VE280 Recitation Class Notes

VE280 SU19 TA Group

UM-SJTU-JI

July 24, 2019

Source code available at `https://github.com/ve280/VE280-Notes`
Slides modified from the version of YAO Yue's, with permission

RC Week 10

Sub-types, Code Reuse and Inheritance

Principals of Object Oriented Programming

There are 5 widely accepted principles throughout the object oriented design:

1. Single responsibility principle
2. Open/close principle
3. **Liskov substitution principle**
4. Interface segregation principle
5. Dependency inversion principle

- The idea of "abstract data type" by first proposed by Barbara Liskov and Stephan Zilles (1974) in "Programming with abstract data types".
- Later on in Liskov's 1988 key note "Data Abstraction and Hierarchy", the SOLID principles of object oriented programming was first proposed.

Principals of Object Oriented Programming

Barbara Liskov (1939 to present)

- MIT computer scientist
- Ford Professor of Engineering
- One of the American's first women PH.D. in Computer Science

- 2004 John von Neumann Medal winner for "fundamental contributions to programming languages, programming methodology, and distributed systems"
- 2008 Turing Award winner, for her work in the design of programming languages and software methodology that led to the development of object-oriented programming

Sub types and Liskov Substitution Rule I

The substitution rule is formally described as follows:

If S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of T (correctness, task performed, etc.)

This is by all means very abstract. We provide a more "concrete" explanation.

Subtype relation is an "IS-A" relationship.

For examples, a `Swan` is a `Bird`, thus a `class Swan` is a subtype of `class Bird`. A bird can fly, can quake and can lay eggs. A swan can also do these. It might do these better, but as far as we are concerned, we don't care. `Bird` is the super-type of the `Swan`.

Pre-conditions & Post-conditions

The term "sub-type" is misguiding because the prefix "sub" suggests the sub-type is "inferior" to the super-type in some sense. But this is actually other way around.

- The assertions in the `REQUIRES` clause and argument types in abstraction specs combined is called a "Precondition".
- The assertions in the `EFFECTS` and `MODIFIES` clause specifies the post conditions.

Sub-types typically perform a combination of belows:

- Supports extra operation. Naturally preserves follows LSR.
- Weakens the pre-conditions. Expands range of input.
- Strengthens the post-condition. Enhances the effect.

In reality sub-types are beefed-up versions of the super-type. They perform more specific jobs or do the same job better.

Inheritance

We now start to look at a specific language feature in C++, namely *inheritance*. The syntax takes the form of the following:

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

When a class (usually called *derived*, *child* class or *subclass*) inherits from another class (*base*, *parent* class, or *superclass*), the derived class is automatically populated with *everything* from the base class. *Everything* includes member variables, functions, types, and even static members. The only thing that does not come along is *friend*ship-ness, which is irony.

Note that, it is one thing that the derived class "has" everything from the base class, it is totally another thing whether the base class can access it. Whether the base class can access the member is dependent on the choice of `access`.

Inheritance access specifiers: `public`

There are a three choices of access, namely `private`, `public` and `protected`. The most commonly used one is `public`.

- Private member of the base class stays private *to the base class*. Even the derived class CANNOT touch them!
- Public member of the base class stays public. This means they are exposed as part of the interface of the derived class.

```
class Base {
    private: int priv; void privMethod();
    public:  int pub;  void pubMethod(); };
class Derived : public Base {
    void bothError() { priv = 0; privMethod(); /* Error */ }
    void bothOK()    { pub = 0;  pubMethod();  /* OK    */ }
} derived;
void bothError() { derived.priv = 0; derived.privMethod(); }
void bothOK   () { derived.pub = 0;  derived.pubMethod(); }
```

Inheritance access specifiers: `private`

When you omit the access specifier, the access specifier is assumed to be `private`. The `private` specifier follows the following rule.

- Private member of the base class stays private *to the base class*. Same as before.
- Public member of the base class are still accessible to the derived class. However they are no longer part of the interface of derived class. I.e. cannot be access from outside.

```
class Base { ... };
class Derived : private Base {
    void bothError() { priv = 0; privMethod(); /* Error */ }
    void bothOK()    { pub = 0;  pubMethod();  /* OK    */ }
} derived;
void test() {
derived.priv = 0; /* Error */ derived.privMethod(); /* Error */
derived.pub = 0;  /* Error */ derived.pubMethod();  /* Error */
};
```

Member access specifiers: `protected`

The first thing you need to hear about this keyword, is **DO NOT ABUSE IT**. Use it with extra care. The fact that base class private member is not accessible to derived class sometimes causes in convenience. We would like something that:

- Not accessible by the outside world.
- Accessible for derived class, if inherited as public.

The key word that satisfies such need is `protected`.

```
class Base { protected: int i; int prot();};
class Derived : public Base {
    void bothOK() { i = 0;  prot();  /* OK */ }
} derived;
void bothError () { derived.i = 0; derived.prot(); };
```

We would really like to restrain ourself from using the keyword, since allowing other class to access a private comes with the possibility of the other class breaking invariants.

Two aspects of inheritance

There are two aspects of inheritance, namely:

Inheritance of code: reusing code

The derived class now comes with the same set of "code", or contents of the base class. This essentially saves us time and effort of coding same thing again and again.

Inheritance of interface

With public inheritance, the derived class will have the same interface (well, at least same method signatures...) as the base class. If used corrected, this creates a *subtype*.

Private inheritance is sometimes referred as *implementation inheritance*, since it allows the derived class can reuse the code of the base class (as if the base class is a private member variable of derived class), without exposing the base class.

We will assume public inheritance in the rest of this chapter.

Remark: Inheritance & subtyping

There are two remarks that I would like to make. First of all, subtypes does not have to be created from inheritance. In the following code, class B is definitely a subtype of class A.

```
class A {public:void quak(){puts("Hello.");} };
class B {public:void quak(){puts("Hello.");} void nop();};
```

On the other hand, having identical interface (or inheritance) does not guarantee subtyping relation. For example (why?):

```
class A { protected: int a = 0;
          public:    int add(int i){ return i + a;} };
class B : public A { public:  B() : a(10) {} };
```

Although inheritance is neither a sufficient nor a necessary condition of subtyping relation, it IS the only subtyping method supported by C++ (without a hack) in runtime. We will see this shortly after.

Pointer, reference and sub classing

From our previous discussion we see that there is no definite rule between a class being as subclass and a class being a subtype. However, from the language perspective, C++ simply **trusts** the programmer that every subclass is indeed a subtype. This is visible from the following rules:
Suppose we have `class Base` and `class Derived`, with `Derived` being a subclass of `Base`. We have the following rule.

- Derived class pointer compatible to base class.
- Derived class instance compatible to base class (possibly `const`) reference.
- *You can assign a derived class object to a base class object.

Two remarks. 1) The last rule needs further explanation. 2) Those rules are NOT true if reversed. For example, you cannot (normally) assign a base class object to derived class object. Assigning a base class pointer to derived class pointers needs special casting.

Pointer, reference and sub classing: Example

Code in code/rc10compatible

```
// class.h
class Base {
public: string str;
Base(const Base& base)
: str(base.str) {
    cout << "cp Base\n"; }
void print() {
    cout << a << endl; }
};
class Derived : public Base {
public: // ...
Derived(const Derived& d)
: Base(d) {
    cout << "cp Derived\n"; }
};
```

```
#include "class.h"
// class.cpp
void test() {
    Derived d; d.a = "hello";
    d.print(); /* hello */
    Base* bp = &d; bp->a = "ha";
    d.print(); /* ha */
    bad(&d); d.print(); /* bad */
    good(d); d.print(); /* good *
    pbase(d); /* good */
}
void bad(Base* base) {
    base->a += "bad"; }
void good(Base& base) {
    base.a += "good"; }
void pbase(const Base& base) {
```

Pointer, reference and sub classing: The third rule

Now the previous example demonstrates the first two roles. We know how things worked with reference and pointers. To explain the third rule, we need to understand first how synthesized methods worked with inheritance. Note that here we mainly focus on how copy constructors works. The case for the assignment operator overload is very similar.

