Design patterns II. (Bridge, Iterator, Factory)

Set and its iteration

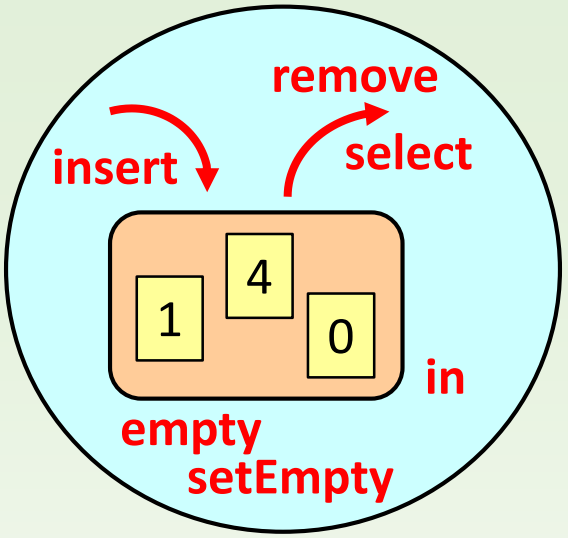
Class-templates

1st task

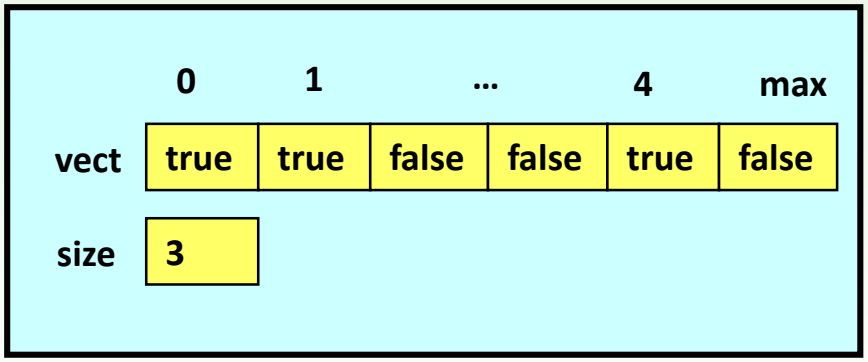
Write a program to create sets of integers.

- Representation of a set depends on if there is an upper bound (max) for its items or not.
  - Usually, items are stored in a sequence.
  - Specifically, a the set os represented by a boolean array, where a number is in the set if the numberth item of the set is *true*.
- It is worthy to hide the representation from the user: when a set is created, the user decides if there is an upper bound for the items, or not, but the program does not say anything about the different representations (the user does not need to know them).

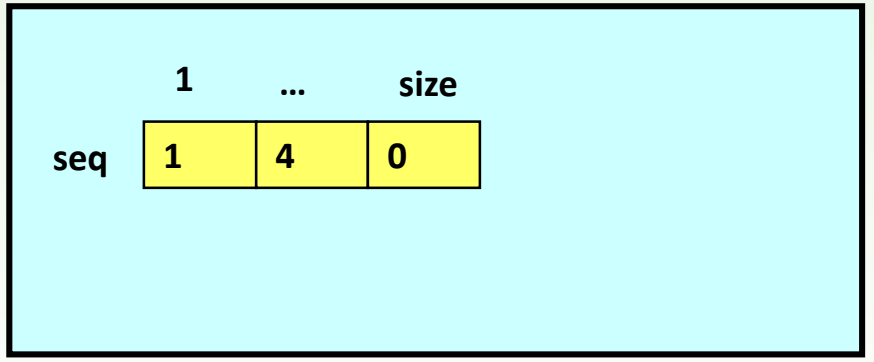
Two representations



Array



Sequence



1. Fixed-size array and the number of the stored items.

2. The computational time of the operations is mainly constant, except select and setEmpty (they are linear).

1. Dynamic sequence.

2. The computational time of the operations is mainly linear, except select and setEmpty (they are constant).

Array representation

set([0..max])

Type-specification

type values

Sets containing natural numbers between 0 and max.

