *next;
};
```

We define the `Node` type as plain-old data, using the `struct` keyword and accessing the members directly. For now, we will stick to `int` as the element type; later, we will convert our abstraction into a template. The remaining `Node` member is a pointer to the next `Node` in our sequence.

The sequence of `Node` objects in Figure 27.3 represents the elements `( 1 2 3 4 )`.

We use a null pointer as a *sentinel* denoting the end of the sequence, storing that in the `next` pointer of the last node. We refer to the sequence of nodes as a *linked list*, as it consists of a list of items linked together by pointers.

Rather than exposing the nodes directly as part of an abstract data type, we define an `IntList` ADT that internally maintains a sequence of nodes. It need only keep track of the location of the first `Node`, since we can get to the others by following `next` pointers:

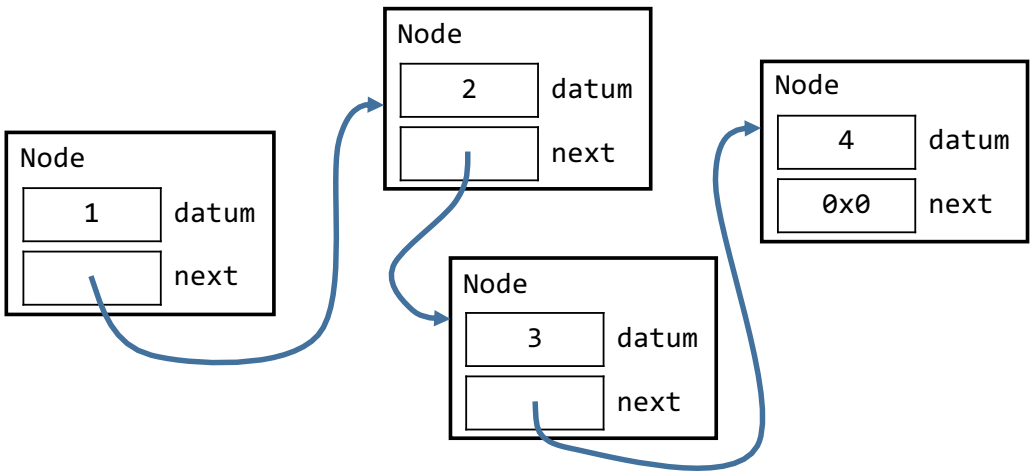

Figure 27.3: Nodes representing a sequence of four elements. A null pointer represents the end of the sequence.

```
class IntList {
  ...
private:
  Node *first;
}
```

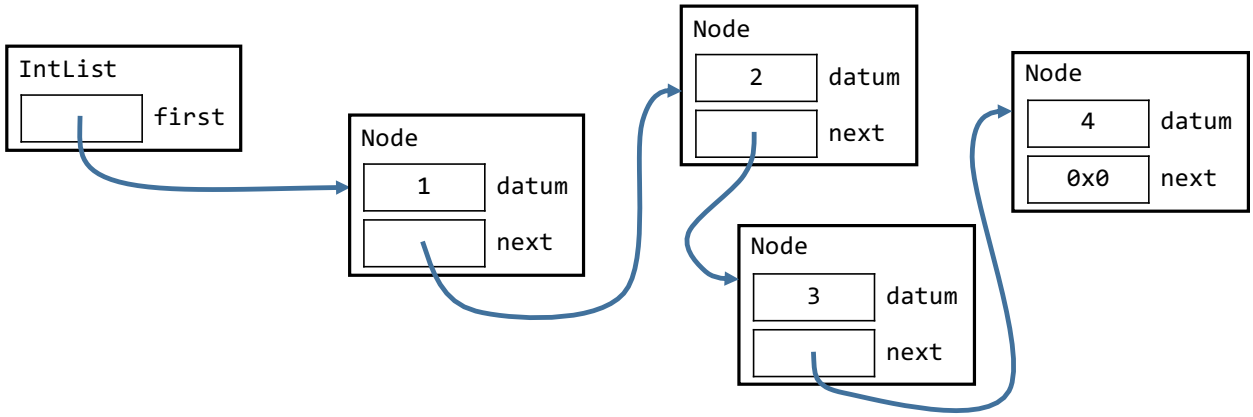

Figure 27.4: An `IntList` object and its nodes.

Since the `Node` objects are stored indirectly from the `IntList` object, they must be in dynamic memory, and `IntList` must ensure that their memory is properly managed.

Before we consider the representation of `IntList` further, we first write some code that uses `IntList` in order to work out its interface:

```
int main() {
  IntList list;             // ( )
  list.push_front(1);       // ( 1 )
  list.push_front(2);       // ( 2 1 )
  list.push_front(3);       // ( 3 2 1 )

  cout << list.front();     // 3
```

(continues on next page)

(continued from previous page)

```
  list.front() = 4;          // ( 4 2 1 )

  list.print(cout);          // 4 2 1

  list.pop_front();          // ( 2 1 )
  list.pop_front();          // ( 1 )
  list.pop_front();          // ( )

  cout << list.empty();      // true (or 1)
}
```

The class should have a default constructor that makes the list empty. It should also have a `push_front()` member function to insert an item at the beginning, as well as `front()` to retrieve the first item. The latter must support both reading and writing the first element. The remaining member functions are `print()`, `pop_front()` to remove the first item, and `empty()`. The following is the resulting interface:

```
class IntList {
public:
  // EFFECTS: Constructs an empty list.
  IntList();

  // EFFECTS:  Returns true if the list is empty.
  bool empty() const;

  // REQUIRES: the list is not empty
  // EFFECTS:  Returns (by reference) the first element.
  int & front();

  // EFFECTS:  Inserts datum at the front of the list.
  void push_front(int datum);

  // REQUIRES: the list is not empty
  // EFFECTS:  Removes the first element.
  void pop_front();

  // MODIFIES: os
  // EFFECTS:  Prints the items in the list, each followed by a space.
  void print(std::ostream &os) const;

  ...
};
```

Notice that `Node` appears nowhere in the interface. It is an implementation detail, so it should be defined as a private member of `IntList`. This makes it a *nested class*, which is a class (or struct) defined as a member of another class.

```
class IntList {
  ...
private:
  struct Node {
    int datum;
    Node *next;
```

(continues on next page)

(continued from previous page)

```
  };

  Node *first;
};
```

Now that we have a representation, the next step is to determine the representation invariants. We have already decided to use a null pointer to denote the end of the sequence of nodes. Similarly, we can store a null pointer in `first` to represent an empty list. Thus, the invariants are as follows:[1]

- `first` is either null or a pointer to a valid `Node`
- the `next` member of all but the last node points to another valid `Node`
- the `next` member of the last node is null

We can now proceed to implement the constructor and the remaining member functions.

The constructor must ensure that the list is initialized to satisfy the representation invariants. Since the default constructor makes the list empty, it must initialize `first` to be null:

```
IntList::IntList() : first(nullptr) {}
```

The `empty()` function just needs to check whether `first` is null:

```
bool IntList::empty() const {
  return first == nullptr;     // or just return !first;
}
```

For `front()`, we will first assert the REQUIRES clause that the list not be empty. If that is satisfied, the representation invariants tell us that `first` is pointing to a valid node. Its `datum` member holds the first element.

```
int & IntList::front() {
  assert(!empty());
  return first->datum;
}
```

The return by reference is necessary to allow code such as:

```
list.front() = 4;
```

The left-hand side of an assignment must be an object, not a temporary value, so `front()` must return an object by reference.

For `push_front()`, we need to consider two cases:

- The list is empty, in which case `first` is null.
- The list is not empty, in which case `first` points to a valid node.

In both cases, we construct a new `Node` in dynamic memory, set its `datum` to be the value that is being inserted, set its `next` to be the existing value of `first`, and set `first` to point to the new `Node`.

```
void IntList::push_front(int datum) {
  Node *p = new Node;
  p->datum = datum;
```

(continues on next page)

[1] There are two further invariants: a node is associated with exactly one list, and there are no cycles among the `next` pointers of the nodes. The latter makes it invalid, for instance, to have one node's `next` point to a second node and have the second's `next` point back to the first node.

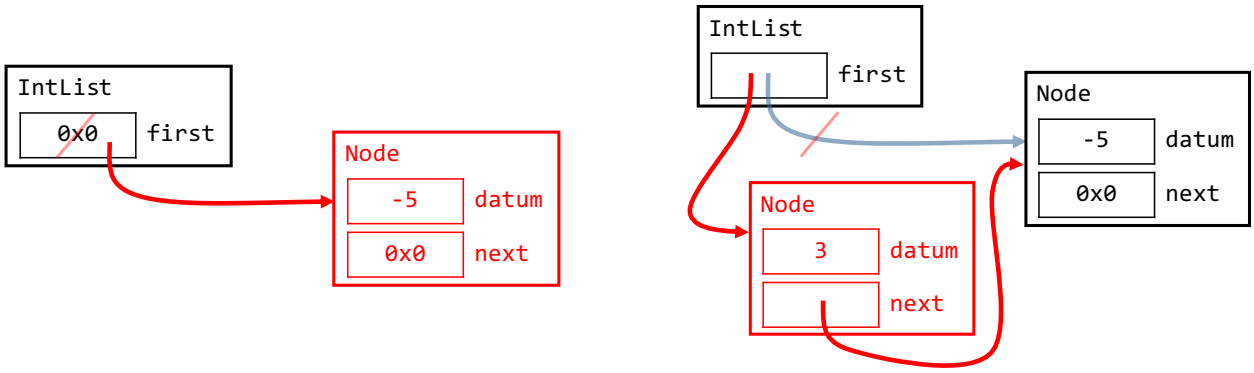

Figure 27.5: Inserting an element to the front of an empty and non-empty list.

(continued from previous page)

```
  p->next = first;
  first = p;
}
```

A more succinct way to accomplish this is with an initializer list that directly initializes each member of the new `Node`; this is permitted since `Node` is a C-style ADT.

```
void IntList::push_front(int datum) {
  first = new Node{ datum, first };
}
```

In `pop_front()`, we again assert the REQUIRES clause. We should then consider two possible cases:

- The list has one item, in which case it will end up as empty.
- The list has more than one item, and it will not end up empty.

In the former case, the sole node is the last one, and the representation invariants tell us that its `next` is null. Thus, setting `first` to the node's `next` suitably makes the list empty. In the second case, the first node's `next` is pointing to the second node in the list, so `first` should end up pointing at that node when the first one is removed. Thus, in both cases, we need to assign the value of `first->next` to `first`.

We also need to properly manage dynamic memory; in particular, we must delete the node that is being removed, as it is memory that no longer will be used.

Consider the following implementation of `pop_front()`:

```
void IntList::pop_front() {
  assert(!empty());
  delete first;
  first = first->next;
}
```

This code dereferences `first` through the arrow operator after the underlying object has been deleted, resulting in undefined behavior. Let's try to fix this by reordering the statements:

```
void IntList::pop_front() {
  assert(!empty());
  first = first->next;
```

(continues on next page)

(continued from previous page)

```
  delete first;
}
```

Now, the assignment to `first` orphans what previously was the first node; we no longer have a means to get to that object, resulting in a memory leak. The code then proceeds to delete what used to be the second node.

Neither ordering works correctly on its own. What we need is a temporary to keep track of the first node, so that we can change the value of `first` but still be able to delete the node that is being removed. The following are two correct implementations:

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = first;           // temporary keeps track of old first
  first = first->next;
  delete victim;
}
```

```
void IntList::pop_front() {
  assert(!empty());
  Node *new_first = first->next;  // temporary keeps track of new first
  delete first;
  first = new_first;
}
```

Figure 27.6 illustrates the first implementation of `pop_front()`.

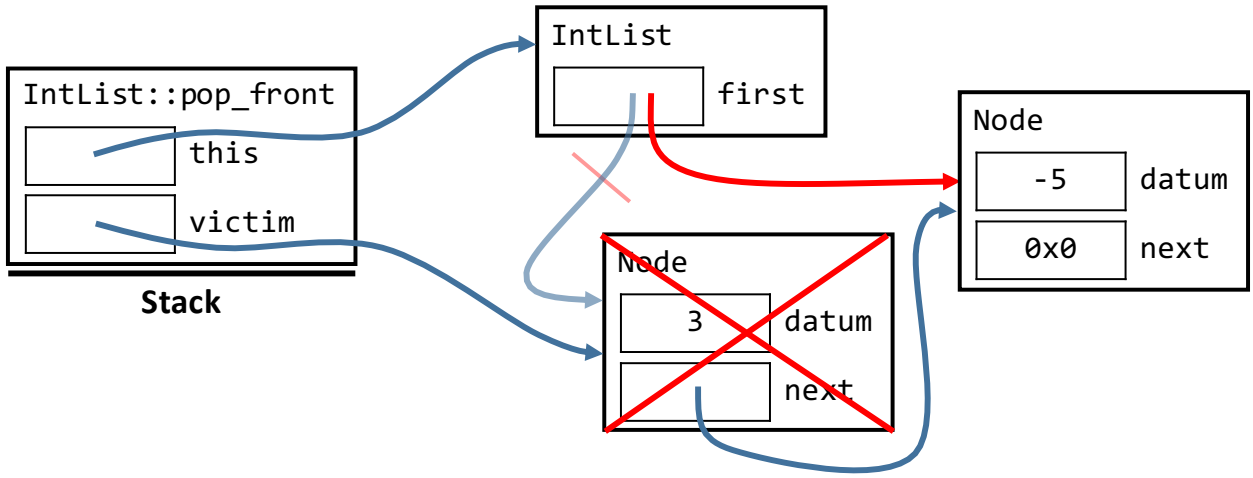

Figure 27.6: Removing an element from the front of a list.

27.1 Traversing a Linked List

Iterating over a linked list's elements from outside the class requires an *iterator* abstraction that we will see next time: the `Node` struct is private, so external code cannot use `Node`s to iterate through the list. Code within `IntList`, on the other hand, does have access to `Node`, so it can traverse the list by starting at the `first` member and following each node's `next` pointer until reaching the null sentinel. The following `print()` member function uses this strategy:

```
void IntList::print(std::ostream &os) const {
  for (Node *node_ptr = first; node_ptr; node_ptr = node_ptr->next) {
    os << node_ptr->datum << " ";
  }
}
```

The loop initializes `node_ptr` as a copy of `first`. If the list is empty, `first` is null, so `node_ptr` will be initialized to null as well. The truth value of a null pointer is false, so the condition of the loop will evaluate to false and the loop will exit. (Alternatively, `node_ptr` can be compared to `nullptr` instead: `node_ptr != nullptr`.)

If the list is not empty, `node_ptr` will not be initially null, and its truth value will be true. The body then executes, and it uses the `->` operator to access the `datum` member of the node that `node_ptr` points to. The loop update then copies the node's `next` pointer into `node_ptr`, moving `node_ptr` to point to the next node. When the iteration reaches the last node in the list, its `next` is null, so the update sets `node_ptr` to be null. This results in the loop condition being false, so the loop exits.

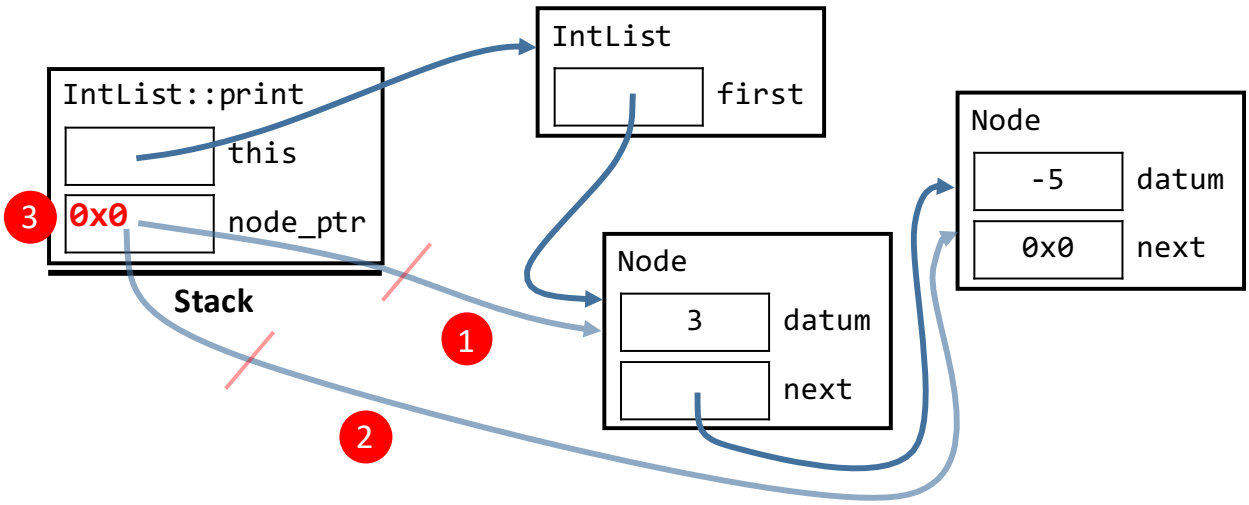

Figure 27.7: Traversal of the nodes in a list, starting at `first`, following `next` pointers, and ending at the null sentinel.

27.2 Linked List Big Three

Inserting an element into a list allocates a `Node` in dynamic memory, so the list must properly manage the node objects to avoid a memory leak. The compiler-generated destructor does not free the nodes' memory, so we need to write a custom destructor instead. The *law of the big three* then tells us that we need to write a custom copy constructor and assignment operator as well.

Both the destructor and the assignment operator must free the list's resources, and both the copy constructor and assignment operator perform a deep copy of another list's resources. To avoid code duplication, we write a `pop_all()` member function to free all the elements and their nodes, and a `push_all()` function to copy all the elements from another list. These functions are not part of the interface, so we write them as private members:

```
class IntList {
  ...
private:
  // EFFECTS:  Removes all the elements from this list.
  void pop_all();

  // EFFECTS:  Adds all elements from other into this list.
  void push_all(const IntList &other);
};
```

With these two functions, we can write the big three as follows:

```
IntList::IntList(const IntList &other) : IntList() {
  push_all(other);
}

IntList & IntList::operator=(const IntList &rhs) {
  if (this != &rhs) {
    pop_all();
    push_all(rhs);
  }
  return *this;
}

IntList::~IntList() {
  pop_all();
}
```

The copy constructor *delegates* to the default constructor to make the list empty, then calls `push_all()` to copy all the elements from the other list. The assignment operator does a self-assignment check, calls `pop_all()` to remove all the existing elements, and invokes `push_all()` to copy the elements from the other list. The destructor just calls `pop_all()` to remove all the elements and free their associated nodes.

The implementation of `pop_all()` is straightforward: we already have a `pop_front()` function that removes a single item, so we just need to repeatedly call it until the list is empty:

```
void IntList::pop_all() {
  while (!empty()) {
    pop_front();
  }
}
```

The `push_all()` function needs to iterate over each of the elements in the other list, adding them one by one to the current list. We can follow the same iteration pattern we used in `print()`, and we have a `push_front()` function that adds a single element:

```
void IntList::push_all(const IntList &other) {
  for (Node *node_ptr = other.first; node_ptr; node_ptr = node_ptr->next) {
    push_front(node_ptr->datum);
  }
}
```

However, this implementation ends up inserting the elements in reverse order: if `other` contains the elements `( 1 2 3 )`, the code would insert 1, then 2 before that, then 3 before that, resulting in `( 3 2 1 )`.

Rather than inserting each element at the beginning of the list, we need to insert at the end with a `push_back()` function.

```
void IntList::push_all(const IntList &other) {
  for (Node *node_ptr = other.first; node_ptr; node_ptr = node_ptr->next) {
    push_back(node_ptr->datum);
  }
}
```

27.3 Insertion and Removal at the End

With our current list representation, `push_back()` must traverse the entire list to insert the new element:

```
void IntList::push_back(int datum) {
  Node *new_node = new Node{ datum, nullptr };
  if (empty()) {
    first = new_node;
  } else {
    Node *node_ptr = first;
    for (; node_ptr->next; node_ptr = node_ptr->next);  // find last node
    node_ptr->next = new_node;        // set last node's next to new_node
  }
}
```

This is a very inefficient algorithm; the core problem is our list implementation only keeps track of the first node, but we need to insert after the last node. We can make `push_back()` much more efficient by changing our list representation to keep track of the last node as well:

```
class IntList {
  ...
private:
  Node *first;
  Node *last;
};
```

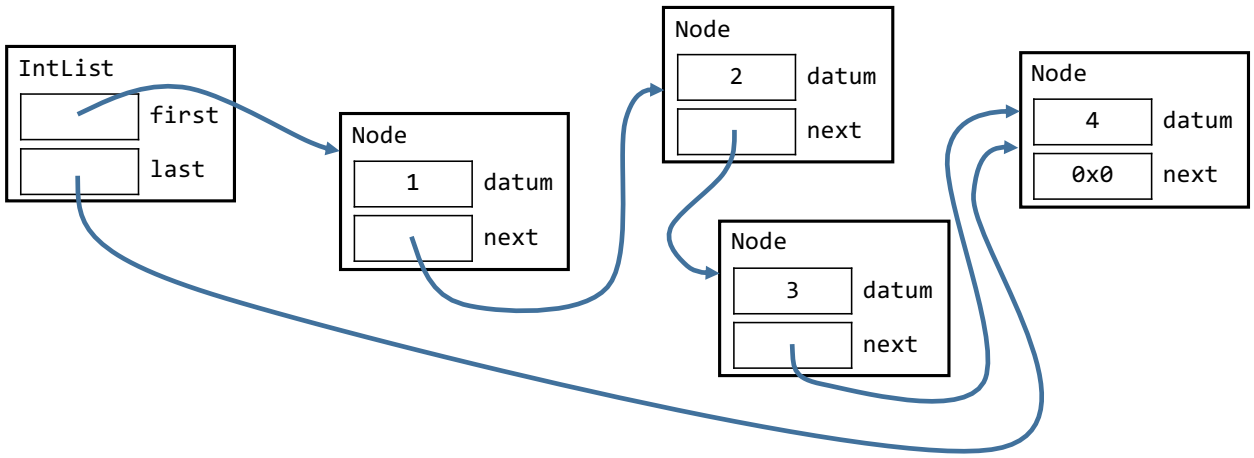

Figure 27.8: Double-ended list representation with `first` and `last` pointers.

We update our representation invariants so that an empty list is represented by both `last` and `first` being null, and that `last` points to the last node in the sequence for a non-empty list.

We can then write `push_back()` as follows:

```
void IntList::push_back(int datum) {
  Node *new_node = new Node{ datum, nullptr };
  if (empty()) {
    first = last = new_node;
  } else {
    last = last->next = new_node;
  }
}
```

In the case of inserting into an empty list, we need to set both `first` and `last` pointing at the new node. (We would need to modify `pop_front()` to do this as well.) When inserting at the end of a non-empty list, we have to set the `next` pointer of what used to be the last node to point to the new node, and we must also update the `last` member of the `IntList` so that it points to the new node.

Now that we have `push_back()`, the logical next step is to provide `pop_back()` as well. Unfortunately, `pop_back()` requires setting the `last` pointer to point to what used to be the second-to-last node, and we can only get to that node by traversing the entire list from the beginning. Here, the problem is that our nodes only have a `next` pointer, so they allow us to traverse in the forward direction but not in reverse. We can modify our representation once more to enable backwards traversal by adding a `prev` pointer to each node:

```
class IntList {
  ...
private:
  struct Node {
    int datum;
    Node *prev;
    Node *next;
  };

  Node *first;
  Node *last;
};
```

This is now a *doubly linked list*, since each node has two links, one to the previous node and one to the next node. Our original list is a *singly linked list*.

The standard library provides both implementations: `std::forward_list` is a singly linked list, while `std::list` is a doubly linked list. The former uses less space than the latter, since the nodes don't have a pointer to the previous node, but it does not provide the ability to iterate backwards over the list.

27.4 List Template

We can generalize the `IntList` class to hold objects of other types by making it a template. Each instantiation is still homogeneous: `List<int>` only holds `int`s, `List<string>` only holds `string`s, and so on.

```
template <typename T>
class List {
public:
  List();
```

(continues on next page)

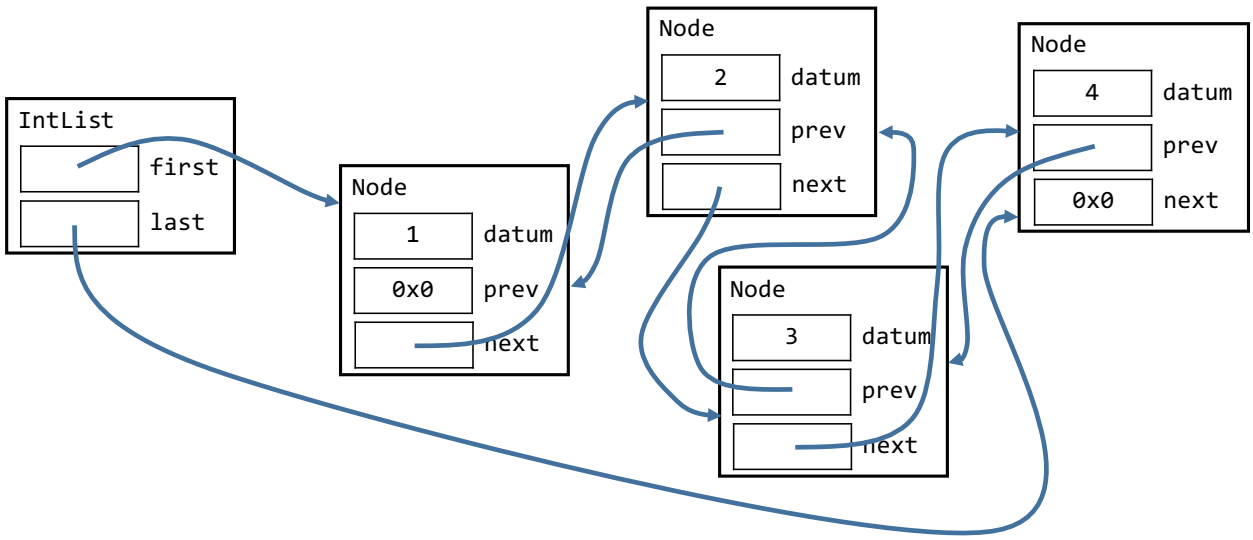

Figure 27.9: Doubly linked list with `next` and `prev` pointers in each node.

(continued from previous page)

```
  void empty() const;

  T & front();

  void push_front(const T &datum);

  void pop_front();

  void push_back(const T &datum);

  void pop_back();

  ...

private:
  struct Node {
    T datum;
    Node *prev;
    Node *next;
  };

  Node *first;
  Node *last;
};
```

By placing the `Node` struct inside the `List` template, each instantiation of `List` will have its own `Node` type; `List<int>::Node`, `List<string>::Node`, and `List<Duck>::Node` are all distinct types.

Now that our container is a template, we pass elements by reference. For `List<int>`, the elements are small enough to copy, but for class-type elements such as in `List<string>` or `List<Duck>`, we want to avoid making unnecessary copies. A good rule of thumb is to pass a function parameter by reference if the function parameter's type is a template parameter, since the latter may be instantiated with a large class type.

CHAPTER

TWENTYEIGHT

IMPLEMENTING ITERATORS

Last time, we saw how to iterate over the elements of a linked list from within the list class itself:

```
template <typename T>
void List<T>::print(std::ostream &os) const {
  for (Node *node_ptr = first; node_ptr != nullptr;
       node_ptr = node_ptr->next) {
    os << node_ptr->datum << " ";
  }
}
```

The strategy is to start with the first node in the list, then follow the nodes' `next` pointers until reaching the null sentinel. We can try to follow the same strategy from outside the class:

```
List<int> list;
...
for (Node *node_ptr = list.first; node_ptr != nullptr;
     node_ptr = node_ptr->next) {
  cout << node_ptr->datum << endl;
}
```

However, this strategy doesn't work from outside the list because the nested `Node` class is private – it is an implementation detail, so outside code should not depend on it anyway. The `first` member is also private, so the initialization of `node_ptr` is invalid. Instead, we need to provide a different interface for iterating over a list.

We have seen two patterns for iterating over a sequential container, traversal by index and traversal by pointer. In traversal by index, we use a loop variable of integral type, increment it in each iteration, and access an element by using the index as an offset into the container:

```
const int SIZE = 5;
int array[SIZE] = { 1, 2, 3, 4, 5 };
// iterate over elements in order with traversal by index
for (int i = 0; i < SIZE; ++i) {
  cout << array[i] << endl;
}
```

To use this pattern with a container, it must support *random access*, the ability to directly access an element through an index. Array-based sequential containers generally support random access. For built-in arrays, it translates directly to pointer arithmetic. Array-based class types such as `std::array` and `std::vector` overload the subscript operator to index into the underlying array, which then turns into pointer arithmetic.

Linked lists do not provide random access; since they are not array-based, accessing an element in the middle of the list cannot be done with pointer arithmetic, but must traverse from the start of the list as in `print()`. Thus, traversal by index is not an appropriate pattern for a linked list.

Traversal by pointer, on the other hand, starts with a pointer to the beginning of a sequence and iterates by moving that pointer forward to each subsequent element, until that pointer reaches past the end of the sequence:

```
const int SIZE = 5;
int array[SIZE] = { 1, 2, 3, 4, 5 };
// iterate over elements in order with traversal by pointer
for (int *ptr = array; ptr != array + SIZE; ++ptr) {
  cout << *ptr << endl;
}
```

A similar pattern would work for a linked list: start off with something like a pointer that "points" to the beginning of the list, move it forward one element at a time, and stop when we get past the end of the list:

```
List<int> list;
...
for (Iterator it = list.begin(); it != list.end(); ++it) {
  cout << *it << endl;
}
```

We use an object called an *iterator* to iterate over the elements. The pattern above is called *traversal by iterator*, and it is a generalization of traversal by pointer.

To traverse with an iterator, it must provide the following operations:[1]

- dereference (prefix *)
- increment (prefix ++)
- equality checks (== and !=)

In addition, the container itself must provide two member functions:

- `begin()` returns an iterator to the start of the sequence
- `end()` returns a "past-the-end" iterator that represents a position that is past the last element of the sequence

An iterator is a class-type object that has the same interface as a pointer. We provide the same interface by overloading the required operators:

```
template <typename T>
class Iterator {
public:
  T & operator*() const;

  Iterator &operator++();

  bool operator==(Iterator rhs) const;

  bool operator!=(Iterator rhs) const;
};
```

The unary * operator is overloaded to return the element the iterator is pointing to by reference. The prefix ++ operator moves the iterator to point to the next element in the sequence, and it returns the iterator itself by reference. This allows the operator to be chained:

```
++++it;
```

[1] There are many kinds of iterators in C++. We will only implement a subset of the operations required for an *input iterator*.

The equality operators determine if the receiver points to the same element as the `rhs` iterator. Unlike most class types, we pass iterators by value – they are generally small, and it is standard practice in C++ to make copies of them when we pass them to a function, just like we would for pointers.

Before we proceed to implement the operators, we need a data representation. The representation of an iterator is specific to a particular kind of container. For a linked list, traversal by iterator is just an abstraction over the traversal in `print()`:

- A list iterator is an abstraction over a pointer to a node.
- The call `list.begin()` returns an iterator constructed from `first`.
- The past-the-end iterator returned by `list.end()` is represented by a null pointer.
- Comparing two iterators compares their underlying node pointers.
- Incrementing an iterator moves its node pointer to the next node using the original node's `next` member.
- Derferencing the iterator obtains the `datum` member of the underlying node.

