Contents

1 Topic List

1.1 Chapter 13

1.1.1 Procedural and Object Oriented Programming

The distinction made between “vocabulary” and “important concepts” is generally somewhat arbitrary, but if there is a one or two word phrase that has important meaning, it will typically be placed under the Vocabulary section of each entry.

Vocabulary

A. **Procedural Programming**: Data is stored in a collection of variables and/or structures with a set of functions that perform operations on the data.

B. **Object-oriented Programming**: Data is stored in a single unit that contains both data and procedures (functions).

C. **Object**: The entity that contains the data and procedures.

D. **Attribute**: The data contained within an object.

E. **Member function**: The procedures an object performs.

F. **Encapsulation**: The combining of data and code into a single object.

G. **Data hiding**: An object’s ability to hide its data from code that is outside the object.

H. **Object Reusability**: The ability for components to be reused by other programs with limited modifications.

I. **Class**: Code that specifies the attributes and member functions that a particular object may have. It serves a similar purpose to a blueprint.

J. **Instance**: An object that has been created from a class. Each instance of a class is its own entity with its own attributes that store its data.

1.1.2 Introduction to Classes

Vocabulary

A. **Access Specifier**: Key words that specify what can access members of a class.

B. **Public Member Function**: These allow users to access the private member variables and perform other tasks with an object of a class.

C. **Prototype**: The declaration of something without actually defining its behavior.

D. **Scope Resolution Operator**: Two colons placed side by side (i.e. `::`) that identify something as a member of a class.

E. **Accessor**: A member function that gets a value from a member variable but does not modify it. They may also be referred to as getter functions.

F. **Mutator**: A member function that modifies a member variable. They may also be referred to as setter functions.

Important Concepts

1. In C++, the class is the construct primarily used to create objects.

2. The members of classes are private by default.

3. Private members *cannot* be accessed outside of the class. Public members *can* be accessed outside of the class.

4. Private and public members can be declared in any order; there are no rules governing the order in which they are placed.

5. The `const` keyword identifies a member function as an accessor. This keyword guarantees that a member function will not modify any data within the class.

1.1.3 Defining an Instance of a Class

Vocabulary

A. **Instantiation**: The defining of a class object. This creates an instance of the class.

B. **Garbage**: The random values that may appear in a member variable that has not been initialized with a value.

C. **Stale Data**: Data that is no longer correct because the data that it is dependent on has been modified.

Important Concepts

1. Class objects are not created in memory until they are defined.

2. Class members are accessed with the dot operator.

3. Member variables are not automatically initialized to 0.

4. Using a mutator function on one instance will not modify the data of another instance of the class.

5. The results of calculations should be retrieved with accessor functions, rather than stored as an attribute, to prevent stale data.

6. Pointers can be defined to point to class objects.

7. If a pointer contains an object, you can access its members with the `'->'` operator. The statement `'a -> b'` is functionally equivalent to `'(*a).b'`.

8. Class object pointers can be used to dynamically allocate objects. In other words, you can use the `'new'` operator to dynamically allocate an instance of a class.

9. After a dynamically allocated object is freed with the `'delete'` operator, you should assign a value of `'NULL'` or `'nullptr'` to the object to prevent errors.

1.1.4 Why Have Private Members?

Important Concepts

1. Objects should protect their important data by making it private and providing a public interface to access that data.

2. The public interface for an object's data can protect it from invalid data.

1.1.5 Separating Class Specification from Implementation

Vocabulary

A. **Class Specification File**: The place where the declarations (but not definitions) of classes are stored. It is typically named after the class itself with a .h extension.

B. **Class Implementation File**: The place where member functions for a class are stored. It is typically named after the class itself with a .cpp extension.

C. **Include Guard**: A type of preprocessor directive that prevents the header file from being included more than once. There are three "keywords" associated with include guards: `#ifndef`, `#define`, and `#endif`. `'#ifndef'` determines whether a constant, typically something like `'SPIDER_H'`, where Spider is the name of the class, has been defined yet. If it hasn't been defined, the `'#define'` directive will define it. After everything has been defined, `'#endif'` will close the block.

Important Concepts

1. When header files are included, double-quotes are used instead of angle brackets.

2. The implementation files cannot be compiled into an executable, as they are not complete programs.

Code Examples

```
// Circle.h
// Specification file
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle {
  private:
  double radius;
  public:
  void setRadius(double);
  double getRadius() const;
};

#endif CIRCLE_H
```

```
// Circle.cpp
// Implementation file
void Circle::setRadius(double r)
{ radius = r; }

double Circle::getRadius() const
{ return radius; }
```

1.1.6 Inline Member Functions

Vocabulary

A. **Inline Function**: A function that is defined inside the declaration of a class, rather than in an implementation file, for example. This eliminates the need to use the class name and scope resolution operator. Inline functions are generally faster than non-inline functions, but result in larger executable files.

B. **Inline Expansion**: When the compiler replaces a call to an inline function with the code of the function itself.

1.1.7 Constructors

Vocabulary

A. **Constructor**: A member function that is automatically called when an object is instantiated. All constructors have the same name as the class they are in. They can be used to initialize the member variables of the object and other setup operations.

B. **Member Initialization List**: An alternate method of initializing the variables of a class. In some situations, it can be faster than initializing the member variables inside of the constructor body.

C. **In-Place Member Initialization**: Allows you to initialize the variables of a class within its declaration. This is exclusive to C++ 11 and later.

D. **Default Constructor**: A constructor that takes no arguments. If no constructor is defined at all, a default constructor that does nothing will automatically be generated by the compiler.

Important Concepts

1. Constructors do not have return types because they cannot return any values.

2. Constructors may be automatically generated, but it is still best practice to define constructors for classes.

3. Creating a pointer to an object that is initialized with `'NULL'` or `'nullptr'`, the constructor does not execute because it does not actually create an object.

4. When an object is created with the `'new'` operator, its default constructor is automatically executed.

1.1.8 Passing Arguments to Constructors

Important Concepts

1. Constructors can have parameters and can accept arguments when objects are created.

2. Constructors can have default arguments. The value(s) can be listed in the parameter list of the declaration or in the header.

3. A constructor that has default arguments for all of its parameters will automatically become the default constructor for the class, and no arguments need to be called when creating an object with that constructor.

4. If every constructor in a class requires arguments, there will be no default constructor and arguments *must* be used when creating an object.