(Formally: power set $\mathcal{P}\{0..max\}$)

operations

setEmpty
h:=∅ h:set([0..max])

insert
h:=h∪{e} h:set([0..max]), e:ℕ

remove
h:=h−{e} h:set([0..max]), e:ℕ

select
e:=mem(h) h:set([0..max]), e:ℕ

empty
l:= h=∅ h:set([0..max]), l:𝕃

in
l:= e∈h h:set([0..max]), e:ℕ, l:𝕃

representation

vect : $\mathbb{L}^{0..max}$
size : ℕ

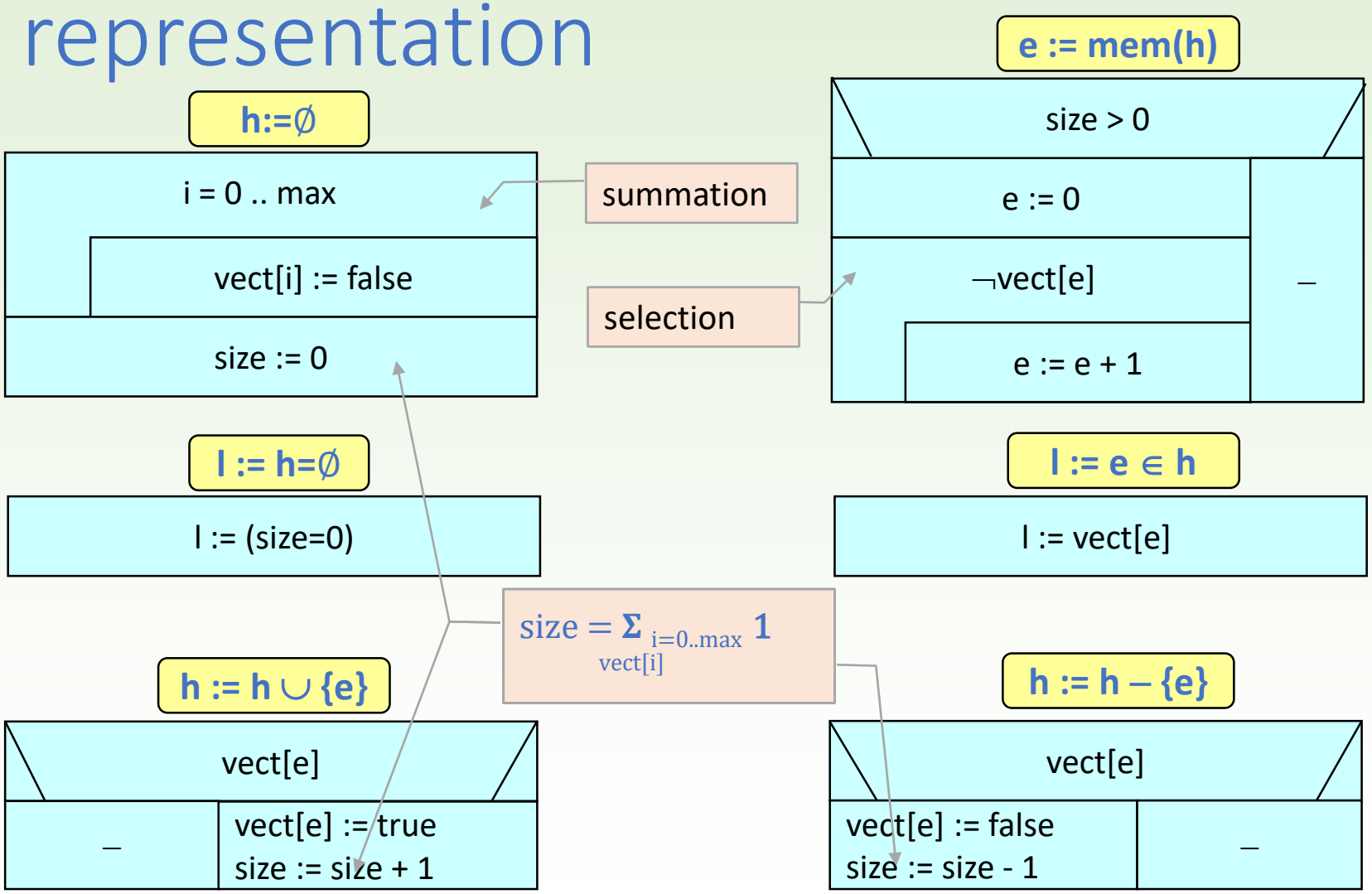
invariant: $size = \Sigma_{i=0..max}\ 1_{vect[i]}$

