VE280 2021FA Final RC part1

L14: Subtypes and Inheritance

Substitution Principle

If S is a subtype of T or T is a supertype of S, written $S <: T$, then for any instance where an object of type T is expected, an object of type S can be supplied without changing the correctness of the original computation.

- Functions written to operate on elements of the supertype can also operate on elements of the subtype.
- Benefits: code reuse.

Distinguish: Substitution vs. Type Convertion

Consider the following examples.

- Example 1: Substitution

  Can we use an `ifstream` where an `istream` is expected? Is there any type conversion happening in this piece of code?

  ```
  void add(istream &source) {
      double n1, n2;
      source >> n1 >> n2;
      cout << n1 + n2;
  }

  int main(){
      ifstream inFile;
      inFile.open("test.in")
      add(inFile);
      inFile.close();
  }
  ```

- Example 2: Type Convertion.

  Can we use an `int` where a `double` is expected? Is there ant type conversion happening in this piece of code?

  ```
  void add(double n1, double n2) {
      cout << n1 + n2;
  }
  ```

```
int main(){
    int n1 = 1;
    int n2 = 2;
    add(n1, n2);
}
```

Creating Subtypes

In an Abstract Data Type, there are three ways to create a subtype from a supertype:

1. Add operations or methods.
2. Strengthen the postconditions of operations
    - Postconditions include:
        - The EFFECTS clause:

          e. g., we print an extra message to `cout` in the new class while returning the same thing
        - The return type:

          e. g., we behave normally under the raw REQUIRES condition and return a special value under the newly allowed REQUIRES condition
3. Weaken the preconditions of operations
    - Preconditions include:
        - The REQUIRES clause:

          e. g., we allow negative integers for `insert()`
        - The argument type

          e. g., `void operation (PosIntSet s)` -> `void operation (IntSet s)` where `PosIntSet` is a subtype of `IntSet`

Inheritance Mechanism

When a class (called derived, child class or subclass) inherits from another class (base, parent class, or superclass), the derived class is automatically populated with almost **everything from the base class.**

- This includes member variables, functions, types, and even static members.
- The only thing that does not come along is friendship-ness.
- We will specifically discuss the constructors and destructors later.

The basic syntax of inheritance is:

```
class Derived : /* access */ Base1, Base2, ... {
private:
    /* Contents of class Derived */
public:
    /* Contents of class Derived */
};
```

Access Specifier

There are three choices of access specifiers, namely `private`, `public` and `protected`.

The accessibility of members are as follows:

specifier	private	protected	public
self	Yes	Yes	Yes
derived classes	No	Yes	Yes
outsiders	No	No	Yes

When declaring inheritance with access specifiers, the accessibility of (members that get inherited from the parent class) in the derived classes are as follows:

Inheritance \ Member	private	protected	public
private	inaccessible	private	private
protected	inaccessible	protected	protected
public	inaccessible	protected	public

When you omit the access specifier, the access specifier is assumed to be `private`, and the inheritance is assumed to be `private` as well.

Pointer and Reference in Inheritance

From the language perspective, C++ simply trusts the programmer that every subclass is indeed a subtype. We have the following rules.

- Derived class pointer is compatible to base class pointer, i.e., we could run `PosIntSet* p1, IntSet* p = p1`
- Derived class instance is compatible to base class reference, i.e., we could run `PosIntSet s, IntSet& p = s`
- You can assign a derived class object to a base class object, i.e., we could run `PosIntSet s, IntSet p = s`

The reverse is generally false. E.g., assigning a base class pointer to derived class pointers needs special casting (dynamic_cast).

Apparent type and Actual type

Apparent type: the declared type of the reference. (IntSet)

Actual type: the real type of the referent. (PosIntSet)

In default situation, C++ chooses the method to run based on its apparent type.

```
PosIntSet s; // Actual type: PosIntSet
IntSet& r  = s; // Apparent type: IntSet

try {
    r.insert(-1);
} catch (...){
    cout << "Exception thrown\n";
}
```

Result: we insert `-1` into `s`, and this breaks the abstraction of the set `s`, and that's why we need `virtual`.

Virtual Functions - Dynamic Polymorphism

`virtual` keyword

A way to tell C++ compiler to choose the **actual type at run-time** before execution.

Using the previous example:

```
class IntSet{
    ...
public:
    ...
    virtual void insert(int v);
    ...
};
```

The above syntax marks insert as a `virtual` function.

