1. Virtual memory
    - Virtual memory is an optimization techniques that is implemented by OS to “enlarge” the physical RAM. Virtual memory is used when the physical memory size is not enough to hold the real program, so the OS will use the VMM (virtual memory manager) to to create a swap (page) file on the disk. Whenever the system wants a block of memory that is not in RAM, the VMM will access the paging file on the disk and replace the content that is not currently needed with the needed contents in the paging file, this process is called paging (swapping). Virtual memory techniques facilitate the process of multitasking but will slow down the performance of the computer.
2. Steps of compilation of a C++ program
    - Preprocessing:
        - The c++ preprocessor copies the contents of the included header file to the source file, replace symbolic constants by using #define and its value
    - Compile:
        - The expanded source code is compiled into assembly language for the platform
    - Assembler:
        - The code generated by the compiler is assembled into the object files
    - Linker:
        - Link all object files by replacing any undefined references. Each of these references may be define in other libraries or other object files. Object code for library function will also be generated as well. An executable file will be generated.
3. Heap and stack
    - How do threads interact with the stack and the heap? How do the stack and heap work in multithreading?
        - In multi-threading application, each thread will have its own stack. But, all threads will share the same heap space.
    - Can stack grow in size or can heap grow in size?
        - Stack can’t grow in size. That’s why stack overflows happen. Heap can grow in size. Once there is not enough space in heap, the OS will assign a some memory to the heap.
    - Which one is faster?
        - Stack is much faster. This is because of the way that memory is allocated on the stack. Allocating memory on the stack is as simple as moving the stack pointer up. And the contents in stack tends to be reused frequently, thus they might be mapped into system cache, which make it faster.
    - Problems related to heap and stack
        - Stack overflows
        - Fragmentation. Excessive fragmentation, although there is enough memory, but not having a big block of memory.
4. C does not support function overloading like C++
    - printf(“%d”,data); printf(“%f”,data1); it seems like a function overloading because the function has the same name but type of parameter is different. But actually C uses a variable argument lists to achieve that.
    - For C++, the way it distinguish different calls to a function which has the same names is that when compiling the program, the compiler will try to change the name of the function according to the type of the parameters. For example:
        - void F(int, int) -> void F_int_int()
        - void F(double,double) -> void F_double_double()
5. Complex declaration of C
    - int (*p) [5] //a pointer points to an array of size 5
    - char *x[3][4] //an array of three arrays of size 4 pointers to char.
    - int* (*a[10])() //array of 10 function pointers to a function which has no parameters and the return type is a pointer to integer.
    - int (*t[])() // an array of function pointers to a function which has not parameters and the return type is a integer
6. A pure virtual function can have an implementation as long as the implementation is not ‘inline’, otherwise, there would be compilation error. Declare as:
    - virtual void fun() = 0;

7. Differences between rvalues and lvalues
    - An lvalue is an expression that refers to such an object. The term lvalue originally referred to objects that appear on the left hand side of an expression, which means the lvalue is modifiable.
    - An rvalue is an expression that is constant, unmodifiable. It includes constant value, and literal constant such as numbers.
8. Differences between an inline function and a marco
    - The major differences is the way they are handled. Inline function is dealt by the compiler while marco is dealt by the C++ preprocessor.
    - Inline function is very similar to a normal function (Differences is that whenever the compiler finds a call to an inline function, it writes a copy of the compiled function definition) while marco is implemented by simply replacing text.
9. Dead diamond in C++
    - A class inherit from two or more classes which inherit from a common base class. This will cause the ambiguity of calling functions that are inherited. We should add virtual keyword. and remember the destructor should be declared as virtual as well.
10. Conversion operator
    - A class has a public method for specific data conversion

```
class Boo
{
double value;
public:
Boo(int i )
operator double()
{
return value;
}
};
Boo BooObject;
double i = BooObject; // assigning object to variable i of type double.
now conversion operator gets called to assign the value.
```

11. Join types in databases:
    - Natural Join:
        - Implicit equal join based on the same name of two tables. (No need to specify the join conditions)
    - Inner Join:
        - Default type of join in a joined table
        - Only show matched for both tables
    - Left outer join:
        - every tuple to the left table must appear in the result
        - If no matching results, padded with NULL
    - Right outer join:
        - every tuple to the right table must appear in the result
        - Same..
    - Full outer join:
        - Every tuple for both tables must appear in the result.
12. Differences between where and having
    - where is used to filter information based on some conditions on the existing attributes. while having is used after grouping for filtering again. E.g say you have two tables employer table and department table, you want to find out the departments where the average salary is larger than say 5000. Firstly, you want to join those two tables, so you use where to join tables, and after join, you use group by keyword to group the results based on the department to select all departments and their respective average salary. Now, the only thing we have to do is to select those departments where avg salary is larger than 5000, then because it is the result of grouping, those attributes or information are not actually existing, thus, we use having to filter gain to get the

information we want.

