Final Test Prep

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

1 Linked Lists 4

1.1 Advantages . . . 4

1.2 Disadvantages . . . 4

2 ADTs 4

2.1 Stack ADT . . . 5

2.1.1 Principle . . . 5

2.1.2 Data Members . . . 5

2.1.3 Member functions . . . 5

2.2 Queue ADT . . . 5

2.2.1 Principle . . . 6

2.2.2 Data Members . . . 6

2.2.3 Member functions . . . 6

2.3 Member Functions . . . 6

2.4 Common errors in the stack and queue . . . 7

2.4.1 Underflow . . . 7

2.4.2 Access Errors on Empty Structures . . . 7

2.5 Deque ADT . . . 7

2.6 Doubly-linked list deque . . . 9

3 Templates 12

3.1 Template Function . . . 12

3.2 Template Class . . . 13

3.3 Class vs typename keyword . . . 13

3.4 Handle friend functions . . . 13

3.4.1 Friendship to a Non-Template Function . . . 13

3.4.2 Friendship to a Template Function . . . 14

3.5 Function Template Specialization . . . 16
3.6 Class/Struct Template Specialization . . . 16
3.7 Template Parameters . . . 16
3.8 Trailing return type . . . 16
3.8.1 Syntax . . . 17
3.8.2 Example . . . 17
3.9 decltype . . . 17
3.9.1 Syntax . . . 17
3.9.2 Example . . . 17
3.10 Template functions with mixed types (Trailing return type) . . . 17
3.11 Template functions with mixed types (Deduced return type) . . . 18

4 Recursion 19
4.1 Recursion to count nodes in a linked list . . . 19

5 Inheritance 20
5.1 Order of constructors and destructors . . . 21
5.2 Constructors . . . 21
5.3 Destructors . . . 21
5.4 advantages and disadvantages of making data members protected versus making them private and accessing them using set and get methods. . . . 21
5.5 Overloading vs Overriding . . . 21
5.6 Upcasting . . . 22
5.7 Downcasting . . . 22
5.7.1 What Happens Without Virtual Functions . . . 22
5.7.2 Downcasting example . . . 23
5.7.3 Base class pointer example . . . 24
5.8 Object Slicing . . . 24
5.9 Multiple Inheritance . . . 26
5.9.1 Why Use Multiple Inheritance? . . . 26
5.9.2 Example . . . 26
5.9.3 Issues with Multiple Inheritance . . . 27
5.10 Virtual inheritance . . . 28
5.10.1 The Diamond Problem . . . 28

5.10.2 Solution with Virtual Inheritance 29

6 Polymorphism 30

Linked Lists

1.1 Advantages

- **Dynamic Size:** Unlike arrays, linked lists don't need to have a predetermined size, making them flexible in terms of storage requirements.
- **Efficient Insertions/Deletions:** Inserting and deleting nodes in a linked list is more efficient than in an array, especially for data structures that require frequent additions/removals. Inserting or deleting at the beginning or end of the list (or at any arbitrary position, given the node reference) can be done in constant time $O(1)$.
- **Memory Utilization:** Memory is allocated as needed for each node, avoiding wasted space.

1.2 Disadvantages

- **Access Time:** Accessing an element in a linked list takes linear time $O(n)$, as you need to traverse from the head node sequentially to the desired node. In contrast, arrays provide constant time $O(1)$ access.
- **Extra Memory Usage:** Each node in a linked list requires additional memory to store a pointer to the next node, or to both previous and next nodes in a doubly linked list.
- **Cache Performance:** Arrays have better cache locality due to contiguous memory allocation, which can result in better performance when iterating through the data sequentially.

ADTs

2.1 Stack ADT

2.1.1 Principle

The array based stack follows the LIFO (last in first out) principle. This means that items are inserted at the top of the stack and removed from the top of the stack.