Here we will use the following base class as an example:

```
// base.h
class Base {
    string str;
public:
    Base() { cout << "default base\n"; }
    Base(const Base& other) { cout << "copy base\n"; }
};
```

Code in `code/rc10synthesize`

Pointer, reference and sub classing: Synthesized methods

```
// ok.cpp
class Derived1 : public Base {};
class Derived2 : public Base {
public: Derived2() = default;
        Derived2(const Derived2& d2) : Base(d2) {
            cout << "copy derived 2\n"; } };
int main() {
    Derived1 d1; Derived1 d1c(d1); /* cp base */
    Derived2 d2; Derived2 d2c(d2); /* cp base; cp d2 */ }
```

A synthesized copy constructor will do things almost identical to synthesized default constructor.

- **Copy construct** the base class. See `Derived1`.
- **Copy construct** every member, if there is any.
- Call the copy constructor of the class.

The case of `Derived2` shows how we do this manually.

Pointer, reference and sub classing: Synthesized methods II

```
// error.cpp
class Derived3 : public Base {
public: Derived3(const Derived3& d3) : Base(d3) {
            cout << "copy derived 3\n"; } };
class Derived4 : public Base {
public: Derived4() = default;
        Derived4(const Derived4& d4) {
            cout << "copy derived 4\n"; } };
int main() {
        Derived1 d3; Derived1 d3c(d3); /* Compile Error */
        Derived2 d4; Derived4 d2c(d4); /* dft base; cp d4 */}
```

Here are mistakes that people usually make. Without default constructor (Derived3), since you already provided a constructor, compiler won't synthesize default constructor for you! Without copy constructing the base (Derived4), the compiler will treat it as if the cpctor is a usual constructor, defaulting constructing the base and all members.

Pointer, reference and sub classing: Copying

Code in code/rc10compatible

```
// class.h
class Base {
public: string str;
Base(const Base& base)
: str(base.str) {
    cout << "cp Base\n"; }
void print() {
    cout << a << endl; }
};
class Derived : public Base {
public: // ...
Derived(const Derived& d)
: Base(d) {
    cout << "cp Derived\n"; }
};
```

```
// copy.cpp
void test() {
    Derived d; d.a = "hello";
    Base b = d;    /* cp Base */
    passByVal(d);
    /* cp Base; hellofoo; */
    d.print() /* hello */
    Derived d2 = d;
    /* cp Base; cp Derived; */
    Derived d2 = b; /* Error */
}

void passByVal(Base base) {
    base.str += "foo";
    base.print(); }
```

Digression: Inheritance and memory map

We now digression to the question of how does this work exactly. If we print the address of the objects in our previous example:

```
void addr1() {Derived d;  Base* b = &d; cout << b; }
void addr2() {Derived d;  Base& b = d; cout << &b; }
```

You will see the same address being printed over and over again. Consider a function a function taking an base class pointer as argument. The function will have no idea the address passed to it, is indeed a base class object, or a derived class object "disguised" as a base class object.

The derived class object always "embeds" a base class object at the begging of its corresponding memory region. Memory layout of Derived object looks like below.

```
struct Derived { Base base; /* Derived members */ };
```

This also explains why base class objects are initialized as if they are member variables, because to a degree they are!

Introducing new methods to create subtypes

Now we go back to the question of subtypes. Recall the 3 three ways of creating subtypes. Two involves modifying existing methods. The last one involves adding extra operations. Adding extra operations is simply adding methods to the derived class.

The term "sub-type" is misleading?

Adding new methods usually enhances the class, yet the enhanced version is called a "sub-type", as if it is inferior in functionality. However, the derived class is not only *enhanced*, but also *specialized*. It is less general than its superclass, which is "sub-".

Should I use "protected"?

If there is no shared data between derived class and base class, e.g. global variables, protected member, static members etc, adding a new method always results in a sub type. Though sometimes you want share some data structure.... Be careful!

First (failed) attempt towards new sub-typing

We now make an attempt of the the other two ways of subtyping. Both relaxing preconditions or tightening postconditions requires modifying existing methods. Since we cannot modify existing functions, our naive attempt is to define a function with the same name as the function being modified, hoping that this function would somehow "replace" the original function.
Consider the following example from your lecture slides. Suppose we have an `IntSet` and `SortedIntSet`.

```
class IntSet {
protected: int count, *data;
public: /* Other methods ommited */
  void insert(int i) {cout << "IntSet\n"; ... } };
class SortedIntSet : public IntSet { public:
  int  max() { /* Get maximum (last) element */ }
  void insert(int i) {cout << "SortedIntSet\n"; ...} };
```

First (failed) attempt of sub-typing, Con't

```
void test1() {
    SortedIntSet set;
    /* A number of insert / del / max */ }
void insert100(IntSet& set) { set.Insert(100); }
void test2() {
    SortedIntSet set; set.insert(10);
    insert100(set);  /* Will print IntSet */
    cout << set.max() << endl; /* very likely be 10 */ }
```

When you run functions like `test1()`, everything will be fine. You cannot break things if you just stick with functions like `test1()`. However, when you do function `insert100()`, a function that assumes sub-typing, you might break your instance's invariance! Breaking the invariance is bad. We know that very well. But we need to look closer. We must ask, how did we fail? And what exactly are the damage?

Static binding causes the problem

Recall how C++ links member function calls?

- All member functions are just usual functions.
- When a member function call happens, the compiler links the function according to the type of the class instance, passing the object as the first argument.

Now in `insert100()`, the method `insert()` is called on object `set`. `set` is an instance of `IntSet`. In this case, the compiler will choose the function `IntSet::Insert()`. Remember that the compiler have no idea what is actually referenced by `set`. When it compiles `insert100`, all it knows is that `set` refer to an object of `IntSet`. It doesn't care if this object is part of a larger object. In fact, up till this point, when you make a function call in the code, the actual function being called is always know at compile time. **The process of binding a function call to the actual definition is *static*.**

Name hiding

On the other hand, you could always:

```
void test3() {
  SortedIntSet set; set.insert(10; /* SortedIntSet */
  IntSet& is = set; set.insert(10); /* IntSet */)}
```

Using a simple reference, you can always access the function you should have replaced. In fact, the `SortedIntSet` actually exposed 2 sets of interfaces:

- `IntSet::insert(int)`, inherited from `IntSet`.
- `SortedIntSet::insert(int)`, which is defines.

Defining a method with same DOES NOT replace the original function. Instead, it only adds a new function! It's just when you say `insert()`, the compiler has two choices. Instead of complaining, it chooses the "closest" match. The second function "hides" the name of the first one. This is called *name hiding*. You can expose the first one with a `using` statement.

Name hiding is not sub-typing (in general)

- `IntSet::insert(int)`, inherited from `IntSet`.
- `SortedIntSet::insert(int)`, which is defines.

We finally examine the damage. How bad is the name hiding?

