Data Structures in C++

A Companion Textbook for Harvey Mudd's CS70

Matthew G. Calligaro

Contents

Preface

This textbook was created as a resource for students taking "Data Structures and Program Development" (CS70), an introductory-level computer science course taught at Harvey Mudd College. This is the first major-level course in the computer science major and covers core CS concepts and simple data structures. When I worked as a TA for CS70, the course did not have an official textbook. Each semester, students expressed desire for materials that closely followed the content of the course to provide an opportunity to review concepts outside of class. This encouraged me to create the resources which evolved into the textbook that you are reading today.

This is not a textbook in the traditional sense—its primary goal is to support students enrolled in Harvey Mudd's CS70, not serve as a stand-alone resource for teaching this material. To that end, it is structured like a review guide and aims to present important concepts as succinctly as possible, avoiding the depth and detail afforded by a full-length textbook. Its content, vocabulary, examples, and level of abstraction are tightly coupled with the lectures, assignments, and exams of CS70. Because CS70 has two prerequisite CS courses, this textbook assumes that the reader is familiar with basic computational concepts such as variables, functions, and for loops.

This text contains three chapters each divided into several smaller sections. The first chapter introduces major topics in C++ including object lifetime, compilation, memory management, and templates. The second chapter explores basic data structures including linear data structures, trees, heaps, and hash sets. The third chapter builds tools for analyzing run time including asymptotic, amortized, and best/worst/expected case analysis.

While this textbook is specifically tailored towards students enrolled in Harvey Mudd's CS70 course, I hope that it may also provide value to students at other institutions or anyone hoping to learn or brush up on core computer science concepts.

Supporting Materials

Hands on experience is one of the best ways to gain a concrete understanding of complex CS topics. To that end, this textbook is supported by a GitHub repository located at https://github.com/MatthewCalligaro/CS70Textbook. This repository is divided into several case studies which roughly correspond to sections from the textbook. Each case study contains example code which produces a program the reader can easily compile and run, as well as recommended exercises and reflection questions. The intent is for readers to clone this repository and modify/experiment with the code and programs in these case studies.

Acknowledgements

I wish to express my deep gratitude toward Professors Chris Stone, Lucas Bang, Julie Medero, Ran Libeskind-Hadas, and Zach Dodds, all of whom have supported me in different ways throughout this process. I am also grateful to the many students of CS70 whom I have had the pleasure of teaching over the last four semesters. They are the ones who inspired me to create this textbook, and their energy, enthusiasm, and feedback have shaped it into what it has become today.

1 Core Concepts in C++

1.1 OBJECTS

An *object* is a collection of data interpreted with a certain meaning. For example, an `int` is an object consisting of 32 bits[1] which are interpreted as representing an integer number. *Types* are categories of objects which are represented and interpreted in the same way. This includes *primitive types* like `int` and `double`, other standard types like `std::string` or `std::list<int>`, and types created by the programmer by writing a class.

C++ is an object-oriented language, which means that programs are represented and understood as a collection of interacting objects. At its core, C++ therefore focuses on the creation, modification, and use of data. This differs from a functional language like Haskell which is built around functions rather than data.

Stack and Heap

Main memory is divided into two regions: the stack and the heap. Objects can be stored in either location, which effects when and how they are created and destroyed. Section 1.2 takes a closer look at the stack and heap.

The *stack* stores data associated with functions. Any variable created in the body of a function without the `new` keyword is placed on the stack. It is automatically destroyed and deallocated at the end of the function call, so the programmer does not need to worry about deleting it. An object can only be placed on the stack if its exact size is known at compile time.

The *heap* provides a location in which the programmer can specify the size of an object at runtime and directly control when it is created and destroyed. The `new` keyword allocates space on the heap for an object. While the stack is organized based on the order in which functions are called, a program can place an object wherever it sees fit on the heap, so objects allocated with different calls to `new` may not be near each other. Objects on the heap are not destroyed and deallocated automatically—instead, we trigger this process with the `delete` keyword.

Object Lifetime

An object's life can be divided into five stages, which occur at different times depending on whether the object is located on the stack or heap.

Stage	What it means	When it occurs (stack)	When it occurs (heap)
Allocation	Find a space for the object in memory.	Opening curly bracket of the function.	Line containing the `new` keyword.
Initialization	Provide an initial value to the data of the object by calling its constructor.	Line on which the variable is declared.	Call to the constructor after the `new` keyword.
Use	Interact with the object such as by reading and modifying its data.	Everything between initialization and destruction.	
Destruction	Delete external data associated with the object by calling its destructor.	Closing curly bracket which ends the variable's scope.	Call to delete.
Deallocation	Allow the memory used by the object to be used again.	Closing curly bracket of the function.	Call to delete.

[1] The number of bits used to represent an int depends on the data model used by your operating system, but in all modern operating systems, an int is 32 bits (see https://en.cppreference.com/w/cpp/language/types).

The following example demonstrates these stages for several objects. Notice that `elliot` is a pointer stored on the stack and is a different object than the `Sheep` on the heap to which it points.

```
int objectLifetime(int x) {  // Allocate space for x, y, z, shawn, and elliot; initialize x

  int y = 2;              // Initialize y
  Sheep shawn("shawn");  // Initialize shawn
  shawn.pet();            // Use shawn

  if (x > y) {        // Use x and y
    int z = x + y;   // Initialize z, use x and y
    x *= z;          // Use x and z
  }                  // Destroy z

  // Initialize elliot, allocate space for and initialize a Sheep on the heap,
  // and use shawn
  Sheep* elliot = new Sheep(shawn);

  elliot->pet();  // Use the Sheep on the heap and elliot
  delete elliot;  // Destroy and deallocate the Sheep on the heap, use elliot

  return x;  // Initialize the returned object
}  // Destroy and deallocate x, y, z, shawn, and elliot
```

Special Data Types

Pointers

Pointers are primitive types used to store the *address* of an object on the stack or heap. Pointer types have the format `<typename>*`, where `<typename>` is the type of the object stored at that address. For example, a variable of type `int*` stores the address of an `int` and is referred to as an `int` pointer. This allows us to string together multiple pointers such as `int**`, which stores the address of an `int*`.

Pointers support the following operations:

- **Pointer arithmetic (`operator+`)**: Adding or subtracting from a pointer adds or subtracts from the address. For example, if `a` is a pointer on the stack storing the heap address `h10`, then `(a + 1)` would evaluate to the heap address `h11` (see Section 1.2 for discussion of the CS70 memory model, which explains what `h10` means).
- **Dereference operator (`operator*`)**: Returns a reference to the object located at the address stored by the pointer. For example, `*a` returns a reference to the object located at the address stored in `a`.
- **Bracket operator (`operator[]`)**: By definition, `a[n]` is equivalent to `*(a + n)`, which is especially useful when a pointer stores the address of an array.
- **Arrow operator (`operator->`)**: By definition, `a->member` is equivalent to `(*a).member`, where `member` is a member variable or method of the object at the location stored in `a`. For example, if `a` is a `Sheep*` and `Sheep` has a member function `pet`, then `a->pet()` would pet the `Sheep` at the location stored in `a`.

The "address of" symbol (&) can be applied to any variable to find the address at which it is located. For example, `&a` gives the address at which the variable `a` is located.

Arrays

C++ has two types of arrays: static and dynamic. A *static array* is stored on the stack and declared without the new keyword, while a *dynamic array* is located on the heap and declared with the new keyword.

```
int staticA[5];              // A static array of 5 default-constructed ints
int* dynamicA = new int[5];  // A dynamic array of 5 default-constructed ints
```

In this example, staticA and dynamicA are both int pointers (int*), even though staticA does not contain an int* anywhere in its definition. This allows us to access the elements of both arrays with pointer math, such as using *(staticA + 1) or *(dynamicA + 1) to access the "first" element of each array (note that there is a "zeroth" element before it). However, it is more idiomatic to use operator[] instead, such as staticA[1] or dynamicA[1].

Since static arrays are stored on the stack, their size must be known at compile time, so the expression between the brackets must evaluate to a constant value.

```
const size_t c = 2;
size_t v = 3;
int static1[c];      // This is alright because c is const
int static2[2 + c];  // This is alright because 2 + c is a constant expression
int static3[v];      // This will cause a compile time error
```

We can pass parameters to the constructors of array objects by listing them in curly brackets after the closing bracket (]) of the array. For example, the following code creates a static and a dynamic array each containing the ints 1 through 5.

```
int sa[5]{1, 2, 3, 4, 5};
int* da = new int[5]{1, 2, 3, 4, 5};
```

References

A reference provides an alias to an existing object, meaning "another name" for the same thing. For example:

```
int d = 7;
int& e = d;
```

will make e a reference to d. Any time we read e, we get the value of d, and any time we change e, we actually change d. A const reference allows us to read but not change the object to which it refers. Continuing the example above:

```
const int& f = d;
std::cout << f << std::endl;  // This will print 7 to standard output
++f;                          // This will cause a compile time error
```

In its entire lifetime, a reference can only alias to the object given during initialization. For this reason, we cannot default initialize a reference since it would have no value to which to alias. After a reference is initialized, setting it equal to a different object will simply call the assignment operator on the object to which the reference aliases.

```
int a = 1;
int b = 2;
int& r = a;  // Makes r an alias to a
r = b;       // Equivalent to writing a = b

// This will print: 2 2 2
std::cout << a << " " << r << " " << b << std::endl;

int& s;  // This will cause a compile time error
```

Classes

A user can write a *class* to define a new type of object. The *data members* of a class are the objects used to store the data associated with a class. On the stack or heap, an instance of a class is stored simply as a collection of its data members. A class will also define *member functions* (also known as *methods*) which can operate on objects of the class. Traditionally, the class declaration is written in a `.hpp` file and the definition is written in a `.cpp` file. The following terms can be used to describe a class:

- **Interface**: The public elements of a class. For example, `insert`, `pop_back`, and `operator=` methods are all part of the `std::vector` interface. A class's interface is found in the public section of its `.hpp` file.
- **Encoding**: The data members that represent a class and dictate how it is stored in memory. For example, the `std::string` class in C++ is encoded as a `char` array. A class's encoding is usually found in the private section of its `.hpp` file.
- **Implementation**: The way in which methods declared in the interface are performed. This consists of the method definitions located in the class's `.cpp` file.

