Exam 2 prep

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States
April 2, 2024

Contents

1 Preface 3

1.1 Structure . . . 3

1.2 Topics . . . 3

2 Pointers 4

2.1 Types of pointers . . . 4

2.2 Indirection operator (*) . . . 4

2.3 "Address of" operator (&) . . . 4

2.4 Relationship between a pointer and the variable it points to . . . 5

3 The const keyword 6

3.1 Reference to const data and its restrictions . . . 6

3.1.1 Restrictions . . . 6

3.2 Restrictions on const member functions . . . 6

3.3 Changing the Data Members of the Object . . . 6

3.4 Calling a Non-Const Member Function . . . 7

4 Default function arguments 8

5 Function and Member Function Overloading 9

5.1 Criteria . . . 9

6 Aspects of an Operator That Cannot Be Changed by Operator Overloading 10

6.1 Must be overloaded as member functions . . . 10

6.2 Must be overloaded as standalone functions . . . 10

6.3 Function Call Generated by the Compiler . . . 11

6.3.1 Member Function Operator Overload . . . 11

6.3.2 Standalone Function Operator Overload 11

7 Dynamic Class members 12

8 Copy constructor 13

8.1 Situations which may result in a call to the copy constructor 13

8.2 Copy constructor and copy assignment operator 14

9 Abstract Data Type Definition 15

10 ADTs 16

10.1 Stack ADT 16

10.1.1 Principle 16

10.1.2 Data Members 16

10.1.3 Member functions 16

10.2 Queue ADT 16

10.2.1 Principle 17

10.2.2 Data Members 17

10.2.3 Member functions 17

10.3 Member Functions 17

10.4 Common errors in the stack and queue 18

10.4.1 Underflow 18

10.4.2 Access Errors on Empty Structures 18

Preface

1.1 Structure

Exam 2 consists of 30 questions. The breakdown by question type is:

- 4 True / False questions (1 point each, 4 points total)
- 10 Multiple Choice questions (2 points each, 20 points total)
- 6 Fill-in-the-Blank(s) questions (varying point values, 19 points total)
- 10 Essay questions (definitions, coding, etc.) (varying point values, 57 points total)

1.2 Topics

The topics that are covered

- **C++ Pointers and References**
- **C++ Variable Declarations**
- **The const Keyword**
- **Default Function Arguments**
- **Function and Member Function Overloading**
- **The this Pointer**
- **The friend Keyword**
- **Operator Overloading**
- **Dynamic Storage Allocation**
- **Destructor**
- **Copy Constructor**
- **Copy Assignment Operator**
- **Abstract Data Type Definition**
- **Stack and Queue ADTs**

Pointers

2.1 Types of pointers

- Pointer

```
type* name;
```

- Pointer to constant data

```
const int* ptrToConst; // A pointer to a constant integer.
↪ You cannot change the integer value via ptrToConst.
int const* ptrToConstAlt; // Alternative syntax, identical
↪ meaning.
```

- Constant pointer

```
int value = 5;
int* const constPtr = &value; // A constant pointer to an
↪ integer. You cannot change what constPtr points to after
↪ initialization.
```

- **Constant pointer to constant data**

```
const int* const constPtrToConst = &value; // A constant
↪ pointer to a constant integer.
```

2.2 Indirection operator (*)

This operator is used to access the value at the address a pointer is holding. It ”dereferences” the pointer.

```
int value = 5;
int* ptr = &value;
int dereferencedValue = *ptr; // dereferencedValue is now 5.
```

2.3 ”Address of” operator (&)

This operator is used to obtain the memory address of a variable.

```
int value = 5;
int* ptr = &value; // ptr now holds the address of value.
```

2.4 Relationship between a pointer and the variable it points to

A pointer holds the memory address of another variable. Through the pointer, you can read or modify the value of the variable it points to (unless it's a pointer to constant data). The relationship is such that the pointer acts as a direct link to another variable's storage location in memory.

The const keyword

3.1 Reference to const data and its restrictions

A reference to const data is declared by placing the const keyword before the reference type. For example:

```
1  const int& refToConst = someInt;
```

In this declaration, refToConst is a reference to an int that cannot be modified through refToConst. It's important to note that the constness applies to the access path through the reference, not necessarily to the object itself. The original variable someInt could be modified through other means, but not through refToConst.

3.1.1 Restrictions

The primary restriction of a reference to const data is that you cannot modify the object it refers to through that reference.

A non-const reference cannot bind to temporary objects, but a const reference can. This feature is often used to extend the lifetime of a temporary object to the lifetime of the reference itself.

When passing objects to functions, using a reference to const data ensures that the function can read the object without copying it, but cannot modify it. This is a form of "read-only" access.

3.2 Restrictions on const member functions

In C++, const member functions are those that do not modify the object on which they are called. Marking a member function with const at the end of its declaration promises not to alter any member variables of the object or to call any non-const member functions that might. This constness applies to the *this pointer, making it a pointer to a const object, thereby preventing modification of any member variables directly or indirectly.

3.3 Changing the Data Members of the Object

In a const member function, you cannot modify any of the object's data members (except those marked as **mutable**). Attempting to do so will result in a compile-time error. This restriction is in place because the primary purpose of a const member function is to provide a guarantee that calling it will not alter the state of the object.

3.4 Calling a Non-Const Member Function

A const member function cannot call non-const member functions on the same object. This is because non-const member functions are not guaranteed to preserve the object's state—they may modify the object. Since a const member function promises not to alter the object, calling a function that could potentially change the object would violate this promise.

```
class MyClass {
    public:
    int value;
    void modify() {
        value = 10;
    }
    void tryModify() const {
        modify(); // Error: 'this' argument to member function
↪   'modify' has type 'const MyClass', but function is not
↪   marked const
    }
};
```

Note:-

An object that is not const can call a const member function or a non-const member function. An object that is const (or a pointer to a const object or a reference to a const object) can **only** call member functions that are const.