13. Calculate power in log (n)

```
int pow(int num,int n)
{
        int result = 1;
        while (n)
        {
                if (n & 1)
                        result *= num;
                num *= num;
                n = n >> 1;
        }
        return result;
}
```

    - ‘&’ symbol is a bitwise operator used to determine if a number is an odd or even. 1 & 1 = 1; 100 & 001 = 0;
    - ‘>>’ is a bit operator, shift n to the right for one position.

14. Get Largest Continuous Sum

```
int getLargestContinuousSum(int* array,int size)
{
        int sum = array[0];
        int max = array[0];
        for (int i = 1;i < size;i++)
        {
                sum += array[i];
                sum = Max(sum,array[i]);
                max = Max(sum,max);
        }
        return max;
}
```

15. Check if a bit is set

```
bool isBitSet(int num,int pos)
{
        return num & (1 << (pos - 1));
}
```

16. Count bit sets for a 32-bit number

```
int countBitSet(int num)
{
        int count = 0;
        for (int i = 0; i < 32; i++)
        {
                if (1 & num)
                        count++;
                num = num >> 1;
        }
        return count;
}
```

17. How does virtual table work?
    - Virtual table is used for achieving polymorphism in Object oriented design. When we have to override the functions that we inherit from the base class we have to declare the functions in the base class to be a virtual function so that the compiler will create a vtable for each this kind of

class. The content of the vtable is the functions pointers to the virtual functions. Meanwhile, if the class has virtual functions, the compiler will add “hidden” code to the constructor of this class to initialize the vpointer to the vtable. Each class that has virtual functions will have only one virtual table. All its objects of this class will share the common vtable. (e.g. Animal -> Tiger. Animal has a virtual function which is int getWeight();) In Tiger class we override this function. In main: we declare a pointer: Animal* a = new Tiger; a->getWeight(); The underlying mechanism is that pointer a will point to the vpointer, and vpointer will find the corresponding function we want (a->vpointer->getWeight())

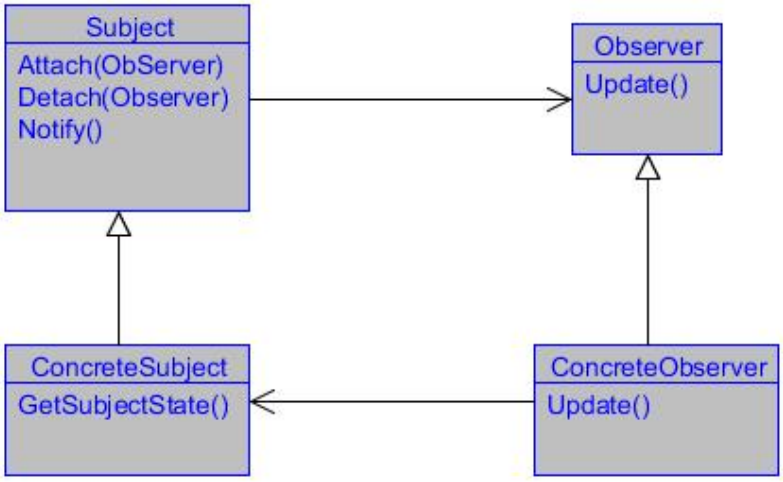
18. If a class has one or more virtual functions, then that class should have a virtual destructor.
19. In c++, the friend keyword can apply to a class or a function. It can access either private members or protected members. We should use friend keyword accordingly. For example, if we want to use I/O stream to input something to an object or output something to an object. We have to create the operator overriding functions outside the class scope. If we didn’t declare the operator outside the class, we might use them as : A << cout, which does not make sense. Then, if we didn’t declare it as a friend inside the class, we will not have access to the private members, which does not make sense as well. Thus, we implement the operator function outside the class while declaring it as a friend inside the class.
20. Call C functions from C++, using “extern “C”” keyword:
    - extern “C” void foo();
    - extern “C” { void foo(); void func(); }; //for multiple declarations.
21. Static member functions can’t access instance variables which is non-static variables or non-static member functions. Static member function or variables are shared by all objects. They don’t belong to specific objects. They are class variables and functions. Because we can call static member function and static variables without creating an instance. Thus, it can’t access the instance variables. Also, static functions and variables don’t have “this” pointer.
22. How does compiler deal with function templates in C++?
    - The compiler does not create separate definitions for all possible types for the function template. Instead, the compiler will only create definitions for the types that are actually used during the compile time. All declared parameters in template must be used.
23. Differences between overriding and overloading
    - Overloading means you have two functions in the same scope while having the same names but having different signatures
    - Overriding means if you have a virtual function in a base class, you can write a function with the same signature in the derived class.
24. Name hiding in C++:
    - Here are two classes, A and B, B inherit from A. A has a member function named Afunc, B has a member function which has the same name as Afunc. In this case, A’s copy of Afunc in B class has disappeared, which means B’s Afunc overrides the inherited Afunc. Thus, we can’t call the A’s Afunc inside B’s objects.
    - Way to resolve it: add ‘using A::Afunc; to make A’s Afunc visible in B’s scope.
    - Reason to have it:
        - If we don’t have it, we might violate the consistency of the whole class hierarchy for the reason that all class above B class will have same behaviour while B will have its own behaviour. Thus, we have to keep it consistent. Apply this rule to all classes.
25. Differences between process and thread:
    - A process is an executing instance of an application. A process can contain multiple threads. A thread can do anything a process can do.
    - Threads used for small tasks, whereas processes are used for more bigger and tougher tasks. Threads within the same process share the same address space, whereas different processes do not.
    - Sections of code that modify data structures shared by multiple threads are called critical sections. Only one thread into the critical section at a time (synchronization).
26. mutable keyword in C++
    - means a variable can be mutated even if the method is a const method that promises it won’t change anything.
27. Different keys concepts in databases
    - A table can have multiple unique, foreign keys but it can only have one primary key.
    - A unique key can have NULL values, but the values in a primary key can never be null.