Thus, we represent an iterator with just a pointer to a node.

```
template <typename T>
class Iterator {
  ...
private:
  Node *node_ptr;
};
```

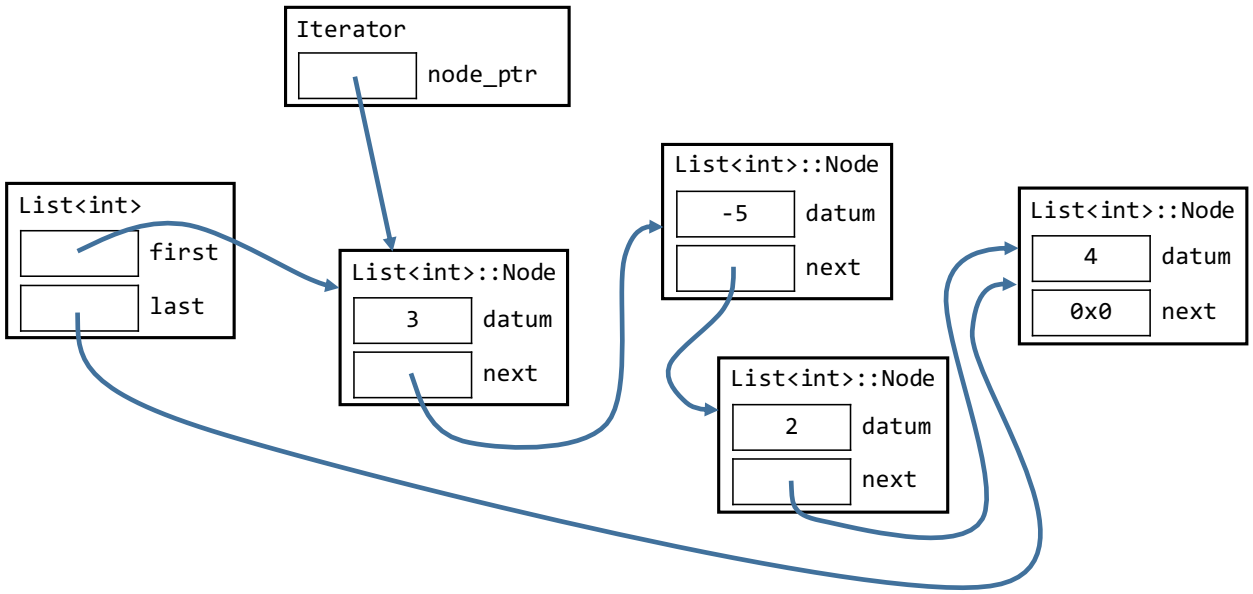

Figure 28.1: A list iterator is an abstraction of a pointer to a node.

As mentioned above, we use a null pointer for an iterator that is past the end of a list. The end condition for the traversal in `print()` is when the node pointer is null – after the traversal reaches the last node, it sets the pointer to the value of the last node's `next` member, which is null according to the list's representation invariant. Thus, it makes sense to use a null pointer to represent a past-the-end iterator.

Now that we have a representation, we should consider representation invariants. It is the case that `node_ptr` will either be null or point to a valid list node when the iterator is created. However, we will see that an iterator may be

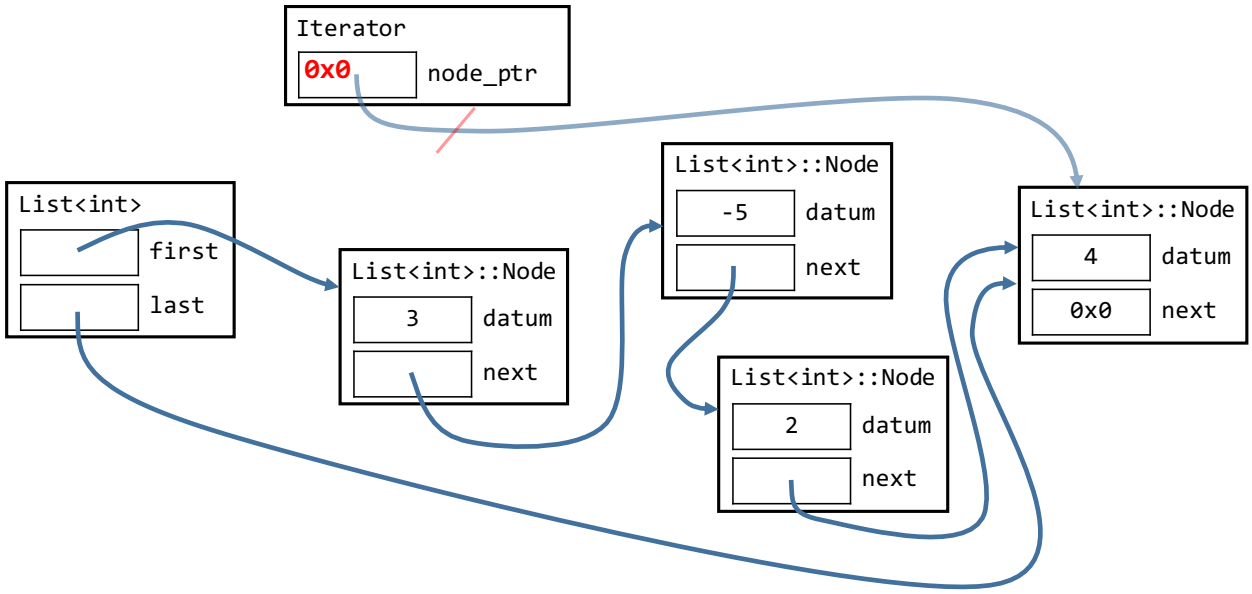

Figure 28.2: A list past-the-end iterator has a null pointer as its stored node pointer.

invalidated, which will result in `node_ptr` pointing to an invalid node. Thus, there is no invariant that will hold for a list iterator's representation.

28.1 Iterator Definition

Before we proceed to implement `Iterator`, observe the following issues with its definition:

- `Node` is not a top-level type, but a member of the `List` class template, so it cannot be named from the outside without the scope-resolution operator.
- The `Node` struct is private, so it cannot be accessed from outside code.
- The iterator type is associated with `List`, so it should be encapsulated within the `List` template.

We can solve these issues by defining `Iterator` as a member of the `List` template:

```
template <typename T>
class List {
  ...
private:
  struct Node {
    int datum;
    Node *next;
  };

public:
  class Iterator {
  public:
    T & operator*() const;

    Iterator &operator++();
```

(continues on next page)

(continued from previous page)

```
    bool operator==(Iterator rhs) const;

    bool operator!=(Iterator rhs) const;
  private:
    Node *node_ptr;
  };

private:
  Node *first;
  Node *last;
};
```

We must define the `Iterator` class after the definition of the `Node` struct so that `Node` is in scope when it is referenced in the `Iterator` class.[2] `Iterator` itself is part of the interface for `List`, so it is defined as public within `List`.

We can now implement the member functions of `Iterator`.

28.1.1 Dereference and Increment Operators

The dereference operator requires that the iterator is *dereferenceable*, which means that it is pointing to a valid element in the container. We cannot in general check that this is the case, but we can check whether the iterator is a past-the-end iterator:

```
// REQUIRES: this is a dereferenceable iterator
// EFFECTS:  Returns the element that this iterator points to.
template <typename T>
T & List<T>::Iterator::operator*() const {
  assert(node_ptr);  // check whether this is a past-the-end iterator
  return node_ptr->datum;
}
```

The `operator*()` function is a member of `Iterator`, which itself is a member of `List<T>` – thus, we need two scope-resolution operators when referring to the function. After asserting that the iterator is not past the end, the function just returns the `datum` member of the underlying node. The return is by reference, so that the element can be modified through the iterator:

```
List<int> Iterator it = ...;
*it = 3;                      // LHS of assignment must be an object
```

The dereference operator does not modify the iterator. In addition, while the function returns an object by reference to non-const, modifying that object does not modify the iterator, since the object is not a member of the iterator itself. Thus, the dereference operator can be declared as a const member function.

The `operator++()` function modifies the iterator by moving it to "point to" the next element, so it cannot be declared as const. As with dereference, the past-the-end iterator cannot be incremented – `node_ptr` would be null, and there wouldn't be a `next` pointer to move the iterator to.

```
// REQUIRES: this is a dereferenceable iterator
// EFFECTS:  Returns the element that this iterator points to.
template <typename T>
```

(continues on next page)

[2] The scope of a class member begins at the member declaration and includes the rest of the class body, all member-function bodies, and all member-initializer lists.

(continued from previous page)

```
typename List<T>::Iterator & List<T>::Iterator::operator++() {
  assert(node_ptr);  // check whether this is a past-the-end iterator
  node_ptr = node_ptr->next;
  return *this;
}
```

The function moves the iterator to the next element by following the `next` pointer of the current node. Once the iterator has been moved, the function returns the iterator by reference, in keeping with the *pattern for prefix increment*.[3]

28.1.2 The `typename` Keyword

The return type of `operator++()` is a reference to `List<T>::Iterator`; `Iterator` is a member of a template that is *dependent* on the template parameter `T`. C++ requires the `typename` keyword before a dependent name that refers to a type, so that the compiler knows that it is a type and not a value.[4]

The following illustrates when the `typename` keyword is required:

```
template <typename U>
void func() {
  IntList list;                     // not a qualified type
  IntList::Iterator it1;    // outer class does not depend on template parameter U
  List<int>::Iterator it2;  // outer class does not depend on template parameter U
  typename List<U>::Iterator it3;   // outer class depends on template parameter U
  int capacity =
    SortedSet<U>::MAX_CAPACITY;     // member is not a type
}
```

An *unqualified* name, one without the scope-resolution operator, never needs the `typename` keyword. In a *qualified* name, if the outer type does not depend on a template parameter, then no `typename` keyword is required. If the outer type does depend on a template parameter, then the `typename` keyword is required when the inner name refers to a type. If the inner name does not refer to a type, the `typename` keyword is erroneous to use, since it explicitly tells the compiler that the inner name is a type.

In practice, compilers can often determine when the `typename` keyword is required, and many C++ programmers rely on the compiler to tell them that it is needed rather than learning the rules:

```
$ g++ --std=c++17 foo.cpp
foo.cpp:5:20: error: expected ';' after expression
  List<U>::Iterator it;
                   ^
                   ;
foo.cpp:5:21: error: use of undeclared identifier 'it'
```

(continues on next page)

[3] The postfix increment operator can be overloaded as well. To distinguish its signature from prefix, C++ uses a dummy `int` parameter for postfix:

```
template <typename T>
typename List<T>::Iterator List<T>::Iterator::operator++(int) {
  assert(node_ptr);  // check whether this is a past-the-end iterator
  Iterator tmp = *this;              // make a copy of this iterator
  node_ptr = node_ptr->next;
  return tmp;                                   // return the copy
}
```

We need not provide a parameter name, since we are not actually using the parameter object. In keeping with the contract for postfix increment, the function returns a copy of the original iterator by value.

[4] The reasons are beyond the scope of this course, but they have to do with the way the compiler instantiates templates.

(continued from previous page)

```
  List<U>::Iterator it;
                   ^
foo.cpp:5:3: error: missing 'typename' prior to dependent type name
      'List<int>::Iterator'
  List<U>::Iterator it;
  ^~~~~~~~~~~~~~~~
foo.cpp:9:3: note: in instantiation of function template specialization
      'func<int>' requested here
  func<int>();
  ^
3 errors generated.
```

28.1.3 Equality Comparisons

Two iterators are defined as equal if either they are both past the end, or they "point to" the same element in the same list. Thus, they are equal exactly when their `node_ptr` members point to the same node.

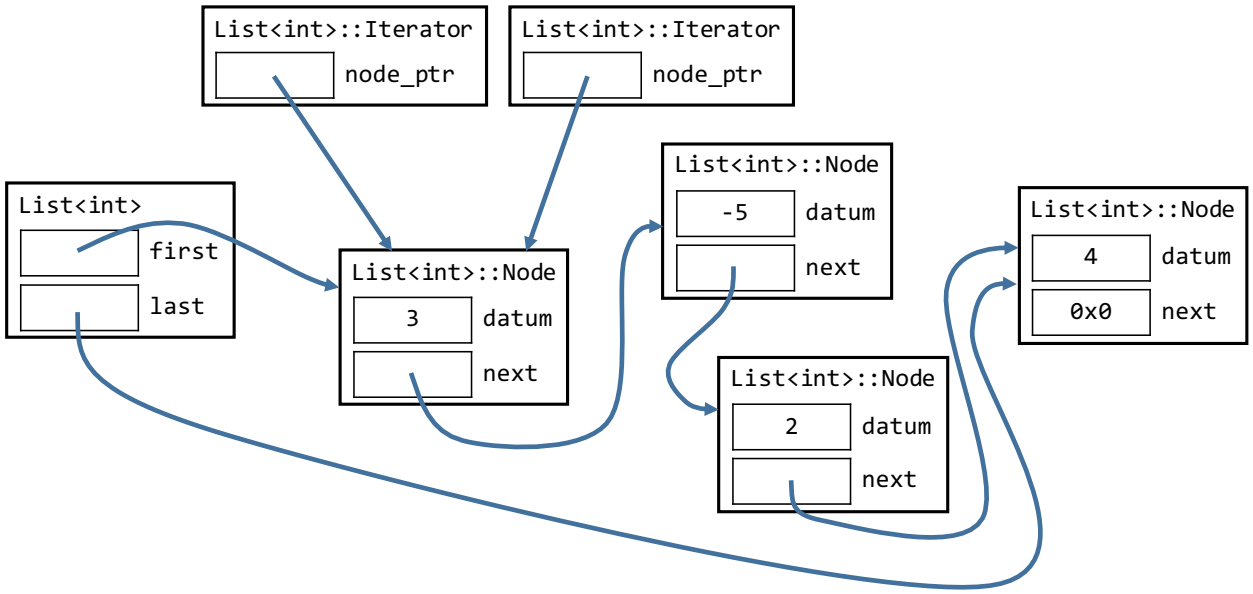

Figure 28.3: Two iterators are equal when their node pointers store the same address, pointing to the same node.

Comparing two iterators does not require them to be dereferenceable or even pointing to a valid element. Thus, the operators do not have a REQUIRES clause.

```
template <typename T>
bool List<T>::Iterator::operator==(Iterator rhs) const {
  return node_ptr == rhs.node_ptr;
}

template <typename T>
bool List<T>::Iterator::operator!=(Iterator rhs) const {
  return node_ptr != rhs.node_ptr;
}
```

We do not need to qualify `Iterator` in the parameter type – once the compiler knows that we are defining a member of `Iterator`, we can refer to the class with an unqualified name.

28.1.4 Creating Iterators

We have defined the operator overloads for `Iterator`. However, we have not provided a means of creating an `Iterator` object. Without a user-defined constructor, we do get an implicit default constructor, but it just initializes `node_ptr` with a junk value. Instead, we define the following two constructors:

- an explicit default constructor that makes the iterator a past-the-end iterator
- a constructor that takes in a pointer to a node

The latter is a private constructor – the `Node` class is not part of the interface for the list, so the constructor that takes a node pointer also is not part of the interface, and the outside world would not be able to call it even if it were.

The constructor definitions are as follows:

```
template <typename T>
class List {
  ...
public:
  class Iterator {
  public:
    // EFFECTS: Constructs a past-the-end iterator.
    Iterator() : node_ptr(nullptr) {}

    ...
  private:
    // EFFECTS: Constructs an iterator from the given node pointer.
    Iterator(Node *node_ptr_in) : node_ptr(node_ptr_in) {}

    Node *node_ptr;
  };
};
```

We can now implement `begin()` and `end()` member functions in `List` that return a start and past-the-end iterator, respectively.

```
template <typename T>
class List {
  ...
public:
  // EFFECTS: Returns an iterator that points to the first element,
  //          or a past-the-end iterator if this list is empty.
  Iterator begin() {
    return Iterator(first);
  }

  // EFFECTS: Returns a past-the-end iterator.
  Iterator end() {
    return Iterator();
  }
};
```

The `begin()` function returns an iterator that points to the first element in the list. However, if the list is empty, `first` is null; this is the representation of a past-the-end iterator, so `begin()` returns such an iterator when the list is empty. The `end()` function returns a default-constructed past-the-end iterator.

Unfortunately, the implementation of `begin()` does not compile:

```
./List.hpp:122:12: error: calling a private constructor of class
      'List<int>::Iterator'
    return Iterator(first);
           ^
```

A private member is only accessible from within the scope of the class that defines the member. The private `Iterator` constructor is a member of `Iterator`, but it is being accessed from outside the `Iterator` class. On the other hand, `Iterator` is within the scope of `List`, so it can access private `List` members such as `Node`. Thus, a nested class can access private members of the outer class, but not vice versa.

28.2 Friend Declarations

The solution to the problem above is a *friend declaration*, in which a class gives an outside entity access to the class's private members.

```
template <typename T>
class List {
  ...
public:
  class Iterator {
    friend class List;
    ...
  };
};
```

A friend declaration can appear anywhere directly within a class body, and it tells the compiler that the given entity is allowed to access private members. The entity may be a function or a type; for instance, it is common practice to define the insertion operator as a friend:

```
class Card {
  ...
  private:
    std::string rank;
    std::string suit;

    friend std::ostream & operator<<(std::ostream &os, const Card &card);
};

std::ostream & operator<<(std::ostream &os, const Card &card) {
  return os << card.rank << " of " << card.suit;
}
```

Friendship is given, not taken. For instance, class `C` may declare `F` as a friend:

```
friend class F;
```

This allows `F` to access the private members of `C`, but it does not allow `C` to access the private members of `F`.

The friend declaration completes our iterator implementation, so we can now perform a traversal by iterator:

```
List<int> list;
for (int i = 1; i < 5; ++i) {
  list.push_back(i);
}

for (List<int>::Iterator it = list.begin(); it != list.end(); ++it) {
  cout << *it << endl;
}
```

This is just an abstraction over a loop that uses a node pointer directly:

```
for (Node *node_ptr = list.first; node_ptr != nullptr;
     node_ptr = node_ptr->next) {
  cout << node_ptr->datum << endl;
}
```

The iterator starts at the `first` member, just like the traversal using a node pointer. Incrementing the iterator moves it to the next node, the same as in the loop above. Dereferencing the iterator accesses the respective node's `datum` member, as in the body of the loop above. The traversal ends when the iterator reaches a null pointer, just like the node-pointer loop. Thus, the iterator provides the same traversal as can be done from within the `List` template, but with the interface of a pointer rather than having the user rely on implementation details.

28.3 Generic Iterator Functions

Iterators provide a common interface for traversing through a sequence of elements, and the standard-library sequences all support the iterator interface.

```
vector<int> vec = { 1, 2, 3, 4 };
for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
  cout << *it << endl;
}
```

The common interface allows us to write generic functions that work on any sequence. For instance, we can write a function template for finding the maximum element in a sequence:

```
// REQUIRES: end is after or equal to begin
// EFFECTS:  Returns an iterator to the maximum element in [begin, end).
//           Returns begin when the sequence is empty.
template <typename Iter_type>
Iter_type max_element(Iter_type begin, Iter_type end) {
  Iter_type max_so_far = begin;
  for (; begin != end; ++begin) {
    if (*max_so_far < *begin) {
      max_so_far = begin;
    }
  }
  return max_so_far;
}
```

The `max_element()` function template takes a begin and end iterator of any type and returns an iterator to the maximum element that is between the two iterators. As is standard in C++, the end iterator is an exclusive bound – the

function stops when it reaches `end` and does not dereference it. Then `max_element()` implements a standard find-the-max algorithm: start with the first element as the maximum so far, then update the max so far when a larger item is encountered. The iterators must be dereferenced to get to the elements to compare them, and the element type must support the < operator.

We can use `max_element()` with any iterator and element type, as long as the element type supports <:

```
vector<int> vec;
.. // fill vec with numbers
cout << *max_element(vec.begin(), vec.end()) << endl;

List<int> list;
.. // fill list with numbers
cout << *max_element(list.begin(), list.end()) << endl;

List<Card> cards;
.. // fill cards with Cards
cout << *max_element(cards.begin(), cards.end()) << endl;

int const SIZE = 10;
double arr[SIZE];
.. // arr fill with numbers
cout << *max_element(arr, arr + SIZE) << endl;
```

As usual with function templates, the compiler deduces the template parameter from the function arguments. The last example illustrates that we can even use `max_element()` with an array; since a pointer has the same interface as an iterator, we just need to provide `max_element()` with a pointer to the first element and another that is just past the end.

The standard `<algorithm>` library contains many function templates such as `std::max_element()` that operate on iterators. However, the standard-library function templates require an iterator to define several member type aliases to work:

```
template <typename T>
class List {
  ...
public:
  class Iterator {
    ...
  public:
    using iterator_category = std::input_iterator_tag;
    using value_type = T;
    using difference_type = void;
    using pointer = void;
    using reference = T &;
  };
};
```

A discussion of these type aliases is beyond the scope of this course.

As another example, the following is a function template that determines whether a sequence contains a duplicate item. It compares every pair of elements with the == operator to determine if any are equal:

```
template <typename Iter_type>
bool no_duplicates(Iter_type begin, Iter_type end) {
  for (; begin != end; ++begin) {
```

(continues on next page)

(continued from previous page)

```
    Iter_type other = begin;        // copy iterator to current element
    ++other;                        // move copy one element forward
    for (; other != end; ++other) {
      if (*begin == *other) {       // compare element with those that come after
        return false;
      }
    }
  }
  return true;
}
```

28.4 Iterator Invalidation

Recall that a dangling pointer points to an invalid object, one that is no longer alive. Since iterators represent indirection to sequence elements, an iterator can also end up pointing to an invalid object. Such an iterator is said to be *invalidated*. The following is an example:

```
List<int> list;
list.push_back(3);
list.push_back(-5);
list.push_back(2);
list.push_back(4);

List<int>::Iterator iter = list.begin();
cout << *iter << endl;

list.pop_front();       // invalidates iter

cout << *iter << endl;  // UNDEFINED BEHAVIOR
```

The code constructs a list and inserts elements into it. It then creates an iterator that is pointing to the first element before proceeding to remove that element. This invalidates the iterator, so that dereferencing the iterator results in undefined behavior.

In general, modifying a sequence can invalidate existing iterators. An iterator may be invalidated even if the element it is pointing to is not removed. For instance, a vector iterator is invalidated when a grow operation occurs, since the element the iterator is pointing to moves to a different location. A member function's documentation should indicate whether it may invalidate iterators, as in the cppreference.com documentation for `std::vector::push_back()`:

> If the new `size()` is greater than `capacity()` then all iterators and references (including the past-the-end iterator) are invalidated. Otherwise only the past-the-end iterator is invalidated.

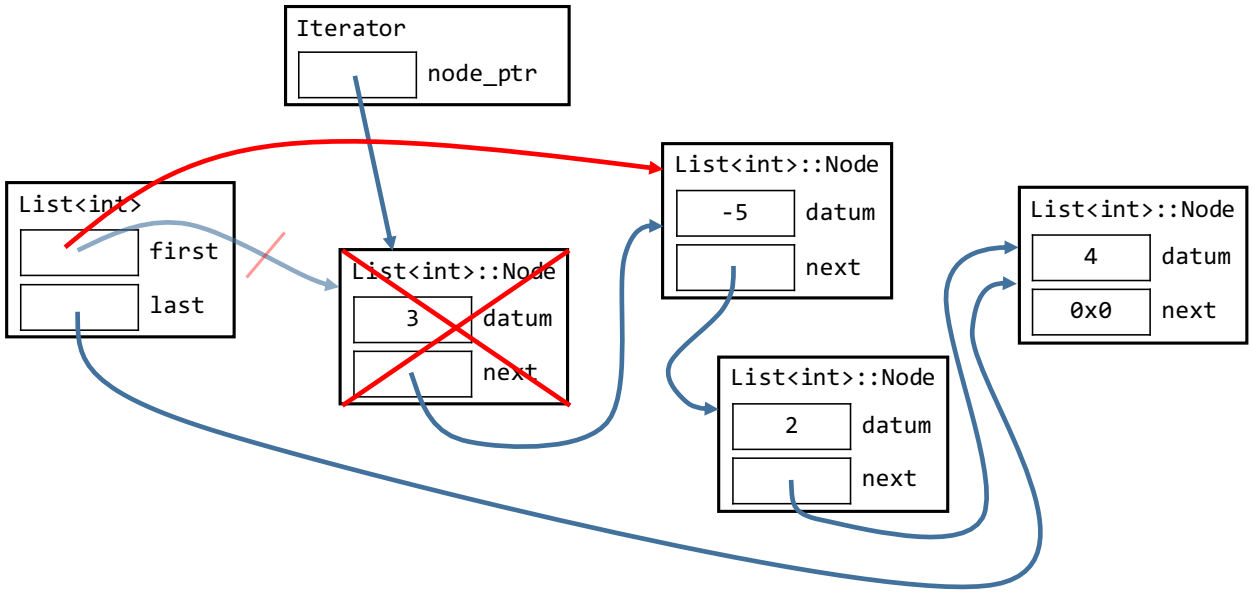

Figure 28.4: Removing the element that an iterator is pointing to invalidates the iterator.

28.5 Type Deduction

Suppose we wanted to write a function template that prints out the elements of a sequence. We could write it to work with iterators:

```
template <typename Iter_type>
void print_all(Iter_type begin, Iter_type end) {
  for (Iter_type it = begin; it != end; ++it) {
    cout << *it << endl;
  }
}
```

On the other hand, we desire an interface that works directly on a sequence container:

```
template <typename Sequence>
void print_all(const Sequence &sequence) {
  for (typename Sequence::Iterator it = sequence.begin();
       it != sequence.end(); ++it) {
    cout << *it << endl;
  }
}
```

Declaring the iterator type is very verbose, as it consists of both a qualified name and the `typename` keyword, since the qualified name is a dependent type. Furthermore, the declaration makes the assumption that the iterator type is named `Iterator`, which is not true for standard-library containers that use lower-case type names.

Rather than writing out the type of `it` explicitly, we can ask the compiler to deduce it for us by using the `auto` keyword:

```
template <typename Sequence>
void print_all(const Sequence &sequence) {
  for (auto it = sequence.begin(); it != sequence.end(); ++it) {
    cout << *it << endl;
```

(continues on next page)

(continued from previous page)

```
  }
}
```

The compiler deduces the type from its initialization. If the return type of `sequence.begin()` is `List<int>::Iterator`, then the type of `it` is deduced to have type `List<int>::Iterator`. On the other hand, if the return type is `vector<Duck>::iterator`, then the type of `it` is deduced to be `vector<Duck>::iterator`. The return type need not be a nested type; if it is `char *`, then `it` will be deduced to have type `char *`. Thus, not only does type deduction save us keystrokes for complicated types, it also abstracts over how the types are defined.

Part IV

Functional Programming

CHAPTER

TWENTYNINE

FUNCTION OBJECTS

Last time, we saw iterators, which are a common interface for traversing over the elements of a sequence. For instance, the following function template works over any sequence of integers, determining if there is and odd element in the sequence:

```
// REQUIRES: begin is before or equal to end
// EFFECTS:  Returns true if any element in the sequence
//           [begin, end) is odd.
template <typename Iter_type>
bool any_of_odd(Iter_type begin, Iter_type end) {
  for (Iter_type it = begin; it != end; ++it) {
    if (*it % 2 != 0) {
      return true;
    }
  }
  return false;
}
```

We can then use `any_of_odd()` with any sequence type, as long as the elements are integers:

```
List<int> list;
vector<int> vec;
int array[10];
...
cout << any_of_odd(list.begin(), list.end()) << endl;
cout << any_of_odd(vec.begin(), vec.end()) << endl;
cout << any_of_odd(array, array + 10)) << endl;
```

The template is generic over the iterator type, but it is not generic over the condition that an element must meet – it only searches for odd elements and no other characteristic. Suppose we wanted instead to determine whether the sequence contains an even element. We could write an `any_of_even()` template:

```
// REQUIRES: begin is before or equal to end
// EFFECTS:  Returns true if any element in the sequence
//           [begin, end) is even.
template <typename Iter_type>
bool any_of_even(Iter_type begin, Iter_type end) {
  for (Iter_type it = begin; it != end; ++it) {
    if (*it % 2 == 0) {
      return true;
    }
  }
```

(continues on next page)

(continued from previous page)

```
  return false;
}
```

This code is almost exactly the same as that for `any_of_odd()`; the only difference is the computation done in the test of the conditional. Most of the code is duplicated, which is undesirable. Furthermore, if we wanted to determine whether a sequence contained an element that met some other condition, say whether an element is positive, we would have to write additional function templates that duplicate code.

Our general strategy for avoiding code duplication in a procedural abstraction is to introduce parameters for the pieces that differ. For a value that differs, we add a function parameter:

```
int power3(int x) {
  int result = 1;
  for (int i = 0; i < 3; ++i) {
    result *= x;
  }
  return result;
}

int power4(int x) {
 int result = 1;
  for (int i = 0; i < 4; ++i) {
    result *= x;
  }
  return result;
}

int power(int x, int n) {          // add parameter to generalize
  int result = 1;
  for (int i = 0; i < n; ++i) {
    result *= x;
  }
  return result;
}

int main() {
  for (int i = 0; i < 10; ++i) {
    cout << power(42, i);          // supply desired argument
  }
}
```

When it is a type that differs, we make the function a template and introduce a template parameter:

```
int max_int(int x, int y) {
  return x < y ? y : x;
}

Card max_card(const Card &x, const Card &y) {
  return x < y ? y : x;
}

template <typename T>              // add template parameter to generalize
T max(const T &x, const T &y) {  // pass objects of template-parameter type
```

(continues on next page)

(continued from previous page)

```
                                    //   by reference
  return x < y ? y : x;
}

int main() {
  cout << max(3, -1) << endl;    // template parameter deduced from arguments
  cout << max(Card(Card::RANK_TWO, Card::SUIT_SPADES),
              Card(Card::RANK_TOW, Card::SUIT_HEARTS)) << endl;
}
```

For a generic `any_of()`, however, it is an actual computation that differs: `*it % 2 != 0` for odd numbers, `*it % 2 == 0` for even numbers, `*it > 0` for positive numbers, and so on. We can use a function to represent such a computation:

```
bool is_odd(int x) {
  return x % 2 != 0;
}

bool is_even(int x) {
  return x % 2 == 0;
}

bool is_positive(int x) {
  return x > 0;
}
```

Once we have a function, the compiler translates it into machine code, which is placed in memory in the text segment when the program runs.

Since the code is in memory, we can construct a pointer to it:

```
bool (*func)(int) = &is_odd;
func = &is_even;
```

As the examples above demonstrate, we can apply the address-of operator to a function to obtain its address, just like for an object. Unlike an object, however, the address-of operator is optional:

```
func = is_positive;
```

The compiler implicitly inserts the operator for us.