A *struct* is simply a special type of class in which everything is automatically public.

Constructors, Assignment Operator, and Destructor

Constructors are special methods used to initialize an object. The *member initialization list* of a constructor specifies the argument(s) to pass to the constructor of each data member during initialization. Any data member not in the member initialization list is default constructed. The code between the curly brackets after the member initialization list is referred to as the body of the constructor. Everything in the constructor body is in the use phase for the object and its data members. For example, suppose that the `Sheep` class is encoded with the following three private data members.

```
Sheep* mother_;
std::string name_;
size_t age_;
```

In the following constructer, `: name_{name}, age_{1}` is the member initialization list, which initializes `name_` with the value of the parameter `name`, initializes `age_` with the value 1, and default constructs `mother_`. When `mother_` is set to `nullptr` in the body of the constructor, this is the use phase for `mother_`, not initialization. This is a bad use of the constructor body; it would have been more efficient and idiomatic to initialize `mother_` to `nullptr` in the member initialization list. The `nap` method is also called in the body of the constructor and thus occurs in the use phase of the `Sheep` that was just initialized. This is a good use of the constructor body, since the `Sheep` must be fully initialized before it can `nap`.

```
Sheep::Sheep(const std::string& name) : name_{name}, age_{1} {
  // Everything in the body of the constructor occurs is in the use phase of the
  // object and its data members
  mother_ = nullptr;  // This is the use phase for mother_, not initialization
  nap();
}
```

We can divide constructors into three categories based on their parameters:

- **Default constructor**: A constructor with no parameters.
- **Copy constructor**: A constructor which takes a constant reference to the type of the class. For example, the `Sheep` copy constructor would be declared as `Sheep(const Sheep& other);`. The purpose of a copy constructor is to create a new object which is a "copy" of the parameter object `other`.
- **Parameterized constructor**: A constructor with one or more parameters. Frequently, these parameters are values used to initialize the data members of the class. A copy constructor is technically a type of parameterized constructor.

An object's *assignment operator* (`operator=`) is called in expressions such as `a = b` to make the left-hand-side object a copy of the right-hand-side object. Unlike a copy constructor which creates a new object, the assignment operator requires that both objects already exist.

An object's *destructor* is called when it is destroyed, regardless of whether it is stored on the stack or the heap. It is usually used to delete any external heap data associated with the object to prevent memory leaks.

The following function demonstrates the use of these special methods for the `Sheep` class.

```
void specialMethods() {
  Sheep shawn;                        // Default constructor (stack)
  Sheep* elliot = new Sheep;          // Default constructor (heap)
  Sheep timmy = shawn;                // Copy constructor (stack)
  Sheep* shirley = new Sheep(*elliot);  // Copy constructor (heap)
  Sheep benjamin = Sheep("benjamin");   // Parameterized constructor (stack)
  Sheep* lola = new Sheep("lola");      // Parameterized constructor (heap)
  shawn = *lola;                      // Assignment operator
  delete elliot;                      // Destructor
  delete shirley;                     // Destructor
  delete lola;                        // Destructor
}  // Calls shawn, timmy, and benjamin's destructors
```

If we do not explicitly define a default constructor, copy constructor, assignment operator, or destructor, the compiler will generate a *synthesized* version for us. These versions have the following behavior:

- **Synthesized default constructor**: Calls the default constructor on all data members.
- **Synthesized copy constructor:** Calls the copy constructor on each data members by passing each constructor the corresponding data members from the parameter object.
- **Synthesized destructor**: Calls the destructor on all data members.
- **Synthesized assignment operator**: Calls the assignment operator on each data members with the corresponding data member from the parameter object.

If we provide a parameterized constructor, the compiler will *not* create a synthesized default constructor unless we explicitly tell it to do so. We can explicitly request a synthesized version in the `.hpp` by writing `= default` or explicitly remove the synthesized version by writing `= delete`.

```
Sheep() = default;                   // Explicitly requests synthesized default constructor
Sheep(const Sheep& other) = delete;  // Explicitly removes synthesized copy constructor
```

These synthesized versions rarely have the desired behavior if any of the object's data members are pointers to objects that belong to the class. Then, the synthesized assignment operator and copy constructor will create *shallow copies*, meaning that their pointers will point to the external data of the original rather than creating their own copy of this data. For example, suppose that the `Sheep` class contains a pointer to a `Hat` object and uses the synthesized copy constructor. If we construct Elliot as a copy of Shawn, then Elliot will not receive his own hat, and his `Hat*` will simply point to Shawn's hat. If Shawn modifies or removes his hat, Elliot's hat will be affected as well. To avoid this problem, we usually prefer to create a *deep copy* by also copying all the external data associated with the class (such as the sheep's hat).

In these situations, the synthesized destructor will cause memory leaks. Continuing the same example, the synthesized `Sheep` destructor will not delete the `Hat` on the heap. Recall that pointers are primitive data types, so calling the destructor of a pointer does nothing. Instead, you must call `delete` on a pointer to free the memory at the address stored in the pointer, something that must be done in a manually written destructor.

This leads us to the *Rule of 3*: if we manually implement the copy constructor, assignment operator, or destructor of a class, we should manually implement all three. If we are not using the synthesized behavior for one of these methods, we likely do not want the synthesized behavior for the other two.

Use in Arrays

To initialize an array, we must initialize each element of the array by calling its constructor. For both static and dynamic arrays, we can specify the parameters to pass each constructor in curly brackets after the closing square bracket of the array. Any object without parameters specified in this list is default constructed. When an array is destroyed, the destructor is called on each element of the array.

Suppose that the Sheep class supports the following constructors.

```
Sheep();
Sheep(const Sheep& other);
Sheep(const std::string& name);
Sheep(const std::string& name, size_t age);
```

The following function shows how these constructors can be used in arrays.

```
void sheepArrays() {
  Sheep a1[2];                                  // Default constructor
  Sheep* a2 = new Sheep[2]{"shawn", "elliot"};  // Parameterized constructor
  Sheep a3[2]{a1[0], a2[0]};                    // Copy constructor

  // a4[0] is initialized with the Sheep copy constructor
  // a4[1] is initialized with the Sheep default constructor
  // a4[2] is initialized with the first Sheep parameterized constructor
  // a4[3] is initialized with the second Sheep parameterized constructor
  // a4[4] is initialized with the Sheep default constructor
  Sheep a4[4] = {a2[1], {}, "timmy", {"benjamin", 2}};

  delete a2;  // Calls the destructor of the two Sheep in a2
}  // Calls the destructor of each Sheep in a1, a3, and a4
```

Objects in Functions

When a function is called, each non-reference parameter object is initialized with its copy constructor as a copy of the argument object. All parameters declared as references are simply aliases to the argument object and do not make a copy. Similarly, if a function returns a non-reference type, the return value is initialized as a copy of the object after the `return` keyword. If the function returns a reference type, it is initialized as a reference to the return object.

The following two functions highlight the difference between using reference and non-reference types in functions. `petSheep` will create two new copies of the `Sheep` passed to the function, while `petSheepRef` creates no new copies.

```
Sheep petSheep(Sheep s) {  // Construct s as a copy of the Sheep argument
  s.pet();   // Pets s, which is a copy of the argument Sheep, not the original
  return s;  // Constructs a copy of s to return
}

Sheep& petSheepRef(Sheep& s) {  // s is simply an alias for the argument Sheep
  s.pet();   // Pets the argument Sheep (the original)
  return s;  // Returns a reference to the argument Sheep (the original)
}
```

Const

The `const` keyword can have different meanings depending on the context in which it is used. When placed between the parameters and body of a method, it promises that the method will not modify the data members of the object. The `const` label must appear in both the function declaration in the `.hpp` and the function definition in the `.cpp`. If a method is labeled as `const` but attempts to modify a data member of the object, it will cause a compile time error.

```
// Returns the age of a Sheep without changing any data members
size_t Sheep::getAge() const { return age_; }
```

If a variable or reference is labeled `const`, it prevents us from modifying the variable. Specifically, calling a non-`const` method on or creating a non-`const` reference to a `const` variable or reference will cause a compile time error. If a pointer is labeled `const`, then dereferencing the pointer returns a `const` reference rather than a reference. For example, suppose that the `Sheep::shear` function is not `const`.

```
void constExamples() {
  Sheep shawn("shawn");
  const Sheep elliot("elliot");
  const Sheep& ref = shawn;
  const Sheep* timmy = new Sheep("timmy");

  elliot.getAge();  // This is alright because getAge is const
  elliot.shear();   // Compile time error (shear is not const)
  ref = elliot;     // Compile time error (operator= is not const)
  timmy->shear();   // Compile time error (shear is not const)

  // Compile time error: we cannot make a non-const reference to a const object
  Sheep& ref2 = elliot;

  delete timmy;
}
```

Parameters are types of variables, so the same rules apply. It is especially common to label a reference parameter as `const` to promise the caller that the function will not modify the argument.

```
// Check if other is our friend without changing other
bool Sheep::isFriend(const Sheep& other) {
  return true;  // Always return true since all sheep are friends
}
```

Inside-Out Rule

When understanding a long data type, the *inside-out rule* states that we should read the type in the following order:

1. Begin with the variable name (the "inside).
2. Read everything working outward to the right.
3. Return to the variable name and read everything working outward to the left.

For example, we would read the type

```
const   int * *   x [3]
 (6)    (5) (4)(3) (1) (2)
```

as "x$_{(1)}$ is a static array with 3 entries$_{(2)}$, where each entry is a pointer$_{(3)}$ to a pointer$_{(4)}$ to an int$_{(5)}$ which we are not allowed to modify$_{(6)}$."

Possible Exam-style Questions

1. Given the encoding for a class,
    a. Write the default constructor, copy constructor, assignment operator, and destructor.
    b. Reason about whether the synthesized constructors, assignment operator, or destructor would have the desired behavior.
2. Given a certain behavior for a class, declare and define a function that achieves this behavior.
    a. Consider whether the function should be `const`.
    b. Consider whether the parameters should be passed directly, by reference, or by pointer.
    c. Consider whether the parameters should be `const`.
    d. Consider whether the return type should be a value, reference, or `cost` reference.
3. Given code that uses a certain class, determine how many times each of the following are called:
    a. Its default, copy, and parameterized constructors.
    b. Its assignment operator.
    c. Its destructor.
4. Given code, identify the five stages of object lifetime for each object.
5. Given code using the types and syntaxes discussed in this chapter, identify and explain all compile time errors.