- A foreign key can reference a non-primary key as long as this key is unique, which means this key is a candidate key.

28. Index in Databases
    - An index is a data structure (most commonly a B- tree) that stores the values for a specific column in a table.
    - Reason for using B-: it is time-efficient. Look-ups, deletions, and insertions can all be done in logarithmic time. And the data in B- tree can be sorted.
    - Hash table is another data structure that could be used for indexes: extremely efficient for looking-up. The value of a specific row is a key for the hash table, the value that is mapped by this keys the pointer to the row data in this table.
    - Disadvantages for a hash table: Hash table is not sorted.
    - Selectivity of SQL: cardinality / (number of records) * 100%. Low selectivity means that there is not many variations in this column (e.g gender has only two values male and female). In this case, it is not appropriate to use index because do a full scan of the whole table is more efficient.
    - Disadvantages: take up space, lower the performance of delete and insert.
    - Need to manually adjust the balance of the B- tree.
29. Differences between interface and abstract class:
    - An interface is a way of defining contract about what this object can do, it doesn't say what is object is.
    - An abstract class is way of defining what is object is. For example, a man can walk. For interface, it doesn't care about it is a man or a dog, it cares about it can actually walk. For abstract class, we know this class is a man, it is not a dog.
    - When to use which? Use abstract classes and inheritance if we want to make statement "A is B". Use interface to make statement "A is capable of walking.
30. Priority Queue (C++)
    - **template<typename _Tp,**

