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 run time 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 our 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 *dynamically allocated 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 dynamically allocated 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 dynamically allocated 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**: Default constructs each data members.
- **Synthesized copy constructor:** Copy constructs each data members, making it a copy of the corresponding data member in the parameter object.
- **Synthesized destructor**: Calls the destructor on all data members.
- **Synthesized assignment operator**: Calls the assignment operator on each data member, making it a copy of the corresponding data member in 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, we 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 dynamically allocated 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 dynamically allocated 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 bracket 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 run time and may place them wherever it sees fit. Dynamically allocated 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

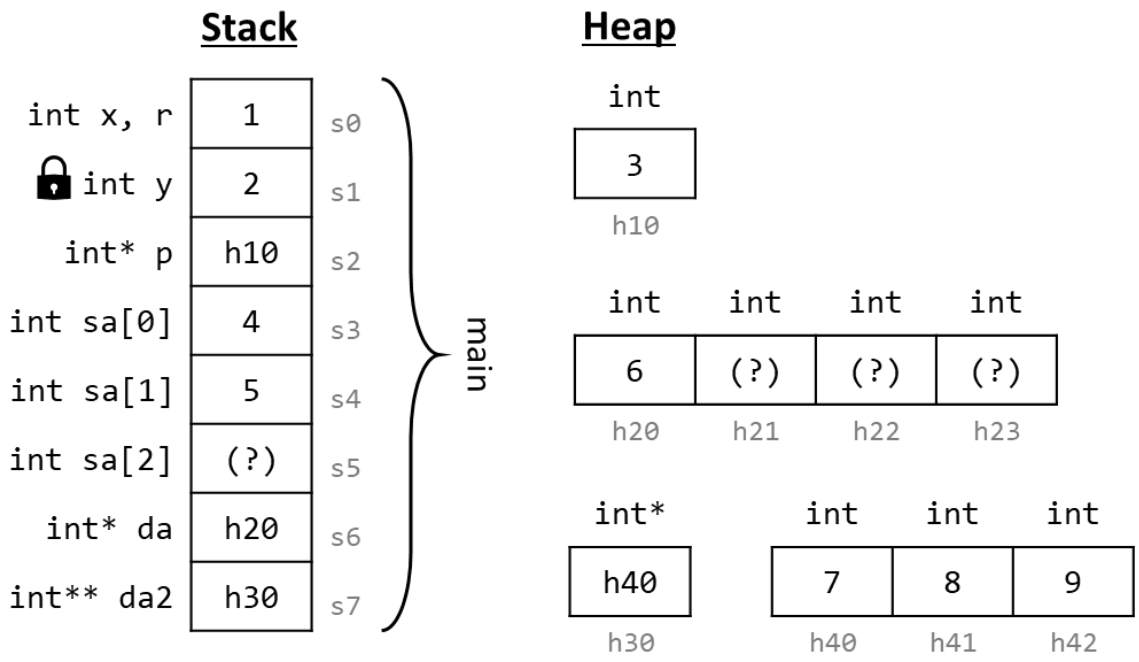
The following examples demonstrate how to use these rules to create a memory diagram for three simple programs.

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.

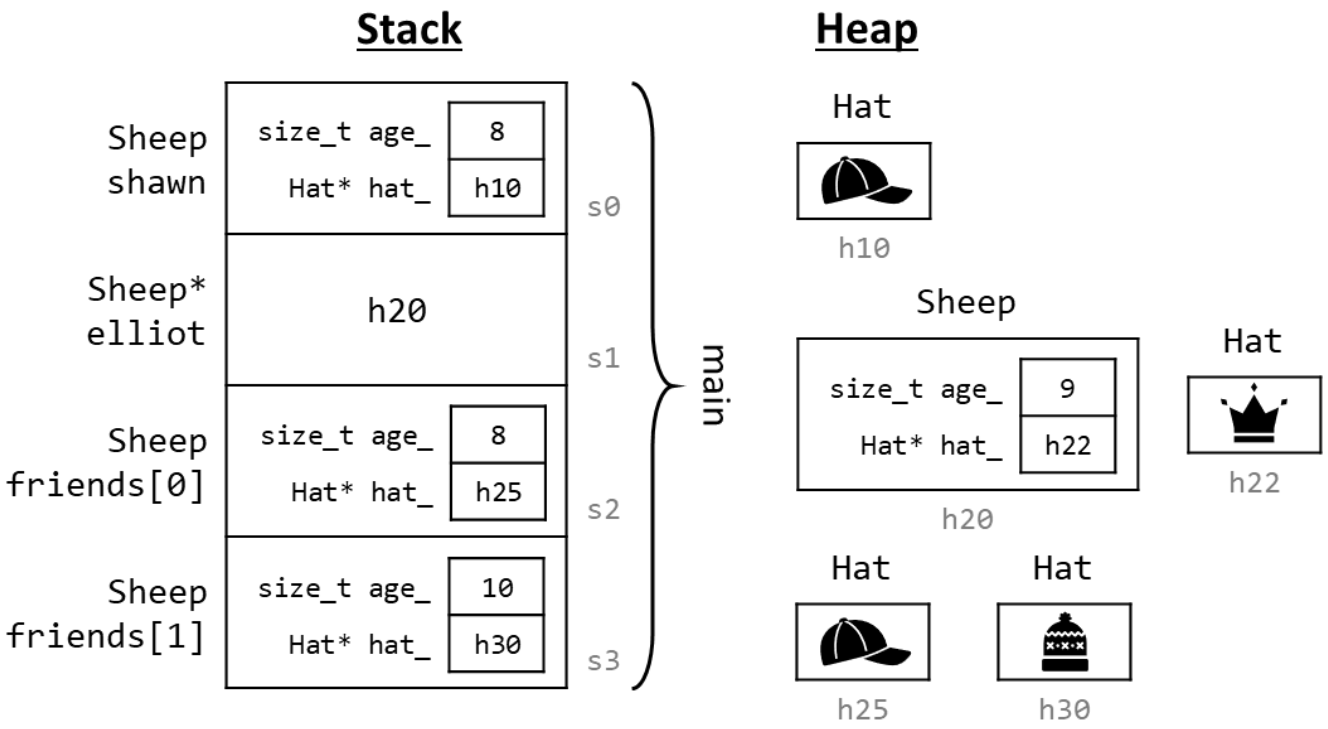


Example 2: Classes

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`.

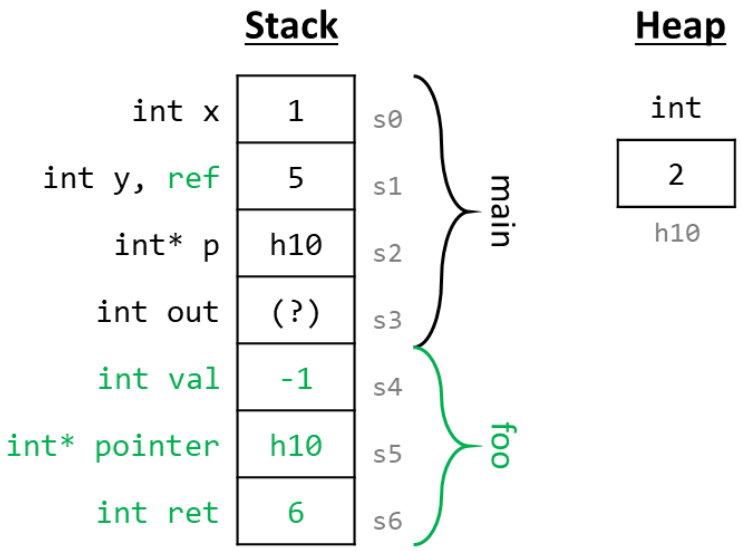


Example 3: Function Calls

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 brackets 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 `int`s 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 `int`s 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.

Segmentation Fault

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* (commonly referred to as a 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.

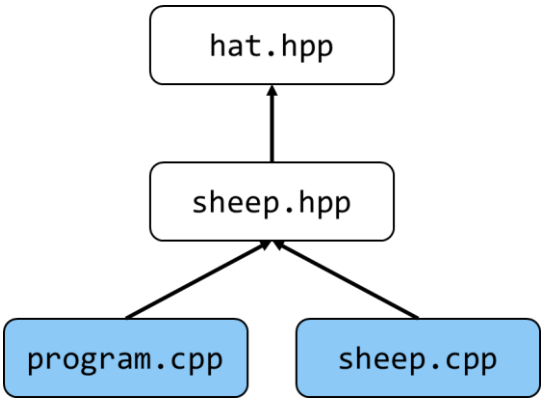
As a consequence of separate compilation, we usually should never `#include` a `.cpp` file. If we ever find the need to include a `.cpp` file, we should separate its interface into a `.hpp` file and include that instead. Note that the `.cpp` and `.hpp` file extensions only exist to help humans understand 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.

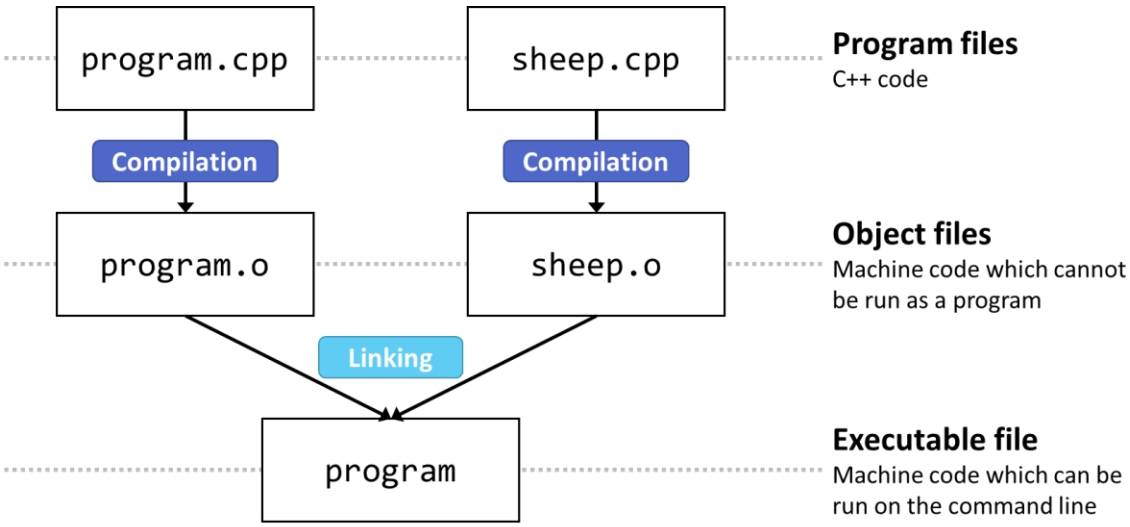
- `program.cpp`: A program (including a `main` function) which uses the `Sheep` class.
- `sheep.hpp`: Declares the interface for the `Sheep` class, which represents a sheep wearing a hat.
- `sheep.cpp`: Implements the `Sheep` class.
- `hat.hpp`: Defines the `Hat` `enum`, which specifies different types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

By default, `operator>>` will only read one word of input at a time, so the previous example would not have read the filename correctly if it contained a space. We can use the `std::getline(istream& is, string& str)` function instead, which reads an entire line of input from the provided `istream` and saves it in the provided `std::string`. For example, we could use `std::getline` to update our previous example to read filenames with spaces.

```
std::string filename;
std::getline(std::cin, filename);
```

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.

1.5 TYPE TRANSFORMATION AND FUNCTION OVERLOADING

Since C++ is a *strongly typed* language, every object must have a definitive type which remains constant throughout the lifetime of the object. If we need to use an object as a different type, we can perform a type transformation to create a new object with the desired type. In many cases, C++ will perform these transformations automatically, so developing a solid grasp of type transformations can help us better understand program behavior.

Type Transformations

In this text, we will use the term *transformation*[3] to refer to the process of changing a variable from one type to another, such as changing an `int` to a `double`. The C++ standard has labeled a small number of special transformations as *promotions*. Certain cases of transforming a smaller integral type to a larger integral type (such as a `bool`, `signed/unsigned char`, or `signed/unsigned short` to an `int`) qualify as *integral promotions*[4]. The only *floating-point promotion* is the transformation from a `float` to a `double`. Any transformation that is not a promotion is a *conversion*. As we will discuss shortly, the distinction between promotion and conversion is important because many operations prefer promotions over conversions when possible.

In all promotions, the destination type can accommodate all values of the source type, which guarantees that no data is lost. For example, any value stored as a `float` (32 bits) can also be expressed as a `double` (64 bits), so no data can be lost when promoting a `float` to a `double`. **However, just because a transformation has no potential of data loss does not make it a promotion**. For example, `char` to `short`, `short` to `long`, and `int` to `double` are all conversions even though the destination type can represent all values from the source type.

Explicit Transformation

An *explicit transformation* is one that the programmer directly requests through a typecast. The safest way to request a typecast in C++ is with `static_cast`. We can also request a typecast by placing the desired type in parenthesis before the object to cast, but this method is less preferred because it can cause more dangerous types of typecasts.

```
int i1 = static_cast<int>(99.9);  // C++ style typecast from double to int (preferred)
int i2 = (int)96.5;               // C-style typecast from double to int (less preferred)
```

Implicit Transformation

An *implicit transformation* is one performed by the complier without being directly requested by the programmer. This occurs when the compiler finds that a different type is needed so automatically performs the necessary transformation.

```
int a = true;         // Implicitly promote true (a bool) to an int
float b = a + 15UL;   // Implicitly convert a (an int) to an unsigned long,
                      // then implicitly convert the result to a float
```

All binary operators such as `+`, `-`, `*`, `==`, and `<` must accept two arguments of the same type. If a binary operator is given two arguments with different types, the compiler will implicitly transform one of the arguments to match the type of the other. To choose which type to transform, the compiler applies the following rules in the following order.

Rule	Example	Result
1. Favor promotions over conversions.	`short + int`	promote `short` to an `int`
2. Favor conversions which cannot lose data.	`int * long`	convert `int` to a `long`
3. Favor floating-point types over integral types.	`int - float`	convert `int` to a `float`
4. Favor unsigned over signed integral types.	`int < unsigned int`	convert `int` to an `unsigned int`

[3] In most literature, the term "conversion" encompasses both type conversions and type promotions. Although it is less standard, this textbook uses the term "transformation" to explicitly distinguish between promotions and conversions.

[4] You can view a complete list of integral promotions at https://en.cppreference.com/w/cpp/language/implicit_conversion#Numeric_promotions.

The compiler creates a warning when it performs certain implicit typecasts to warn the user that data may be lost without them realizing it. To resolve these warnings, we can make the conversion explicit with a typecast.

User-Defined Conversions

A user-defined parameterized constructor with a single parameter can automatically be used for implicit conversions, which are referred to as *user-defined conversions*. For example, suppose that the `Hat` class has the following parameterized constructor.

```
Hat(int numFeathers);
```

This would allow the compiler to perform a user-defined implicit conversion from an `int` to a `Hat`.

```
Hat tophat(0);    // Traditional use of the Hat parameterized constructor
Hat fedora = 3;   // Implicit use of the Hat parameterized constructor
```

To prevent a constructor from being used for implicit conversions, we can place the `explicit` keyword before the constructor declaration. We are not allowed to perform more than one user-defined conversion in a transformation sequence. For example, suppose that the `Sheep` class has the following parameterized constructor.

```
Sheep(const Hat& hat);
```

The transformation sequence from `int` to `Sheep` is not allowed because it involves two user-defined conversions (`int` to `Hat`, then `Hat` to `Sheep`).

```
Hat bowler = Hat(2.5);  // Implicitly convert a double to an int to a Hat (okay)
Sheep shawn = Hat(2);   // Implicitly convert a Hat to a Sheep (okay)
Sheep elliot = 1;       // This would cause a compile time error because it
                        // requires two user-defined conversions
```

You can learn more about type transformations at https://en.cppreference.com/w/cpp/language/implicit_conversion.

Function Overloading

Function overloading is the process of creating multiple functions or methods with the same name (referred to as *overloads*). Two overloads must have **at least one** of the following differences:

- A different number of parameters.
- At least one parameter with a different type.
- Different const-ness (that is, one method is `const` and the other is not).

Notably, we cannot overload a function based on its return type. When an overloaded function is called, the compiler must decide which overload to use. First, it determines all *viable overloads* by selecting each overload that has the correct number of parameters, a valid transformation sequence from each argument type to the corresponding parameter type, and the correct const-ness. If there are multiple viable overloads, the compiler ranks the transformation sequence from each argument to parameter type of each overload with the following ranking system:

1. **Exact match**: The argument type is the same as the parameter type.
2. **Promotion**: The argument type can be promoted to the parameter type.
3. **Conversion**: The argument type can be converted to the parameter type with a standard conversion sequence.
4. **User-defined conversion**: The argument type can be converted to the parameter type with a conversion sequence including a parameterized constructor provided by the user.

An overload is the *best viable overload* if each transformation sequence is ranked at least as high as the corresponding sequence in all other overloads, and it is better in some way than each other viable overload for at least one parameter. If there do not exist a single "best" overload, we will receive a compile time error stating that the function call is ambiguous.

For example, suppose that the function `foo` has been overloaded with the following two declarations.

```
void foo(int i, double d);  // Overload 1
void foo(double d, int i);  // Overload 2
```

Then, the following function calls would use result in the following behavior. The two examples highlighted in orange would result in compile time errors.

```
int i = 0;
short s = 1;
long l = 2;
double d = 0.1;
float f = 0.2;
bool b = true;
std::string str("hello");
```

Function call	Overload 1 rankings	Overload 2 rankings	Result
`foo(i, d);`	Exact match, exact match	Conversion, conversion	Use version 1
`foo(b, f);`	Promotion, promotion	Conversion, conversion	Use version 1
`foo(f, l);`	Conversion, conversion	Promotion, conversion	Use version 2
`foo(s, i);`	Promotion, conversion	Conversion, exact match	Call ambiguous (no winner)
`foo(str, i);`	invalid, conversion	invalid, exact match	No viable overload

You can learn more about function overloading at https://en.cppreference.com/w/cpp/language/overload_resolution.