We can call a function through a pointer by first dereferencing the pointer:

```
bool (*func)(int) = is_odd;
(*func)(-2);                    // returns false
```

Like with the address-of operator, the compiler can implicitly insert this dereference for us:

```
func(-2);                       // returns false
```

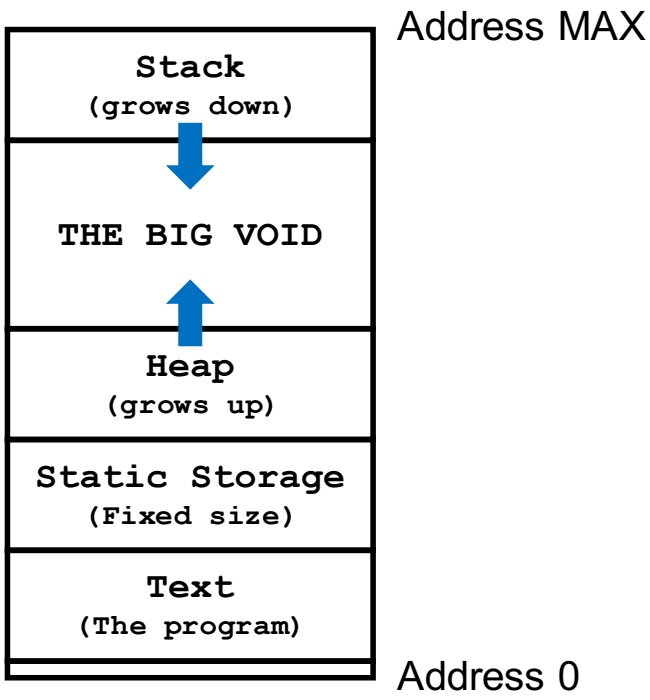

Figure 29.1: The memory for a program includes a segment that stores the program's code.

29.1 Function-Pointer Types

Before we proceed to implement a generic `any_of()`, let us examine the syntax of a function pointer more closely.

```
bool (*func)(int);
```

C++ declarations are generally read from right to left. However, the parentheses around `*func` associate the `*` symbol with the name `func`. Without the parentheses, we would have the following:

```
bool *func2(int);
```

This is a declaration of a function called `func2` that takes in an `int` and returns a pointer to a `bool` – the `*` is associated with the return type rather than the name `func2`.

With the parentheses, the `*` has the same meaning as for other variables: it indicates that the variable we are declaring is a pointer. Thus, `func` is a pointer. The rest of the declaration tells us what kind of pointer it is: a pointer to a function that takes an `int` as a parameter and returns a `bool`.

To declare an appropriate function pointer, we can use the following steps:

- Start with a function signature:

  ```
  int max_int(int x, int y);
  ```

- Remove the parameter names, which serve only as documentation in a declaration:

  ```
  int max_int(int, int);
  ```

- Replace the function name with a variable name and the * symbol to indicate it is a pointer, surrounded by parentheses:

```
int (*func3)(int, int);
```

The result is that `func3` is a pointer to a function that takes two `int`s and returns an `int`.

29.2 Function-Pointer Parameters

We can now write a generic `any_of()` that is parameterized both by an iterator type as well as a function to test an element:

```
template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end, bool (*func)(int)) {
  for (Iter_type it = begin; it != end; ++it) {
    if (func(*it)) {
      return true;
    }
  }
  return false;
}
```

Since different iterators may have different types, we use a template parameter to allow an arbitrary kind of iterator. For the test of whether an element satisfies a condition, we add a function parameter that is a pointer to a function. We call it using parentheses like any other function, and the compiler automatically dereferences the pointer to get to its code.

We can then specify which function to use when calling `any_of()`:

```
List<int> list;
vector<int> vec;
int array[10];
...
cout << any_of(list.begin(), list.end(), is_odd) << endl;
cout << any_of(vec.begin(), vec.end(), is_even) << endl;
cout << any_of(array, array + 10, is_positive)) << endl;
```

The compiler implicitly takes the address of a function when we pass it to a function pointer.

Functions that take in an item and return a truth value are quite common, and they are called *predicates*. Thus, a better name for the function-pointer parameter in `any_of()` would be `pred` rather than `func`.

```
// REQUIRES: begin is before or equal to end
// EFFECTS:  Returns true if pred returns true when applied to at
//           least one element in the sequence [begin, end).
template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end, bool (*pred)(int)) {
  for (Iter_type it = begin; it != end; ++it) {
    if (pred(*it)) {
      return true;
    }
  }
  return false;
}
```

29.3 Functors

With `any_of()` and `is_positive()`, we can determine whether a sequence contains an element that is greater than zero. What if we are interested in other thresholds, such as 32 or 212? We don't want to write separate functions for each value, as this duplicates code:

```
bool greater32(int x) {
  return x > 32;
}

bool greater212(int x) {
  return x > 212;
}
```

Since what differs is just a value, we can write a function that has a parameter for the threshold value:

```
bool greater(int x, int threshold) {
  return x > threshold;
}
```

Unfortunately, we cannot use this with `any_of()`: it requires a pointer to a function that takes in one argument, not two:

```
main.cpp:29:11: error: no matching function for call to 'any_of'
  cout << any_of(arr, arr + SIZE, greater) << endl;
          ^~~~~~
main.cpp:13:6: note: candidate function not viable: no known conversion from
      'bool (int, int)' to 'bool (*)(int)' for 3rd argument
bool any_of(Iter_type begin, Iter_type end, bool (*pred)(int)) {
     ^
1 error generated.
```

Furthermore, we need some way of specifying the threshold, and passing the `greater()` function directly to `any_of()` does not do so. What we need is something that internally stores the threshold value and is callable with just one argument.

More specifically, we want a *first-class entity*, which is an entity that supports the following:

- It can store state.
- It can be created at runtime.
- It can be passed as an argument or returned from a function.

Unfortunately, functions are not first-class entities in C++: they cannot be created at runtime, and they are very limited in how they can store information.[1]

Class types, on the other hand, do define first-class entities. A class-type object can store state in member variables, can be created at runtime, and can be passed between functions. So a class type could satisfy our needs, as long as there were a way to call it like a function. In fact, C++ does allow a class-type object to be called if the class overloads the function-call operator. We refer to such a class as a *functor*.

A functor is a class type that provides the same interface as a function, much like an iterator is a class type that provides the same interface as a pointer. The following is a `GreaterN` class that stores a threshold and is also callable with a single argument:

[1] A function may have a static local variable, but it can only store a single value at a time. We need an arbitrary amount of storage – for instance, we may have any number of threshold values we care about for `greater()`, so we need a way of creating multiple copies of the function that each have their own stored threshold value.

```
class GreaterN {
public:
  // EFFECTS: Creates a GreaterN with the given threshold.
  GreaterN(int threshold_in) : threshold(threshold_in) {}

  // EFFECTS: Returns whether or not the given value is greater than
  //          this GreaterN's threshold.
  bool operator()(int x) const {
    return x > threshold;
  }

private:
  int threshold;
};
```

The function-call operator must be overloaded as a member function, so it has an implicit `this` pointer that allows us to access a member variable. We have declared the `this` pointer as a pointer to const, since the function does not modify the `GreaterN` object. We also get to decide what the parameters are, as well as the return type. We have chosen a single parameter of type `int` and a `bool` return type, since we want a `GreaterN` object to act as a predicate on `int`s.

We can create and use `GreaterN` objects as follows:

```
int main() {
  GreaterN greater0(0);
  GreaterN greater32(32);
  GreaterN greater212(212);

  cout << greater0(-5) << endl;     // 0 (false)
  cout << greater0(3) << endl;      // 1 (true)

  cout << greater32(9) << endl;     // 0 (false)
  cout << greater32(45) << endl;    // 1 (true)

  cout << greater212(42) << endl;   // 0 (false)
  cout << greater212(451) << endl;  // 1 (true)
}
```

We have declared `GreaterN` objects as local variables and initialized them by calling the constructor, the same as other class-type objects. We can then use a `GreaterN` object as the first operand of the function-call operator. We pass whatever arguments are needed to invoke the overloaded member function, and as with any non-static member function, the `this` parameter is implicitly passed.

A `GreaterN` object provides the same interface as the functions required by `any_of()`. However, it is not a function or a pointer to a function; its type is `GreaterN`, and it cannot be passed to the version of `any_of()` we wrote previously. Instead, we need to write a new version that allows predicates of any type. Since it is a type that differs, we add a template parameter to refer to that type:

```
// REQUIRES: begin is before or equal to end
// EFFECTS:  Returns true if pred returns true when applied to at
//           least one element in the sequence [begin, end).
template <typename Iter_type, typename Predicate>
bool any_of(Iter_type begin, Iter_type end, Predicate pred) {
  for (Iter_type it = begin; it != end; ++it) {
    if (pred(*it)) {
```

(continues on next page)

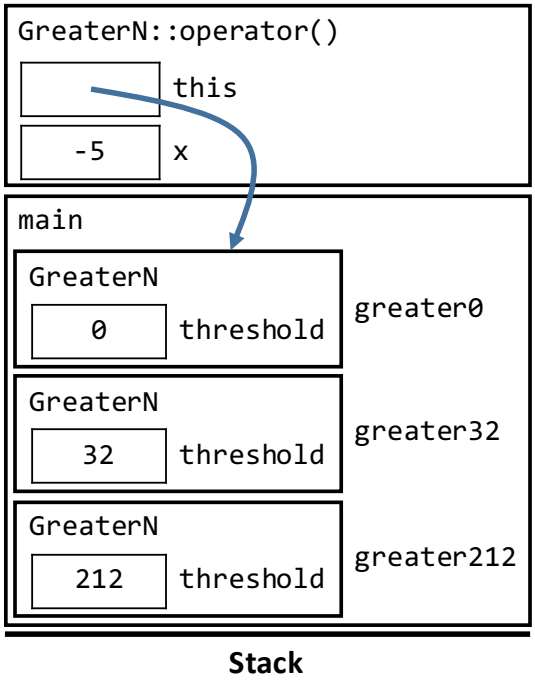

Figure 29.2: Invoking the overloaded function-call operator of a `GreaterN` object.

(continued from previous page)

```
      return true;
    }
  }
  return false;
}
```

Like iterators, functors are generally passed by value. We can now use `any_of()` with a `GreaterN` object:

```
List<int> list;
... // fill list with numbers
GreaterN greater0(0);
cout << any_of(list.begin(), list.end(), greater0);      // pass existing functor
cout << any_of(list.begin(), list.end(), GreaterN(32));  // pass temporary functor
```

The compiler deduces the template parameter `Iter_type` as `List<int>::Iterator`, and the parameter `Predicate` as `GreaterN`. We can still call `any_of()` with a function pointer:

```
cout << any_of(list.begin(), list.end(), is_odd);        // pass function
```

In this case, the compiler deduces `Predicate` as the function-pointer type `bool (*)(int)`.

By parameterizing `any_of()` with the predicate type, we can now call it on sequences of objects that are of types other than `int`. As long as a sequence element can be passed to the predicate, the code will work.

```
bool is_empty(const string &s) {
  return s.empty();
}
```

(continues on next page)

(continued from previous page)

```
int main() {
  vector<string> vec;
  ... // fill vec with strings
  cout << any_of(vec.begin(), vec.end(), is_empty);
}
```

Exercise 1 Write a function `list_count()` that counts the number of elements in a list for which the given predicate returns true. Assume that `Node` is defined as

```
struct Node {
  int datum;
  Node *next;
};
```

and use recursion to traverse the sequence of nodes.

```
// EFFECTS: Returns the number of elements in the list that
//          starts at the given node for which the predicate
//          returns true.
template <typename Predicate>
int last(const Node *node, Predicate pred) {
  // your code here
}
```

29.4 Comparators

Recall the `max_element()` function template from last time:

```
// REQUIRES: end is after or equal to begin
// EFFECTS:  Returns an iterator to the maximum element in [begin, end).
//           Returns begin when the sequence is empty.
template <typename Iter_type>
Iter_type max_element(Iter_type begin, Iter_type end) {
  Iter_type max_so_far = begin;
  for (; begin != end; ++begin) {
    if (*max_so_far < *begin) {
      max_so_far = begin;
    }
  }
  return max_so_far;
}
```

The function template uses the < operator to compare elements, so it only works on types that provide that operator. It will not work with a type such as `Duck` that does not overload the less-than operator. However, we may still want to compare `Duck`s according to some criteria, such as by their names or their ages. Thus, we need a way of specifying how `max_element()` should compare objects. We can do so by adding another parameter to represent a *comparator*, which takes two objects and returns a value that indicates how they compare to each other.

```
// REQUIRES: end is after or equal to begin
// EFFECTS:  Returns an iterator to the maximum element in [begin, end)
```

(continues on next page)

(continued from previous page)

```
//           according to the given comparator. Returns begin when
//           the sequence is empty.
template <typename Iter_type, typename Comparator>
Iter_type max_element(Iter_type begin, Iter_type end,
                      Comparator less) {
  Iter_type max_so_far = begin;
  for (; begin != end; ++begin) {
    if (less(*max_so_far, *begin)) {
      max_so_far = begin;
    }
  }
  return max_so_far;
}
```

Standard comparators in C++ do a less-than comparison, so we have modified `max_element()` to take in such a comparator. The type of the comparator is a template parameter so that either a functor or a function pointer can be used. The code then calls the comparator with two arguments to determine if the first is less than the second.

We can write a `Duck` comparator that compares two ducks by name:

```
class DuckNameLess {
public:
  bool operator()(const Duck &d1, const Duck &d2) const {
    return d1.get_name() < d2.get_name();
  }
};
```

Here, we have written the comparator as a functor; since it doesn't require any storage, it could have been written as a function as well.[2] We can then call `max_element()` as follows:

```
vector<Duck> vec;
... // fill vec with Ducks
cout << (*max_element(vec.begin(), vec.end(), DuckNameLess())).get_name();
```

We pass a default-constructed `DuckNameLess` object as the comparator and get back an iterator to the `Duck` with the maximum name. We then dereference the iterator to get to the `Duck` object and call `get_name()` on it.[3]

Given our modifications to `max_element()`, we can no longer call it without a comparator argument, even for types that support the < operator. However, the standard <algorithm> library provides a functor template `std::less` that just invokes the < operator, so we can use it with types that do have the operator:

```
vector<int> vec;
... // fill vec with numbers
cout << *max_element(vec.begin(), vec.end(), std::less<int>());
```

It is also possible to write `max_element()` to default to using `std::less` when a comparator is not explicitly provided, but that is beyond the scope of this course.

[2] While either a functor or function would work, functors are actually more efficient since calling them does not require any indirection. The compiler can statically determine from the functor type which overloaded operator is called, while calling through a function pointer generally requires a dereference at runtime. Thus, functors are usually preferred over functions given a choice between the two.

[3] Standard-library iterators overload the -> operator, so we could use that operator to call `get_name()` directly rather than manually applying dereference followed by the dot operator.

29.5 Iterating over a Sequence

Another common pattern is to iterate over a sequence, performing some operation on each element individually. We can write a `for_each()` function template that implements this pattern, taking in a function pointer or functor that applies to a single element:

```
// REQUIRES: end is after or equal to end
// EFFECTS: Applies func to each of the elements in the sequence
//          [begin, end) and returns func.
template <typename Iter_t, typename Func_t>
Func_t for_each(Iter_t begin, Iter_t end, Func_t func) {
  for (Iter_t it = begin; it != end; ++it) {
    func(*it);
  }
  return func;
}
```

We return the `func` argument, in case it contains data that are useful to the caller. For instance, the following functor stores the sum of integer elements:

```
class Accumulator {
public:
  void operator()(int x) {
    sum += x;
  }

  int get_sum() const {
    return sum;
  }

private:
  int sum = 0;
};
```

We can then compute the sum of the elements in a sequence:

```
vector<int> vec;
... // fill vec with numbers
Accumulator summer;
summer = for_each(vec.begin(), vec.end(), summer);
cout << summer.get_sum() << endl;
```

To print out the elements in a sequence to a stream, we can write a functor template for printing:

```
template <typename T>
class Printer {
public:
  Printer(std::ostream &os) : output(os) {}

  void operator()(const T &item) const {
    output << item << std::endl;
  }
```

(continues on next page)

(continued from previous page)

```
private:
  std::ostream &output;
};
```

The member variable must be a reference, since streams do not generally support copying. We can then use `Printer` to print out each element in a sequence:

```
int main() {
  List<int> list;
  ... // fill list with numbers
  ofstream fout("list.out");
  for_each(list.begin(), list.end(), Printer<int>(fout));
}
```

CHAPTER

THIRTY

IMPOSTOR SYNDROME

In addition to building our programming skills, it is also important to develop an appropriate awareness of our own abilities. Unfortunately, many of us encounter *impostor syndrome*, where we do not internalize our own accomplishments. Despite demonstrating success in a course like this, we often fear being exposed as a "fraud."

Recognizing impostor syndrome is the first step to overcoming it. The following are common characteristics of people who have impostor syndrome:

- They attribute their success to external factors.
- They fear being revealed as a fraud.
- They convince themselves that they are not good enough.
- They have a hard time accepting compliments for their accomplishments.

Impostor syndrome differs from implicit bias in that the latter affects how we subconsciously view others, while the former affects how we view ourselves. The National Center for State Courts defines implicit bias as:

> The bias in judgment and/or behavior that results from subtle cognitive processes (e.g., implicit attitudes and stereotypes) that often operate at a level below conscious awareness and without intentional control.

Of course, these implicit attitudes also affect how we view ourselves, so even though impostor syndrome affects everyone, it tends to have a higher effect on people in underrepresented groups.

The result of impostor syndrome is to doubt ourselves, having feelings such as:

- "I was hired to fill a diversity quota."
- "Everyone else seems smarter than me."
- "If I'm not getting an A, how am I going to survive in future courses?"
- "Nobody else here is like me – I don't belong in this class".
- "I don't like asking questions in class because I'm afraid others will realize I don't know what I'm doing."
- "Maybe EECS isn't for me; I should drop."

The truth is that almost everyone suffers from feelings like this. Surveys on impostor syndrome have found that "about 70 percent of people from all walks of life – men and women – have felt like impostors for at least some part of their careers."[4] The likelihood is that many of us are experiencing this right now, in this course.

There are steps we can take to overcome impostor syndrome:

- Stop comparing ourselves to others.
- Encourage each other.
- Join student organizations and connect with other students.

[4] Gravois, J. (2007). You're not fooling anyone. The Chronicle of Higher Education, 54(11), A1.

- Accept our accomplishments.
- Find a mentor.

We do not have to experience this alone; instead, we should support each other and find others we can lean on.

The following are additional resources on impostor syndrome:

- University of Michigan CAPS page on impostor syndrome
- TED talk on how students of color confront impostor syndrome
- American Psychological Association article on impostor syndrome

CHAPTER

THIRTYONE

RECURSION

We have seen several forms of abstraction, where we use something for what it does rather than how it works. It can be useful to refer to an abstraction when we are still in the middle of implementing it. For instance, our definition of a linked-list node refers to itself:

```
struct Node {
  int datum;
  Node *next;
};
```

Such a definition is *recursive*, where it uses itself as part of its definition.

Functions can also be recursive. As an example, consider the factorial of a nonnegative integer:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ n * (n-1) * \cdots * 1 & \text{otherwise} \end{cases}$$

We can implement an iterative function to compute this:

```
// REQUIRES: n >= 0
// EFFECTS: Computes and returns n!.
int factorial(int n) {
  int result = 1;
  while (n > 0) {
    result *= n;
    --n;
  }
  return result;
}
```

On the other hand, we can observe that $(n-1)! = (n-1) * \cdots * 1$, so that we can mathematically define factorial in terms of itself:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

Such a mathematical definition is called a *recurrence relation*. It translates into code as follows:

```
// REQUIRES: n >= 0
// EFFECTS: Computes and returns n!.
int factorial(int n) {
  if (n == 0 || n == 1) {
    return 1;
```

(continues on next page)

(continued from previous page)

```
  } else {
    return n * factorial(n - 1);
  }
}
```

This function is recursive, since it calls itself as part of its definition. We can then call `factorial()` as follows:

```
int main() {
  int result = factorial(3);
  cout << "3! = " << result << endl;
}
```

The activation records created by this program are shown in Figure 31.1.

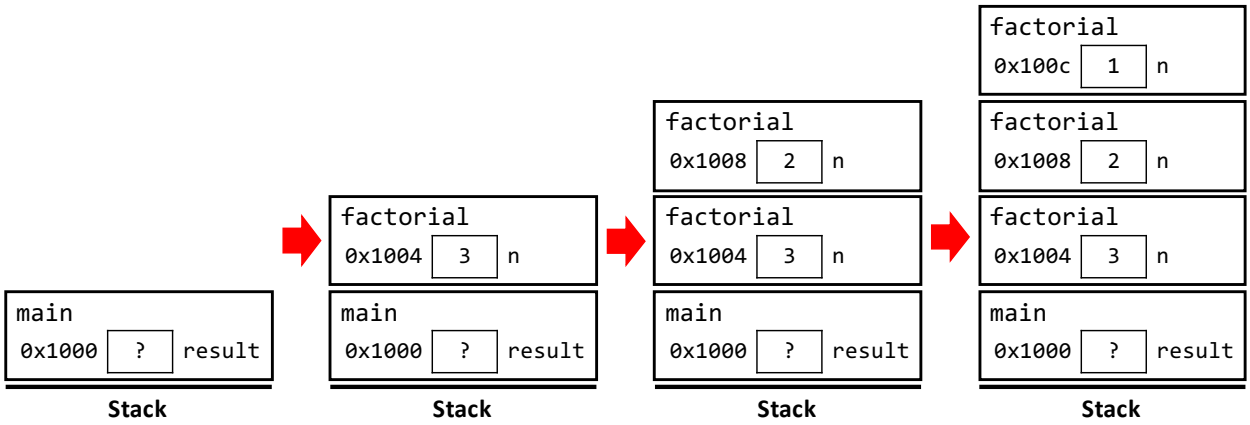

Figure 31.1: Activation records for a call to the recursive version of `factorial()`.

When the call `factorial(3)` is evaluated, an activation record is created, and the parameter `n` is initialized with value 3. The body of `factorial()` runs, and it calls `factorial(2)`. This creates another activation record, and its parameter `n` is initialized to 2. Within the context of that activation record, the body of `factorial()` is executed, and it calls `factorial(1)`. Another activation record is created, with `n` initialized to 1. The body runs, returning 1, which replaces the function call in the caller. The activation record for `factorial(1)` is destroyed, and `factorial(2)` resumes. The latter computes `2 * 1` and returns 2. The activation record for `factorial(2)` is destroyed, and `factorial(3)` continues where it left off. It computes `3 * 2`, returning 6. Its activation record is reclaimed, and `main()` resumes, initializing `result` to 6, the correct value for 3!.

Operationally, the recursive definition works because each invocation of `factorial()` gets its own activation record, and the body of the function is executed within the context of that activation record. More conceptually, it works by using `factorial()` as an abstraction, even while we are still in the midst of implementing that abstraction! We call this "the recursive leap of faith" – we trust that the abstraction works, even if we haven't finished writing it yet.

In general, a recursive abstraction requires the following:

- *base cases*, which are cases we can implement directly without recursion
- *recursive cases*, where we break down the problem into *subproblems* that are *similar* to the problem but "smaller," meaning closer to a base case

As another example, the following recursive function prints out the integers between a start and end:

```
// REQUIRES: end >= start
// MODIFIES: cout
// EFFECTS:  Prints the numbers in [start, end) in order.
void print_numbers(int start, int end) {
  // base case
  if (start == end) {
    return;
  }
  // recursive case
  cout << start << endl;
  print_numbers(start + 1, end);
}
```

The base case is when the start and end are the same, in which case the function immediately returns without printing anything. The recursive case prints out a single number and then recursively computes the subproblem of printing the numbers in $[start + 1, end)$. The latter is smaller than the original problem, since it requires printing out one fewer number.

Let us consider another example, that of computing the number of ducks on a duck farm. Suppose the farm starts with five baby ducklings. Assume that a duck lays three eggs each month, once the duck is at least a month old itself. Furthermore, an egg takes a month to hatch, and our ducks never die. How many ducks does the farm have after n months?

We start by working out the first few months by hand. In month 0, there are 5 baby ducklings, which are not old enough to lay eggs. In month 1, there are now 5 grown ducks, each of which lays 3 eggs, resulting in 15 eggs in total. Let's use a table to keep track of what we have:

Month	Adult Ducks	Ducklings	Eggs	Total Ducks
0	0	5	0	5
1	5	0	15	5
2	5	15	15	20
3	20	15	60	35
4	35	60	105	95
5	95	105	285	200

In month 2, the 15 eggs hatch, resulting in 15 ducklings. The 5 adult ducks lay 15 more eggs. In month 3, these eggs hatch, resulting in 15 ducklings. The previous 15 ducklings are now adults, so that we have a total of 20 adult ducks, which lay 60 eggs. And so on in months 4 and 5. The total number of ducks is 200 at month 5.

We observe that the number of adult ducks in any particular month is the same as the total number of ducks in the previous month. The number of ducklings in a month is three times the number of adult ducks in the previous month, which is the same as the total number of ducks in the month before. This results in the following recurrence relation for the total number of ducks:

$$ducks(n) = \begin{cases} 5 & \text{if } n = 0 \text{ or } n = 1 \\ ducks(n-1) + 3 * ducks(n-2) & \text{otherwise} \end{cases}$$

The base cases are the first two months, when there are 5 total ducks. (We cannot apply the recurrence for these cases, since it would rely on the number of ducks in month -1, which is ill-defined.)

We can now write a function to compute the number of ducks:

```
// REQUIRES: n >= 0
// EFFECTS:  Computes the number of ducks in month n, assuming 5
```

(continues on next page)

(continued from previous page)

```
//           ducklings to start.
int num_ducks(int n) {
  // base cases
  if (n <= 1) {
    return 5;
  }
  // recursive case
  return num_ducks(n - 1) + 3 * num_ducks(n - 2);
}
```

Observe that there are two subproblems here, `ducks(n - 1)` and `ducks(n - 2)`. Both are closer to a base case than `ducks(n)`, so the recursive computation still works.

Recursive algorithms are not just for math problems. The following is an algorithm for lifting a box of heavy books:

```
void lift_box(Box &box) {
  if (too_heavy(box)) {              // recursive case
    Book book = remove_book(box);
    lift_box(box);                   // "smaller" box
    add_book(box, book);
  } else {                           // base case
    move_box(box);
  }
}
```

Here, the base case is when we can directly move the box, such as when it is empty. Otherwise, we reduce the problem of moving a heavy box to the subproblem of a box with one less book, using recursion to solve that subproblem. We take the recursive leap of faith that the function will correctly solve that subproblem. We then need only add back the book we removed.

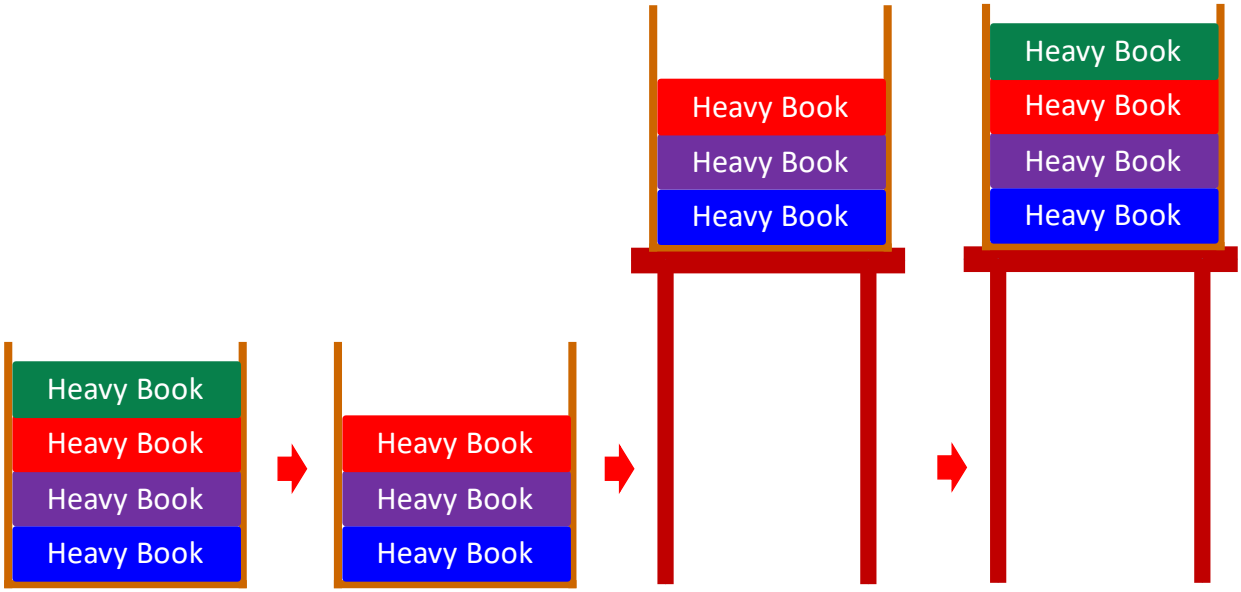

Figure 31.2: Recursive algorithm for lifting a box of heavy books.

As another non-math example, we write a recursive function to reverse the contents of an array. For an empty array or an array with one element, nothing needs to be done, constituting the base case. Otherwise, we reverse a smaller array,

one that excludes the ends, and then swap the two ends:

```
// EFFECTS: Reverses the array starting at 'left' and ending at
//          (and including) 'right'.
void reverse(int *left, int *right) {
  if (left < right) {
    reverse(left + 1, right - 1);
    int temp = *left;
    *left = *right;
    *right = temp;
  }
}
```

The function reverses the elements in $[left, right]$. The subproblem is the set of elements in $[left + 1, right - 1]$, which is closer to the base case of an array with zero or one element. As always, we take the recursive leap of faith that the subproblem will be computed correctly.

We can call `reverse()` as follows:

```
int main() {
  const int SIZE = 8;
  int array[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8 };
  reverse(array, array + SIZE - 1);             // reverse whole array
  reverse(array + 1, array + 3);                // reverse elements 1-3
}
```

31.1 Tail Recursion

Consider the efficiency of the `reverse()` function above. For an array of size n, it performs $\lfloor n/2 \rfloor$ swap operations, taking $O(n)$ time. The algorithm also uses $O(n)$ space: each recursive call has an activation record, and there are $\lfloor n/2 \rfloor + 1$ total calls.

On the other hand, the following iterative algorithm uses only a single activation record, resulting in constant, $O(1)$, space:

```
// EFFECTS: Reverses the array starting at 'left' and ending at
//          (and including) 'right'.
void reverse(int *left, int *right) {
  while (left < right) {
    int temp = *left;
    *left = *right;
    *right = temp;
    ++left;
    --right;
  }
}
```

The fundamental difference is that this algorithm solves the subproblem **after** doing the extra work for the original problem, i.e. swapping the two ends. An equivalent recursive implementation would be the following:

```
// EFFECTS: Reverses the array starting at 'left' and ending at
//          (and including) 'right'.
void reverse(int *left, int *right) {
```

(continues on next page)

(continued from previous page)

```
  if (left < right) {
    int temp = *left;
    *left = *right;
    *right = temp;
    reverse(left + 1, right - 1);
  }
}
```

Here, the recursive call is a *tail call*, meaning that the call is the last thing that happens in the function, and no work is done after the tail call returns.[1] Since no work is done after the tail call, there is no need to retain the activation record of the caller, and it can be discarded.

Many compilers recognize tail calls and perform *tail-call optimization (TCO)*, where the activation record for the tail call reuses the space of the caller's activation record. In the `g++` compiler, TCO is enabled at optimization level 2 (`-O2`). TCO is not restricted to recursive functions; as long as a function call is the last thing that happens in its caller, it is a tail call and the optimization can apply. However, tail-call optimization is most important for recursive functions, since it can reduce the space usage of the computation by a large factor.

A function is said to be *tail recursive* if it is recursive, and all recursive calls are tail calls. Rather than using a linear amount of space, a tail-recursive computation requires only constant space when tail-call optimization is applied. For instance, the tail-recursive version of `reverse()` uses space for only a single activation record under TCO.