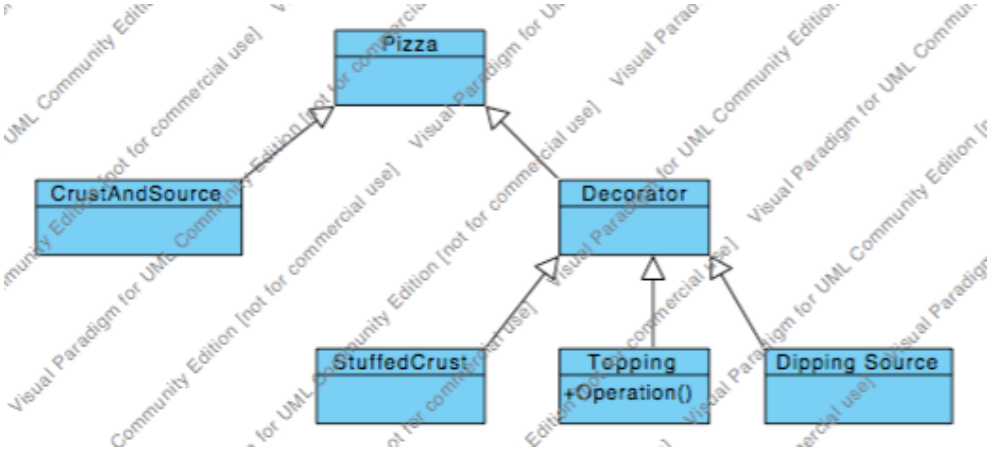
      typename _Sequence = vector<_Tp>,

      typename _Compare = less<typename _Sequence::value_type> >
    - First parameter is the typename
    - Second para is the vector containing the data
    - Third is the compare operations. Default is the Max heap which means this para is default less<int>
31. Private inheritance
    1. When a class B is not class A but class B has to use some methods that are in class A or override a virtual method. Use private inheritance. All public, private, protected members of superclass all become private members in subclasses. Subclass can only access public and protected members but private they can't
32. Design Patterns:
    - Singleton pattern:
      - Keep constructor as private;
      - Declare a static pointer instance as private;
      - Declare a static accessor for instance as public;
      - Declare a static cleaner as private;
      - Call cleaner at exit.
    - Observer Pattern:



- Subject is the thing that is going to be observed by observer. It has a vector to store all the observers. Attach and detach method is used to register those observers so that once their status changes, they can inform observers to check their status and behaviour accordingly.
- Observer is observing Subject. It has only one method which is update (notify), which is expected to be called by subjects in order to inform them the status changes.
- ConcreteSubject is the real concrete class.
- ConcreteObserver is the real concrete class of observer class. It has a pointer to the concreteSubject. In its constructor, this pointer has to be initialized.
- Once the status of concrete subject changes, it will apply its notify function to call the functions in the observer which is update. Then each observer will call the notify function in the concrete subject class to check its status and behave accordingly.

◦ Decorator Pattern:



- Attach additional responsibilities to an object dynamically (good for extending functionalities). Add behaviours to objects at runtime.
- Pizza is used as an abstract class, CrustAndSource is a concrete class. Decorator is some sort of Pizza classes. It has a protected pointer to the abstract class. Its descturctor should be responsible for deleting the objects.

◦ Factory method pattern:

- You have set of components classes but won't know exactly which one you will need to instantiate at the runtime. (create different objects without knowing beforehand what sort of objects it needs to create)
- It's called factory because it creates various type of objects without necessarily knowing what kind of object it creates or how to create it.

- Design an interface for creating an object, but let subclasses decide which class to instantiate
- E.g Video game with 2 kinds of enemies: turtle and bullet. System randomly sends turtles and bullets but bullets become more frequent when you get closer to the end.

```
class Level
{
    public:
    Enemy* createEnemy() = 0;
};

Then specialize in subclasses:
class NormalLevel: public Level
{
    public:
        Enemy* createEnemy()
        {
            create mostly turtles;
        }
};

class Castle : public Level
{
    public:
        Enemy* create Enemy() {}
};

Level* l = new NormalLevel;
Enemy* e = l->createEnemy();
```

- Template Method Pattern
  - Many objects share some common grounds but they have their own characteristics. For example, everybody will have same skeleton, but everyone will have their own characteristics such as eye colour, hair colour…Each block will share the same interface for shift left, right, drop down…but the way they rotate on the board is different.

```
class Turtle
{
public:
    void draw() { drawHead(); drawShell(), drawFeet();}
private:
    virtual drawShell() = 0;// Only this method is different, because the colour of the shell is
different.
    void drawHead();
    void drawFeet();

};

class RedTurtle: public Turtle
{
    void drawShell(); // draw a red shell
};

class GreenTurtle: public Turtle
{
    void drawShell(); // draw a green shell
};
```

