Effective C++ Tutorial Summary based on S. Mayers books

==========
CONSTRUCTORS, DESTRUCTORS AND ASSIGNMENT OPERATORS
==========

1. Prefer `const` and `inline` to `#define`
    - prefer the compiler to the preprocessor
        - there is sometimes not possible to replace #ifdef preprocessor directive by compiler code but preprocessor should be use as less as possible
    - when you use #define preprocessor change your variable with constant value aliased by it before compilation. Therefore alias will not be placed in symbol table. It can be confusing while you will receive compilation error containing value instead of alias.
    - usage complex expressions in macros could return unexpected results ex.
        - `#define max(a,b) a>b ? a : b`
        - invoking with `max(x++, y)` - x can be incremented two times (if x++ is higher than y or one time otherwise)
        - in such cases we should use ***static inline template function*** instead
2. Prefer `<iostream>` then `<stdio.h>`
    - using operator>>() and operator<<() is more unified. You do not need to remember formatting rules as for scanf() as well as you can overload those operators for each customized class
    -
3. Prefer `new` and `delete` than `malloc()` and `free()`
    - malloc() and free() does not invoke constructors and destructors of objects
    - mixing malloc()/delete and new/free() causes undefined behaviour, it can work but not in all cases - avoid such mixing, please
4. Prefer C++ comments:
    - because of unable to nest C comments prefer C++ comments where possible

==========
DYNAMIC MEMORY MANAGEMENT
==========

5. Use the same form of corresponding uses of `new` and `delete`:
    - when you use `new` for single element use `delete` for single element for it
    - when you use `new` for array use `delete[]` for array as well
    - if you mix it the result is undefined
    - this rule should also be applied for typedef:
        - typedef string stringArray[10];
        - string* myStringArray = new stringArray;
        - we should use delete[] because we are deleting array
6. Use delete on pointer members in destructors:
7. Be prepared for out-of-memory conditions:

- `bad_alloc` exception is thrown when `operator new()` is unable to allocate memory

8. Adhere to convention when write operator new() or operator delete:
   - operators should return appropriate value, throw exceptions when needed and being prepared for dealing with requests for no memory
   - operator new() should throw bad_alloc when out-of-memory occurs
   - pseudocode of sample of relevant operator new() can be found in Effective C++ book by Scott Meyers
   - remember that `operator new()` is inherited by subclasses
9. Avoid hiding the normal form of `operator new()`:
   - if we declare operator new() in the class containing other parameters than normal operator new(), then normal new() will not work. There is needed to declare normal operator new(), too to avoid hiding that operator and allow use it for such class' objects.
10. Write operator `delete` when you write `operator new()`:

==========
CONSTRUCTORS, DESTRUCTORS AND ASSIGNMENT OPERATORS
==========

11. !! Declare copy constructor and assignment operator for classes with dynamically allocated memory
    - if we do not have copy constructor and copy assignment operator declared then compiler uses default copy assignment operator to assign class. This default copy assignment operator uses default copy assignment operator for each member of class. Therefore when we have dynamically allocated members (pointers) one pointer will be overwritten by the second one and we will have memory leak of overwritten memory from first pointer.
    - second problem is that both objects of such class points to the same memory allocated object so if one object will change something such change will be visible in second object as well, which breaks encapsulation of object.
    - also when two pointers points to the same data, we will have double deletion problem when will will destroy both objects
12. Prefer initialization lists than assignments in constructors
    - for const member variables you need to use initialization list in the constructor because const values can only be initialized not assigned
    - references can be also only initialized - not assigned, so for references there is needed to use initialization list
    - performance is another reason of uses initialization list than assignments. Initialization list avoid copy of arguments which makes better performance.
    - the only case when there is better to use assignment was when we have lot of members and wanted to initialize them the same way in many constructors. Then we used init() separate function for it. However from C++11 we can use new syntax to invoke one constructor from another so we do not have to use init() function to initialize members in the same way, yet.