Possible Exam-style Questions

1. Decide whether a type transformation is a conversion or a promotion.
2. Given code, identify all implicit type transformations and compile time errors.
3. Given a binary operator with arguments of different types, determine which type transformation occurs.
4. Given multiple overloads of a function and several calls to the function, identify any compile time errors and determine which overload is used in each function call.

Suggested Exercises

Exercise 1

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

Exercise 2

Write a program with several implicit and explicit conversions and see which type is preferred in each case. Write a class with a parameterized constructor and use this for an implicit conversion. Write several overloaded functions and see which version is used for a variety of calls. Keep track of any behavior that creates warnings or errors.

1.6 TEMPLATES

Suppose that we want to write generic code that can handle a variety of types, such as a data structure which can store any type of object. Unlike in JavaScript or Python, C++ does not support ambiguous types that are not known at compile time. Instead, we can write a *template* which describes how to perform a task with a generic type, and during compilation, the compiler uses this template to create an explicit version with the type we need.

Function Templates

A *function template* is a function with one or more template parameters. Unlike a normal function parameter, the value of a template parameter must be known at compile time. Every time the function is called with different template argument(s), the compiler will create a new version of the function at compile time. Most commonly, we define the template parameter(s) as the types of the function parameters, which allows our function to operate on a large variety of arguments. For example, suppose that we want to write a `genericMax` function which chooses the larger of any two objects of the same type (as long as that type supports `operator>=`). We could create the following function template, which has a template parameter `T` defining the type of the objects to compare.

```
template <typename T>  // T is a template parameter defining a type
T& genericMax(T& first, T& second) {
  // This assumes that the type T supports operator>=
  return first >= second ? first : second;
}
```

We can call `genericMax` by specifying the template argument in triangle brackets (`<>`) after the function name. Every time the compiler finds a call to `genericMax` with a new template argument, it creates a new version of `genericMax` at compile time to handle that new type.

```
// 1. This calls genericMax with a template argument of int (T = int), which
// creates a version of genericMax at compile time which handles ints
int a = genericMax<int>(10, 7);

// 2. This calls genericMax with a template argument of string, which creates
// a new version of generic max at compile time which handles strings
std::string b = genericMax<std::string>("hello", "world");

// 3. Here, we do not provide a template argument, but based on the types of
// the arguments 3.7 and 9.2, the compiler infers that the template argument
// is double, so this is equivalent to:
//   double c = genericMax<double>(3.7, 9.2);
double c = genericMax(3.7, 9.2);

// 4. Since we already called genericMax with a template argument of int, this
// will use the version of genericMax created in part 1 and will NOT create a
// new version of genericMax
int d = genericMax(14, 7);
```

After compilation, the compiler will have made three versions of `genericMax`: one to handle `ints`, one to handle `doubles`, and one to handle `std::strings`. For example, the version to handle `ints` would be equivalent to the following function.

```
int& genericMaxInt(int& first, int& second) {
  return first >= second ? first : second;
}
```

Whenever templating a function on a `typename`, we must be careful about assumptions made about the template parameter type. The following code will cause a compile time error because `genericMax` assumes that `T` supports `operator<=`, but the `Sheep` class does not.

```
Sheep shawn("Shawn");
Sheep elliot("Elliot");

// This will cause a compile time error because Sheep does not support operator<=
Sheep e = genericMax<Sheep>(shawn, elliot);
```

Non-typename Template Parameters

While it is most common to use `typenames` as template parameters, we can also use other types such as `ints` and `bools` as template parameters. This can be useful because the values of template parameters are known at compile time, while the value of function parameters are not known until run time.

For example, suppose that we want to write a function which reads in several numbers and stores them in a static array of `doubles` for processing. Since we must know the size of a static array at compile time, we cannot specify the number of entries with a function parameter.

```
void processData1(int dataSize) {
  // This will cause a compile time error because the value of dataSize is not
  // known at compile time
  double data[dataSize];

  // Read and process data...
}
```

Instead, we can solve this problem by specifying the number of entries with a template parameter.

```
template <int DATA_SIZE>
void processData2() {
  // This is alright because the value of DATA_SIZE is known at compile time
  double data[DATA_SIZE];

  // Read and process data...
}
```

A major drawback of this approach is that the compiler will create a new version of `processData2` for every new value of `DATA_SIZE`. If we have datasets with several different sizes, this will create a lot of nearly duplicate code, which can create a very large compiled executable. A more general solution is to store `data` on the heap instead of the stack, allowing us to specify the size at run time.

Class Templates

In some cases, we may wish for an entire class to use a template parameter, such as generic a data structure which can store elements of the specified type. In these cases, we should write a *class template*. Just like a function template, a class template takes one or more template parameters whose values must be known at compile time. Every method of the class template is itself a function template taking the same template parameters as the class template.

Recall that under the separate compilation model, a class is separated into a `.hpp` file containing the interface and the a `.cpp` file containing the implementation. Files using the class only include the `.hpp`, and during compilation, we separately compile the `.cpp` file into an object (`.o`) file.

This system is quite different when for class templates. The class does not contain a `.cpp` file because versions of the implementation are automatically generated and linked by the compiler. This means that we do not need to separately compile the class template into an object file. Files using our class must receive both the class template and the class method templates, so all of this information must go into the `.hpp` file for the class. In order to help us mentally separate the class implementation and interface, we split the `.hpp` file into two files: `classname.hpp` with the class template and `classname-private.hpp` with the method templates. We then tell the preprocessor to combine the two files by `#including` `classname-private.hpp` at the bottom of `classname.hpp`. The separation between `classname.hpp` and `classname-private.hpp` exists only to improve readability; as far as the compiler is concerned, they are one big file.

Just like with a function template, the compiler will determine which versions of the class template and method templates to generate at compile time based on how they are used in the code. The compiler will always generate the fewest versions possible of each template, so it will not automatically generate every method of the class just because a particular version of the class is used.

Once again, a class template can have template parameters which are not `typenames`. For example, the `const_iterator` and `iterator` of a data structure are usually identical except for a few return types, so it makes sense to implement them with a single class template. This class template has a `bool` template parameter `IS_CONST` which is `true` for `const_iterator` and `false` for `iterator`. We can use the value of `IS_CONST` to determine the return type of `operator->` and `operator*` at compile time. This approach is used in the MinHeap iterator implementation provided on the supporting website.

An Example Class Template

Suppose that we want to create a wrapper for a pointer called `SafePointer` which automatically calls `delete` on the pointer address in the `SafePointer` destructor. This can help protect against memory leaks, and is the main idea behind the `std::unique_ptr` class provided in the standard library[5]. In order for `SafePointer` to work with all types of pointers, we need to write a class template with a `typename` template parameter specifying the pointer type.

safepointer.hpp

```
// Remember to include a header guard to protect against including multiple
// copies of this file
#ifndef TEMPLATES_SAFEPOINTER_HPP_
#define TEMPLATES_SAFEPOINTER_HPP_

/**
 * \class SafePointer
 * \brief A wrapper for a pointer which helps protect against memory leaks
 * \note This is a simplified version of the std::unique_ptr class
 */
template <typename T>  // The template parameter for this class template
class SafePointer {
  // Just like in a traditional class declaration, we declare all of the
  // class methods and data members as either private or public
 public:
  SafePointer();
  SafePointer(T* pointer);
  SafePointer(const SafePointer& other) = delete;
  SafePointer& operator=(const SafePointer& other) = delete;
  ~SafePointer();
```

[5] See http://www.cplusplus.com/reference/memory/unique_ptr/ for more information about the `std::unique_ptr` class.

```
  T& operator*();
  T* get();

private:

  T* pointer_;
};

// During preprocessing, this will paste the contents of safepointer-private.hpp
// at this point, which effectively combines safepointer.hpp and
// safepointer-private.hpp into a single file.  We divide them into two files to
// separate the interface and implementation of SafePointer, but as far as the
// compiler is concerned, they are one file.
#include "safepointer-private.hpp"

#endif  // TEMPLATES_SAFEPOINTER_HPP_
```

safepointer-private.hpp

```
// This file does not need a headerguard because its contents will be included
// within the header guard of safepointer.hpp.

// All of the SafePointer method templates are found below

template <typename T>
SafePointer<T>::SafePointer() : pointer_{nullptr} {}

template <typename T>
SafePointer<T>::SafePointer(T* pointer) : pointer_{pointer} {}

template <typename T>
SafePointer<T>::~SafePointer() {
  if (pointer_ != nullptr) {
    delete pointer_;
  }
}

template <typename T>
T& SafePointer<T>::operator*() {
  return *pointer_;
}

template <typename T>
T* SafePointer<T>::get() {
  return pointer_;
}
```

You can view and download the source for safepointer.hpp and safepointer-private.hpp on the supporting website.

The following function demonstrates how to use the `SafePointer` class. Similarly to a function template, we pass the template argument to the class template in triangle brackets (<>) after the class name, such as `SafePointer<int>`.

```
void safePointerExample() {
  // This will cause the compiler to create an int version of the SafePointer
  // default constructor
  SafePointer<int> p1;

  // This will cause the compiler to create an int version of the SafePointer
  // parameterized constructor
  SafePointer<int> p2(new int(12));

  // This will cause the compiler to create a double version of the SafePointer
  // parameterized constructor
  SafePointer<double> p3(new double(1.27));

  // This will cause the compiler to create an int version of the SafePointer
  // operator*
  *p2 += 3;

  std::cout << *p2 << std::endl;

  // This will cause the compiler to create a double version of SafePointer
  // operator*
  std::cout << *p3 << std::endl;

  // The SafePointer destructor will clean up all heap memory on the closing
  // curly bracket, so no explicit deletes are necessary
}  // This will cause the compiler to create int and double versions of the
   // SafePointer destructor
```

Because the compiler always creates the fewest possible versions of each template, it never created a `double` version of the `SafePointer` default constructor or any version of `SafePointer::get`, since these methods were never called.

Assumptions about Typename Template Parameters

In all code with a typename template parameter, it is important to consider the assumptions made about this type. Consider the following function template.

```
template <typename T>
void assumptions(T item) {  // Calls T's copy constructor (because item is passed by value)
  T temp;       // Calls T's default constructor
  item++;       // Calls T's operator++
  item = temp;  // Calls T's assignment operator
}  // Calls T's destructor on both temp and item
```

`T` must support a copy constructor, default constructor, increment operator, assignment operator, and destructor. If we attempt to call `assumptions` with a type that does not support all of these operations, we will receive a compile time error when the compiler attempts to create that specific version of `assumptions`. This is especially dangerous if a user can see the interface but not the implementation of our function/class template. Thus, we should always specify all assumptions made about a typename template parameter in the interface comments of a function or class template.

Possible Exam-style Questions

1. Given code using function or class templates, determine how many versions of each template will be created at compile time.
2. Given a function or class template with a typename template parameter, determine all of the assumptions made about the typename parameter.
3. Write a simple function template or a small piece of a templated class.
4. Given the implementation of a non-templated data-structure, identify which changes are needed to convert it to a class template which can store any object type.
5. Describe a meaningful function or class template with a template parameter that is not a typename.

Suggested Exercises

Exercise 1

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

Exercise 2

Write a program containing the following:

- A function template with a typename template parameter. Write down all of the assumptions made about this parameter.
- A class template with a typename template parameter. Write down all of the assumptions made about this parameter.
- A function template which meaningfully uses a non-typename template parameter.
- A main method which uses all of these in meaningful ways.
- A Makefile.

Exercise 3

If you are a student taking the Harvey Mudd CS70 course, convert your `IntList` homework to a class template so that it can store any type that supports a copy constructor.

1.7 INHERITANCE

If we want one class to build upon the existing behavior of another class, we can write a *derived class* which *inherits* from a *base class*. By default, this gives the derived class all of the behavior and capability of the base class and allows the derived class to add its own additional data members and methods. Inheritance is a powerful tool for enabling code reuse, since it allows us to factor out the common behavior between several classes into a single base class from which they all inherit.

Unlike Java and C#, C++ allows for *multiple inheritance*, meaning that a single derived class can inherit from multiple base classes. This leads to significant complexity that is beyond the scope of this textbook, so this chapter will only discuss *single inheritance* (referring to when a derived class inherits from a single base class).

Polymorphism

A fundamental tenant of inheritance is that a derived class is always a superset its base class. That is, a derived class can only add additional capability to a base class (in the form of new data members and new methods); it will always still have the data members and methods of the base class. In memory, a derived class is simply stored as the base class data members directly followed by the derived class data members.

This principle allows for *polymorphism*, which states that a derived class can always be used in place of its base class. That is, because a derived class will always support the entire interface of its base class, we can always supply the derived class where the base class was expected, and we can always transform a derived class to its base class.

For example, suppose that the `Sheep` class inherits from the `Animal` class. Polymorphism allows us to do the following.

```
// Implicitly transform a Sheep to an Animal
Animal shawn = Sheep("Shawn");

// Create an Animal reference to a Sheep
Sheep elliot("Elliot");
Animal& ref = elliot;

// Create an Animal pointer to a Sheep
Animal* pointer = new Sheep("Timmy");
```

We cannot, however, use a base class in place of its derived class, because the derived class will have additional functionality that the base class does not necessarily support. As a result, all of these would cause compile time errors.

```
// Error: cannot transform an Animal into a Sheep
Sheep error1 = Animal();

// Error: cannot create a Sheep& referring to an Animal
Animal animal;
Sheep& error2 = animal;

// Error: cannot create a Sheep* pointing to an Animal
Sheep* error3 = new Animal();
```

Static and Dynamic Dispatch

We can create a base class reference or pointer to an object of a derived class because the derived class supports the public interface of the base class. However, this reference or pointer will only support the public interface of the base class, not the derived class. At compile time, the compiler only knows that the reference/pointer is of the base class type; it cannot know that it "actually" points/refers to an object of the derived class.

For example, suppose that `MotorBike` class inherits from the `Bike` class. Both classes have an `accelerate` method, and the `MotorBike` class has a `refillTank` method.

```
class Bike {
 public:
  void accelerate();
};

class MotorBike : public Bike {
 public:
  void accelerate();
  void refillTank();
};
```

Because the `Bike` interface does not support a `refillTank` method, we cannot call `refillTank` on a `Bike` pointer or reference, even if it "actually" points/refers to a `MotorBike`.

```
MotorBike motorBike;
Bike& bReference = motorBike;
Bike* bPointer = new MotorBike();

// These will both cause compile time errors because Bike does not support
// a refillTank method
bReference.refillTank();
bPointer->refillTank();
```

Instead, our only option is to cast the `Bike` pointer or reference to a `MotorBike` pointer or reference.

```
// To call refillTank, we need to cast the Bike& or Bike* to a MotorBike& or MotorBike*
static_cast<MotorBike&>(bReference).refillTank();
static_cast<MotorBike*>(bPointer)->refillTank();
```

At compile time, we only know that `bReference` and `bPointer` are `Bikes`; only at run time do we learn that `bReference` and `bPointer` point/refer to `MotorBikes`. This raises the question: if we call `accelerate` on `bReference` or `bPointer`, would it call the `Bike` or `MotorBike` version of `accelerate`? Both behaviors are valid and have a specific names:

- **Static dispatch**: Use the type known at compile time to determine which version of the method to call. In this example, static dispatch would call the `Bike` version of `accelerate`.
- **Dynamic dispatch**: Use the type known at run time to determine which version of the method to call. In this example, dynamic dispatch would call the `MotorBike` version of `accelerate`.

Unlike in Java and C#, C++ uses static dispatch by default when a derived class and a base class have methods of the same name. This means that the following code will call the `Bike` version of accelerate.

```
// These will call the Bike version of accelerate (static dispatch)
bReference.accelerate();
bPointer->accelerate();
```

If we want dynamic dispatch, we must label the method with the `virtual` keyword in the base class. For example, we could update the class declarations of `Bike` and `MotorBike` as follows:

```
class Bike {
 public:
  virtual void accelerate();  // Make accelerate virtual for dynamic dispatch
};

class MotorBike : public Bike {
 public:
  void accelerate() override;  // Optional: mark accelerate as override
  void refillTank();
};
```

Notice that we chose to mark `MotorBike::accelerate` with the `override` identifier. This is not required, but it explicitly tells the compiler that we want `MotorBike::accelerate` to override the `virtual` `Bike::accelerate` method. This helps the compiler generate more meaningful errors or warnings if we make a mistake.

Now, when we call accelerate on `bReference` and `bPointer`, it will call the `MotorBike` version of `accelerate`.

```
// These will now call the MotorBike version of accelerate (dynamic dispatch)
bReference.accelerate();
bPointer->accelerate();
```

In order to perform dynamic dispatch, the object must be accessed through a reference or a pointer. If the object is stored directly as an object of the base class, is implicitly transformed to the base class, stripping away all of the additional aspects of the derived class.

```
// Implicitly transforms the right-hand side into a Bike, removing all
// MotorBike qualities from it
Bike bike = MotorBike();

// Therefore, this will always call the Bike version of accelerate, regardless
// of whether accelerate is virtual
bike.accelerate();
```

To summarize, we must meet the following two criteria for a method to be dynamically dispatched in C++:

1. The method must be marked with the `virtual` keyword in the base class.
2. The object on which the method is called must be stored as a pointer or a reference, not directly.

Constructors and Destructors

The constructor of a derived class is responsible for calling the appropriate constructor for the base class and initializing the data members unique to the derived class. If no base class constructor is specified, the derived class constructor will automatically call the default constructor of the base class. We can also specify exactly which base class constructor to call in the member initialization list of the derived class constructor.

