Pre-Placement Grind

Hardik Rajpal

September 29, 2023

Contents

1 Misc. **2**
1.1 Misc. Confirmed Optimizations . . . 2

2 Week 1 **3**
2.1 Searching Algorithms . . . 3
2.2 Sorting Algorithms . . . 4

3 Week 2 **6**

4 Week 3 **7**
4.1 Complete Search . . . 7

5 Week 4 **9**
5.1 Greedy Algorithms . . . 9
5.1.1 Union Find . . . 9
5.1.2 Greedy Matching . . . 11
5.1.3 Misc Data Structures . . . 11

6 Week 5 **12**
6.1 Dynamic Programming . . . 12
6.1.1 Common Patterns . . . 12

7 OOP (in C++) **13**
7.1 OOPs . . . 13
7.2 Inheritance from TutorialsPoint . . . 13
7.2.1 Multiple Inheritance . . . 14
7.3 Overloading . . . 14
7.3.1 Overloading ++ (and –) . . . 15
7.4 Polymorphism . . . 15
7.4.1 Static Linkage . . . 15
7.4.2 Dynamic Linkage . . . 16
7.5 Data Abstraction and Encapsulation . . . 17
7.6 Abstract Classes a.k.a C++ Interfaces . . . 17
7.7 Notes from interview questions . . . 17
7.7.1 Copy Constructors . . . 18
7.8 From LearnCPP.com . . . 20
7.8.1 14.4 Const objects . . . 20
7.8.2 14.6 Access functions . . . 21
7.9 Friends . . . 21
7.9.1 Friend non-member Functions . . . 22

8 Aptitude Test Pointers **23**
8.1 Number Sequence . . . 23

Chapter 1

Misc.

1.1 Misc. Confirmed Optimizations

1. For maps,

```
    auto iter = m.find(k);
    if(iter!=m.end()){return iter->second;}
    //is much faster than:
    if(m.count(k)){return m[k];}
```

2. Replace sets that check for inclusion by bit operations with an integer if the number elements < 32 for ints and 64 for long longs.
3. Pass params by reference where possible.
4. Instead of a reference parameter, consider using a pointer to that variable stored in a class data member.
5. Replace data types by smaller data types where possible:
   - long long by int
   - int by char
6. Replace fixed-length vectors by array<type,fixed-length>.
7. Replace maps by vectors if they are indexed by integers within a fixed range. Ex: the alphabet as indices.
8. That die roll problem optimization. (Needs to be phrased more mathematically.)
9. Use of suffix arrays + descending order (or its prefix counterpart) to cut of paths in backtracking.

Chapter 2

Week 1

2.1 Searching Algorithms

Notes from GFG

These are algorithms to check for the existence of an element or to retrieve it from a data structure. The retrieval can also involve only returning the position (index) or a pointer to the element. There are two types:

1. Sequential search: check every element based on a pre-determined sequence (ex. linear, alternating, etc.), and return the matches.
2. Interval search: Designed for searching in **sorted** data structures. They involve **repeatedly** dividing the search space into intervals which can be excluded entirely after certain checks (ex. binary search).

Some search algorithms are discussed below:

1. **Linear Search:** Straighforward for-loop iterating over all elements in an array.
   Time: O(n)
   Space: O(1)
2. **Sentinel Linear Search:** Reduces the number of comparisons by eliminating the need to check if the index is within bounds. This is accomplished by appending the target element to the end of the array, and treating its index in the result as "not found."
   Time: O(n)
   Space: O(1)
3. **Binary Search:** It's used for sorted arrays. It involves comparing the element at the center of the interval (defined initially as the entire array), with the target element. One of the halves of the interval is picked based on this comparison. The interval shrinks until the target is found or an interval of size one is not equal to the element. It can be implemented recursively or iteratively, each involving a step similar to $m = l + \frac{(r-l)}{2}$ while $l \leq r$
   Time: .
   Space: O(log(n)) O(1)
4. **Meta Binary Search:** Seems unimportant but check it here
   Time: O(log(n))
   Space: O(1)
5. **K-ary Search:** The search space is divided into k intervals in each step and one of them is picked to proceed further by comparing the target element to the interval markers.
   Time: O(log(n)). The reduction is of a constant term: $log_k 2$
   Space: O(1)