2.1.2 Data Members

- **stk_array** - Stack array pointer. A pointer to the data type of the items stored in the stack; points to the first element of a dynamically-allocated array.
- **stk_capacity** - Stack capacity. The number of elements in the stack array.
- **stk_size** - Stack size. The number of items currently stored in the stack. The top item in the stack is always located at subscript stk_size - 1. Member Functions

2.1.3 Member functions

- **Default constructor:** Sets stack to initial empty state. The stack capacity and stack size should be set to 0. The stack array pointer should be set to nullptr.
- **size()** Returns the stack size.
- **capacity()** Returns the stack capacity.
- **empty()** Returns true if the stack size is 0; otherwise, false.
- **clear()** Sets the stack size back to 0. Does not deallocate any dynamic storage.
- **top()** Returns the top item of the stack (stk_array[stk_size - 1]).
- **push()** Inserts a new item at the top of the stack.
- **pop()** Removes the top item from the stack.
- **Copy constructor** Similar to the copy constructor for the example Vector class in the notes on dynamic storage allocation.
- **Copy assignment operator** Similar to the copy assignment operator for the example Vector class in the notes on dynamic storage allocation.
- **Destructor** Deletes the stack array.
- **reserve()** Reserves additional storage for the stack array.

2.2 Queue ADT

2.2.1 Principle

The queue follows the FIFO (first in first out) principle. This means items are inserted at the back of the queue and removed from the front.

2.2.2 Data Members

- **q_array** - Queue array pointer. A pointer to the data type of the items stored in the queue; points to the first element of a dynamically-allocated array.
- **q_capacity** - Queue capacity. The number of elements in the queue array.
- **q_size** - Queue size. The number of items currently stored in the queue.
- **q_front** - Queue front. The subscript of the front (or head) item in the queue.
- **q_back** - Queue back. The subscript of the back (or rear or tail) item in the queue.

2.2.3 Member functions

2.3 Member Functions

- **Default constructor** Sets queue to initial empty state. The queue capacity and queue size should be set to 0. The queue array pointer should be set to nullptr. q_front should be set to 0, while q_back is set to q_capacity - 1.
- **size()** Returns the queue size.
- **capacity()** Returns the queue capacity.
- **empty()** Returns true if the queue size is 0; otherwise, false.
- **clear()** Sets the queue size back to 0 and resets q_front and q_back to their initial values. Does not deallocate any dynamic storage or change the queue capacity.
- **front()** Returns the front item of the queue (q_array[q_front]).
- **back()** Returns the back item of the queue (q_array[q_back]).
- **push()** Inserts a new item at the back of the queue.
- **pop()** Removes the front item from the queue.
- **Copy constructor Similar** to the copy constructor for the example Vector class in the notes on dynamic storage allocation. A key difference is that we cannot assume that the items in the queue are stored in elements 0 to q_size - 1 the way we can in the Vector or an array-based stack. It is therefore necessary to copy the entire queue array.
- **Copy assignment operator** Similar to the copy assignment operator for the example Vector class in the notes on dynamic storage allocation. A key difference is that we cannot assume that the items in the queue are stored in elements 0 to q_size - 1 the way we can in the Vector or an array-based stack. It is therefore necessary to copy the entire queue array.
- **Destructor** Deletes the queue array.
- **reserve()** Reserves additional storage for the queue array. The process of copying the original array contents into the new, larger array is complicated by the fact that the exact locations of the queue items within the queue array are unknown and that there is no guarantee that q_front is less than q_back.

2.4 Common errors in the stack and queue

2.4.1 Underflow

- **Stack Underflow:** This happens when calling pop() or top() on an empty stack. The pop() function tries to remove the top element of the stack, and top() tries to access the top element without removing it. If the stack is empty, there's no top element to access or remove, leading to an underflow condition.
- **Queue Underflow:** Similar to stack underflow, this occurs when calling pop() or front() on an empty queue. The pop() function (or dequeue() in some implementations) attempts to remove the front element, and front() tries to access the front element. If the queue is empty, these operations cannot be completed, resulting in underflow.

Note:-

Stack underflow happens when we try to pop (remove) an item from the stack, when nothing is actually there to remove. This will raise an alarm of sorts in the computer because we told it to do something that cannot be done.

2.4.2 Access Errors on Empty Structures

While not always classified separately from underflow errors, trying to access the front or back of an empty queue, or the top of an empty stack, without removing elements, can also lead to errors. These are specific cases of underflow where the error is due to an attempt to read from an empty structure rather than to modify it.

- **Accessing the Top of an Empty Stack:** Invoking top() on an empty stack does not modify the stack but still results in an error because there's no element to return.
- **Accessing the Front or Back of an Empty Queue:** Similarly, calling front() or back() on an empty queue tries to access elements that do not exist, leading to errors.