Suggested Exercises

Exercise 1

Complete the exercises and answer the reflection questions from the object lifetime case study on the supporting website.

Exercise 2

Create your own data structure which contains the following private data members:

- A dynamic array.
- A static array.
- A pointer to an object of another class which you write.

Then, implement the following methods for this data structure:

- Default, copy, and parameterized constructors.
- Assignment operator.
- Destructor.
- At least one `const` method.
- At least one non-`const` method.

Compile and use your data structure. Use Valgrind to ensure that you have no memory leaks.

1.2 MEMORY

Memory is a type of computer hardware used to store data. Every object in a program must be stored somewhere in memory. As discussed in the previous section, an object can be stored in either the stack or the heap. While stack memory is automatically managed by the program, the programmer has more direct influence over the use of heap memory. Understanding how memory is used helps us write programs that use memory more efficiently and avoid memory-related issues such as leaks and segmentation faults.

The CS70 Memory Model

Modern computer architectures rely on elaborate memory systems complicated by multiple levels of caching and shared memory between processors. CS70 is not a computer systems class, so the class and this textbook use a highly abstracted model of computer memory. This model is a tool to help us reason about important concepts like memory leaks, but **it is not an accurate representation of what happens in a computer**. If you are interested in learning a more accurate model of computer memory, consider consulting a computer systems textbook such as *Computer Systems: A Programmer's Perspective* by Randal Bryant and David O'Hallaron[2].

According to this abstraction, all objects are stored on either the stack or the heap. A *memory address* is a positive integer representing a location in memory. For example, we use `s0` to denote the first element on the stack and `h0` to denote the first element on the heap. In our abstraction, each address indicates a "box" in memory that is large enough to store exactly one object (we pretend that all objects fill the same amount of space in memory). Every box should be labeled with a type, and if a box is associated with a variable, it is labeled with the variable name as well. If the variable is `const`, we draw a lock next to its name

The content of a box represents the data of the object and uses the following rules depending on the object type:

- **Primitive type or std::string**: We write its value inside of the box. Remember that a pointer is a primitive type which stores an address (such as `s1` or `h1`).
- **Non-primitive object**: We draw each data member as a smaller box inside of the box. Each data member should be labeled with a name and a type.
- **Reference**: We write the name of the reference variable next to the box to which it refers as a second name for that box. This shows that the reference is simply an alias to an existing object. Therefore, references do not take up space in memory according to our model (although in reality, this is not always true).

As a general principle, C++ attempts to avoid unnecessary work which is not directly requested by the programmer. As a result, when an object is allocated on either the stack or the heap, C++ does not take the time to clear out whatever data was previously stored at that location. Similarly, when memory is deallocated, C++ simply leaves whatever value was there. As a result, if a variable has been allocated but not yet initialized, we denote its value as `(?)` since we do not know the contents of the stack or heap before our program runs.

The Stack

We model the stack as a single column of boxes with address `s0` at the top and indices growing downwards. Every variable on the stack is labeled with its type after it is allocated and its name after it is initialized. If the variable is a static array**,** we create a box for each element with indices increasing downwards. We do not show the return value or the return address in our model.

When a function is called, a new stack frame is added to the bottom of the stack allocating space for the following objects in the following order:

1. All non-reference parameters in the order in which they appear in the parameter list.
2. All local variables of the function in the order in which they appear in the function.

When we reach the closing curly brace of the function, we deallocate the function's stack frame by erasing it.

[2] If you are a student at Harvey Mudd, you will have the opportunity to learn more about memory in Computer Systems (CS105).

The Heap

The heap is a collection of boxes each labeled with a type and an address of the form `h<number>`. These boxes are not given names, and the addresses can be assigned randomly. For example, if one call to `new` places an object at address `h5`, the next call to `new` will not necessarily place its object at `h6`. Unlike on the stack, which is strictly organized based on the order of variables in code, on the heap, the program can decide where to place objects at runtime and may place them wherever it sees fit. Dynamic arrays are shown as horizontal rows of boxes with consecutive addresses (such as `h20...h24` for a five-element array). A box on the heap is erased when that memory location is deallocated through a call to `delete`.

Memory Model Examples

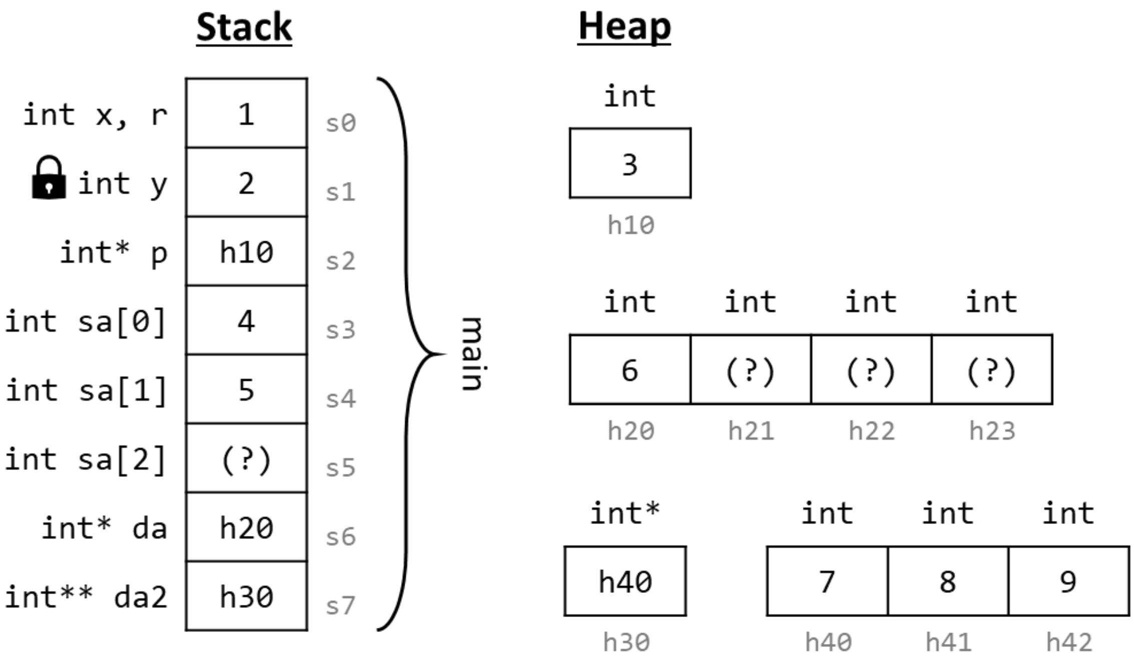
The following examples demonstrate how to use these rules to create a memory diagram for three simple programs.

Example 1: References, Pointers, and Arrays

This first program uses references, pointers, and arrays with the primitive type `int`.

```
 1  void main() {
 2    int x = 1;
 3    int& r = x;
 4    const int y = 2;
 5    int* p = new int(3);
 6    int sa[3] = {4, 5};
 7    int* da = new int[4]{6};
 8    int** da2 = new int*(new int[3]{7, 8, 9});
 9
10    delete p;
11    delete[] da;
12    delete[] * da2;
13    delete da2;
14  }
```

The following diagram shows the state of memory just before line 10 is executed.

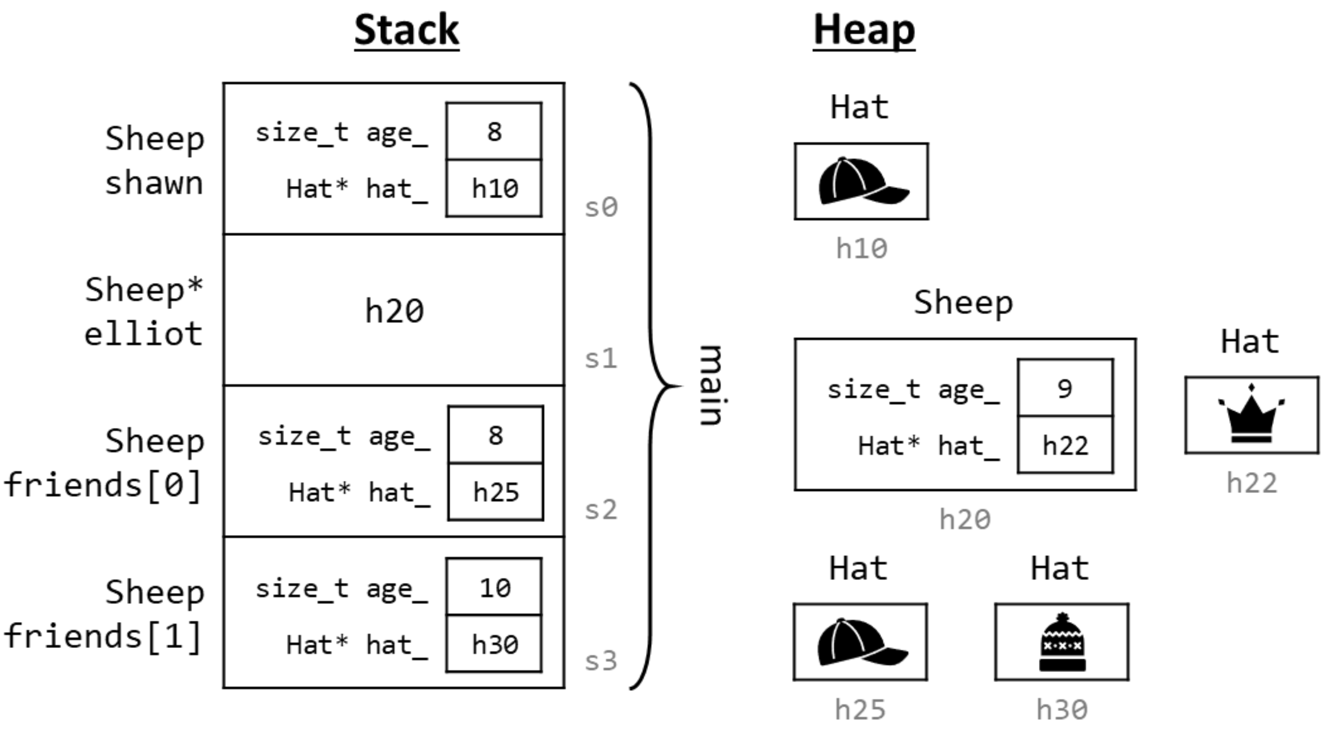


Example 2: Classes

The next program creates instances of the Sheep and Hat classes. The Sheep class has two private data members: a size_t age_ storing its age and a Hat* hat_ storing the address of its Hat on the heap (the hat is created when the sheep is constructed). Sheep has a parameterized constructor taking an age, a copy constructor (which makes a deep copy), and a destructor (which deletes hat_).

```
1  void main() {
2    Sheep shawn(8);
3    Sheep* elliot = new Sheep(9);
4    Sheep friends[2] = {shawn, 10};
5
6    delete elliot;
7  }
```

The following diagram shows the state of memory just before line 6 is executed. Notice that friends[0] is a deep copy of shawn because it has its own Hat at h25 rather than a pointer to shawn's Hat at h10.