- First of all, substitution rule still works here. Replacing `IntSet` with `SortedIntSet` will not break any existing code. So `SortedIntSet` is (arguably!) a subtype of `IntSet`.
- We can also think in terms of invariants. All methods exposed by a class must always maintain invariance. In our case `SortedIntSet` assumes a different set of invariance (the ordering), `IntSet::insert` will break the invariance. That's what actually goes wrong.
- On the other hand, in this case `SortedIntSet::insert` will not break the invariance of `IntSet`. However, since two classes are sharing data, this will NOT be the case in general. If the `SortedIntSet::insert` breaks the invariance of

RC Week 10

Virtual-ness and runtime polymorphism

The need for polymorphism

Why are we spending so many time on the name hiding things. Well, the example motivates the following discussions.

- From the subtyping perspective, we want something that allows us to implement other two ways of subtyping.
- As we have shown, doing this often requires actually "replacing" the method call, not just hiding it.
- For functions using that call, let's say `insert100`, the actual function being called would be dependent on what the reference actually is. The fact that identical code behaves differently for different object is called *polymorphism*.
- We will only know that the actual object being referenced at runtime, which means the binding process must be dynamic, i.e. happens also in the runtime. What we want is a form of *dynamic polymorphism*.

All that boils down to the what is known as *virtual* functions.

Apparent type and *actual type*

Before we head out to define what virtual functions are we first start out to make two definitions:

Apparent Type Apparent type is the type annotated by the type system. It is the static type information. It is the you remarked to the compiler.

Actual Type It is the data type of the actual instance. It is the data type that describes what exactly is in the memory.

In our previous example, in function `insert100()`, the apparent type of the variable `set` is `IntSet`, while what's in the memory is actually a `SortedIntSet` (The actual type).
If we note again we have discussed using these new term, normally C++ resolves function bindings based on apparent type. Since apparent types are always known at compile time, function binding will be able to be resolved at compile time.

Dynamic *polymorphism* through *virtual*-ness

What we want is dynamic function binding, the ability to bind a function call based on an object's actual type, instead of the apparent type. This is done through the `virtual` keyword. Using previous example:

```
class IntSet { /* ... */
public:
    virtual void insert(int i) { /* Impl ommited */ }
};
```

The above syntax marks `insert` as a `virtual function (method)`. Virtual methods are methods replaceable by subclasses. When a method call is made, if the method you are calling is a virtual function (based on the apparent type), the language bind the call according to the actual type. In this way, the function `insert100` achieves dynamic polymorphism, the ability to change its behavior based on the actual type of the argument.

Inheritance and keyword `override`

Virtual functions are functions that allows being replaced by subclasses. This is done by the subclass defining a function with identical name as the base class.

```
class SortedIntSet : public IntSet {
public: void insert(int i) { ... } ...  };
```

The act of replacing a function is called *overriding* a base class method. As you can imagine, if you somehow forgot the `virtual` keyword in the base class, overriding would become name hiding, which is subtle but dangerous. C++ provides a keyword to allows make clear your intent:

```
void insert(int i) override { ... }
```

Would 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 a best practice always mark override whenever possible**.

Fix our example with `virtual`

With the changes we have noted, now this is fine

```
void insert100(IntSet& set) { set.Insert(100); } /*same*/
IntSet is; insert100(is);              /* print 'IntSet' */
SortedIntSet sis; sis.insert(10);
insert100(sis); cout << sis.max(); /* SortedIntSet; 100 */
```

Now when `set.Insert()` is called inside `insert100`, the apparent type is still `IntSet`. however, the compiler finds out that `insert` is a virtual method. Thus it generates code that makes the call based on the actual time at runtime.
At runtime, when `IntSet` instance is passed, the actual type is just `IntSet`, so it binds it `IntSet::insert()`. The other situation, the actual type is `SortedIntSet`, so it binds it to `SortedIntSet::insert()`. Now what if modify `insert100` to?

```
void insert100(IntSet /*byVal*/ set) {set.Insert(100);}
```

Inheritance and keyword `final`

On the other hand, *virtualness is automatically inherited*. For instance, if have class and another function:

```
class QuickSortedIntSet : public SortedIntSet { ...
public: void insert(int i) { ... } };
void insert200(SortedIntSet& set) {set.Insert(200);}
```

You DO NOT need to change `SortIntSet` to achieve below:

```
QuickSortedIntSet qsis;
insert100(qss); insert200(qss); /* QuickISIS*/
```

Sometimes, for instance, `SortedIntSet` would like to stop its methods being override by further subclass (Why?). In this case, you can mark its method using the keyword `final`.

```
/* SortedIntSet */ void insert(int) override final {...}
```

The order of two keyword does not matter. Trying to override a method with `final` causes compilation error.

Remark: Use overriding with extra care

The overriding mechanism provides the programmer with a mechanism to change the **base** class behavior. The emphasis is that you not only able to modify your own class, *you would also be able to modify the behavior of another class, with it knowing it!* It sounds dangerous, and you make make careful use of it.
First of all, overriding not always creates subtypes. For instance if we override `IntSet::insert()` with an the following:

```
/* SortedIntSet */ void insert(int) override {numElts--; }
```

This function will very likely break the invariant of `IntSet`. The substitution would fail, and there goes the sub typing relations.

Reuse of implementation and object composition

Secondly, virtual methods are almost the most misused feature of C++! We have noted inheritance has two affects: reuse of implementation and interface. Many would "accidentally" involve the latter, while what they actually want the first!
If you just want the implementation, there are two ways to go:

- Private inheritance, as we have discussed before,
- Or even better, declare the base class object as a member instead of deriving from it. This is called *composition*.

Let's take a look at an example:
Suppose we have a `class Graph` that abstracts the concept of a graph. It supports `Graph::InsertNode(string name)` and `Graph::addEdge(string node1, string node2)`. Suppose we have a class `Binary` tree that inherits `Graph`. It supports one more operation `Tree::GetRoot()`. The question is whether `Tree` is a sub-type of `Graph`?

Remark: Differentiating `IS-A` and `HAS-A`

Unfortunately the answer is negative. The reason should be obvious. There is essentially no requirement on where the edges and nodes are (precondition is weak). But you can't add arbitrary edges and nodes must connect to at most 2 other nodes (stronger precondition). So a `Tree` is not a `Graph` in the subtype sense.

However we would naturally see that a `Tree` type must have a lot common code as the `Graph`. We would like to avoid rewriting them. There are two ways to do this:

- Through `private` inheritance (implementation inheritance). We would not discuss this.
- By setting `Graph` as a member of a `Tree` type. We forward calls to `Graph`. This is a typical case of a `Has-A` implementation.

Compiling technology supporting virtualness

Note we are now in the field of undefined behavior. We are now going to discuss how virtual functions are implemented internally. Knowing this will help you solve exam questions. However, what we are going to discuss here are not ground truth. I.e. things could change from platform to platform.
Internally, virtual functions are implemented using VTables:

- Virtual methods are grouped together in an array of function pointers, as if they are member variables. This array is called *Virtual Table*, or *vtable*
- When the compiler makes a virtual method call, it calls the function pointer in the corresponding entry of the vtable.
- When derived objects instantiates itself, it replaces entries of function it need to override in the vtable, with its own implementation.

Virtual Tables and performance: No free lunch

Virtual function comes with preformance hit

- The cost of one extra layer of indirectness. There exists one more pointer dereference to find the target function. That's one more memory access.
- Cost of unknown call target. Modern processors will "prefetch", or guess the future instructions and execute them in advance. Since the function call target is unknown, this will not be possible for virtual functions
- Cost of unable to inline methods. For simple methods, compiler will try to inline them. Since the binding happens at runtime for virtual methods, this is no longer possible.