1.1.9 Destructors

Vocabulary

A. **Destructor**: Member functions with the same name as the class, preceded by a tilde character.

Important Concepts

1. Destructors are automatically called when an object is destroyed. A common use-case for destructors is to free memory that was dynamically allocated within a class object.

2. Destructors cannot accept arguments and cannot have a return type.

3. When you use the `'delete'` operator on a pointer to a dynamically allocated object, the destructor will automatically be called on that object.

1.1.10 Overloading Constructors

Vocabulary

A. **Constructor Delegation**: Calling different constructors within a constructor to perform certain steps in creating an object. This is exclusive to C++ and beyond.

Important Concepts

1. Just like any other functions, including member functions, constructors in C++ can be overloaded. This allows a class to have multiple constructors. As long as there is a different list of parameters, there can be multiple constructors.

2. A class may only have one default constructor and only one destructor. This extends to constructors with all default parameters: you cannot have both a constructor with no arguments and a constructor with all default parameters.

3. All member functions except the destructor can be overloaded.

1.1.11 Private Member Function

Vocabulary

A. **Private Member Function**: A member function that can only be used by functions within the class. Like private member variables, it cannot be called or accessed by functions outside of the class.

Important Concepts

1. Private member functions may only be called by other member functions in the class.

2. Member functions that could harm the data of an object if used improperly should be private.

1.1.12 Arrays of Objects

Important Concepts

1. You are able to create and work with arrays of class objects.

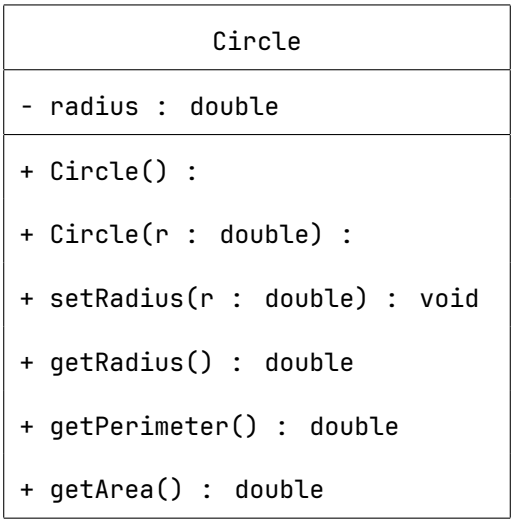
2. When no initializer list is used, the default constructor will be called for each object in the array.

3. In order to define an array of objects with non-default constructors, each object must be individually defined with an initializer list. If there are fewer initializers in the list than there are spaces in the array, the remaining objects will be created with a default constructor.

4. Objects in an array are accessed like anything else in an array.

1.1.13 Unified Modeling Language

1. **Unified Modeling Language (UML)**: A standard set of diagrams used to graphically depict object-oriented systems. There are three sections of a UML diagram: the top stores the class name, the middle stores the member variables, and the bottom contains the member functions. A `'-'` before a member name indicates that it is private, while a `'+'` indicates that it is public. The data types of functions and variables is indicate with a colon followed by the type. This also applies to function parameters. A full example is provided below and an explanation is given as well.



1. The name of the class is `'Circle'`.

2. The class only has one attribute, `'radius'`. There is a minus beside it, so it is private. Finally, there is a colon followed by the `'double'` type, so we know that its type is `double`.

3. There are six member functions, all of which are public: two constructors, a mutator function, and three accessor functions. We know they are public because of the `'+'` signs before the name.

We know that two of the functions are constructors because they have the same name as the class. We know which functions are accessors and mutators because of their names.

4. The accessor member functions all return doubles, as indicated by the colon followed by the `'double'` type keyword, the mutator member function returns `'void'`, and the constructors return nothing, as indicated by the lack of a type keyword.

1.2 Chapter 14

1.2.1 Instance and Static Members

Vocabulary

A. **Instance Variable**: These are variables that each instance of a class has, and only that class has it. They are entirely separate from the instance variables of any other class. If one instance's attribute of a certain name is modified, it will not affect the attribute of any other classes.

B. **Static Member Variable**: These are variables that all instances of a class share. They do not belong to any instance in particular, and may be accessed and modified by any instance of the class. There will only ever be one copy of a static member variable stored in memory at any given time. Unlike static member functions, static member variables can be accessed and modified by non-static functions by any instances of the class.

C. **Static Member Function**: These are functions that can only operate on static member variables, and not on instance variables.

Important Concepts

1. All instances of a class have access to the static variables of the class.

2. Static member variables may be stored and modified even if there are no instances of the class defined.

3. Static member functions may be called even if there are no instances of the class defined.

4. C++ automatically stores 0 in all uninitialized static member variables.

5. Static member variables are actually defined outside of the class declaration, and their lifetime is the lifetime of the program, not of any instances of the class.

6. Static member functions operate on static member variables before any instances of the class are ever defined.

7. There are two ways to call static member functions: If there are instances of the class defined, you can call them like normal. If there are no instances of the class defined, however, you must use the name of the class, followed by the scope resolution operator, and finally the name of the static member function.

1.2.2 Friends of Classes

Vocabulary

A. **Friend function**: Functions that are not members of a class, but still have access to the private members of the class. They may be stand-alone functions or they can also be members of other classes.

B. **Forward Declaration**: Informing the compiler that something will be declared later in the program. They are required when a piece of code must be processed before the code of whatever has been forward declared, such as when a friend function or class is being used.

Important Concepts

1. Both functions and entire classes can be declared friends of another class.

1.2.3 Memberwise Assignment

Important Concepts

1. Memberwise assignment occurs when an object is initialized with another object's data. Each member of one object is copied to its counterpart in the other object.

2. If memberwise assignment occurs on a pointer, there will be two pointers that point to the same location in memory. If the data that is pointed to is deleted by one object, the other object won't know and this will create an error.

1.2.4 Copy Constructors

Vocabulary

A. **Copy Constructor**: A special constructor that is automatically when an object is initialized with another object's data. It has the same form as other constructors, but uses a reference parameter of the same class type as itself. It is *only* called when an object is created, not just anywhere that assignment is occurring.

Important Concepts

1. You must use a reference object as the parameter in copy constructors.

2. Since copy constructors are used to copy data, constant reference parameters should be used, instead of regular reference parameters, to prevent inadvertent modification of the original object's data.