First, for each class with virtual functions, the compiler creates a **vtable** (or **virtual table**) with one function pointer for each virtual function initialized to the appropriate implementation. Then, each instance of a class with virtual methods has both the class state, plus a pointer to the appropriate vtable.

Virtual methods are methods **overridable by subclasses.** When a method call is made, if the method you are calling is a virtual function (based on the apparent type), the compiler binds the method according to the **virtual table**. In this way, the function `insert` achieves dynamic polymorphism, the ability to change its behavior based on the actual type of the argument.

`override` keyword

The act of replacing a function is called overriding a base class method. The syntax is as follows.

```
class PosIntSet: public IntSet{
    ...
public:
    ...
    void insert(int v) override;
    ...
};
```

`override` cause the compiler to verify if a function is indeed overriding a base class method. If the base class method is not a virtual function, compiler will complain. The keyword is introduced in C++11. **It is considered good programming style to always mark override whenever possible.**

Operator Overloading

C++ lets us redefine the meaning of the operators when applied to **objects of class type**. This is known as operator overloading. Like any other function, an overloaded operator has a return type and a parameter list.

Unary operators

An overloaded unary operator has no (explicit) parameter if it is a member function and one parameter if it is a nonmember function.

```
A& A::operator++(); // ++a
A A::operator++(int); // a++
A& A::operator--(); // --a
A A::operator--(int); // a--
void operator++(& A); // ++a for nonmember function
```

Binary operators

An overloaded binary operator would have one parameter when defined as a member and two parameters when defined as a nonmember function.

```
A& A::operator= (const A& rhs);
A& A::operator+= (const A& rhs);
A& A::operator-= (const A& rhs);
A operator+ (const A& lhs, const A& rhs);
istream& operator>> (istream& is, A& rhs);
ostream& operator<< (ostream& os, const A& rhs);
bool operator== (const A& lhs, const A& rhs);
bool operator!= (const A& lhs, const A& rhs);
```

```
bool operator< (const A& lhs, const A& rhs);
bool operator<= (const A& lhs, const A& rhs);
bool operator> (const A& lhs, const A& rhs);
bool operator>= (const A& lhs, const A& rhs);
// especially
const T& A::operator[] (size_t pos) const;
T& A::operator[] (size_t pos);
```

`friend` Keyword

We may want to access private member of class instances. You could provide an accessing operator for each of the member, but often it is not a good idea. One workaround is specifically grant access to the protected members. This can be done by using the `friend` keyword:

```
class Bar {
    friend void foo (const MyClass &mc);
}
```

It doesn't matter where this is marked public or private.

`friend` can also grant access to classes, and Baz now can access private mamber of Bar:

```
class Bar {
    friend class Baz;
}
```

Pay attention that friend is not mutual. If Class A declares Class B as friend. Class B can access Class A's private member, but the other way around doesn't work.

L15: Interfaces and Invariant

Abstract Base Class - Classes as Interfaces

The interface is the **contract for using things of this type (mainly about what I can do with this type).**

The normal C++ class failed to be a perfect interface, it also did not achieve perfect information hiding. It mixes details of the implementation with the definition of the interface. Although the method implementations can be written separately from the class definition and are usually in two separate files. Unfortunately, **the data members still must be part of the class definition.** Since any programmer using your class see that definition, those programmers **know something about the implementation.**

What we prefer is to create an "interface-only" class as a base class, from which an implementation can be derived. Such a base class is called an **Abstract Base Class**, or sometimes a **Virtual Base Class.**

Note:

- An Abstract Base Class **cannot have an real implementation for its methods** since it does not have any data members.
- Also, we **could not create an instance of Abstract Base Class** since it doesn't have constructors.

Pure `virtual`

Because there will be no implementation, we need to declare methods in a special way

```
/* IntSet */ virtual void insert(int i) = 0;
```

In this case we say the method is pure virtual.

If a class contains one or more pure virtual methods, we say the class is an **abstract class**. **You only need to have one pure virtual function for a class to be "abstract".** Pure virtual class are also called abstract base classes, or interfaces.