Using the `final` keyword will help. If the compiler is able to determine the actual type, it may choose to preform *de-virtualization*. Those costs could be quite huge if the method is used frequently. In old time the cost is often indurable. Modern computers are more powerful, things get better.

Using VTable to solve problem

```
struct Foo {
    virtual void f() = 0;
    virtual void g() = 0;
};
struct Bar : public Foo {
    void f() {};
    void g() {};
    void h() {};
};
struct Baz : public Bar {
    void    f() {};
    void    g() {};
    virtual void h() {}; };
struct Qux : public Baz {
    void f() {};
    void h() {}; };
```

```
Bar bar; bar.g(); // Bar::g
Qux qux; qux.g(); // Baz::g
Baz baz; baz.h(); // Baz::h
Foo& f1 = qux; f1.g();
// Baz::g
Bar& b1 = qux; b1.h();
// Bar::h
Baz& b2 = qux; b2.h();
// Qux::h
Bar* bar = &Quz; bar->h();
// ?
const Foo& cbar = Baz;
cbar.g(); // ?
```

Using VTable to solve problem

The idea will be to draw out the VTable for each object. E.g. When you run `b1.h()`, you would find that the apparent type of `b1` is `Bar`. And `Bar` does not have an entry for `h()`. So this is static linkage. On the other hand, if you run `b2.h()`, `b2` is `Baz`. We thus follow the vtable to find the function call.

Vtable for `Qux` instance:

```
struct Qux {
 struct Baz {
  struct Bar {
   struct Foo {
     vtbl f() = Qux::f;
     vtbl g() = Baz::g;
   }
  }
  vtbl h() = Qux::h
 }
}
```

Pure virtual functions and classes

It is possible that we do not supply a implementation when defining a base class. In this case, the corresponding entry in the vtable would simply be left unfilled:

```
/* 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 a *pure virtual class*. You only need to have one pure virtual function for a class to be "purely virtual".

You **CANNOT** instantiate a *pure virtual class*. This is somewhat obvious. If you ever have an instance of `IntSet`, what should happen when you call `IntSet::insert()`?

Pure virtual class are also called *abstract base classes*, or *interfaces*. We will explain this terminology shortly after. It is often that the name abstract base classes starts with a case letter `I` (/ai/) for *interface*.

Abstract base classes

Why would we ever wanted a pure virtual classes since we cannot instantiate it? Because we would want to model abstract *concepts*. We would like to say things like *Matricies are subclass of summable object*. We would like to increase code reuse. Consider the following class:

```
class ISummable { public:
    /* Add item x to itself */
    virtual void add(ISummable& x) = 0; };
```

This class models the objects that are summable. Based on this modeling, we could write the following very general function:

```
void sum(ISummable elem[],size_t size, ISummable& rst) {
    for (int i = 0; i < size; ++i) rst.add(elem[i]); }
```

This will work for anything that is `Summable` object. When a class derives from an interface and provides an implementation, we say it `implements` the interface.

Programming with interfaces

Abstract base classes provides very strong decoupling technique, in a large project:

- One team of engineers write algorithms using the interface.
- Another, independent team of engineer write classes that implements the interface.
- The only coupling point is the interface itself. Once the interface is agreed upon, two teams can work independently.

This is even better if you ever would like to update your code:

- To support new features, simply write **more** classes. There is no need change any existing code.
- To update existing feature, only rewrite and recompile the part related. I.e. you only recompile the algorithm, or the classes implementing the interface.
- Dynamic linking technology, you could update your software by shipping the end user only **part** of the entire binary.

Programming with interfaces

Modern software engineering technologies usually "abuses" the Liskov Substitution rules by write software that depends only on interfaces. In this way maximum utilization is achieve. For instance, we now provide an example of what is known as *"Factory" design pattern*. Let's say you are asked to write a encryption program that intends to support a number of ciphers. You first abstract out what a cipher is:

```
class EncryptionEngine {
public:
  virtual string name() = 0;
  virtual string generateKey() = 0;
  virtual string encrypt(...) = 0;
  virtual string decrypt(...) = 0;
  virtual ~EncryptionEngine() = default;
};
```

Programming with interfaces, II

Then you write a few engines, and write a *factory method*:

```
EncryptionEngine *
encryptionEngineFactory(std::string engine) {
  engine =string_tolower(engine);
  if (engine == "aes") return new AesEngine;
  if (engine == "rc6") return new Rc6Engine;
  if (engine == "dummy") return new DummyEngine;
  throw UnknownEngine(engine);
}
```

In the main, use a unique_ptr to preform memory management.

```
string algorithm = getEngineName();
unique_ptr<EncryptionEngine> engine(
  encryptionEngineFactory(algorithm)
); cout << engine->enrypt("abc", "DEADBEEF");
```

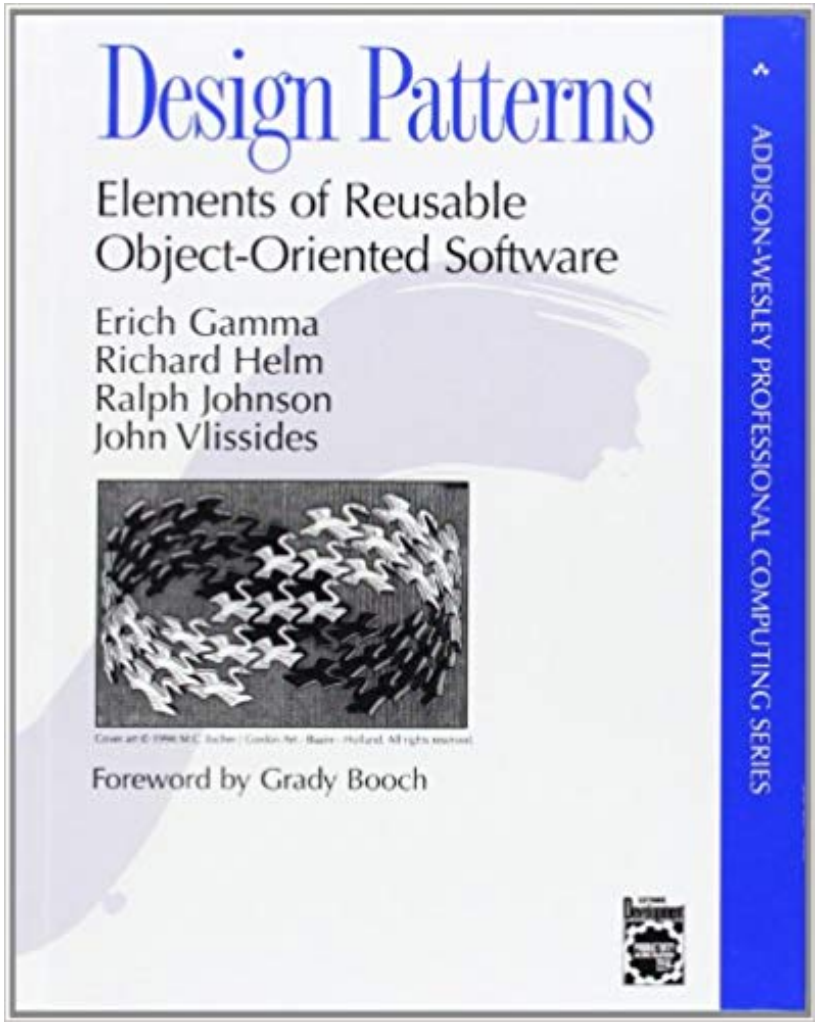
Programming with interfaces, III

This method gives you a number of advantages:

- First of all, it allows you to decouple the encryption algorithm from the code that handles you user input. The user input may come from GUI, input or a file. We simply don't care. The main routine would be exactly the same
- Then, you no longer need to duplicate any code to support more algorithms. This prevents possible mistakes.
- Finally, this gives you the flexibility of dynamically add, remove, disable, enable some algorithms. You could complicate your factory methods, for instance, check if your user have paid for your program. The list of available engines could come from Internet, or from a configuration file.

Remark on *design patterns*

There exists a number (over 20) of different (object oriented) design patterns. The urge for design pattern rise with the popularizing of JAVA. Almost all of them relies on interfaces. There exists a book written by GOF called *Design Patterns* listing all of them. The book is considered one of the most classical text book for software engineering.



Is it a subtype?

Designing whether something is or is not a subtype of a particular super type is probably in the heart of program structure design.

A Remark

The problem of whether two objects form a "IS-A" relation must be understood in a realistic context. Consider two classes `RegularIntSet` and `SortedIntSet`. Suppose both of them supports a `max()` function.

- You can argue that `RegularIntSet` is a sub-type of `SortedIntSet` because they support the same set of operations.
- You can also claim that `RegularIntSet` is NOT a sub-type since the `max()` of `RegularIntSet` is significantly slower than it's counterpart. One could argue the performance is part of the abstraction.

RC Week 11

Generics, polymorphism and templated containers

Preliminary: A container of pointer

As a kind reminder we briefly discuss containers of pointers here.
The key idea in the whole discussion again is the problem of the ownership.
One must be crystal clear that a container of pointer actually owns the object, which means:

- Pass to a container only things you have ownership.
- When you pop from a container, you must take over ownership.
- When an object has been passed to the container, it's the container's responsibility and authority to treat the object. You should not use the object in any way since you no longer owns the object.

With this in mind we take a look a few slides:

Templated Container of Pointers