In order for a computation to be tail recursive, it must do all its work in the *active flow* of the computation, before making a recursive call. A computation that does its work in the *passive flow* must wait until a recursive call completes before doing work, so that the recursive call is not a tail call.

As a concrete example, the prior recursive implementation of `factorial()` is not tail recursive, since it does its work in the passive flow:

```
int factorial(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

The multiplication must be done after the recursive call returns, so the function is not tail recursive.

For the function to be tail recursive, we need the multiplication to be in the active flow. To do so, we compute n in the initial call to `factorial()` on n, $n * (n-1)$ in the call on $n-1$, $n * (n-1) * (n-2)$ in the call on $n-2$, and so on until we reach 1. At each step, we keep track of the product so far:

```
int factorial(int n, int resultSoFar) {
  if (n == 0 || n == 1) {
    return resultSoFar;
  } else {
    return factorial(n - 1, n * resultSoFar);
  }
}
```

To call this function, we need to seed `resultSoFar` with the multiplicative identity of 1:

[1] Whether or not a function call is a tail call is not a syntactic property, but is determined by whether or not work must be done after the call returns. For example, if a nontrivial destructor for a local variable must be run after the call returns, the call is not a tail call.

```
cout << factorial(5, 1) << endl;
```

However, this is a different interface than our previous versions of `factorial()`. To retain the same interface, we move the actual computation to a helper function and abstract the call to it:

```
static int factorial_helper(int n, int resultSoFar) {
  if (n == 0 || n == 1) {
    return resultSoFar;
  } else {
    return factorial_helper(n - 1, n * resultSoFar);
  }
}

int factorial(int n) {
  return factorial_helper(n, 1);
}
```

Helper functions are a common pattern for tail-recursive computations that require a seed value. Not all tail-recursive algorithms require a helper function, however; the tail-recursive `print_numbers()` and `reverse()` functions above do not need a seed value, so they work without helper functions.

31.2 Kinds of Recursion

We have now seen several different kinds of recursion. A function is *linear recursive* if it is recursive, but each invocation of the function makes at most one recursive call. Such a function reduces each recursive case to a single subproblem. The various recursive `factorial()` and `reverse()` functions above are all linear recursive.

A function is *tail recursive* if it is linear recursive, and every recursive call is the last thing that happens in the invocation that makes the recursive call. We have seen both tail-recursive and non-tail-recursive variants of `reverse()` and `factorial()`.

A function is *tree recursive* if a single invocation of the function can make more than one recursive call. Such a function subdivides a recursive case into multiple subproblems. The `num_ducks()` function above is tree recursive. Drawing out the recursive call graph for a tree-recursive function, we end up with a branching structure that resembles a tree, explaining the nomenclature.

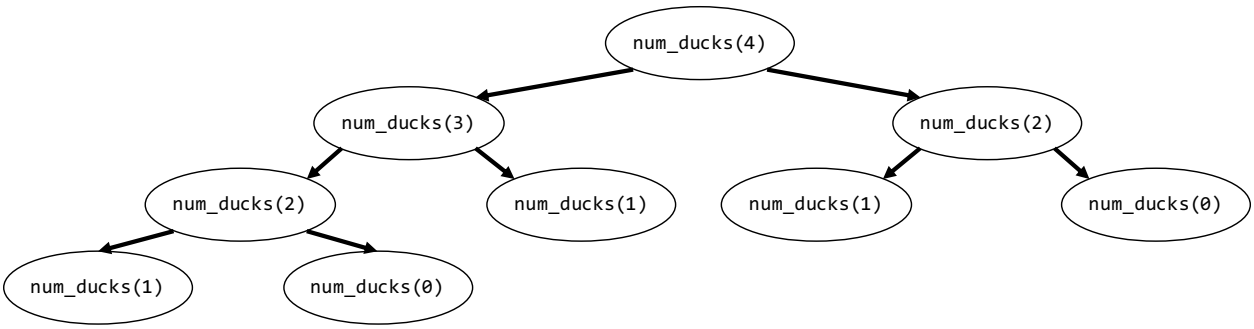

Figure 31.3: Call structure of a tree-recursive function.

31.3 Iteration vs. Recursion

Iteration and recursion are two approaches to solving complex problems. Conceptually, an iterative algorithm often divides a computation into individual discrete steps, the combination of which forms a solution to the whole problem. In contrast, recursive algorithms generally express a computation in terms of smaller versions of the same problem.

Both iteration and recursion have the same computational power; an iterative algorithm can be converted to a tail-recursive implementation and vice versa. A non-tail-recursive computation can also be rewritten iteratively; however, the iterative version may require explicit storage. The non-tail-recursive algorithm can store data in multiple activation records, while the iterative one only has a single activation record to work with, so it may need an explicit data structure such as a vector to store its data. Techniques such as *dynamic programming* can be used in general to convert a recursive algorithm to an iterative one, but they are beyond the scope of this course.

CHAPTER

THIRTYTWO

STRUCTURAL RECURSION

Last time, we discussed the concept of recursion, where an abstraction uses itself as part of its implementation, and we saw its application to procedural abstractions. We turn our attention now to *structural recursion*, where we use recursion in defining the representation of an abstract data type.

32.1 Recursive Lists

The data representation of a linked list is an example of structural recursion: the `Node` type uses itself in its own representation:

```
struct Node {
  int datum;
  Node *next;
};
```

This representation satisfies the requirements for a recursive abstraction:

- The empty list is the base case, represented by a null pointer.
- A non-empty list is a recursive case; it can be subdivided into the first node and the rest of the list, which represents a list in its own right.

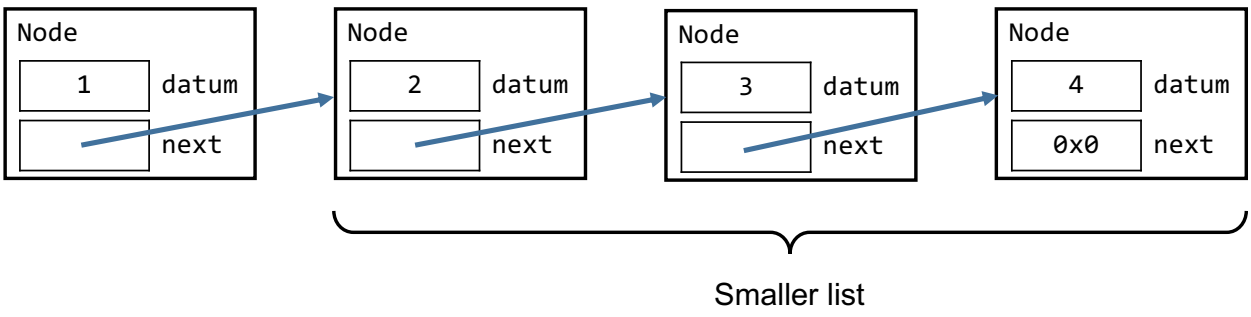

Figure 32.1: The representation of a non-empty list consists of a node followed by the representation of a smaller list.

Independent of its representation, a linked list can actually be defined recursively as either an empty list, or a datum followed by a smaller list.

Given the recursive definition of a list, it is natural to process a list with a recursive function. The base case of the recursive function will be the minimum-size list allowed by the function, and larger lists will be handled in the recursive case. As an example, the following is a recursive function to compute the length of a list:

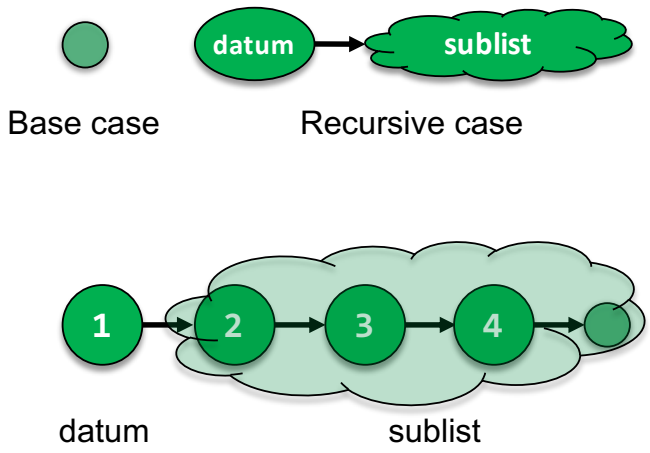

Figure 32.2: Recursive definition of a list.

```
// REQUIRES: node represents a valid list
// EFFECTS:  Computes the length of the list that starts at the
//           given node.
int length(const Node *list) {
  if (list == nullptr) {                // empty list
    return 0;
  } else {                              // non-empty list
    return 1 + length(list->next);      // list->next is a smaller list
  }
}
```

The length of an empty list is 0. The length of a non-empty list is one more than the length of the rest of the list; the elements in the list consist of the initial datum and the elements in the rest of the list. We use the `length()` function itself as an abstraction to compute the number of elements in the remainder of the list, taking the recursive leap of faith that it will compute the right answer.

As another example, consider the problem of finding the maximum element in a list. Unlike for `length()`, the minimal list is not an empty one, since an empty list has no elements in it. Instead, the minimum required size is a list with a single datum, in which case the maximum element is just that lone datum, constituting our base case. For a larger list, we can break it down recursively as follows:

1. Find the maximum element in the rest of the list. This element is at least as large as any other element in the rest of the list.
2. Compare the first element to the max of the remainder. The larger of the two is transitively at least as large as the elements in the rest of the list.

The following implements this algorithm:

```
// REQUIRES: node represents a valid, non-empty list
// EFFECTS:  Returns the maximum element in the list that starts at
//           the given node.
int list_max(const Node *list) {
  if (list->next == nullptr) {              // list has only one element
    return list->datum;
  } else {                                  // list has more than one element
    return std::max(list->datum,            // compare first datum to
```

(continues on next page)

(continued from previous page)

```
                    list_max(list->next));  // max of rest of list
  }
}
```

The base case is a list with one element. Such a list has an empty `next` list, so we check for that and return the list's lone datum. The recursive case computes the max of the rest of the list and then uses `std::max()` to determine the maximum of that item and the first item in the list. As always, we take the recursive leap of faith, assuming that the recursive call to `list_max()` computes the right answer for the smaller list.

Exercise 2 Write a function `list_sum()` that recursively computes the sum of the elements in a list.

```
// EFFECTS: Computes the sum of the elements in the list that
//          starts at the given node.
int list_sum(const Node *node) {
  // your code here
}
```

Exercise 3 Write a function `last()` that recursively finds and returns the last element of a non-empty list.

```
// REQUIRES: node represents a valid, non-empty list
// EFFECTS:  Returns the last element in the list that starts at
//           the given node.
int last(const Node *node) {
  // your code here
}
```

32.2 Trees

A list is a linear-recursive data structure: each non-empty list is subdivided into a datum and a single smaller list. A *tree* is a data structure that is, naturally, tree recursive. A non-empty tree is subdivided into a datum and several smaller trees. In this course, we only consider *binary trees*, where non-empty trees are subdivided into exactly two smaller trees.

The term *tree* stems[1] from the fact that its branching structure resembles that of a botanical tree. Terminology with respect to tree data structures borrows from both botanical and family trees.

- The *root* is the node that originates the branching structure. In our diagrams, the root is pictured at the top of the tree.
- A non-empty tree consists of a *parent* node and two *child* nodes. For a binary tree, there is a *left child* and a *right child*. Nodes that have the same parent are *siblings*.
- A node whose children are all empty is a *leaf*.
- The size of a tree is the number of elements it contains.
- The *height* of a tree is the number of levels at which it has elements. Equivalently, it is the length of the longest path from the root to a leaf node.

Algorithms on trees are often written as tree-recursive functions, so that the recursive case makes more than one recursive call. The general strategy is to directly compute the result for the smallest tree allowed by the function, constituting the base case. The recursive case makes recursive calls to compute the results for the left and right children, then combines those results with the datum at the root to compute the answer for the whole tree.

[1] Pun intended.

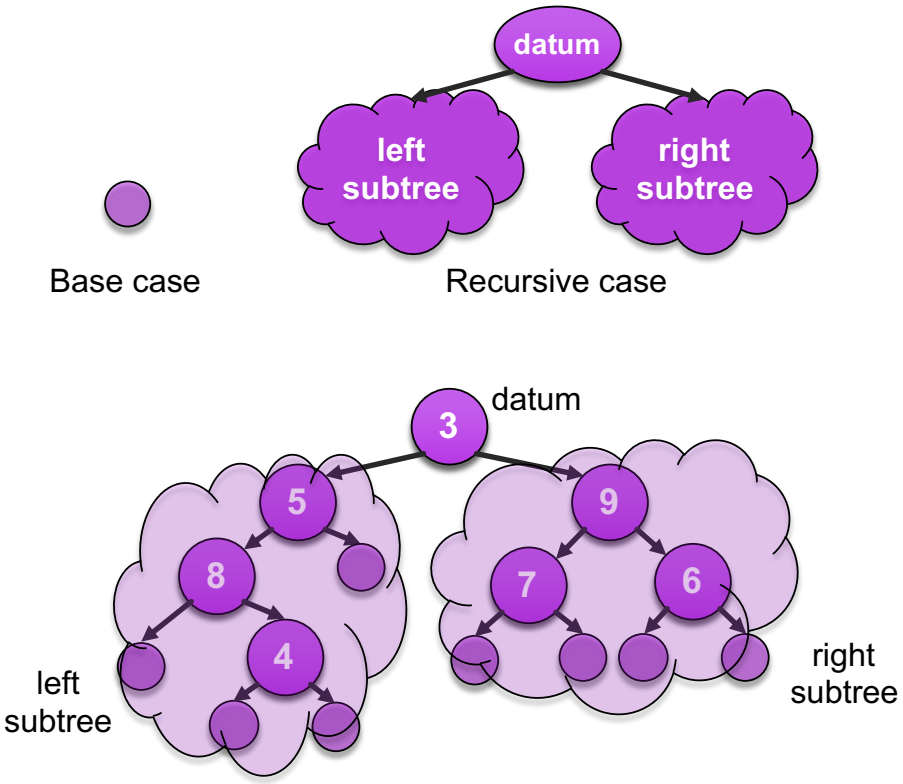

Figure 32.3: Recursive definition of a tree. A non-empty tree consists of a datum and two smaller trees.

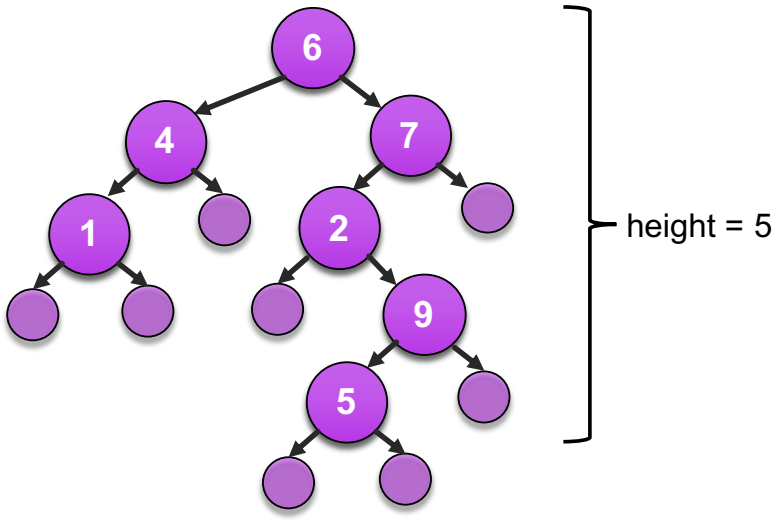

Figure 32.4: The height of a tree is the number of levels that contain elements, or equivalently, the length of the longest path from root to a leaf.

As an example, the following algorithm computes the size of a tree:

- The size of an empty tree is zero.
- The size of a non-empty tree is the size of the left child, plus the size of the right child, plus one for the root datum.

Before we can implement this algorithm in code, we need a data representation. As with a list, we use a `Node` struct, but it now has a datum and pointers to left and right `Node`s:

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

Like a list, we use a null pointer to represent an empty tree.

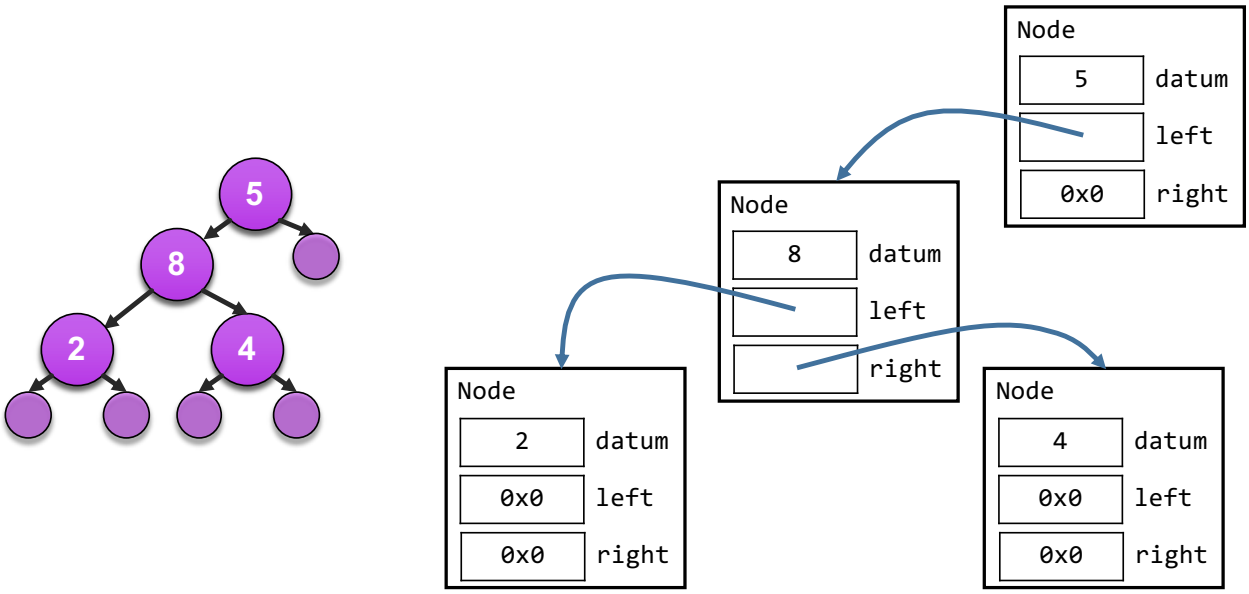

Figure 32.5: Data representation of a tree, using a node for each element and null pointers for empty trees.

We can now implement the `size()` function:

```
// REQUIRES: node represents a valid tree
// EFFECTS:  Returns the number of elements in the tree represented
//           by the given node.
int size(const Node *tree) {
  if (!tree) {                                    // empty tree
    return 0;
  } else {                                        // non-empty tree
    return 1 + size(tree->left) + size(tree->right);
  }
}
```

As with other recursive functions, we take the recursive leap of faith that `size()` computes the right answer on smaller trees.

The height of a tree is the number of levels it contains. An empty tree contains no levels, so its height is zero. For

a non-empty tree, we exploit the alternate definition of height, that it is the length of the longest path from root to leaf. The longest such patch is just one node longer than the longest path in the child subtrees, since the root adds one additional node to the part of the path contained in a child. Thus, we compute the height as follows:

```
// REQUIRES: node represents a valid tree
// EFFECTS:  Returns the height of the tree represented by the given
//           node.
int height(const Node *tree) {
  if (!tree) {                                        // empty tree
    return 0;
  } else {                                            // non-empty tree
    return 1 + std::max(height(tree->left),
                        height(tree->right));
  }
}
```

We use `std::max()` to obtain the longer path from the two child trees, then add one to the result to account for the root node.

Exercise 4 Write a function `tree_contains()` that determines whether the given value is contained in the given tree.

```
// EFFECTS: Returns whether the tree rooted at the given node
//          contains the given value.
bool tree_contains(const Node *node, int value) {
  // your code here
}
```

Exercise 5 Write a function `count_leaves()` that counts the number of leaf nodes in the given tree.

Hint: You will likely need two base cases.

```
// EFFECTS: Returns the number of leaf nodes in the tree rooted
//          at the given node.
int count_leaves(const Node *node) {
  // your code here
}
```

32.2.1 Tree Traversals

The following function prints every element in a tree to standard output:

```
// REQUIRES: node represents a valid tree
// MODIFIES: cout
// EFFECTS:  Prints each element in the given tree to standard out,
//           with each element followed by a space.
void print(const Node *tree) {
  if (tree) {               // non-empty tree
    cout << tree->datum << " ";
    print(tree->left);
    print(tree->right);
  }
}
```

This algorithm processes a root datum before processing any of the data in the children. It thus implements a *preorder traversal*. For the tree in Figure 32.4, it prints the elements in the order `6 4 1 7 2 9 5`.

Moving the print statement results in alternate orderings. The following implements an *inorder traversal*:

```
if (tree) {              // non-empty tree
  print(tree->left);
  cout << tree->datum << " ";
  print(tree->right);
}
```

The data in the left child is processed first, then the root datum, then the data in the right child. For the tree with height 5, it prints `1 4 6 2 5 9 7`.

A *postorder* traversal processes the root datum after the data in the children:

```
if (tree) {              // non-empty tree
  print(tree->left);
  print(tree->right);
  cout << tree->datum << " ";
}
```

For the same tree, the postorder traversal prints `1 4 5 9 2 7 6`.

The structure of a binary tree can be uniquely determined by knowing any two of these orderings.

CHAPTER

THIRTYTHREE

BINARY SEARCH TREES (BSTS)

Last time, we saw binary trees, which are a recursive data structure that is either empty, or consists of an element and two subtrees. A *binary search tree (BST)* is a binary tree whose elements are stored in an order that maintains a sorting invariant. Specifically, a binary search tree is either:

- empty, or
- a root datum with two subtrees such that:
  1. The two subtrees are themselves binary search trees.
  2. Every element in the left subtree is strictly less than the root datum.
  3. Every element in the right subtree is strictly greater than the root datum.[1]

Figure 33.1 shows an example of a binary search tree.

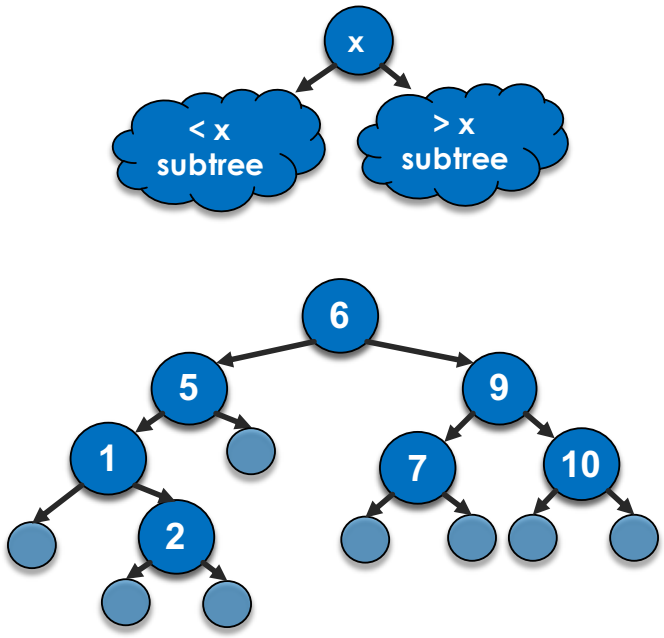

Figure 33.1: A binary search tree.

Every element in the left subtree is less that the root datum, and every element in the right subtree is greater than the root. The left and right subtrees meet the requirements for a binary search tree. Thus, the whole tree is a binary search tree.

[1] Our binary search trees do not permit duplicate elements, since we will be using them to build set and map abstractions. If a BST does permit duplicates, it will use a modified definition that allows items equal to the root in one of the subtrees.

Figure 33.2 depicts trees that do not meet the definition of a binary search tree.

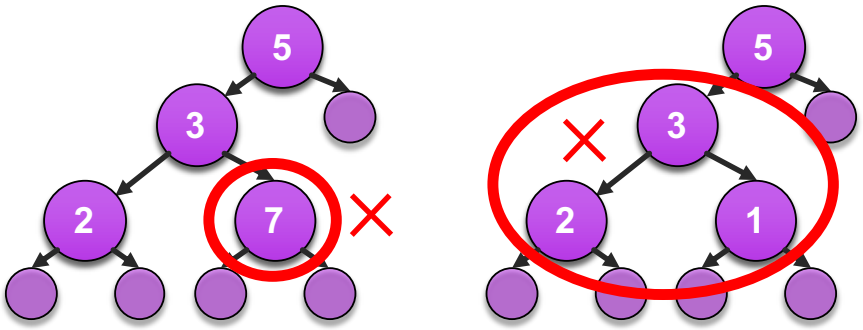

Figure 33.2: Trees that violate the invariants for a binary search tree.

In the tree on the left, the left subtree contains the element 7, which is not smaller than the root element 5. This violates the second condition in the recursive definition above of a BST. In the tree on the right, the right subtree is empty, which is a valid binary search tree. However, the left subtree is not a valid BST, since it does not meet the sorting invariant for a BST. Thus, the tree on the right is not a binary search tree.

A binary search tree is thus named because searching for an element can be done efficiently, in time proportional to the height of the tree rather than the size. A search algorithm need only recurse on one side of the tree at each level. For example, in searching for the element 2 in the BST in Figure 33.1, the element must be in the left subtree, since 2 is less than the root element 6. Within the left subtree, it must again be to the left, since 2 is less than 5. Within the next subtree, the 2 must be to the right of the 1, leading us to the actual location of the 2.

More generally, the following algorithm determines whether or not a BST contains a particular value:

- If the tree is empty, it does not contain the value.
- Otherwise, if the root datum is equal to the value, the tree contains the element.
- If the value is less than the root element, it must be in the left subtree if it is in the tree, so we repeat the search on the left subtree.
- Otherwise, the value is greater than the root element, so it is in the right subtree if it is in the tree at all. Thus, we repeat the search on the right subtree.

The first two cases above are base cases, since they directly compute an answer. The latter two are recursive cases, and we take the recursive leap of faith that the recursive calls will compute the correct result on the subtrees.

The algorithm leads to the following implementation on our tree representation:

```
// REQUIRES: node represents a valid binary search tree
// EFFECTS:  Returns whether or not the given value is in the tree
//           represented by node.
bool contains(const Node *node, int value) {
  if (!node) {                              // empty tree
    return false;
  } else if (node->datum == value) {        // non-empty tree, equal to root
    return true;
  } else if (value < node->datum) {         // less than root
    return contains(node->left, value);
  } else {                                  // greater than root
    return contains(node->right, value);
  }
}
```

This implementation is linear recursive, since at most one recursive call is made by each invocation of `contains()`. Furthermore, every recursive call is a tail call, so the implementation is tail recursive. This example illustrates that the body of a linear- or tail-recursive function may contain more than one recursive call, as long as at most one of those calls is actually made.

Let us consider another algorithm, that of finding the maximum element of a BST, which requires there to be at least one element in the tree. If there is only one element, then the lone, root element is the maximum. The root is also the maximum element when the right subtree is empty; everything in the left subtree is smaller than the root, making the root the largest item. On the other hand, when the right tree is not empty, every element in that subtree is larger than the root and everything in the left subtree. Then the largest element in the whole tree is the same as the largest element in the right subtree.

Summarizing this, we have the following algorithm:

- If the right subtree is empty, then the root is the maximum.
- Otherwise, the maximum item is the maximum element in the right subtree.

We implement the algorithm as follows:

```
// REQUIRES: node represents a valid non-empty binary search tree
// EFFECTS:  Returns the maximum element in the tree represented by
//           node.
int tree_max(const Node *node) {
  if (!node->right) {                // base case
    return node->datum;
  } else {                           // recursive case
    return tree_max(node->right);
  }
}
```

As with `contains()`, `tree_max()` is tail recursive, and it runs in time proportional to the height of the tree.

33.1 The `BinarySearchTree` Interface

In the full linked-list definition we saw in *Linked Lists*, we defined a separate `IntList` class to act as the interface for the list, defining `Node` as a member of that class. We then generalized the element type, resulting in a `List` class template. We follow the same strategy here by defining a `BinarySearchTree` class template as the interface for a BST.

```
template <typename T>
class BinarySearchTree {
public:
  // EFFECTS:  Constructs an empty BST.
  BinarySearchTree();

  // EFFECTS:  Constructs a copy of the given tree.
  BinarySearchTree(const BinarySearchTree &other);

  // EFFECTS:  Replaces the contents of this tree with a copy of the
  //           given tree.
  BinarySearchTree & operator=(const BinarySearchTree &other);

  // EFFECTS:  Destructs this tree.
```

(continues on next page)

(continued from previous page)

```
  ~BinarySearchTree();

  // EFFECTS:  Returns whether this tree is empty.
  bool empty() const;

  // EFFECTS:  Returns the number of elements in this tree.
  int size() const;

  // EFFECTS:  Returns whether the given item is contained in this
  //           tree.
  bool contains(const T &value) const;

  // REQUIRES: value is not in this tree
  // EFFECTS:  Inserts the given item into this tree.
  void insert(const T &value);

private:
  // Represents a single node of a tree.
  struct Node {
    T datum;
    Node *left;
    Node *right;
    // INVARIANTS: left and right are either null or pointers to
    //             valid Nodes
  };

  // The root node of this tree.
  Node *root;
  // INVARIANTS: root is either null or a pointer to a valid Node
};
```

As with a list, we define `Node` as a nested class of `BinarySearchTree` to encapsulate it within the latter ADT. Since it is an implementation detail and not part of the BST interface, we define `Node` as a private member.

The `contains()` member function differs from the one we defined before; the member function just takes in a data item, while our previous definition operates on a node as well. We define the latter as a private helper function and call it with the root node:

```
template <typename T>
class BinarySearchTree {
  ...
public:
  bool contains(const T &value) const {
    return contains_impl(root, value);
  }

private:
  bool contains_impl(const Node *node, const T &value) {
    if (!node) {
      return false;
    } else if (node->datum == value) {
      return true;
    } else if (value < node->datum) {
```

(continues on next page)

(continued from previous page)

```
      return contains_impl(node->left, value);
    } else {
      return contains_impl(node->right, value);
    }
  }

  Node *root;
};
```

Observe that `contains_impl()` does not refer to a `BinarySearchTree` or any of its members. Thus, there is no need for it to have a `this` pointer to a `BinarySearchTree` object. We can declare the function as a *static member function* to eliminate the `this` pointer.