As mentioned before, **you can not create an instance of an abstract class.** However, you can always define references and pointers to an abstract class, and use that to refer to an instance of derived class which implements the interface. That is how we deal with `Player` in P4:

```
Player* player = (type == "simple")? getSimplePlayer() : getCountingPlayer();
```

Invariant

An invariant is a set of conditions that must **always evaluate to true** at certain well-defined points; otherwise, the program is incorrect. For ADT, there is so called **representation invariant.**

It describes the conditions that must hold on those members for the representation to correctly implement the abstraction. It must hold immediately before exiting each method of that implementation, including the constructor.

Each method in the class can assume that the invariant is true on entry if the following 2 conditions hold:

- The representation invariant holds **immediately before exiting each method** (including the constructor);
- Each data member is truly private (other functions outside the class could not modify the data member).

E.g., `numElm` should always equal the number of elements in `IntSet` after calling each method (`insert`, `remove`).

Check your understanding

```
#include <iostream>
using namespace std;
class Foo {
    public:
    void f() { cout << "a"; };
    virtual void g() = 0;
    virtual void c() = 0;
};
class Bar : public Foo
{
    public:
    void f() { cout << "b"; };
    virtual void g() { cout << "c"; };
    void c() { cout << "d"; };
    virtual void h() { cout << "e"; };
};
class Baz : public Bar
{
    public:
    virtual void f() { cout << "f"; };
    virtual void g() { cout << "g"; };
    void c() { cout << "h"; };
    void h() { cout << "i"; };
};
class Qux : public Baz
{
    public:
    void f() { cout << "j"; };
    void h() { cout << "k"; };
};
int main() {
    Bar bar;    bar.g();
    Baz baz;    baz.h();
    Qux qux;    qux.g();
    Foo &f1 = qux;
    f1.f();    f1.g();    f1.c();
    Bar &b1 = qux;
    b1.f();    b1.c();    b1.h();
    Baz &b2 = qux;
    b2.f();    b2.c();    b2.h();
    return 0;
}
```

The answer is shown below.

```
int main() {
    Bar bar;    bar.g(); // c
    Baz baz;    baz.h(); // i
    Qux qux;    qux.g(); // g
    Foo &f1 = qux;
    f1.f();    f1.g();  f1.c(); // a g h
    Bar &b1 = qux;
    b1.f();    b1.c();  b1.h(); // b h k
    Baz &b2 = qux;
    b2.f();    b2.c();  b2.h(); // j h k
    return 0;
}
```

L16: Dynamic Memory Allocation & Destructor

Why dynamic memory?

- Local varibales can only be fixed size, we want to change the size at runtime.
- We don't know the exact lifetime of the objects.

Allocation

```
Type* obj0 = new Type; // Default construction
Type* obj1 = new Type(); // Default construction
Type* obj2 = new Type(arg1, arg2); // Constructor with 2 params
Type* objA0 = new Type[size]; // Default cons each elt
Type* objA1 = new Type[size](); // Same as obj A0
```

However, you cannot allocate an array of objects using non-default constructors.

When using `new` and `new[]`, following things happens:

1. Allocates space in heap (for one or an array of objects).
2. Constructs object in-place (mainly be calling **constructor**).
3. Returns the “first” address.

Deallocation

Use `delete` and `delete[]` to deallocate single object and arrays respectively.

Difference between delete and delete[]

```
string *S = new string[3];
delete[] S;
```

```
string *s = new string;
delete s;
```

Memory Leak

The usage of `new/delete` is very easy. The difficult point is when and where to use them.

Basically, memory leak happens **when you lose the address of some dynamic memory** (then it would be impossible for you to `delete` that memory).

```
// Each time foo() is called, there is new memory allocated.
// And since p is a local variable, each time p will point to a new address
void foo() {int* p = new int(0); /* Code */}
```

Each time `foo()` is called, some memory is occupied and will not be released even after the program terminates. Gradually, you will drain out all memory of your computer, which is very bad.

You may try to solve the problem by writing:

```
void foo() {int* p = new int(0); /* Code */ delete p;}
```

we need a universal pattern (or strategy) to ensure there is no memory leak.

RAII

Stands for **Resource Acquisition Is Instantiation,** also known as **Scope-based Resource Management**. This is a C++ programming technique which **binds the life cycle of a resource** that must be acquired before use **to the lifetime of an object**.

In simple words:

- Resource is allocated in constructors and constructors only.
- Resource is released in destructors.
- Object “owns” the resources. The resource is managed by the object and the object only.
- The resources would share it’s life cycle with object. As long as there is no object leak there is no resource leak. Fortunately we know automatic objects are destroyed by the compiler when they go out of scope, impossible to have object leak.