3. If reference parameters are not used, an infinite loop of object creations will occur, leading to a crash.

4. Memberwise assignment is performed by the default copy constructor.

1.2.5 Operator Overloading

Vocabulary

A. **`this` Pointer**: The `'this'` pointer is a special pointer that is automatically passed as a hidden argument to all non-static member functions, and always points to the instance of the class making the function call.

B. **Operator Functions**: Special member functions of a class that define how a particular operator works. They appear as the word `'operator'` followed by whatever operator is being defined, such as `'operator='`, which is the assignment operator.

C. **Self-assignment**: A type of assignment where an object is assigned to itself. Under certain circumstances, such as when assignment deletes dynamically allocated data, this can lead to major problems.

D. **Dummy Parameter**: A nameless, typically temporary parameter.

Important Concepts

1. Operator overloading allows you to redefine how operators work when used with class objects.

2. It is commonplace to name the parameter of the operator function being used `'right'`, as it falls on the right side of the operator when it is called.

3. Operator functions can also be used like other member functions. An example usage of this is as follows: `objectOne.operator=(objectTwo);`. This statement is identical in function to `'objectOne = objectTwo'`.

4. The assignment operator should have checks to prevent self-assignment. This is as simple as the following statement: `if (this != &right)`. As long as the two objects are not equal, then whatever is contained within the if-statement will execute.

5. In order to return the object that is pointed to by the `'this'` pointer, you must dereference it, then return it as normal: `return *this;`.

6. Because there is no separate operator for them, in order to overload a postfix operator, you must use a dummy parameter and a temporary local variable. The dummy parameter is like any other function parameter, except it is only of type `'int'`. The temporary local variable will be an object of the class that is created for the sole purpose of modifying its data and returning it. This is in contrast to the prefix operator, where no temporary local variable is needed and no parameter is passed to the operator function.

7. The only difference between overloading a relational operator and any other binary operator is that relational operators should always return a boolean value, rather than an object of the class.

8. The stream insertion `'<<'` and stream extraction `'>>'` operators can also be overloaded. The insertion operator must return the type `'ostream'` and must have a reference parameter of type `'ostream'` and a reference parameter of the object's class. The extraction operator is very similar, but uses `'istream'` in place of `'ostream'`.

9. The subscript operator (`'[]'`) can also be overloaded like any other operator. The primary purpose of overloading this operator is to create array-like data structures, such as an array that performs bounds checking (unlike the regular array in C++). It can only have one parameter, and that parameter is what is placed inside the brackets when calling the subscript operator. It is common to name this parameter `'sub'`, as it is the "subscript" value.

1.2.6 Object Conversion

Important Concepts

1. Special operator functions can be written that will convert a class object to any other type. To overload these operators, you simply use `'operator'` followed by the type to convert to, such as `double` or `int`.

1.2.7 Aggregation

Vocabulary

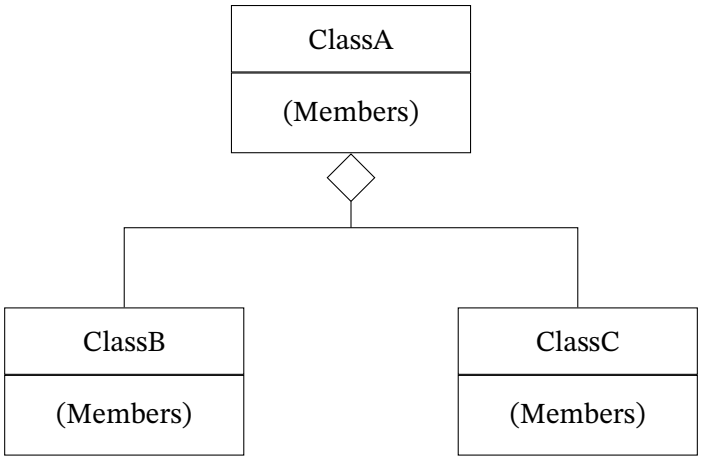
A. **Aggregation**: When a class contains an instance of another class.

B. **Aggregate Class**: The class that contains instances of other classes.

Important Concepts

1. Aggregate classes can be use member initialization lists to call the constructors for each of its member objects.

2. Aggregation can be depicted in UML diagrams by a diamond shape connecting one or more separate UML diagrams.



1.2.8 Rvalue References and Move Semantics

Vocabulary

A. **Lvalue**: Values that persist beyond the statement that created them *and* have names that make them accessible to other statements in the program.

B. **Rvalue**: Values that are temporary and cannot be accessed beyond the statement that created them.

C. **Lvalue Reference**: This is what we generally consider to be a "reference". In C++ 11 and beyond, these are referred to as lvalue references, instead of just 'references'.

D. **Rvalue Reference**: These are a certain type of reference that can only refer to temporary objects that are about to be destroyed and would otherwise have no name. They are used almost identically to regular lvalue references, except with a double ampersand instead of a single ampersand (&& versus &). They cannot refer to data that is an referable by an lvalue.

E. **Move Assignment Operator**: An operator that allows you to avoid unnecessary operators by directly transferring the data stored by one object to another, rather than copying the data then deleting the original data.

F. **Move Constructor**: A special type of constructor that performs move assignment. It should be used when creating objects by initialization from temporary objects. If a move assignment operator is added to a class, then a move constructor should also be added to the class.

Important Concepts

1. Move operations transfer resources from source objects to target objects. They are suitable for situations where the source object is temporary and will be deleted shortly after the move operation.

2. By assigning a value to an rvalue reference, you allow other code to access that value – even other rvalue references.

3. Any time a class contains a pointer or reference to outside data, a copy constructor, a move constructor, a copy assignment operator, and a move assignment operator should be included as well. These will reduce the number of operations performed and increase the performance of the code.

4. Move constructors and assignment operators are automatically called when they are appropriate, so you do not need to do any extra work beyond creating them to reap their benefits.

5. If you create your own version of a move constructor, move assignment operator, copy constructor, or copy assignment operator, the compiler will no longer automatically generate the code for any of them. This means you will then have to create your own versions of each one.

1.3 Chapter 15

1.3.1 What is Inheritance?

Vocabulary

A. **Inheritance**: The ability of one class to inherit certain parts of a parent, or base class. These parts are limited to member variables and functions in versions of C++ before C++ 11, but in C++ 11 and onward, classes may inherit some of the constructors of their base class(es).

B. **Base Class**: These are the classes which other classes inherit data and functions from. They may also be referred to as parent classes.