6. **Jump Search:** The sorted array is examined in jumps of the optimal size $\sqrt{n}$,until the element being examined is greater than the target element. The interval is then shrunk to the previous interval. The shurnken interval can be examined linearly or with another jump search.
Time: O($2\sqrt{n} = O(\sqrt{n})$), or $O(n^{1/2} + n^{1/4} + n^{1/8}...) = O(\sqrt{n})$
Space: O(1)

7. **Interpolation Search:** It improves over binary search only if the data is uniformly distributed. It involves selecting the splitting point of the current search space by comparing the target value to the current lower and upper bounds of the space. Linear interpolation involves the following equations:
$slope = (arr[r] - arr[l])/(r - l)$
$m = l + slope \times (x - arr[l])$
Time: O(log(log(n))) on average, O(n) WCS.
Space: O(1)

8. **Exponential or Unbounded (Binary) Search:** We examine the search space from the lower end l, comparing $l + 2^k - 1$ with the target element x, where k is the number of comparisons so far, until $x < arr[l + 2^k - 1]$. Then, we examine the interval bounded by $l + 2^{k-1} - 1$ and $l + 2^k - 1$, using binary search.
Time: O(log(n)), where n is the length of the array or where the first occurrence of the target element exists in an unbounded array.
Space: O(1)

9. **Fibonacci Search:** The array must be sorted. We first find the Fibonacci number $f(m)$ that exceeds the length of the given array. We compare the target element to the element at $arr[f(m-2)]$. We pick an interval based on the outcome.
Time: O(log(n))
Space: O(1)

Misc

- The preferred formula for evaluating the middle point of the interval in binary search is $m = l + (r - l)/2$, and not $m = (l + r)/2$, as the latter can suffer due to overflow.
- Global variables can also be used to maintain a "best value yet" while searching through a space with binary search. For ex. find the first element $\geq$ x in an array.
- Problems where an array can be mapped to a boolean variable and is guarranteed to have either
  - F...FT...T or
  - T...TF...F

  and our aim is to find the boundary between true and false values can be translated to a binary search problem, with the target as the point where the variable changes: arr[i] != arr[i+1].
- Remember the `break` statement in iterative binary search if the middle point element is equal to the target.
- One can also binary search for a target range's starting point, instead of just a target. See this problem.
- In some cases, we might want to keep the current middle point `m` in the search space, here we resort to replacing either one of `r = m - 1` or `l = m + 1` by `= m` and change the loop invariant `l <= r` to `l < r`.

2.2 Sorting Algorithms

These algorithms rearrange a given array in ascending order. Various other orders can be achieved by modifying the comparison operator. A sorting algorithm is **stable** if it preserves the relative order of equal elements.

Merge Sort

The first part of the algorithm recursively handles halves of the given array. The second part merges the halves sorted by the first part. It takes O(nlog(n)) time in the **all cases**. O(n) space is necessary for the merging side of affairs. Implemented recursively. It's advantages include stability, parallelizability and lower time complexity. It's disadvantages include higher space complexity and not being in-place, and that it's not always optimal for small datasets.

Quick Sort

It involves recursively picking an element (**the pivot**) from the unsorted array, placing it so that all elements less than it are before and all those greater than it are after. Then calling this function on the sub-arrays after and before the chosen element. //TODO pseudo code

Quick Sort

TODO pseudo code

The Others

1. **Selection Sort:** The given array is viewed in two parts; sorted and unsorted. Every iteration involves **selecting** the minimal element and swapping it with the first element of the unsorted part. Hence, the boundary of the sorted part is expanded and that of the unsorted part has contracted. All of this happens inplace. It isn't stable.
   Time: O(n^2)
   Space: O(1)

2. **Bubble Sort:** This involves repeatedly traversing the array, swapping any two **adjacent** elements if they are in the incorrect (descending) order, until we encounter a run with no swaps. It is stable. With each iteration, the last elements of the array are sorted in ascending order.
   Time: O(n^2)
   Space: O(1)

3. **Insertion Sort:** It involves iterating over the array once, and in each iteration, if the current element is less than its left neighbour, we move it leftwards until its left neighbour is lower than it. It is in-place and stable.
   Time: O(n^2)
   Space: O(1)

Chapter 3

Week 2

TODO move notes over from the other notebook.