implementation

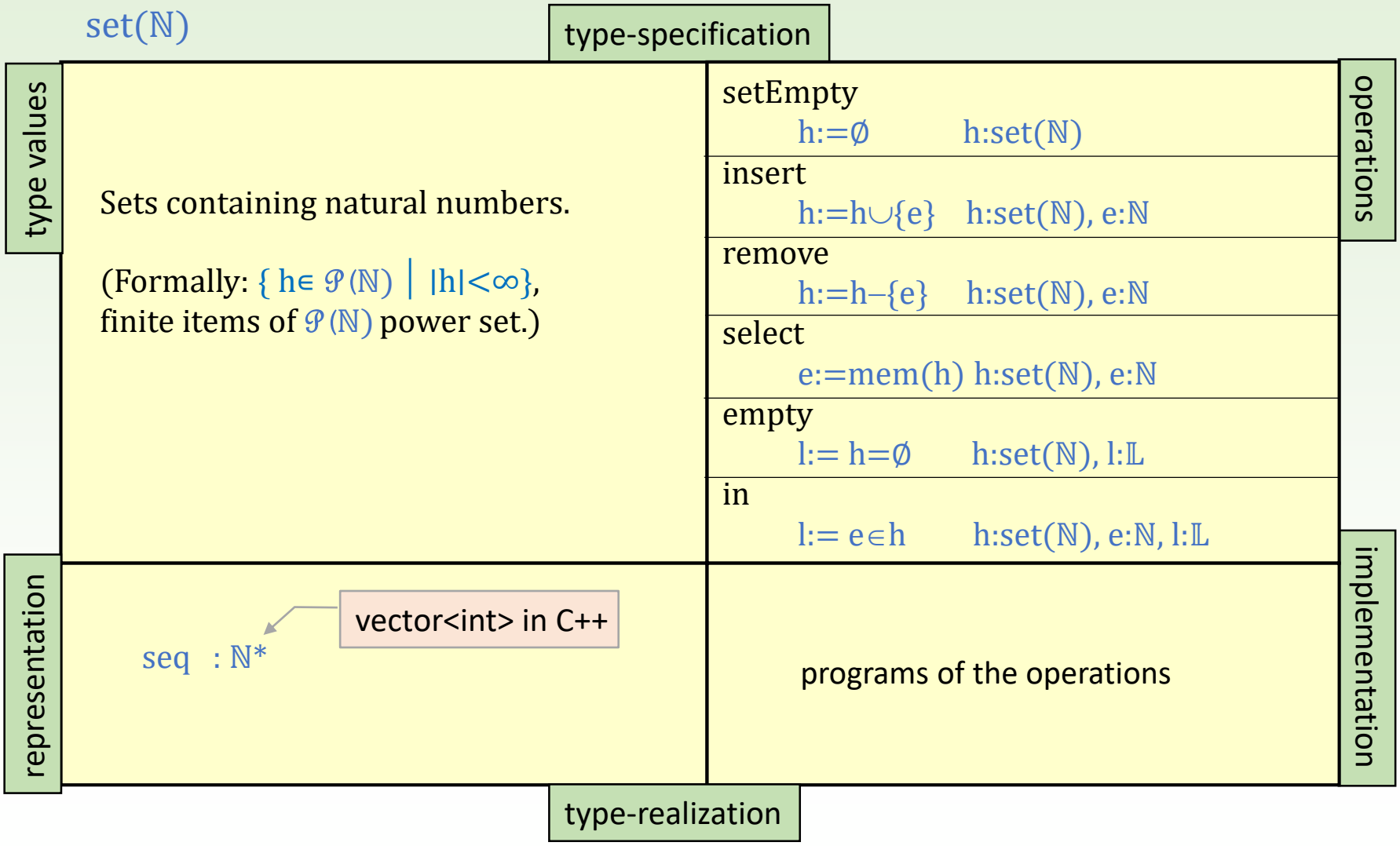
programs of the operations

type-realization

Operations in array representation



Sequence representation



Operations in sequence representation

h:=∅

seq := <>

e := mem(h)

seq.size() > 0

e := seq[1]	–

l := h=∅