```
template <typename T>
class BinarySearchTree {
  ...
public:
  bool contains(const T &value) const {
    return contains_impl(root, value);
  }

private:
  static bool contains_impl(const Node *node, const T &value) {
    if (!node) {
      return false;
    } else if (node->datum == value) {
      return true;
    } else if (value < node->datum) {
      return contains_impl(node->left, value);
    } else {
      return contains_impl(node->right, value);
    }
  }

  Node *root;
};
```

Just like a *static member variable* is associated with a class rather than an individual object, a static member function is also not associated with an individual object, and it cannot refer to non-static member variables.

A public static member function can be called from outside the class using the scope-resolution operator, the same syntax for referring to a static member variable:

```
BinarySearchTree<int>::contains_impl(nullptr, -1);
// compile error because contains_impl() is not public
```

33.2 BST-Based Set

Previously, we have seen array-based set implementations, one that used an unsorted array and another a sorted array. We can also implement a set using a binary search tree to store the data:

```
template <typename T>
class BSTSet {
public:
  // EFFECTS: Inserts the given value into this set, if it is not
  //          already in the set.
  void insert(const T &value) {
    if (!elts.contains(value)) {
      elts.insert(value);
    }
  }

  // EFFECTS: Returns whether value is in this set.
  bool contains(const T &value) const {
    return elts.contains(value);
  }

  // EFFECTS: Returns the size of this set.
  int size() const {
    return elts.size();
  }

private:
  BinarySearchTree<T> elts;
};
```

If the underlying BST is *balanced*, meaning that each subtree within the BST has close to the same number of elements in its left and right child, then the height of the tree is in $O(\log n)$, where n is the size of the tree. Thus, the `contains()` and `insert()` operations take logarithmic time, rather than the linear time they would take on an unsorted set.

Unfortunately, our BST implementation does not guarantee that it will be balanced. In fact, inserting items in increasing order results in a maximally unbalanced tree as in Figure 33.3, resembling a list rather than a tree structure.

There are more complicated binary-search-tree implementations that do guarantee balance, but they are beyond the scope of this course.

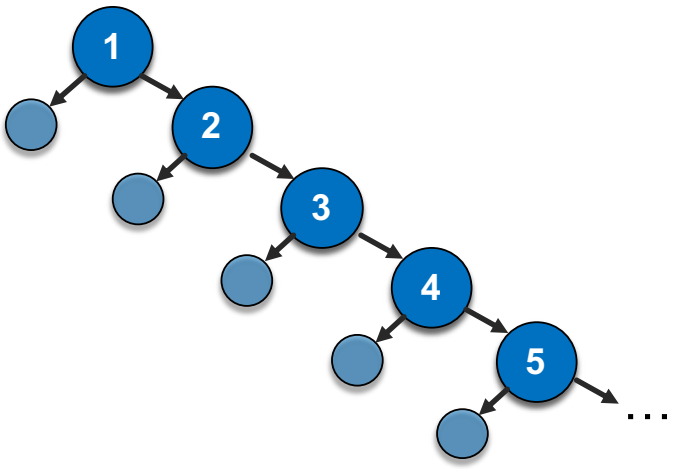

Figure 33.3: Inserting elements into a non-balancing binary search tree in order results in a linear structure, with height equal to the number of elements.

CHAPTER

THIRTYFOUR

MAPS AND PAIRS

A *map* is a data structure that maps *keys* to *values*. It is thus an *associative* data structure, since it associates a value with each key. As an example, we may want to associate exam scores with individual test takers:

aliceywu	100
akamil	23
taligoro	100
jjuett	73

We can store these data in a standard-library map:

```
std::map<std::string, int> scores;
scores["aliceywu"] = 100;
scores["akamil"] = 23;
scores["taligoro"] = 100;
scores["jjuett"] = 73;
cout << scores["akamil"] << endl;    // prints 23
cout << scores["aliceywu"] << endl;  // prints 100
```

The `map` class template has two required template parameters, the first for the key type and the second for the value type. We are mapping string IDs to integer scores, so we use a `map<string, int>`. We can then use the subscript operator with a key to insert a key-value pair into the map, or to look up the value associated with a key.

A key feature of the standard-library map is that it is not erroneous to look up a non-existent key. Instead, the map inserts and returns a *value-initialized* datum for the key. For primitive types, value initialization produces the zero value for the type. (This is distinct from default initialization, which produces an undefined value.)

```
cout << scores["gmatute"] << endl;  // prints 0
```

A map is similar to a set in that the keys in the map are unique; each key is associated with at most one value. However, the values themselves are not unique. The map above has two keys that are associated with the value 100.

Given the similarity with a set, we can also implement a map using a binary search tree. However, rather than using the key-value pair for ordering and uniqueness, we need to use just the key, and the value merely tags along for the ride, as shown in Figure 34.1.

To make this work, we need a heterogeneous data type for the `datum` member of a node, so that it can store separate key and value items. We can define our own struct or class, or we can use the `pair` template in the standard `<utility>` library.

```
std::pair<int, bool> p1;   // default constructor value initializes both items
p1.first -= 5;                                        // first now has value -5
```

(continues on next page)

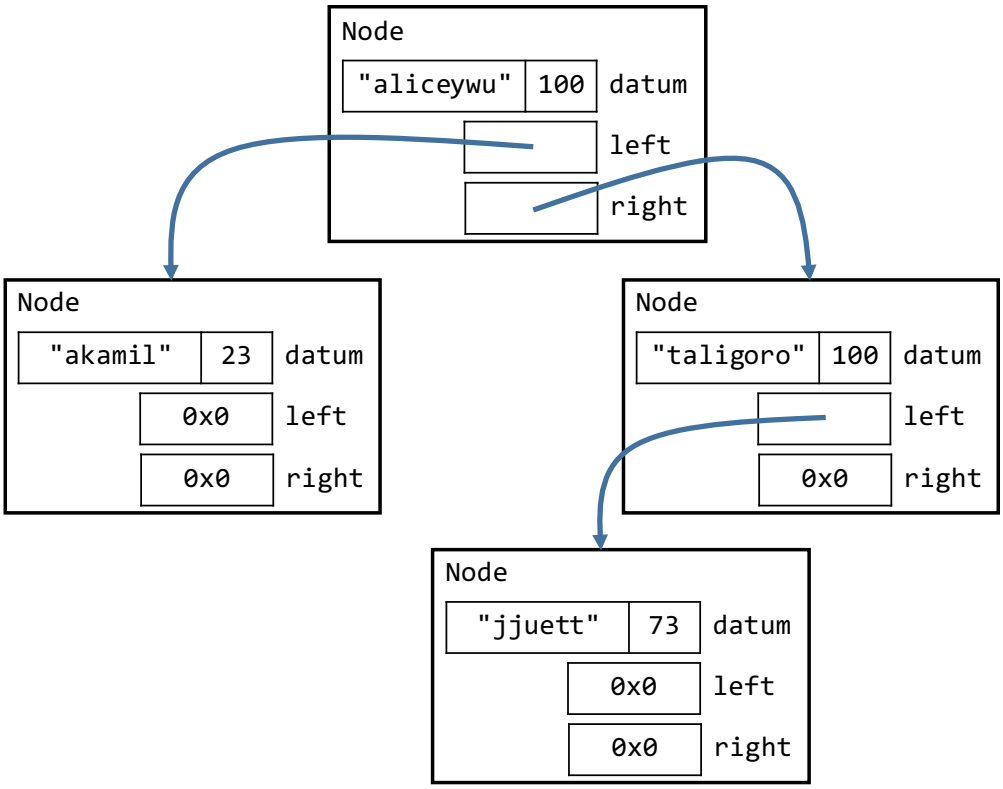

Figure 34.1: A map represented as a binary search tree.

(continued from previous page)

```
p1.second = false;

std::pair<string, int> p2 = { "hello", 4 };         // explicit initialization
cout << p2.first << ": " << p2.second << endl;              // prints hello: 4
```

A pair stores a first and second item, which can be of different types. The `pair` template is parameterized by the types of these items.

We can use a pair and a BST to implement a map:

```
template <typename Key_type, typename Value_type>
class Map {
public:
  // EFFECTS: Returns whether this map is empty.
  bool empty() const;

  // EFFECTS: Returns the number of key-value pairs in this map.
  size_t size() const;

  // EFFECTS: Returns the value associated with the given key. If
  //          the key is not in the map, inserts the key,
  //          associating it with a value-initialized object of type
  //          Value_type, and returns the newly inserted value.
  Value_type& operator[](const Key_type& key);
```

(continues on next page)

(continued from previous page)

```
private:
  // Type alias for a key-value pair.
  using Pair_type = std::pair<Key_type, Value_type>;

  // Comparator that compares pairs by their key components.
  class PairLess {
  public:
    bool operator()(...) const;
  };

  // Data representation.
  BinarySearchTree<Pair_type, PairLess> entries;
};
```

For this to work, we need a `BinarySearchTree` that can take a custom comparator, to compare key-value pairs by just the value component. This comparator can be defaulted to `std::less`, which compares elements according to the < operator:

```
template <typename T, typename Compare=std::less<T>>
class BinarySearchTree {
  ...
};
```

All comparisons within the tree now must use an object of the `Compare` type. We leave the details as an exercise for the reader.

Part V

Odds and Ends

CHAPTER

THIRTYFIVE

ERROR HANDLING AND EXCEPTIONS

Thus far, we have largely ignored the problem of handling errors in a program. We have included REQUIRES clauses in many of our functions, specifying that the behavior is undefined when the requirements are violated. However, for functions that interact with data external to the program such as user input, files, or other forms of I/O, it is unrealistic to assume that the data will always be available and valid. Instead, we need to specify well-defined behavior for what happens when the requirements for such a function are violated.

In writing a complex program, we subdivide it into individual functions for each task. Each function only knows how to handle its own job, and it returns results to its caller. As a result, it generally is not the case that the function that detects an error is equipped to handle it. Consider the following example:

```
// EFFECTS: Opens the file with the given filename and returns the
//          contents as a string.
string read_file_contents(const string &filename) {
  ifstream fin(filename);
  if (!fin.is_open()) {
    ???                       // file did not open; now what?
  }
  ...
}
```

The job of `read_file_contents()` is to read data from a file. It doesn't know anything further about the program, so it isn't in a position to determine what should be done in case of an error. Depending on the program, the right course of action may be to just print an error message and quit, prompt the user for a new filename, ignore the given file and keep going, and so on. Furthermore, the `read_file_contents()` function may actually be used in several different contexts, each of which requires a different form of error recovery. The only thing that `read_file_contents()` should be responsible for is detecting and conveying that an error occurred, and a different function further up in the call stack should do what is required to handle the error.

Thus, we need a mechanism that separates error detection from error handling, as well as a means for notifying a caller that an error occurred. We will look at several different strategies for doing so.

35.1 Global Error Codes

A common strategy in the C language is to set the value of a global variable to an error code, which provides information about the type of error that occurred. For example, many implementations of C and C++ functions use the `errno` global variable to signal errors. The following is an example that calls the `strtod()` standard-library function to read a double from a C-style string:

```
#include <cerrno>
#include <cstdlib>
```

(continues on next page)

(continued from previous page)

```
#include <iostream>

using std::cout;
using std::endl;

int main(int argc, char **argv) {
  char *end;
  errno = 0;                                // reset errno before call
  double number = std::strtod(argv[1], &end);
  if (errno == ERANGE) {                    // check errno after call
    cout << "range error" << endl;
  } else {
    cout << number << endl;
  }
}
```

When given the representation of a number that is out of range of the `double` type, `strtod()` sets `errno` to `ERANGE`:

```
$ ./main.exe 1e1000
range error
```

The strategy of setting a global error code can be (pardon the pun) error-prone. For example, `errno` must be set to zero before calling a standard-library function that uses it, and the caller must also remember to check its value after the function call returns. Otherwise, it may get overwritten by some other error down the line, masking the original error.

35.2 Object Error States

Another strategy used by C++ code is to set an error state on an object. This avoids the negatives of a global variable, since each object has its own error state rather than sharing the same global variable. C++ streams use this strategy:

```
int main(int argc, char **argv) {
  std::ifstream input(argv[1]);
  if (!input) {
    cout << "error opening file" << endl;
  }
}
```

As with a global variable, the user must remember to check the error state after performing an operation that may set it.

35.3 Return Error Codes

Return error codes are another strategy used in C and C++ code to signal the occurrence of an error. In this pattern, a value from a function's return type is reserved to indicate that the function encountered an error. The following is an example of a `factorial()` function that uses this strategy:

```
// EFFECTS: Computes the factorial of the given number. Returns -1
//          if n is negative.
int factorial(int n) {
```

(continues on next page)

(continued from previous page)

```
  if (n < 0) {
    return -1;                    // error
  } else if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

As with a global error code or an object state, it is the responsibility of the caller to check the code to determine whether an error occurred. Furthermore, a return error code only works if there is a value that can be reserved to indicate an error. For a function such as `atoi()`, which converts a C-style string to an int, there is no such value. In fact, `atoi()` returns 0 for both the string `"0"` and `"hello"`, and the caller cannot tell whether or not the input was erroneous.

35.4 Exceptions

All three strategies above have the problem that the caller of a function must remember to check for an error; neither the compiler nor the runtime detect when the caller fails to do so. Another common issue is that error checking is interleaved with the regular control flow of the caller, as in the following:

```
int main(int argc, char **argv) {
  double number = std::stod(argv[1]);
  string input;
  cin >> input;
  if (!input) {
    cout << "couldn't read input" << endl;
  } else if (input == "sqrt") {
    cout << std::sqrt(number) << endl;
  } else if (input == "log") {
    cout << std::log(number) << endl;
  }
  ...
}
```

The control flow for the error case is mixed in with that of non-error cases, making the code less readable and harder to maintain. Furthermore, the code above fails to check for errors in `sqrt()` or `log()`, which is not immediately clear due to the interleaving of control flow.

What we want is a mechanism for error handling that:

- Separates error detection from error handling, providing a general means for signaling that an error occurred.
- Separates the control flow for error handling from that of non-erroneous cases.
- Detects when an error is not handled, which by default should cause the program to exit with a meaningful message.

The mechanism provided by C++ and other modern languages is *exceptions*.

The following is an example of code that uses exceptions:

```
// Represents an exception in computing a factorial.
class FactorialError {};
```

(continues on next page)

(continued from previous page)

```
// EFFECTS: Computes the factorial of the given number. Throws a
//          FactorialError if n is negative.
int factorial(int n) {
  if (n < 0) {                      // error case
    throw FactorialError();         // throw an exception
  }
  if (n == 0) {                     // non-error case
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}

int main(int argc, char **argv) {
  try {
    int n = std::stoi(argv[1]);
    int result = factorial(n);
    cout << n << "! = " << result << endl;
  } catch (const std::invalid_argument &error) {
    cout << "Error converting " << argv[1] << " to an int: "
         << error.what() << endl;
  } catch (const FactorialError &error) {
    cout << "Error: cannot compute factorial on negative number"
         << endl;
  }
}
```

The individual components of the exception mechanism used above are:

- The `FactorialError` class is an exception type, and objects of that type are used to signal an error.
- When the `factorial()` function detects an error, it *throws* an exception object using the `throw` keyword. In the example above, the function throws a default-constructed `FactorialError`. The standard-library `stoi()` function throws a `std::invalid_argument` object when given a string that does not represent an integer.
- The compiler arranges for the exception to propagate outward until it is handled by a *try/catch* block. In the example above, the first catch block is executed when a `std::invalid_argument` is thrown, while the second is run when a `FactorialError` is thrown.

When an exception is thrown, execution pauses and **can only resume at a catch block**. The code that is between the throw and the catch that handles the exception is entirely skipped. This is a good thing; if the rest of the code in `factorial()` were to run, the function would try to compute the factorial of a negative number, which would recurse indefinitely (or at least until either the program runs out of stack space, or `n` reaches the most negative `int`, at which point the subtraction `n - 1` would produce undefined behavior).

The result of running this program on different inputs is as follows:

```
$ ./main.exe 3
3! = 6
$ ./main.exe -1
Error: cannot compute factorial on negative number
$ ./main.exe hello
Error converting hello to an int: stoi: no conversion
```

The error message in the last example, specifically the part returned by `error.what()`, is implementation-dependent.

35.4.1 Exception Objects

C++ allows an object of any type to be thrown as an exception. However, a proper exception carries information about what caused the error, and using exceptions of different types allows a program to distinguish between different kinds of errors. More specifically, the example above illustrates that a program can perform different actions in response to each exception type.

There are several exceptions defined by the standard library, such as `invalid_argument` or `out_of_range`. The standard-library exceptions all derive from `std::exception`, and it is common for user-defined exceptions to do so as well:

```
class EmailError : public std::exception {
public:
  EmailError(const string &msg_in) : msg(msg_in) {}

  const char * what() const override {
    return msg.c_str();
  }

private:
  string msg;
};
```

A user-defined exception is a class type, so it can carry any information necessary to pass between the function that detects an error and the one that handles it. For `EmailError` above, the constructor takes a string and stores it as a member variable, allowing any message to be specified when creating an `EmailError` object:

```
throw EmailError("Error sending email to: " + address);
```

The `what()` member function is a virtual function defined by `std::exception`, and `EmailError` overrides it to return the message passed to the constructor. We can call it when catching an exception to obtain the message:

```
try {
  ...
} catch (const EmailError &error) {
  cout << error.what() << endl;
}
```

Exception types are often defined in an inheritance hierarchy, with a derived class representing a more specific kind of error. Deriving from `std::exception` is an example of this, but we can also define exception types that derive from `EmailError`:

```
class InvalidAddressError : public EmailError {
  ...
};

class SendFailedError : public EmailError {
  ...
};
```

We will shortly see how exception hierarchies interact with try/catch blocks.

35.4.2 Try/Catch Blocks

A try/catch block consists of a try block followed by one or more catch blocks:

```
try {
  int n = std::stoi(argv[1]);
  int result = factorial(n);
  cout << n << "! = " << result << endl;
} catch (const std::invalid_argument &error) {
  cout << "Error converting " << argv[1] << " to an int: "
       << error.what() << endl;
} catch (const FactorialError &error) {
  cout << "Error: cannot compute factorial on negative number"
       << endl;
}
```

The try/catch can only handle exceptions that occur within the try part of the block, including in functions called by code in the try. When a thrown exception propagates to a try block, the compiler checks the corresponding catch blocks in order to see if any of them can handle an exception of the type thrown. Execution is immediately transferred to the first applicable catch block:

- The catch parameter is initialized from the exception object. Declaring the parameter as a reference to const avoids making a copy.
- The body of the catch block is run.
- Assuming the catch block completes normally, execution proceeds past the entire try/catch. The remaining code in the try or in the other catch blocks is skipped.

In matching an exception object to a catch block, C++ takes into account subtype polymorphism – if the dynamic type of the object is derived from the type of a catch parameter, the catch is able handle that object, so the program uses that catch block for the exception. The following is an example:

```
void grade_submissions() {
  vector<string> students = load_roster();
  for (const string &name : students) {
    try {
      auto sub = load_submission(name);
      double result = grade(sub);
      email_student(name, result);
    } catch (const FileError &error) {
      cout << "Can't grade: " << name << endl;
    } catch (const EmailError &error) {
      cout << "Error emailing: " << name << endl;
    }
  }
}
```

The first catch block above handles objects of any type derived from `FileError`, while the second handles objects of any type derived from `EmailError`.

C++ also has a "catch-all" that can handle any exception type:

```
try {
  // some code here
} catch (...) {
```

(continues on next page)

(continued from previous page)

```
  cout << "Caught an unknown error" << endl;
}
```

The `...` as a catch parameter enables the catch block to handle any type of object. However, this should in general be avoided – instead, a particular try/catch should only catch the specific kinds of errors that it is able to recover from. For instance, consider the following implementation of the `load_roster()` function:

```
vector<string> load_roster() {
  try {
    csvstream csvin("280roster.csv");      // may throw an exception
    // Use the stream to load the roster...
  } catch (const csvstream_exception &e) {
    cout << e.what() << endl;
    // return ???
  }
}
```

The code uses the csvstream library to read a spreadsheet in CSV format. The library may throw a `csvstream_exception` object when opening the file, such as if the file doesn't exist. However, the `load_roster()` function is not the best place to handle this exception; as discussed previously, the appropriate error recovery depends on the application, and it is the caller of `load_roster()` that knows what to do in case of an error. Thus, `load_roster()` should avoid catching the exception, letting it propagate to its caller.

35.4.3 Exception Propagation

When an exception is thrown, if the current function is not within a try block, the exception propagates to the function's caller. Similarly, if the current function is within a try block but there is no associated catch block that can handle an object of the exception's type, the exception also propagates to the caller.

To handle an exception, the program immediately pauses execution, then looks for an exception handler as follows. The process starts at the statement that throws the exception:

- Determine if the current statement is within the try part of a try/catch block in the current function. If not, the function call is terminated, and the process repeats in the caller.
- If the execution is within a try, examine the catch blocks in order to find the first one that can handle an exception of the given type. If no matching catch is found, the function call is terminated, and the process repeats in the caller.
- If a matching catch block is found, the catch parameter is initialized with the exception object, and the body of the catch immediately runs. Assuming it completes successfully, execution proceeds past the entire try/catch.

As described above, if the code is not within a try block that can handle the exception object, execution returns immediately to the caller, and the program looks for a handler there. The exception propagates outward until a viable handler is found.

If the exception proceeds outward past `main()`, the default behavior is to terminate the program. This ensures that either the exception is handled, or the program crashes, informing the user that an error occurred.

35.4.4 Exception Examples

The following is a `DriveThru` class that keeps track of menu items and their prices:

```
class InvalidOrderException : public std::exception {};

class DriveThru {
public:
  // EFFECTS: Adds the given item to the menu.
  void add_item(const string &item, double price) {
    menu[item] = price;
  }

  // EFFECTS: Returns the price for the given item. If the item
  //          doesn't exist, throws an InvalidOrderException.
  double get_price(const string &item) const {
    auto it = menu.find(item);
    if (it != menu.end()) {
      return it->second;
    }
    throw InvalidOrderException();
  }

private:
  // A map from item names to corresponding prices
  map<string, double> menu;
};
```

The `get_price()` member function looks up the price of an item, returning it if it is on the menu. If not, it throws an exception of type `InvalidOrderException`. We use the `find()` member function of `map` to avoid inserting a value-initialized price into the map when an item does not exist.

The following is a program that reads order items from standard input, computing the total price:

```
int main() {
  DriveThru eats280;
  ... // add items to eats280

  double total = 0;
  string item;
  while (cin >> item && item != "done") {
    try {
      total += eats280.get_price(item);
    } catch (const InvalidOrderException &e) {
      cout << "Sorry, we don't have: " << item << endl;
    }
  }
  cout << "Your total cost is: " << total << endl;
}
```

The program ignores items that aren't on the menu, printing a message indicating that it is not available. In such a case, `get_price()` throws an `InvalidOrderException`. Since the throw statement is not within a try in `get_price()`, the exception propagates outward to `main()`. The call to `get_price()` **is** within a try, and there is an associated catch that matches the type of the exception. Execution proceeds to that catch, which prints an error message. Then execution continues after the full try/catch, ending an iteration of the loop and reading in a new item to start the next iteration

As another example, we write a subset of a program that loads and grades student submissions for an assignment. We use the following classes:

```
// An exception signifying an error when reading a file.
class FileError : std::exception {
  ...
};

// An exception signifying an error when sending an email.
class EmailError : std::exception {
  ...
};

// A student submission.
class Submission {
  ...
public:
  // EFFECTS: Grades this submission and returns the result.
  double grade();
};
```

`FileError` and `EmailError` are exceptions that the program will use. We saw the definition of `EmailError` previously, and `FileError` is similarly defined. A `Submission` object represents a student submission, and we elide the definition here.

The following functions perform individual tasks in the grading program:

```
// EFFECTS: Emails the given student the given grade. Throws
//          EmailError if sending the email fails.
void email_student(const string &name, double grade);

// EFFECTS: Loads the roster from 280roster.csv. Throws
//          csvstream_exception if the file cannot be read.
vector<string> load_roster() {
  csvstream csvin("280roster.csv"); // may throw an exception

  vector<string> roster;
  // Use the stream to load the roster...

  return roster;
}

// EFFECTS: Loads the submission for the given student. Throws
//          FileError if the submission cannot be loaded.
Submission load_submission(const string &name) {
  std::ifstream input(name);

  if (!input) {
    throw FileError();
  }

  return Submission(input);
}
```

All three functions throw exceptions when they are unable to perform their task. The `email_student()` func-

tion throws an `EmailError` if sending fails. The `load_roster()` function throws a `csvstream_exception` if reading the roster file fails; in actuality, the exception will be thrown by the `csvstream` constructor, but since `load_roster()` allows the exception to propagate outward, it documents that such an exception may be thrown. Finally, `load_submissions()` throws a `FileError` if a submission file fails to open.

The function that directs the grading process is as follows:

```
// EFFECTS: Loads and grades all submissions, sending email to each
//          student with their result. Throws csvstream_exception
//          if the roster cannot be loaded.
void grade_submissions() {
  vector<string> students = load_roster();
  for (const string &name : students) {
    try {
      auto sub = load_submission(name);
      double result = sub.grade();
      email_student(name, result);
    } catch (const FileError &e) {
      cout << "Can't grade: " << s << endl;
    } catch (const EmailError &e) {
      cout << e.what() << endl;
    }
  }
}
```

This function handles `FileError` and `EmailError` exceptions by just printing a message to standard out. The try/catch is within the for loop so that failure for one student does not prevent grading of other students. If a `csvstream_exception` is thrown by `load_roster()`, it gets propagated to the caller of `grade_submissions()`.

Lastly, we have the `main()` function:

```
int main() {
  try {
    grade_submissions();
    cout << "Grading done!" << endl;
  } catch (const csvstream_exception &e) {
    cout << "Grading failed! << endl;
    cout << e.what() << endl;
    return EXIT_FAILURE;
  }
}
```

This function handles a `csvstream_exception` by printing a message to standard out and then returning with a nonzero exit code, indicating a failure.

We consider a few more small examples to better understand how exceptions are propagated and handled. The examples use the following exception types:

```
class GoodbyeError {};

class HelloError {};

class Error {
public:
  Error(const string &s) : msg(s) {}
```

(continues on next page)

(continued from previous page)

```
  const string & get_msg() {
    return msg;
  }

private:
  string msg;
};
```

Objects of `GoodbyeError` will generally be thrown by a `goodbye()` function, while `HelloError` objects will be thrown by `hello()`.

The first example is as follows:

```
void goodbye() {
  cout << "goodbye called" << endl;
  throw GoodbyeError();
  cout << "goodbye returns" << endl;
}

void hello() {
  cout << "hello called" << endl;
  goodbye();
  cout << "hello returns" << endl;
}

int main() {
  try {
    hello();
    cout << "done" << endl;
  } catch (const HelloError &he) {
    cout << "caught hello" << endl;
  } catch (const GoodbyeError &ge) {
    cout << "caught goodbye" << endl;
  }
  cout << "main returns" << endl;
}
```

In this example, `hello()` is called from within a try in `main()`. So if an exception is thrown and propagates to `main()`, the associated catch blocks will attempt to handle the exception. Within `hello()`, the message `hello called` is printed, followed by a call to `goodbye()`. The latter prints out `goodbye called` and then throws a `GoodbyeError` object. Execution immediately pauses, and the program checks if it is within a try in `goodbye()`. It is not, so it then checks the caller to see if it is currently in a try there. There isn't one in `hello()`, so the program then checks for a try in `main()`. The execution state is indeed within a try in `main()`, so the program checks the catch blocks, in order, to see if there is one that can handle a `GoodbyeError`. The second catch can do so, and its code is run, printing `caught goodbye`. Execution then proceeds past the try/catch, so the print of `main returns` executes.

Observe that the remaining code in `goodbye()`, `hello()`, and the try block in `main()` were skipped when handling the exception. The full output is as follows:

```
hello called
goodbye called
caught goodbye
main returns
```

Consider another example:

```
void goodbye() {
  cout << "goodbye called" << endl;
  throw GoodbyeError();
  cout << "goodbye returns" << endl;
}

void hello() {
  cout << "hello called" << endl;
  try {
    goodbye();
  } catch (const GoodbyeError &ge) {
    throw HelloError();
  } catch (const HelloError &he) {
    cout << "caught hello" << endl;
  }
  cout << "hello returns" << endl;
}

int main() {
  try {
    hello();
    cout << "done" << endl;
  } catch (const HelloError &he) {
    cout << "caught hello" << endl;
  } catch (const GoodbyeError &ge) {
    cout << "caught goodbye" << endl;
  }
  cout << "main returns" << endl;
}
```

This is the same as the first example, except now the call to `goodbye()` is within a try in `hello()`. When the `GoodbyeError` object is thrown, the program determines that it is not within a try in `goodbye()`, so it checks whether it is in a try in `hello()`. It is, so the program checks whether there is a catch block that can handle a `GoodbyeError`. The first catch block can do so, so its code is run. This throws a `HelloError`, so the program checks whether execution is within a try in `hello()`. It is not – execution is within a catch block, not within a try. So the program proceeds to the caller, checking whether execution is in a try in `main()`. The call to `hello()` is indeed within a try, so the program examines the catch blocks. The first one handles a `HelloError`, so its body is executed, printing `caught hello`. Then execution proceeds past the try/catch to the print of `main returns`.

Observe that a try/catch can only handle exceptions that are thrown within the try block; it does not deal with exceptions that are thrown from one of its catch blocks, so an outer try/catch must handle such exceptions instead.

```
hello called
goodbye called
caught hello
main returns
```

The following example uses the `Error` class defined above, which has a constructor that takes a string:

```
void goodbye() {
  cout << "goodbye called" << endl;
  throw Error("bye");
```

(continues on next page)

(continued from previous page)

```
  cout << "goodbye returns" << endl;
}

void hello() {
  cout << "hello called" << endl;
  try {
    goodbye();
  } catch (const Error &e) {
    throw Error("hey");
  }
  cout << "hello returns" << endl;
}

int main() {
  try {
    hello();
    cout << "done" << endl;
  } catch (const Error &e) {
    cout << e.get_msg() << endl;
  } catch (...) {
    cout << "unknown error" << endl;
  }
  cout << "main returns" << endl;
}
```

The throw statement in `goodbye()` throws an `Error` object constructed with the string `"bye"`. There is no try in `goodbye()`, so the program checks whether it is currently within a try in `hello()`. Execution is indeed within the try there, and the catch block can handle the `Error` object. The catch block throws a different `Error` object, initialized with the string `"hey"`. As with the previous example, this throw statement is not within a try in `hello()`, so the program checks for a try in `main()`. Both catch blocks can handle the `Error` object; the second is a catch-all that can handle any exception. However, the program considers the catch blocks in order, so it is the first one that runs. Its body retrieves the message from the `Error` object, printing out `hey`. Then execution proceeds past the try/catch. The full output is below:

```
hello called
goodbye called
hey
main returns
```

One more example is the following:

```
void goodbye() {
  cout << "goodbye called" << endl;
  throw GoodbyeError();
  cout << "goodbye returns" << endl;
}

void hello() {
  cout << "hello called" << endl;
  try {
    goodbye();
  } catch (const Error &e) {
```

(continues on next page)

(continued from previous page)

```
    throw Error("hey");
  }
  cout << "hello returns" << endl;
}

int main() {
  try {
    hello();
    cout << "done" << endl;
  } catch (const Error &e) {
    cout << e.get_msg();
    cout << endl;
  } catch (...) {
    cout << "unknown error" << endl;
  }
  cout << "main returns" << endl;
}
```

Here, the `goodbye()` function throws a `GoodbyeError`, and the throw is not within a try in `goodbye()`, so the program looks for a try in `hello()`. Execution is within a try there, so the program examines the catch blocks to see whether one can handle a `GoodbyeError`. The lone catch block cannot, so the program propagates the exception to the caller and looks for a try in `main()`. The code is within a try in `main()`, so the program examines the catch blocks in order. The first cannot handle a `GoodbyeError`, but the second can, so the latter runs and prints `unknown error`. Execution continues after the try/catch, printing `main returns`. The full result is the following:

```
hello called
goodbye called
unknown error
main returns
```

35.5 Error Handling vs. Undefined Behavior

The error-detection mechanisms we discussed provide well-defined behavior in case of an error. As such, a function that uses one of these mechanisms should document it, describing when and what kind of error can be generated:

```
// EFFECTS: Computes the factorial of the given number. Returns 0 if
//          n is negative.
int factorial(int n);

// EFFECTS: Emails the given student the given grade. Throws
//          EmailError if sending the email fails.
void email_student(const string &name, double grade);
```

These functions do not have REQUIRES clauses that restrict the input; violating a REQUIRES clause results in undefined behavior, whereas these function produce well-defined behavior for erroneous input.