Destructors

- Be named as ~ClassName.
- Takes no argument and returns nothing (Not even void).

- If one expect the class to be inherited, the destructor should be declared as virtual.
- Release resource allocated only in this class (don't release base class resources!).

Destructors are called automatically when exiting a scope, throwing an exception, terminating the program or using `delete`. When destructing an object:

- It calls the destructor of the class.
- Calls the destructors for each member of current class.
- Calls destructor of the base class.
- Does above **recursively** until no more destructors to invoke. Finally it releases the memory.

Destructor v.s. Constructor

About virtuality:

- constructor cannot be `virtual`
- If one class is going to be inherited by other classes, its destructor must be `virtual`

About calling time:

- constructor is called when an object is being created (either global, local or dynamic). However, constructors of base classes will not be called unless you **explicitly** do so (only call the default constructor).
- destructor is called automatcially when an object reaches the end of its lifetime. Destructors of base classes are then called recursively.

Constructors and Destructors in Inheritance

The order of constructor and destructor call in an inheritance system would be:

Consider the following example:

```
class Parent {
public:
    Parent() { cout << "Parent::Constructor\n"; }
    virtual ~Parent() { cout << "Parent::Destructor\n"; }
};

class Child : public Parent {
public:
    Child() : Parent() { cout << "Child::Constructor\n"; }
    ~Child() override { cout << "Child::Destructor\n"; }
};

class GrandChild : public Child {
public:
    GrandChild() : Child() { cout << "GrandChild::Constructor\n"; }
    ~GrandChild() override { cout << "GrandChild::Destructor\n"; }
};

int main() {
    GrandChild gc;
}
```

The output would be:

```
Parent::Constructor
Child::Constructor
GrandChild::Constructor
GrandChild::Destructor
Child::Destructor
Parent::Destructor
```

Dynamic Array of Pointers

```
#include <iostream>
using namespace std;

class TestClass {
private:
    int val;
public:
    TestClass() { cout << "A default constructor" << endl; }
    TestClass(int a): val(a) { cout << "A one-param constructor" << endl; }
    ~TestClass() { cout << "A destructor" << endl; }
};

int main() {
```

```
    size_t num = 3;

    TestClass* pointer_to_array = new TestClass[num];
    delete[] pointer_to_array;

    TestClass** array_of_pointers = new TestClass*[num];

    for (size_t i = 0; i < num; i++) array_of_pointers[i] = new TestClass(i);

    // First delete the objects pointing to, then delete the memory for
    // pointers themselves
    for (size_t i = 0; i < num; i++) delete(array_of_pointers[i]);
    delete[] array_of_pointers;
    return 0;
}
```

This example shows the difference between a dynamic array of values and a dynamic array of pointers. You can compile it and investigate what happens at each step.

Thinking: Can the order of `delete` be reversed?

Common Issues

- Try to free the non-dynamic allocated memory

```
void MLF()
{
    char * str = "abc";
    delete str;
}
```

- Memory Leak when changing the pointer

```
void MLF()
{
    int * pint = new int(3);
    ++pint;
    delete pint;
}
```

- Memory Leak in Inheritance (Remember the virtual mechanism)

```
class Base {
protected:
    int *p;
```

```
public:
    Base() : p(new int(10)) {}
    ~Base() {delete p;}
};

class Derived : public Base {
    int *q;
public:
    Derived() : Base(), q(new int(20)) {}
    ~Derived() {delete q;}
};

// Leak!
void foo() {
    Base* ptrA = new Derived;
    delete ptrA;
}

// Safe
void bar() {
    Derived* ptrB = new Derived;
    delete ptrB;
}
```