For example, suppose that the `Volcano` class inherits from the `Mountain` class. The `Mountain` class contains a single private data member (`height_`), and the `Volcano` class adds one additional private data member (`eruptionChance_`).

```
class Mountain {
 public:
  Mountain();
  Mountain(size_t height);

 private:
  size_t height_;
};

class Volcano : public Mountain {
 public:
  Volcano();
  Volcano(size_t height, double eruptionChance);

 private:
  double eruptionChance_;
};
```

Consider the following implementation of the `Volcano` constructors. The `Volcano` default constructor implicitly calls the `Mountain` default constructor because no `Mountain` constructor is specified in its member initialization list. The `Volcano` parameterized constructor begins by explicitly calling the `Mountain` parameterized constructor in its member initialization list.

```
// Volcano default constructor: implicitly calls Mountain default constructor
Volcano::Volcano() : eruptionChance_{0} {}

// Volcano parameterized constructor
Volcano::Volcano(size_t height, double eruptionChance)
    : Mountain(height),  // Explicitly calls the Mountain parameterized constructor
    eruptionChance_{eruptionChance} {}
```

A derived class can also use the synthesized constructors and assignment operators. These have the following behavior:

- **Synthesized Default Constructor**: Calls the base class default constructor, then default constructs the data members unique to the derived class.
- **Synthesized Copy Constructor**: Calls the base class copy constructor, then copy constructs the data members unique to the derived class.
- **Synthesized Assignment Operator**: Calls the base class assignment operator, then calls the assignment operator on the data members unique to the derived class.
- **Synthesized Destructor**: Calls the destructor on the data members unique to the derived class, then calls the bass class destructor.

The destructor of a derived class automatically calls the base class destructor when it is finished. Thus, a derived class destructor is only responsible for cleaning up data associated with its own data members, not the data associated with the base class.

Abstract Classes

Dynamic dispatch allows us to override base class methods in derived classes. In some cases, we only care about the overrides provided by the derived classes; the method in the base class simply serves as a placeholder. For example, suppose that we write a generic `Animal` class which serves as the base class for several types of animal classes (`Sheep`, `Cow`, etc.). The `Animal` class may include a `sayHello` method which each derived class overrides to return its respective greeting (the Sheep says "baa", the Cow says "moo", etc.). `Animal::sayHello` only serves as a placeholder to be

overridden; it does not make sense to instantiate an object of the `Animal` class (what is a generic animal?), so the generic `Animal::sayHello` should never be called. In a situation like this, we should make `Animal::sayHello` a *purely virtual method* by declaring it as `= 0`.

```
class Animal {  // Animal is an abstract class
  virtual std::string sayHello() = 0;  // sayHello is a purely virtual method

  // Assume that the rest of the Animal declaration goes here...
};
```

A purely virtual method does not contain an implementation; it simply serves as a placeholder to be overridden by derived classes. An *abstract class* is any class with at least one purely virtual method, and we are not allowed to make instances of abstract classes. Abstract classes are useful for factoring out shared behavior of other classes, even if that shared behavior on its own is not a meaningful object. `Animal` is thus a perfect example; we can factor out shared characteristics and behavior of our animals (including data members like `numLegs_` or the ability to `sayHello`), even though it would not make sense to create an instance of this "generic animal".

If a derived class inherits from an abstract class, it must explicitly implement every purely virtual method in the base class in order to be instantiated. Otherwise, the derived class is also abstract.

Unlike in Java and C#, abstract classes are not explicitly marked as abstract (abstract is not a keyword in C++). Instead, the compiler simply determines whether a class is abstract by searching for purely virtual methods.

Possible Exam-style Questions

1. Given code which calls methods belonging to a base and derived class, determine whether static or dynamic dispatch occurs.
2. Given code which uses base and derived classes, identify and fix compile time and run time errors.
3. Given code which uses static dispatch, identify the changes necessary to use dynamic dispatch.
4. Given a base class, write the constructors and destructor for a derived class. Explain the behavior of the synthesized constructors and destructor.
5. Given a base and derived class, convert the base class into an abstract class.
6. Explain the difference between static and dynamic dispatch and provide a situation in which each is preferred.

Suggested Exercises

Exercise 1

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

Exercise 2

Write a base class and a derived class in which the derived class overrides at least one virtual and at least one non-virtual method of the base class. Write a program which calls these methods on pointers, references, and direct values of instances of these classes. Keep track of which version of each method is being called.

Using these same classes, write code which causes an inheritance-related compile time error. Write code which causes an inheritance-related run time error.

2 Data Structures

2.1 INTRODUCING DATA STRUCTURES

A *data structure* is an object which organizes and stores multiple objects of the same type. A data structure generally needs method(s) to add data, find data, and (sometimes) remove data. Different data structures organize data in different ways, allowing them to perform different operations with varying degrees of efficiency. In real world applications, it is often most convenient to use a data structure provided by the standard library (such as `std::stack`, `std::list`, `std::map`, etc.) rather than writing our own data structure from scratch. However, it is worthwhile to learn about the major data structures so we can understand their strengths of weaknesses and choose the best data structure to solve real world problems.

It is worth reemphasizing the distinction between interface and implementation. An *abstract data type* (ADT) specifies the operations a data structure supports. For example, the stack ADT supports a push and a pop operation. On the other hand, a data structure specifies how an ADT is implemented. For example, we can implement a stack as a singly linked list. A data structure user only needs to focus on its ADT, they need not consider the implementation of the data structure to use it.

Public Interface

Most data structures support some or all of the following public interface:

- **Default constructor**: Creates an empty data structure.
- **Copy constructor**: Creates a new data structure containing a deep copy of every element in the original.
- **Destructor**: deletes all elements stored in the data structure.
- **Assignment operator**: Updates a data structure to store a deep copy of every element in the original.
- **swap**: Switches the elements between two data structures such that the first contains the elements of the second and vice versa.
- **operator==**: Compares two data structures as equal if they contain the same elements (and in the same order, if order matters for the data structure). Two data structures do not need to be at the same memory location to compare as equal.
- **operator!=**: Returns the opposite of `operator==`.
- **begin** and **end**: Provides iterators pointing to the data structure, which are discussed in depth in the next chapter.
- One or more methods to add new elements to the data structure.
- One or more methods to search for elements in the data structure.
- One or more methods to remove elements from the data structure.

For example, you can view the public interfaces of data structures provided by the C++ standard library such as std::list (a doubly linked list), std::vector (an resizable array), std::map (a binary tree), and std::unordered_map (a hash table).

Possible Exam-style Questions

1. In your own words, explain the difference between a data structure and an abstract data type (ADT).

Suggested Exercises

The templates case study on the supplemental website includes an implementation of a binary min heap data structure in minheap.hpp and minheap-private.hpp. Read through these two files and identify how this data structure satisfies the public interface described above.

2.2 ITERATORS

Different data structures store data in a variety of complex ways. *Iterators* provide a standardized way to access the elements of a data structure without concerning ourselves with its underlying implementation. An iterator is an object which acts like a pointer to an element within the data structure. Thus, if we dereference an iterator with `operator*`, we receive a reference to an element in the data structure, just like when we dereference a pointer.

Iterators intrinsically create a linear ordering of the elements in a data structure. In a linear data structure, this ordering matches the actual ordering of the data, but for more complex data structures like trees, this ordering does not necessarily reflect the way the elements are stored in memory. The ordering imposed by an iterator may be arbitrary (such as a breadth-first traversal of a tree), or it might distill a meaningful order from an otherwise complicated data structure (such as an in-order traversal of a tree).

Public Interface

C++ supports five types of iterators[6], but this text will focus only on forward and bidirectional iterators. A *forward iterator* can only move in one direction from the beginning of a data structure to the end. A *bidirectional iterator* supports all of the behavior of a forward iterator but can also move backwards.

A forward iterator should implement the following public interface:

- **Default Constructor**: Creates a default iterator. This iterator will not point to a particular element so should not be dereferenced or incremented (see undefined behavior), but two default iterators must be considered equal by `operator==`.
- **Dereference operator (operator*)**: Returns a reference to the element to which the iterator points.
- **Arrow operator (operator->)**: Returns a pointer to the element to which the iterator points.
- **Equality operator (operator==)**: Returns `true` when comparing two iterators which point to the same element in the same object. Comparing iterators from different objects (even of the same type) is undefined behavior.
- **Increment operator (operator++)**: Moves the iterator to point to the next element in the data structure.

A bidirectional iterator must also implement the decrement operator:

- **Decrement operator (operator--)**: Moves the iterator to point to the previous element in the data structure.

A data structure which supports an iterator must implement the following public interface:

- **begin**: Returns an iterator pointing to the first element in the data structure.
- **end**: Returns an iterator pointing to one past the last element in the data structure. If an iterator points to the last element in a data structure and is incremented with `operator++`, it should be equal to the iterator returned by `end` for that data structure.

By these definitions, `begin == end` for any empty data structure. A common misconception is that `end` is always associated `nullptr`. This is not true: the values of `end`'s private data members will depend on how the data structure and iterator are encoded. For example, some iterators are implemented without any pointers, and `end` for an resizable array will store a memory address other than `nullptr`.

We can use iterators to easily iterate through the elements of a data structure. For example, the following for loop will print every element of an `std::list<int>` called `myList`.

```
// Iterate through myList to print every element
for (std::list<int>::iterator i = myList.begin(); i != myList.end(); ++i) {
  std::cout << *i << std::endl;
}
```

[6] You can read about all five types of iterators at http://www.cplusplus.com/reference/iterator/.

This pattern is so common that C++11 introduced the *range-based for loop* to automatically iterate through any data structure with an iterator. The following code is semantically identical to the previous example.

```
// Iterate through myList to print every element with a range-based for loop
for (int element : myList) {
  std::cout << element << std::endl;
}
```

Although the range-based for loop abstracts away the iterator, it uses an iterator behind the scenes to update `element`.

Accessing Private Data Members

We traditionally make an iterator a *nested class* of its data structure by placing the iterator class declaration inside of the data structure class declaration. This automatically gives the iterator access to the private elements of the data structure, which is often necessary to implement `operator++` and `operator--`.

However, this relationship is asymmetric: the data structure does not automatically have access to the private elements of the iterator. The iterator can declare the data structure as a `friend` class to give the data structure access to the all private elements of the iterator. For example, if we want to give our `MyList` class access to the private elements of its iterator, we should add the following line to the iterator class declaration.

```
friend class MyList;
```

Undefined Behavior

Several improper uses of iterators are *undefined behavior*, meaning that the C++ standard does not specify what that behavior does. A program can handle undefined behavior however it pleases: it may crash immediately or fail silently in a way that causes strange behavior in the future. We should never write a program that performs undefined behavior.

The following cases describe the most common forms of undefined behavior for iterators:

- Incrementing or dereferencing an iterator equal to `end`. This includes `begin` in an empty data structure.
- Decrementing an iterator equal to `begin`.
- Dereferencing, incrementing, or decrementing a default constructed iterator.
- Comparing iterators that point to elements in two different objects.
- Dereferencing, incrementing, decrementing, or comparing to an invalid iterator (discussed below).

Non-const methods of a data structure may invalidate certain iterators. A method's interface should specify exactly which iterators are invalidated when the function is called. Const methods will never invalidate any iterators.

For example, the public interface for the erase method of the `std::list` class states that it invalidates any iterators pointing to the element which was erased (see the "Iterator validity" heading at the bottom of the std::list::erase documentation). As a result, the following example includes undefined behavior.

```
// Create a list of ints with 3 values
std::list<int> myList{1, 2, 3};

// Create an iterator pointing to the first element of myList
std::list<int>::iterator fst = myList.begin();

// Erase the first element of myList, which invalidates fst
myList.erase(myList.begin());

// This is undefined behavior because fst is an invalid iterator
int x = *fst;
```

Constant Iterators

A `const_iterator` behaves exactly like an iterator except that the dereference operator (`operator*`) returns a `const` reference rather than a reference. As a result, we can use a `const_iterator` to read but not modify the elements in a data structure. If we place the `const` keyword in front of an iterator, the iterator itself cannot be modified, which prevents it from being incremented (`operator++`) or decremented (`operator--`).

Iterator type	operator* returns	mutable?
`iterator`	a reference	yes (can `++` or `--`)
`const_iterator`	a `const` reference	yes (can `++` or `--`)
`const iterator`	a reference	no (cannot `++` or `--`)
`const const_iterator`	a `const` reference	no (cannot `++` or `--`)

If a data structure or a reference/pointer to a data structure is declared with the `const` keyword, `begin` and `end` will return `const_iterators` rather than `iterators` (this is an example of overloading on constness). An `iterator` can always be transformed into a `const_iterator`, but a `const_iterator` cannot be transformed into an `iterator`.

```
void constIterators(const std::vector<int>& vec) {
  // This will cause a compile time error: vec.begin() returns a const_iterator
  // because vec is const, and we cannot convert a const_iterator to an iterator
  std::vector<int>::iterator i = vec.begin();

  // This is alright: vec.begin() returns a const_iterator because vec is const
  std::vector<int>::const_iterator c = vec.begin();
}
```

`cbegin` and `cend` are similar to `begin` and `end` but always return `const_iterators`.

```
// Create a list of ints with 3 values
std::list<int> myList{1, 2, 3};

// begin returns an iterator because myList is not const.  That iterator is
// then implicitly transformed into a const_iterator
std::list<int>::const_iterator c1 = myList.begin();

// No transformation is necessary because cbegin returns a const_iterator,
// so this approach is more efficient
std::list<int>::const_iterator c2 = myList.cbegin();
```

It is usually best to implement `iterator` and `const_iterator` with a single class template since they only differ in the return types of `operator*` and `operator->`. This class template (traditionally called `Iterator`) has a bool template parameter `IS_CONST`, so `const_iterator` is simply an alias for `Iterator<true>` and `iterator` is simply an alias for `Iterator<false>`. We use the value of `IS_CONST` to determine the return type of `operator*` and `operator->` at compile time. For example, the following statement defines the type returned by `operator*`.

```
// At compile-time this evaluates to "const T&" if IS_CONST is true
// and "T&" if IS_CONST is false
using reference = typename std::conditional<IS_CONST, const T&, T&>::type;
```

`T` is another template parameter specifying the type of elements stored in the data structure.

Possible Exam-style Questions

1. Given a complex data structure, determine the ideal encoding for its iterator.
2. Given a partial implementation of an iterator, fill in the implementation of the dereference operator or increment operator.
3. Given the implementation of an iterator, write `begin` and `end` for the corresponding data structure.
4. Given code using iterators, identify any undefined behavior or errors.
5. Consider the line `int* d = new int[10];` Explain how `d` is a valid bidirectional iterator for this array. What would `begin` and `end` be for this array?
6. Discuss the differences between `const_iterator` and `iterator` and provide cases when each are necessary.
7. Explain how to overload `begin` on constness so that it can returns both `const_iterators` and `iterators`.

Suggested Exercises

Exercise 1

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

Exercise 2

If you are a student taking the Harvey Mudd CS70 course, update the iterator of your `TreeStringSet` homework to perform an in-order traversal of the tree. Carefully select the optimal data members to encode your iterator. Recall that `TreeStringSet` is not allowed to have pointers from children to their parents.

Update your iterator to also support a `const_iterator` by using a single class template. Overload `TreeStringSet::begin` and `TreeStringSet::end` to return the `const_iterator` when appropriate, and add `cbegin` and `cend` methods to `TreeStringSet`.

2.3 LINEAR DATA STRUCTURES

The simplest family of data structures are *linear data structures*, which store elements in a line. These data structures organize elements based on the order in which they are added, not on characteristics of the elements. Linear data structures require relatively little overhead so are often an ideal choice for simple applications.

Interfaces

Because linear data structures store data in a line, they intrinsically have a "front/first" and a "back/last" element. The following operations are therefore especially important:

- **push_front**: Add an element to the front.
- **push_back**: Add an element to the back.
- **pop_front**: Remove the front element.
- **pop_back**: Remove the back element.
- **random access**: Read the *i*th element.

Stack, *queue*, *steque* (stack-ended queue), and *dequeue* (double ended queue) are four abstract data types (ADTs) which support different combinations of these operations. The fixed and resizable array interfaces are included for reference.

Interface	push_front	push_back	pop_front	pop_back	random access	Implementation
Stack	✓		✓			Singly linked list, resizable array
Queue		✓	✓			Singly linked list
Steque	✓	✓	✓			Singly linked list
Deque	✓	✓	✓	✓		Doubly linked list
Fixed array					✓	Array
Resizable array		✓		✓	✓	Resizable array

Stack, queue, steque, and deque are all interfaces—they give no information about how the data structure is implemented. The final column of the table provides possible implementations of each interface.

The C++ standard library provides stack and queue implementations called `std::stack` and `std::queue`.

Implementation

There are three major linear data structure implementations: fixed arrays, resizable arrays, and linked lists. Each implementation can perform different operations in constant time and comes with its own tradeoffs and considerations.

Fixed Arrays

The simplest linear data structure is the *fixed array*, which we simply declare with the `new` keyword.

```
int* array = new int[10];
```

Fixed arrays allow for constant time random access and require no additional overhead, though it is often useful to store the length of the array in a `size_t`. If we know the exact size of our data structure ahead of time, fixed arrays are often an ideal choice. However, there is no way to grow or shrink a fixed array, so they are not well suited for situations in which we must continuously add and remove elements.

A pointer into a fixed array serves as an iterator for the array. `begin` is the pointer storing the address of the first element of the array, and `end` is the pointer storing the address one past the last element of the array.