C. **Derived Class**: These are the classes which inherit data and functions from other classes. They may also be referred to as child classes.

D. **"Is A" Relationship**: When a class is inheriting from a parent class, there is an "is a" relationship between them. Generally, parent classes are generalized versions of something, while the children, or derived classes, are more specialized versions of them. So, for example, a spider *is an* Arachnid. A human *is a* mammal. These relationships are created by inheritance.

E. **Base Class Access Specification**: This is a type of access specification that determines what and how derived classes will inherit the members of the base class.

1.3.2 Protected Members and Class Access

Vocabulary

A. **Protected Member**: A third type of member variable or function. They are like private members, except they can be accessed by functions in a derived class. Nothing else outside of the base and derived class' members can access these members.

Important Concepts

1. A table is given below that indicates how base class specification affects how derived classes have access to the base class' members. Something to remember is that private members of the base class are *always* inaccessible to its derived classes. Another thing to keep in mind is that all inherited members are *at least* as hidden as the base class specification that is used. For example, private base class access specification will make protected and public members private. Protected will make protected and public members protected. The one exception is public specification, where protected and public members remain as they are.

Access Spec. in Base Class → Base Class Access Spec. ↓	Private Member	Protected Member	Public Member
`: private BaseClass`	Inaccessible	Private	Private
`: protected BaseClass`	Inaccessible	Protected	Protected
`: public BaseClass`	Inaccessible	Protected	Public

1.3.3 Constructors and Destructors in Base and Derived Classes

Vocabulary

A. **Constructor Inheritance**: This is when derived classes inherit the constructors of their parent class(es). There are three constructors that cannot be inherited: the default constructor, the copy constructor, and the move constructor. Any others are able to be inherited. Even if a class inherits a constructor from another class, it can still have its own constructors.

Important Concepts

1. The base class constructors is called before the derived class constructor. Destructors are called the other way around, with the derived class' destructor being called before the base class' destructor.

2. When defining the constructor of a derived class, you can pass arguments to the base class' constructor. Any value that is in the proper scope may be passed as an argument to the base class.

1.3.4 Redefining Base Class Function

Important Concepts

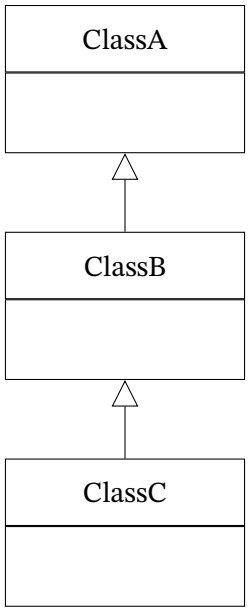
1. Base class member functions can be redefined in derived classes. This can allow for increased or more specialized functionality within the derived classes.

2. Recall that for member functions of a derived class that have the same name as the member functions in a base class, you must use the scope resolution operator for the compiler to be able to distinguish between them.

1.3.5 Class Hierarchies

Important Concepts

1. Base classes can also be derived from other classes. There is no limit to how many classes can be inherited from, so it is valuable to create charts like the one below. The more general the class, the higher it is towards the top of the chart.



1.3.6 Polymorphism and Virtual Member Function

Vocabulary

A. **Polymorphism**: Allows object reference variables or pointers of different types to call the correct member functions, depending on the type of the object being defined. It is named polymorphism because it allows the functions of base classes to take whatever form is necessary in their derived classes. In order to use polymorphism, functions must use reference variables or pointers – never objects passed by value.

B. **Virtual Member Function**: A type of member function that enables dynamic binding, allowing it to be overridden in derived classes.

C. **Binding**: The process that allows C++ to determine which function to call when a member function of an object is called.

D. **Static Binding**: The type of binding where function calls are matched to functions at *compile time*.

E. **Dynamic Binding**: The type of binding where function calls are matched to functions at *runtime*. This is enabled by making a function virtual.

F. **Redefining**: This process occurs when a non-virtual function is modified in a derived class.

G. **Overriding**: This process occurs when a virtual function is modified in a derived class.

H. **Virtual Destructor**: Reverses the order of execution of destructors in derived classes. Without the virtual destructor, pointers and dynamically allocated objects that exist in the derived class will not be properly deleted, causing many problems.

I. **`'override'` Keyword**: Tells the compiler that the function is supposed to override a function in the base class. If no overriding occurs, an error will occur.

J. **`'final'` Keyword**: Tells the compiler that a function cannot be overridden in a derived class. If a derived class attempts to modify a function declared `'final'`, the compiler will throw an error.

Important Concepts

1. Pointers to base classes can be assigned the address of a derived class object. The downside to this approach is that these pointers are only able to use base class members. If the derived class has extra member functions not found in the original, then you will not be able to use those extra functions.

2. If a class has virtual functions, it should also have a virtual destructor.

1.3.7 Abstract Base Classes and Pure Virtual Functions

Vocabulary

A. **Abstract Base Class**: A type of class that is not instantiated itself, but serves purely as a base class for other classes. It is a generic class from which other classes are defined. A class becomes an abstract base class by the inclusion of one or more pure virtual functions.

B. **Pure Virtual Function**: A special type of function that has no body or definition. They *must* be overridden in derived classes. Additionally, if a class contains a pure virtual function, it cannot be instantiated as an object.

1.3.8 Multiple Inheritance

Vocabulary

A. **Multiple Inheritance**: This is when a derived class inherits from two or more base classes. This is different to chains of inheritance, where there is ultimately only a single base class. To inherit from multiple classes, you simply separate each base class with a comma.

Important Concepts

1. There can be ambiguity when calling the member functions of base classes from a derived class, so the scope resolution operator (`::`)should be used when functions have the same name in order to inherit from the correct class. When this occurs, the derived class should also always redefine or override the member functions in question. If none of these precautions are taken, the compiler will throw an error.

1.4 Chapter 18

1.4.1 Introduction to the Linked List ADT

Vocabulary

A. **Linked List**: A dynamically allocated data structure that stores data by connecting nodes together with a series of pointers. Because linked lists are stored non-contiguously, you can add and remove nodes at any point, without having to reallocate memory or copy the list to a new structure.

B. **Self-referential Data Structure**: A data structure that contains a pointer to an object of the same type as is being declared.

1.4.2 Linked List Operations

Vocabulary

A. There are five basic operations that can be performed on a linked list: traversal, appendage, insertion, deletion, and destruction.