2.5 Deque ADT

The changes that must be made to turn the queue from the section above into a deque, is to have

1. **push_front**: Add item at the front of the queue
2. **push_back**: Add item at the back of the queue
3. **pop_front**: Remove item from the front of the queue
4. **pop_back**: Remove item from the back of the queue

As opposed to the standard queue where we only have **push()** and **pop()** methods (pushing to the back and popping from the front).

```
void push_back(int value) {
    if (m_size == m_capacity) {
        reserve((m_capacity > 0 ? m_capacity * 2 : 1));
    }

    m_back = (m_back + 1) % m_capacity;
    arr[m_back] = value;
    ++m_size;
}

void push_front(int value) {
    if (m_size == m_capacity) {
        reserve((m_capacity > 0 ? m_capacity * 2 : 1));
    }

    m_front = (m_front - 1 + m_capacity) % m_capacity;
    arr[m_front] = value;
    ++m_size;
}

void pop_front() {
    if (!empty()) {
        m_front = (m_front + 1) % m_capacity;
        --m_size;
    }
}

void pop_back() {
    if (!empty()) {
        m_back = (m_back - 1 + m_capacity) % m_capacity;
        --m_size;
    }
}
```

2.6 Doubly-linked list deque

To modify the singly linked list queue into a doubly-ended queue (deque), you'll need to:

- **Modify the node Structure:** Add a prev pointer to allow traversal in both directions.
- **Implement push_front:** Add a function to insert elements at the front of the deque.
- **Implement push_back:** Add a function to insert elements at the back of the deque.
- **Implement pop_front:** Add a function to remove elements from the front of the deque.
- **Implement pop_back:** Add a function to remove elements from the back of the deque.

Note:-

In order to make a linked-list deque, the linked list **must** be doubly

```
void push_back(int value) {
    node* new_node = new node(nullptr, m_back, value);

    if (empty()) {
        m_front = new_node;
    } else {
        m_back->next = new_node;
    }

    m_back = new_node;
    ++m_size;
}

void push_front(int value) {
    node* new_node = new node(m_front, nullptr, value);

    if (empty()) {
        m_back = new_node;
    } else {
        m_front->prev = new_node;
    }

    m_front = new_node;
    ++m_size;
}

void pop_front() {
    if (m_front == nullptr) return;

    node* del = m_front;
    m_front = m_front->next;

    if (m_front == nullptr) {
        m_back = nullptr;
    } else {
        m_front->prev = nullptr;
    }

    delete del;
    --m_size;
}
```

```
void pop_back() {
    if (m_back == nullptr) return;

    node* del = m_back;
    m_back = m_back->prev;

    if (m_back == nullptr) {
        m_front = nullptr;
    } else {
        m_back->next = nullptr;
    }

    delete del;
    --m_size;
}
```

Templates

Templates in C++ are a powerful feature that allows writing generic and reusable code. They enable functions and classes to operate with different data types without being rewritten for each specific type.

3.1 Template Function

A template function defines a family of functions that work with different data types. Here's how to write and use a template function:

```
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << add<int>(3, 4) << std::endl; // Instantiates add<int>
    std::cout << add<double>(2.5, 3.1) << std::endl; // Instantiates add<double>
    return 0;
}
```

3.2 Template Class

```cpp
template <typename T>
class Stack {
    std::vector<T> data;

public:
    void push(T value) {
        data.push_back(value);
    }
    void pop() {
        data.pop_back();
    }
    T top() const {
        return data.back();
    }
    bool empty() const {
        return data.empty();
    }
};

int main() {
    Stack<int> intStack; // Instantiates Stack<int>
    intStack.push(10);
    intStack.push(20);
    std::cout << intStack.top() << std::endl;

    Stack<double> doubleStack; // Instantiates Stack<double>
    doubleStack.push(1.1);
    doubleStack.push(2.2);
    std::cout << doubleStack.top() << std::endl;
    return 0;
}
```

3.3 Class vs typename keyword

The choice between using **class** and **typename** in template declarations in C++ is largely a matter of style and historical context, as both keywords serve the same purpose