l := seq.size() = 0

l := e ∈ h

l, ind:=search $_{i=1..seq.size()}$(seq[i] = e)

h := h ∪ {e}

linear search

l, ind:=search $_{i=1..seq.size()}$(seq[i]=e)

l

–	seq := seq ⊕ <e>

h := h – {e}

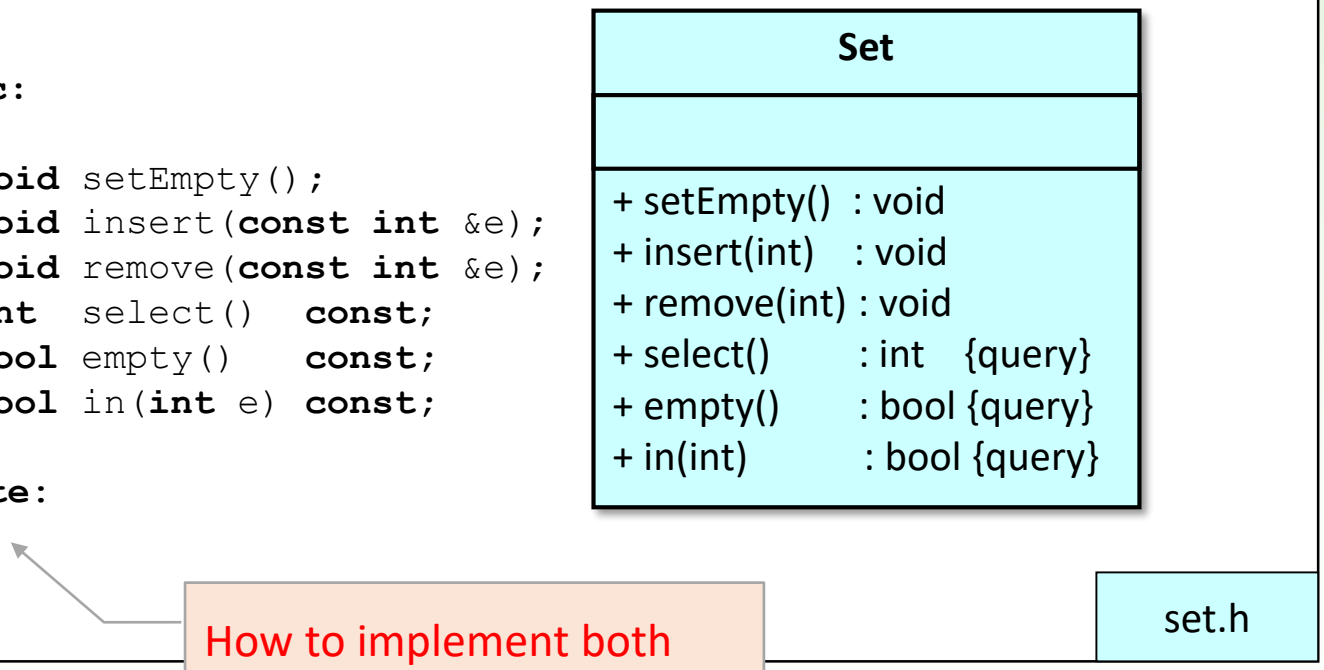
l, ind:=search $_{i=1..seq.size()}$(seq[i]=e)