1.4.3 Variations of the Linked List

Vocabulary

A. **Doubly Linked List**: A variation of the regular linked list where each node has two pointers instead of one: one points to the previous node, while the other points to the next node.

B. **Circularly Linked List**: A variation of the regular linked list where each node still only has one pointer, but the last pointer in the chain points to the first node in the list.

1.5 Chapter 19

1.5.1 Introduction to the Stack ADT and Dynamic Stacks

Vocabulary

A. **Stack**: A data structure that holds a sequence of elements. In a stack, elements are stored in a last-in first-out manner, where the item that was inserted most recently is what is retrieved. This means that the first element stored in the stack is the last element that will be retrieved.

B. **Static Stack**: A stack that uses an array as its underlying data structure. Static stacks must have `isFull` and `isEmpty` functions. These functions must be run in order to add and remove elements from the stack, or it could lead to invalid operations being run.

C. **Dynamic Stack**: A stack that uses a linked list instead of an array. They have the advantage of not having to specify the size at declaration, as well as never being full while the computer has memory available.

Important Concepts

A. The two main stack operations are `push` and `pop`. `push` adds a new element to the stack, while `pop` removes it (and sometimes retrieves it, depending on how the stack is implemented).

1.5.2 The STL `stack` Container

Vocabulary

A. **`stack`**: The STL implementation of a stack. By default, it is implemented as a `deque`, but it can also be implemented based on a `vector` or `list`.

Important Concepts

A. The three ways of declaring an STL `stack` are:

   (a) `stack<T, vector<T>> stackName;`, which creates a vector-based stack

   (b) `stack<T, list<T>> stackName;`, which creates a linked-list-based stack

   (c) `stack<T> stackName`, which creates a deque-based stack (and is also the default).

B. In the STL `stack`, `pop` does not retrieve a value – it only removes it. To retrieve a value from the stack, you need to use the `top` function.

1.5.3 Introduction to the Queue ADT and Dynamic Queues

Vocabulary

A. **Queue**: A data structure that holds a sequence of elements. In a queue, elements are stored in a first-in, first-out manner. This means that the first element inserted into a queue will be the first to be retrieved.

Important Concepts

A. The main difference between stacks and queues are how elements are added and removed from the structure, as the advantages of a dynamic queue over a static queue are identical to those of dynamic stacks over static stacks.

B. The two main operations on a queue are `enqueue` and `dequeue`. The former adds an element to the "end" of the queue, while the latter removes the element at the "front" of the queue.

C. With static queues, an inefficiency is created when elements are removed, as you either have to shift every element forward one, or move the indices of the "front" and "end" of the queue. If the second approach, more efficient approach is taken, it is useful to think of the queue as storing items in a circle, where the start and end of the queue can move around freely if there is empty space.

D. Again, if the approach of moving indices around is taken with static queues, detecting whether the queue is full or empty can be somewhat difficult. Two common solutions are to always leave one element empty between the front and end, then determining that the queue is full when the rear index is within two positions of the front, and the other solution is to use a counter variable to track how many items are in the queue at a given time.

1.5.4 The STL `deque` and `queue` Containers

Vocabulary

A. **`deque`**: The STL implementation of a double-ended queue. Unlike a regular queue, a deque provides quick access to the element at the front *and* back of the container. It shares the operations of a queue, but they are named differently: `push_back` is `enqueue` and `pop_front` is `dequeue`.

B. **`queue`**: The STL implementation of a queue. It provides the operations `push` and `pop`, which are simply `enqueue` and `dequeue`, respectively.

1.6 Chapter 20

1.6.1 Introduction to Recursion

Vocabulary

A. **Recursive Function**: A function that calls itself within its body.

B. **Depth of Recursion**: The number of times a recursive function calls itself.

1.6.2 Solving Problems with Recursion

Vocabulary

A. **Base Case**: The case(s) where the problem can be solved without recursion. Importantly, there can be more than one base case.

B. **Recursive Case**: The case(s) where the problem must be solved recursively. In general, the recursive case needs to reduce the problem in some way (i.e., get it closer to a base case).

C. **Direct Recursion**: Recursion that occurs when a function directly calls itself.

D. **Indirect Recursion**: Recursion that occurs when a function calls another function that calls the original function.

Important Concepts

A. Recursion is never required to solve problems, but it can make them easier to solve. This can come at the cost of program performance, however, as it can lead to extra memory having to be allocated for parameters, local variables, and memory addresses every time the function is called again.

1.6.3 Exhaustive Algorithms

Vocabulary

A. **Exhaustive Algorithm**: An algorithm that finds the best combination by looking at every possible combination.

1.6.4 Recursion vs. Iteration

Important Concepts

A. Recursive algorithms are less efficient than iterative algorithms, so the most optimal solution is always iterative. However, there are some problems where it is much simpler to create a recursive algorithm than an iterative algorithm.

1.7 Chapter 21

1.7.1 Definition and Applications of Binary Trees

Vocabulary

A. **Binary Tree**: A nonlinear linked structure where every node has pointers to two other nodes, and every node but the root node has a single predecessor. Binary trees reduce the time it takes to sort through large amounts of data, which is why they are important. Trees that are specifically created to be searched through are *binary search trees*.

B. **Tree Pointer**: The pointer of the binary tree that points to the first element in the tree.

C. **Child Node**: Nodes within a tree that have predecessors. Any given node can have at most two children, but one or both can be empty. These nodes should be set to `nullptr`.

D. **Leaf Node**: A node that does not have any child nodes.

E. **Left Subtree**: The subtree of a node that contains elements that are less than the node itself.

F. **Right Subtree**: The subtree of a node that contains elements that are greater than or equal to the node itself.

1.7.2 Binary Search Tree Operations

Vocabulary

A. **Inserting a Node**: The function below takes a node, `nodePtr`, and recursively searches through each subtree of the node to find where to put the new node, `newNode`. Importantly, nodePtr in this function is a reference pointer, which means that any changes made to it will be made to the node that was passed to the function.

B. **In-Order Traversal**: This type of traversal visits the left subtree, then the root node, then the right subtree. As the name implies, this type of traversal goes through the elements in order.

C. **Pre-Order Traversal**: This type of traversal visits the root node, then the left subtree, then the right subtree.

D. **Post-Order Traversal**: This type of traversal visits the root left subtree, then the right subtree, then the root node.

E. **Level-Order Traversal**: This type of traversal visits each level of the subtree, starting at the root node, then moving down.