3.4 Handle friend functions

3.4.1 Friendship to a Non-Template Function

This is straightforward. You directly declare a non-template function as a friend inside your template class. This grants that specific function access to all instances of the template class, regardless of the type parameter.

```cpp
template <typename T>
class MyClass {
    friend void someFunction(MyClass<T>&);
};
```

3.4.2 Friendship to a Template Function

More commonly, you want a template function to be a friend to a template class. This allows each instantiation of the function template to access the corresponding instantiation of the class template.To achieve this, you need to forward declare the function template and then declare it as a friend inside your class template. The tricky part is that the syntax for declaring a template function as a friend inside a template class can vary based on what you're trying to achieve:

How to forward declare:

```cpp
    template<typename T>
    class myclass;

    template<typename T>
    void foo(myclass<T>&);


    template<typename T>
    class myclass {

    public:
        friend void foo <T>(const myclass<T>& obj);
    };

template <typename T>
void foo(myclass<T>& obj) {
    // Define
}
```

Different types:

- More general form used when you want the friendship to apply to all instantiations of the function template

```
template <typename T>
class MyClass {
    friend void someFunction<>(MyClass<T>&); // Specific instantiation
};
```

- All instantiations of the function template are friends:

```
template <typename T>
class MyClass {
    template <typename U>
    friend void someFunction(MyClass<U>&); // All instantiations
};
```

- This form ties the friendship to the specific template instantiation of both MyClass and someFunction using the same template argument T. The function template that takes the same template parameters:

```
template <typename T>
class MyClass {
    friend void someFunction<T>(MyClass<T>&); // Matched instantiation
};
```

3.5 Function Template Specialization

Concept 1: **Template specialization** allows you to define a different implementation for a particular data type.

```
template <typename T>
T min(T x, T y) {
    return (x < y) ? x : y;
}

template <>
const char* min(const char* x, const char* y) {
    return (strcmp(x, y) < 0) ? x : y;
}
```

3.6 Class/Struct Template Specialization

```
template<typename T>
struct foo {
    T x = 20;
};

template<>
struct foo<char> {
    char x = 'z';
};
```

3.7 Template Parameters

Concept 2: Templates can have more than one parameter, including non-type parameters.

```
template <typename T, int size>
class FixedArray {
    private:
    T arr[size];
    // implementation
};
```

3.8 Trailing return type

In traditional C++, the return type of a function is declared at the beginning of the function declaration. However, C++11 introduced a new syntax that allows the return type to be specified after the parameter list, using auto at the beginning and -> Type after the parameter list.

3.8.1 Syntax

```
auto functionName(parameters) -> returnType {
    // function body
}
```

3.8.2 Example

```
auto foo(int a, int b) -> int {
    return a + b;
}
```

3.9 decltype

Concept 3: **decltype** is a keyword in C++ introduced in C++11, which stands for "declared type". It is used to query the type of an expression without actually evaluating that expression. This can be particularly useful in template programming and type deduction, where the type of an expression might not be known until compile time.

3.9.1 Syntax

```
decltype(expression) variable_name;
```

Here, **variable_name** will have the same type as the type of **expression**. It's important to note that **expression** is not evaluated; **decltype** only deduces its type.

3.9.2 Example

```
int a = 5;
decltype(a) b = 5;

cout << typeid(b).name() << endl; // Output: i
```

3.10 Template functions with mixed types (Trailing return type)

Concept 4: To address the challenge of determining the return type for a template function that accepts two different types, we can utilize a strategy involving **auto** and a **trailing return type** with **decltype**. This approach effectively resolves the ambiguity of the return type in such template functions.

```cpp
template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u) {
    return t + u;
}
```

3.11 Template functions with mixed types (Deduced return type)

Alternatively, C++14 introduced the concept **deduced return type**. Which provides a simpler way to handle the situation described above

```cpp
template<typename T, typename U>
decltype(auto) foo(T a, U b)  {
    return a + b;
}
```

Recursion

- Recursion is a technique where the solution to a problem depends on solutions to smaller instances of the problem.
- A recursive function or method calls itself.
- A recursive call is always conditional – there must be some case (called the base case) where recursion does not take place. A recursive call should make progress towards the base case.
- Recursion is never required in C++. A recursive algorithm may always be rewritten with either a loop or a loop and a stack.
- In C++, a recursive algorithm is often less efficient in terms of memory usage and speed than the equivalent non-recursive algorithm. However, the recursive version may be shorter and easier to code.
- You should be able to write a simple recursive function or method to do something (like counting the nodes in a linked list).