On the other hand, when code does produce undefined behavior, it cannot be detected through any of the error-handling mechanisms above. For example, dereferencing a null or uninitialized pointer does not necessarily throw an exception, set a global error code, or provide any other form of error detection. It is the programmer's responsibility to avoid undefined behavior, and there are no constraints on what a C++ implementation can do if a program results in undefined behavior.

Part VI

Supplemental Material

CHAPTER

THIRTYSIX

INTRODUCTION TO C++

This courses uses C++ as the language in which we explore fundamental concepts in programming. Here, we provide a basic overview of C++ for students who are familiar with other languages such as Python or Java.

36.1 A Simple Program

Let's start with a simple program that prints `Hello world!` to the screen:

```
#include <iostream>

int main() {
  std::cout << "Hello world!" << std::endl;
  return 0;
}
```

The entry point of a C++ program is the `main()` function, which is a top-level (global) function that has a signature of the form `int main()` or `int main(int argc, char *argv[])`. A *function signature* consists of a return type, the name of the function, and a list of parameters. The *parameters* are what the function takes as input, and the *return type* is the kind of value that the function returns. For `main()`, it can either have no parameters, or it can have parameters that allow it to take *command-line arguments*, which are specified when the program is run. For simplicity, we start with a `main()` that does not have parameters.

Following the signature of the function, we have the *function body*, which is the code that gets executed when the function is called. The body itself is comprised of *statements*, which are executed in order from top to bottom. In our program above, the body of `main()` has two statements: one that prints to the screen, and a second that exits the function with a return value of 0.

Examining the first statement more closely, we see that it is composed of expressions and operators. An *expression* is a fragment of code that *evaluates* to some value. The simplest expressions are *literals*, which hardcode a specific value in the program. For instance, `"Hello world!"` is a string literal that denotes the sequence of characters `H`, `e`, and so on. A *name* is also an expression, and what it evaluates to depends on what the name is bound to in the environment in which the name is evaluated. In the program above, `std::cout` is bound to an object representing the *standard output stream (stdout)*, which allows printing to the *console* (also referred to as *shell* or *terminal*) in which the program is run. Similarly, `std::endl` is bound to an object that causes a newline to be printed when inserted to an output stream. Finally, these expressions are connected via the binary << operator; in this context, we refer to this as the *stream-insertion operator*. The code is inserting the string `"Hello world!"` into the standard output stream, followed by inserting `std::endl` to obtain a newline.

Both `std::cout` and `std::endl` are defined in the `iostream` *library header*. The first line of the program (`#include <iostream>`) instructs the C++ compiler to *include* the contents of the `iostream` library header, which makes `std::cout` and `std::endl` available for us to use.

A name such as `std::cout` is called a *qualified name* – it consists of the `std` qualifier, followed by the `::` *scope-resolution operator*, followed by the `cout` name. The `std` qualifier refers to the `std` *namespace*, which is a scope in which most standard-library entities are defined. Thus, the qualified name `std::cout` refers to the `cout` that is defined within the `std` namespace, as opposed to some other `cout` that might be defined in a different scope. Often, we place a directive in our program to be able to use an entity such as `cout` without qualifying it with `std::`. The following does so for both `cout` and `endl`:

```
using std::cout;   // we can now use cout without qualification
using std::endl;   // we can now use endl without qualification
```

The following does so for every entity defined in the `std` namespace:

```
using namespace std; // we can now use anything in std without qualification
```

Use this with caution, however – there are lots of names defined in the `std` namespace, so this makes it much more likely for our names to conflict with those from the standard library. (In particular, a `using namespace` directive should never be used in a header file.)

The last statement in our program is `return 0;`. This returns the value 0 from the `main()` function, which conventionally indicates that the program completed successfully. We would use a nonzero value instead to indicate an error if something went wrong.

In the specific case of returning from `main()`, a return statement is actually optional. (This is not the case for returning from other functions, unless they have a `void` return type.) If we don't have a return statement in `main()`, the compiler implicitly adds a `return 0;` for us. Thus, the program above could be written equivalently as:

```
#include <iostream>

using std::cout;   // we can now use cout without qualification
using std::endl;   // we can now use endl without qualification

int main() {
  cout << "Hello world!" << endl;
}  // implicit return 0; added by the compiler
```

Now that we have examined all the pieces of our simple program, let's compile and run it in a shell. Assuming the code is in the file `hello.cpp`, we can compile and run it as follows:

```
$ g++ -std=c++17 -o hello.exe hello.cpp
$ ./hello.exe
Hello world!
```

Here, we use the `$` symbol to denote the *shell prompt*, which is displayed by the shell to indicate that it is waiting for our commands. (On most machines, the prompt is more complicated, but we simplify it to just a dollar sign.) We then type the compilation command `g++ -std=c++17 -o hello.exe hello.cpp`, which invokes the `g++` compiler. The `-std=c++17` tells the compiler to use the C++17 version of the language. The `-o hello.exe` tells the compiler to use `hello.exe` as the name of the resulting executable file. Finally, `hello.cpp` is the filename of our C++ program. When the compiler has finished, we type `./hello.exe` to run the resulting executable, and we see the message `Hello world!` printed to the screen.

In this course, we encourage using both the shell as well as an *integrated development environment (IDE)*. Refer to this tutorial for how to set up and become familiar with using a shell and IDE on your machine.

36.2 Static Typing

We saw above that the signature of the `main()` function includes the return type, which is `int` in the case of `main()`. The reason the return type is included is that C++ is a *statically typed language*, meaning that every value's type must be known at compile time. To do so, the compiler generally requires us to specify the type of a variable, as well as the parameter types and return type of a function. For instance, the following introduces a local variable `x` with type `int` and initial value 3:

```
int x = 3;
```

The syntax for defining a variable begins with the type of the variable, followed by the name, followed by an optional initialization. (If no initialization is provided, the variable undergoes *default initialization*. For primitive types such as `int`, that means the initial value of the variable is undefined.) The `int` type is a *primitive type*, which is a category consisting of the simplest types provided by a language. Common primitive types in C++ include:

- `int`: signed (positive, negative, or zero) integer values in some finite range, typically $[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$ on most machines
- `std::size_t`: unsigned (positive or zero only) integer values in some finite range, typically $[0, 2^{64} - 1]$
- `double`: *double-precision* floating-point values, typically using a 64-bit representation
- `bool`: a boolean value, either true or false
- `char`: a character value, generally representing at least the 128 values in the ASCII standard
- `void`: used as the return type of a function to indicate that it does not return a value

As another example, let's define a `square()` function that computes the square of a number. We'll write it to take a `double` value as input and return a `double` value as the result:

```
double square(double x) {
  double result = x * x;
  return result;
}
```

As with `main()`, the signature for `square()` starts with the return type, which is `double`. Then we have the name of the function, followed by the parameter list. We take in a single `double` value as input, so we have a single parameter, which we are calling `x`. In the body of the function, we define a new variable `result` that is initialized with the value of `x * x`, and we then return the value of the variable.

Since we only use the variable once, we can simplify our function by eliminating it and using the expression `x * x` directly where we need it:

```
double square(double x) {
  return x * x;
}
```

Since an expression evaluates to a value, it has a type corresponding to that value. In the case of `x * x`, its type is `double` since it is the product of two `double` values. Thus, the return statement returns a `double` value, matching the declared return type of the function.

We can now invoke `square()` as follows:

```
#include <iostream>

double square(double x) {
  return x * x;
```

(continues on next page)

(continued from previous page)

```
}

int main() {
  std::cout << square(3.14) << std::endl;
  double x = square(4);
  std::cout << square(x) << std::endl;
}
```

The `square()` function must be declared above where we use it in `main()` – in C++, the scope of a global function or variable is from the point of its declaration until the end of the file. (The *scope* of an entity is the region of code where it may be used.) We can then invoke the function in `main()`.

Observe that in the definition

```
double x = square(4);
```

we invoke `square()` on the literal `4`, which actually has type `int` rather than the declared `double` type of the parameter of the `square()` function. In most cases, C++ automatically converts between `int` and `double`, so that we can use a value of one of these types where the other is expected. On the other hand, a call such as `square(std::cout)` as erroneous, since there is no conversion between the type of `std::cout` (which happens to be `std::ostream`) and the required `double` type.

36.3 Compound Data

In addition to primitive types, C++ has data types that represent more complex objects. For instance, the `std::ostream` type (defined in the `iostream` header) represents an output stream, and it is the type of `std::cout`. The `std::string` type is another common data type that represents string objects, and it is defined in the `string` header:

```
#include <iostream>
#include <string>

int main() {
  std::string s = "Hello world!";
  std::cout << s << std::endl;
}
```

Often, we want to keep track of a collection of objects of the same type, such as an arbitrary number of integer values. We can use a `std::vector` to do so, which is an example of a *template* that is parameterized by some other type. For instance, `std::vector<int>` is a collection of `int` values, while `std::vector<std::string>` is a collection of `std::string` values. The following demonstrates how to use a vector:

```
#include <iostream>
#include <vector>

int main() {
  std::vector<int> scores = { 84, 91, 77, 95, 83 };
  for (std::size_t i = 0; i < scores.size(); ++i) {
    std::cout << "Score " << i << " = " << scores[i] << std::endl;
  }
}
```

In this code, we define a `scores` variable of type `std::vector<int>`, and we provide an initial set of values as a comma-separated list enclosed by curly braces. We then iterate over the indices of the vector, which start at 0 and end at one less than the size of the vector, which we obtain via the expression `scores.size()`. We access an element of the vector using square brackets, with the vector object on the left-hand side and the index between the brackets (`scores[i]`). Using this syntax, we have to be careful not to go past the end of the vector, which would result in *undefined behavior*, meaning that anything could happen – e.g. crashing the program, overwriting some other piece of data, stealing your files, or even nothing at all. Alternatively, we can do `scores.at(i)`, which checks whether the index is in range and guarantees an error if it is not – the program crashes, but we get some useful information, and it definitely won't steal our files.

We can also add elements to a vector after it has been created:

```
scores.push_back(42);
```

This appends the element 42 at the end of the vector. Similarly, the expression `scores.pop_back()` removes the element at the end of the `scores` vector (assuming there is at least one element in the vector; otherwise we get the dreaded undefined behavior).

Vectors are useful for ordered collections of data that all have the same type. However, sometimes we want to keep track of multiple pieces of data that have individual meanings, and that might even have different types. For instance, a complex number has a real part and an imaginary part; while both might be represented as `double`s, we want to keep track of which part is which through a name rather than maintaining an ordering between them. A *struct*[1] gives us this ability to introduce a compound object, comprised of multiple pieces each with individual names. The following is an example of defining a new `Complex` type:

```
struct Complex {
  double real;
  double imaginary;
};
```

We start with the `struct` keyword, followed by the name of the type we are defining, followed by an open curly brace. We then specify the components of the struct, using syntax similar to variable declarations. These components are called *member variables* in C++[2]. The struct definition is terminated by a closing curly brace and a semicolon. Once we have this definition, we can write a function to print out a `Complex` value:

```
void Complex_print(Complex number) {
  std::cout << number.real;
  if (number.imaginary >= 0) {
    std::cout << '+';
  }
  std::cout << number.imaginary << std::endl;
}
```

As shown above, we can access a member variable using the dot operator. The expression `number.real` accesses the `real` component of the `number` object, and `number.imaginary` refers to the `imaginary` component. The function prints the + character to separate the two components if the imaginary part is nonnegative – otherwise, the imaginary part would have a minus sign, so we wouldn't want a + between the two components.

We can create objects of `Complex` type and print them as follows:

```
Complex c1 = { 3.14, -1.7 };
Complex c2;
c2.real = 2.72;
```

(continues on next page)

[1] In other languages the term *class* is used instead. In C++, the `struct` and `class` keywords are closely related, and we will *see the details later*.
[2] Other languages uses the terms *fields* or *attributes* to refer to these components.

(continued from previous page)

```
c2.imaginary = -4;
Complex_print(c1);
Complex_print(c2);
```

As the code demonstrates, we can use curly braces to initialize a `Complex` variable, which initializes the components in order: `c1.real` gets initialized to `3.14`, and `c1.imaginary` gets initialized to `-1.7`. Alternatively, we can rely on default initialization as is the case for `c2`. However, this ends up initializing the two components to undefined values, so we need to replace their values before we can proceed to use them.

As another example, we define a struct to keep track of a person's exam score. We need to keep track of the person's name, as well as the score value itself:

```
struct Grade {
  std::string name;
  int score;
};
```

We can now use the `Grade` type as follows:

```
Grade g1 = { "Sofia", 99 };
Grade g2;
g2.name = "Amir";
g2.score = 23;
std::cout << g1.name << " earned a " << g1.score << std::endl;
std::cout << g2.name << " earned a " << g2.score << std::endl;
```

36.4 Value Semantics

One of the distinguishing features of a programming language is the relationship between variables and *objects*, which are pieces of data in memory. In some languages, a variable is directly associated with an object, so that using the variable always accesses the same object (as long as the variable is in scope). We say that the language has *value semantics* if this is the case. In other languages, a variable is an indirect reference to an object, and the variable may be modified to refer to a different object. This scheme is known as *reference semantics*.

C++ has value semantics, unlike other languages such as Java[3] or Python that primarily have reference semantics. We can illustrate the difference between these semantic choices using the `Complex` type we defined previously. Consider the following code:

```
#include <iostream>

struct Complex {
  double real;
  double imaginary;
};

void Complex_print(Complex number) {
  std::cout << number.real;
  if (number.imaginary >= 0) {
    std::cout << '+';
  }
```

(continues on next page)

[3] Java actually has value semantics for primitive types and reference semantics for class types.

(continued from previous page)

```
  std::cout << number.imaginary << std::endl;
}

int main() {
  Complex c1 = { 3.14, -1.7 };
  Complex c2 = c1;
  c2.real = 2.72;
  Complex_print(c1);
  Complex_print(c2);
}
```

In this code, we define a variable `c1` of type `Complex` and give it an initial value. We then copy it to a `c2` variable. If we modify `c2`, the change does not affect `c1`, since the two variables each have their own object with which they are associated. In memory, this looks something like the picture in Figure 36.1.

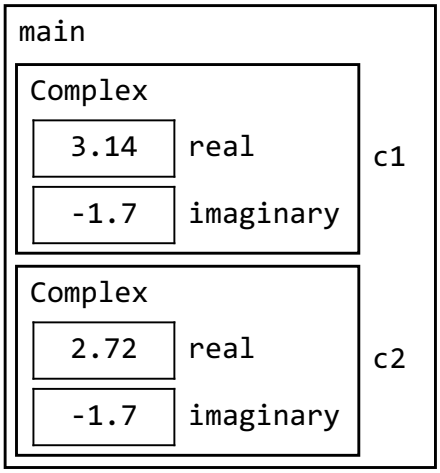

Figure 36.1: Two variables corresponding to separate `Complex` objects in memory.

Compiling and running the program produces:

```
$ g++ -std=c++17 -o complex.exe complex.cpp
$ ./complex.exe
3.14-1.7
2.72-1.7
```

We see that as expected, the modification to `c2` does not affect `c1`. Compare this to a similar Python program:

```
class Complex:
    def __init__(self, real, imaginary):
        self.real = real
        self.imaginary = imaginary

def Complex_print(number):
    print(number.real, end='')
    if number.imaginary >= 0:
        print('+', end='')
    print(number.imaginary)
```

(continues on next page)

(continued from previous page)

```
c1 = Complex(3.14, -1.7)
c2 = c1
c2.real = 2.72
Complex_print(c1)
Complex_print(c2)
```

In memory, the variables look like the diagram in Figure 36.2.

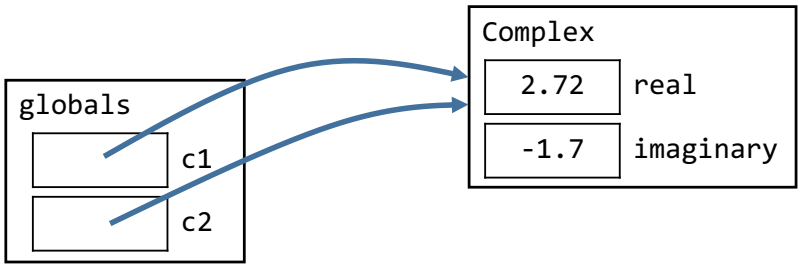

Figure 36.2: Two variables indirectly referring to the same `Complex` object.

Running the program produces the following:

```
$ python3 complex.py
2.72-1.7
2.72-1.7
```

This confirms that `c1` and `c2` refer to the same object, since the modification to `c2.real` also affected `c1`.

We can also observe C++'s value semantics when we call a function. Suppose we want a function that modifies a `Complex` object to be its conjugate, flipping the sign of the imaginary component of the complex number. The following is an attempt to write this function:

```
void Complex_conjugate(Complex number) {
  number.imaginary = -number.imaginary;
}
```

Suppose we insert the following statement prior to the print statements in our program above:

```
Complex_conjugate(c1);
```

Compiling and running the code, we get:

```
$ g++ -std=c++17 -o complex.exe complex.cpp
$ ./complex.exe
3.14-1.7
2.72-1.7
```

Nothing changes! This is because the `number` parameter of the `Complex_conjugate()` function is its own object, and it receives a copy of `c1`'s value. The diagram in Figure 36.3 illustrates this in memory at the end of the `Complex_conjugate()` call.

The copy within the `Complex_conjugate()` function does have its `imaginary` component modified, but not the object within the `main()` function. This behavior is called *pass by value*, since the value of `c1` (the *argument* of the function call) is copied into the `number` parameter object.

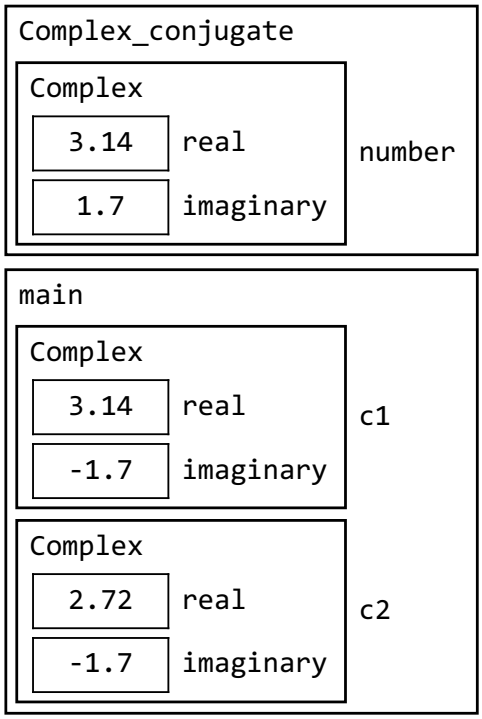

Figure 36.3: A function that received a copy of an argument object.

C++ also supports another mechanism for passing arguments: *pass by reference*. In this scheme, the parameter is an *alias* of the object passed in as an argument rather than being its own object. We specify that a parameter should use pass by reference by placing the ampersand (&) symbol to the left of the parameter name:

```
void Complex_conjugate(Complex &number) {
  number.imaginary = -number.imaginary;
}
```

Compiling and running the code with this modified definition, gives us:

```
$ g++ -std=c++17 -o complex.exe complex.cpp
$ ./complex.exe
3.14+1.7
2.72-1.7
```

We see that `c1` has indeed been conjugated. The memory diagram in Figure 36.4 depicts what the code looks like in memory at the end of the call to `Complex_conjugate()`. In the figure, we denote an alias with a dashed line, and we do not include a box next to the `number` name to reflect the fact that it is not associated with a new object.

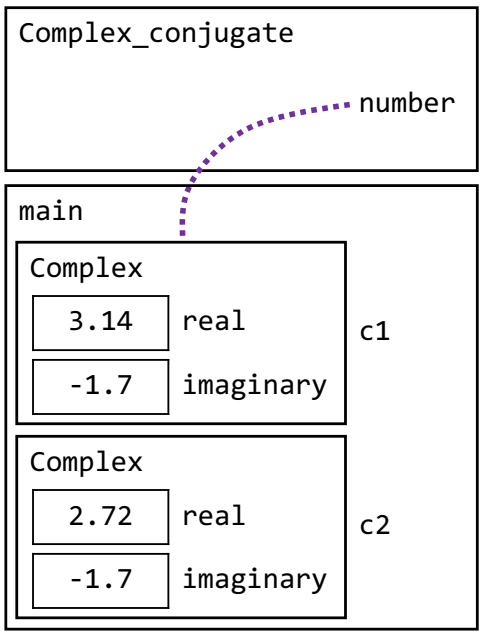

Figure 36.4: A function that received an alias to an existing argument object.

36.5 Example: Stickman

Having seen the basics of C++, let us proceed to write a larger C++ program that implements a *stickman* game (also called hangman), which involves guessing the letters in a word or phrase. First, let's sketch out how we want the game to run. If the puzzle is the word `hello`, an unsuccessful game may play out like the following:

```
$ ./stickman.exe hello



     _ _ _ _ _

Enter a lowercase letter to guess:
a

 o


     _ _ _ _ _

Enter a lowercase letter to guess:
b

 o
 |
     _ _ _ _ _

Enter a lowercase letter to guess:
c
```

(continues on next page)

(continued from previous page)

```
 o
/|
     _ _ _ _ _

Enter a lowercase letter to guess:
d

 o
/|\
     _ _ _ _ _

Enter a lowercase letter to guess:
e

 o
/|\
     _ e _ _ _

Enter a lowercase letter to guess:
f

 o
/|\
/    _ e _ _ _

Enter a lowercase letter to guess:
g

 o
/|\
/ \  _ e _ _ _

Better luck next time!
```

We want to print out underscores corresponding to the letters in the puzzle, with a space between each of them. We prompt the player for a guess, reading from the *standard input stream* corresponding to input from the console. If the player guesses a letter that is not in the puzzle, we reveal a portion of a stick figure. If the player guesses a letter that is in the puzzle, we reveal all occurrences of that letter and do not add to the stick figure. If the player guesses incorrectly six times, the full stick figure is revealed, and the player loses.

The following is a successful run of the game (loosely following the frequency distribution of English letters):

```
$ ./stickman.exe hello




     _ _ _ _ _

Enter a lowercase letter to guess:
e
```

(continues on next page)

(continued from previous page)

```
    _ e _ _ _

Enter a lowercase letter to guess:
t

 o

    _ e _ _ _

Enter a lowercase letter to guess:
s

 o
 |
    _ e _ _ _

Enter a lowercase letter to guess:
a

 o
/|
    _ e _ _ _

Enter a lowercase letter to guess:
i

 o
/|\
    _ e _ _ _

Enter a lowercase letter to guess:
o

 o
/|\
    _ e _ _ o

Enter a lowercase letter to guess:
h

 o
/|\
    h e _ _ o

Enter a lowercase letter to guess:
r

 o
/|\
/   h e _ _ o

Enter a lowercase letter to guess:
```

(continues on next page)

(continued from previous page)

```
l

 o
/|\
/    h e l l o

Congratulations!
```

36.5.1 Program Constants

Now that we have a general sense of how the program should behave, we can go ahead and start on our design for its structure. First, let's define some constants corresponding to the stick figure. The full figure is as follows:

```
 o
/|\
/ \
```

This spans three lines. Unfortunately, a string literal in C++ cannot directly include a newline (line break) character. However, we can use the *escape sequence* `\n` to tell the compiler we want a newline character. For instance, the following prints `Hello` on one line, followed by `world!` on the next line[4]:

```
std::cout << "Hello\nworld!" << std::endl;
```

Escape sequences in general start with a backslash in C++. If we actually want a backslash character itself, it too needs to be escaped: `\\`. Thus, the stick figure can be represented with the following string:

```
" o\n/|\\\n/ \\"
```

The string starts with a space to center the head of the stick figure, then the newline escape sequence `\n` to end the line, then the right arm and torso, followed by the `\\` escape sequence for a backslash to represent the left arm, followed by another `\n` newline, then the right leg, a space, and finally the escaped left leg.

We can now work our way backwards to obtain partially revealed stick figures. There are two things we need to ensure so that the figures line up in the output:

- Each figure has exactly two newline characters.
- Each figure has exactly three characters on the third line.

We can use a `std::vector<std::string>` to store the figures in order:

```
const std::vector<std::string> STICK_FIGURES = {
  "\n\n   ",
  " o\n\n   ",
  " o\n |\n   ",
  " o\n/|\n   ",
  " o\n/|\\\n   ",
  " o\n/|\\\n/  ",
  " o\n/|\\\n/ \\"
};
```

[4] Note that `std::endl` is **not** the same as the newline character. It has the effect of printing the newline character to the target stream, but it also *flushes* the stream, which immediately transfers the inserted data into the underlying output. In the absence of a flush, the stream is allowed to buffer the data in memory, which generally allows for better performance than writing characters immediately.

We prefix the type with the `const` qualifier to denote that this is a constant, and the compiler will prevent us from modifying it. By convention, we name the constant using all capital letters, with underscores between words. We'll also use a constant for the maximum number of wrong guesses:

```
const int MAX_MISSES = 6;
```

36.5.2 The `Game` Struct

Next, we consider what data we need to represent the state of a game. We have to keep track of both the answer to the puzzle, as well as what pieces have been guessed by the player and what pieces remain. We also need to track the number of missed guesses. We will additionally keep a count of the number of letters remaining – this isn't strictly necessary, as we can recompute it when we need it, but it will make our job easier to keep track of it separately.

Before we define a struct, we also need to determine what underlying data types are appropriate to represent each member variable. The counts can be represented by the `int` type, and the answer by the `std::string` type. We can also represent what the player has and hasn't guessed with a `std::string` – letters that have not been guessed will be replaced with an underscore, while those that have been revealed will have the same values as in the answer. We can now define the struct:

```
// A struct to represent the state of a game of stickman.
struct Game {
  // The number of wrong guesses made so far.
  int miss_count;

  // The number of remaining unguessed positions in the puzzle.
  int remaining_letters;

  // The current state of the puzzle, with underscores representing
  // unguessed letters.
  std::string puzzle;

  // The answer to the puzzle.
  std::string answer;
};
```

Here, we have documented each component with a comment above it that describes its purpose.

36.5.3 Task Decomposition

Let's now think about the functions we need in our program. We should break down each discrete task in the program to its own function, so that each function has a small job, and so we can test each piece individually. The following are some tasks that our program needs to do:

- Construct a `Game` object from a `std::string` answer, with the appropriate initial values for each member variable.
- Print the a `Game` object, showing the player the current state of the game.
- Obtain a letter guess from the player, checking whether or not the guess is a valid letter.
- Update the `Game` based on a guessed letter.
- Perform the top-level game loop until the game ends.

Let's write function signatures corresponding to each of these tasks. We will write them in the form of a *declaration*[5], which excludes the body of a function, replacing it with a semicolon.

- The function that constructs a `Game` takes a `std::string` as an argument and returns a `Game`:

```
Game make_game(std::string answer);
```

- The function that prints a `Game` takes a `Game` object and does not return anything.

```
void print_game(Game game);
```

  We actually don't need a copy of the `Game` object, so we can specify pass by reference instead to obtain an alias to the existing object:

```
void print_game(Game &game);
```

- The function that obtains a guess from the player doesn't take any arguments, and it returns a character value:

```
char get_guess();
```

- The function that updates the game must take the `Game` object via pass by reference, so that we don't get a copy of the `Game`. It also takes in the guessed letter:

```
void update_game(Game &game, char guess);
```

- The top-level function takes the answer as a string and does not return anything:

```
void play_game(std::string answer);
```

36.5.4 Implementation

We can now proceed to implement these functions, starting with `make_game()`. The function needs to define a `Game` object and set its member variables:

```
Game game = { 0, 0, answer, answer };
```

We've initialized the `puzzle` member variable to be a copy of the `answer` parameter to start, but we need to replace each letter with an underscore. Let's assume that we only hide lowercase letters, and that other characters are shown to the player without needing to be guessed. This allows our puzzle to contain spaces and other punctuation. We can write a separate function to determine whether or not a character is a lowercase letter:

```
bool is_lowercase(char value) {
  return value >= 'a' && value <= 'z';
}
```

This logic takes advantage of the fact that in the ASCII standard (and most other character standards), the lowercase letters are adjacent and in order. Thus, a lowercase letter must be both greater than or equal to the character `'a'` and no more than the character `'z'`.

We can now loop over the `puzzle` member variable, replacing each lowercase character with an underscore:

[5] Technically, a definition is also a declaration. A declaration that is not also a definition is more precisely called an *incomplete* or *forward* declaration.

```
for (std::size_t i = 0; i < game.puzzle.size(); ++i) {
  if (is_lowercase(game.puzzle[i])) {
    game.puzzle[i] = '_';
    ++game.remaining_letters;
  }
}
```

Iterating over a string works the same way as iterating over a vector. We use a `for` loop from 0 up to the size of the string, and we use square brackets to index into the string to obtain the character at that position. Unlike in some other languages, C++ strings are *mutable*, meaning that they can be modified, and we do so here by replacing each lowercase letter with an underscore character. We also increment the number of remaining letters each time we encounter a lowercase letter.

Putting this all together, the following defines our `make_game()` function:

```
Game make_game(std::string answer) {
  Game game = { 0, 0, answer, answer };
  for (std::size_t i = 0; i < game.puzzle.size(); ++i) {
    if (is_lowercase(game.puzzle[i])) {
      game.puzzle[i] = '_';
      ++game.remaining_letters;
    }
  }
  return game;
}
```

To print a game, we print out the stick figure corresponding to the current number of incorrect guesses:

```
std::cout << STICK_FIGURES[game.miss_count] << " ";
```

We add a bit of space between the figure and the puzzle. We then print out each of the characters in the puzzle, separated by spaces:

```
for (std::size_t i = 0; i < game.puzzle.size(); ++i) {
  std::cout << " " << game.puzzle[i];
}
```

In addition, we add extra newlines to visually separate the turns from each other. The full function definition is as follows:

```
void print_game(Game &game) {
  std::cout << endl;
  std::cout << STICK_FIGURES[game.miss_count] << " ";
  for (std::size_t i = 0; i < game.puzzle.size(); ++i) {
    std::cout << " " << game.puzzle[i];
  }
  std::cout << "\n" << endl;
}
```

Next, we implement the `get_guess()` function to obtain a guess from the player. At a high level, this function needs to do the following:

- Obtain a string from the standard input stream.
- Check whether the string is a valid guess – it must be a single letter, and the letter must be lowercase.

- If the guess is invalid, repeat the process.
- If the guess is valid, return the guessed letter.

There's one more case we need to handle – what happens if we reach the end of the stream, when no more input is available? (On most systems, a user can manually end the standard input stream with the Ctrl-d keyboard input. We will also see later that input can be *redirected from a file*, in which case the input stream ends when it reaches the end of the file.) In such a case, we will return immediately with an error value that is different from any valid guess:

```
const char ERROR_CHAR = '\0';   // special "null" character
```

The following is a definition of `get_guess()`:

```
char get_guess() {
  std::cout << "Enter a lowercase letter to guess:" << std::endl;
  std::string input;
  while (std::cin >> input) {
    if (input.size() != 1) {
      cout << "Error: guess must be exactly one letter" << std::endl;
    } else if (!is_lowercase(input[0])) {
      cout << "Error: guess must be between a and z" << std::endl;
    } else {
      return input[0];
    }
  }
  return ERROR_CHAR;
}
```

The `std::cin` object represents the standard input stream, similar to how `std::cout` is the standard output stream. We *extract* from an input stream using the >> operator, which is called the *stream-extraction operator* in this context. To extract a string, we use a `std::string` object as the right-hand side of the operator. It is common practice to perform this extraction in the condition of a loop. If the extraction succeeds, the condition has a true value, and the body of the loop runs. If the extraction fails (e.g. in the case of the end of the stream), the loop condition is false, and the loop exits. In this function, we return `ERROR_CHAR` when the extraction fails.