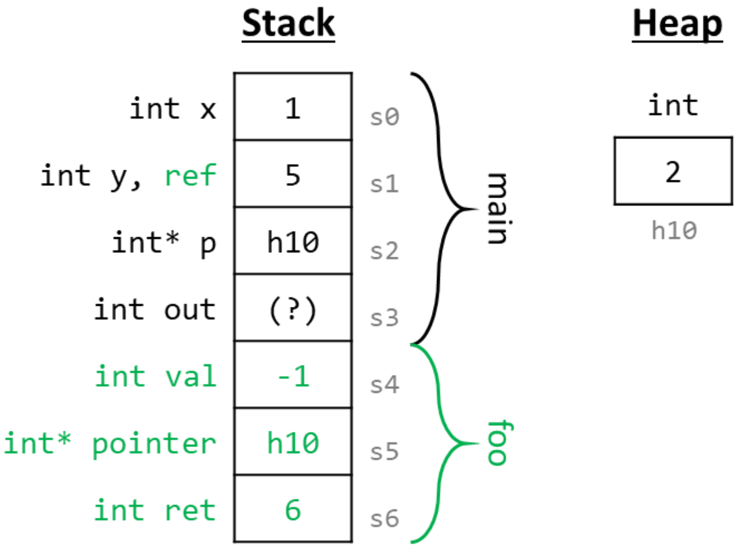


Example 3: Function Calls

The final program calls the function foo.

```
 1  int foo(int val, int& ref, int* pointer) {
 2    val = -val;
 3    *pointer = ref;
 4    ref = 5;
 5    int ret = val + ref + *pointer;
 6    return ret;
 7  }
 8
 9  void example3() {
10    int x = 1;
11    int y = 2;
12    int* p = new int(3);
13    int out = foo(x, y, p);
14  }
```

The following diagram shows the state of memory just before line 6 (the return line of foo) is executed. Data associated with main is shown in **black**, and data associated with foo is shown in **green**.



New and Delete

We will now take a closer look at the behavior of new and delete in the context of our memory model.

New

Singleton new allocates space for one object of the specified type on the heap. The object is also initialized by calling its constructor with the arguments listed after the type.

Step	Singleton new	Example: int* a = new int;
1	Find an unused memory address on the heap.	Find address h10 (10 was chosen randomly).
2	Initialize the object at that address by passing the arguments after the type to the object's constructor.	Default construct an int at h10 (since no arguments were passed).
3	Return the memory address.	Return h10 (so a is initialized with the value h10).

Array new allocates space for a contiguous array of objects of the specified type on the heap. We can use curly braces after the type to specify a list of arguments to the corresponding constructor of each object.

Step	Array new	Example: `int* b = new int[5]{1, 2};`
1	Find a contiguous group of unused memory addresses on the heap.	Find addresses h20 through h24 (again, these numbers were chosen randomly).
2	Initialize an object at each address by passing the corresponding argument(s) from the argument list.	Copy construct an int with value 1 at h20 and 2 at h21, and default construct ints at h22 through h24.
3	Return the memory address of the first object.	Return h20 (so b initialized with the value h20).

Delete

The delete keyword is used to free memory on the heap. Singleton delete is followed by a pointer containing the address of a single object on the heap.

Step	Singleton delete	Example: `delete a;`
1	Go to the heap address stored in the pointer.	Go to h10.
2	Call the destructor on the object at that address.	Call the int's destructor (which does nothing).
3	Free the memory at that address.	Free h10.

Array delete is followed by a pointer containing the first address of an array of objects on the heap.

Step	Array delete	Example: `delete[] b;`
1	Go to the heap address stored in the pointer.	Go to h20.
2	Call the destructor on every object in the array.	Call the destructor of the ints located at h20 through h24 (which does nothing).
3	Free the memory at all addresses in the array.	Free h20 through h24.

To avoid a memory leak, every singleton new must have a corresponding singleton delete, and every array new must have a corresponding array delete. These corresponding deletes may appear far away, possibly in a different function.

Memory Errors

Memory errors are a category of run time errors that occur due to improper handling of memory. Some of these errors such as segfaults will cause the program to crash immediately, while others such as memory leaks will cause problems that are harder to detect.

Segfault

Certain memory addresses are restricted, such as the address of a location inside of the operating system or outside of physical memory. A *segmentation fault* (ie segfault) occurs when a program attempts to access restricted memory. This will cause the program to stop immediately and usually prints an error such as `Segmentation fault (core dumped)`. Most frequently, this happens when a program attempts to dereference nullptr.

```
int* p = nullptr;
*p = 3;  // This will cause a segfault because we attempt to dereference nullptr
```

A segfault might also occur if a program attempts to dereference an uninitialized pointer. An uninitialized pointer will take whatever data was previously stored at its location and interpret it as an address. If that happens to be a restricted address, dereferencing the pointer will cause a segfault.

```
int* p;
*p = 4;  // This will cause a segfault if p happens to store a restricted address
```

Memory Leak

A *memory leak* occurs when a program forgets to free memory on the heap when it is no longer needed. To prevent this, we must ensure that every call to `new` has a corresponding call to `delete`. If an object is responsible for other objects on the heap (such as the `Sheep`/`Hat` example), it must free those objects with the correct calls to `delete` in its destructor to avoid a memory leak.

```
void leak() {
  int* singleton = new int(1);
  int* array = new int[3];
}  // Since we did not delete singleton or array, this function will leak memory
```

Invalid read or write

An *invalid read* or *invalid write* occurs when a program attempts to read or write to an address which it should not. All segfaults are caused by invalid reads or writes, but this category of memory errors also includes smaller mistakes. For example, attempting to access a value one past the end of an array usually will not cause a segfault (since it will not reach restricted memory), but it is always an invalid read.

A *dangling pointer* is any pointer that does not store the address of a valid object of the correct type. Reading or writing to the object at the address stored in a dangling pointer will always be invalid. These are some of the most common causes of dangling pointers:

- Once we call `delete` on a pointer, it becomes a dangling pointer.
- An uninitialized pointer is dangling.
- If a pointer stores a stack address in a certain stack frame and the function owning that frame returns, the pointer becomes a dangling pointer.

```
int* foo() {
  int x = 5;
  return &x;
}

void invalidReadWrite() {
  int array[3] = {0, 1, 2};
  array[3] = 3;  // Invalid write: index outside of array boundary

  int* p1;
  int* p2 = new int;
  delete p2;
  int* p3 = foo();

  // At this point, p1, p2, and p3 are all dangling pointers, so the following
  // three lines each contain an invalid read
  std::cout << *p1 << std::endl;  // p1 was not initialized
  std::cout << *p2 << std::endl;  // p2 has already been deleted
  std::cout << *p3 << std::endl;  // p3 points to a stack address that was freed
}
```

Dangling pointers on their own are not a problem and cannot be avoided. An error only occurs when we try to use the object at the address stored in a dangling pointer.

Invalid free

An *invalid free* occurs when a program calls `delete` with a memory address which it is not supposed to delete. This most frequently occurs for the following reasons:

- **Deleting a stack address**: Since the stack is automatically deallocated, we should never try to delete an object on the stack.
- **Deleting a dangling pointer**: If a pointer does not point to a valid object, it is always illegal to call `delete` on that pointer.
- **Double delete**: If we delete an address once, it is illegal to delete the same address again until a new object has been initialized at that address. This is a special case of deleting a dangling pointer.

```
void invalidFree() {
  int x = 0;
  int* p1 = &x;
  int* p2;
  int* p3 = new int;
  int* p4 = p3;
  delete p3;

  delete p1;  // Illegal free: attempts to delete a stack address
  delete p2;  // Illegal free: attempts to delete an uninitialized pointer
  delete p4;  // Illegal free: double delete
}
```

Use of uninitialized values

An object which has been allocated but not initialized will take on whatever value was at that address previously. Since we do not know the contents of the stack or heap when we begin running a program, there is no way of predicting these uninitialized values. It is alright for a variable to store an uninitialized value, but as soon as we try to use that value (such as in a conditional), it is undefined behavior.

```
void uninitializedValues() {
  int x;
  int y = x;  // This is okay since we have not used x or y yet

  if (x < 0) {  // This is not okay
    std::cout << "negative" << std::endl;
  }

  std::cout << y << std::endl;  // This is not okay
}
```

Detecting Memory Errors

Drawing a memory diagram can be very helpful for detecting memory errors. As we fill in the diagram, we will likely notice if we attempt to perform any strange behavior such as delete an address twice or use an uninitialized `(?)` value.

Valgrind is an excellent debugging tool for identifying several types of memory errors. Suppose that we are in a directory with the executable `program`. Then, the command `valgrind --leak-check=full ./program` will run program with Valgrind. This has the same effect as running `program` on its own, but also identifies and prints significant debugging about any memory errors (including the line number of the error). If a program ever has a segfault, one of the best first steps is to run the program in Valgrind to identify the line number at which the segfault occurs.

Possible Exam-style Questions

1. Given some code, draw a memory diagram at different points during the program's execution.
2. Use a memory diagram to justify why one program is more memory efficient than another.
3. Use a memory diagram to reason about whether a parameter should be passed or returned by reference.
4. Given some code with several memory errors, identify and fix each error.
5. Given a class, write a destructor which prevents memory leaks.