4.1 Recursion to count nodes in a linked list

```
1  size_t countNodes(node* p) {
2      if (p == nullptr) {
3          return 0;
4      }
5      return 1 + countNodes(p->next);
6  }
```

Inheritance

Inheritance is a way to compartmentalize and reuse code by creating a new class based on a previously created class.

- The previously created class is called a **superclass** or **base class**.
- The new class derived from a base class is called a **subclass** or **derived class**. It represents a smaller, more specialized group of objects than its base class.
- A derived class may add new data members, add new methods, and override methods in the base class.
- Inheritance is used to represent an **"is a" relationship** between a derived class and a base class. A derived class object is also an instance of all of its base classes (e.g., a Circle is a Shape).
- Be able to code a derived class using **public inheritance**.
- Be able to code a constructor for a derived class, including one that passes arguments to the base class constructor using **constructor initialization list** syntax.
- Know the order in which constructor and destructor bodies will execute when you create or destroy an object of a derived class.
- Members of a class with **protected access** can be directly accessed by methods of the class, friends of the class, and methods of derived classes of the class.
- Be able to describe the advantages and disadvantages of making data members protected versus making them private and accessing them using set and get methods.
- Know the difference between **overloading** a method or function and **overriding** a method:
  - **Overloading** refers to a new method or function with the same name as an existing one in the same scope, but with a different signature (number of arguments, data types of arguments, order of data types, whether or not a method is const).
  - **Overriding** a method means writing a method in a derived class with the same name, arguments, and return data type as a method in the base class. The derived class method effectively replaces the base class method.
- Know how to call a base class version of a method from within a derived class method that overrides it.
- Know how to perform an **upcast** – a conversion of a derived class pointer or reference type to its base class pointer or reference type. In C++ this does not require an explicit type cast.
- A base class pointer or reference can only be used to call methods that are declared in the base class.
- Know how to perform a safe **downcast** – a conversion of a base class pointer or reference to one of its derived class pointer or reference types – using the `dynamic_cast` operator. A `dynamic_cast` requires that runtime type information be enabled. Know how to test whether or not a `dynamic_cast` was successful.
- C++ supports **multiple inheritance** – a derived class may have more than one base class.

5.1 Order of constructors and destructors

5.2 Constructors

- **Base Class Constructor First:** When a derived class object is created, the base class constructor is called before the derived class constructor.
- **Derived Class Constructor:** After the base class constructor finishes executing, the derived class constructor is executed.

5.3 Destructors

- **Derived Class Destructor First:** When a derived class object is destroyed, the derived class destructor is called first.
- **Base Class Destructor:** After the derived class destructor finishes, the base class destructor is called.

5.4 advantages and disadvantages of making data members protected versus making them private and accessing them using set and get methods.

The use of protected data members allows for a slight increase in performance, because we avoid incurring the overhead of a call to a "set" or "get" member function. Unfortunately, protected data members often yield two major problems. First, the derived class object does not have to use a "set" member function to change the value of the base class's protected data. A derived class object can easily assign an illegal value to a protected data member. Second, derived class member functions are more likely to depend on base class implementation details. Changes to the base class may require changes to some or all of the derived classes of that base class.

Declaring data members private, while providing non-private member functions to manipulate and perform validation checking on this data, enforces good software engineering. The programmer should be able to change the base class implementation freely, while still providing the same services to the derived class. The performance increases gained by using protected data members are often negligible compared to the optimizations that compilers can perform. It is appropriate to use the protected access modifier when a base class should provide a service (i.e., a member function) only to its derived classes and should not provide the service to other clients.

5.5 Overloading vs Overriding

As stated above, **Overloading** refers to a new method or function with the same name as an existing one in the same scope, but with a different signature (number of arguments, data types of arguments, order of data types, whether or not a method is const)

Overriding a method means writing a method in a derived class with the same name, arguments and return data type as a method in the base class. The derived class method effectively replaces the base class method.

5.6 Upcasting

Upcasting is the process of converting a derived class pointer or reference to a base class pointer or reference.

- Upcasting is safe because every derived object is also an instance of its base class.
- It allows for polymorphism when the base class has virtual functions.
- No explicit cast is required, and the conversion is implicit.