Chapter 4

Week 3

4.1 Complete Search

Subset Processing

We use the function below with 0. (n = size of given set.)

```
void search(int k) {
    if (k == n) {
        // process subset
        subsets.push_back(subset);
    }
    else{
        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}
```

Listing 4.1: Subset Generation

```
for (int b = 0; b < (1<<n); b++) {
    //b runs from 00..00 to 11...11
    vector<int> subset;
    for (int i = 0; i < n; i++) {
        if (b&(1<<i)){
            subset.push_back(i)
        };
    }
}
```

Permutation Generation

TODO: write up permutation ideas. TODO: selection of subsets satisfying a property.

Backtracking En General

If the dimensions of inputs are smaller than usual, backtracking is an option. As with other algorithms, you want to maximize this as much as possible. Optimizations are possible by:

1. Transforming the inputs so as to reduce the search space.
   Example: If you are searching for a subset whose sum is a given target, Searching the space of frequency map is better than searching subsets in the untransformed set, at least when duplicates are abundant.

2. Cutting off fruitless search paths as soon as possible. (Pruning the search tree.)

3. Specifying "min" requirements before taking a path, and equivalently, specifying "max" allowed values in a path to be explored further.
4. Optimizing the data structures used to record the current state and restrictions. Particularly,
    - Using vectors instead of maps where possible.
    - Using bitmap `int`s when only inclusion is to be checked.
5. Instead of using min/max to bring index values within range, which will likely incur repeated searches at the boundary, use an if block to disregard paths associated with values that exceed the bounds.
6. A modification of the needle may speed up the search. For example, the search for a word may be sped up by searching for its reversed word if the end letter is less frequent than the letter at the start.

The abstract code for backtracking looks like ths:

```
//declare global/class member variables.
void search(int p){
    //p signifies path/position being inspected
    //in the search space.
    if(searchTerminalConditions()){
        if(globalVarSolutionValid){
            //update collection of solutions.
        }
    }
    else{
        for(possible path of exploration){
            //(1)update global vars so as to take this path.
            search(p+1);
            //(2)undo the updates made to global variables.
            //(not necessary if (1) overrides/uses previous updates.)
        }
        //undo any leftover changes made to global variables.
    }
}
```

Pruning: A way of adding intelligence to the backtracking algorithm and reducing the time spent in fruitless paths. Additionally, we can leverage symmetries of the search space to check only a fraction of the entire possible solution set. Clearly, optimizations at the start of the search tree save a lot more time than those at the end.

Meet in the Middle

Another name for **Divide and Conquer**. It refers to splitting the search space up into two halves and combining the results of the two halves. It works if there is an efficient way to combine the results. Even 1 level of splitting (and extracting solutions from the halves using brute force) can have worthwhile optimizations: $O(2^n) \implies O(2^{n/2})$.

Chapter 5

Week 4

5.1 Greedy Algorithms

Reading Notes

Greedy Solutions focus on looking at the problem in smaller steps, and at each step we select the option that offers the most obvious and immediate benefit. It's sort of like assuming there's only one maximum point in the search space, and hence, we just move in the direction with the most inclination. Some popular greedy algorithms are:

- Dijkstra's shortest path.
- Kruskal's minimum spanning tree.
- Prim's minimum spanning tree.
- Huffman encoding.

With greedy algorithms, we often have to repeatedly pick the minimal element from a collection; hence using a `priority_queue` or a `multiset` is often helpful.

5.1.1 Union Find

The data structure can also show up in greedy algorithms. Given below is the most optimized implementation of `find` and `combine`.

```
vector<T> items;//given vector of items.
vector<int> root;//representative roots of trees array.
vector<int> rank;//for combine optimization.
int find(int u){
    if(root[u]==u){return u;}
    //instead of return find(root[u]), do:
    root[u] = find(root[u]);//path compression
    return root[u];
}
void combine(int u, int v){
    int ru, rv;
    ru = find(u);rv = find(v);
    if(ru!=rv){
        //u, v in different trees.
        if(rank[ru] < rank[rv]){
            root[ru] = rv;
            rank[rv] += rank[ru];
            //combined tree has least possible height.
        }
        else{
            root[rv] = ru;
```

```
            rank[ru] += rank[rv];
        }
    }
}
```

Variations of `root` array

The usual union-find implementation's root elements satisfy `root[r] == r`. However, we can also use negative numbers at `root[r]` (which can't be the index of any parent), and check for `root[r] < 0` when searching for the root. Such a setup allows for recording information in the domain of negative numbers at the root, say, the size of the tree, but negated. The combine function then simply sets the combined tree's root value to the confluence of values at `rv` and `ru`.