13. List members in the initialization list in the order that they are declared
    - it is because class members are initialized in the order of declaration in the class - not in order of putting to initialization list, so if you put some variable which was declared later first in the initialization list and then use this value in other initialization list parameters this value will not be initialized yet so you can get undefined behaviour
14. Make sure base classes have virtual destructors
15. Have `operator=()` return reference to this()
16. Assign all data members in `operator=()`
    - remember to assign all data members in assignment operator when you are writing class
    - remember to assign new data members in assignment operator when you are updating class
    - in assignment operator of derived classes you should assign data members from base classes as well - the best way is explicitly invoke `operator=()` of base class
    - if you compiler does not allow explicitly invoke operator=() you can use static_cast to invoke it implicitly: `static_cast<Base&>(*this) = rhs;`
    - `Derived(const Derived& rhs) : y(rhs.y);` - this is incorrect copy constructor of derived class. It is because for Base class default constructor is invoked here (instead of copy constructor of Base class). To invoke copy constructor of Base class, too, it should look like here:
      `Derived(const Derived& rhs): Base(rhs), y(rhs.y);`
17. Check for assignments to self in `operator=()`
    - at the beginning of assignment operator there should be something like this: if(this == &rhs) return *this;
    - it is in order to prevent by assigning data members from the same objects if the same object will be passed as right value of operator=(). This can cause problem of ***aliasing***. It occurs when two pointers of different class points to the same data. Then we can have double deletion problem.
    - in the above statement we assume that objects are the same when its addresses are the same.

==========

CLASSES AND FUNCTIONS: DESIGN AND DECLARATION

==========

Page 108 defines questions which every developer should ask himself when he is designing class:

1. How should objects be created and destroyed?
2. How does object initialization differs from object assignemnt?
3. What does it mean to pass object of a new type by value?
4. What are constraints of a legal value of a new type?
5. Does a new type fit into a inheritance graph?
6. What kind of type conversions are allowed?

7. What operators and functions does make sense for a new type?
8. What standard operators and functions should be explicitly disallowed?
9. Who should have access to the members of a new type?
10. How general is new type? Should it be one class or template class?

18. Strive for class interface that are complete and minimal
    - do not put lot of functions into class interface - only minimal number of functions required for class functionality
    - do not define operators which are not going to be need
19. Differentiate between member functions, non-member functions and friend functions
20. Avoid data members in the public interface
    - it breaks encapsulation
21. Use const whenever possible:
    - const exposes intention of author of code to not change its value which is assured by compiler
    - const is something like self-documenting code
22. Prefer pass-by-reference than pass-by-value:
    - better performance - does not need to copy (especially for big objects)
    - allow to invoke derived-class-functions when passed by base class reference - like for pointers
23. Do not try to return reference when you must to return object:
    - we do not return reference because it is only name for existing object so it can change in other place anytime
    - we do not return pointer to object created inside function because we will have memory leak
    - we should return object constructed in-place (to avoid reassignment and additional copying object):
        - return Rational(a,b);
24. Choose carefully between function overloading and parameter defaulting:
    - if you will use reasonable default parameter and you want to employ only single algorithm within function you should use parameter defaulting - otherwise use function overloading
25. Avoid overloading on the pointer and numerical type
26. Guide against potential ambiguity
    - multiple inheritance is fire with possibilities for potential ambiguity
27. Explicitly disallow use of implicitly generated member functions you don't want
    - make such function private (ex. declare `operator=()` private if you do not want to use it)
    - in C++11 there is possible to explicitly disallow it using `= delete` feature.
28. Partition the global namespace:
    - use namespaces for your set of class to avoid ambiguity of variables, classes, functions names