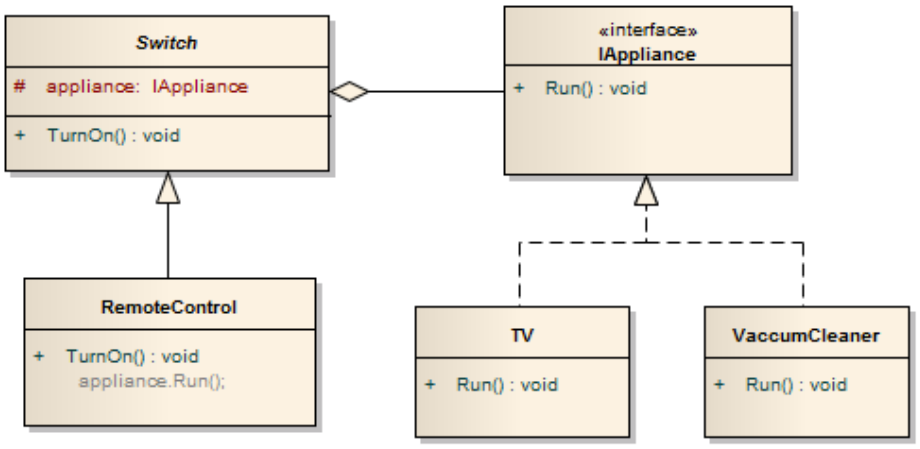
- Visitor Pattern
  - For implementing double dispatch - make decisions depends on the runtime types of two

objects.

- A virtual method is a single dispatch, solely make decision based on the runtime type of an object.

```
class Enemy
{
    public:
        virtual void strike(weapon& w) = 0;
}

class Turtle: public Enemy
{
    virtual void strike( weapon& w)
    {
        w.strike(*this);
     }
}

class Bullet: public Enemy
{
    virtual void strike (Weapon &w)
    {
        w.strike(*this);
    }
}

class weapon
{
    virtual void strike(Turtle& t) = 0;
    virtual void strike(Bullet& b) = 0;
};

class stick:public weapon
{
    void strike(Turtle& t)
    {
        //strike turtle with stick
    }
    void strike(Bullet& b)
    {
        //strike bullet with stick
    }
}
```

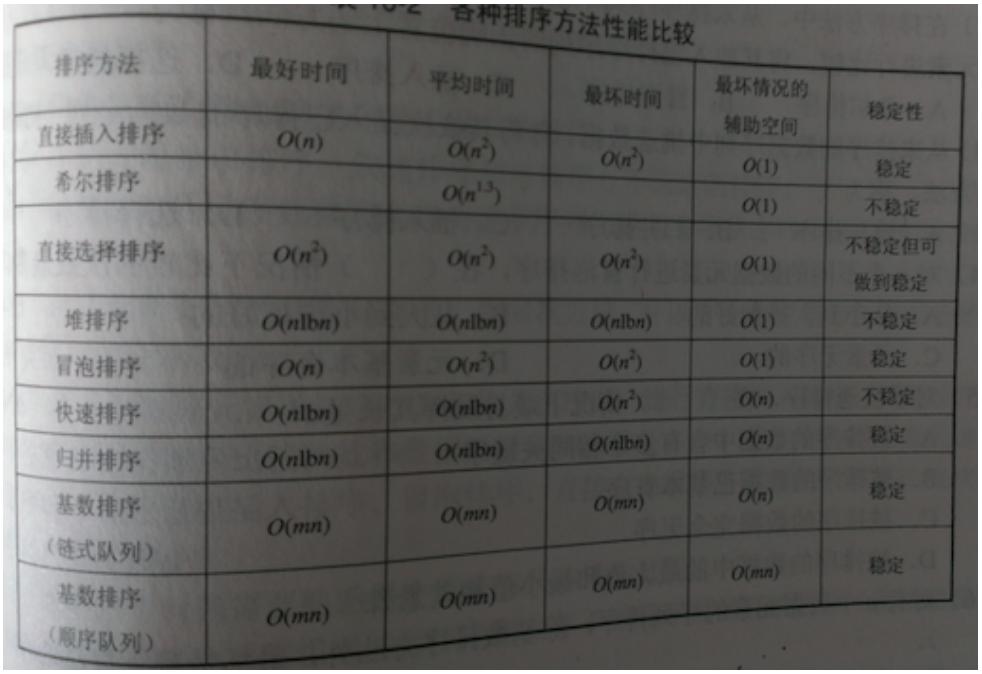
- Pimpl idiom:
  - it is for hiding the implementation. Take all private members to a single struct, only keep a pointer to this struct using forward declaration, if the private members changes, only .cc files that include the implementation should be recompiled.
- Bridge Pattern (allow to develop abstraction and implementation part independently)
  - Having two parts, one part is abstraction, the other part is implementation. Clients only need to the abstraction part but they don't necessary to know the implementation part.
  - In abstraction part: there is a protected pointer to the implementation class.
  - In the implementation part: the abstract part details all the operations it has. The concrete class implement those operations.