Kruskal's MST Algorithm

1. Have a min-heap of all edges.
2. Iterate through the heap, merging the trees of the vertices of each edge. For each non-trivial merge, update a counter. Additionally, add the edge to the list of edges for the MST or its weight to the weight of the MST.
3. Once the merge counter is at $|V|$ - 1, break.

Prim's MST Algorithm

1. Pick a starting vertex. Initialize an empty min-heap of edges. Maintain a count of visited vertices.
2. Mark current vertex as visited.
3. Add all edges going out of the current vertex to the heap.
4. Iterate through the heap until an edge to an unvisited point is found.
5. Set this point as the current point. Iterate until count of visited vertices = $|V|$.

Dijkstra's Shortest Path

1. Pick a starting vertex. Maintain an array of minimum distances to reach any vertex from a visited vertex. For visited vertices, this should be -1.
2. Update distances of array elements as min(old distance, distance from current point which is INT_MAX if they are not neighbours). While iterating, record the array element with minimum distance to it.
3. Set the recorded element as the current vertex and continue until the current vertex is the target vertex.

Modifications can be made to record the predecessors in the paths or calculate the weights of the paths.

```
int distance(vector<int> &pi, vector<int> &pj);
int dijkstras(vector<vector<int>>& ps, int s, int target) {
    int n = ps.size(), res = 0, i = s;
    vector<int> min_d(n, INT_MAX);
    while (i != target) {
        min_d[i] = -1;
        int min_j = i;
        for (int j = 0; j < n; ++j){
            if (min_d[j] != -1) {//visited vertices.
                min_d[j] = min(min_d[j],distance(ps[i],ps[j]));
                min_j = min_d[j] < min_d[min_j] ? j : min_j;
            }
        }
        res += min_d[min_j];
        i = min_j;
```

```
    }
    return res;
}
```

Listing 5.1: Shortest Path

```
int distance(vector<int> &pi, vector<int> &pj);
int dijkstras(vector<vector<int>>& ps, int s, int target) {
    int n = ps.size(), res = 0, i = s,connected = 0;
    vector<int> min_d(n, INT_MAX);
    while (connected < n) {
        min_d[i] = -1;
        connected++;
        int min_j = i;
        for (int j = 0; j < n; ++j){
            if (min_d[j] != -1) {//visited vertices.
                min_d[j] = min(min_d[j],distance(ps[i],ps[j]));
                min_j = min_d[j] < min_d[min_j] ? j : min_j;
            }
        }
        res += min_d[min_j];
        i = min_j;
    }
    return res;
}
```

Listing 5.2: Dijkstra's MST

Stack Based Questions

These usually involve finding the (lexicographically) minimal subsequence. We maintain a stack to track the sequence selected so far. To reverse a stack to get the subsequence, the most optimal method is:

```
while(s.size()){
    ans.push_back(s.top());
    s.pop();
}
reverse(ans.begin(),ans.end());
```

Heap+Queue

Honestly I've only seen one question with this paradigm. However, it's worth a shot if you realize you have to process numbers starting always with the largest/smallest element, and have to track elements being available/unavailable over time. I know that's a very vague and oddly specific situation, but I couldn't just walk by a problem and not make this note.
Additionally, in scheduling problems, consider trying to find a way to arrange the given tasks, which might result in a closed form solution.

5.1.2 Greedy Matching

Given two arrays to match elements such that the matching function can be put into a total order over the elements (ISTG I will word this better, later), we can sort two arrays and take the first matches offered by traversing one array, selecting the first matched element with the element being traversed.

5.1.3 Misc Data Structures

Multiple problems tagged "Greedy" are really just a matter of organizing the input data in a structure such as a (frequency) map or a heap. Or we're just sorting the input array. So, consider this when thinking of approaching a question greedily.

Chapter 6

Week 5

6.1 Dynamic Programming

A common optimization to look out for when writing the code for dynamic programming problems, try to ensure that

1. The code doesn't compute paths that aren't going to be useful.
2. The code doesn't recompute any path more than once.

As per this article, the approach to most dynamic programming problems can be broken down to:

1. Find recursive relation.
2. Recursive (top-down).
3. Add Memoization.
4. Iterative + memoization (bottom-up).
5. Further optimizations.
    - Discarding paths
    - Reducing space complexity.