5.7 Downcasting

Downcasting is the process of converting a base class pointer or reference to a derived class pointer or reference

- Downcasting is potentially unsafe, so it requires an explicit cast.
- dynamic_cast should be used for safe downcasting to check if the cast is valid at runtime.
- Downcasting typically requires polymorphic classes with virtual functions to enable dynamic_cast.

In C++, the dynamic_cast operator, used for safe downcasting, requires that the base class has at least one virtual function. This is because dynamic_cast relies on runtime type information (RTTI), which is only available for polymorphic classes.

- **RTTI Availability:**
  - RTTI is used to store type information at runtime, which dynamic_cast uses to determine the exact type of the object being cast.
  - RTTI is only generated by the compiler for polymorphic classes, which are classes that have at least one virtual function.
- **Polymorphic Behavior:** Virtual functions enable polymorphic behavior, allowing derived classes to override base class functions. This is the essence of polymorphism, which dynamic_cast utilizes to ensure safe downcasting.
- **Checking Actual Type:** The type information stored in RTTI allows dynamic_cast to check if the object being cast is indeed of the target derived type. If not, it returns nullptr (for pointers) or throws a std::bad_cast exception (for references).

5.7.1 What Happens Without Virtual Functions

- **No RTTI:** If the base class has no virtual functions, it is not polymorphic, and the compiler doesn't generate RTTI for it. Without RTTI, dynamic_cast cannot validate types at runtime.
- **Compile-Time Error:** Attempting to use dynamic_cast on a class without virtual functions will lead to a compile-time error indicating that the class type is not polymorphic.

5.7.2 Downcasting example

```cpp
#include <iostream>

class Base {
    public:
    virtual ~Base() = default; // Make the class polymorphic
};

class Derived1 : public Base {
    public:
    void func1() {
        std::cout << "Derived1 Function\n";
    }
};

class Derived2 : public Base {
    public:
    void func2() {
        std::cout << "Derived2 Function\n";
    }
};

int main() {
    // Case 1: Valid downcast, will not get nullptr
    Base* basePtr1 = new Derived1(); // Base pointer to Derived1
    Derived1* derived1Ptr = dynamic_cast<Derived1*>(basePtr1);
    if (derived1Ptr) {
        derived1Ptr->func1(); // This will execute correctly
    } else {
        std::cout << "Downcast to Derived1 failed\n";
    }

    // Case 2: Invalid downcast, will get nullptr
    Base* basePtr2 = new Derived2(); // Base pointer to Derived2
    Derived1* derived1PtrInvalid =
    ↪ dynamic_cast<Derived1*>(basePtr2);
    if (derived1PtrInvalid) {
        derived1PtrInvalid->func1(); // This will not execute
    } else {
        std::cout << "Downcast to Derived1 failed\n"; // This
    ↪ will be printed
    }

    // Clean up
    delete basePtr1;
    delete basePtr2;

    return 0;
}
```

Note:-

When a Base* points to a Base object and we attempt to downcast it to a derived type using dynamic_cast, the cast will fail and return nullptr. This is because the actual type of the object being pointed to is Base, not the derived type.

5.7.3 Base class pointer example

```
class base {

public:
    virtual void print() const {
        cout << "Base class" << endl;
    }

};


class derived : public base {
    void print() const override {
        cout << "Child class" << endl;
    }
};


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

    base* bptr = new derived();

    bptr->print(); // Child class
    bptr->base::print(); // Base class

    return EXIT_SUCCESS;
}
```

Note:-

Notice we are able to call the private method, this is because the virtual function mechanism directs the call to the most derived method.

5.8 Object Slicing

Object slicing occurs when a derived class object is assigned or copied to a base class object, causing the derived class's specific members to be "sliced off."

```
base b1 = base();
derived d1 = derived();

b1 = d1; // Slicing
d1 = b1; // Does not work
```

5.9 Multiple Inheritance

Multiple inheritance is a feature in C++ that allows a derived class to inherit from more than one base class.

5.9.1 Why Use Multiple Inheritance?

- **Combining Functionality:** When a derived class needs to combine the functionalities of multiple base classes.
- **Mixins:** Allows implementing mixins, which are small base classes that provide specific functionalities to derived classes.
- **Interface Implementation:** Multiple inheritance can also be used to implement interfaces (abstract base classes) in C++.