The function starts by prompting the player to enter a guess. It then reads input in a loop, checking whether or not the input is a valid guess. If the guess is invalid, a message describing the problem is printed, and the loop moves on to the next iteration, reading another input. If the guess is valid, the lone character in the input string is returned.

We now go ahead and implement the `update_game()` function. The function needs to iterate over the characters in the answer to check if any are the same as the guess. If so, we additionally need to check whether the letter has already been guessed. If it has not been guessed, the corresponding position in the `puzzle` string has an underscore. In this case, we replace the underscore with the actual letter and decrement the count of remaining letters:

```
for (std::size_t i = 0; i < game.answer.size(); ++i) {
  if (game.answer[i] == guess && game.puzzle[i] == '_') {
    game.puzzle[i] = guess;  // replace the _ with the actual letter
    --game.remaining_letters;
  }
}
```

The function also needs to update the `miss_count` member variable depending on whether the guess was a correct one or not. We can't know this until we have traversed the entire puzzle, and we use a separate boolean to track this. The full function definition below demonstrates this logic:

```
void update_game(Game &game, char guess) {
  bool correct_guess = false;
  for (std::size_t i = 0; i < game.answer.size(); ++i) {
    if (game.answer[i] == guess && game.puzzle[i] == '_') {
      game.puzzle[i] = guess;  // replace the _ with the actual letter
      --game.remaining_letters;
      correct_guess = true;
    }
  }
  if (!correct_guess) {
    ++game.miss_count;
  }
}
```

Lastly, we can write the top-level function that plays a game. It starts by creating a `Game` object via `make_game()`. It then has a loop that:

- Prints the game using `print_game()`.
- Obtains a guess from the player via `get_guess()`. If `ERROR_CHAR` is returned, the game immediately exits.
- Updates the game using `update_game()`.

Other than the `ERROR_CHAR` condition, the game terminates either when all letters have been guessed, or the player has made the maximum number of incorrect guesses. Thus, the following is the main loop:

```
while (game.remaining_letters > 0 && game.miss_count < MAX_MISSES) {
  print_game(game);
  char letter = get_guess();
  if (letter == ERROR_CHAR) {
    cout << "Quitting." << endl;
    return; // quit the game
  }
  update_game(game, letter);
}
```

After the game is over, we print the game once more to show its final state. We then print a message to the player depending on whether or not they won, as shown in the full function definition below:

```
void play_game(std::string answer) {
  Game game = make_game(answer);
  while (game.remaining_letters > 0 && game.miss_count < MAX_MISSES) {
    print_game(game);
    char letter = get_guess();
    if (letter == ERROR_CHAR) {
      std::cout << "Quitting." << std::endl;
      return;  // quit the game
    }
    update_game(game, letter);
  }
  print_game(game);
  if (game.remaining_letters == 0) {
    std::cout << "Congratulations!" << std::endl;
  } else {
    std::cout << "Better luck next time!" << std::endl;
```

(continues on next page)

(continued from previous page)

```
  }
}
```

36.5.5 Testing and File Organization

Before we write a `main()` function that plays the game, we might want to write some code that tests individual functions in our program. For example, the following code does a basic test of the `make_game()` function:

```
#include <cassert>
#include <string>

void test_make_game() {
  std::string answer = "hello world!";
  Game game = make_game(answer);
  assert(game.miss_count == 0);
  assert(game.remaining_letters == 10);
  assert(game.puzzle == "_____ _____!");
  assert(game.answer == answer);
}

int main() {
  test_make_game();
}
```

The `test_make_game()` function creates a `Game` object from the `"hello world!"` answer string. It then asserts that each of the member variables of the `Game` object has the expected value. The `assert()` construct is defined in the `cassert` library header, which we have included at the top.

Observe that the test code has its own `main()` function, so it needs to be in a separate `.cpp` source file than the `main()` function that actually plays a game of stickman. Let's assume that the function definitions above are placed in `stickman.cpp`, and our test code is in `test.cpp`. The compiler will process these two files individually, even if they are both provided in a single compilation command. How can we make the compiler aware of the functions defined in `stickman.cpp` when it is compiling `test.cpp`?

The compiler actually only needs access to the declarations of the functions we use, not the definitions. It does need access to the definition of the `Game` struct, since it needs to know what member variables the struct has. Thus, common practice is to place struct definitions, function declarations, and constants in a separate *header* file, conventionally with a file extension such as `.hpp`. The following code can be placed in `stickman.hpp`:

```
#include <string>
#include <vector>

// Stick figures corresponding to each possible number of misses.
const std::vector<std::string> STICK_FIGURES = {
  "\n\n   ",
  " o\n\n   ",
  " o\n |\n   ",
  " o\n/|\n   ",
  " o\n/|\\\n   ",
  " o\n/|\\\n/  ",
  " o\n/|\\\n/ \\"
};
```

(continues on next page)

(continued from previous page)

```
// Maximum number of missed guesses.
const int MAX_MISSES = 6;

// Used to indicate an error when reading user input.
const char ERROR_CHAR = '\0';   // special "null" character

// A struct to represent the state of a game of stickman.
struct Game {
  // The number of wrong guesses made so far.
  int miss_count;

  // The number of remaining unguessed positions in the puzzle.
  int remaining_letters;

  // The current state of the puzzle, with underscores representing
  // unguessed letters.
  std::string puzzle;

  // The answer to the puzzle.
  std::string answer;
};

// EFFECTS: Returns a properly initialized Game object corresponding
//          to the given answer phrase.
Game make_game(std::string answer);

// REQUIRES: game.miss_count <= MAX_MISSES
// MODIFIES: cout
// EFFECTS: Prints the state of the game to standard output.
void print_game(Game &game);

// MODIFIES: cout, cin
// EFFECTS: Repeatedly prompts the user for a guess consisting of a
//          single lowercase letter until a valid guess is provided,
//          or the end of stream is reached. Returns the guess in
//          the first case, or ERROR_CHAR in the second.
char get_guess();

// MODIFIES: game
// EFFECTS: Updates the game's puzzle and miss count according to
//          whether the guess is a letter in the answer and has not
//          been previously guessed.
void update_game(Game &game, char guess);

// MODIFIES: cout, cin
// EFFECTS: Plays a game of stickman with the given answer, reading
//          guesses from standard in and writing game details to
//          standard out. The game ends if the end of standard input
//          is reached, the player exhausts the maximum number of
//          incorrect guesses, or the player correctly guesses all
//          letters.
```

(continues on next page)

(continued from previous page)

```
void play_game(std::string answer);
```

We start by including the `string` and `vector` standard libraries, since the code in `stickman.hpp` uses both strings and vectors. We then have our constant definitions, followed by the definition of the `Game` struct. Lastly, we have function definitions for each task in the game. We have included documentation in the form of *RMEs* (*requires*, *modifies*, and *effects*):

- The *requires* clause specifies what is required to be true prior to calling the function. These are also called *preconditions*. All bets are off if these conditions are violated: we get the dreaded undefined behavior. The function's implementation is allowed to assume that these conditions are met – it is not required to check them in any way.
- The *modifies* clause lists the objects outside the function that might modified by a call to the function. We generally include `cout` if the standard output stream is written to, and `cin` if input is read from the standard input stream. Pass-by-reference parameters are included if their contents might be modified.
- The *effects* clause tells us what the function actually does, i.e. what the return value means and how any objects in the *modifies* clause are actually modified. The effects are sometimes also called *postconditions*. This clause only specifies the "what", not the "how"; we will *come back to this later*.

Now that we have this code in `stickman.hpp`, we can include it in both `test.cpp` and `stickman.cpp`. The contents of `test.cpp` are as follows:

```
#include <cassert>
#include "stickman.hpp"

void test_make_game() {
  std::string answer = "hello world!";
  Game game = make_game(answer);
  assert(game.miss_count == 0);
  assert(game.remaining_letters == 10);
  assert(game.puzzle == "_____ _____!");
  assert(game.answer == answer);
}

int main() {
  test_make_game();
}
```

The `#include "stickman.hpp"` directive pulls in the code from `stickman.hpp` into the current file. We use quotes around the filename rather than the angle brackets we use with a library header. Observe that we no longer need the `#include <string>` directive, since `stickman.hpp` already contains that.

The full contents of `stickman.cpp` are as follows:

```
#include <iostream>
#include "stickman.hpp"

using std::cin;
using std::cout;
using std::endl;
using std::size_t;
using std::string;

static bool is_lowercase(char value) {
```

(continues on next page)

(continued from previous page)

```
  return value >= 'a' && value <= 'z';
}

Game make_game(string answer) {
  Game game = { 0, 0, answer, answer };
  for (size_t i = 0; i < game.puzzle.size(); ++i) {
    if (is_lowercase(game.puzzle[i])) {
      game.puzzle[i] = '_';
      ++game.remaining_letters;
    }
  }
  return game;
}

void print_game(Game &game) {
  cout << endl;
  cout << STICK_FIGURES[game.miss_count] << " ";
  for (size_t i = 0; i < game.puzzle.size(); ++i) {
    cout << " " << game.puzzle[i];
  }
  cout << "\n" << endl;
}

char get_guess() {
  cout << "Enter a lowercase letter to guess:" << endl;
  string input;
  while (cin >> input) {
    if (input.size() != 1) {
      cout << "Error: guess must be exactly one letter" << endl;
    } else if (!is_lowercase(input[0])) {
      cout << "Error: guest must be between a and z" << endl;
    } else {
      return input[0];
    }
  }
  return ERROR_CHAR;
}

void update_game(Game &game, char guess) {
  bool correct_guess = false;
  for (size_t i = 0; i < game.answer.size(); ++i) {
    if (game.answer[i] == guess && game.puzzle[i] == '_') {
      game.puzzle[i] = guess;  // replace the _ with the actual letter
      --game.remaining_letters;
      correct_guess = true;
    }
  }
  if (!correct_guess) {
    ++game.miss_count;
  }
}
```

(continues on next page)

(continued from previous page)

```
void play_game(string answer) {
  Game game = make_game(answer);
  while (game.remaining_letters > 0 && game.miss_count < MAX_MISSES) {
    print_game(game);
    char letter = get_guess();
    if (letter == ERROR_CHAR) {
      cout << "Quitting." << endl;
      return;  // quit the game
    }
    update_game(game, letter);
  }
  print_game(game);
  if (game.remaining_letters == 0) {
    cout << "Congratulations!" << endl;
  } else {
    cout << "Better luck next time!" << endl;
  }
}
```

We include `iostream`, since that is not included by `stickman.hpp`. We don't have struct or constant definitions, since those are defined in the `stickman.hpp` header. Instead, we just have the definitions for each function.

We made a few minor changes from before:

- We added `using` directives so that we can use specific standard-library entities without the `std::` qualification.
- We preceded the definition of `is_lowercase()` with the `static` keyword. This is common practice for a helper function, and it prevents the function from conflicting with a function of the same name defined in some other source file.

We can now compile and run the test code. We provide both `.cpp` source files to the compilation command, but **not** the `.hpp` file – its contents are pulled directly into the `.cpp` files via the `#include "stickman.hpp"` directive:

```
$ g++ -std=c++17 -o test.exe test.cpp stickman.cpp
$ ./test.exe
```

The assertions all succeed, so we don't see any output.

Lastly, we can write a separate `play.cpp` file that has a `main()` function to play the game. The following are the contents of the file:

```
#include <iostream>
#include "stickman.hpp"

int main(int argc, char *argv[]) {
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " <answer> " << std::endl;
    return 1;
  }
  play_game(argv[1]);
}
```

We use an alternate signature for `main()` that gives us access to the *command-line arguments* given to the program. The first argument is always the name of the program executable. We take the game answer as the second argument, so we start by checking whether there are at least two arguments. If not, we print an error message and return with a nonzero value. If an answer is provided, we invoke the top-level `play_game()` function with the answer.

We compile and run the program as follows:

```
$ g++ -std=c++17 -o play.exe play.cpp stickman.cpp
$ ./play.exe world!



     _ _ _ _ _ !

Enter a lowercase letter to guess:
e

 o

     _ _ _ _ _ !

Enter a lowercase letter to guess:
t

 o
 |
     _ _ _ _ _ !

Enter a lowercase letter to guess:
s

 o
/|
     _ _ _ _ _ !

Enter a lowercase letter to guess:
o

 o
/|
     _ o _ _ _ !

Enter a lowercase letter to guess:
r

 o
/|
     _ o r _ _ !

Enter a lowercase letter to guess:
a

 o
/|\
     _ o r _ _ !

Enter a lowercase letter to guess:
l
```

(continues on next page)

(continued from previous page)

```
 o
/|\
     _ o r l _ !

Enter a lowercase letter to guess:
d

 o
/|\
     _ o r l d !

Enter a lowercase letter to guess:
w

 o
/|\
     w o r l d !

Congratulations!
```

CHAPTER

THIRTYSEVEN

SOLVING PROBLEMS WITH RECURSION

To conclude our discussion of recursion, we take a look at several complex problems and how to approach them from a recursive standpoint. Our strategy will be to:

- Identify the cases we can solve directly. These are base cases.
- Determine how to express a general case in terms of smaller subproblems. This is the recursive case.

For the latter, the first step is to identify appropriate subproblems, and the second is to figure out how to express the solution for the whole problem in terms of the solutions to the subproblems. This leads to a recurrence relation, which we can then implement in code.

37.1 Pancake Sort

Suppose you want to sort a stack of irregular pancakes such that they are in order, with the smallest at the top and the largest at the bottom. So as not to contaminate the pancakes, you would like to do so just by repeatedly flipping a subset of the pancakes with a spatula, which reverses the set of pancakes above the spatula, as shown in Figure 37.1. The goal is to come up with a generalized algorithm for sorting the pancakes just by flipping substacks from any point in the stack to the top.

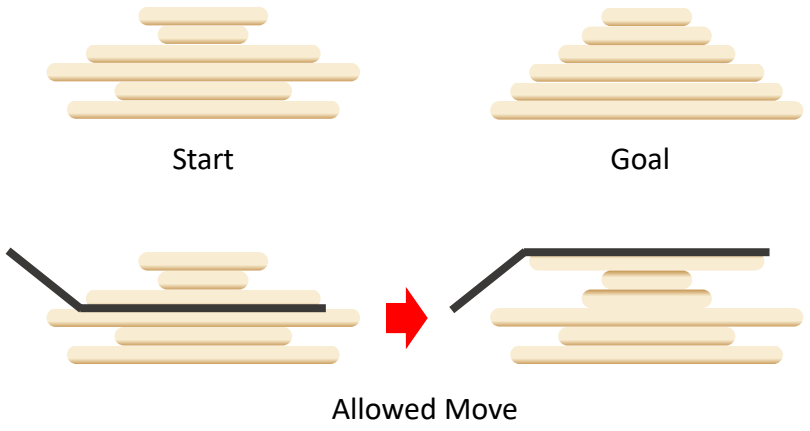

Figure 37.1: The pancake problem is to sort a stack of pancakes, where the only allowed move is to flip a subset of the stack from a point in the middle to the top.

We start by identifying cases we can solve directly. An empty stack or one with just a single pancake are trivially sorted, so these are the base cases.

We then need to identify a subproblem. For a stack of n pancakes, a natural subproblem is to sort a stack of $n - 1$ pancakes. So we now have to figure out how to reduce the problem of sorting n pancakes to that of sorting $n - 1$

pancakes – we just need the largest pancake on the bottom, and then the substack above that has $n - 1$ pancakes that need sorting.

The final step is to come up with a way of getting the largest pancake to the bottom. We know that if that pancake is at the top, we can flip the whole stack to get it to the bottom. So now the problem is how to get the largest pancake to the top. Placing the spatula under that pancake and flipping the stack above does the trick.

Thus, we have identified our recurrence:

- Nothing need be done for a stack of zero or one.
- For n pancakes where $n > 1$, we place the spatula under the largest pancake and flip, moving it to the top. We then flip the whole stack, moving the largest pancake to the bottom. We then repeat the process on the $n - 1$ pancakes above the largest.

Figure 37.2 illustrates this recurrence.

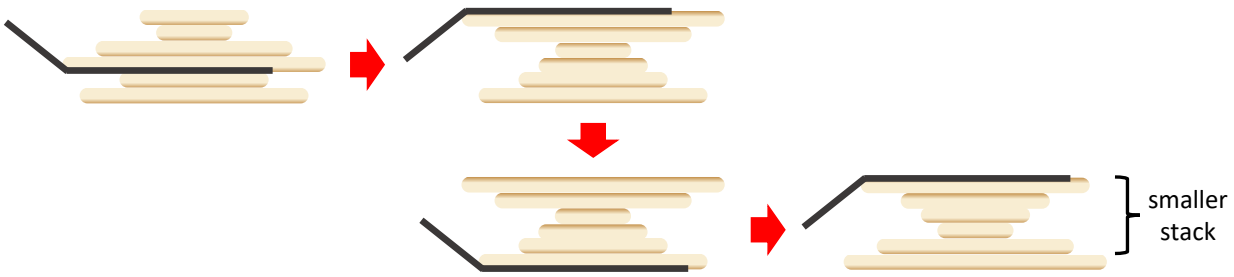

Figure 37.2: The pancake-sort algorithm flips the largest pancake to the top, then the whole stack, then recurses on a smaller stack.

We can implement this algorithm in code to sort an array of integers, considering the top of the stack to be at the beginning of the array and the bottom at the end. We make use of the `reverse()` function *we wrote previously* to flip a subset of the stack.

```
void pancake_sort(int *stack, int size) {
  if (size > 1) {
    // find position of largest element
    int *largest = std::max_element(stack, stack + size);
    // flip the stack from the top to the largest element
    reverse(stack, largest);
    // flip the whole stack
    reverse(stack, stack + size - 1);
    // recurse on a smaller stack
    pancake_sort(stack, size - 1);
  }
}
```

Here, we use `std::max_element()` from the <algorithm> library to find the largest item in the stack.

37.2 Tower of Hanoi

The Tower of Hanoi puzzle consists of three rods with n disks of varying size arranged in sorted order on the first rod, as shown in Figure 37.3. The objective is to move the entire stack of disks to another rod, while obeying the following constraints:

- Only one disk can be moved at a time.
- Only the top disk at a rod can be moved.
- A larger disk may never be placed on top of a smaller one.

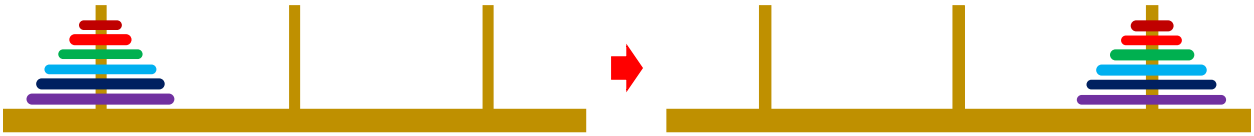

Figure 37.3: The Tower of Hanoi problem moves a stack of disks from an initial rod to a target rod.

To come up with an algorithm to solve this puzzle, we first identify the base cases that can be solved directly. For $n = 1$, the lone disk can be moved directly without violating the constraints, constituting our base case.

For $n > 1$, the subproblem is moving $n - 1$ disks between rods. Then our task is to determine how to express the solution for moving n disks in terms of the solution to moving $n - 1$ disks. We observe that we can completely ignore the largest disk when moving the $n - 1$ smaller disks; since any of the latter can be placed on the largest disk, the rod with the largest disk acts just like an empty rod. Thus, we can move the $n - 1$ smaller disks as a stack of their own, from the start rod to the middle rod. We can then move the largest disk directly to the empty target rod. Then all that is left is to move the stack of $n - 1$ smaller disks to the target rod.

Figure 37.4 illustrates this algorithm. We take the recursive leap of faith, assuming that we can move the smaller $n - 1$ stack as a whole using recursion, without worrying about the details of how it is done. As long as the subproblem is closer to the base case than the original problem, we get to make the assumption that the subproblem will be solved correctly.

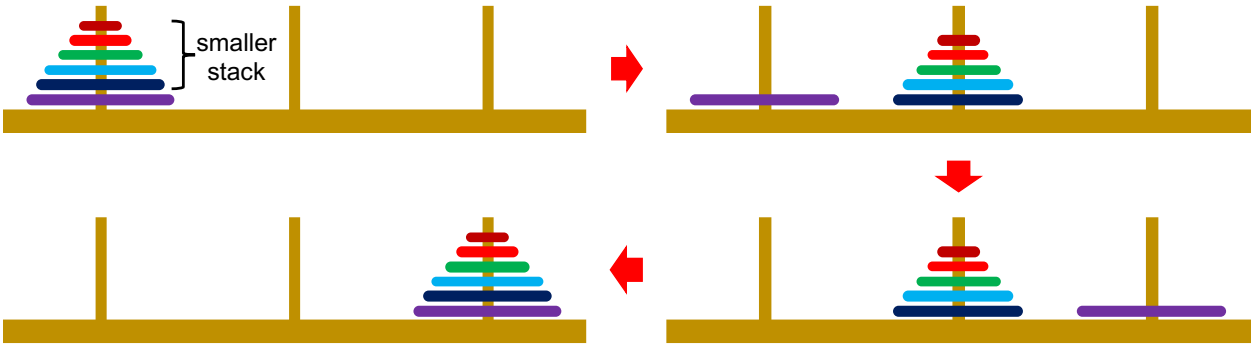

Figure 37.4: An algorithm for the Tower of Hanoi is to use recursion to move a smaller stack and move the largest disk on its own.

The following code implements the algorithm, printing out the moves required to move n disks from a start to end rod:

```
// MODIFIES: cout
// EFFECTS:  Prints a move of disk n from start to end.
void move(int n, int start, int end) {
  std::cout << "Move disk " << n << " from rod " << start
            << " to rod " << end << std::endl;
```

(continues on next page)

(continued from previous page)

```
}

// REQUIRES: n >= 1
// MODIFIES: cout
// EFFECTS:  Prints the sequence of moves required to move n disks
//           from start to end, using temp as the temporary rod.
void hanoi(int n, int start, int temp, int end) {
  if (n == 1) {
    move(n, start, end);
  } else {
    hanoi(n - 1, start, end, temp);
    move(n, start, end);
    hanoi(n - 1, temp, start, end);
  }
}
```

The result of `hanoi(5, 1, 2, 3)` is as follows:

```
Move disk 1 from rod 1 to rod 3
Move disk 2 from rod 1 to rod 2
Move disk 1 from rod 3 to rod 2
Move disk 3 from rod 1 to rod 3
Move disk 1 from rod 2 to rod 1
Move disk 2 from rod 2 to rod 3
Move disk 1 from rod 1 to rod 3
Move disk 4 from rod 1 to rod 2
Move disk 1 from rod 3 to rod 2
Move disk 2 from rod 3 to rod 1
Move disk 1 from rod 2 to rod 1
Move disk 3 from rod 3 to rod 2
Move disk 1 from rod 1 to rod 3
Move disk 2 from rod 1 to rod 2
Move disk 1 from rod 3 to rod 2
Move disk 5 from rod 1 to rod 3
Move disk 1 from rod 2 to rod 1
Move disk 2 from rod 2 to rod 3
Move disk 1 from rod 1 to rod 3
Move disk 3 from rod 2 to rod 1
Move disk 1 from rod 3 to rod 2
Move disk 2 from rod 3 to rod 1
Move disk 1 from rod 2 to rod 1
Move disk 4 from rod 2 to rod 3
Move disk 1 from rod 1 to rod 3
Move disk 2 from rod 1 to rod 2
Move disk 1 from rod 3 to rod 2
Move disk 3 from rod 1 to rod 3
Move disk 1 from rod 2 to rod 1
Move disk 2 from rod 2 to rod 3
Move disk 1 from rod 1 to rod 3
```

37.3 Counting Change

As a final example, let us consider the number of ways we can change a dollar into coins, using any quantity of half dollars (50¢), quarters (25¢), dimes (10¢), nickels (5¢), and pennies (1¢). The following are three possible ways:

- $1 = 1 half dollar, 1 quarter, 2 dimes, 1 nickel
- $1 = 2 quarters, 2 dimes, 30 pennies
- $1 = 100 pennies

There are too many ways to enumerate them all manually. Instead, let's take a look at a smaller example of the same problem: the number of ways to make change for 11¢ using dimes, nickels, and pennies. There are exactly four:

- 11¢ = 1 dime, 1 penny
- 11¢ = 2 nickels, 1 penny
- 11¢ = 1 nickel, 6 pennies
- 11¢ = 11 pennies

We can categorize these solutions according to which coins they use, as shown in Figure 37.5.

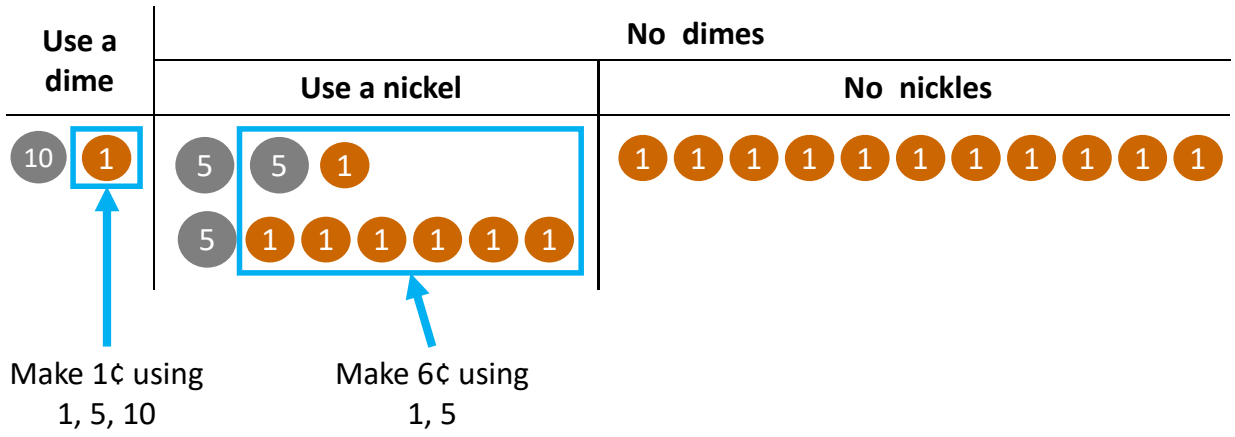

Figure 37.5: The ways to count change can be subdivided into those that use a particular coin and those that don't.

If we decide to use a particular coin, we are left with the subproblem of making a smaller amount of change using the same set of coins. For example, if we choose to use a dime, we need only make 1¢ using dimes, nickels, and pennies. On the other hand, if we choose not to use a particular coin, we must make the full amount of change with a smaller set of coins. In our example, if we decide not to use a dime, we must make the full 11¢ with only nickels and pennies. If we then use a nickel, we are left with making 6¢ with only nickels and pennies.

Figure 37.6 illustrates a decision tree for how to make change. At any point in time, we have a target amount and a set of available coins. We then decide whether to use the largest of our available coins. Either decision leads to a subproblem:

- Making a smaller amount of change with the same set of available coins.
- Making the same amount of change with a smaller set of available coins.

Thus, we have a recurrence that expresses the solution for a full problem in terms of two subproblems.

We also need to identify base cases that we can solve directly:

- If the change amount is zero, there is only one way to do that: use no coins.
- If the change amount is negative, there is no way to do that, since we do not have negative coins.

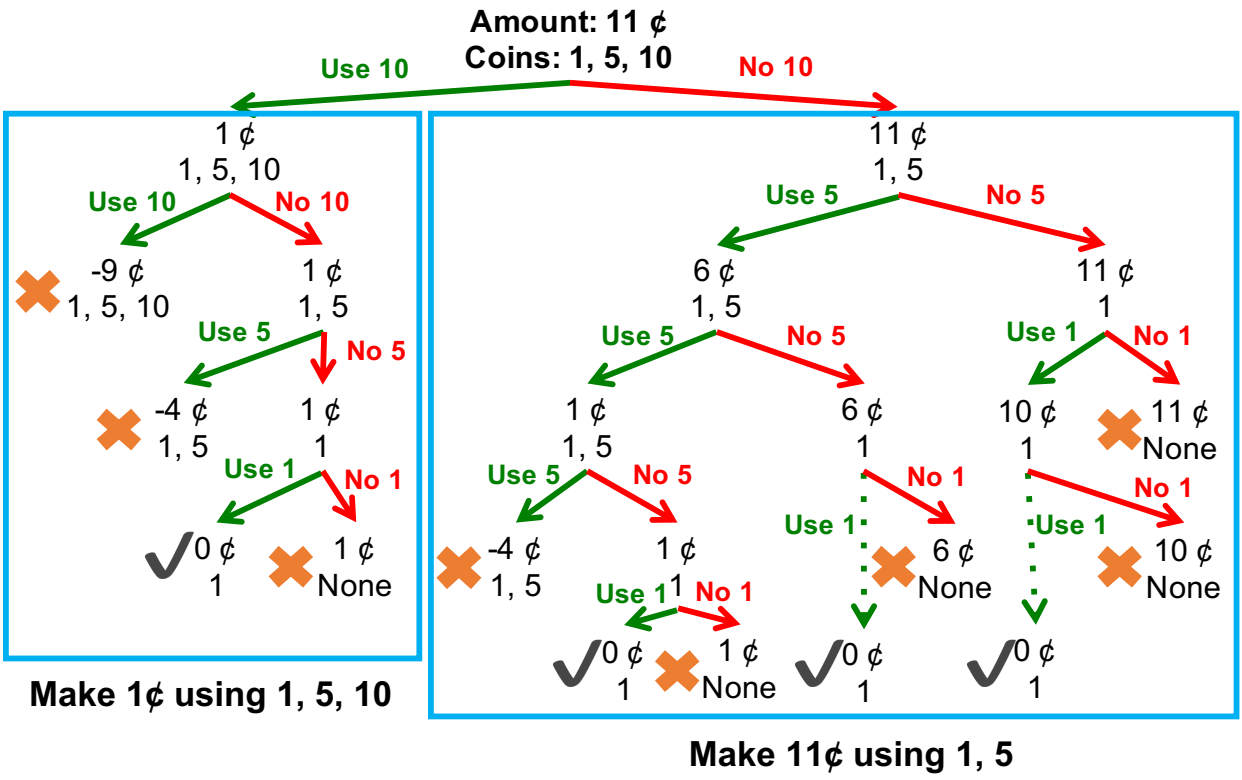

Figure 37.6: The decision tree for which coin to use consists of a subtree that uses a particular coin and another that doesn't.

- If the change amount is positive but we have no coins available, we cannot make that change.