6.1.1 Common Patterns

Min (Max) Path to Reach Target

Distinct Ways

Merging Intervals

DP on Strings

Decision Making

Chapter 7

OOP (in C++)

7.1 OOPs

- Access-specifiers:
  1. private: can only be accessed inside the class.
  2. protected: can be accessed inside the class and inside derived classes.
  3. public: can be accessed everywhere.

7.2 Inheritance from TutorialsPoint

```
1   class DerivedClass: access-specifier BaseClass{
2       //access-specifier is one on public/private/protected.
3   };
```

Listing 7.1: Syntax

- Allows us to define a class in terms of another class.
- Derived classes inherit properties of base classes.
- Inheritance implements "is-a" relationship. Ex: mammal is-a animal, dog is-a mammal =¿ dog is-a animal also holds.
- Derived classes inherit all properties of base classes except:
  1. Constructors, destructors and copy constructors.
  2. Overloaded operators.
  3. Friend functions.
- The base classes can be inherited through public, protected or private inheritance, which is specified the access specifier before its name in the declaration of the derived class. The results are:
  1. Public: access permissions of public and protected members of the base class are carried forward in the inherited class.
  2. Protected: access permissions of public and protected members of the base class are lowered to protected.
  3. Private: access permissions of public and protected members of the base class are lowered to private.

7.2.1 Multiple Inheritance

TODO: problems/issues with multiple inheritance.

```
class DerivedClass: access-specifier baseA, access-specifier baseB ... {
//access-specifier is one of public/protected/private.
};
```

Listing 7.2: Syntax

7.3 Overloading

TODO order of usage around operators study.

- A single identifier (function name/operator) corresponds to two different implementations, based on the argument list supplied to it.
- Overload resolution refers to the compiler's task of selecting the most appropriate implementation when it encounters a call to an overloaded function.
- Note: operators can be overloaded outside classes too:

```
        //As a member function:
        Box operator+(const Box&);
        //Not as a member function:
        Box operator+(const Box&, const Box&);

```

- In general, use const and & for operands to
  - Avoid accidentally modifying them in the operation.
  - Avoid time spent copying them around.
- Most operators can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

- Operators that can't be overloaded:
  1. ::
  2. .*
  3. .
  4. ?:
- Unary operators include: ++ (post,pre), – (post,pre), -, and !

7.3.1 Overloading ++ (and −)

```
class Digit{
//postfix: a++ : returns an rval.
Digit operator++(int){
    ... return digit;
}
//prefix: ++a : returns an lval. (original variable)
Digit& operator++(){
    ... return *this;
}
}
```

- Both definitions have something unique:
  1. & in the prefix definition, to ensure it returns an lvalue.
  2. (int) in the postfix definition is necessary to distinguish it from the prefix definition, as c++ doesn't support return-value-based overloading.
- Also note that though operator++(int) looks like ++a (prefix), it's actually for a++ (postfix).

7.4 Polymorphism

Polymorphism means a call to a member function (after resolution of overloads), can lead to different implementations being called, based on the type of object that invokes the function.

7.4.1 Static Linkage

```
class Shape {
   public:
      int area() {
         cout << "Parent class area :"...
      }
};
class Rectangle: public Shape {
   public:
      int area () {
         cout << "Rectangle class area :"...
      }
};
class Triangle: public Shape {
   public:
      int area () {
         cout << "Triangle class area :"...
      }
};
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);
   shape = &rec;
   shape->area();
   shape = &tri;
   shape->area();
   //both calls print "Parent class area:..."
   return 0;
}
```

Without any prefixes in the functions defined in the derived classes (that are identifiable with functions in the base class), the compiler assumes that any calls to these functions from an object of type base*/base always needs the implementation from the parent class. This is known as static linkage or static resolution (of the function call) or early binding, as the implementation for the area function is fixed at runtime to that of the

base class (for objects of or pointers to the base class).
Note that calls from `rec` or `tri` would have called their respective functions, not the base class' function.

7.4.2 Dynamic Linkage

Prefixing function identifiers that are shared across derived and base classes with `virtual`, allows for dynamic linkage, or dynamic resolution or late binding. With the said prefix, the compiler identifies the right implementation to call by the contents of the object or pointer being used to invoke the function. This is polymorphism.