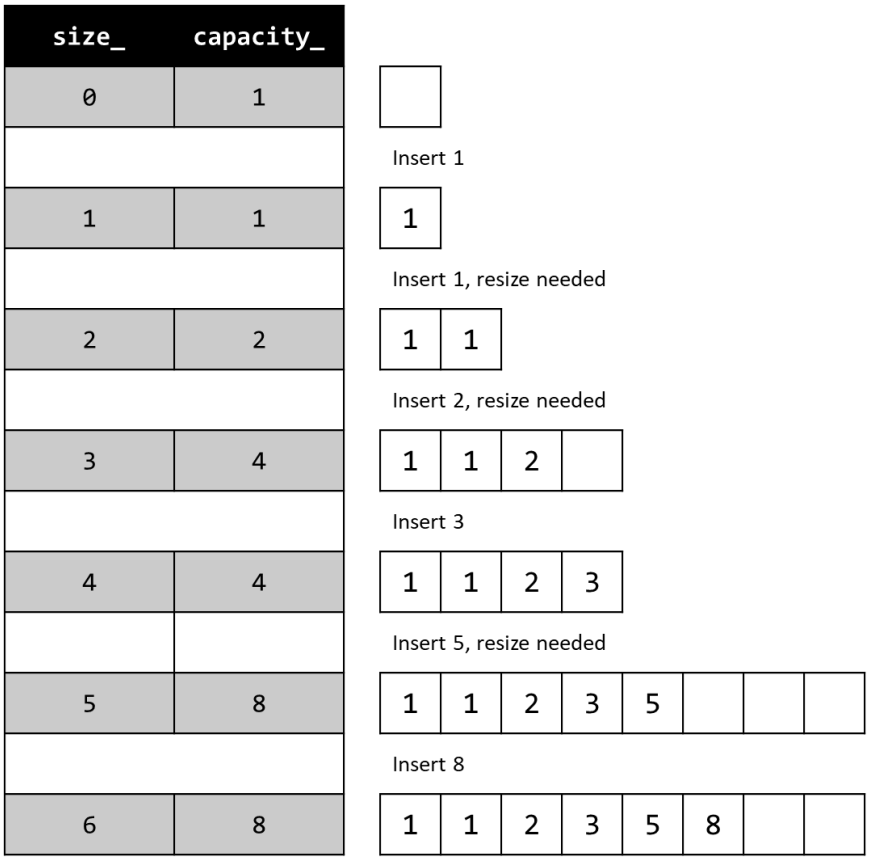
Resizable arrays

A *resizable array*[7] provides the outward appearance of an array which can grow and shrink, allowing us to continuously add and remove elements. Under the hood, a resizable array stores elements in a fixed array, allowing for efficient random access. Under the following circumstances, we "resize" this fixed array by creating a fixed array of a new length and copying over the elements from the old array:

- If we try to insert an element but the fixed array is full, we first create a new fixed array that is twice as long.
- If we remove an element and the fixed array is now less than a quarter full, we create a new fixed array that is half as long.

This strategy of doubling and halving allows for amortized constant insert and remove. That is, operations that require a resize are themselves quite expensive, but a total of n operations will only take $O(n)$ time because these expensive operations are far apart. For a complete proof, see example 2 from the chapter on amortized analysis.

Depending on the types of elements being stored, the fixed array may store pointers to elements instead of elements directly. In addition to this fixed array, a resizable array needs two additional private data members: a `size_t size_` storing the number of elements in the data structure and a `size_t capacity_` storing the length of the current fixed array. When we remove an element, there is no need to zero out that entry in the array; it is more efficient to simply decrement `size_` and leave the "deleted" element in place.



A diagram showing the step-by-step result of inserting the first six Fibonacci numbers into a resizable array.

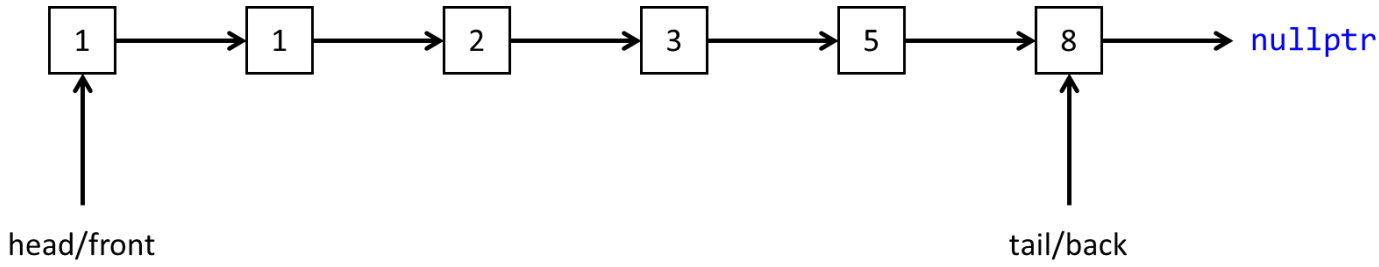
Because a resizable array uses a fixed array under the hood, we can once again use a pointer into the array as an iterator. Insert and remove invalidate all outstanding iterators if they trigger a resize.

The C++ standard library provides a resizable array called `std::vector`.

[7] Resizable arrays go by several other names including dynamic arrays, growable arrays, array lists, and vectors.

Linked List

While we can efficiently add and remove elements from the end of a resizable array, we cannot efficiently add and remove elements from the beginning (this requires us to shuffle over every element of the array, which takes $O(n)$ time). A *linked list* solves this problem by storing data as a collection of separate nodes. In a *singly linked list*, each node is a simple object storing an element and a pointer to the next node. The list itself stores a pointer to the first (head) and last (tail) node. We use `nullptr` to represent any node that does not exist (such as the head/tail of an empty list or the "next" of the final node).



A singly linked list containing the first six Fibonacci numbers.

A singly linked list can implement the stack, queue, and steque ADTs. To support `pop_back`, we must use a *doubly linked list*, in which each node contains a pointer to the next node and a pointer to the previous node. The dequeue ADT therefore requires a doubly linked list. When implementing a linked list, we should watch out for edge cases in the empty and one element cases based around updating the head and tail pointer.

A linked list iterator is traditionally a nested class encoded with a pointer to the `Node` containing the element to which the iterator points. `operator++` uses the next pointer of that node. A singly linked list can only support a forward iterator, while a doubly linked list can support a bidirectional iterator.

The C++ standard library provides a singly linked list called `std::forward_list` and a doubly linked list called `std::list`.

Comparing Implementations

The following table compares the advantages and disadvantages of these three linear data structure implementations. In summary, fixed arrays are the most efficient but require that we know the size of our data ahead of time. Resizable arrays allow us to add and remove elements while retaining the efficiency and random access of fixed arrays, but inserting and removing elements can occasionally take a very long time if they require a resize. A linked list allows for easy access to both the front and the back at the cost of random access, data contiguity, and significant overhead.

Quality	Fixed Array	Resizable Array	Linked List
Supported operations	Random access	Random access, `push_back`, `pop_back`	`push_front`, `pop_front`, `push_back`; also `pop_back` if doubly linked
Write/insert efficiency	Very fast (no overhead)	Usually very fast, but very slow if a resize is needed	Some overhead needed to update pointers
Space overhead	One `size_t` to store size, array is exactly as big as it needs to be	Two `size_ts`, array is at most twice as big as it needs to be	One (singly linked) or two (doubly linked) pointers per element
Memory contiguity[8]	Contiguous (good for caching)	Contiguous (good for caching)	Every element is separate (bad for caching)

[8] When memory is stored contiguously rather than in several small pieces, this increases efficiency due to caching. If you are a student at Harvey Mudd, you will have the opportunity to learn more about caching in Computer Systems (CS105).

Possible Exam-style Questions:

1. In your own words, explain the advantages and disadvantages of fixed arrays, resizable arrays, and linked lists.
2. Given an `.hpp` file, identify the linear ADT.
3. Explain why a dequeue cannot be implemented with a singly linked list.
4. Explain how to implement a stack with a singly linked list and a resizable array. Explain the advantages of each implementation.
5. Given a scenario, determine the ideal linear ADT (stack, queue, steque, or deque):
    a. Which data structure would an iterator need to traverse a binary tree in order?
    b. Which data structure would an iterator need for a breadth-first traversal of a binary tree?
6. Explain why resizable arrays are a bad choice if we are most concerned with worst case efficiency.

Suggested Exercises

Exercise 1

By using two resizable arrays, we can implement a queue that can `push_front` and `pop_back` in amortized constant time. Implement this data structure by either using two `std::vectors` or by writing your own resizable array class from scratch.

Exercise 2

If you are a student taking the Harvey Mudd CS70 course, modify your `IntList` homework to support `pop_back` in constant time. You will need to convert it to a doubly linked list.

2.4 TREES

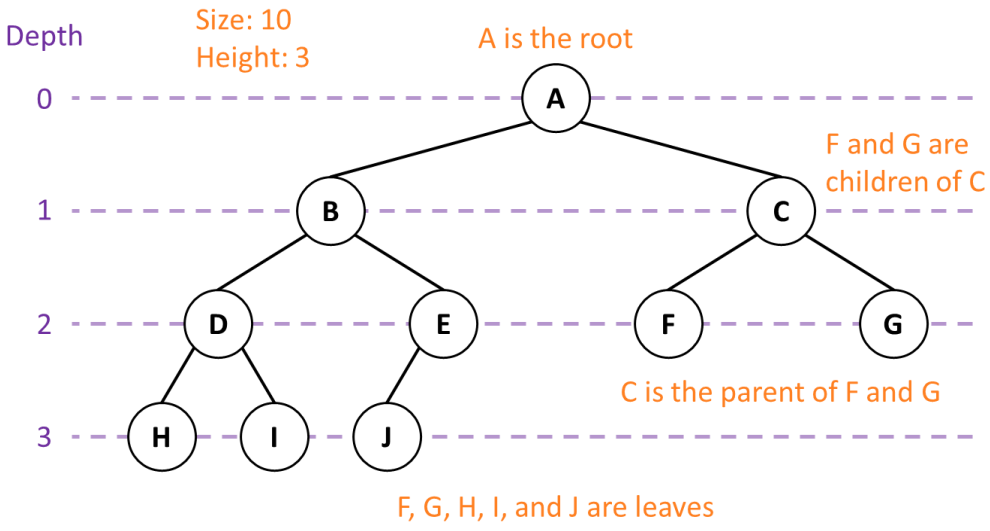
Because linear data structures store elements in a line, searching for an element usually requires traversing the entire data structure. For example, if we insert n numbers into a resizable array or a linked list, it will take $O(n)$ time to find the smallest number and $O(n)$ time to find if a particular number has been added. *Trees* store data in a more complicated recursive structure which allows us to search for data far more quickly. If we enforce certain rules about where we insert new data, we can find and insert elements in $O(\log n)$ time.

There are several types of trees, and many of them have complicated insertion rules. One of the best ways to understand these rules is to watch the result of inserting several numbers. Professor David Galles of the University of South Florida has created visual simulators for most of the trees discussed in this textbook (look under the "Indexing" bullet)[9]: https://www.cs.usfca.edu/~galles/visualization/Algorithms.html.

Terminology

The following terminology is important for discussing trees:

- **Tree**: A connected acyclic graph.
- **Node**: An element (aka vertex) in a tree, consisting of a value and pointers to its children.
- **Leaf**: A node without any children.
- **Root**: The node from which every other node in a tree descends; the "topmost" node.
- **Binary tree**: A tree in which every node has at most two children.
- **Binary search tree (BST)**: A binary tree in which for every node, its left child and all of its descendants have smaller values and its right child and all of its descendants have larger values. This rule is sometimes referred to as the BST property or the BST invariant.
- **Depth**: The number of edges between the root and a given node.
- **Height**: The depth of the deepest leaf. Thus, a one-element tree has a height of 0, and we define an empty tree to have a height of -1.
- **Size**: The number of nodes in a tree.
- **Complete tree**: The most possibly balanced tree for a given number of nodes, obtain by filling in 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.
- **Perfect tree**: A tree in which every row is completely filled and thus every leaf has the same depth. For binary trees, a perfect tree is simply a complete tree with $2^n - 1$ nodes for some natural number n.

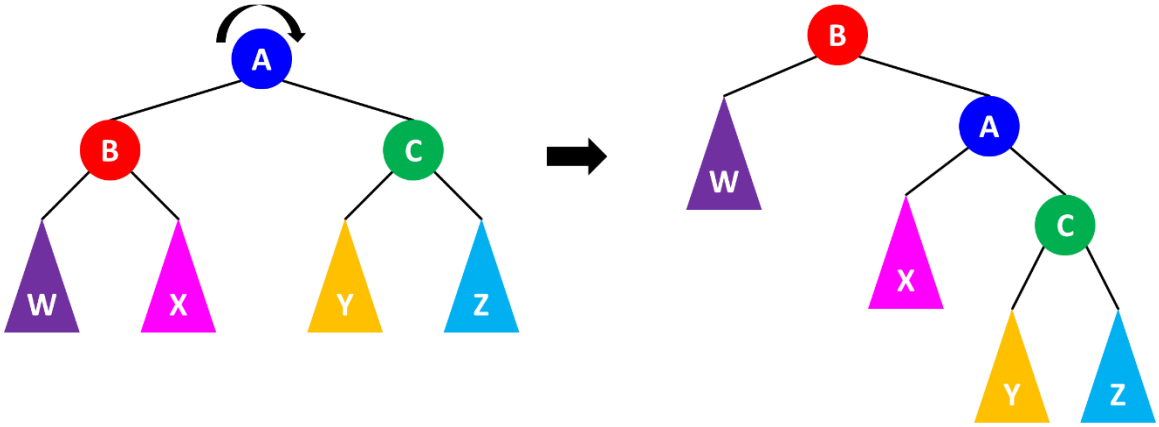


A complete tree with ten nodes. Each node is shown as a circle with a capital letter.

[9] Note that certain trees have multiple "variants", so the exact behavior may vary from examples used in this textbook or the Harvey Mudd CS70 course.

Trees can be defined recursively, as each node can be thought of as a root of its own subtree. In several types of binary search trees, the following operations are particularly important:

- **Insert at leaf**: This operation inserts a new value as a leaf while maintaining the BST property. We traverse the tree from the root downward, and at each node, we recur on the left child if the inserted value is smaller or the right child if the inserted value is larger. We repeat this process until we find an empty space and place the new value there, making it a leaf.
- **Right rotation**: A method of swapping a node and its left child which maintains the BST property (see diagram below). A right rotation turns the tree clockwise.
- **Left rotation**: A mirror image of a right rotation which instead switches a node with its right child. A left rotation turns the tree counterclockwise.
- **Insert at root**: This operation inserts a new value as the root of a subtree while maintaining the BST property. First, the new value is inserted as a leaf using the "insert at leaf" strategy. Then, this value is brought upward by continuously performing a rotation at each of its parents until it becomes the root of the subtree. Because any node in a tree can be considered the root of its own subtree, this procedure allows us to insert a value anywhere along the path of nodes we pass on the way down when inserting at leaf.
- **Search**: To search for an element, we start at the root and work downwards. At each node, we compare the search value to the node value and recur on the left child if it is smaller or the right child if it is larger. This strategy ensures that we will find the node with the search value if it is in the tree.



A tree before and after a right rotation is performed at the root node. W, X, Y, and Z represent subtrees that could contain of zero or more nodes and are themselves unaffected by the rotation.

The following function implements a right rotation. A left rotation has a similar but symmetric implementation.

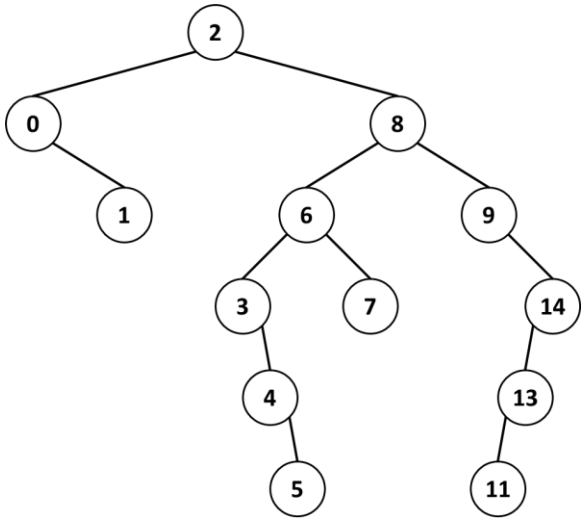
```
// Performs a right rotation about curRoot, the root of a subtree
void rightRotation(Node*& curRoot) {
  Node* newRoot = curRoot->left_;
  curRoot->left_ = newRoot->right_;
  newRoot->right_ = curRoot;

  // Finally, we update curRoot so that the external pointer to this subtree
  // points to the newRoot
  curRoot = newRoot;
}
```

The following mnemonic can help distinguish between left and right rotations. If you point your right thumb towards the root of the tree, your fingers will naturally curl clockwise, the direction of a right rotation. If you point your left thumb towards the root of the tree, your fingers will naturally curl counterclockwise, the direction of a left rotation.

Basic Binary Search Trees

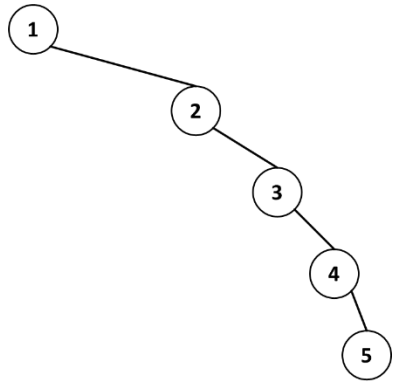
Simulator: https://www.cs.usfca.edu/~galles/visualization/BST.html

In a basic binary search tree, we add all elements using the insert at leaf strategy. The structure of the tree is highly dependent on the order in which we insert elements. For example, the first element will always be the root, no matter how many elements we add. If the data we insert is well randomized, the tree will be relatively well balanced with a height of $O(\log n)$. Insert and search both do work based on the height of the tree, so both operations take $O(\log n)$ time in the well-balanced case.



The BST produced by inserting the numbers 2, 0, 8, 9, 6, 3, 4, 5, 1, 7, 14, 13, 11.

However, a basic BST will not always be well balanced. For example, if we insert values in sorted order, the tree will become a straight line, or a *stick*. The height of the tree is then $O(n)$, so insert and search take $O(n)$ time.



The BST produced by inserting the numbers 1, 2, 3, 4, 5.

Random BST

If we have access to every value we want to add beforehand, we can randomize the order of these elements and insert them into a basic BST. On average this will only be 1.39 times less balanced than the corresponding complete tree. Because 1.39 is a constant factor, insert and search take $O(\log n)$ time in the average case. This strategy is referred to as a *random tree*.