F. **Searching the Tree**: To search a binary tree, you compare the value that you are looking for against both the left and right subtree, and continues searching through subtrees until it either finds the value or finds that it isn't there.

G. **Deleting a Node**: When deleting a node, there are three situations:

1. The node has no children. In this case, you simply remove the reference to the node in its parent node and delete the node with `delete`.

2. The node has one child. In this case, you redirect the pointer from the node's parent to the node's child, then delete the node with `delete`.
3. The node has two children. In this case, you must swap the node for its successor, which is the minimum node in the right subtree. An alternative is to replace it with the maximum node in of the left subtree.

2 Textbook Questions

3 Code Examples

3.1 Chapter 13

3.1.1 Class Member Access Specification

```
class Rectangle {
  private:
  double someAttribute;

  protected:
  double length;
  double width;

  public:
  virtual double getArea() const;
  void setLength(const double &);
};
```

3.1.2 Inline Functions and Constructors

```
// Specifically in Rectangle.h
class Rectangle {
  // Inline function
  void setWidth(const double &w) {
```

```
5     width = w;
6   }
7
8   // Inline constructor
9   Rectangle() {
10    width = 0;
11    height = 0;
12  }
13 };
```

3.1.3 Specification vs. Implementation File

```
1  // Rectangle.h
2  // The specification file
3  #ifndef RECTANGLE_H
4  #define RECTANGLE_H
5
6  class Rectangle {
7    private:
8    double length;
9    double width;
10   public:
11   void setLength(const double &);
12   double getLength() const;
13 };
14
15 #endif CIRCLE_H
```

```
1  // Circle.cpp
2  // The implementation file
3
4  void Rectangle::setLength(const
       double &len) {
5    length = len;
6  }
7
8  double Rectangle::getLength() const {
9    return length;
10 }
```

3.1.4 Passing Arguments to Classes

```
1 // The object has a constructor with two
2 // arguments - width and length
3 Rectangle object(10.0, 10.0);
```

3.1.5 Passing Arguments to Base Class Constructors

```
Box::Box(const double &w, const double &l, const double &h) : Rectangle(w, l) {
  height = h;
}
```

3.1.6 Accessor vs. Mutator Function

```
// Accessor function - notice the const keyword at the end
double getLength() const;

// Mutator function
void setLength(const double &);
```

3.1.7 Pointer Objects and Dynamically Allocating Objects

```
// Note: Rectangle constructor is not actually
// called until the second statement
Rectangle *dynamicRectangle = nullptr;
dynamicRectangle = new Rectangle(10, 10);

// Regular pointer
Rectangle dummy(15, 15);
Rectangle *rectanglePointer = nullptr;
rectanglePointer = &dummy;

rectanglePointer->setLength(20);
dynamicRectangle->setWidth(13);

// Safely handle the dynamic pointer
delete dynamicRectangle;
dynamicRectangle = nullptr;
```

3.1.8 Pointer from Base to Derived

```
// In Box.h (Rectangle is parent/base of Box)
class Box : public Rectangle {}
```

```
3
4  // In main.cpp                    w, l, h
5  Rectangle *rectPointer = new Box(1, 3, 5);
6  rectPointer->getLength(); // Will be 3
7  rectPointer->getHeight(); // Error: getHeight is not defined for Rectangle
8
9  // Safely handle the dynamic pointer
10 delete rectPointer;
11 rectPointer = nullptr;
```

3.1.9 Member Initialization List

```
1  // In Rectangle.cpp
2  // A default constructor
3  Rectangle::Rectangle() : width(0.0), length(0.0) {}
4
5  // A constructor with arguments
6  Rectangle::Rectangle(const double &wid, const double &len)
7  : length(wid), width(len) {}
```

3.1.10 Default Constructors

```
1  // This is a default constructor
2  Rectangle::Rectangle() {}
3
4  // But this would also be a default constructor
5  Rectangle::Rectangle(const double &wid = 0, const double &len = 0)
```

3.1.11 Static Members and Functions

```
1  // In Rectangle.h
2  // Important note: static functions CANNOT be const
3  static void getNumberOfInstances();
4  static double NumberOfInstances();
5
6  // In Rectangle.cpp
```

```
double Rectangle::numberOfInstances = 0;
void Rectangle::getNumberOfInstances() {
  return numberOfInstances;
}

// main.cpp
// To call without using a class instances
Rectangle::getNumberOfInstances();

// To call using an instance
rectangleInstance.getNumberOfInstances();
```

3.1.12 Copy Constructor

```
// In Box.cpp
Box::Box(const Box &object) {
  height = object.height;

  // Follow the same "formula" for other attributes
}

// In main.cpp
Box A(1, 2, 3);
Box B = A;
```

3.1.13 Move Constructor

```
// In Box.cpp
Box::Box(Box &&temp) {
  // When using pointers you need to do
  // attribute = nullptr;
  // after stealing the data from temp.attribute
  height = temp.height;
  temp.height = 0;

  // Follow the same "formula" for other attributes
}

// In main.cpp
```

```
Box A = Box(3, 2, 1);
```

3.1.14 Copy Assignment

```
// In Box.cpp
Box &Box::operator=(const Box &right) {
  // Self-assignment check (Not strictly necessary)
  if (this != &right) {
    height = right.height;

    // Follow the same "formula" for other attributes
  }

  return *this;
}

// In main.cpp
Box A(2, 2, 2);
Box B(3, 3, 3);
A = B;
```

3.1.15 Move Assignment

```
// Move assignment
// In Box.cpp
Box &Box::operator=(Box &&right) {
  // Self-assignment check (Not strictly necessary)
  if (this != &right) {
    // swap is from #include <algorithm>
    // You need 'using namespace std;' to use it like this.
    swap(height, right.height);

    // Follow the same "formula" for other attributes
  }

  return *this;
}

// In main.cpp
```

```
Box A(1, 2, 3);
A = Box(2, 2, 1);
```