33. Measure of design quality:
    - Coupling: the degree to which distinct program modules depend on or affect each other
    - Cohesion: how closely elements of a module are related to each other.
34. Casting in C++:
    - dynamic_cast<expression> can only be used only with pointers and references to objects. Make decisions bases on an object's run-time type information (RTTI). Book* book = new Book(); ComicBook* pcb = dynamic_cast<ComicBook*>(book); Will fail because book is actually pointing to a book not a comic book. pcb would be NULL. otherwise it is ok. Always successful when casting to a base class.
    - static_cast<expression> same as dynamic_cast but no safety checking.
    - reinterpret_cast<expression>: convert any pointer type to anything
    - const_cast<expression>: either cast a non-const object to a const object or cast a const object to a non-const object.
    - RTTI:
        - exposes information about an object's data type at runtime.
        - dynamic_cast and typeid are using RTTI.
        - RTTI is available only for polymorphic classes which have at least one virtual method.
    - typeid:
        - keyword to determine the class of an object at runtime. It is preferred than dynamic_cast because the latter one has to traverse all the class hierarchy at runtime while typeid is a constant-time procedure.
        - e.g

```
Person person;
Employee employee;
Person* ptr = &employee;
Person& ref = employee;
// The string returned by typeid::name is implementation-defined
std::cout << typeid(person).name() << std::endl;   // Person (statically known at compile-time)
std::cout << typeid(employee).name() << std::endl; // Employee (statically known at compile-time)
std::cout << typeid(ptr).name() << std::endl;      // Person* (statically known at compile-time)
std::cout << typeid(*ptr).name() << std::endl;     // Employee (looked up dynamically at run-time
                                                   //           because it is the dereference of a
                                                   //           pointer to a polymorphic class)
std::cout << typeid(ref).name() << std::endl;      // Employee (references can also be polymorphic)

Person* p = nullptr;
try {
   typeid(*p); // not undefined behavior; throws std::bad_typeid
               // *p, *(p), *((p)), etc. all behave identically
}
catch (...)
{}

Person& pRef = *p; // Undefined behavior: dereferencing null
typeid(pRef);      // does not meet requirements to throw std::bad_typeid
                   // because the expression for typeid is not the result
                   // of applying the unary * operator
```

35. Exception handling:
    - When an exception is thrown, controls get back through the call chain (unwinds the stack) until a

handler is found, if no handler is found, the program terminates.
    - throw vs. throw s. throw will retina the actual type of the exception, throw s throws a specific object.
    - Catch exception through references to avoid slicing and to preserve polymorphic behaviours.
    - catch (…) will catch all exceptions.
    - What happens during stack unwinding?
        - All stack-allocated data is cleaned up
        - destructor runs, stack memory deallocated.
        - Heap-allocated memory is not destroyed.
36. RAII(Resource Acquisition is initialization)
    - Everytime you need to acquire some resource, wrap it in a stack allocated object, whose destructor will release it. e.g (a input file stream ifstream f("filename") guranttes to release the resources when f is poped from stack, then a destructor for f runs.
37. Virtual inheritance: used to solve the 'diamond problem'. Will keep a single instance of the common base class. e.g class A{}; class B: public virtual A{}; class C: public virtual A{}; class D: public B, public C
38. Sorting:
    - Stable sort keeps the items with same sorting key in order. (3,2) (3,1) (1,2) -> (1,2) (3,2) (3,1) (stable sort) (1,2) (3,1) (3,2) (unstable sort)
    - When sorting elements that have two keys (e.g first name and last name). Last name firstly sorted then first name latterly sorted. We have to use stable sort twice. Firstly sort the last name, later sort the first name.
    - 
各种排序方法性能比较

排序方法	最好时间	平均时间	最坏时间	最坏情况的辅助空间	稳定性
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希尔排序		$O(n^{1.3})$		$O(1)$	不稳定
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定但可做到稳定
堆排序	$O(n\text{lb}n)$	$O(n\text{lb}n)$	$O(n\text{lb}n)$	$O(1)$	不稳定
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
快速排序	$O(n\text{lb}n)$	$O(n\text{lb}n)$	$O(n^2)$	$O(n)$	不稳定
归并排序	$O(n\text{lb}n)$	$O(n\text{lb}n)$	$O(n\text{lb}n)$	$O(n)$	稳定
基数排序（链式队列）	$O(mn)$	$O(mn)$	$O(mn)$	$O(n)$	稳定
基数排序（顺序队列）	$O(mn)$	$O(mn)$	$O(mn)$	$O(mn)$	稳定