Suggested Exercises

Exercise 1

Draw a memory diagram for each example function from the object lifetime case study on the supporting website. Update this diagram as each line is executed.

Exercise 2

Write a program with the following features:

- Uses both the stack and the heap.
- Includes at least one function call.
- Uses at least one object with multiple data members.
- Includes at least one memory error.

Draw a memory diagram for your program and update it line by line as though you are executing the program. Attempt to identify your memory error with your memory diagram.

1.3 COMPILATION

A computer can only execute *machine code*, which refers to hardware instructions encoded in binary. Thus, before we can execute a program written in C++, we must first *compile* the program by translating the C++ code into machine code. Understanding the compilation process will help us write code which successfully compiles and better diagnose compilation-related issues.

Preprocessor

Before the main stage of compilation, the C++ *preprocessor* passes through the code to resolve preprocessor directives. A *preprocessor directive* is a line of code beginning with the # symbol, such as `#define MAKE_SHEEP_HPP_` or `#include <std::string>`. Many preprocessor directives have fallen out of favor in modern C++ programming, but the following two are still heavily used today.

Include

The `#include` preprocessor directive copies the contents of the specified file or header and pastes it at the exact location of the directive. If we wish to include a predefined *header* such as `string` or `iostream` from the standard library, we place it between angle brackets (`<>`). If we wish to include a local file such as an `.hpp` file we wrote, we place it between quotation marks (`""`).

```
#include <string>      // Includes the string header from the standard library
#include "sheep.hpp"   // Includes the sheep.hpp file written by us
```

The only difference between `<>` and `""` is the location at which the preprocessor searches for the code to include.

Header guards

On its own, `#include` has the danger of including the same code multiple times. For example, if `sheep.hpp` also included the `string` header, the above example would include `string` twice. Not only is this inefficient, it can lead to a compile time error since C++ does not allow the same class or function to be defined twice.

The standard way to solve this problem is with a *header guard*, which defines a unique *macro* associated with each `.hpp` file. A header guard consists of the following preprocessor directives. We begin the file with `#ifndef MACRO_NAME` and end the file with `#endif`. `#ifndef` is an abbreviation for "if not defined" and tells the preprocessor to only include the code between the `ifndef` and `endif` if `MACRO_NAME` has not already been defined. Next, we use the directive `#define MACRO_NAME` right after the `ifndef` to define the macro. We do not need to give the macro a value since `ifndef` only checks if the macro is defined. This has the following effect: the first time the preprocessor encounters the file, `MACRO_NAME` has not yet been defined, so it evaluates the code in the if block. This defines `MACRO_NAME` and includes the file once. If the preprocessor encounters the file again, `MACRO_NAME` has already been defined, so it does not enter the if block and skips the entire file. For this to work, every file must use a unique `MACRO_NAME`, so we traditionally name each macro based on the file path and filename.

The following code shows a header guard for `sheep.hpp`. The comment `// SHEEP_HPP_` after `#endif` is not required but is recommended to remind the reader of the if to which the `#endif` corresponds.

```
#ifndef SHEEP_HPP_
#define SHEEP_HPP_

// (Contents of sheep.hpp)

#endif  // SHEEP_HPP_
```

You can learn more about other preprocessor directives at http://www.cplusplus.com/doc/tutorial/preprocessor/.

Separate Compilation

After the preprocessor finishes preparing a file, the compiler translates the C++ code into machine code. If this file contains a `main` function and a definition corresponding to every declaration, we can compile the file into an *executable* which can be run on the command line. For larger programs, however, we prefer to use *separate compilation*, which involves compiling different files into separate pieces of machine code and *linking* these pieces into a final executable.

To understand the advantages of separate compilation, suppose that we have written a `Sheep` class which is used by several other programs. Without separate compilation, we would need to declare and define the entire `Sheep` class in `sheep.cpp` and require all users of this class to `#include sheep.cpp`. With separate compilation, we instead separate `Sheep` into two files, placing the class declaration in `sheep.hpp` and the class definition in `sheep.cpp`. Any file which uses the `Sheep` class would only `#include sheep.hpp`. During compilation, we would compile that file and and `sheep.cpp` separately, then link the machine code from these two compiltaions to create a completed executable. This provides two major advantages:

1. **Separating interface and implementation**: By design, `sheep.hpp` contains the interface for our class and `sheep.cpp` contains the implementation. Generally, users of a class are only interested in its interface—they want to know what they can do with the class without understanding how it happens. This separation also allows us to change the class implementation without changing the interface.
2. **Less compilation**: Without separate compilation, we must recompile the entire program every time we change any aspect of it. For example, if we modify a program which includes the `Sheep` class, we must recompile the `Sheep` implementation because it is `#included`, even though the `Sheep` implementation has not changed. Similarly, if we change the `Sheep` implementation, every user of the `Sheep` class must recompile their entire program. On the other hand, with separate compilation, we only need to recompile code that changed.

Since most of the code in CS70 and this textbook compiles in under a second, the compilation difference may seem trivial. However, in real world applications such as a large industry code base, recompiling an entire program can take several hours. Then, separate compilation can mean the difference between taking a few seconds rather than a few hours to compile our changes.

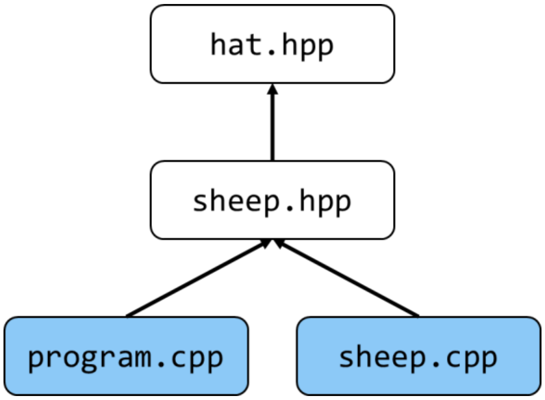
As a consequence of separate compilation, we usually should never `#include` a `.cpp` file. If we ever find the need to include a `.cpp` file, we should separate its interface into a `.hpp` file and include that instead. Note that the `.cpp` and `.hpp` file extensions only exist to help humans understand the difference between files. As far as the computer is concerned, both are text files, and it will treat them identically.

An Example of Separate Compilation

To see this concept in action, we will consider the compilation case study from the supporting website. In this example, our program consists of four files. You can view the source code for these files on the supporting website.

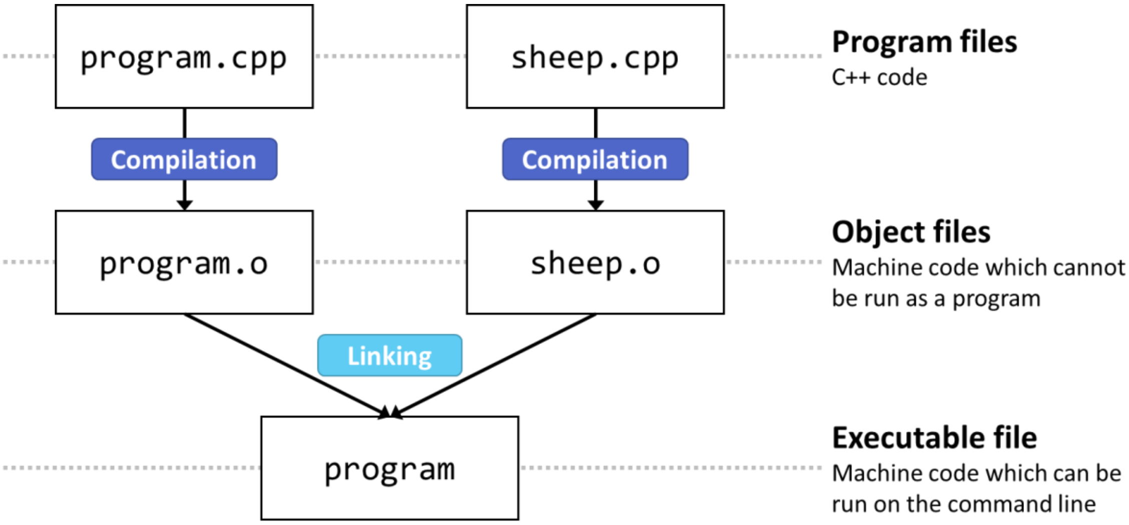
- `program.cpp`: A program (including a `main` function) which uses the `Sheep` class.
- `sheep.hpp`: Declares the interface for the `Sheep` class, which represents a sheep wearing a hat.
- `sheep.cpp`: Implements the `Sheep` class.
- `hat.hpp`: Defines the `Hat enum`, which specifies different types of hats.

The following *dependency graph* shows the `#include` structure for these files. Specifically, `sheep.hpp` includes `hat.hpp`, and `program.cpp` and `sheep.cpp` both include `sheep.hpp`. After preprocessing, a given `.cpp` file will include the contents of every file which it can reach in the dependency graph. For example, `sheep.cpp` includes the contents of both `sheep.hpp` and `hat.hpp`, even the line `#include "hat.hpp"` does not appear anywhere in `sheep.cpp`. However, because `hat.hpp` is copied into `sheep.hpp` before `sheep.hpp` is copied into `sheep.cpp`, `sheep.cpp` still receives a copy of `hat.hpp`.



A dependency graph showing the include structure of these files. The file at the source of each arrow `#includes` *the file at the destination of that arrow.*

The following compilation diagram shows the compilation and linking steps needed to create the `program` executable. First, we separately compile the two program files (ending in `.cpp`) into object files (ending in `.o`). An *object file* is a piece of machine code which is not a full program. For example, `program.o` does not have the implementation of the `Sheep` class needed to satisfy the declaration it `sheep.hpp`, and `sheep.o` does not have a `main` function. As the final step, we link these two object files into the final executable. `program` contains the implementation of the `Sheep` class from `sheep.o` and a `main` function from `program.o`, so it has all of the pieces needed to be run on the command line.



A compilation diagram showing the steps necessary to create the final executable.