Pure Virtual Functions

If a function is always intended to be overridden in the derived classes and there's no meaningful definition in the base class, we can just declare the function in the base class and set it to zero to avoid compilation errors about no definition being found for the function in the base class.

```
class Shape{
    //virtual int area();
    //Above line compiles if there are no to area() from any objects of shape/derived classes.
    //In the presence of such calls, even if area() is defined in derived classes and their
    //objects are used to call area() (from the right pointer, or a pointer of type Shape*)
    //compilation errors ensue.
    //However,
    virtual int area()=0;//goes through compilation successfully.
};
```

Note:

- We say a function demonstrates polymorphism if we can use a pointer of the base class to access functions of different derived classes and have different implementations being used based on the derived class to which the object belongs.
- Without the virtual keyword, even if functions with identical identifiers (name and arguments considered) are declared in derived and base classes, polymorphism is not observed. Function calls from pointers of the base class' type call its own implementation, not that of derived classes.
- Once a function is declared as virtual in a class, it demonstrates polymorphism across all derived descendant classes, even without the virtual keyword being present intermediate classes.
- The `final` keyword can be used to throw a compilation error if a function that we don't want any base classes to override is overridden:

```
class Rectangle:public Shape{
    public:
    int area()final{
        cout<<"Rect Area"<<endl;
        return 0;
    }
};
//Now if a class called square attempts to override area, a compilation error is thrown.
```

- Using the final keyword in a non-virtual function (that was not declared to be virtual in any of the ancestors) throws a compilation error.
- **Object Slicing:** When an object of a derived class is assigned to an object (not a pointer) of a base class, only members inherited from the base class are kept and the others are discarded. Thus, all functions that may have been overridden are reverted to their definitions in the base class.

```
void printarea(Shape s){
    s.area();
}
void printareaReference(Shape &s){
    s.area();//NO slicing.
}
```

```
Shape s; Rect r;
s = r;//slicing.
Shape &s = r;
printarea(r);//slicing. prints "Shape area..."
printareaReference(r);//No slicing. prints "Rectangle area..."
```

In JAVA, and other languages where each non-primitive variable is actually a reference, object slicing doesn't happen.

7.5 Data Abstraction and Encapsulation

The idea is to write classes with a well-defined boundary between:

1. Implementation: how the class works, the variables and functions it needs for its work.
2. Interface: the function calls and variables accessible to the users of the class.

Data abstraction allows:

- Implementation of a class to evolve without affecting code that uses it.
- Prohibiting users from possibly disturbing the state of the objects of the class, which may affect correctness of its functions. Ex. A user sets the `top` pointer inside a stack's implementation to the start of the array, without updating the length, which is non-zero. This results in a segmentation fault.

It is enforced using access specifiers. Data encapsulation is about bundling all the related data and functions that use it into one class, keeping as much implementation detail from the user as possible.

7.6 Abstract Classes a.k.a C++ Interfaces

An abstract class is a class with at least one **pure virtual function**. Such classes define an interface that all derived classes have to support (have an implementation of).

7.7 Notes from interview questions

- The constructors of parent base classes are called in the order that they are declared and before the constructor of the derived class.
- Note that destructors are called in the completely reverse order of constructors. (Think of it as a stack of objects of base classes, with the derived class at the top).

```
    class Derived: public Base1, public Base2...
    //base1 constructor.
    //base2 constructor.
    //derived constructor.
    ...
    //derived destructor.
    //base 2 destructor.
    //base 1 destructor.
```

- When two sibling classes are used to derive a new class, multiple (redundant) copies of the base class's members are made. This can be avoided by using the virtual keyword in the declaration of the sibling classes:

```
        class base{};
        class b1: virtual public base{};//
        class b2: virtual public base{};//without virtual in both of them, copies are
    made.
        class derived: public b1, public b2{};

```

  - Without `virtual`, each parent class maintains its copy of variables from the base class, and sizeof(derived) = (sizeof(b1)+size(b2)).
  - With `virtual`, sizeof(derived) = (sizeof(b1 without b data)+sizeof(b2 without b data) + sizeof(b)+16B)
  - The 16B are for book keeping. It can also be 8B on some systems.
  - Note that all parent classes are prefixed with `virtual` and share ancestors, have a single copy of the ancestor's variables in the derived class.
  - The book-keeping info grows with 8B for each new parent class. It doesn't grow with the size of the base class (or any class).