l

seq[ind] := seq[seq.size()] seq := seq ⊖ seq[seq.size()]	–

Public part of class Set

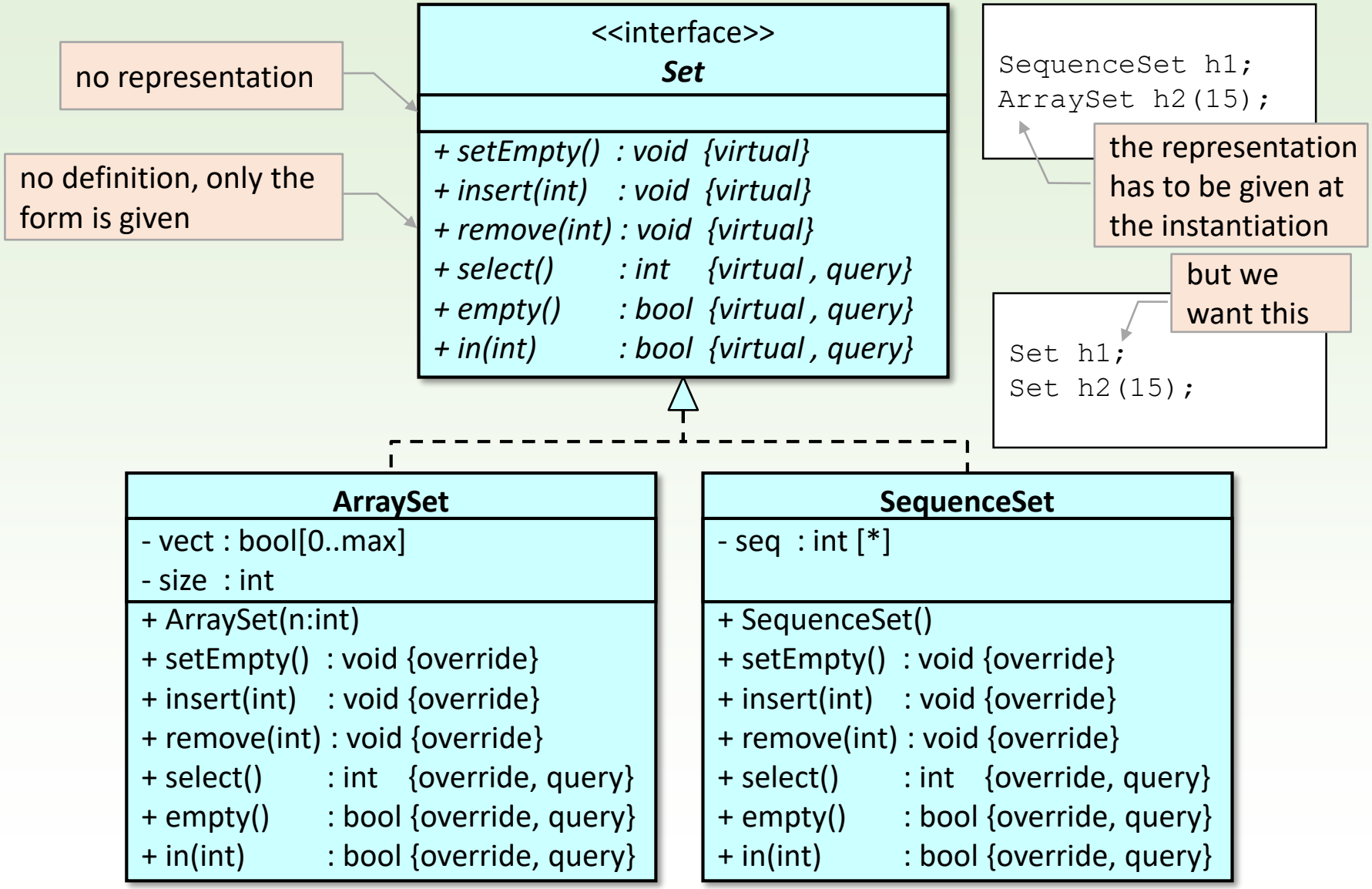
```
class Set
{
    public:

        void setEmpty();
        void insert(const int &e);
        void remove(const int &e);
        int  select()  const;
        bool empty()   const;
        bool in(int e) const;

    private:
        …
};
```