If we make a change to one or more files, we must repeat each step downstream from the changed file(s) to the final executable. We will consider three examples:

1. **Change** `program.cpp`: We must recompile `program.cpp` into `program.o` and repeat the linking step. We do not need to recompile `sheep.cpp` into `sheep.o`.
2. **Change** `sheep.cpp`: We must recompile `sheep.cpp` into `sheep.o` and repeat the linking step. We do not need to recompile `program.cpp` into `program.o`.
3. **Change** `hat.hpp`: From the dependency graph shows that `program.cpp` and `sheep.cpp` both include `hat.hpp`, so changing `hat.hpp` will change both `program.cpp` and `sheep.cpp` (since the contents of `hat.hpp` are copied into `program.cpp` and `sheep.cpp` during preprocessing). Thus, we must repeat both compilation steps and the linking step.

If we were not using separate compilation, we would have needed to recompile everything in all three examples. By using separate compilation, we avoided some unnecessary compilation in the first two examples.

Linker errors

A *linker error* occurs when the linker does not have all of the pieces needed to create an executable program. Most frequently, this happens when the linked object files do not have a definition to match every declaration. For example, suppose that in `sheep.cpp`, we forgot to implement the `pet` function declared in `sheep.hpp`. This would cause a linker error during the linking stage, but it would not cause a compilation error since an object file can use a function that is declared but not defined. This issue can also occur if we forget to pass one of the object files to the linker, such as if we forget to include `sheep.o` in the linking command.

Another common linker error occurs when the linked object files do not have exactly one definition of the `main` function. An executable program must have exactly one `main` function which serves as the entry point for the program. For example, we would experience a linker error if `sheep.cpp` and `program.cpp` both defined a `main` function, or if neither defined a `main` function.

Compilers

A *compiler* is a program which performs compilation, and a *linker* is a program which performs linking. In practice, a compiler and a linker are usually packaged into a single program, which is colloquially referred to as a compiler. `clang++` and `gcc` are two of the most popular C++ compilers.

When we run a compiler from the command line, we can pass it *flags* which provide additional instructions for compilation or linking. The following flags tell the compiler to do the following things:

- `-c`: Create an object file rather than an executable program.
- `-g`: Compile for debugging, which (among other things) will allow us to see line numbers associated with errors.
- `-l<library>:` For a linking command, tells the linker to include the specified library. The `-lopencv_core` flag, for example, tells the compiler to link the opencv core library. The standard library is linked automatically.
- `-Ox`: Specifies how much the compiler should attempt to optimize the code, with `-O0` being the least optimized and `-O3` being the most optimized. These optimizations will never change code behavior.
- `-o <name>`: Name the output file with the name following the `-o` flag.
- `-std=<standard>`: Specifies which C++ standard to use. For example, `-std=c++1z` uses the C++ 2017 standard and `-std=c++11` uses the C++ 2011 standard.
- `-Wall`: Specifies certain warnings to show.
- `-Wextra`: Specifies additional warnings to show.
- `-pedantic`: Specifies even more warnings to show.
- `-Werror`: Treats warnings as errors, which means that the code will fail to compile if it creates any warnings.

For example, the following command would compile `sheep.cpp` into `sheep.o` using the `clang++` compiler. Even though we do not specify the output name, the compiler assumes that the name should be the original filename with the `.o` extension because we requested an object file with the `-c` flag.

```
clang++ -c -g -O0 -Wall -Wextra -pedantic -std=c++1z sheep.cpp
```

The following command would use `clang++` to link `program.o` and `sheep.o` to create an executable named `program`.

```
clang++ -o program -Wall -Wextra -pedantic -std=c++1z program.o sheep.o
```

Makefiles

For larger projects, it can be quite tedious to manually write and execute the necessary compilation and linking commands every time we wish to update our program. Further, we must keep track of the exact files we changed to accurately determine the necessary compilation steps.

Luckily, the *Make* tool allows us to automate the compilation process. First, we define rules in a *Makefile* which specify the compilation and linking steps associated with our program. Then, any time we need to update our program, we can run Make from the command line to automatically determine and execute the necessary commands.

Each Makefile *rule* consists of three parts:

1. **Target**: The name of the file that is created by running the command.
2. **Dependencies**: A list of other files which affect the target file. For a compilation rule, the dependency list should include the `.cpp` file being compiled and every `.hpp` file which is reachable from that `.cpp` file in the project dependency graph. For a linking step, the dependency list should include every object file being linked.
3. **Command**: A command which can be executed on the command line, such as a compilation or linking command. Usually, this command generates the target file.

The following rule specifies how to create the `sheep.o` file by compiling `sheep.cpp`. It has the dependencies `sheep.cpp`, `sheep.hpp`, and `hat.hpp` since the dependency graph shows that `sheep.hpp` and `hat.hpp` are included in `sheep.cpp`.

```
sheep.o: sheep.cpp sheep.hpp hat.hpp
  clang++ sheep.cpp -c -std=c++1z -g -Wall -Wextra -pedantic
```

To execute a particular rule, we run `make <target>` from the command line, such as `make sheep.o`. Before executing a rule, Make will first check each of the rule's dependencies with the following procedure:

- If the dependency is the target of another rule, Make first checks that the dependency is up to date by recursively using this procedure evaluate and potentially execute that rule.
- If the dependency is not the target of a rule but does exist, Make determines if the dependency has changed since the last time this rule was executed.
- If the dependency is not the target of a rule and does not exist, Make will create an error, since it does not have all of the pieces necessary to execute this rule.

After running these checks, Make will only execute the command if the target does not exist or if one or more of the dependencies have changed since the last time the rule was executed (meaning the target is out of date). This ensures that we perform the minimum number of compilation steps, which is an important part of maximizing efficiency through separate compilation. However, if we do not provide the correct dependency list for each target, Make may skip important steps that it does not realize are necessary.

To illustrate this process, consider this Makefile for our running example.

```
# Create the program executable by linking program.o and sheep.o
program: program.o sheep.o
  clang++ -o program program.o sheep.o -std=c++1z -g -Wall -Wextra -pedantic

# Create program.o by compiling program.cpp
program.o: program.cpp sheep.hpp hat.hpp
  clang++ program.cpp -c -std=c++1z -g -Wall -Wextra -pedantic

# Create sheep.o by compiling sheep.cpp
sheep.o: sheep.cpp sheep.hpp hat.hpp
  clang++ sheep.cpp -c -std=c++1z -g -Wall -Wextra -pedantic
```

If we run `make program`, Make will begin by evaluating the `program.o` and `sheep.o` dependencies, each of which have a corresponding rule. Thus, Make will recursively check both of those rules and execute them if any of their dependency files have changed. If either of the rules are executed, Make will then execute the `program` rule to create an updated version of the `program` executable.

Additional Capabilities

Beyond this core functionality, Make provides several additional capabilities which allow us to create more powerful and efficient Makefiles. First, the target of a rule does not necessarily have to be a file created by the rule; we can also use a *phony target* to specify commands that do not create a single file. It is idiomatic to include rules for the following two phony targets:

1. `all`: This rule has no command and its dependency list includes every executable file associated with the project. Thus, we can run `make all` at any time to update all of the executable files of the project.
2. `clean`: This rule has no dependencies and its command removes all generated files (such as object files and executable files). Thus, we can run `make clean` at any time to remove all of the files which we can easily generate again if needed.

If we run `make` without any arguments, it evaluates the first rule by default. It is therefore idiomatic to place the `all` rule first so that the command `make` updates all executable files (the most common command).

We can also declare variables in a Makefile with the syntax `VAR_NAME = value` and use the variable with the syntax `$(VAR_NAME)`. It is idiomatic to include the following four variables:

1. `CXX`: The compiler to use, such as `clang++`.
2. `CXXFLAGS`: The compiler flags to use, such as `-g -std=c++1z -Wall -Wextra -pedantic`
3. `TARGET`: All executable programs which can be created by the Makefile. Thus, `$(TARGET)` should be the dependency of the `all` rule.
4. `LIBRARIES`: All non-standard libraries which must be specified in the linking step, if relevant.

These variables factor out duplicate code from commands which allows us to make changes in a single place. For example, if we decided to change the compiler or add an additional compiler flag, we only need to make this change in one place rather than in every command.

Finally, Make provides several special macros which can be used in commands to reference portions of the target or dependency list. The following three are the most important for basic compilation and linking commands.

- `$@`: The target name. This is helpful for specifying the output filename of a linking command.
- `$<`: The name of the first dependency. This is helpful for specifying the file to compile in a compilation command (as long as we always list this file as the first dependency).
- `$^`: The name of all dependencies, with duplicates removed. This is helpful for specifying the list of object files to link in a linking command.

With these in mind, we can improve our Makefile as follows. Notice that we do not include the `LIBRARY` variable because our program does not use any non-standard libraries.

```
# Declare variables
CXX = clang++
CXXFLAGS = -g -std=c++1z -Wall -Wextra -pedantic
TARGET = program

# Create the target executable
all: $(TARGET)

# Create the program executable by linking program.o and sheep.o
program: program.o sheep.o
  $(CXX) -o $@ $^ $(CXXFLAGS)

# Create program.o by compiling program.cpp
program.o: program.cpp sheep.hpp hat.hpp
  $(CXX) $< -c $(CXXFLAGS)
```

```
# Create sheep.o by compiling sheep.cpp
sheep.o: sheep.cpp sheep.hpp hat.hpp
  $(CXX) $< -c $(CXXFLAGS)

# Remove all generated files (all object files and executable files)
clean:
  rm -rf *.o $(TARGET)
```

To see this Makefile in action, visit the compilation case study on the supporting website.

Possible Exam-style Questions

1. Given several files that include each other in different ways:
   a. Create a dependency graph.
   b. Create a compilation diagram.
   c. Create a Makefile with the proper dependencies.
   d. Reason about which compilation and linking steps must be run after a given file is changed.
2. Given code and a Makefile that creates a linker error, identify and fix the source of the error.
3. Given code and a Makefile with incorrect dependencies, identify the effect of these mistakes and fix them.
4. Create a header guard for a file and explain why it is necessary.
5. Explain the advantages of separate compilation.
6. Explain the difference between `.cpp` and `.hpp` files.

Suggested Exercises

Exercise 1

Complete the exercises and answer the reflection questions from the compilation case study on the supporting website.

Exercise 2

Create several simple `.cpp` and `.hpp` files that include each other in different ways. Create a Makefile that compiles and links these files into a single executable. Make changes to individual files and see which compilation and linking steps are necessary to update the executable.

1.4 INPUT AND OUTPUT

For a program to interact with the real world, it often must take *input* from or write *output* to external sources such as a file or the terminal. In this section, we will learn how to handle different forms of input and output (IO) using the C++ standard library.