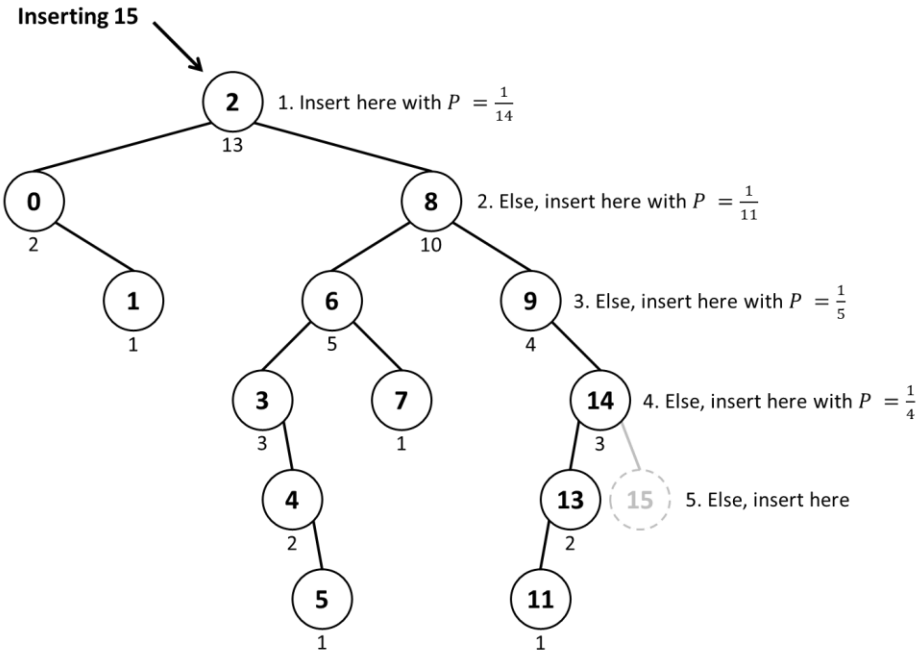
Unfortunately, a random tree requires that we know all values beforehand, so can only be used in certain cases.

Self-balancing Binary Search Trees

Self-balancing binary search trees are a family of binary search trees which uses special insertion strategies to help produce well-balanced trees. Some of these strategies guarantee good balance, while others only increase the probability of good balance.

Randomized BST

A *randomized binary search tree* models the behavior of a random tree without requiring that we know every value beforehand. At each step of an insertion, we choose to insert at root with probability $P = \left(\frac{1}{N+1}\right)$, where N is the number of elements in that subtree. This probability is based on the question "if this subtree was organized randomly, what is the probability that the value we are inserting is the root." This requires that we know the number of elements in each subtree in constant time, so we each node must store the size of its subtree. As a result, randomized trees are less space efficient than other binary search trees.



This diagram shows the potential steps needed to insert a value into a randomized tree. Subtree size is shown underneath each node (this value must be stored by the nodes).

Unlike other self-balancing BSTs, a randomized BST requires a random number generator. Because the tree relies on randomness to stay balanced, we can potentially get a poorly balanced tree or even a stick if we get "unlucky".

Splay Tree

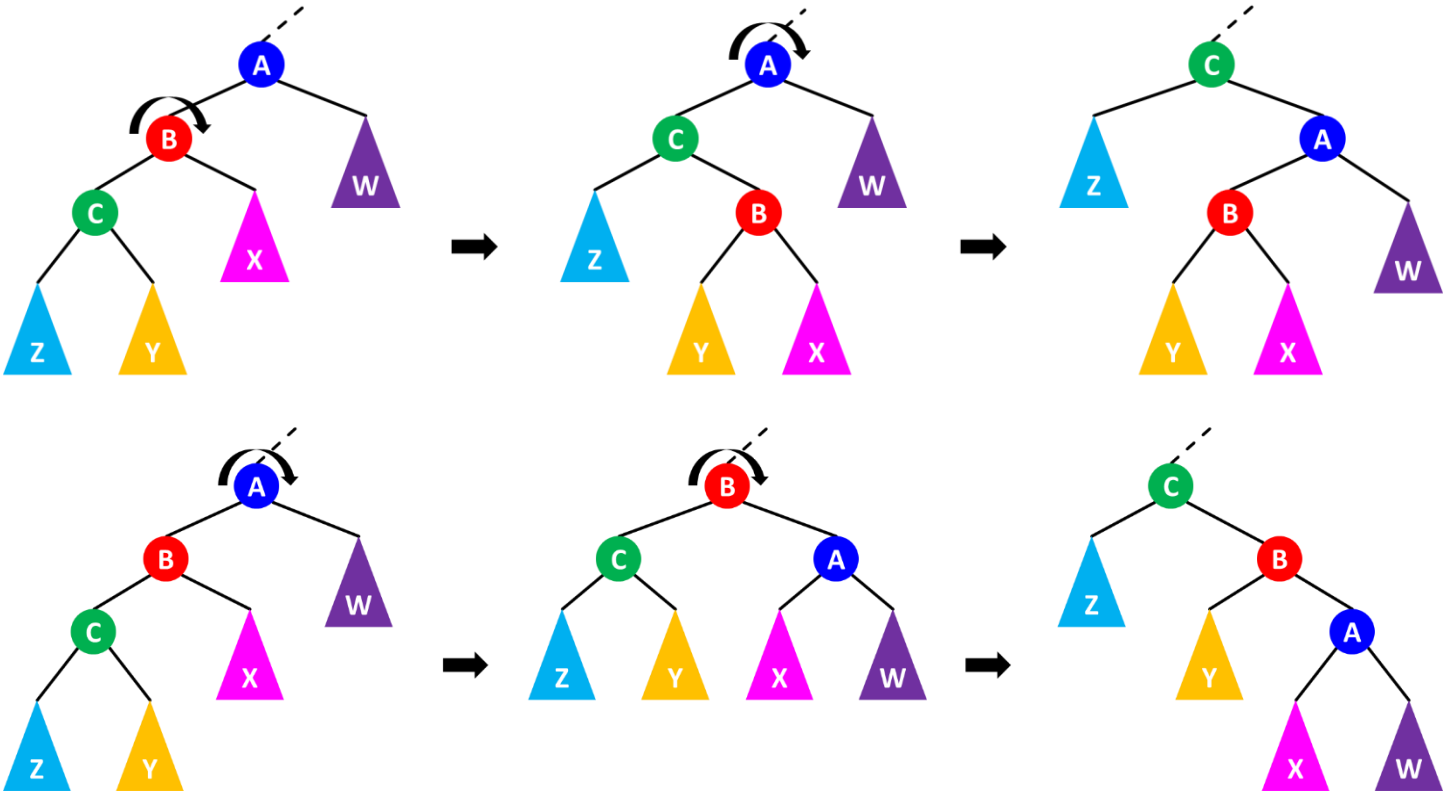
Example Simulator: https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

In many situations, we need to access certain values in our dataset more frequently than others. *Splay trees* help facility this by caching more frequently accessed data at the top of the tree. A splay tree operates like a basic BST with the following two changes:

1. Every insert and search moves the desired element to the root of the tree. As a result, the more frequently we access an element, the higher up in the tree it will be, so the more quickly we can access it.
2. Insert and search perform *splay rotations* whenever possible.

We move elements to the root of a tree through rotations. Traditionally, these rotations are done one step at a time, always focusing on the current parent of the element being brought up. In a splay tree, we consider rotations in pairs,

and if both rotations are in the same direction, we rotate the grandparent first, then the parent (see diagram). This is called a splay rotation and naturally breaks up sticks.



This diagram compares how to bring up element z using a traditional rotation (top) vs. a splay rotation (bottom). W, X, Y, and Z represent subtrees that could contain of zero or more nodes and are themselves unaffected by the rotations.

Splay trees are the only type of tree in which search is a non-`const` method. Splay trees do not strictly enforce balance and only become well balanced with use. For example, if sorted elements are inserted into a splay tree, they will start off as a stick until the tree is used.

2-3-4 Tree

Example Simulator: https://www.cs.usfca.edu/~galles/visualization/BTree.html (set Max Degree to 4, and you can check the "Preemptive Split/Merge" box to use proactive instead of reactive splitting)

A *2-3-4 tree* is a non-binary tree but is guaranteed to always be perfect. It has three types of nodes:

- **2-node**: Has one value with 2 children, just like a node in a binary tree.
- **3-node**: Has two values with 3 children.
- **4-node**: Has three values with 4 children.

2-3-4 trees have the following rules:

1. Within a node, any value to the left of another must be smaller.
2. In a 3- or 4-node, children between parent values must be numerically between those parent values.
3. The tree must be perfect (all leaves have the same depth).
    a. As a result of this rule, every node must have either all of its children or none of its children.

There are two valid implementations of a 2-3-4 tree: one which proactively splits 4-nodes whenever possible, and one which reactively splits 4-nodes only when necessary. We can compare the insertion strategies for each.

Insertion into a 2-3-4 tree with reactive splitting (comments unique to reactive splitting are shown in **bold green**):

1. Traverse from the root downward to find a leaf node in which the new value can be added.
2. If the leaf is a 2- or 3-node, add the new value to make it a 3- or 4-node. In this case, we or done.
3. **Otherwise, if the leaf is a 4-node, first promote the middle value from the 4-node to its parent.**
    a. **If the parent node is a 2- or 3-node, simply combine the promoted value with the parent node.**
    b. **If the parent node is a 4-node, promote its middle value and repeat the process again.**

Insertion into a 2-3-4 tree with proactive splitting (comments unique to proactive splitting are shown in **bold green**):

1. Traverse from the root downward to find a leaf node in which the new value can be added. **On the way down, every time you come across a 4-node, preemptively split it by promoting the middle value.**
2. Insert the value into the leaf node. **Because we preemptively split, the leaf node must be a 2- or a 3-node, so the value will fit.**

Reactive splitting takes more steps when a split is necessary, but proactive splitting may split nodes unnecessarily. Neither strategy is clearly superior.

Because 2-3-4 trees are always perfectly balanced, they have guaranteed $O(\log n)$ insert and search even in the worst case. However, they can be cumbersome to implement since they require three types of nodes which can efficiently switch between types.

Red-Black Tree

Example Simulator: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

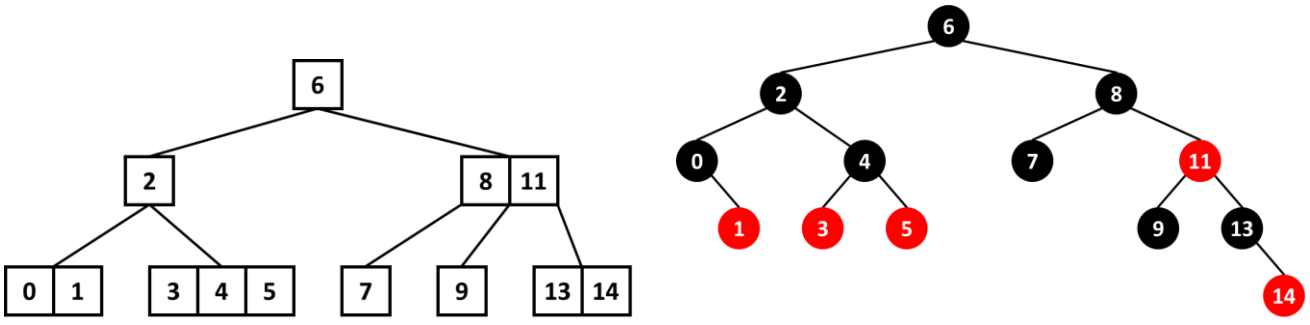
A red-black tree is a binary tree which models the behavior of a 2-3-4 tree, allowing for much simpler implementation. Red-black trees have two types of nodes: black nodes representing the "center" of a 2-3-4 node and red nodes representing the edge values of 3- or 4-nodes. For example:

- A 2-node becomes a single black node.
- In a 3-node, the left value becomes a black node and the right becomes its red right child. It is equally valid to make the right value the black node instead as long as we are consistent.
- In a 4-node, the middle value becomes a black node and the left and right values become its red children.

From this conversion, a red-black tree will always satisfy the following rules:

1. The root will be black.
2. A red parent will never have a red child.
3. Any path from the root to an unfilled child (ie a `nullptr`) will have the same number of black nodes.

Rules 2 and 3 ensure that the deepest leaf is at most twice as deep as the shallowest leaf. Thus, just like a 2-3-4 tree, red-black trees strictly enforce balance, so search and insert are guaranteed $O(\log n)$. In fact, there exists a bijection between 2-3-4 trees and red-black trees–any valid 2-3-4 tree can be converted to a valid red-black tree, and any valid red-black tree can be converted to a valid 2-3-4 tree.



The equivalent 2-3-4 and red-black trees created from inserting the numbers 2, 0, 8, 9, 6, 3, 4, 5, 1, 7, 14, 13, 11.

To implement insert, we begin by inserting at leaf. Then, we "repair" the tree by either recoloring or rotating nodes as needed. The exact details are a bit complex and beyond the scope of this textbook, but several good examples can be found online.

AVL Tree

Example Simulator: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Like a red-black tree, an *AVL tree* guarantees $O(\log n)$ insert and search by enforcing balance rules. Specifically, each node tracks its *balance factor*: the height of its left subtree minus the height of its right subtree. The AVL tree then enforces one simple rule: every node must have a balance factor between -1 and 1.

This rule is a stricter balance requirement than a red-black tree, so any AVL tree can be colored as a valid red-black tree. In the asymptotic worst case, a red-black tree can have a height of at most $2 \log n$, while an AVL tree can have a height of at most $1.44 \log n$. Lookup in an AVL tree is on-average faster since it is usually more balanced, but insertion is on-average slower since it usually requires more rotations to maintain the stricter requirement.

To implement insert, each node must keep track of its balance factor. We begin by inserting at leaf, and on the way back up, we update the balance factor of each node along the insert path. If any node has a balance factor of -2 or 2, we perform the proper rotation to restore balance.

Comparing Binary Search Trees

The following table compares the binary search tree discussed in this textbook. Height determines the efficiency of insert and search. Difficulty approximates the complexity of implementation, ranging from 1 (easiest) to 10 (hardest).

	Height				
Tree	**Average**	**Worst**	**Difficulty**	**Advantages**	**Disadvantages**
Basic	$O(\log n)$	$O(n)$	1	Easiest to implement	Likely to have poor balance if data is not randomized
Random	$O(\log n)$	$O(n)$	2	Easy to implement	Must know all data beforehand, can get unlucky
Randomized	$O(\log n)$	$O(n)$	4	Easiest self-balancing BST	Requires a random number generator, nodes must store subtree size, can get unlucky
Splay	$O(\log n)$	$O(n)$	5	Caches recently accessed values towards the top	Search modifies the tree, only becomes balanced with use
2-3-4	$O(\log n)$	$O(\log n)$	10	Guaranteed perfect tree	Difficult to implement, requires major space and/or time overhead
Red-black	$O(\log n)$	$O(\log n)$	8	Guaranteed balance, faster insert than AVL	Slower lookup than AVL, nodes must store color
AVL	$O(\log n)$	$O(\log n)$	5	Guaranteed balance, faster lookup than red-black	Slower insert than red-black, nodes must store balance factor

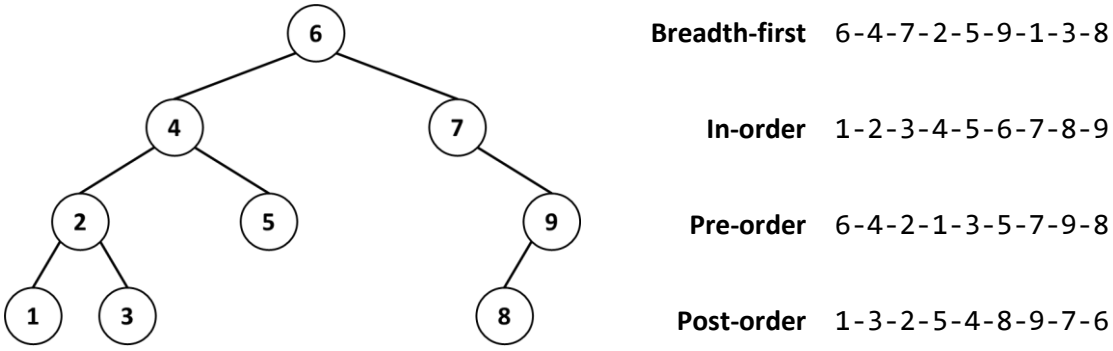
For a basic BST, the worst case occurs when the values are inserted in order, which creates a stick. In random and randomized trees, the worst case is statistically unlikely but is possible if the random number generator is "unlucky". A splay tree begins as a stick if the values are inserted in order but becomes balanced overtime as we search for values. 2-3-4, red-black, and AVL trees all enforce exact balance requirements which ensure that even in the worst case, the tree is always within a certain factor of perfectly balanced.

Asymptotic run time hides constant factors so does not provide enough information alone to choose between these data structures. For example, AVL trees have faster lookup but slower insert than Red-black trees because they enforce a stricter balance requirement. Splay trees have a poor worst-case height, but they have other benefits which make them the best choice for certain situations.

Tree Traversal

There are four major ways to iterate through a binary search tree:

1. **Breadth-first traversal**: Begin at the root and iterate through the tree left to right, top to bottom. This orders the nodes by increasing depth.
2. **In-order depth-first traversal**: For each node, explore the left subtree, visit the node, and explore the right subtree. This orders the nodes from smallest value to largest value.
3. **Pre-order depth-first traversal**: For each node, visit the node, explore the left subtree, and explore the right subtree.
4. **Post-order depth-first traversal**: For each node, explore the left subtree, explore the right subtree, and visit the node.



A diagram showing the four major traversal orders for an example BST.

Since children do not store pointers to their parents, we cannot traverse a tree by simply storing a pointer to the current node. Instead, we need an additional data structure depending on the type of traversal:

- In breadth-first traversal, we use a queue to store the nodes we must explore in the future. Every time we arrive at a node, we add its children to the back of the queue. The next node to explore is always the first element of the queue, and we never need to backtrack.
- In depth-first traversal, we use a stack to backtrack when necessary. Before we travel to the child of a node, we add the node to the stack so that we can get back to it in the future. The exact backtracking pattern will depend on the type of depth-first traversal.

Possible Exam-style Questions

1. Given a list of integers, draw a basic BST, splay tree, 2-3-4 tree, red-black tree, or AVL tree showing the state if those numbers are inserted in the provided order.
2. Given a 2-3-4 tree, draw the equivalent red-black tree.
3. Given a red-black tree, draw the equivalent 2-3-4 tree.
4. Given a tree, identify if any of its rules have been violated.
5. Reason about the tradeoffs between the different types of trees.
6. Given a situation, choose the ideal type of tree.
7. Describe a situation in which a splay tree can achieve an expected search time better than $O(\log n)$.
8. Implement a piece of the insertion algorithm for one of the self-balancing binary search trees.