==========
CLASSES AND FUNCTIONS: IMPLEMENTATION
==========

29. Avoid returning handles to internal data:
    - if not needed, do not return values to to pointers (“handles”) to members of a class
    - it can unrestricted access to private members for external objects (there external pointer can points to internal private data after such return). That situation can also cause aliasing - two pointers will point to the same data which is private
    - in the above situation always pointer data should be returned as const
30. Avoid member functions that return non-const pointers or references to members less accessible than themselves.
    - similar problem as above. it can cause breaking encapsulation - if you make public function returning pointer or reference to private data, such private data can be publicly accessible outside of function through that pointer or reference
    - if you definitely have to return such pointer or reference (from performance’s point of view) you should always return it to a `const` object.
31. Never return reference to local object or to a dereferenced pointer initialized by new within the function
    - when you return reference to local object result will be undefined because local object will be destroyed after return (being out-of-scope of function)
    - when you return dereferenced pointer initialized by new within the function there is needed to remember to use delete operator outside of function for each such pointer returned from function. It is very unreliable and can cause memory leaks. There is bigger problem for temporary objects which are not being saved to any variable. For such elements memory leak is obvious.
32. Postpone variable definitions as long as possible
    - when you define variable at the beginning of function which throw exception before its use, you need to pay unneeded performance for variable construction
    - we should also postpone definition of class until we have initialization argument for it
33. Use inlining judiciously
    - inlining can improve performance by putting code of function instead of invoking function, therefore inlining often increases size of code, as well
    - inlining sometimes decreases size of code - for very short inline functions
    - compiler can refuse inlining function - it is often being done for complicated functions (containing loops or being recursive)
    - inline should be signalized in the header files
    - library designers should avoid inline functions because when later such function would change users of library have to recompile library. For normal function relinking is enough.
    - inline function causes problems with debugging (changes numbers of lines, hard to make breakpoints etc.)

34. Minimize compilation dependencies between files:
    - include header files in .cpp files (not in .h files) whenever possible

==========
INHERITANCE AND OBJECT-ORIENTED DESIGN
==========

35. Make sure public inheritance model “isa”
    - `class Bird { public: void fly()}; class Penguin :public Bird` - example of not being isa model. It is because `Penguin` cannot `fly()`. There should be two additional classes in inheritance hierarchy between Bird and Penguin derived from Bird - FlyingBird and NonFlyingBird then inheritance: `class Penguin : public NonFlyingBird`
36. Differentiate between inheritance of interface and inheritance of implementation
    - make pure virtual functions for if you want to allow inheritance of interface of function
    - make non-pure virtual function if you want to allow inheritance of implementation of function
    - make non-virtual function if you want to allow mandatory implementation (implementation which cannot be changed in any derived class)
37. Never redefine inherited nonvirtual function
    - redefinition could lead to unexpected behaviour when we will expect invoking function from Base (B) class and we will receive invoking from Derived (D) class
    - if D really need to redefine nonvirtual function from B it means that D does not fulfill “isa” B rule of
38. Never redefine inherited default parameter value
    - object ***static type*** is type which is written in code during declaration before pointer character (*)
    - object ***dynamic type*** is type which is created using new operator
    - Shape *rect = new Rectangle(); - Shape is static type and Rectangle is dynamic type of our object
    - dynamic types can change through program runs - static type cannot
    - virtual functions are ***dynamically bound*** which means that particular function called is determined by dynamic type of object
39. Avoid casts down the inheritance hierarchy
    - every time when you need to downcast inheritance or check type of derived class to invoke such function it notify you about incorrect design
    - in above situation you should create another class which bear such function and add into inheritance hierarchy between Base class and Derived class
    - if you really have to downcast use safe casting `dynamic_cast` - it will return you dynamic type of object being cast or NULL if casting is impossible
40. Model “has-a” or “is-implemented-in-terms-of” through composition
41. Differentiate between interitance and templates
    - if type of class does not affect behaviour of a class use template
    - if type of class affects behaviour of a class use inheritance

42. Use private inheritance judiciously
    - when you use private inheritance there is impossible to use Derived class as Base class
    - private inheritance means that implementation only should be inherited - interface should be ignored
    - you should use composition when you think about private inheritance
43. Use multiple inheritance judiciously
    - multiple inheritance causes ambiguity
    - multiple inheritance can cause *diamond inheritance hierarchy*
44. Say what you mean, understand what you are saying:
45. Know what functins C++ silently writes and calls
    - C++ compiler declares following functions of class if not defined:
        - copy constructor
        - copy assignemnt operator
        - destructor
        - pair of address-of operators (normal and const address-of operator)
    - those function will be generated only when needed (used inside code)
    - additionally if you do not defined any constructors compiler will create default constructor automatically
46. Prefer compile-time and link-time errors than runtime errors
    - better performance - without runtime checks programs are smaller and faster
    - detecting errors at runtime is much much harder than in compile and link-time
47. Ensure that non-local static objects are initializer before they’re used
48. Pay attention to compiler warnings
49. Familiarize yourself with the standard library
50. Improve your understanding of C++