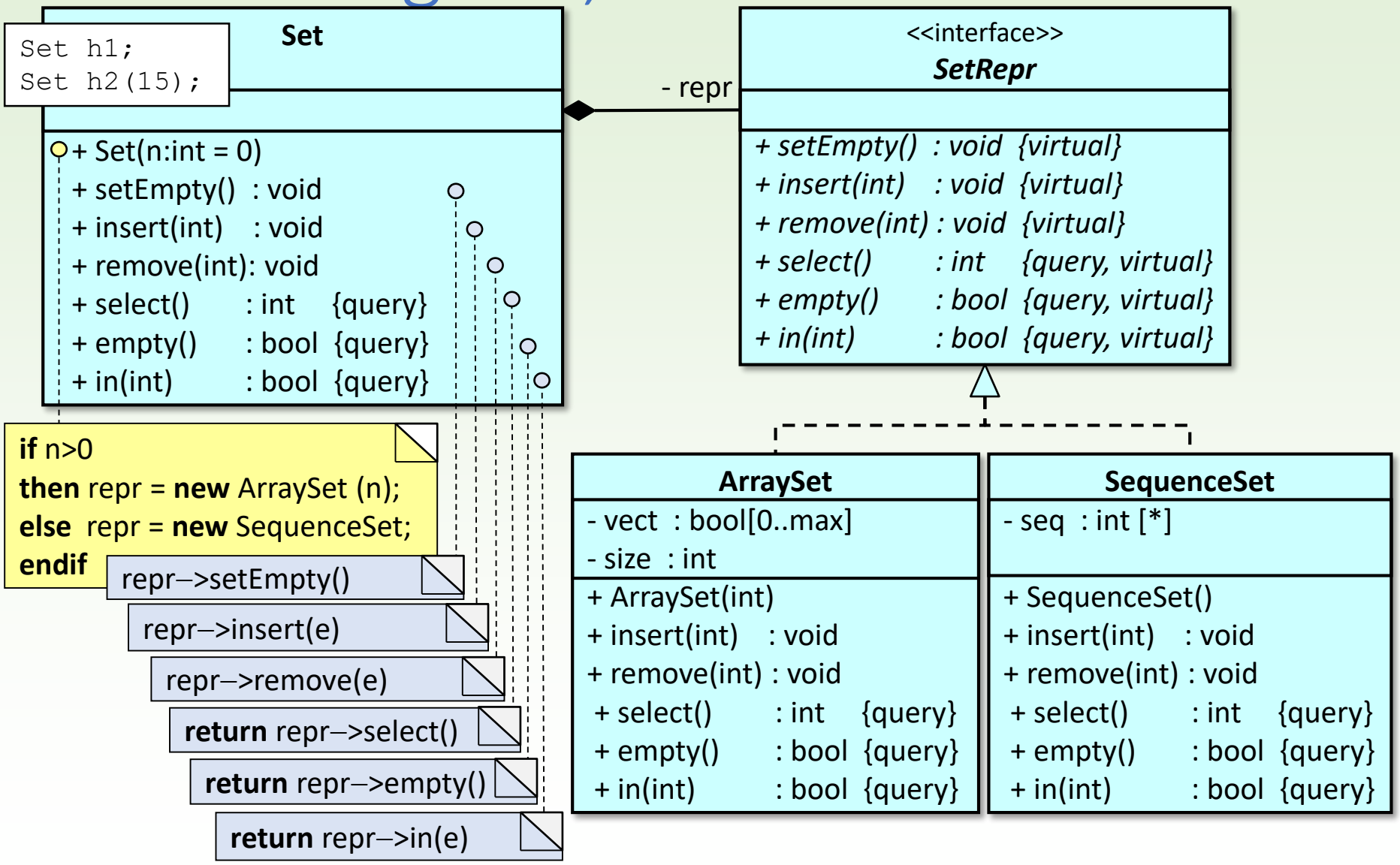
How to implement both representations ??