```
template <class T>
class List {
  public:
    ...
    void insert(T v);
    T    remove();
  private:
    struct node {
        node *next;
        T     o;
    };
    ....
};
```

```
template <class T>
class List {
  public:
    ...
    void insert(T *v);
    T    *remove();
  private:
    struct node {
        node *next;
        T    *o;
    };
    ....
};
```

Container of Pointers

Use

- To avoid the bugs related to container of pointers, one usual "pattern" of using container of pointers has an **invariant**, plus three **rules** of use:
  - **At-most-once invariant**: any object can be linked to at most one container at any time through pointer.
  1. **Existence**: An object must be **dynamically allocated** before a pointer to it is inserted.
  2. **Ownership**: Once a pointer to an object is inserted, that object becomes the property of the container. No one else may use or modify it in any way.
  3. **Conservation**: When a pointer is removed from a container, either the pointer must be inserted into **some** container, or its referent must be **deleted**.

Do not repeat your self

Containers are objects to contain other objects, they do not have an intrinsic meaning. From now on we focus on Containter of pointers (though we use a container of value in this example). Consider the example of a container of `char` and `int`:

```
struct node {
    node *next;
    int   v;
};

class IntList {
    node *first;
  public:
    void insert(int v);
    int remove();
    ...
};
```

```
struct node {
    node *next;
    char  v;
};

class CharList {
    node *first;
  public:
    void insert(char v);
    char remove();
    ...
};
```

Generics and polymorphism

We would imagine they share the same implementation. Now our DRY principal suggest we should find a way to abstract out the almost identical implementation.
Now we must familiar ourself to two widely used terms:

Generics A generic algorithm is an algorithm that does not depend on a specific type. Type can be specified later.

Polymorphism Polymorphism is a property of code. A piece of code is said to be polymorphic, if it works on "a range of" types.

Clearly polymorphism is a technique to achieve generics. What we need to implement here is a generic algorithm (data structure), and we are going to use polymorphic code to achieve our purpose.

Polymorphism tool box

We first start by examining our polymorphism toolbox. In general there are 3 ways achieve (different levels) of polymorphism. Polymorphic code always involve a time where the "real" implementation starts to step in. In general there are two timings:

Compile-time The compiler "injects" the real implementation into our polymorphic code when it is being compile. Sometimes called *static polymorphism*.

Runtime The polymorphic behavior is determined when it is needed at runtime. A typical example is `virtual` functions. It introduces overhead.

We now examine our choices:

Overloading Function / operator overloading. Limited power.

Subtyping Virtual functions. Polymorphism through dynamic dispatch. This is runtime polymorphism.

Parametric Templates in C++. Type as a parameter.

Polymorphic containers

Both of the second and third solution can be used to solve our problem at hand. We first look at the second one. We now introduces `Polymorphic containers`.
In a polymorphic container, the basic idea is to create a universal super type. Every type it it's subtype. And a container is written in terms of the universal super type.
The universal super type looks like the following:

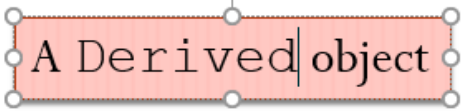
```
struct Object {
^^Ivirtual ~Object() {}
^^Ivirtual Object* clone() = 0;
}
```

Every class to be pushed into the container should be a subtype of this class. There are two questions: why are both methods virtual? Why do we need the `clone()` method.

Polymorphic containers

Note when we copy construct the constructor we also use copy copy constructor to copy the contained objects.

```
void List::copyList(node *list) {
  if (list != NULL) {
    Object *o;
    copyList(list->next);
    o = new Object(*list->value);
    insert(o);
  }
}
```



But unfortunately this simply cannot work. Remember that constructors cannot be virtual, since it is not possible for a base class object to setup invariants for a derived class object.
Note above code does compile (why?). However what we get from such construction is an empty, useless `Object` class instance.

Polymorphic containers: copying

The solution is to ask the derived class instance to make a copy itself. This gives us the `clone()` method in the previous slides. This methods asks the object to make a copy of itself and returns a ptr to base class. Now our copy method can be written as follows:

```
void List::copyList(node *list){
    if (list != NULL) {
        Object *o; copyList(list->next);
        o = list->value->clone(); insert(o); } }
```

Note this design makes sense from an interface point of view. In order for an object to be put inside a container it must be copyable. For normal container this is checked by the compile for the copy constructor (since the compiler knows the type in question). For polymorphic container since the compiler do not know what type is contained in advance so we must specify the constraint by ourself, namely through adding a `clone()` method.

Type Erasure

We would like to make a final note. Consider when you pop something out of the container. The container only knows that the value is of type `Object*`. But you know it is actually an instance of `Derived*`. And most likely you need to use it as a `Derived` object. You need to transform a base class pointer to a derived class pointer.

This way you will need a cast:

```
Derived* bp = dynamic_cast<Derived *>(list.remove());
```

The `dynamic_cast<>` checks at runtime if the pointer of base class is actually a pointer of derived class. If not so it returns `nullptr`.

Type Erasure

Note all this must happen at run-time. No checks will be done at compile time. This has both up-side and down-side:

- Since actual type is involved only when used this allows for heterogeneous containers. The contained object does not have to of the same type. This is a huge gain.
- However the cost is also significant! Since all checks are done at runtime, there is not type check at compile time either. Type mismatch will be deferred to runtime. This makes writing type safe code much harder!

Since essentially we are "erasing" the actual type information of the objects when we put them into the container, this strategy is often called *type erasure*.

Problematic as it seems, `JAVA` choses to use type erasure when it comes to generics. This is one of the debatable feature of the language.

Poor man's template C++

Now we turn to `template`, which is so called *parametric polymorphism*. Before we introduce that in C++, I would like to familiarize you with a piece of code in C++:

```
#define SWAP(type) swap_##type
#define SWAP_IMPL(type) \
static void SWAP(type) (type &x, type &y) {\
    type t = x; x = y; y = x; \
}
SWAP_IMPL(int);
SWAP_IMPL(double);
int main() {
    int x =10, y =20;          SWAP(int)(x, y);
    double p = 0.0, q = 1.0; SWAP(double)(p, q);
}
```

Poor man's template in C++

The code seems very cryptic at first. We now look at what it does by expanding the macros. `SWAP(int)` would be expanded into `swap_int`. The two sharp signs means concatenating. An `SWAP_IMPL(swap)` will be expanded into.