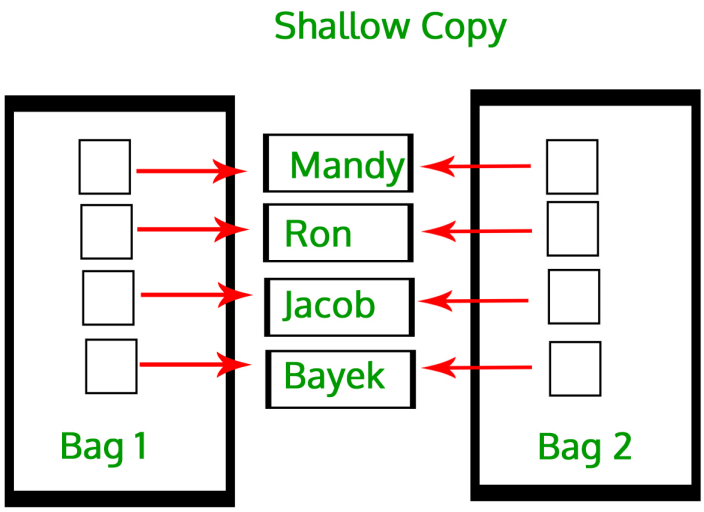
- Double Free (Only care about your own resource!)

```
class Base {
protected:
    int *p;
public:
    Base() : p(new int(10)) {}
    virtual ~Base() {delete p;}
};

class Derived : public Base {
    int *q;
public:
    Derived() : Base(), q(new int(20)) {}
    virtual ~Derived() override {delete p; delete q;}
};

// Double Free
void foo() {
    Base* ptrA = new Derived;
    delete ptrA;
}

// Double Free
void bar() {
    Derived* ptrB = new Derived;
    delete ptrB;
}
```

L17: Deep Copy

Shallow Copy & Deep Copy

Because C++ does not ***know much about your class***, the ***default copy*** and ***default assignment operator*** it provides use a copying method known as a member-wise copy, also known as a shallow copy.



This works well if the fields are ***values***, but may not be what you want for fields that point to ***dynamically allocated memory***. The pointer will be copied. but the memory it points to will not be copied: the field in both the original object and the copy will then point to the same dynamically allocated memory, this causes problem at erasure, causing **dangling pointers**.

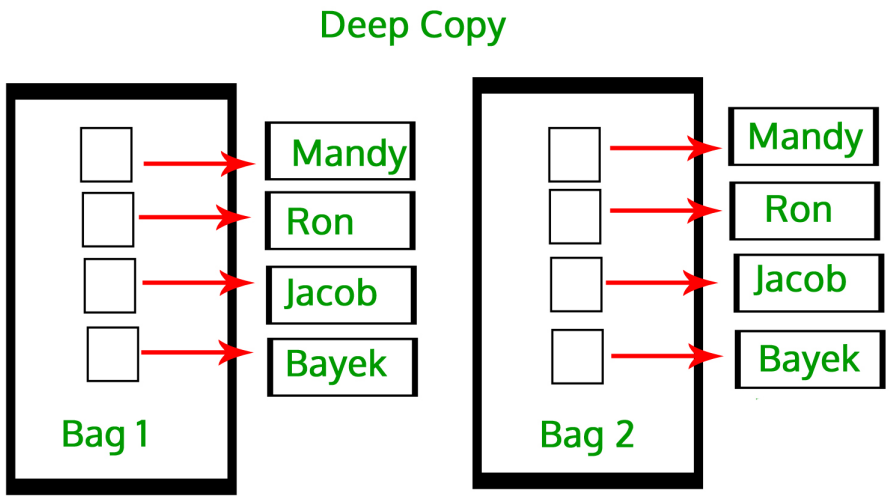
```
#include <iostream>
using namespace std;
const int MAX_CAPACITY = 10;
class Bag
{
    string *items;
    public:
    Bag();
    void insert(string str); // implementation omitted
};
Bag::Bag() : items(new string[MAX_CAPACITY])
{
}
int main()
{
    Bag bag1;
    bag1.insert("VE280");
    Bag bag2 = bag1;
}
```

What is the terrible result?

1. When you change the value of items in bag2, then the items in bag1 also changes.
2. What if you have a destructor to destruct this class?

What does deep copy do?

Instead, a *deep copy* copies all fields, and makes copies of dynamically allocated memory pointed to by the fields.



The Rule of the Big 3/5

If you have any dynamically allocated storage in a class, you must follow this Rule of the Big X, where X = 3 traditionally and X = 5 after c++11.

Whenever an object owns resources, any resources, not just memory, it should implement 5 methods: **a destructor**, **a copy constructor**, a move constructor, **a copy assignment operator**, and a move assignment operator.