3.1.16 Self-Assignment Check

```
if (this != &right) {
  // operations
}
```

3.1.17 All (Differing) Overloads

```
// Assignment operator
const Box operator=(const Box &right) {
  if (this != &right) {
    height = right.height;

    // Follow the same "formula" for other attributes
  }

  return *this;
}
```

```
// Regular math operators
Box operator+(const Box &right) {
  Box temp;

  temp.height = height + right.height;

  // Follow the same "formula" for other attributes

  return temp;
}
```

```
// Prefix Operators
// No parameters
// *this is returned
Box Box::operator++() {
  ++height;

  return *this;
}
```

```
// Postfix Operators
// int Dummy Parameter
// Temp object is returned
Box Box::operator++(int) {
  Box temp(width, length, height);

  height++;

  return temp;
}
```

```
// Relational operators
// Like math operators but returns 'bool'
bool Box::operator>(const Box &right) {
  if (height > right.height) {
    return true;
  } else {
    return false;
  }
}
```

```
// Stream INSERTION Operator (<<)
// 'ostream' is used
// The &strm parameter CANNOT
// be a constant reference
ostream &operator<<(ostream &strm,
const Box &object) {
  strm << object.height;

  return strm;
}
```

```
// Stream EXTRACTION Operator (>>)
// 'istream' is used
// Both parameters are
// non-constant references
istream &operator>>(istream &strm,
Box &object) {
  strm >> object.height;

  return strm;
}
```

```
// How insertion and extraction should be declared in class specification file
#ifndef BOX_H
#define BOX_H
#include<iostream>

// Forward declaration
class Box;

```

```
// Overloads
ostream &operator<<(ostream &, const Box &);
istream &operator>>(istream &, Box &);

// Class declaration
class Box {};
```

```
// Subscript operator - []
// Only really good for array-like structures

Cat &Box::operator[](const int &subscript) {
  if (subscript < 0 || sub >= numberOfCats) {
    // Error handling stuff
    return 0;
  }

  return listOfCatsInBox[subscript];
}
```

3.1.18 Aggregation

```
class Box {
  // There will be an instance of a Cat inside the Box class
  // Box is the Aggregate Class
  Cat catInTheBox;
  // Other Stuff
}
```

3.1.19 Lvalue and Rvalue References

```
// Lvalue References
double variable = 3.2;
double &lvalue = variable;

// Rvalue References
double &&rvalue = 9.3;
```

3.1.20 Constructor Inheritance

```
// Constructor Inheritance
// Constructors in Rectangle.h
class Rectangle {
  Rectangle();
  Rectangle(const double &);
  Rectangle(const double &, const double &);
};

// In Box.h
class Box : public Rectangle {
  using Rectangle::Rectangle;

  Box(const double &);
};

// In main.cpp
// Calls Rectangle::Rectangle()
Box boxOne;

// Will call Box::Box(const double &) because
// any constructors in the derived class
// with the same parameters as the inherited
// constructors will take precedence.
Box boxTwo(5);

// Calls Rectangle::Rectangle(const double &, const double &)
Box boxThree(2, 5);
```

3.1.21 Virtual Function

```
// In Rectangle.h
virtual double getWidth() const;

// In Rectangle.cpp
double Rectangle::getWidth() const {
  return width;
}
```

3.1.22 Override and Final

```
// In Box.h
// Needs to be virtual to be final
virtual double getArea() const final;

// In Box.cpp
// Compiler will throw an error if a derived class of
// Box tried to override this function
double Box::getArea() const {
  return 2 * (length * width + height * width + length * height);
}

// In Rectangle.h
// Needs to be virtual to be override
virtual double getArea() const override;

// In Rectangle.cpp
double getArea() const {
  return length * width;
}
```

3.1.23 Abstract Base Class and Pure Virtual Function

```
// In Shape.h
class Shape {
  public:
  // Pure virtual function -> Makes Shape Abstract
  // Instances of Shape cannot exist
  virtual double getArea() const = 0;
}
```

3.1.24 Multiple Inheritance vs. Chain of Inheritance

```
// Multiple Inheritance
// Notice: It is inheriting from
// two classes at once
class Cat : public Pet, public Feline
    {}
```

```
// Chain of Inheritance
// Notice: Only inheriting
// one at a time
class Mammal : public Organism {}
class Feline : public Mammal {}
class Cat : public Feline {}
```

3.1.25 Static vs. Dynamic Binding

```
// Static Binding
// In Rectangle.h
// Area of Rectangle is
// length * width
double getArea() const;

// In Box.cpp
double getArea() const {
  return 2 * (length * width)
  + 2 * (length * height)
  + 2 * (height * width);
}

// In main.cpp
Rectangle *rectPtr = nullptr;
rectPtr = new Box(1, 1, 1);

// This will only display 1, since
// getArea is not virtual and will
// use Rectangle's definition
cout << rectPtr->getArea() << '\n';
```

```
// Dynamic Binding
// In Rectangle.h
// Area of Rectangle is
// length * width
virtual double getArea() const;

// In Box.cpp
double getArea() const {
  return 2 * (length * width)
  + 2 * (length * height)
  + 2 * (height * width);
}

// In main.cpp
Rectangle *rectPtr = nullptr;
rectPtr = new Box(1, 1, 1);

// This will display 6, since
// getArea is virtual and has
// been overridden by Box
cout << rectPtr->getArea() << '\n';
```

3.1.26 Friend Functions and Classes

```
// In Box.h
```

```
// Forward declaration of Cat
class Cat;

class Box {
  // Friend function of Box
  friend Cat getCatFromBox() const;

  // Friend class if Box
  friend Cat;
}
```

3.2 Singly Linked List (Template) Implementation

3.2.1 Class Header File

```
#ifndef SINGLY_LINKED_LIST
#define SINGLY_LINKED_LIST
template <class T>
class SinglyLinkedList {
  private:
  struct Node {
    T data;
    Node *next;
  };

  Node *head;

  public:
  SinglyLinkedList() { head = nullptr; }
  ~SinglyLinkedList();

  void append(const T &);
  void insert(const T &);
  void remove(const T &);
  void display() const;
};
#endif // SINGLY_LINKED_LIST
```

3.2.2 Destructor

```
template <class T>
SingleLinkedList<T>::~SingleLinkedList() {
  Node *current_node;
  Node *next_node;

  // Set current_node to the start of the list
  current_node = head;

  // Moves current_node through the list and deletes every element
  // individually
  while (current_node != nullptr) {
    next_node = current_node->next;

    delete current_node;

    current_node = next_node;
  }
}
```