```
static void swap_int (int &x, int &y) {
    int t = x; x = y; y = x;
}
```

This is a very straight forward swapping two integers. As you can see this way we can reuse the `swap` function easily for different types by simpling use `SWAP_IMPL` to create an implementation for different types and call them with `SWAP()`.
Further more we can put the two macro definitions into a header file and include the when we need to.

Templates

A template in C++ does exactly the same (for now. It if far more complicated if you explore deeper). When you write a template, for example:

```
template <class T>
class List {
  public:
    ...
    void insert(T *v);
    T    *remove();
  private:
    struct node {
        node *next;
        T    *o;
    };
    ....
};
```

- `T` is called a template parameter. It is usually a `class`, but it can be a value (e.g. an `int`).
- The traditional way of specifying the type parameter is through `template<class T>`. But after C++11 one could use `template<typename T>`.
- A template is much like a macro. When it comes to using it, the compiler uses an actual type to replace the type argument, and generate a version of implement for that type.

Template instantiation

When you use a templated class, for example:

```
List<int> intList;
```

The compiler will automatically produce a version of the actual code with the `int` as the parameter. This process is called *template instantiation*. Template instantiation is conceptually the compiler writing `SWAP_IMPL(int)` for us.
We would like to make a footnote here. A template is by no means a type. A template is not a type since you cannot have a variable of `List` type clearly. It can be `List<int>` or `List<double>`, but simply is not `List`. A template is incomplete. It becomes a type when you put an actual type argument into it.
In some literature templates are refered as a *dependent type*. You should notice this behavior is very much like a *function*, which takes in a type and spits out another type. This function is "sort of" evaluated in compile type.

A comment on the syntax

Templates and their implementation is almost always written in a header file (I would say always if not for the project). Libraries consists only of template classes are often called header libraries. The reason behind this should be simple from the previous poor-man's template example.
There are in general two ways of implementing a templated class method.

- You can implement it inside the declaration.
- You can implement separately.

For the second method:

```
template <class T> void List<T>::isEmpty() {...}
```

The beginning `templated <class T>` must be there. Note the method is implemented within the namespace `List<T>`. This makes sense since namespaces corresponds to types, and `List<T>` is a type, `List` is not.

A comment on the syntax

We would like to make a comment on the following slide (ch21,30):

- The function header of the constructor is

```
List<T>::List()
List<T>::List(const List &l)
```

Must have <T>! No <T>! No <T>!

- The function header of the destructor is

```
List<T>::~List()
```

Must have <T>! No <T>!

- The function header of the assignment operator is

```
List<T> &List<T>::operator=(const List &l)
```

Must have <T>! No <T>!

30

A comment on the syntax

We would like to make a comment on the following slide (ch21,30):

- The function header of the constructor is

```
List<T>::List()
List<T>::List(const List &l)
```

Must have <T>! No <T>! No <T>!

- The function header of the destructor is

```
List<T>::~List()
```

Must have <T>! No <T>!

- The function header of the assignment operator is

```
List<T> &List<T>::operator=(const List &l)
```

Must have <T>! No <T>!

30

A comment on the syntax

You should find this to be very weired. Take a look at

```
List<T>& List<T>::operator=(const List &l);
```

Now what is the type of `l`? We have commented that `List` does not name a type. But clearly `l` is function argument and it must have a type. The professor commented "No <T>!".
After a little research this is called a injected-class-name. Basically if you write `List` is a templated class, it will be inferred as `List<T>`, so we could use `List` as a shorthand of `List<T>` inside class declaration. We would like to note:

- It is syntactically correct to write `List<T>`.
- It is also the recommended way as long as it does not impact readability too much, since it enhances clarity and makes more sense.

Note the name of the ctor/dtor must be `List`, not `List<T>`.

A comment on the syntax

We would like to emphasize on one thing, consider the following code

```
// Create a static list of integers
List<int> li;
// Create a dynamic list of integers
List<int> *lip = new List<int>;
// Create a dynamic list of doubles.
List<double> *ldp = new List<double>;
```

Essentially forms `List<int>` are no difference to a regular `int`, `IntSet` etc. Anywhere a type can be use, such form can be use. For example it is possible to inherit from a templated type:

```
class Foo : public: List<int> {...}
```

And then override some of its methods.

Compile time polymorphism trade-offs

Now we are at the core of compile time polymorphism. Templates are instantiated at compile time. The instantiation process is essentially the compiler deciding when to plug in the actual implementation. This gives us the following trade-offs:

- It enhance code safety, by a lot. After substituting the type parameter with the actual type the compiler will be able to perform type checks. Type mismatches will be caught so that code correctness is improved.
- It improves performance (compared to polymorphic containers). Since the dispatch is done at compile time there is no run-time overhead. Compile-time dispatch also allows for further (much more) optimizations.

Above two are the major reasons people use templates. This is also why standard libraries choose to use templates to implement generic containers, since C++ is a language that focuses on performance and static typing.

Compile time polymorphism trade-offs

Now we take a look at the down sides

- It prolongs compile time (a lot). The template system of C++ is so powerful that it self is Turing complete: meaning it is possible to write algorithms in terms of templates, or infinite loops that crashes the compiler.
- It increases the executable size. Each template instantiation creates more code. This gets worse combined with the C++'s "one file per object" rule, as we will see next.

The first point is not really a pure down side. Some people heavily exploit the first point and ends up creating a whole new area of programming, called *meta-programming*.
The second down side is generally OK for most desktop applications since people have large memories. But for embedded applications where memories are scares, this basically rules STLs out from the embedded use.
Another problem with templates is it complicates the compiler by a lot. However it is not a bad news for you.

Operator overloading

We now fill in the final piece of puzzle, i.e. operator overloading. We have introduced this in the previous slides. See page 253/254. Again just as a reminder there are two ways to write an overloaded operator:

- As a class method of the first operand (if there are multiple operands)
- Write as a normal method.

The second approach is very common for overloading the extraction / push operator on a I/O stream (pay attention to the return type of the function). For example:

```
ostream& operator<< (ostream &s, const MyClass &r) {...}
```

allows you to print to `cout` and `fstream` (why?) by:

```
Ofstream file(...); MyClass obj(...);
file << obj; cout << obj;
```

`friend` keyword

In our previous example `operator<<` might need to access private member of `MyClass` instance. You could provide an accessing operator for each of the member, but often it is not a good idea (why?).
One workaround is specifically grant `operator<<(ostream &s, const MyClass &r)` access to the protected members. This can be done by using the `friend` keyword:

```
class MyClass {
    friend ostream& operator<< (ostream &s, const MyClass &r);}
```

It doesn't matter where this is marked `public` or `private`. Note `friend` can also grant access to regular functions and even classes:

```
class MyClass {
    friend class Bar; friend int foo(double foo);}
```

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

RC Week 12

Elementary data structures

Data structures 101

There are always two interleaved concepts when we talk about data structures:

Abstraction This is how they are used, for example, as a queue, or as a stack etc. This includes design choices of supporting random access, supporting enumeration etc.

Implementation This focuses on how the abstraction is implemented. For example a stack can be implemented by a array (if you dynamically resize it), or by a linked list, or by a doubly linked list, or by a hash map or a binary tree.

It is always that you first choose your abstraction base on what you need. Then you decide on the implementation. Of course there sometimes exists sort of "default" implementation, which is reasonably fast on most (if not all) operations. We will learn about them, but remember the that a `stack` is not always a linked list, but it is most likely to be.

Measuring performance

We need a way of measuring how fast an operation is. We do this by examining *time complexity* of an operation. We will study this in VE281, now we just give you a preliminary idea. Essentially we assume there is n element in the container, and we measure how much time it takes to perform such operation once.

Super Fast	Means operation takes $O(1)$ (constant time) no matter how many elements in the container.
Fast	$O(\log n)$. The base of the logarithm is irrelavent.
Reasonable	$O(n \log n)$ or $O(n)$. Acceptable for large dataset.
It depends	$O(n^2)$, (n^3). Depends on scale.
Emmmm	Any polynomial with order larger than 3
Exponential	$O(b^n)$ with arbitrary base. Usual operations taking exponential time is not acceptable unless no other methods are available.
NO!NO!NO!	Super exponential algorithms. For example $O(n!)$.

A stack

A stack is a sequential data structure that:

- (Usually super fast) insert and remove at front.
- Objects first pushed onto the stack comes out last. It is often called First-In-Last-Out. Thus it is often referred as a FILO.

It does not (need to) support

- Any form of random access. i.e. no insert / remove even read in the middle.
- Enumeration in any order
- Fast querying of any sort.

A stack describes an abstraction. Essentially any implementation that guarantees the above two property (abstraction) can be used to implement a stack.

A queue

A stack is a sequential data structure that:

- (Usually super fast) insert at front.
- (Usually super fast) remove at the end.
- Objects first pushed onto the stack comes out first. It is often called First-In-First-Out. Thus it is often referred as a FILO.

It does not (need to) support

- Any form of random access. i.e. no insert / remove even read in the middle.
- Enumeration in any order
- Fast querying of any sort.
- Remove at the front and insert at the end.

A queue describes an abstraction. Essentially any implementation that guarantees the above three property (abstraction) can be used to implement a queue.

A deque

deque stands for *double ended queue*. A deque is a sequential data structure that:

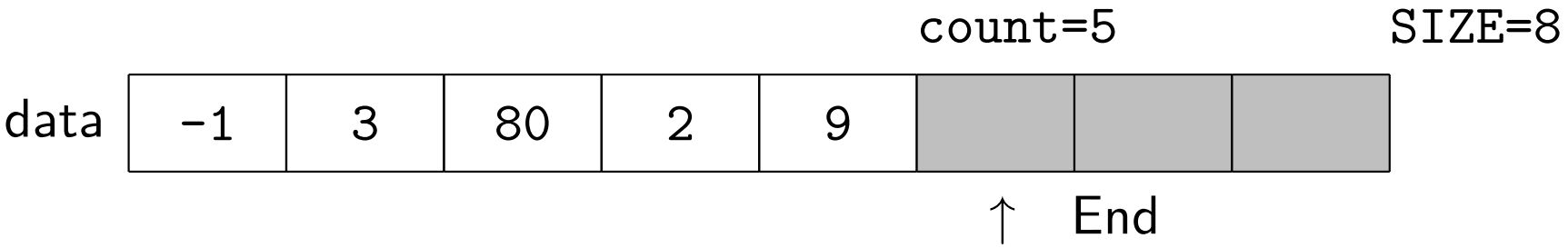
- (Usually super fast) insert / remove at front.
- (Usually super fast) insert / remove at the end.
- As it's name suggests it looks like a double ended queue. It's hard to rigorously (clear in the sense of mathematics) define what a deque does.

It does not (need to) support

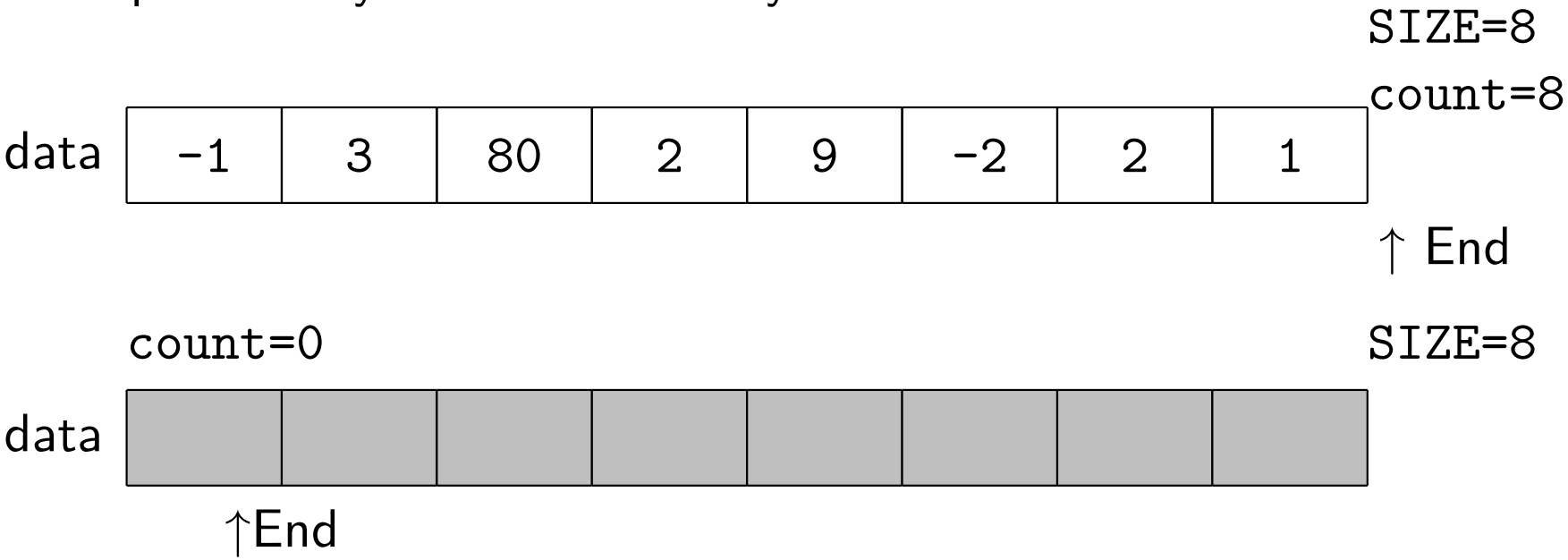
- Any form of random access. i.e. no insert / remove even read in the middle.
- Enumeration in any order
- Fast querying of any sort.

Implementation by Linear List

A *linear list* is simply a fancy name for `array`.



Note particularly the two boundary cases:



Left close right open convention

One convention that is used throughout any sequentially iterable data structure is the *Left-close-right-open*. Remember this is more than a direct array. The rule is essentially the following:

- The end pointer (or anything equivalent) points to one-pass-the-end location.
- The front pointer points to the first element (if there is one).

This rule is so important that the standard is willing to put a syntax hole just for this convention. The standard says:

If you have an array defined in the following manner:

```
int x[10] = {0};
```

- `&(x[11]) == x + 11`, `&(x[11]) - x`, `&(x[11]) > x` : are all undefined behavior.
- `&(x[10]) == x + 10`, `&(x[10]) - x == 10`, , `&(x[10]) > x` are guaranteed to hold.

Left close right open convention

Following the convention is very beneficial for us:

- `end` always puts to the next place to add new elements.
- Thus `data[size]` is always the next place to add new elements.
- `end - data == size` always hold.

These property are very helpful in keeping track invariants within a class. They are especially very helpful in writing loops