A reminder:

```
class MyClass {
    // Member variables
public:
    MyClass(MyClass &that); // Copy constructor
    MyClass &operator=(const MyClass &that); // Overload '=', assignment operator
    void detroy(); // Destructor helper
    ~MyClass(){detroy();} // Destructor
    // Other member functions omitted
};

MyClass::MyClass(MyClass &that)
{
    if (this == &that){   // Extremely important considering self-assignment
```

```
        return;
    }
    else{
        destory(); // Destruct this
        // Do deep copy
    }
}

MyClass & MyClass::operator=(const MyClass &that)
{
    if (this == &that){   // Extremely important considering self-assignment
        return *this;
    }
    else{
        destory(); // Destruct this
        // Do deep copy
    }
}
```

These are 5 typical situations where resource management and ownership is critical. You should never leave them unsaid whenever dynamic allocation is involved. Traditionally **copy constructor/destructor/copy assignment operator** forms a rule of 3. Move semantics is a feature available after C++11, which is not in the scope of this course.

Usually, we would need to implement some private helper functions `removeAll()` and `copyFrom()`, and use them in the big 3. Consider the `Dlist` example.

- A destructor

```
template <class T>
Dlist<T>::~Dlist() {
    removeAll();
}
```

- A copy constructor

```
template <class T>
Dlist<T>::Dlist(const Dlist &l): first(nullptr), last(nullptr) {
  copyAll(l);
}
```

- An assignment operator

```
template <class T>
Dlist<T> &Dlist<T>::operator=(const Dlist &l) {
    if (this != &l) {
        removeAll();
```

```
        copyAll(l);
    }
    return *this;
}
```

Exercise

Recall binary tree and in-order traversal. We define that a good tree is a binary tree with ascending in-order traversal. How to deep copy a template good tree provided interface:

```
template <class T>
class GoodTree {
        T *op;
        GoodTree *left;
        GoodTree *right;
public:
    void removeAll();
    // EFFECTS: remove all things of "this"
    void insert(T *op);
    // REQUIRES: T type has a linear order "<"
    // EFFECTS: insert op into "this" with the correct location
    //                  Assume no duplicate op.
};
```

You may use `removeAll` and `insert` in your `copyAll` method.

The sample answer is as follows.

```
template <class T>
void GoodTree<T>::copy_helper(const GoodTree<T> *t) {
    if (t == nullptr)
        return;
    T *tmp = new(t->op);
    insert(tmp);
    copy_helper(t->left);
    copy_helper(t->right);
}

template <class T>
void GoodTree<T>::copyAll(const GoodTree<T> &t) {
    removeAll();
    copy_helper(&t);
}
```

L18: Dynamic Resizing

Why do we need Dynamic Resizing?

In many applications, we do not know ***the length of a list in advance***, and may need to grow the size of it when running the program. In this kind of situation, we may need dynamic resizing.

Array Example

When do we use Dynamic Resizing?

When the array is at maximum capacity, we will grow the array. `grow()`:

- The grow method won't take any arguments or return any values.
- It should never be called from outside of the class, so add it as a private method taking no arguments and returning void.

How to implement a `grow()` function?

In general, there are four steps:

1. Allocate a bigger array.
2. Copy the smaller array to the bigger one.
3. Destroy the smaller array.
4. Modify elts/sizeElts to reflect the new array.

If the implementation of the list is a dynamically allocated array, we need the following steps to grow it:

- Make a new array with desired size. For example,

```
int *tmp = new int[new_size];
```

- Copy the elements from the original array to the new array iteratively. Suppose the original array is `arr` with size `size`.

```
for (int i = 0; i < size; i++){
    tmp[i] = arr[i];
}
```

- Replace the variable with the new array and delete the original array. Suppose the original array is `arr`:

```
delete [] arr;
arr = tmp;
```

- Make sure all necessary parameters (representation variants) are updated. For example, if the `size` of array is maintained, then we can do:

```
size = new_size;
```

Common selections of `new_size`

- `size + 1`: This approach is simplest but most inefficient. Inserting `N` elements from capacity 1 needs `N(N-1)/2` number of copies.
- `2*size`: Much more efficient than `size+1`. The number of copies for inserting `N` elements becomes smaller than `2N`.
- What about even larger (eg: `size^2`)? Usually not good, for it occupies far too much memory.

Reference

[1] Weikang, Qian. VE280 Lecture 14-18, 22.

[2] Jiajun, Sun. VE280 RC6. 2021FA.

[3] Yunuo, Chen. VE280 RC7. 2021FA.

[4] Changyuan, Qiu. VE280 Final Review Slides Part 1. 2020FA.

[5] Chen, Huang. FinalRC_21-26. 2020FA