Default function arguments

To specify a default value for a parameter, you include an equals sign (=) followed by the default value in the function declaration. Default parameters must be the trailing parameters in the function signature; once you give a parameter a default value, all subsequent parameters in that function must also have default values.

```
1  void displayMessage(const std::string& message, bool newline =
   ↪  true) {
2      std::cout << message;
3      if (newline) std::cout << std::endl;
4  }
```

Note:-

If a function is declared before it is defined (such as in a header file), the default parameters should be specified in the declaration, not the definition. This helps to avoid confusion and ensures that the function signature is consistent across different translation units.

Function and Member Function Overloading

5.1 Criteria

know the criteria used by the compiler to distinguish between two or more functions or member functions with the same name and in the same scope:

- The number of arguments
- The data types of the arguments
- The order of the data types
- Whether or not a member function is const

Note:-

Note that the return data type of the function is not one of the criteria used.

Aspects of an Operator That Cannot Be Changed by Operator Overloading

- **Precedence:** The precedence of an operator determines how expressions involving multiple operators are parsed. For example, * has higher precedence than +, so a + b * c is treated as a + (b * c). Operator precedence is fixed and cannot be altered through overloading.
- **Number of Arguments:** Operators inherently have a fixed arity. Unary operators (like !, , + (unary plus), and - (unary minus)) operate on one operand. Binary operators (like +, -, *, /) operate on two operands. Overloading an operator does not change the number of operands it works with.
- **Direction of Evaluation:** The order in which the operands of an operator are evaluated is determined by the language's evaluation strategy and cannot be changed by overloading. For most operators in C++, the evaluation order is unspecified.
- **Behavior with Built-in Types:** Overloading an operator for a custom class or struct does not affect how that operator works with built-in types. For example, overloading + for your class does not change how + works for integers or other built-in types.

6.1 Must be overloaded as member functions

- Assignment operator (=)
- Subscript operator ([])
- Function call operator (())
- Member access operators (->, ->*)
- Any compound assignment operators (like +=, -=, etc.) technically don't have to be member functions, but in practice, they are implemented as members to modify the object's state directly.

Note:-

When an operator function is implemented as a member, the leftmost operand must be a class object (or a reference to a class object) of the operator's class.

6.2 Must be overloaded as standalone functions

For operators other than the ones described above, overloaded operator functions can be member functions or standalone functions that are not part of a class.

If the left operand of an overloaded operator is an object of a C++ standard library class or is not an object at all, then the operator function must be implemented as a standalone function. A standalone operator function is usually made a friend of the class to improve performance by avoiding the overhead of calls to accessor and mutator methods.

6.3 Function Call Generated by the Compiler

When you overload an operator, the compiler translates uses of that operator into function calls. The way this translation occurs depends on whether the operator is overloaded as a member function or a standalone function.

6.3.1 Member Function Operator Overload

If the operator is overloaded as a member function, the object on the left-hand side of the operator is used to invoke the member function, and the right-hand side operand is passed as an argument to the function.

```
class MyClass {
    public:
    MyClass operator+(const MyClass& rhs) const; // Member
    ↪ function
};

MyClass a, b, c;
c = a + b; // Translates to c = a.operator+(b);
```

6.3.2 Standalone Function Operator Overload

If the operator is overloaded as a standalone function, both operands are passed as arguments to the function.

```
class MyClass {
    // friend declaration is often used for standalone overloads
    ↪ to allow access to private members
    friend MyClass operator+(const MyClass& lhs, const MyClass&
    ↪ rhs);
};

MyClass a, b, c;
c = a + b; // Translates to c = operator+(a, b);
```

Dynamic Class members

A “shallow” copy of an object copies only the object but not the dynamic storage that it owns. A “deep” copy of an object copies the object and the dynamic storage that it owns.

A class that allocates dynamic storage for one or more of its data members requires coding all three of following member functions:

- Destructor
- Copy constructor
- Copy assignment operator

Copy constructor

8.1 Situations which may result in a call to the copy constructor

1. When an object is declared and initialized with another object of the same class
2. When an object is passed by value
3. When an object is returned by value

8.2 Copy constructor and copy assignment operator

```
class c {
    size_t sz = 0;
    int* a = nullptr;
    public:
    c() = default;
    ~c() {
        delete[] a;
    }

    c(int arr[], size_t sz) {
        this->sz = sz;
        int* tmp = new int[this->sz];
        for (size_t i=0; i<sz; ++i) {
            tmp[i] = arr[i];
        }
        this->a = tmp;
    }

    c(const c& obj) {
        this->sz = obj.sz;
        int* tmp = new int[obj.sz];

        for (size_t i = 0; i<obj.sz; ++i) {
            tmp[i] = obj.a[i];
        }
        this->a = tmp;
    }

    c& operator=(const c& obj) {
        if (this != &obj) {
            delete[] this->a; // Free old memory

            this->sz = obj.sz;
            int* tmp = new int[obj.sz];

            for (size_t i = 0; i<obj.sz; ++i) {
                tmp[i] = obj.a[i];
            }
            this->a = tmp;
        }
        return *this;
    }
};
```

Abstract Data Type Definition

An abstract data type is a data type defined in terms of what data may be stored and the operations that may be performed on it. It does not specify how the data is represented in memory.

ADTs

10.1 Stack ADT

10.1.1 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.

10.1.2 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

10.1.3 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.

10.2 Queue ADT

10.2.1 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.

10.2.2 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.

10.2.3 Member functions

10.3 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.

10.4 Common errors in the stack and queue

10.4.1 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.

10.4.2 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.