5.9.2 Example

```cpp
#include <iostream>

class Base1 {
public:
    void func1() {
        std::cout << "Base1 Function" << std::endl;
    }
};

class Base2 {
public:
    void func2() {
        std::cout << "Base2 Function" << std::endl;
    }
};

class Derived : public Base1, public Base2 {
    // Derived class inherits from both Base1 and Base2
public:
    void func3() {
        std::cout << "Derived Function" << std::endl;
    }
};

int main() {
    Derived d;
    d.func1(); // Inherited from Base1
    d.func2(); // Inherited from Base2
    d.func3(); // Own function

    return 0;
}
```

5.9.3 Issues with Multiple Inheritance

1. **Ambiguity:**
    - If two base classes have the same method or attribute name, calling it from the derived class can cause ambiguity.
    - This can be resolved using the scope resolution operator :: to specify which base class method to call.
2. **Diamond Problem:**
    - If two base classes inherit from the same class and a derived class inherits from both base classes, it leads to ambiguity in the derived class about which base class's implementation to use.
    - This can be addressed using virtual inheritance to ensure only one copy of the shared base class exists.

5.10 Virtual inheritance

Virtual inheritance in C++ is a mechanism to prevent multiple ”copies” of a base class when using multiple inheritance. It ensures that the derived class has only one shared instance of the base class, thus preventing ambiguity and redundant base class objects.

5.10.1 The Diamond Problem

The diamond problem occurs when two classes inherit from the same base class and another class inherits from those two classes. This results in ambiguity and multiple copies of the shared base class.

```cpp
#include <iostream>

class Base {
    public:
    int data;
    Base() : data(0) {}
};

class Derived1 : public Base {};

class Derived2 : public Base {};

class FinalDerived : public Derived1, public Derived2 {};

int main() {
    FinalDerived obj;

    // Error: Ambiguity, which Base::data to access?
    // obj.data = 10;
    // Explicit resolution:
    obj.Derived1::data = 10;
    obj.Derived2::data = 20;

    std::cout << obj.Derived1::data << ", " <<
    ↪ obj.Derived2::data << std::endl;

    return 0;
}
```

There are two separate instances of the Base class, one inherited through Derived1 and one through Derived2 This leads to ambiguity and multiple instances.

5.10.2 Solution with Virtual Inheritance

Virtual inheritance addresses this problem by ensuring that the shared base class is inherited virtually. This makes the derived class have a single, shared instance of the base class.

```cpp
#include <iostream>
class Base {
public:
    int data;
    Base() : data(0) {}
};

class Derived1 : virtual public Base {};

class Derived2 : virtual public Base {};

class FinalDerived : public Derived1, public Derived2 {};

int main() {
    FinalDerived obj;

    // No ambiguity, only one instance of Base exists
    obj.data = 10;

    std::cout << obj.data << std::endl;

    return 0;
}
```

Polymorphism

- **Polymorphism:** The ability of objects belonging to different types to respond to method calls of the same name, each one according to an appropriate type-specific behavior. In C++, polymorphism is implemented through the use of virtual methods.
- You should know how to code a virtual method. A virtual method called using a pointer or reference will use dynamic binding. Other method or function calls use static binding.
  - With dynamic binding, which version of a method to call is determined at runtime based on the data type of the object a pointer or reference points to (the dynamic type), rather than the data type of the pointer or reference (the static type).
  - With static binding, the data type of the object name, pointer to an object or reference to an object is used to determine which version of a method or function to call at compile time.
- A pure virtual method (also called an abstract method) is a virtual method with no implementation, only a prototype (that ends with = 0).
- An abstract class in C++ is one that contains one or more pure virtual methods. A class that is not abstract is referred to as a concrete class.
  - You cannot create an object of an abstract class.
  - You can use an abstract class as a base class for inheritance.
  - You can declare a pointer to an object of an abstract class or a reference to an object of an abstract class. Such a pointer or reference will normally be used to point to an object of one of the abstract class's derived classes.
- A derived class must implement all of the pure virtual methods in an abstract base class or it is also an abstract class.
- An interface is an abstract class that contains only pure virtual methods and symbolic constants (static const data members).