The following implements the resulting algorithm in code:

```
// EFFECTS: Returns the number of ways to make amount in change
//          using only the coin denominations in kinds.
int count_change(int amount, const int kinds[], int num_kinds) {
  if (amount == 0) {
    // one way to make nothing: use no coins
    return 1;
  } else if (amount < 0 || num_kinds < 1) {
    // cannot make negative amount, or anything with no coins
    return 0;
  } else {
    return
      // use the largest coin, reducing the amount of change to make
      count_change(amount - kinds[num_kinds - 1], kinds, num_kinds) +
      // don't use the largest coin, reducing the available coins
      count_change(amount, kinds, num_kinds - 1);
  }
}
```

This tree-recursive implementation is very inefficient, repeating the same computations many times. Techniques such as *memoization* and *dynamic programming* can drastically improve the efficiency, but they are beyond the scope of this course. (With such a technique in place, the result is 292 for the number of ways to make change for a dollar.)

CHAPTER

THIRTYEIGHT

CONTAINERS OF POINTERS

Recall the *linked-list class template* we defined previously:

```
template <typename T>
class List {
public:
  List();

  void empty() const;

  T & front();

  void push_front(const T &datum);

  void pop_front();

  void push_back(const T &datum);

  void pop_back();

  ...

private:
  struct Node {
    T datum;
    Node *prev;
    Node *next;
  };

  Node *first;
  Node *last;
};
```

We declared the parameters of `push_front()` and `push_back()` to be passed by reference, avoiding making a copy of the argument value. However, a `Node` stores a value of type `T`, so a copy is made when a node is created:

```
template <typename T>
void List<T>::push_back(const T &datum) {
  Node *node = new Node{ datum, last, nullptr };
  ...
}
```

With this in mind, consider the following example that inserts local objects into a list:

```
int main() {
  Llama paul("Paul");
  Llama carl("Carl");

  List<Llama> todo;
  todo.push_back(paul);
  todo.push_back(carl);

  for (auto &llama : todo) {
    llama.feed();
  }
}
```

The code creates two local `Llama` objects and inserts them into a list. It then iterates over the list to feed each llama. However, as shown in Figure 38.1, the llamas in the list are copies of the local objects, so feeding them does not affect the original llamas, which go hungry.

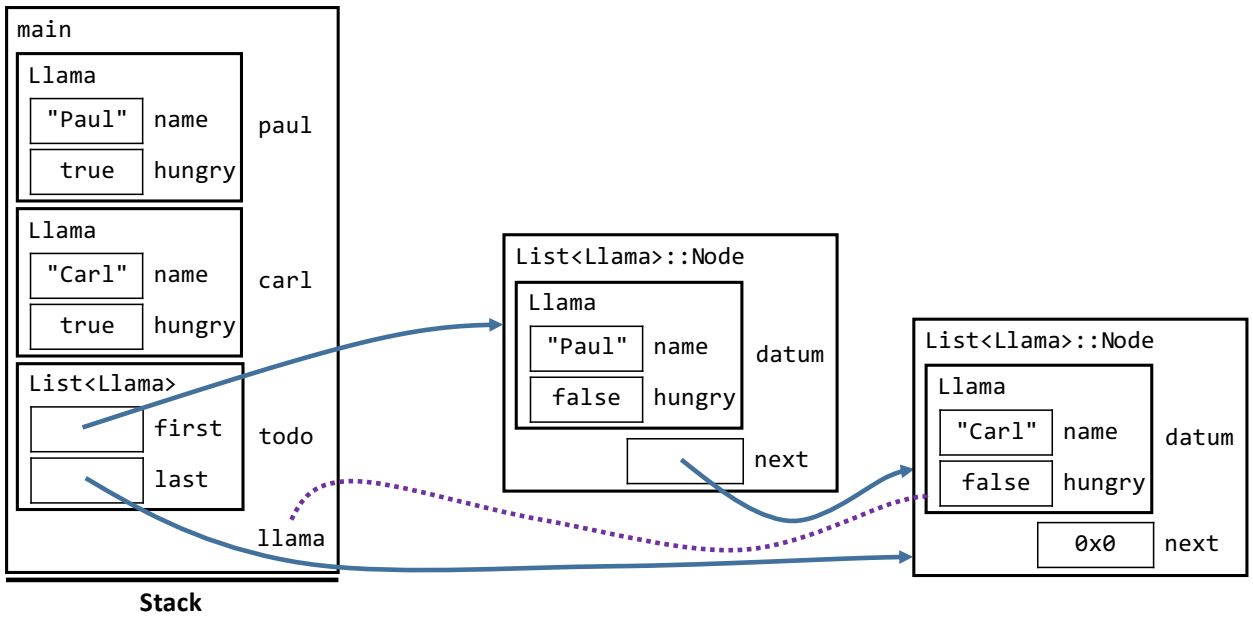

Figure 38.1: Inserting objects into a container creates copies of those objects.

We can use indirection to avoid making a copy, storing pointers to llamas in a list rather than llamas themselves:

```
int main() {
  Llama paul("Paul");
  Llama carl("Carl");

  List<Llama *> todo;
  todo.push_back(&paul);
  todo.push_back(&carl);

  for (auto lptr : todo) {
    lptr->feed();
```

(continues on next page)

(continued from previous page)

```
  }
}
```

The code iterates over the list by value rather than by reference; however, the values are pointers, so they still indirectly refer to the appropriate `Llama` objects when they are copied. Figure 38.2 illustrates the result in memory. Our llamas are properly fed and no longer go hungry.

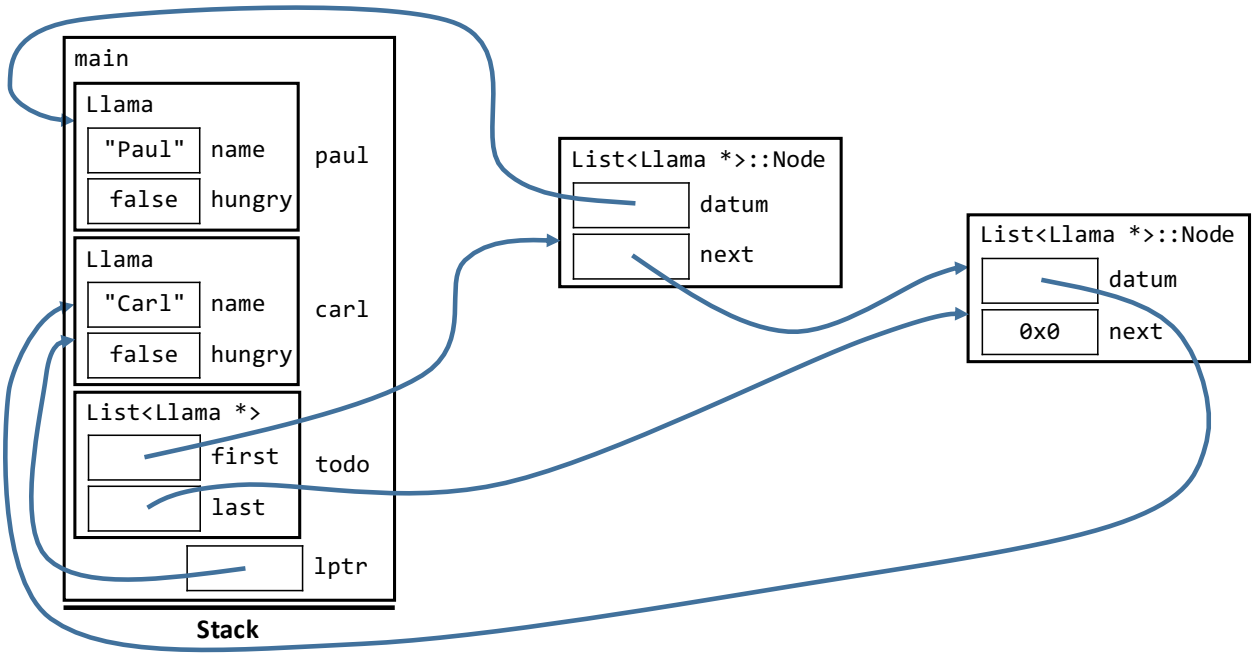

Figure 38.2: Storing pointers in a container.

By storing pointers in a container, we avoid making copies of the underlying objects.

Containers of pointers are also useful for keeping track of multiple orderings of the same objects. For instance, we may want to store our llamas both in order of age as well as alphabetically by name. The following code creates `Llama` objects in dynamic memory and stores pointers to them in two different lists:

```
int main() {
  List<Llama *> by_age;
  by_age.push_back(new Llama("Paul"));
  by_age.push_back(new Llama("Carl"));

  List<Llama *> by_name;
  by_name.push_back(by_age.back());
  by_name.push_back(by_age.front());
}
```

Figure 38.3 shows the resulting storage.

The code above, however, has a memory leak; the destructors for the lists only free the memory for the `Node` objects, so the `Llama` objects do not get deleted. We need to manually delete them before the lists that we are using to track them go away:

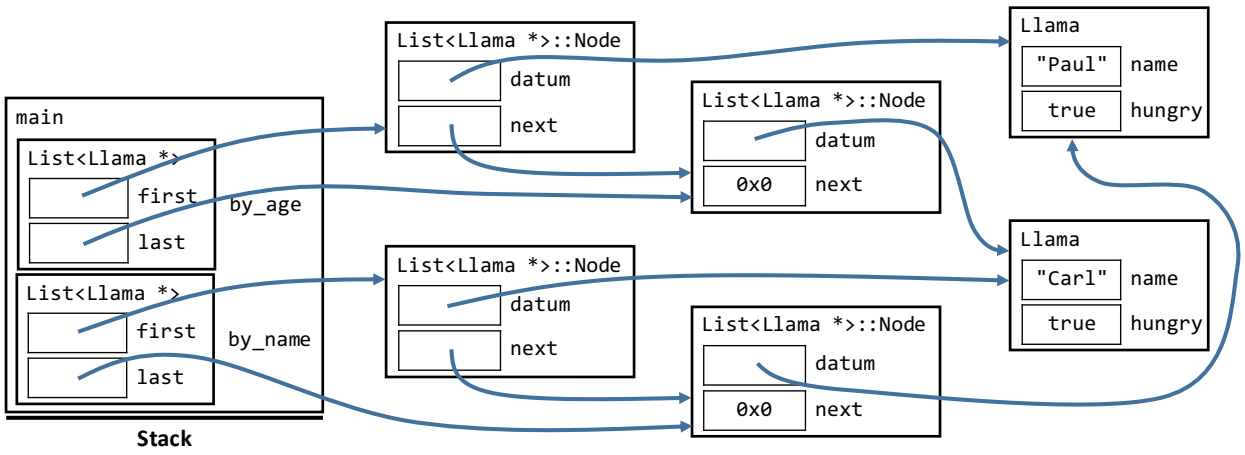

Figure 38.3: Two containers that store pointers to the same objects.

```
int main() {
  List<Llama *> by_age;
  by_age.push_back(new Llama("Paul"));
  by_age.push_back(new Llama("Carl"));

  List<Llama *> by_name;
  by_name.push_back(by_age.back());
  by_name.push_back(by_age.front());

  for (auto llama : by_age) {
    delete llama;
  }

  for (auto llama : by_name) {
    delete llama;                 // UNDEFINED BEHAVIOR
  }
}
```

This code is erroneous: it deletes each `Llama` object twice, resulting in undefined behavior. We need to be careful to avoid memory errors when we have multiple containers referring to the same dynamic objects. What we should do is designate one container as the canonical "owner" of the objects, only deleting them when that container is about to die:

```
int main() {
  List<Llama *> by_age;                   // "owner" of llama objects
  by_age.push_back(new Llama("Paul"));
  by_age.push_back(new Llama("Carl"));

  List<Llama *> by_name;
  by_name.push_back(by_age.back());
  by_name.push_back(by_age.front());

  for (auto llama : by_age) {  // delete llamas when by_age is dying
    delete llama;
  }
}
```

Alternatively, we can store the objects directly in the container that "owns" them, so that the destructor for the container does the work of reclaiming those objects:

```
int main() {
  List<Llama> by_age;                    // "owner" of llama objects
  by_age.push_back(Llama("Paul"));
  by_age.push_back(Llama("Carl"));

  List<Llama *> by_name;
  by_name.push_back(&by_age.back());
  by_name.push_back(&by_age.front());
}                      // llamas die automatically when by_age dies
```

Here, we construct the `Llama` objects as temporaries when passing them to `push_back()`. Standard-library containers have `emplace_back()` functions that avoid creating temporaries, and we pass the constructor arguments directly to that function:

```
int main() {
  List<Llama> by_age;                    // "owner" of llama objects
  by_age.emplace_back("Paul");
  by_age.emplace_back("Carl");

  List<Llama *> by_name;
  by_name.push_back(&by_age.back());
  by_name.push_back(&by_age.front());
}                      // llamas die automatically when by_age dies
```

The result in memory is shown in Figure 38.4.

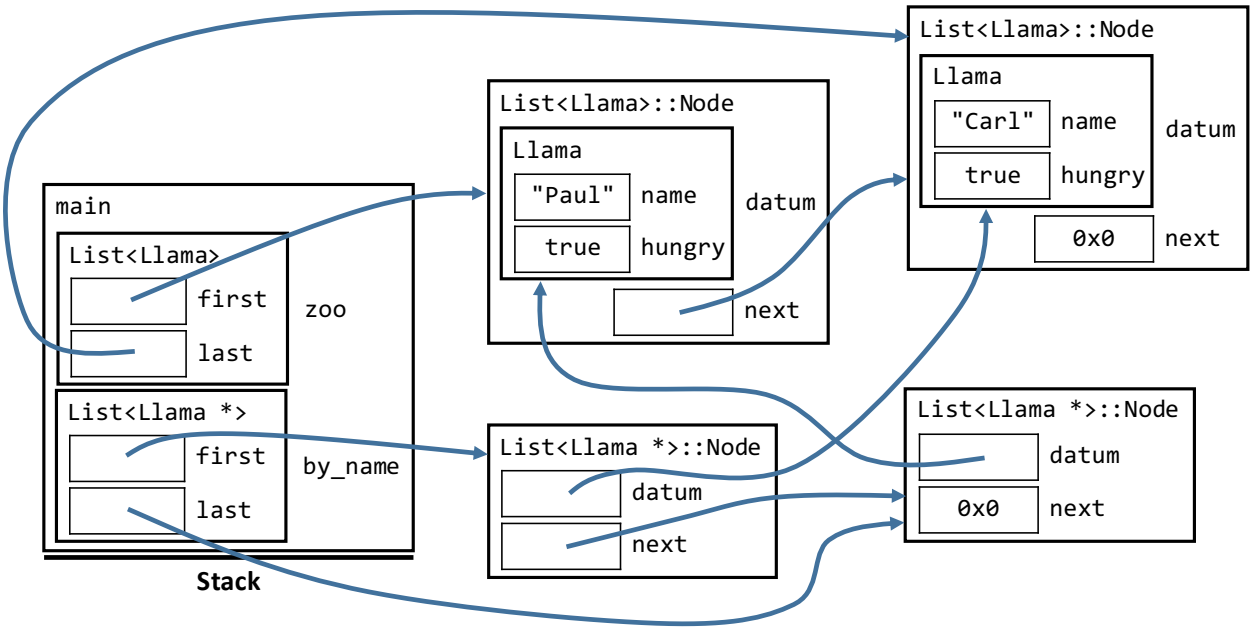

Figure 38.4: Storing pointers to elements of a different container.

The llamas die automatically when the container in which they reside dies, at the end of `main()` in the code above.

38.1 Sorting Containers of Pointers

The following code stores pointers to llamas in a container and then sorts them:

```
int main() {
  vector<Llama *> llamas = { new Llama("Paul"), new Llama("Carl") };
  std::sort(llamas.begin(), llamas.end());
  ...
}
```

While this code compiles, it sorts the pointers in the container by the address values they store, not by some property of the `Llama` objects. This is because `std::sort()` uses the < operator by default, which for pointers just compares the addresses they contain. The result depends on where the two `Llama` objects were placed in memory, which is implementation-dependent.

Instead, we need to supply our own comparator to `std::sort()`, and the comparator can use whatever property we choose of the underlying `Llama` objects:

```
class LlamaPointerNameLess {
public:
  bool operator()(const Llama *l1, const Llama *l2) const {
    return l1->get_name() < l2->get_name();
  }
};
```

Here, we have chosen to sort llamas by name. The comparator operates on pointers to `Llamas`, since that is what the container stores. We then call `std::sort()` as follows:

```
std::sort(llamas.begin(), llamas.end(), LlamaPointerNameLess());
```

The third argument is a default-constructed, temporary `LlamaPointerNameLess` object. Now `std::sort()` will use that to compare the `Llama *` elements, resulting in them being ordered by the names of the respective llamas.

We can use different comparators to maintain different orderings of the same objects:

```
vector<Llama> llamas = { Llama("Paul"), Llama("Carl") };

vector<Llama *> by_age;
for (auto &llama : llamas) {
  by_age.push_back(&llama);
}
std::sort(by_age.begin(), by_age.end(), LlamaPointerAgeLess());

vector<Llama *> by_name = by_age;
std::sort(by_name.begin(), by_name.end(), LlamaPointerNameLess());
```

38.2 Containers of Polymorphic Objects

Keeping track of polymorphic objects, meaning objects of different derived classes of the same base class, requires indirection *as we saw previously*. Containers of pointers enable this indirection:

```
int main() {
  vector<Animal *> zoo;
  zoo.push_back(new Gorilla("Colo"));
  zoo.push_back(new Llama("Paul"));
  zoo.push_back(new Panda("Po"));

  for (auto animal_ptr : zoo) {
    animal_ptr->talk();   // prints different messages for each animal
  }

  ...
}
```

As long as the `talk()` function is virtual, dynamic binding will be used, and each animal will print its own sound.

When we are done with the objects, we need to delete them ourselves, since they are in dynamic memory:

```
for (auto animal_ptr : zoo) {
  delete animal_ptr;
}
```

As we saw previously, this requires the `Animal` destructor to be declared as `virtual` so that dynamic binding is used to call the appropriate destructor for each object.

CHAPTER

THIRTYNINE

C AND C++ STRINGS

A *string* is a sequence of characters, and it represents text data. C++ has two string abstractions, which we refer to as *C-style strings* and *C++ strings*.

39.1 C-Style Strings

In the original C language, strings are represented as just an array of characters, which have the type `char`. The following initializes a string representing the characters in the word `hello`:

```
char str[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

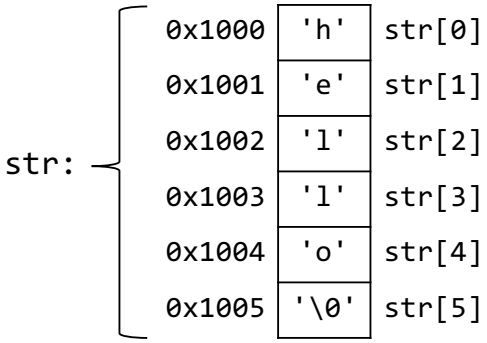

Figure 39.1: Array representation of a string.

Character literals are enclosed in single quotes. For example `'h'` is the character literal corresponding to the lower-case letter h. The representation of the string in memory is shown in Figure 39.1.

A C-style string has a *sentinel* value at its end, the special *null character*, denoted by `'\0'`. This is not the same as a null pointer, which is denoted by `nullptr`, nor the character `'0'`, which denotes the digit 0. The null character signals the end of the string, and algorithms on C-style strings rely on its presence to determine where the string ends.

A character array can also be initialized with a string literal:

```
char str2[6] = "hello";
char str3[] = "hello";
```

If the size of the array is specified, it must have sufficient space for the null terminator. In the second case above, the size of the array is inferred as 6 from the string literal that is used to initialize it. A string literal implicitly contains the null terminator at its end, so both `str2` and `str3` are initialized to end with a null terminator.

The `char` type is an atomic type that is represented by numerical values. The *ASCII* standard specifies the numerical values used to represent each character. For instance, the null character `'\0'` is represented by the ASCII value 0, the digit `'0'` is represented by the ASCII value 48, and the letter `'h'` is represented by the ASCII value 104. Figure 39.2 illustrates the ASCII values that represent the string `"hello"`.

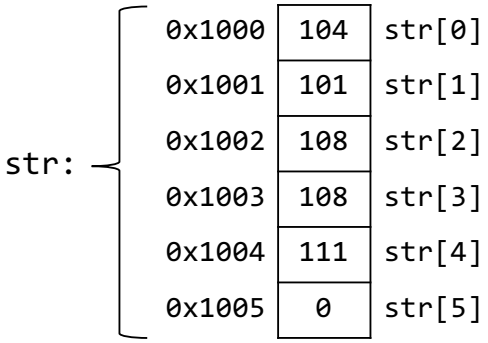

Figure 39.2: ASCII values of the characters in a string.

An important feature of the ASCII standard is that the digits 0-9 are represented by consecutive values, the capital letters A-Z are also represented by consecutive values, and the lower-case letters a-z as well. The following function determines whether a character is a letter:

```
bool is_alpha(char ch) {
  return (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z');
}
```

In C++, atomic objects with value 0 are considered to have false truth values, while atomic objects with nonzero values are considered to be true. Thus, the null terminator is the only character that has a false truth value. We will make use of that when implementing algorithms on C-style strings.

Since C-style strings are just arrays, the pitfalls that apply to arrays also apply to C-style strings. For instance, a `char` array turns into a pointer to `char` when its value is required. Thus, comparisons and assignments on C-style strings cannot be done with the built-in operators:

```
char str1[6] = "hello";
char str2[6] = "hello";
char str3[6] = "apple";
char *ptr = str1;         // manually convert array into pointer;
                          // ptr points to first character of str1

// Test for equality?
str1 == str2;             // false; tests pointer equality

// Copy strings?
str1 = str3;              // does not compile; RHS turns into pointer

// Copy through pointer?
ptr = str3;               // sets ptr to point to first character of str3
```

When initializing a variable from a string literal, the variable can be an array, in which case the individual characters are initialized from those in the string literal:

```
char str1[6] = "hello";
```

The variable can also be a pointer, in which case it just points to the first character in the string literal itself. String literals are stored in memory; however, the C++ standard prohibits us from modifying the memory used to store a string literal. Thus, we must use the `const` keyword when specifying the element type of the pointer:

```
const char *ptr = "hello";
```

39.1.1 String Traversal and Functions

The conventional pattern for iterating over a C-style string is to use *traversal by pointer*: walk a pointer across the elements until the end is reached. However, unlike the traversal pattern we saw previously where we already knew the length, we don't know the end of a C-style string until we reach the null terminator. Thus, we iterate until we reach that sentinel value:

```
// REQUIRES: str points to a valid, null-terminated string
// EFFECTS:  Returns the length of str, not including the null
//           terminator.
int strlen(const char *str) {
  const char *ptr = str;
  while (*ptr != '\0') {
    ++ptr;
  }
  return ptr - str;
}
```

Here, we compute the length of a string by creating a new pointer that points to the first character. We then increment that pointer[1] until reaching the null terminator. Then the distance between that pointer and the original is equal to the number of non-null characters in the string.

We can also use the truth value of the null character in the test of the while loop:

```
int strlen(const char *str) {
  const char *ptr = str;
  while (*ptr) {
    ++ptr;
  }
  return ptr - str;
}
```

We can also use a for loop, with an empty initialization and body:

```
int strlen(const char *str) {
  const char *ptr = str;
  for (; *ptr; ++ptr);
  return ptr - str;
}
```

The built-in <cstring> header contains a definition for `strlen()`.

We saw previously that we cannot copy C-style strings with the assignment operator. Instead, we need to use a function:

[1] The type `const char *` denotes a pointer that is pointing at a constant character. This means that the pointed-to character cannot be modified through the pointer. However, the pointer itself can be modified to point to a different character, which is what happens when we increment the pointer.

```
// REQUIRES: src points to a valid, null-terminated string;
//           dst points to an array with >= strlen(src) + 1 elements
// MODIFIES: *dst
// EFFECTS:  Copies the characters from src into dst, including the
//           null terminator.
void strcpy(char *dst, const char *src) {
  while (*src) {
    *dst = *src;
    ++src;
    ++dst;
  }
  *dst = *src;   // null terminator
}
```

The function takes in a destination pointer; the pointed-to type must be non-const, since the function will modify the elements. The function does not need to modify the source string, so the corresponding parameter is a pointer to `const char`. Then each non-null character from `src` is copied into `dst`. The last line also copies the null terminator into `dst`.

The `strcpy()` function can be written more succinctly by relying on the behavior of the postfix increment operator. There are two versions of the increment operator, and their evaluation process is visualized in Figure 39.3:

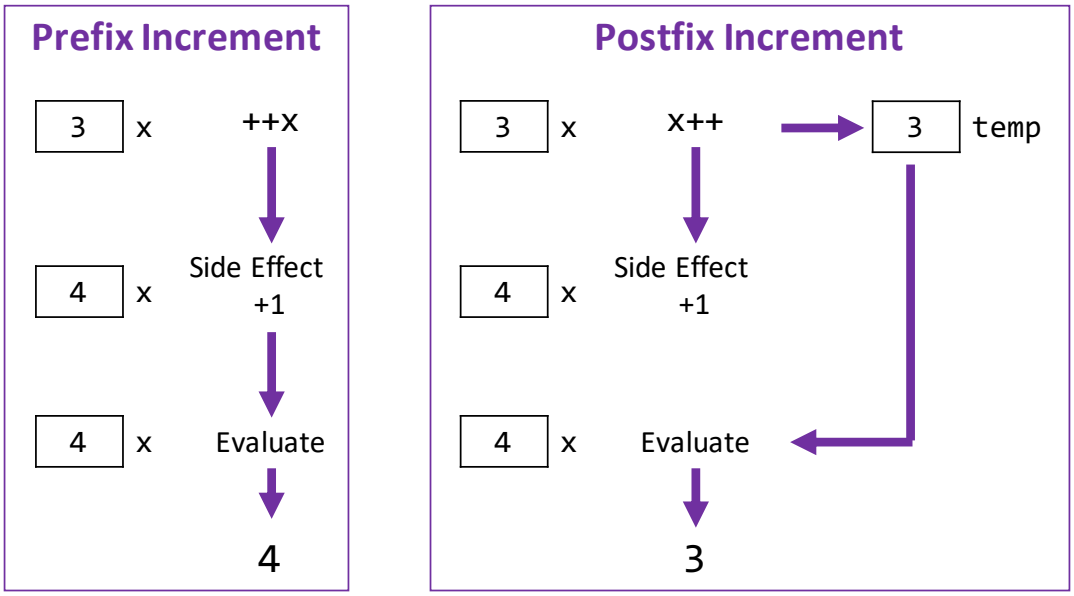

Figure 39.3: Evaluation process for prefix and postfix increment.

- The prefix increment operator, when applied to an atomic object, increments the object and evaluates to the object itself, which now contains the new value:

```
int x = 3;
cout << ++x;   // prints 4
cout << x;     // prints 4
```

- The postfix increment operator, when applied to an atomic object, increments the object but evaluates to the old value:

```
int x = 3;
cout << x++;   // prints 3
cout << x;     // prints 4
```

There are also both prefix and postfix versions of the decrement operator (`--`).

A word of caution when writing expressions that have side effects, such as increment: in C++, the order in which subexpressions are evaluated within a larger expression is for the most part unspecified. Thus, the following results in implementation-dependent behavior:

```
int x = 3;
cout << ++x << "," << x;   // can print 4,4 or 4,3
```

If the second `x` in the print statement is evaluated before `++x`, then a 3 will be printed out for its value. On the other hand, if the second `x` is evaluated after `++x`, a 4 will be printed out for its value. Code like this, where a single statement contains two subexpressions that use the same variable but at least one modifies it, should be avoided.

Another feature that our shorter version of `strcpy()` will rely on is that an assignment evaluates back to the left-hand-side object:

```
int x = 3;
int y = -4;
++(x = y);        // copies -4 into x, then increments x
cout << x;        // prints -3
cout << (y = x);  // prints -3
```

The succinct version of `strcpy()` is as follows:

```
void strcpy(char *dst, const char *src) {
  while (*dst++ = *src++);
}
```

The test increments both pointers, but since it is using postfix increment, the expressions themselves evaluate to the old values. Thus, in the first iteration, `dst++` and `src++` evaluate to the addresses of the first character in each string. The rest of the test expression dereferences the pointers and copies the source value to the destination. The assignment then evaluates to the left-hand-side object, so the test checks the truth value of that object's value. As long as the character that was copied was not the null terminator, it will be true, and the loop will continue on to the next character. When the null terminator is reached, the assignment copies it to the destination but then produces a false value, so the loop terminates immediately after copying over the null terminator.

The <cstring> library also contains a version of `strcpy()`.

39.1.2 Printing C-Style Arrays

Previously, we say that printing out an array prints out the address of its first character, since the array turns into a pointer. Printing out a pointer just prints out the address value contained in the pointer.

On the other hand, C++ output streams have special treatment of pointers to `char`. If a pointer to `char` is passed to `cout`, it will assume that the pointer is pointing into a C-style string and print out every character until it reaches a null terminator:

```
char str[] = "hello";
char *ptr = str;
cout << ptr;          // prints out hello
cout << str;          // str turns into a pointer; prints out hello
```

This means that we must ensure that a `char *` is actually pointing to a null-terminated string before passing it to `cout`. The following results in undefined behavior:

```
char array[] = { 'h', 'e', 'l', 'l', 'o' };  // not null-terminated
char ch = 'w';                                // just a character
cout << array;  // undefined behavior -- dereferences past end of array
cout << &ch;    // undefined behavior -- dereferences past ch
```

To print out the address value of a `char *`, we must convert it into a `void *`, which is a pointer that can point to any kind of object:

```
cout << static_cast<void *>(&ch);  // prints address of ch
```

39.2 C++ Strings

C++ strings are class-type objects represented by the `string` type[2]. They are not arrays, though the implementation may use arrays under the hood. Thus, C++ strings are to C-style strings as vectors are to built-in arrays.

The following table compares C-style and C++ strings:

	C-Style Strings	C++ Strings
Library Header	`<cstring>`	`<string>`
Declaration	`char cstr[]; char *cstr;`	`string str`
Length	`strlen(cstr)`	`str.length()`
Copy Value	`strcpy(cstr1, cstr2)`	`str1 = str2`
Indexing	`cstr[i]`	`str[i]`
Concatenate	`strcat(cstr1, cstr2)`	`str1 += str2`
Compare	`!strcmp(cstr1, cstr2)`	`str1 == str2`

A C++ string can be converted into a C-style string by calling `.c_str()` on it:

```
const char *cstr = str.c_str();
```

A C-style string can be converted into a C++ string by explicitly or implicitly calling the `string` constructor:

```
string str1 = string(cstr);  // explicit call
string str = cstr;           // implicit call
```

C++ strings can be compared with the built-in comparison operators, which compare them lexicographically: the ASCII values of elements are compared one by one, and if the two strings differ in a character, then the string whose character has a lower ASCII value is considered less than the other. If one string is a prefix of the other, then the shorter one is less than the longer (which results from comparing the ASCII value of the null terminator to a non-null character).

C-style strings cannot be compared with the built-in operators – these would just do pointer comparisons. Instead, the `strcmp()` function can be used, and `strcmp(str1, str2)` returns:

- a negative value if `str1` is lexicographically less than `str2`
- a positive value if `str1` is lexicographically greater than `str2`
- 0 if the two strings have equal values

[2] Technically, `string` is an alias for `basic_string<char>`, so you may see the latter in compiler errors.

The expression `!strcmp(str1, str2)` is often used to check for equality – if the two strings are equal, `strcmp()` returns 0, which has truth value false.

CHAPTER

FORTY

ABOUT

These notes were written by Amir Kamil in Winter 2019 for EECS 280. They are based on the lecture slides by James Juett and Amir Kamil, which were themselves based on slides by Andrew DeOrio and many others.