Suggested Exercises

Exercise 1

On paper, insert the numbers 2, 15, 10, 12, 3, 11, 4, 5, 6, 13, 8, 1, 7, 9, 14 into the following types of trees:

- Basic BST
- Randomized BST (use a random number generator such as https://www.random.org/integers/)
- Splay tree
- 2-3-4 tree
- Red-black tree
- AVL tree

Check your answers against the simulators provided by Professor David Galles: https://www.cs.usfca.edu/~galles/visualization/Algorithms.html.

Which trees appear the most balanced? Which trees were the easiest to insert into by hand?

You can generate your own random series at https://www.random.org/sequences/.

Exercise 2

If you are a student taking the Harvey Mudd CS70 course, update your `TreeStringSet` homework to implement a splay, red-black, or AVL tree instead of a randomized tree.

2.5 PRIORITY QUEUES

In some applications, we are only care about the most extreme element at any given time, often when that is the element we must process. The *priority queue* ADT addresses this need by allowing us to look at and remove the element with the highest priority. Specially, it supports the following interface:

- **Push**: Add a new element.
- **Pop**: Remove the highest priority (aka top) element.
- **Peek**: Give access to the highest priority (aka top) element.

The definition of priority depends on the type of elements stored, but for numbers, we often define smaller numbers as having higher priority.

The C++ standard library provides a priority queue called `std::priority_queue`.

Binary Search Tree Implementation

In a binary search tree, the leftmost leaf always stores the minimum value, so we can easily implement a priority queue with a BST. If we use an AVL or red-black tree and add a pointer to the leftmost node, we can implement the priority queue interface as follows:

- **Push**: $O(\log n)$, add the element with the usual insertion procedure.
- **Pop**: $O(\log n)$, remove the leftmost node with the usual deletion procedure.
- **Peek**: $O(1)$, dereference the pointer to the leftmost node.

BSTs require a lot of extra infrastructure to support general search which is not utilized by the priority queue interface. This additional overhead reduces efficiency and uses unnecessary space, so the priority queue implementations described next are usually preferred.

Binary Heap Implementation

Example Simulator: https://www.cs.usfca.edu/~galles/visualization/Heap.html

A *binary heap* is a complete binary tree specifically designed to implement the priority queue ADT. Binary heaps do not enforce the binary search tree property; they only require that each node is smaller than all of its children. There is no distinction between the left and right child.

Recall that a complete tree is the "most balanced" possible tree for a given number of nodes and is created by filling the tree from top to bottom, left to right. We can encode a complete tree as a resizable array which follows this ordering (ie, a breadth-first traversal of the tree). It is considered best practice to use 1-based indexing here, leaving index 0 blank, since it provides the following nice properties:

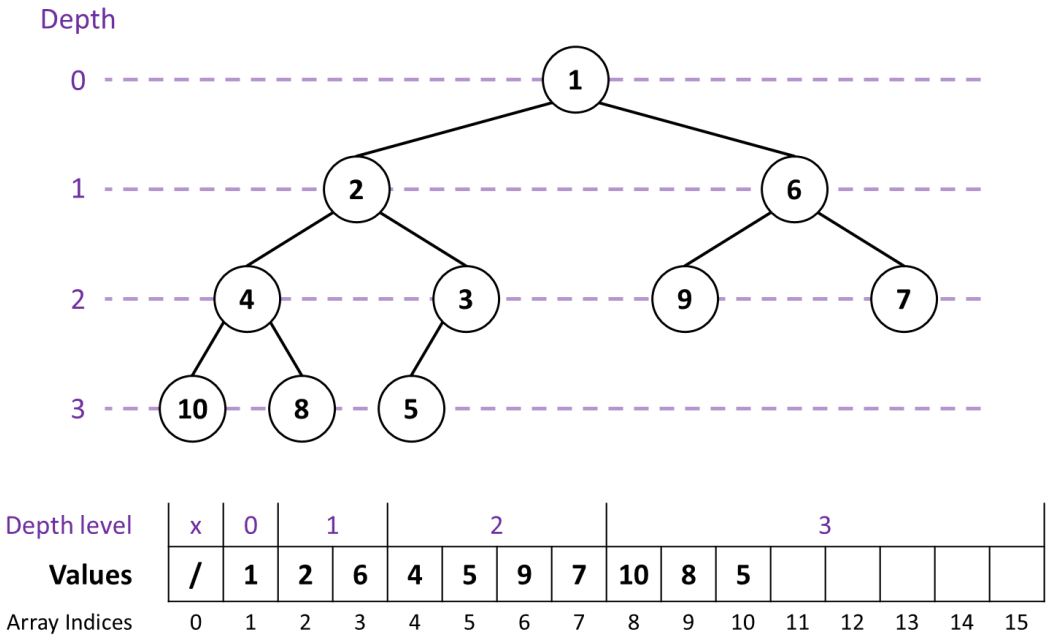
- The first leaf of a tree with n nodes is at index $i = \left\lfloor \frac{n}{2} \right\rfloor + 1$.[10]
- If a node is at index i, then its left child is at index $2i$, its right child at index $2i + 1$, and its parent is at index $\left\lfloor \frac{i}{2} \right\rfloor$.
- We must double the resize our array exactly when we begin a new level of the tree.

Whenever we insert a new node, we can simply add it to the end of the array, as this corresponds to the next open spot in the complete tree. Due to the properties stated above, we can move between a parent and child in constant time.

We can insert elements in amortized $O(\log n)$ time with the following strategy:

1. Place the element at the next available leaf (the end of the array).
2. If the parent is larger than the new element, directly swap the parent and the child (not a rotation).
3. Repeat step 2 recursively until the parent is smaller than the new element or we reach the root.

[10] $\lfloor x \rfloor$ represents the floor function, which rounds down the expression between the brackets to the nearest integer.



The binary heap produced by inserting the numbers 3, 4, 7, 10, 5, 9, 6, 8, 2, 1. Both the binary tree and array representations are shown for reference.

We can pop the top element in amortized $O(\log n)$ time with the following strategy:

1. Remove the rightmost leaf (the last element in the array) and place it at the root (overwriting the old root).
2. If either of the children of this new root are smaller, swap it with the smaller child.
3. Repeat step 2 for the new value until it is smaller than both of its children.

By design, the first entry in the array will always be the highest priority element, so peek simply returns a reference to this entry. Push, pop, and peek thus have the same asymptotic complexity for a self-balancing BST and a binary heap. However, a binary heap has several advantages which make it preferable for implementing the priority queue ADT:

- **Ease of implementation**: Binary heaps are far easier to implement than AVL or red-black trees, especially since these BST implementations must support element deletion.
- **Optimal balance**: AVL and red-black trees only guarantee a constant factor within perfect balance. Since a binary heap is a complete tree by design, it is always as balanced as possible.
- **Memory contiguity**: While trees store each node as a separate entry on the heap, binary heaps store the entire tree as a single contiguous array, increasing efficiency due to caching.
- **Space efficiency**: Every node in a binary tree requires two additional pointers and a data member to store either color (red-black) or balance factor (AVL). A binary heap does not have this unnecessary overhead.
- **Less overhead**: In order to maintain balance, self-balancing binary trees require significant overhead in the form of rotations and updating node data members. Binary heaps do not.

Push and pop are amortized $O(\log n)$ because they may trigger an array resize.

Fibonacci Heaps

Fibonacci heaps are a more complex priority queue implementation with asymptotically better performance:

- **Push**: Amortized $O(1)$.
- **Pop**: Amortized $O(\log n)$.
- **Peek**: $O(1)$.

Their implementation is beyond the scope of this textbook, but several good explanations can be found online.

Possible Exam-style Questions

1. Provide a situation for which a priority queue is well suited.
2. Given a list of integers, draw a binary heap after those numbers are inserted in the provided order.
3. Given a binary heap drawn as a binary tree, draw the corresponding array representation.
4. Given a binary heap drawn as an array, draw the corresponding binary tree representation.
5. Explain why we prefer to implement binary heaps as resizable arrays rather than collections of nodes. Why do we not use this approach for binary search trees?
6. Give three reasons why the binary heap is preferable to the binary search tree for implementing a priority queue.
7. Implement push or pop for a binary heap.

Suggested Exercises

Exercise 1

On paper, insert the numbers 2, 15, 10, 12, 3, 11, 4, 5, 6, 13, 8, 1, 7, 9, 14 into a binary heap. Draw both the binary tree and resizable array representations. Check your answer against the simulator provided by Professor David Galles: https://www.cs.usfca.edu/~galles/visualization/Heap.html.

You can generate your own random series at https://www.random.org/sequences/.

Exercise 2

The supporting website contains a full implementation of a binary heap. Read through this implementation, paying close attention to the encoding and the `insert`, `peakMin`, and `deleteMin` methods.

Exercise 3

Write a program which uses either the `std::priority_queue` class. Your program should use the `push`, `pop`, and `top` methods.

Exercise 4

If you are a student taking the Harvey Mudd CS70 course, update your `TreeStringSet` homework to support the priority queue ADT. You will need to add a method to remove elements. Make sure to support peek in constant time.

2.6 HASH SETS AND HASH TABLES

The humble array (or resizable array) is one of the best data structures in terms of simply storing data: it has zero overhead, stores objects contiguously, and allows access to any element with a single pointer addition. Arrays are great when we know exactly where to look for our data, but they are not particularly good at helping us *find* our data if we don't know where it is. For example, if I have an array of `Sheep` and I know that Shawn is at index 22, I can easily access Shawn in constant time. However, if I don't know where Shawn is, finding him will require searching the entire array.

Hash sets and hash tables address this issue by providing an efficient mechanism for placing and finding objects in an array. In the average case, these data structures can insert and lookup elements in constant time, asymptotically better than the $O(\log n)$ time provided by self-balancing trees. Hash sets/tables are a favored data structure in many real-world applications and form the backbone of many scripting languages such as JavaScript and Python.

We will start by clarifying the distinction between hash sets and hash tables. A *hash set* stores a collection of elements called *keys* and is used to identify whether a key is in the set. For example, we could use a hash set to store all of the words in the English language and quickly check if an new string is part of this list (here, the key type is `std::string`). On the other hand, a *hash table* maps keys to values, allowing us to find a value by searching for its key. For example, we could use a hash table to map sheep names (keys) to sheep (values), allowing us to quickly find a sheep based on its name (here, the key type is `std::string`, and the value type is `Sheep`). Hash sets and hash tables have the same conceptual implementation; the only difference is that a hash table will store a value object (or a pointer to a value) with every key.

Hash Functions

The innovation of a hash set/table is its ability to quickly identify where to place or search for a key in an array. This is achieved through a *hash function*, which converts keys into integer numbers called *hash values* (usually of type `size_t`). By taking the hash value modulo the length of the array, we can map a key to an index in our array.

For example, consider a simple hash function accepting strings which adds the ASCII value of each character. Suppose that we wanted to find the index of the string `"shawn"` in our hash set, which uses an array of length 31.

$$hash(\text{"shawn"}) = 115 + 104 + 97 + 119 + 110 = 545$$
$$index = 545 \,\%\, 31 = 18$$

We would therefore expect to find `"shawn"` at index 18 of our array.

A good hash function should have the following characteristics:

1. **Deterministic**: The same key must always produce the same hash value, or else we cannot rely on our hash function to find our key after we place it in the array.
2. **Efficient**: Since we must call the hash function every time we insert or lookup a key, it should calculate its hash value as quickly as possible.
3. **Uniform distribution**: Keys should be mapped evenly across all possible values of a `size_t`, which reduces the chance that multiple keys map to the same array index.
4. **Avalanche effect**: A small change in the key should result in a big change in the hash value, which reduces the chance that similar keys map to the same array index.

Characteristics 3 and 4 help reduce *collisions*, which refer to when multiple keys map to the same index of an array. Our previous example does a poor job of avoiding collisions: most sheep names will produce hash values between 0 to 1000 (poor uniform distribution), and similar sheep names will have similar hash values (poor avalanche effect). To achieve these characteristics, real hash functions are mathematically much more complex and usually involve prime numbers and bitwise operations.

Collision Resolution

While a good hash function can help avoid collisions, no hash function can avoid them altogether (except for a perfect hash, which we will discuss later). Thus, a hash set/table must use one of the following *collision resolution strategies* to handle when multiple keys map to the same array index.

Separate Chaining

In *sperate chaining*, we store multiple elements at each array index by storing a linked list at each index. This means that the array has type `std::forward_list`[11]`<T>*`, where `T` is the key type. To insert a key, we simply add it to the front of the linked list at its corresponding index. To check if a key exists in the hash set, we check it against every element in the linked list at its corresponding array index.

Separate chaining is intuitive to think about, easy to implement, and allows for easy insertion. However, it has a large amount of space overhead since each linked list needs to store extra information in the form of pointers between nodes. Further, because elements in a linked list are not contiguous, this implementation is less cache friendly, which causes less efficient run time.

Open addressing

In *open addressing*, we store exactly zero or one elements at each array index. Thus, when multiple keys map to the same index, we need to find another index at which to place the key. Let's assume that our key maps to the index `index`. We define some function $f(n) \in N \to N$ which converts the step number n to an offset amount $f(n)$. Then, we check `index` $+ f(0)$, `index` $+ f(1)$, `index` $+ f(2)$, and so on until we find an empty index.

Three of the most popular functions are as follows:

1. **Linear Probing**: $f(n) = n$.
2. **Quadratic Probing**: $f(n) = n^2$.
3. **Double Hashing**: $f(n) = kn$, where k is determined by a second hash function[12] called on the key. Therefore, colliding keys usually have different values of k.

In less mathematic terms, we can think of $f(n)$ as a "stepping pattern" defining how we move past the original `index`:

1. **Linear Probing**: Step by 1 index at a time.
2. **Quadratic Probing**: Step by 1, then 3, then 5, and so on.
3. **Double Hashing**: Step by a constant amount k each time, where k is determined by a second hash function called on the key.

When using quadratic probing or double hashing, we must have a prime array size. Otherwise, we can loop forever without finding an empty index by infinitely stepping over empty indices. In double hashing, we must also ensure that k is neither 0 nor a multiple of the length of the array.

In open addressing, the underlying data structure is an array of pointers to keys (`T**`), where `T` is the type of the key. We use a `nullptr` to represent empty indices in our array. When we insert a value at an index, we create it on the heap and store the address at that index in the array. To check if an item exists, we use our hash function to find the original `index` and traverse the array with our stepping pattern until we either find a match or a `nullptr`.

Most hash sets/tables in standard libraries use open addressing due to the reduced space overhead (no linked list pointers) and contiguous data. However, if a certain application requires more insertions than lookups, separate chaining may be preferable because it allows for constant time insertion regardless of the number of collisions.

---

[11] We use an `std:forward_list` (singly linked) instead of an `std::list` (doubly linked) to reduce overhead.

[12] In practice, we can achieve double hashing with a single hash function by using the bottom 32 bits of the hash for the index and the top 32 bits of the hash for the step size.

Load Factor and Resizing

The *load factor* of a hash set/table is defined as:

$$\lambda = \frac{number\ of\ items}{number\ of\ buckets}$$

For separate chaining, the load factor calculates the average number of items per linked list, and in open addressing, the load factor calculates the fullness of the array. To ensure constant-time lookup and insertion in the average case, we must resize the hash set every time the load factor exceeds a constant threshold. A reasonable threshold for separate chaining is between 1 to 4, and a reasonable threshold for open addressing is between 0.5 to 0.7 (we clearly cannot exceed 1 for open addressing). Upon resizing, we double (or approximately double) the array size and reinsert all of the values, since many keys will now map to new indices. Although these resize operations are expensive $O(n)$, they are increasingly far apart, so do not effect the asymptotic expected case run time (see the amortized analysis for a resizable array).

Unlike a resizable array, however, insertion into a hash set/table is not amortized constant time. In the worst case, our hash function always returns the same value, meaning that every single key will collide. In both collision resolution strategies, this will cause every insert and lookup to take $O(n)$ time (completely independent of the resizing issue). The takeaway is twofold: first, the performance of a hash set/table is highly dependent on the quality of the hash function, and hash sets/tables cannot make the same worst case guarantees as red-black or AVL trees.

Perfect Hash

There is one exception to the second point; we *can* make a strong worst-case guarantee if we have a *perfect hash function*. A perfect hash function guarantees that a certain set of n predefined keys have no collisions. A *minimal perfect hash function* also promises to map those keys to the numbers 0 through $n - 1$.

With a perfect hash function, we no longer need to implement a collision resolution strategy—we can simply place each key at the index provided by the perfect hash function. Because there are no collisions, we can check if a key is in the hash set by only checking against the first bucket to which it maps (guaranteed constant time). This is a powerful approach when we have a known set of elements beforehand and want to check if an unknown element is part of this group as efficiently as possible. Unfortunately, this approach is not ideal if our set of keys is constantly changing, since we would need to make a new perfect hash every time.

Possible Exam-style Questions:

1. Explain the pros and cons of open addressing compared to separate chaining.
2. Explain the pros and cons of the linear, quadratic, and double hashing step patterns.
3. Given a hash function, a starting bucket size, and some keys, simulate inserting those keys into a hash set. Identify and handle collisions.
4. Explain why a prime bucket size is required for quadratic and double hashing but is not for linear probing and separate chaining. Explain why a prime bucket size is still advantageous for linear probing and separate chaining.
5. Given an implementation of a hash set or hash table, identify issues that could lead to errors or poor performance.
6. Suppose that ASCII has been replaced by a new standard for mapping characters to numbers. Unfortunately, the 26 lowercase alphabetic characters are no longer contiguous and correspond to seemingly random numbers. Describe how you could use a perfect hash and an array to determine in constant time if a `char` corresponds to a lowercase alphabetic character

Suggested Exercises

Exercise 1

Implement a hash set which uses open addressing, and implement all three stepping patterns (linear probing, quadratic probing, and double hashing). Implement your hash set as a class template parameterized on the type of object stored in the hash set.

Exercise 2

Implement a hash table with separate chaining. Your hash table should be a class template with two type parameters: the key type and the value type.

Exercise 3

Implement a minimal perfect hash and a corresponding lookup table which can efficiently check if a character is a vowel.