Streams

The C++ standard library implements input and output with *streams*, which represent an IO source as an infinite collection of characters. Some of the most common types of streams include:

- `ostream`: An output stream representing a destination to which we can send characters with `operator<<`.
- `istream`: An input stream representing a source from which we can read characters with `operator>>`.
- `iostream`: An input/output stream which supports both `ostream` and `istream` operations.
- `ofstream`: A type of `ostream` which writes to a specified file.
- `ifstream`: A type of `istream` which reads from a specified file.
- `stringstream`: A type of `iostream` which can easily be converted to and from an `std::string`.
- `cout`: A specific `ostream` which writes to standard output, the default place to show information to the user.
- `cin`: A specific `istream` which reads from standard input, the default place to read information from the user.
- `cerr`: A specific `ostream` which writes to standard error, the default place to show errors to the user.

The following program shows how to use different types of streams. `std::endl` denotes the newline character for the local operating system (`\n` for Linux and `\r\n` for Windows).

```
int main() {
  // Write a message to the command line ending in a newline character
  std::cout << "Please enter a filename:" << std::endl;

  // Read a string from the command line
  std::string filename;
  std::cin >> filename;

  std::cout << "Enter a number of repetitions" << std::endl;

  // Read a size_t from the command line
  size_t reps;
  std::cin >> reps;

  // Open an output file stream to the file provided by the user
  std::ofstream file;
  file.open(filename);

  for (size_t i = 0; i < reps; ++i) {
    // Write a line to the end of the file
    file << i << " hello world!" << std::endl;
  }

  // Close the output file stream
  file.close();

  return 0;
}
```

Standard Input and Output

From the previous example, `std::cout`, `std::cin`, and `std::cerr` represent three of the most important streams. A program should show information to the user by writing to *standard output* (`std::cout`), which by default prints the information to the console window. If a program experiences an error or notices the user attempting to perform a bad action, it should write a message to *standard error* (`std::cerr`), which by default is also printed to the console window. A program should take input from the user by reading from *standard input* (`std::cin`), which by default reads the information typed into the console window.

In bash or an equivalent command-line shell, we can redirect any or all of these streams to other locations such as files with the following notation:

- `> out.txt`: redirects standard output to the file `out.txt`, creating a new file if one does not already exist.
- `2> error.txt`: redirects standard error to the file `error.txt`, creating a new file if one does not already exist.
- `< in.txt`: redirects the contents of the file `in.txt` to standard input (`in.txt` must already exist).

For example, suppose that the executable `program` reads data from standard input and writes data to standard output and standard error. The following command would give `program` input from `in.txt`, send its output to `out.txt`, and send any error messages to `error.txt`.

```
./program < in.txt > out.txt 2> error.txt
```

Redirection can be a useful tool in several of situations. For example, if we need to repeatedly run a program with the same input, we can save the input to a file and redirect standard input to that file to avoid typing the input every time. If a program has a large output, we can redirect standard output to a file to make it easier to navigate. If we want to store a program's output or error messages for later, we can redirect standard output or error to a file.

Implementing Print and Operator<<

When designing a class, we may wish to include a `print` method which writes a character representation of an instance of the class to an `ostream`. This method should take in an `ostream` reference and return a reference to the same `ostream`. We can think of the `ostream` as a "clipboard"—the object receives a clipboard from the user, writes itself to the clipboard, and returns it back to the user. Here is an example `print` method for the `Sheep` class.

```
std::ostream& Sheep::print(std::ostream& os) const {
  // Write a character representation of the Sheep to the ostream
  os << "Name: " << name_ << ", Age: " << age_;

  // Return a reference to the same ostream which was passed in
  return os;
}
```

To have the desired behavior, `print` must take and return the `ostream` by reference. If the `ostream` was passed by value, `print` would create a local copy in its stack frame and add characters to that local copy without modifying the original. The user's `ostream` object would not receive the new characters. We always declare `print` as a `const` method because it should not modify the object in the process of printing it.

The global `operator<<` function can also be used to write an object to an `ostream`. It takes two arguments: the first argument (corresponding to the object on the left side of the `<<` symbol) is a reference to the `ostream` to which to write, and the second argument (corresponding to the object on the right side of the `<<` symbol) is a `const` reference to the object to print. Whenever the compiler sees the pattern `ostream << object`, it will call the overload of `operator<<` which matches the type of `object` (if such an overload exists). Note that while `print` is a method belonging to a class, `operator<<` is a global function which does not belong to any class.

Once we have written a `print` method for a class, we can easily overload `operator<<` simply by calling `print` on the object. For example, we could overload `operator<<` for `Sheep` as follows:

```
std::ostream& operator<<(std::ostream& os, const Sheep& sheep) {
  return sheep.print(os);  // Leverage the existing Sheep::print method
}
```

With this overload, we can now write a `Sheep` object to standard output as follows.

```
Sheep shawn;
std::cout << shawn << std::endl;
```

Possible Exam-style Questions

1. Write a small piece of code which reads and writes from standard input and output.
2. Write a small piece of code which reads and writes from a file.
3. Given some code which uses streams, reason about the behavior of the code and identify any issues.
4. Provide a command which redirects standard input, output, and/or error for a program as instructed.
5. Given a class, implement a `print` method and overload `operator<<` to leverage this `print` method.

Suggested Exercises

Exercise 1

Complete the exercises and answer the reflection questions from the input and output case study on the supporting website.

Exercise 2

Write a program which uses `std::cout`, `std::cin`, and `std::ifstream` to read the contents of a file and write it to the command line. The user should specify the filename on the command line.

Exercise 3

Find a class that you previously wrote and implement a `print` method for it. Overload `operator<<` to call this `print` method.

Glossary

DEFINITIONS

2-3-4 tree: A non-binary search tree which enforces perfect balance by using three sizes of nodes.
Abstract data type (ADT): A public interface which defines the behavior of a data structure without specifying how it is implemented. For example, the stack and priority queue are abstract data types.
Address: An integer representing a location in memory.
Allocation: When memory is set aside in the stack or heap for an object.
Amortized analysis: A method for analyzing the total run time of a series of operations when the run time of each operation is structured but variable.
Approximate theorem: A mathematic function specifying roughly how many times a cost metric is run based on the size of the input.
Argument: A value passed to a function during a function call. For example, 1 and `"hello"` are the two arguments passed to `foo` in the line `foo(1, "hello");`.
Array: A linear data structure consisting of a fixed number of objects stored contiguously in memory.
Asymptotic complexity: A method of comparing the run time of a program to a comparison function which ignores constant factors and only considers arbitrarily large input sizes.
AVL tree: A self-balancing BST which enforces balance rules that are stricter than a red-black tree.
Base class: If class B inherits from class A, then A is the base class.
Best-case run time: The shortest possible run time of a piece of code for a given input size.
Binary search tree (BST): A binary tree in which each node's left child and all of its descendants have smaller values and each node's right child and all of its descendants have larger values.
Binary tree: A tree in which every node has at most two children.
Breadth-first traversal: Visiting nodes in a graph or tree ordered by increasing distance from the origin.
Class: A definition of a type specifying the data stored in memory and the methods which can operate on the type.
Collision resolution: The process of addressing when multiple keys in a hash set/table map to the same bucket.
Compilation: The process of translating human-readable code such as a C++ program to machine code.
Compiler: A program which performs compilation and linking.
Complete tree: The most possibly balanced tree obtained by filling nodes row by row from left to right. In a complete tree, the depth of every leaf is within one of the depth of every other leaf.
Complexity class: A method of broadly categorizing problems based on how the run time grows as a function of input size. For example, P, NP, and NP-hard are complexity classes.
Constructor: A special method used to create an object.
Conversion: Any type transformation which is not a promotion.
Copy constructor: A constructor which takes a constant reference to an object of the same type as the only argument.
Cost metric: A numeric aspect of a program which is used as a proxy for the run time of the program. For example, one cost metric might be the total number of comparisons made between two objects of a certain type.
Data member: A variable declared in a class which is stored inside of instances of that class. An object is stored as a collection of its data members.
Deallocation: Freeing the memory associated with an object so it can be used again.
Decidability: Whether a problem can ever be solved, even with unlimited time.
Deep copy: A copy of an object which also creates a copy of all external data associated with the object.
Default constructor: A constructor without any parameters.
Depth (tree): The number of edges between the root and a given node in a tree.
Depth-first traversal: Visiting nodes in a graph or tree by following a path until it reaches a dead end and back tracking.
Deque: Also known as a "double ended queue", a linear ADT which supports both push and pop from the front and back.
Derived class: If class B inherits from class A, then B is the derived class.
Destruction: The process of freeing all external memory associated with an object before deallocating it.