39. Hash map in C++
    - map
        - is implemented as a balanced binary search tree (usually a red/black tree) (no need to have a hash function)
    - unordered_map
        - use the hash table where the key is hashed to a slot in the table and the value is stored in a list tied to that key.
    - An `unordered_map` should give slightly better performance for accessing known elements of the collection, but a `map` will have additional useful characteristics (e.g. it is stored in sorted order, which allows traversal from start to finish). `unordered_map` will be

faster on insert and delete than a `map`

- **`Find the first unique element in a string`**
  - **(1) Use counting hash, create an count array which has a size of 256. Traverse all the string, increase the count by 1 once encountered. Traverse count array and return the element where the count array value is 1.**
- **Find the first unique element in an integer array**
  - **Counting hash. O(n) time complexity, O(max - min + 1) heap space;**
  - **hashtable. add a flag field into each bucket to indicate if the number in this bucket is duplicate. O(n) time complexity, O(n) heap space.**
  - **BST. Add a field into each node to indicate if the number in this node is duplicate. O(nlong(n)), O(n) heap space.**
- **Check if an array has duplicate elements.**
  - **Counting hash. O(n) time complexity, O(max - min + 1)**
  - **heap space, where n is the number of elements in the array.**
  - **hashtable. O(n) time complexity, O(n) heap space, where n is the total number of elements in the array**
  - **BST. O(log(n)) time complexity, O(n) heap space where n is the number of distinct elements in the array.**
  - **Quicksort. O(nlog(n)) time complexity, O(n) worst stack space.**
  - **Heapsort. O(nlog(n)) time complexity, O(1).**
- **Find a missing number in an integer array range from 1~1000;**

40. Project revisit:
41.
    - Basically this project video game which is very similar to tetrics while having many new features. I developed this game under linux environment with many design patterns and c++ programming skills applied.
    - What is the game?
      - is am improved version of tetrics, each block will have their own properties such as transparent block which is very rare but really powerful. It can overlap with any kind of blocks. It’s a bonus in the game. Some blocks are skippable which means the user can decide whether to use it or not. Some blocks will be considered more valuable than other blocks which means user will get more scores if they delete those blocks. There are many different kinds of levels, the system will determine when to increase or decrease the level according to the performance of the user. The probability for each blocks in different levels are different. There are also game board and score board objects that are responsible for updating the graphic view and scores of the user respectively.
    - What kind of design patterns applied?
      - Observer pattern
        - Firstly the traditional observer design pattern has a subject and a observer. In this project, since the GameBoard is composing cells. Thus, in this project I combine the subject and observer together which means a GameBoard is either a

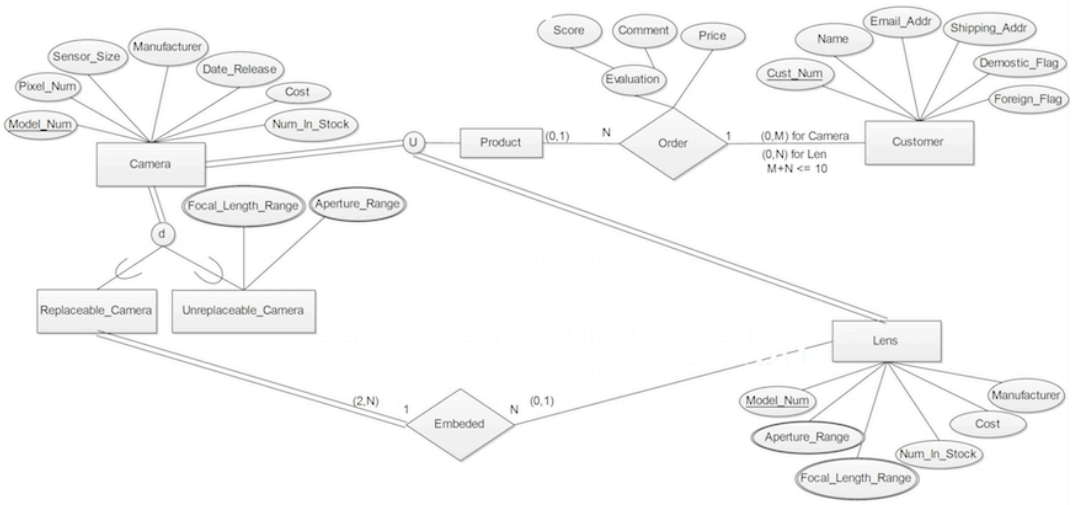
subject and an observer. The cells in the GameBoard are considered to be observed by GameBoard Class and a TextDisplay class who is responsible for updating all command line outputs because it has to get updated about the status of each cell either alive or dead in order to update the outputs in the command line interface.