3 Run Time Analysis

3.1 INTRODUCING RUN TIME ANALYSIS

In order to make informed decisions about the best solution to a problem, we often consider the efficiency of different approaches. *Run time analysis* uses a methodological and reproducible approach to measure (either theoretically or experimentally) how long a piece of code will take to execute. It is especially relevant for data structures, since efficiency is one of the most important factors when choosing between data structures which will store many elements.

To discuss the theoretical run time of a piece of code, we must define the following:

- **Cost metric**: The thing that we are interested in measuring, such as the number of calls to a particular function, the number of times a particular line is run, etc.
- $\boldsymbol{n}$: An external parameter defining the "size" of the problem, such as the number of elements upon which our program operates or the upper bound of the outermost for loop.

There are several varieties of run time analysis which characterize how many times a cost metric is run for a given n. The following list is ordered from most specific to least specific:

1. **Exact theorem**: Finds a function $f(n)$ giving the exact number of times the cost metric is run for an input of size n. This may involve using summations (possibly with substitution of variables) to represent for loops.
2. **Approximate theorem**: Finds a function $f(n)$ giving the approximate number of times the cost metric is run for an input of size n. It is similar to exact theorem but may leave out lower order terms or small constants.
3. **Asymptotic complexity**: Finds a comparison function $g(n)$ such that for some constants n_0 and c, $f(n) \leq cg(n)$ for all $n > n_0$. In other words, it compares the run time to a function which grows at an equal or faster rate within a constant factor for large n.
4. **Complexity class**: Places the problem into broad categories such as P, NP, NP hard, etc. These categories are especially helpful for identifying problems which grow exponentially hard based on n.
5. **Decidability**: Determines if the program is guaranteed to ever finish, even with an infinite amount of time.

Finally, we can create a *physical benchmark* by running a program on a real machine for varying n and physically measuring the run time.

It is often important to identify the best type of analysis for a given situation. In general, exact or approximate theorem are best for comparing two algorithms that do the same thing. For example, if we are choosing between two sorting algorithms, we should consult exact or approximate theorem to decide which algorithm is best suited for our needs. Asymptotic complexity can hide major constant factors which matter when choosing between algorithms.

Asymptotic complexity is best for considering how a change in input will affect run time. For example, it allows us to easily answer the question "how will the run time change if we double the input size?".

Finally, complexity class and decidability help us determine whether a problem can be feasible solved. If our problem is undecidable or NP hard with large inputs, we will need to restructure the problem or goal.

Possible Exam Style Questions

1. Given some code, identify n and a reasonable cost metric.
2. Given a scenario, determine the most appropriate type of run time analysis.

Suggested Exercises

Find a program or piece of code that you have previously written in which the run time depends on some external factor. Identify n and a reasonable cost metric.

3.2 SUMMATIONS

In order to calculate the exact theorem for code containing for loops, it often helps to translate these for loops to summations. A *summation* is a sequence of numbers added together, such as $1 + 2 + 3 + 4 + 5$. We often represent summations with Σ notation, which behaves similarly to a for loop. Specifically, the expression below the Σ defines the *index of summation* with a starting value, the number above the Σ defines the inclusive upper bound for the index of summation, and the expression after the Σ is added to the sum for every value of the index of summation.

For example, we can represent the previous summation as

$$\sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5 = 10.$$

Once we convert a for loop to a summation, we can use properties of mathematics to derive a closed form. These conversions are beyond the scope of this textbook. You can also use mathematic solvers such as MATLAB, Mathematica, or Wolfram Alpha (https://www.wolframalpha.com/) to find a closed form.

Basic Summations

In Σ notation, the index of summation increases by one for each term. If a for loop follows this property, we can directly translate it into Σ notation. For example, consider the following for loop.

```
for (size_t i = 5; i < 10; ++i) {
  sum += 2 * i + 3;
}
```

We can translate this into Σ notation with `i` as the index of summation, since `i` is incremented by one for each iteration of the for loop. Notice that the upper bound is 9, not 10, because the upper bound is inclusive in Σ notation.

$$\sum_{i=5}^{9} 2i + 3 = 13 + 15 + 17 + 19 + 21 = 85$$

If a for loop uses external variables, we can also include these in Σ notation.

```
size_t n, x;
std::cin >> n;
std::cin >> x;

for (size_t i = 1; i < n; ++i) {
  sum += i * x;
}
```

We can translate this for loop into the following Σ expression (assuming that n is not zero), which yields the following closed form. Notice that n and x appear in both the Σ expression and in the closed form.

$$\sum_{i=1}^{n-1} ix = x + 2x + 3x + \cdots + (n-1)x = \frac{(n-1)(n-2)}{2}x$$

Variable Substitution

For loops can support any arbitrary expression in the iteration portion of the loop. Unfortunately, Σ notation does not have this same freedom. Thus, if the iteration portion of the for loop has any behavior other than incrementing the iteration variable by one, we cannot directly translate the for loop to Σ notation. Instead, we can use *variable substitution*, in which we define a new variable related to the iteration variable which increases by one every time.

This is best demonstrated by example. Consider the following for loop.

```
for (size_t i = 2; i < n; i = i * i) {
  sum += i;
}
```

Since i does not increment by one, we will create a variable i' that corresponds to i, starts at 1, and increments by 1 for each iteration of the loop. The following table shows the values of i and i' for the first six iterations of the for loop.

i	2	4	16	256	65536	4,294,967,296
i'	1	2	3	4	5	6

Next, we find an expression for i in terms of i' and vice versa.

$$i = 2^{2^{i'-1}}$$

$$i' = \log_2(\log_2(i)) + 1$$

At this point, it is a good idea to check our expressions by plugging in some of the values we calculated in our table. This is left as an exercise for the reader.

In our original for loop, i reaches a maximum value of $n - 1$. We can add one last entry to our correspondence table for this maximum value and use the expression derived previously to find the corresponding value of i'. Because i' must be an integer, we use the floor function $\lfloor x \rfloor$ to round down to the nearest integer.

i	2	4	16	256	65536	4,294,967,296	$n-1$
i'	1	2	3	4	5	6	$\lfloor \log_2(\log_2(n-1)) + 1 \rfloor$

Finally, if the expression within the for loop depends on the value of i, we substitute in our expression for i in terms of i'. We now have all of the tools necessary to create a summation over i' that is equivalent to our for loop.

$$f(n) = \sum_{i'=1}^{\lfloor \log_2(\log_2(n-1))+1 \rfloor} 2^{2^{i'-1}}$$

Multiple Summations

If our code contains multiple nested for loops, we can translate each loop into a summation using the strategies discussed previously, which yields multiple nested summations. Consider the following example.

```
for (size_t i = 0; i < n; i += 2) {
  for (size_t j = 1; j < i; j *= 2) {
    sum += 2 * i * j;
  }
}
```

We will create a variable substation for both i and j since neither increment by one.

i	0	2	4	6	8	10	$n-1$
i'	1	2	3	4	5	6	$\frac{n-1}{2}+1=\frac{n+1}{2}$

$$i = 2(i'-1)$$

$$i' = \frac{i}{2}+1$$

j	1	2	4	8	16	32	$i-1$
j'	1	2	3	4	5	6	$\log_2(i-1)+1$

$$j = 2^{j'-1}$$

$$j' = \log_2(j)+1$$

We use these substitutions to create the following summation. In the upper bound of the inner summation, we substituted $i = 2(i'-1)$ since i has been replaced with i' in the outer summation.

$$f(n) = \sum_{i'=1}^{\frac{n+1}{2}} \sum_{j'=1}^{\log_2(2(i'-1)-1)+1} 2 * 2(i'-1) * 2^{j'-1} = \sum_{i'=1}^{\frac{n+1}{2}} \sum_{j'} 2^{j'+1}(i'-1)$$

Possible Exam Style Questions

1. Given one or more nested for loops and a cost metric, find an exact theorem for the run time by converting the loop(s) to summation(s) using Σ notation.
2. Perform variable substitution to convert a for loop with a non-standard increment to an equivalent summation.

Suggested Exercises

Create a program containing multiple nested for loops with non-standard increments. Use variable substitution to find an exact theorem in Σ notation for the run time. Use a mathematic solver to find a closed form. Execute your for loop with a counter for a variety of n, and verify that every final count matches your exact theorem.

3.3 ASYMPTOTIC ANALYSIS

In more complicated situations, calculating an exact run time theorem can be impractical or even impossible. *Asymptotic analysis* allows us to reason about run time much more easily by comparing our run time to a well-known function without needing to calculate an exact cost.

Formal Definition

Let g be some comparison function such as $g(n) = n^2$. We can then define three sets in terms of g:

- $\boldsymbol{O(g)}$ ("big O") is the set of all functions f such that for some positive constants c and n_0, $f(n) \leq cg(n)$ for all $n > n_0$. Thus, we can think of g as a figurative "upper bound" on f, although it is not a strict upper bound.
- $\boldsymbol{\Omega(g)}$ ("big Omega") is the set of all functions f such that for some positive constants c and n_0, $f(n) \geq cg(n)$ for all $n > n_0$. Thus, we can think of g as a figurative of "lower bound" on f.
- $\boldsymbol{\Theta(g)}$ ("big Theta") is the set of all functions f such that for some positive constants c, k, and n_0, $cg(n) \leq f(n) \leq kg(n)$ for all $n > n_0$. Thus, we can think of g as a figurative "upper and lower bound" on f. By definition, $\Theta(g) = O(g) \cap \Omega(g)$, which means that f is in $\Theta(g)$ if and only if it is in both $O(g)$ and $\Omega(g)$.

For example, if $g(n) = n^2$, here are some example functions in each set. Since $O(g)$, $\Omega(g)$, and $\Theta(g)$ are always infinite sets, we cannot "list out all of $O(g)$".

$$O(n^2) = \{n, 5n + 5, n \log n, 50n^2, 12n^2 + 10n, 65536, \dots\}$$

$$\Omega(n^2) = \{10n^4, 2^n, n!, n^n, nn! + 12n, n^3 - 50n^2, n^2, \dots\}$$

$$\Theta(n^2) = \left\{n^2, 10n^2, \frac{1}{50}n^2, n^2 + 50n + 3, \frac{n(n-1)}{2}, \dots\right\}$$

If f is the exact run time of our program, we often wish to choose the simplest g such that f is in $\Theta(g)$. This allows us to reason about our runtime as the simpler comparison function g without needing to calculate f. We then say that the program "has a big Theta complexity of $\Theta(g)$". Note that every run time has more than one big Theta complexity; in fact, for any f, there exists an infinite number of comparison functions g such that $f \in \Theta(g)$. Thus, we generally prefer to use the simplest g possible.

If you already know the exact run time of your program, you can usually choose the best g by selecting the highest order term and removing any constant multipliers. For example:

Exact run time	Recommended comparison	Big Theta complexity
$f(n) = 42$	$g(n) = 1$	$f(n) \in \Theta(1)$
$f(n) = 3n + 5$	$g(n) = n$	$f(n) \in \Theta(n)$
$f(n) = n \log_2 n$	$g(n) = n \log n$	$f(n) \in \Theta(n \log n)$, see[13]
$f(n) = \frac{n(n-1)}{2}$	$g(n) = n^2$	$f(n) \in \Theta(n^2)$
$f(n) = 2^n + n^{64}$	$g(n) = 2^n$	$f(n) \in \Theta(2^n)$
$f(n) = n! + 3^n$	$g(n) = n!$	$f(n) \in \Theta(n!)$
$f(n) = n! + n^n$	$g(n) = n^n$	$f(n) \in \Theta(n^n)$

[13] Although $f(n)$ has a log base 2 term, notice that $g(n)$ uses the baseless log (n). Logarithms of different bases are always within a constant factor of each other due to the change of base formula, so we do not need to specify the logarithm base in $g(n)$.

When people colloquially use big O complexity, they usually choose the *smallest* simple comparison function, the same comparison function preferred for big Theta complexity. For example, if $f(n) = 3n$, while it is true that f has a big O complexity of $O(n!)$, most people prefer to use the comparison function $g(n) = n$ instead.

Building Intuition

The formal definitions provided previously allow us to determine if a known run time has a particular big O or big Theta complexity. In practice, however, we often wish to quickly identify an appropriate comparison function without calculating an exact run time. In this section, we will build intuition for some of the most common big O complexities.

Big O complexity	Name	Description
$O(1)$	Constant	The cost remains the same regardless of any change in n
$O(n)$	Linear	When n doubles, cost approximately doubles
$O(n^2)$	Quadratic	When n doubles, cost approximately increases four-fold
$O(n^3)$	Cubic	When n doubles, cost approximately increases eight-fold
$O(2^n)$	Exponential	When n increases by 1, cost approximately doubles
$O(\log n)$	Logarithmic	When n multiplies by a constant factor, cost increases by approximately 1

In the following examples, assume that n is a `size_t` which was defined previously.

Constant Time

If the amount of work does not change based on n, the program is constant in n.

```
for (size_t i = 0; i < 5; ++i) {
  std::cout << n << std::endl;
  ++costMetric;
}
```

n	0	1	2	3	4	5	6	7	8	9	10
cost	5	5	5	5	5	5	5	5	5	5	5

The cost remains fixed at 5 regardless of the value of n, so this example is $O(1)$ (constant time).

Linear Time

If we perform a constant amount of work for each element, the program is linear in n.

```
for (size_t i = 0; i < n; i += 2) {
  ++costMetric;
}
```

n	0	1	2	3	4	5	6	7	8	9	10
cost	0	1	1	2	2	3	3	4	4	5	5

When n doubles from 4 to 8, our cost doubles from 2 to 4, so this example is $O(n)$ (linear time).

Quadratic Time

If we perform on the order of n work for each element, the program is quadratic in n.

```
for (size_t i = 0; i < n; ++i) {
  for (size_t j = 0; j < i; ++j) {
    ++costMetric;
  }
}
```

n	0	1	2	3	4	5	6	7	8	9	10
cost	0	0	1	3	6	10	15	21	28	36	45

When n doubles from 4 to 8, our cost approximately quadruples from 6 to 28, so this example is $O(n^2)$ (quadratic time).

Exponential Time

If the amount of work multiplies for each additional element, the program is exponential in n. This often occurs when we make multiple recursive calls on problems that are only a constant size smaller.

```
size_t fib(size_t n) {
  return (n < 2) ? n : fib(n - 1) + fib(n - 2);
}
```

Our cost metric is the total number of calls to `fib` needed to calculate `fib(n)`. When building up our table, it helps to refer to previous values. For example, `Cost(fib(5)) = 1 + Cost(fib(4)) + Cost(fib(3))`.

n	0	1	2	3	4	5	6	7	8	9	10
cost	1	1	3	5	9	15	25	41	67	109	177

Notice that when n increases by 1, our cost nearly doubles, so this example is $O(2^n)$ (exponential time).

Logarithmic Time

If the size of the problem must multiply by a constant factor (such as double) before we do another unit of work, the program is logarithmic in n. This often occurs when we divide the problem in half and throw out one half at each step.

```
for (size_t i = 1; i <= n; i *= 2) {
  ++costMetric;
}
```

n	0	1	2-3	4-7	8-15	16-31	32-63	64-127	128-255
cost	0	1	2	3	4	5	6	7	8

Notice that n must approximately double for our cost to increase by one, so this example is $O(\log n)$ (logarithmic time). "Logarithmic" refers to logarithms of all bases, since by the change of base formula, all bases are within a constant proportional factor of each other. Thus, the following code is also $O(\log n)$ even though n must approximately triple for our cost to increase by one.

```
for (size_t i = 1; i <= n; i *= 3) {
  ++costMetric;
}
```

Possible Exam Style Questions

0. Given some code, identify a comparison function g such that the runtime of the code is in $\Theta(g)$.
1. Given a comparison function g, provide a few functions in $O(g)$, $\Omega(g)$, and $\Theta(g)$.
2. Given functions f and g, determine if f is in $O(g)$, $\Omega(g)$, and/or $\Theta(g)$.
3. Given some code, identify an improvement that would change the big Theta complexity.

Suggested Exercises

Find a program or piece of code that you have previously written in which the run time depends on some external factor. Identify the most applicable big O complexity for your code.

3.4 BEST, WORST, AND EXPECTED CASE

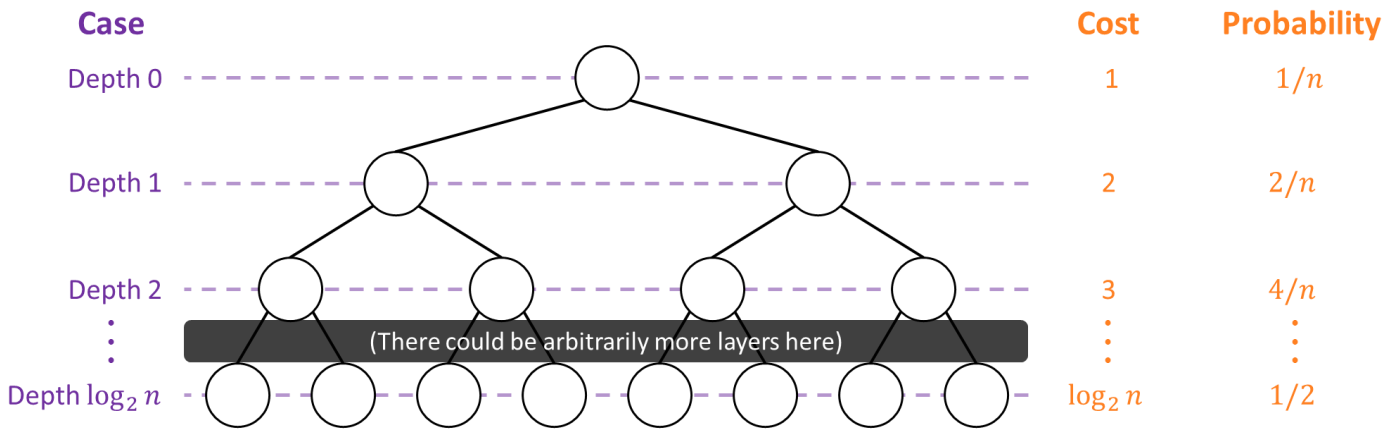
In some scenarios, run time can vary based on program state or random chance. To best characterize potential run times, we often consider the following three cases:

- **Best case**: The scenario in which our metric is run the fewest number of times.
- **Worst case**: The scenario in which our metric is run the most number times.
- **Expected (or average) case**: The average number of times our metric is run, calculated as a weighted sum across all possible cases.