3.2.3 Append

```
template <class T>
void SingleLinkedList<T>::append(const T &value) {
  Node *new_node;
  Node *current_node;

  // Initialize new_node with its data and populate the next pointer with
  // nullptr
  new_node = new Node;
  new_node->data = value;
  new_node->next = nullptr;

  // If the list is empty, make new_node the head
  // Else, traverse the list until the end and append the node to the end
  if (!head) {
    head = new_node;
  } else {
    // Set the traversal node to the first element (the head)
```

```
    current_node = head;

    // While the next node in the chain isn't null, set the traversal pointer to
    // be the next node
    while (current_node->next) {
      current_node = current_node->next;
    }

    // Adds new_node to the list
    current_node->next = new_node;
  }
}
```

3.2.4 Insert

```
template <class T>
void SingleLinkedList<T>::insert(const T &value) {
  Node *new_node;
  Node *current_node;
  Node *previous_node;

  new_node = new Node;
  new_node->data = value;

  // Make new_node the head if the list is empty
  if (!head) {
    head = new_node;
    new_node->next = nullptr;
  } else {
    current_node = head;
    previous_node = nullptr;

    // Move current_node to the correct place
    while (current_node != nullptr && current_node->data < value) {
      previous_node = current_node;
      current_node = current_node->next;
    }

    // If the new node is going to be the first in the list
    if (previous_node == nullptr) {
      head = new_node;
```

```
      new_node->next = current_node;
    } else {
      previous_node->next 7= new_node;
      new_node->next = current_node;
    }
  }
}
```

3.2.5 Remove (Delete Node)

```
template <class T>
void SingleLinkedList<T>::remove(const T &value) {
  Node *current_node;
  Node *previous_node;

  // Exit if the list is empty
  if (!head) {
    return;
  }

  // If the head is the element to be removed
  if (head->data == value) {
    current_node = head->next;
    delete head;
    head = current_node;
  } else {
    current_node = head;

    // Move current_node to the correct node
    while (current_node != nullptr && current_node->data != value) {
      previous_node = current_node;
      current_node = current_node->next;
    }

    // Change pointers around and delete current_node
    if (current_node) {
      // Removes current_node from the list by removing the reference to it
      // within the list
      previous_node->next = current_node->next;
      delete current_node;
    }
```

```
  }
}
```

3.2.6 Display

```
template <class T>
void SingleLinkedList<T>::display() const {
  Node *current_node;

  // Set current_node to the beginning of the list
  current_node = head;

  // Traverse the list
  while (current_node) {
    // Display the current element
    cout << current_node->data << '\n';

    // Set current_node to the next node in the list
    current_node = current_node->next;
  }
}
```

3.3 Recursive Functions

3.3.1 Factorial

```
int factorial(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

3.3.2 GCD

```
int gcd(int x, int y) {
  if (x % y == 0) {
    return y;
  } else {
    return gcd(y, x % y);
  }
}
```

3.3.3 Count the Nodes in a Linked List

```
int LinkedList::countNodes(Node *nodePtr) const {
  if (nodePtr != nullptr) {
    return 1 + countNodes(nodePtr-next);
  } else {
    return 0;
  }
}
```

3.3.4 Display the Nodes in a Linked List in Reverse Order

```
void LinkedList::displayReverse(Node *nodePtr) const {
  if (nodePtr != nullptr) {
    displayReverse(nodePtr->next);
    cout << nodePtr->value << " ";
  }
}
```

3.3.5 Binary Search

```
int binarySearch(int array[], int first, int last, int value) {
  if (first > last) {
    return -1;
  }

  int middle = (first + last) / 2;
```

```
7
8   if (array[middle] == value) {
9     return middle;
10  }
11  if (array[middle] < value) {
12    return binarySearch(array, middle + 1, last, value);
13  } else {
14    return binarySearch(array, first, middle - 1, value);
15  }
16 }
```

3.3.6 The Towers of Hanoi

```
1  // num: The number of discs to move
2  // fromPeg: The peg you are moving the discs froms
3  // toPeg: The peg you are moving the discs to
4  // tempPeg: The peg to use as a temporary peg
5  void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg) {
6    if (num > 0) {
7      moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
8      moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
9    }
10 }
```

3.3.7 Quick Sort

```
1  void quickSort(int set[], int start, int end) {
2    int pivotPoint;
3
4    if (start < end) {
5      pivotPoint = partition(set, start, end);
6      quickSort(set, start, pivotPoint - 1);
7      quickSort(set, pivotPoint + 1, end);
8    }
9  }
10
11 int partition(int set[], int start, int end) {
12   int pivotValue, pivotIndex, mid;
13
```

```
  mid = (start + end) / 2;
  swap(set[start], set[mid]);
  pivotIndex = start;
  pivotValue = set[start];

  for (int scan = start + 1; scan <= end; scan++) {
    if (set[scan] < pivotValue) {
      pivotIndex++;
      swap(set[pivotIndex], set[scan]);
    }
  }

  swap(set[start], set[pivotIndex]);
  return pivotIndex;
}
```

3.4 Binary Search Tree Functions

3.4.1 Insert

```
void BinaryTree::insert(Node* &nodePtr, Node* &newNode) {
  if (nodePtr == nullptr) {
    nodePtr = newNode;
  } else if (newNode->value < nodePtr->value) {
    insert(nodePtr->left, newNode);
  } else {
    insert(nodePtr->right, newNode);
  }
}
```

3.4.2 In-order Traversal

```
void BinaryTree::displayInOrder(Node *nodePtr) const {
  if (nodePtr) {
    displayInOrder(nodePtr->left);
    cout << nodePtr->value << '\n';
    displayInOrder(nodePtr->right);
```

```
  }
}
```

3.4.3 Pre-order Traversal

```
void BinaryTree::displayPreOrder(Node *nodePtr) const {
  if (nodePtr) {
    cout << nodePtr->value << '\n';
    displayPreOrder(nodePtr->left);
    displayPreOrder(nodePtr->right);
  }
}
```

3.4.4 Post-order Traversal

```
void BinaryTree::displayPostOrder(Node *nodePtr) const {
  if (nodePtr) {
    displayPostOrder(nodePtr->left);
    displayPostOrder(nodePtr->right);
    cout << nodePtr->value << '\n';
  }
}
```

3.4.5 Search

```
bool BinaryTree::searchNode(T value) {
  Node *nodePtr = root;

  while (nodePtr) {
    if (nodePtr->value == num) {
      return true;
    }
    else if (num < nodePtr->value) {
      nodePtr = nodePtr->left;
    } else {
      nodePtr = nodePtr->right;
```

```
12      }
13    }
14  }
```