```
for (int i = 0; i < size; i++) /* Use data[i] */;
```

Or in terms of pointer

```
for (int* p = front; p < front + count; p++) /* Use *p */;
```

Or more generally, with `c` being an instance of `Container`:

```
for (ptr p = c.begin(); p != c.end(); p = p.next() ) ... ;
```

Performance of linear list

We assume a linear list with one end pointer. Here pointer refers to "logical pointer", meaning any information that allows for obtaining the one-pass-the-end location in constant time. Note in the tables *random access* refers to accessing through index.

Operation	Performance	Note
Insert at front	Super fast	Neglecting stretching
Deletion at front	Super fast+	Update `size`
Insert at end	Reasonable	Move all elements
Deletion at end	Reasonable	Move all elements
Insert at middle	Reasonable	Move following elements
Deletion at middle	Reasonable	Move following elements
Random access	Super fast	Arrays are stored continuously
Forward iteration	Easy	iterate by index
Backward iteration	Easy	iterate by index
Memory consumption	Least	(Arguably, unused space)

Performance of Singly Linked list

We assume a singly linked list with both front and end pointer.

Operation	Performance	Note
Insert at front	Super fast	allocate a node
Deletion at front	Super fast	delete a node
Insert at end	Super fast	allocate a node
Deletion at end	Reasonable*	Why?
Insert at middle	Super fast	play around pointers
Insert at middle	Super fast	play around pointers
Random access	Reasonable*	Need to follow the links
Forward iteration	Easy	Use the link, Luke.
Backward iteration	Slow	Why?
Memory consumption	A little	1 extra ptrs / elem

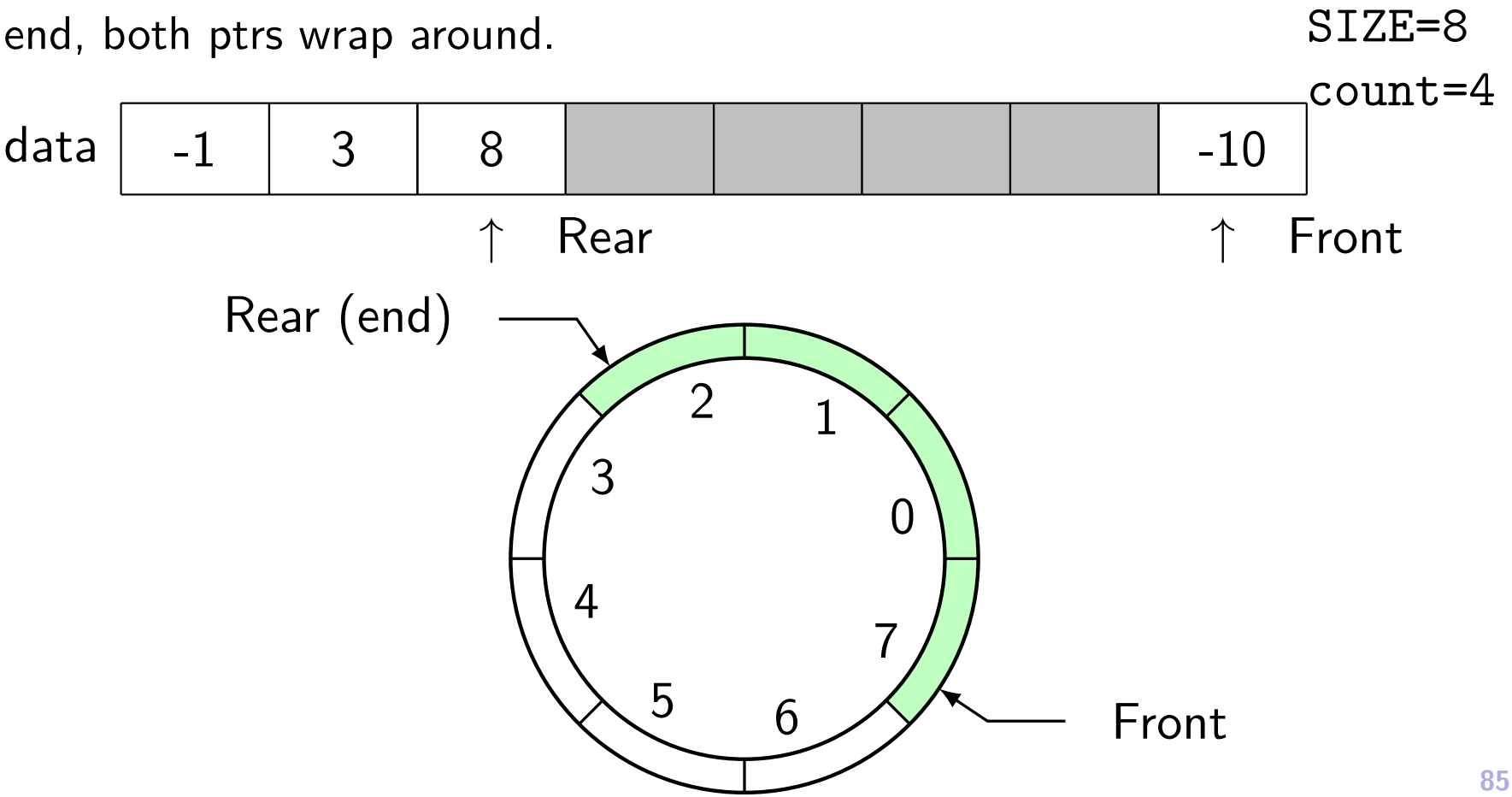
Performance of Doubly Linked list

We assume a doubly linked list with both front and end pointer.

Operation	Performance	Note
Insert at front	Super fast	allocate a node
Deletion at front	Super fast	delete a node
Insert at end	Super fast	allocate a node
Deletion at end	Super fast	Compare with previous
Insert at middle	Super fast	play around pointers
Insert at middle	Super fast	play around pointers
Random access	Reasonable*	Could be improved.
Forward iteration	Easy	Use the link, Luke.
Backward iteration	Easy	Thanks to the reverse link.
Memory consumption	A little*	2 extra ptrs / elem

Implementation of Circular Array

A circular array is sequential data structure. It is based on an array. It uses both front and rear ptrs to indicate begin and end. When a new element is added into the array, front moves to begin and rear moves to end, both ptrs wrap around.



Performance of Circular Array

We assume a circular array with **fixed** size and both front and rear pointer.

Operation	Performance	Note
Insert at front	Super fast	
Deletion at front	Super fast	Move front ptr
Insert at end	Super fast	
Deletion at end	Super fast	Move rear ptr
Insert at middle	Reasonable	How?
Deletion at middle	Reasonable	How?
Random access	Super fast	How?
Forward iteration	Easy	
Backward iteration	Easy	
Memory consumption	Least	Unused space

RC Week 12

Standard Template Library

History of the standard template library

In November 1993 Alexander Stepanov presented a library based on generic programming to the ANSI/ISO committee for C++ standardization. The committee's response was overwhelmingly favorable and led to a request from Andrew Koenig for a formal proposal in time for the March 1994 meeting.The Stepanov and Lee document 17 was incorporated into the ANSI/ISO C++ draft standard.

The prospects for early widespread dissemination of STL were considerably improved with Hewlett-Packard's decision to make its implementation freely available on the Internet in August 1994. This implementation, developed by Stepanov, Lee, and Musser during the standardization process, became the basis of many implementations offered by compiler and library vendors today.

std::vector

std::list

std::map

std::unordered_map