- In multilevel inheritance (A-¿B-¿C), any function calls from an object of type C are linearly searched for up the hierarchy, and the first implementation is taken.
- Pointers of a parent type can hold a child, but child pointers being assigned to parent objects raises compilation errors.
- Pointers of a parent type can only access members (variables and functions) declared and declared public in the parent.
- When a derived class defines a function with the same name as some function its base class, all functions (even with different signatures) of the base class with the same name become inaccessible to objects.
- However, using a pointer of the base type to point to the object of the derived class, both the overridden method and the unoverridden overload of a method with the same name can be accessed.
- Or, using a scope resolution operator:

```
        d.Base::fun(5);//goes through.
        Base &b = d;
        b.fun(5);//goes through.

```

- Accessing members of the root class, using an object of a class derived from two non-virtual sibling descendants of the root, leads to compilation errors. To avoid this, either declare the siblings as virtual descendants or use scope resolution operators (of the sibling classes, not the root!).

Initializer Lists

- Initializer lists of a derived class can't include members of the base class. They need to be initialized using the contructor of the base class.

Diamond Problem of Multiple Inheritance

TODO

7.7.1 Copy Constructors

```
class Sample{
    int id;
    Sample(Sample &t)
    {
        id=t.id;
    }
};
//defines what do to do when:
Sample a,b;
a = b;//calls copy constructor of a.
```

- Used to intialize members of a newly created object by copying members of an already existing object.
- It takes a reference parameter of an object of the same class.
- This is known as copy initialization, a.k.a. member-wise initialization.
- If not defined explicitly by the programmer, the compiler defines it for us.

Types of Copy Constructors

1. Default Copy Constructor: The implementation offered by the compiler which copies the bases and members of an object in the same order that a constructor would intialize the bases and members of the object.
2. User Defined Copy Constructor: needed when an object owns pointers or non-shareable references, such as to a file. A destructor and assignment operator should also ideally be written in this case to assist in transfer/destruction of said references.

A copy constructor is called when:

- An object of the class is **returned by value.**
- An object of the class is **passed by value** as an argument.
- An object is constructed based on another object of the same class. Ex: `Shape s1 = 1,2; Shape s2(s1)` or `Shape s2 = s1;`
- The compiler generates **a temporary object.**

Note that it's not called when a previously declared object is assigned another object. This calls the assignment operator.

```
Shape s1,s2;
s1 = {1,3};
Shape s3(s1);//calls copy constructor.
Shape s4 = s1;//calls copy constructor.
s2 = s1;//calls assignment operator.
```

Note that a call to the copy constructor is not guarranteed as the compiler performs optimizations like **return value optimization** and **copy elision** to avoid unnecessary copies where possible. (TODO)
Other points:

- Use a user-defined copy constructor when the default copy constructor results in a shallow copy (say, if some members are pointers). Deep-copy is only possible in a user-defined constructor.
- Copy constructors can be made private, and this makes objects of the class non-copyable. It's particularly useful (as a lazy technique to avoid shallow copies) if the class has pointers of dynamically allocated resources. The right way is to write a deep-copy-constructor and make it public.
- A copy constructor which takes the object argument by value leads to a compilation error, as at runtime it would have lead to an infinite chain of copy constructor calls.
- Use const in the argument to make sure:
  1. The source object isn't accidentally modified.
  2. The copy-constructor can be called with temporary objects created by the compiler, which can't be bound to non-const references.

```
        //if copy constructor doesn't say const Shape &s1,
        Shape s2 = fun();//fun returns s2 by value.
        //the above code throws a compilation error, at the last line.
        //If const is present, it compiles.

```

- In default constructors, default constructors of parents are called before those of derived classes, but, in copy constructors, the parent's default constructors (not copy constructors are called), unless the implementation of the derived class' copy constructor specifically calls their copy constructors.

Copy Elision (a.k.a. Copy Omission)

The compiler avoids making copies of objects (in pass by value/return by value scenarios) where possible.