Destructor: A special method called on an object during its destruction to clean up any external data associated with the object before deallocating it.
Doubly-linked list: A linked list in which every node has a pointer to the next node and the previous node.
Dynamic dispatch: The process of deciding which version of an inherited method to use at run time.
Dynamic: Refers to something that must be handled at run time such as dynamic dispatch or a dynamic array.
Encoding: The data members of a class, which specify how the object is stored in memory.
Enumerated type (enum): A type whose values are special identifiers corresponding to integer values. For example, `enum color {red, blue, green};` creates a new type `color` with values `red` (corresponding to 0), `blue` (corresponding to 1), and `green` (corresponding to 2).
Exact theorem: A mathematic function specifying the exact number of times a cost metric is run based on the size of the input.
Executable: A program (consisting of machine code) which can be run by the operating system.
Expected run time: The weighted average of all possible run times of a piece of code for a given input size.
Explicit transformation: A type transformation directly requested by the programmer through a typecast.
Extendible array: An array which increases its size by a multiplicative factor (such as doubling its size) every time it is full.
Floating point type: A primitive type such as `float` or `double` which stores real (ie decimal) numbers.
Function overloading: The process of defining multiple functions with the same name.
Function: A piece of code that can be called from elsewhere in a program. A function can take zero or more arguments from the caller and will return zero or one value to the caller.
Hash function: A function which takes an object of a certain type and returns a corresponding integer number.
Hash set: A data structure which stores elements in an array based on the index given by a hash function.
Hash table: A data structure which stores key-value pairs in an array based on the index given by a hash function on the key.
Heap (data structure): An implementation of the priority queue ADT which organizes data in a complete tree.
Heap (memory): A area of memory in which the programmer explicitly controls object allocation and deallocation.
Height (tree): The depth of the deepest leaf in a tree. A one-element tree therefore has a height of 0, and we define an empty tree to have a height of -1.
Identifier: A name given by the programmer to something such as a variable, function, class, etc.
Implementation: The method definitions of a class specifying how each method works.
Implicit transformation: A type transformation performed by the compiler as needed without being explicitly requested by the programmer.
Inheritance: The process of connecting a derived class and a base class such that the derived class automatically contains all of the data members and methods of the base class.
Initialization: The process of giving starting value(s) to the data of an object.
Inside-out rule: States that we should read a long type starting at the variable name (the inside) and working outward.
Instance: An object of a particular class. For example, from the line `Sheep shawn;`, we say that `shawn` is an instance of the `Sheep` class.
Integral type: A primitive type such as `short` or `int` which stores integer numbers.
Interface: The public methods of a class, which specify what can be done with objects of that class.
Iterator: A special type of class used to move through the elements of a data structure in order.
Keyword: A word with special meaning to a programming language, such as `int`, `const`, or `while` in C++.
Leaf: A node in a tree without any children.
Linear data structure: A data structure in which elements are stored in an ordered line such as an array or linked list.
Linked list: A linear data structure consisting of a line of nodes connected by pointers.
Linking: The process of connecting multiple pieces of machine code into a single executable.
Load factor: The "fullness" of a hash set/table measured as the number of elements divided by the number of buckets.
Machine code: Instructions written in binary which can be directly read by computer hardware.
Makefile: A special file consisting of rules for executing certain commands on the console.
Member function: See method.
Member initialization list: The portion of a constructor which specifies the argument(s) to pass to the constructor of each data member during initialization.
Method: A special type of function that is associated with an object. The object is implicitly passed to the function and can be accessed with the `this` keyword.

Node: A piece of a data structure consisting of a stored object and pointer(s) to other nodes. Trees and linked lists both use nodes. Unlike in an array, multiple nodes are not necessarily stored contiguously in memory.
Object file: A file containing a piece of machine code which is insufficient to create a complete program.
Object: In the context of C++, a collection of memory and a way to reason about the data stored in that memory. Both a primitive variable like an `int` and an instance of a user defined class are objects.
Open addressing: A method of hash set/table collision resolution which places at most one element per bucket and has a structured method for finding other buckets if the desired bucket is occupied.
Parameter: A variable defined in a function header that is initialized with an argument to the function. For example, in the function `foo(int x)`, `x` is a parameter of type `int`.
Parameterized constructor: A constructor with one or more parameters.
Perfect hash: A hash function that maps a domain of n values to the integers 0 through $n - 1$, with exactly one value per number.
Perfect tree: A tree in which every row is completely filled and thus every leaf has the same depth. A perfect tree is a special case of a complete tree.
Pointer: A special object which stores the memory address of another object.
Polymorphism: The idea that a derived class will always support the complete interface of its base class, meaning that a derived class can always be used in place of its base class.
Primitive type: Special types provided by C++ to store simple types of data, such as `int`, `double`, and `bool`.
Priority queue: An abstract data type which provides access to the smallest (or largest) element in the data structure.
Promotion: One of a collection of special type transformations which are preferred over conversions. While all promotions guarantee no data loss, not all type transformations that guarantee no data loss are promotions.
Queue: A linear ADT which supports the `push_back` and `pop_front` interfaces.
Random tree: A binary search tree in which the values are randomized before being inserted, decreasing the probability of a poorly balanced tree.
Randomized tree: A self-balancing BST which randomly chooses the level at which to insert new elements.
Red-black tree: A self-balancing BST which enforces balance rules that are equivalent to a 2-3-4 tree.
Reference: An alias to (another name for) an object which already exists.
Rotation (tree): A method of moving the nodes in a tree which maintains the BST requirement while switching the depths of a parent and one of its children.
Rule of three: The idea that if we manually implement an object's copy constructor, assignment operator, or destructor, we should manually implement all three methods.
Self-balancing BST: A binary search tree which automatically takes steps to maintain balance to help prevent one side from becoming far longer than the other.
Separate chaining: A method of hash set/table collision resolution which encodes each bucket as a linked list, allowing multiple elements to occupy the same bucket.
Shallow copy: A copy of an object with pointers to the external data of the original object rather than its own copies of the external data.
Singly-linked list: A linked list in which every node has a pointer to the next node but not a pointer to the previous node.
Splay tree: A self-balancing BST which performs splay rotations when possible and moves every inserted and looked up value to the root through rotations.
Stack (memory): A highly organized portion of memory used to store the data used by functions. The programmer cannot directly control when memory is allocated or deallocated from the stack.
Stack: A linear ADT which supports the `push_front` and `pop_front` interfaces.
Standard input: An input stream which defaults to the text the user types on the command line.
Standard output: An output stream which is by default printed to the command line.
Static dispatch: The process of deciding which version of an inherited method to use at compile time.
Static: Refers to something that must be handled at compile time such as a static array or static dispatch.
Steque: A linear ADT which combines the stack and queue interfaces to support `push_front`, `push_back`, and `pop_front`.
Stream: A flow of data. Depending on the type of stream, it may be able to send data, receive data, or both.
Summation: A mathematic symbol use to represent repeated addition. For example, $\sum_{n=1}^{5} 2n = 2 + 4 + 8 + 10$.
Synthesized method: A method automatically generated by the compiler. For example, the copy constructor and destructor (among others) are automatically synthesized if they are not specified.

Syntax: Rules dictating how the words and symbols in a programming language can be organized.
Syntactic sugar: A syntax of a programming language which makes code easier to read without adding additional functionality. For example, the `auto` keyword is syntactic sugar because the programmer could always write out the explicit type instead.
Tree: A data structure which organizes nodes into a connected, acyclic graph. In other words, nodes are organized into parent-child relationships with the potential for multiple children per node.
Type transformation: The process of changing an object from one type to another.
Type: A category of objects.
Undefined behavior: Code that can compile without error and can have any behavior at run time (such as crashing the program, corrupting data, etc.). A program should never include undefined behavior.
Use phase: Anything that happens to an object between the initialization and destruction phase.
Variable: An object labeled with an identifier.
Variable substitution: A method of representing the run time of a for loop as a summation, which can be helpful in determining exact theorem.
Vector: The C++ standard library implementation of the extendible array data structure.
Worst-case run time: The longest possible run time of a piece of code for a given input size.

C++ KEYWORDS

The following list defines some of the common keywords in C++. You can view a complete list of C++ keywords at https://en.cppreference.com/w/cpp/keyword.

Auto: Used in place of an explicit typename to tell the compiler to deduce the type at compile time. The type is still static and cannot change after declaration; `auto` does *not* enable dynamic typing (so is not like `var` in JavaScript).
Bool: A primitive type which can either be `true` or `false`.
Break: Used in a for or while loop to cause the loop to terminate at that point.
Char: A primitive type representing a single character (usually 8 bits).
Class: Allows the user to define a new type consisting of data members and methods.
Const: When applied to a variable, specifies that the variable cannot change. When applied to a method, specifies that the method cannot change the data members of the class.
Continue: Used in a for or while loop to skip from the current location to the beginning of the next iteration of the loop.
Default: Used in a method declaration to explicitly tell the compiler to synthesize the method.
Delete: When applied to a pointer, tells the compiler to free the memory at the location stored in the pointer. When applied to a method declaration, explicitly tells the compiler not to synthesize the method.
Double: A primitive type representing a double-precision floating point number (usually 64 bits).
Else: Defines a block of code after an if statement that is run when the condition of the if statement evaluates to `false`.
Enum: Defines a type whose values are a collection of identifiers which each correspond to an integer value.
Explicit: Placed before a single-parameter parameterized constructor to prevent the constructor from being used for implicit type transformations.
False: One of the two possible values of a `bool` which promotes to the `int` 0.
Float: A primitive type representing a single-precision floating point number (usually 32 bits).
For: Declares a loop that is similar to a `while` loop but also declares of one or more variables of the same type and one or more expressions which are executed after each iteration of the loop body.
Friend: Used in a class declaration to give another class access to the private elements of the class.
If: Used with a condition to define a block of code which is run once if the condition evaluates to `true`.
Inline: Placed before a function or method to encourage (but not require) the compiler to directly copy the assembly instructions of the function at each location it is called.
Int: A primitive type representing an integer value (usually 32 bits).
Long: A primitive type representing an integer value (usually 64 bits).
Namespace: Used to declare a namespace block, which allows the compiler to distinguish between identically named identifiers in other namespaces.
New: Used to allocate an object on the heap.

Nullptr: Encodes the memory address 0, which is used to represent something that does not exist.

Operator: A special method or function called implicitly by the use of a symbol. For example, `a == b` calls `operator==`.

Override: Tells the compiler that a method in a derived class overrides a virtual method of the same name in the base class. Technically, override is an "identifier with special meaning", not a keyword.

Private: Labels methods and data members in a class which can only be accessed by that class and its friends.

Protected: Labels methods and data members in a class which can only be accessed by that class, its friends, and any derived classes.

Public: Labels methods and data members in a class which can be accessed by everyone.

Return: Exits a function or method and returns the value directly after the `return` keyword (if a value is given).

Short: A primitive type representing an integer value (usually 16 bits).

Signed: When placed before an integral type (such as `signed int`), the type can store negative numbers.

Static: When placed before a data member in a class, the compiler creates exactly one instance of that data member shared by all instances of the class. When placed before a method in a class, the method can no longer access `this`.

Struct: A class in which all members are public.

Template: Used before a function or class template to indicate a template parameter.

This: A pointer to the object on which a method is called.

True: One of the two possible values of a `bool` which promotes to the `int` 1.

Typename: Specifies that the following identifier represents a type.

Unsigned: When placed before an integral type (such as `unsigned int`), the type cannot store negative numbers.

Using: Among other things, can declare an identifier associated with a class. For example, a data structure may include the using-declaration `using iterator = Iterator` so that users of the class can access its iterator.

Virtual: Placed before a method to specify that the method should use dynamic dispatch.

Void: Used as the return type of functions which do not return anything.

While: Used with a condition to define a block of code which is continuously executed as long as the condition evaluates to `true`.