In each case, all five types of run time analysis apply. For example, the best case may have a cost of $2n + 5$, which is a big O complexity of $O(n)$.

For example, let's consider the cost of finding a value in a perfect binary search tree with n elements and a metric of comparisons, assuming that the value exists in the tree.

- **Best case**: The value is stored in the root, which requires 1 comparison**.**
- **Worst case**: The value is stored in a leaf, which requires $\log_2(n)$ comparisons because the path from the root to any leaf contains $\log_2(n)$ nodes in a perfect tree.
- **Expected case**: We consider the value residing at each depth as a separate case and calculate the cost and probability of each case.



Expected time is the sum of the cost of each case (number of comparisons) multiplied by the probability of that case.

$$Expected\ Time = \left(1 * \frac{1}{n}\right) + \left(2 * \frac{2}{n}\right) + \left(3 * \frac{4}{n}\right) + \cdots + \left(\log_2(n) * \frac{1}{2}\right) \approx \log_2(n) - 1$$

Thus, we need to perform an average of $\log_2(n) - 1$ comparisons to find our value.

Possible Exam-style Questions

1. Given some code with random or unknown information:
    a. Identify the best and worst cases.
    b. Derive an exact cost for the best, worst, and expected cases.
    c. Determine the asymptotic (big O) run time of the best, worst, and expected cases.
2. Explain why expected cost is not simply the average between the best- and worst-case costs.

Suggested Exercises

Determine the best, worst, and expected cost of searching for an element in a linked list if the element exists in the linked list. Determine the best and worst-case cost of searching for an element in a binary search tree if we do not guarantee that the tree is balanced.

3.5 AMORTIZED ANALYSIS

If a single operation has an average cost of c_{avg}, then n operations will on average cost nc_{avg}. However, this is not an upper bound on the cost of n operations. Average cost generally applies to any situation with variable cost, so if the cost varies due to probability or external factors, we could get the worst-case cost every time if we are very "unlucky". If c_{worst} is the worst-case cost of a single operation, nc_{worst} is always an upper bound on the cost of n operations.

Amortized analysis is a special type of run time analysis which sometimes allows us to establish a tighter upper bound on total cost than nc_{worst}. Specifically, if we calculate an amortized cost $c_{amortized}$ of a single operation, then a series of n operations are guaranteed to have a cost less than or equal to $nc_{amortized}$. Amortized analysis is applicable for operations where cost is variable but predictable. For example, insert cost into a resizable array is variable (based on whether we trigger a resize), but predictable because we know exactly when a resize will occur.

Note that an amortized cost is itself not an upper bound on the cost of a single operation. For example, we will show in Example 2 that insert into a resizable array has an amortized cost of $O(1)$. This does not mean that insert is always constant time; an insert that triggers a resize will take $O(n)$ time. However, it does guarantee that inserting n elements into a resizable array will have a total cost of $O(n)$.

To summarize, if an operation has an amortized cost $c_{amortized}$, performing n of those operations will have a total cost that is bounded above by $nc_{amortized}$, even though some operations will cost more than $c_{amortized}$.

Ruble Method

While there are several ways to perform amortized analysis[14], the ruble (aka banker's) method is perhaps the easiest to understand. A ruble is a "currency" which can pay for any constant amount of work. We charge the user a certain number of rubles for each operation and show that we always have a ruble to pay for each constant piece of work. If we perform n operations, we will charge at most $n * (cost\ of\ the\ most\ expensive\ operation)$ rubles, meaning we have done at most that many pieces of constant work.

Specifically, the ruble method consists of the following steps:

1. Choose how many rubles to charge for each operation.
2. Decide how these rubles are spent.
3. Show that each constant piece of work is accounted for by a ruble.

For the third step, it is often helpful to state and enforce an *invariant* (something that is always true).

Example 1: Multi-pop stack

A multi-pop stack is a data structure that supports `pop` and `push` just like a stack, but also supports `multiPop(k)` which pops the top k elements from the stack. If we perform n operations, a multi-pop operation can take at most $O(n)$ time, since we can push $n-1$ elements and then pop them all at once. However, we will show that each operation has amortized constant run time.

We will charge the following number of rubles per operation:

- **Push**: 2 rubles
- **Pop**: 0 rubles
- **Multi-pop**: 0 rubles

When we push an element to the stack, it uses the first ruble to pay for the constant work of adding itself to the stack and keeps the second ruble in its pocket. When we call `pop` or `multiPop`, each removed element uses the ruble in its pocket to pay for the constant work associated with removing itself from the stack. This maintains the invariant that every element on the stack has a ruble in its pocket, meaning that every element can always pay for its own removal.

[14] If you are a student at Harvey Mudd, you will have the opportunity to learn more methods in Algorithms (CS140).

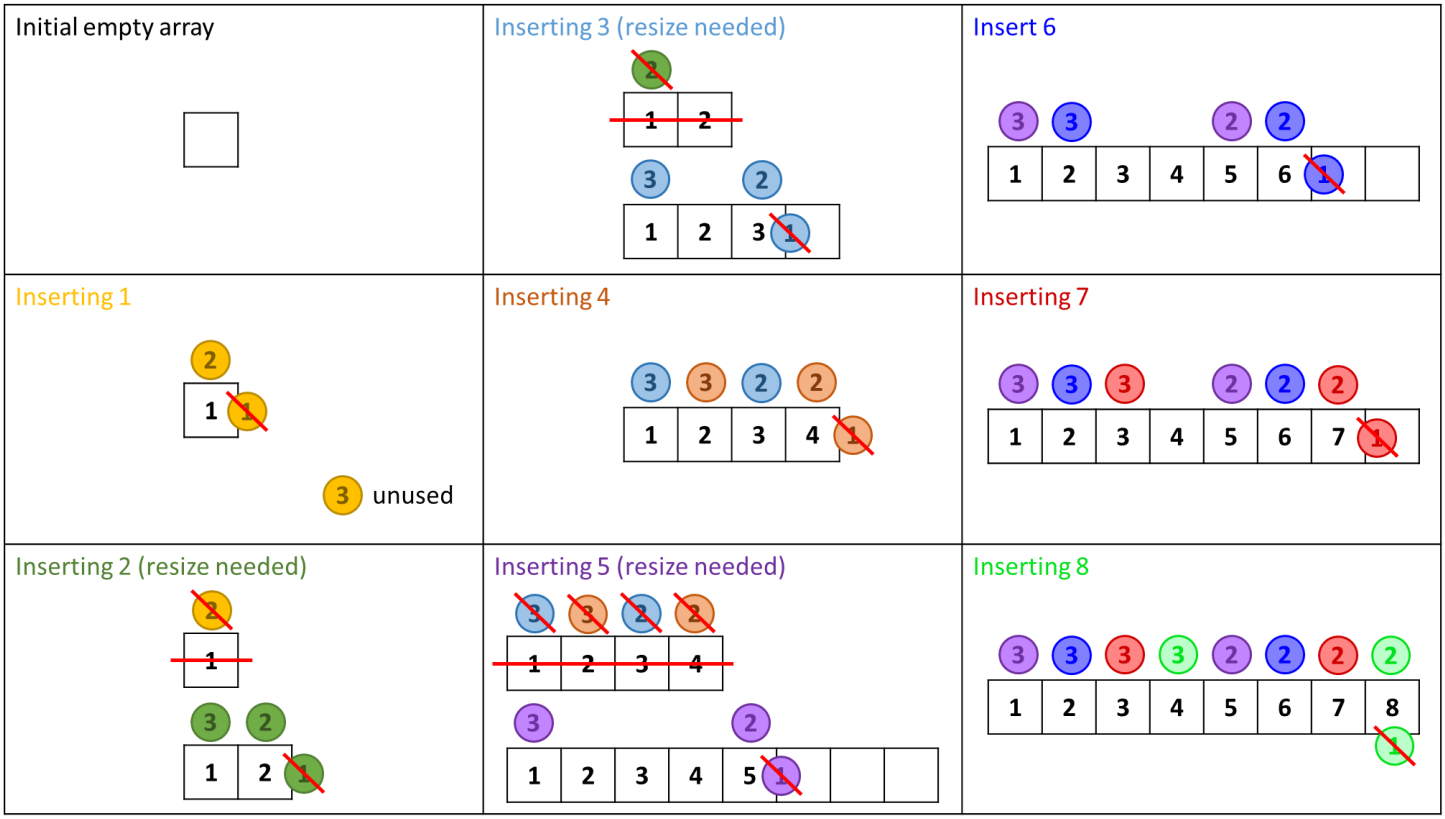
If we perform n operations, this will cost at most $2n$ rubles (since the most expensive operation costs 2 rubles), meaning that n operations will perform at most $2n$ pieces of constant work. Thus, performing n operations has an asymptotic run time of $O(n)$, so each operation has an amortized constant run time.

Example 2: Resizable Array

In a resizable array, `push_back` places an element in the first empty space in the array. If we `push_back` into a full array, we must first create an array that is twice as large and copy over each item. This worst case takes $O(n)$ time, but we will show that `push_back` takes amortized constant time. For the sake of this analysis, we will assume that the resizable array starts with an array of length 1.

We will charge 3 rubles for `push_back`: one ruble pays for the constant work of writing the element into the array, the second ruble is stored in the element's pocket, and the third ruble is given to the first element in the array which does not already have a ruble in its pocket. This maintains the invariant that if the array is full, every element will have a ruble in its pocket. When we insert into a fully array, each element uses its stored ruble to pay for the constant work of copying itself to the new array.

Calling `push_back` n times costs $3n$ rubles, meaning that at most $3n$ pieces of constant work are done. Thus, inserting n elements has an asymptotic run time of $O(n)$, so each individual `push_back` operation takes amortized constant time (specifically, an amortized cost of 3).



This diagram shows the rubles used in the first eight inserts into a resizable array. Rubles are shown as colored circles, with the color denoting the insert for which the ruble was charged. Each ruble is crossed out in red when it is spent.

Possible Exam-style Questions

1. Explain in your own words the difference between amortized cost and average cost.
2. Given a scenario, use the ruble method to determine the amortized cost of an individual action.
3. Given a situation, decide whether amortized analysis is applicable.

Suggested Exercises

Consider a binary counter, a number represented in binary which counts up from zero. Every time we call the `increment` function, the counter's value increases by one, and it takes constant time for a single bit to toggle between 0 and 1 (bits cannot toggle in parallel). First, show that an arbitrary `increment` operation can take at worst $O(\log n)$ time. Next, show that `increment` has an amortized constant cost using the ruble method.

Appendix

A. VERSION CONTROL

In many large-scale software projects, multiple developers must work on the same set of files. A *version control system* organizes and tracks collections of changes to enable multiple people to modify the same files without conflict. These changes are organized into a history, allowing us to see how a file has developed over time and revert to previous versions if necessary. Nearly all large software companies use some type of version control.

Git is one of the most popular version control systems for programmers and is used by both individual developers and major corporations like Microsoft. *GitHub* is a website which can be used to store projects versioned with Git. The distinction between these two is important: Git is a system for keeping track of changes, while GitHub is simply one of many places we can store projects versioned with Git. We can version a project with Git but store it somewhere other than GitHub.

This textbook explores version control through the context of a project on GitHub versioned with Git. While several other version control systems exist such as Apache Subversion and Perforce Helix, they operate with the same core principles as Git. Once you understand Git, you can easily understand most other version control systems.

Fundamentals

A *repository* is a project in which changes are tracked with Git. If we are using GitHub, a version of this repository will exist in the cloud on GitHub's website. We can *clone* this cloud version to create a local copy of the repository on our machine. At any point in the future, we can *pull* changes from the cloud to bring our local repository up to date with the repository on GitHub.

After we make changes to files in our local repository, we need to organize these changes into a revision which can be added to the version history. We first add the changes we care about to the *staging area* and then *commit* these changes to create a revision. Each commit includes a message describing the changes made by the revision. Finally, we can *push* one or more committed revisions back to GitHub so that the version on the cloud is up to date with the version on our computer.

When we pull new changes from the cloud, they may attempt to overwrite some of our local changes that we have not yet pushed. This creates a *merge conflict*, since the changes on GitHub cannot be automatically merged with our local changes. We can resolve these merge conflicts in a text editor or development environment by manually choosing which changes to keep. By pushing this resolution, we can bring our local repository back in sync with the repository on GitHub.

Branches

To reduce merge conflicts and better organize changes, we can use *branches* to create multiple versions of a repository. These branches exist on both GitHub and our local repository, and each branch keeps track of its own changes. When we commit a change to one a branch, it does not affect any other branches.

By default, every repository starts with a single branch called *master*. Many repository owners choose to create a *develop* branch in which contributors can work on new features before they are merged into master. If we have a large project with several different features, we may create a branch for each feature so that each can be worked on independently. Once the work in a branch is finished, we create a *pull request* to merge it into another branch, which will merge the unique changes of the comparing branch into the base branch. For example, once our work in develop is completed, we can create a pull request to merge develop into master.

Git Commands

While there are several ways to use Git, one of the easiest is through the command line. Here are some of the most important Git commands:

- `git clone <web URL>`: Downloads the repository at the given URL to our machine, creating a local repository at the current directory.
- `git pull`: Updates our local repository with the changes on GitHub for the current branch.
- `git add <filename>`: Adds the changes to the given file to the staging area for the current branch.
- `git add .`: Adds all changes in our working directory to the staging area for the current branch.
- `git commit -m "<message>":` Gathers all of the changes in the staging area into a new revision for the current branch with the provided message describing those changes.
- `git push`: Uploads the local commits for the current branch to the GitHub repository.

Here are some of the most important commands relating to branches:

- `git branch <branch name>`: Creates a new branch in our local repository with the given name.
- `git checkout <branch name>`: Switches to the given branch.
- `git branch -d <branch name>`: Deletes the given branch.

Here are other helpful commands:

- `git rm <filename>`: Removes a file from our working directory and adds this change to the staging area.
- `git checkout -- <filename>`: Reverts a file to its state in the most recent commit.
- `git status`: Summarizes information about the current branch including the number of commits, the files with changes, and the files in the staging area.
- `git branch`: Shows all branches with an asterisk next to the current branch.

Best Practices

The following best practices help us avoid merge conflicts and gain the most benefit from version control:

- **Begin every work session by pulling**: This helps reduce the probability of merge conflicts by making sure that our local repository is up to date with the GitHub repository.
- **End every work session by pushing**: This also helps reduce merge conflicts by ensuring that other people do not make changes that conflict with ours before the next time we work.
- **Commit frequently**: Frequent, smaller commits create a more granular history, making it easier to understand how a file has changed over time. This also provides more "checkpoints" that we can revert to if necessary.
- **Do not commit generated files**: In general, a repository should only include source files (such as `.cpp` and `.hpp` files) and should not include files which can be generated from these source files (such as `.o` files or executables). This helps avoid merge conflicts and keeps our repository smaller and thus faster to work with.
- **Keep master clean**: In most circumstances, it is best to only push changes to master when they have been well tested. Features that are in progress should go on a different branch.
- **Do not push breaking changes to branches that other people use**: If we are in the middle of fixing a problem, we should not push broken code to a branch that other people use. Instead, we should create a new feature branch that only we use. In general, it is best to do all of our work in personal branches and merge these branches into shared branches when they are ready.

You can learn more about Git and GitHub at https://help.github.com/en/github.

Possible Exam-style Questions

1. Given a situation and a desired effect, describe which Git commands should be used in which order.
2. Given a situation and a series of Git commands, describe which files have been changed and where.
3. Explain the value of branches and identify a situation in which branches would be helpful.
4. Given a situation, identify whether a merge conflict has occurred and if so, explain how to resolve it.

Suggested Exercises

Create a new repository on GitHub and complete the following tasks:

1. Create a local copy of the repository on your machine.
2. Make changes to multiple files and push these changes.
3. Create a develop branch and make changes on develop.
4. Create a pull request from develop into master.
5. Create a second local copy of the repository on your machine and create conflicting changes between these two copies. Push and pull these changes to create a merge conflict and resolve the conflict.

B. INLINE FUNCTIONS

As we know from our memory model, we must create a new stack frame every time we call a function, which requires some overhead. If a function compiles to few assembly instructions, it may make more sense to simply copy the body of the function to every place that we call it, which is referred to as *inlining* the function. The compiler makes this decision for us by considering:

- The number of assembly instructions to which the function compiles.
- The number of places where the function is called.
- The optimization level used when compiling (determined by the `-Ox` flag).

For example, suppose that we compile the following simple program.

```
int triple(int x) {
  return x * 3;
}

int main() {
  int x = 3;
  std::cout << triple(x) << std::endl;
  return 0;
}
```

If the compiler decides to inline `triple`, the result would be equivalent to compiling the following program.

```
int main() {
  int x = 3;
  std::cout << x * 3 << std::endl;
  return 0;
}
```

We can also encourage the compiler to inline a certain function by adding the `inline` keyword before the function.

```
inline int triple(int x) {
  return x * 3;
}
```

The `inline` keyword is nonbinding–the compiler is still allowed to define and call the function normally it believes that inlining is a bad idea. The hint simply makes the compiler more likely to inline the function.

If we inline a function, we need to know the definition at compile time or else we cannot swap out the body. Thus, the separate compilation paradigm of putting declarations in the `.hpp` and definitions in the `.cpp` prevents inlining. For example, suppose that `A.cpp` includes `B.hpp`, which contains an inline function `foo` defined in `B.cpp`. In order to inline `foo`, `A.cpp` must have access to `foo's` definition when it is compiled, but it would not have access to it until the linking step. As a result, we must define inline functions in the `.hpp` file, not the `.cpp` file.

Possible Exam-style Questions

1. Given a function and its usage, reason about whether it should be inlined.
2. Explain why inline functions are defined in the `.hpp` file instead of the `.cpp` file.

Suggested Exercises

Write an inline function and code which uses that function.

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`.