7.8 From LearnCPP.com

7.8.1 14.4 Const objects

```
const Date today {2020, 10, 14};//valid.
const Date today = {2020, 10, 14};//valid.
//Note that const objects must be initialized, unless a default
//constructor is defined.
```

Objects that are declared with `const` keyword (as a local variable, or a function argument) impose certain restrictions (that upon violation lead to compiler errors.):

- Their members variables can't be changed, neither via direct access nor calls to member functions that change them.
- Additionally, const objects can't call non-const member functions. `const` before the definition body indicates that the member function doesn't modify the members of the class; it doesn't impose any restrictions on the returned value or aruments.

```
class Date{
    //can't be called by a const object, despite not
    //altering any variables in its definition:
    void print(){
        cout<<"non-const member function.";
    }
    //can be called by const objects:
    void print2() const {
        cout<<"const member function.";
    }
}
```

- If the declaration and definition are written separately, `const` must be present after the function signature in both places.

```
class Date{
    void print() const;
};
void Date::print()const{
    ...
}
```

- An attempt to modify the class inside a const function raises a compilation error, even inside unreachable if-blocks.
- Within the definition of a const member function, `this` is a const pointer to a const object.
- No constructor can be declared as a constant, as they need to modify the member variables, regardless of what their implementation says.
- It is perfectly fine to call const member functions from non-const objects.
- Functions can be overloaded based on whether they are const or not. So, const objects call the const variant while non-const objects call the non-const variant. This is usually done if constness changes the return value.

```
//These are valid overloads:
int fun(){

};
int fun()const{

};
//These are invalid as const keyword specifiers return type.
int fun(){}
const int fun(){}
//Also note that
const int fun(){};
//is exactly the same as:
int const fun(){};
//and the two are different from
int fun()const{};
```

7.8.2 14.6 Access functions

- Trivial member functions to access selectd private data members.
- Of two types: Setters (mutators) and getters (accessors).
- Getters are made const so they can be called on const objects while setters have to be non-const.
- For efficiency, getters can be written to return constant lvalue references, instead of returning by value:

```
const std::string &getName()const{return m_name;}
```

- Note that such functions' returned references become invalid the moment the object is destroyed. So, the references should not be stored (and accessed) beyond the lifetime of the object.

```
const std::string & ref = createEmployee().getName();
//we store the reference to a property in the rvalue implicit
//temporary object, created by the compiler.
//accessing ref later leads to undefined behaviour.
```

- References returned from functions to private members should be constant; otherwise, they permit direct modification of private members.

Ref-qualifier overloads

```
const std::string& getName() const & { return m_name; } // when called on an lvalue (single &)
    , return member by reference
const std::string getName() const && { return m_name; } // when called on an rvalue (double
    &&), return member by value
// Alternately, we can disable the use of a member functions for rvalue objects
const std::string& getName2() const && = delete;         // when called on an rvalue, emit a
    compilation error
```

TODO friend functions.

7.9 Friends

A friend is a class or function (member or non-member) that has been granted full access to the private and protected members of another class. Using friends, classes can selectively give full access to their members without unnecessary impacts.
The friendship is established by (declared in) the class removing its access control for some other entity.

7.9.1 Friend non-member Functions

With non-member functions, an object of the class must be accepted as an argument to access the relevant data.

```
class Shape{
    int area;
    friend double paintcost(const Shape& shape);
};
double paintcost(const Shape& shape){
    return shape.area*costunit;//accesses private member and compiles.
}
```

Note that the friend function can also be defined inside the class, and remain a non-member function because of the `friend` keyword.

Chapter 8

Aptitude Test Pointers

8.1 Number Sequence

These questions are hella annoying. So, I've written down a list of possibilities here that I can refer to while cheating in the test. (That's a joke.)

1. A.P, G.P, AGP. HP.
2. Check difference, 2nd difference ...
3. Incorporating a well-known series.
4. Consider $x^{f(x)}$ for monotonic series with large gaps.
   (a) Primes
   (b) Fibonacci (spot by nth difference - n-1th difference)
5. Two series of alternating numbers interleaved or alternating next() function.
   (a) The next function alternates between constant difference and constant factor.
   (b) The next function alternates between f(x)=ax+d and g(x)=bx+c
6. If the numbers go up and down, it's a result of interleaving or the relation between neighbours keeps switching between two functions.
7. Given alphabet series,
   (a) Convert to positions and reverse positions.
   (b) Consider vowel and consonant relations.
8. The sequence might involve manipulating a permutation of the previous element lmao.