- Template method pattern
  - There are many different kinds of blocks with different properties. But on top of all, they are blocks and share some common grounds such as symbols, control methods such as shift left, right, down, drop..etc. Thus I created a n abstract class AbstractBlock it has some methods of controlling blocks because the way blocks are controlled is the same. The variations for different blocks is their shape, and reference coordinate because once we rotate a block or drop a block, due to the differences of their shape and reference coordinates the way to determine if the block can fit in the place once it moves to left or drop on to the bottom is different. So I made these functions pure virtual. Each block can define their own behaviours.
- Decorator pattern
  - As I just said, sometimes a bonus block will be generated by the system according to the performance of the user. Those blocks can either be transparent, skippable. shiftLeft, shiftRight, drop should be non-pure virtual, because their behaviours are fixed. rotate, counterroate should be pure virtual and let their subclasses to define their own behaviours.
- Factory method pattern
  - Make decisions based on the runtime. The system will have to decide which block will have high probability to come next. For example, there are three levels. In first level blocks are easy to fit and handle, z block will have very low probability to appear. If it is in level 3, z will have a high probability to appear. Thus I created abstract class Level, and inherit from this class to generate several subclasses to represent level 1,2,3,etc…The products would be all kinds of blocks. Factories are those levels.
- Singleton pattern
  - Each once the game starts, there can only be 1 instance of game board and scoreboard at the same time. Apply singleton pattern to enforce this property.

- How to improve?
  - Avoid using temporary objects (passing by reference, use shorthand symbol +=,-=)
  - Make some small functions inline such as returning a value, modifying a value.

42. Digital Camera Purchasing system:

- A store will be selling two basic varieties of digital cameras depending on an ability to replace lenses.
- Properties of all cameras that are relevant include the manufacturer, model number, date of product release, sensor size, pixel number, retail cost and the number currently in stock.
- Properties of cameras without an ability to replace lenses that are relevant include a focal length range and an aperture range.
- Cameras with an ability to replace lenses are related to at least two or more lenses.
- Properties of a lens that are relevant include the manufacturer, model number, focal length range, aperture range, retail cost and the number currently in stock.
- A prime lens is any lens with only one possible value for a focal length range.
- Online customers are either domestic customers or foreign customers.
- Properties of customers that are relevant include a unique customer number, a customer name, an email address and a shipping address.
- Each customer has up to ten outstanding purchase orders, including possibly none at all.
- Each purchase order is for either a camera or a lens, and will also have a selling price.
- Each camera or lens will have at least one customer evaluation.
- A customer evaluation is given by an individual customer and consists of a score between 1 and 5 and a customer comment.



43. Lowest Common Ancestor problem:
    - In a binary tree

```
BNode* LCA(BNode* root, BNode* n1, BNode* n2)
{
        if (!root)
                return NULL;
        if (root == n1 || root == n2)
                return root;
        BNode* l = LCA(root->left,n1,n2);
        BNode* r = LCA(root->right,n1,n2);
        if (l && r)
                return root;
        return l? l : r;
}

BNode* LCA_BST(BNode* root, BNode* n1, BNode* n2)
{
        if (!root)
                return NULL;
        if (root->val > n1->val && root->val > n2->val)
                return LCA_BST(root->left,n1,n2);
        if (root->val < n1->val && root->val < n2->val)
                return LCA_BST(root->right,n1,n2);
        return root;
}

BNode* LCA_BST_Iterative(BNode* root,BNode* n1,BNode* n2)
{
        while (root)
        {
                if (root->val > n1->val && root->val > n2->val)
                        root = root->left;
                else if (root->val < n1->val && root->val < n2->val)
                        root = root->right;
                else
                        return root;
        }
        return root;
}
```

44. Implementation of Hash table
    - Use map with a list to implement the separate chaining (map<T1, list<T2>)

```
template <typename T1,typename T2>
class Hash_Table
{
private:
        std::map<T1,std::list<T2> > m;
public:
        Hash_Table(){}
        ~Hash_Table(){};
        void add(const T1& key,const T2& val);
        void del(const T1& key,const T2& val);
        void modify(const T1& key,const T2& origin,const T2& val);
        std::list<T2> find(const T1& key);
        void print(const T1& key);
};
```