set.h

Class diagram, 1st version

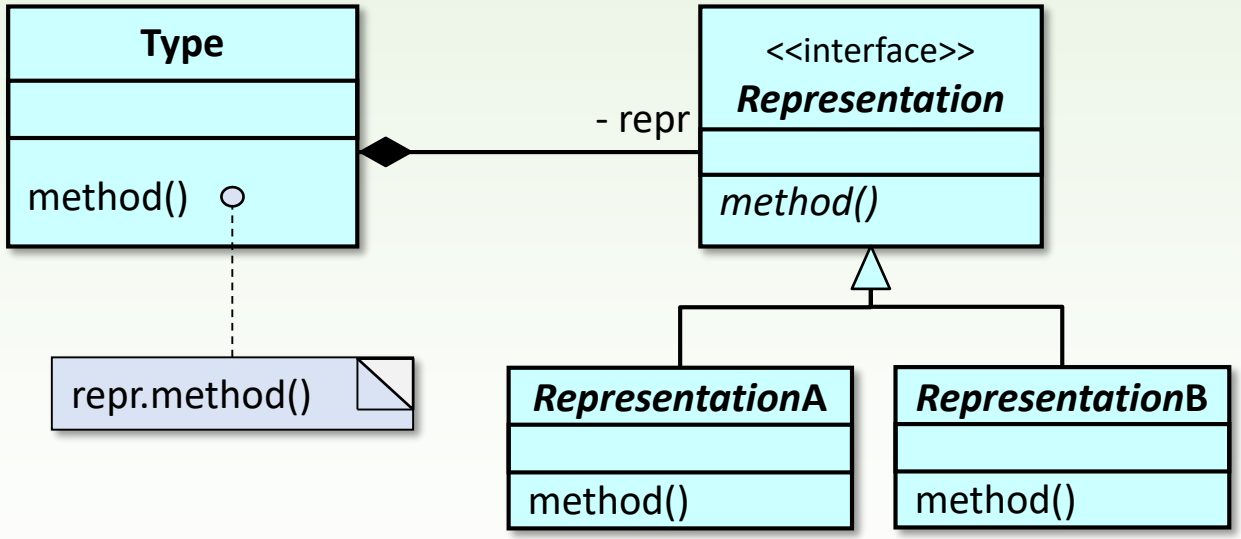


Class diagram, 2nd version



Bridge design pattern

- The representation of a class is separated from the class itself, so that it may be changed flexibly.



Design patterns are class diagram patterns that help object-oriented modeling. They play a significant role in reusability, modifiability, and ensuring efficiency.

Class Set, inline way

```
#include "setrepr.h"
#include "array_set.h"
#include "sequence_set.h"

class Set {
public:
     Set(int n = 0) {
          if (n>0) _repr = new ArraySet(n);
          else     _repr = new SequenceSet;
     }
     ~Set() { delete _repr; }
     void setEmpty()      { _repr->setEmpty(); }
     void insert(int e)   { _repr->insert(e); }
     void remove(int e)   { _repr->remove(e); }
     int  select() const { return _repr->select(); }
     bool empty()  const { return _repr->empty(); }
     bool in(int e)const { return _repr->in(e); }
private:
     SetRepr *_repr;

     Set(const Set& h);
     Set& operator=(const Set& h);
};
```

Set
+ Set(n:int = 0) + setEmpty() : void + insert(int) : void + remove(int) : void + select() : int {query} + empty() : bool {query} + in(int) : bool {query}

The default copy contructor and assignment operator would not work well.
If this is private, they cannot be used.
Later on, they can be defined and made public.

set.h

Side note: default copy and assignment

```
class Integer {
public:
      Integer()  { _p = new int; *_p = 0; }
    ~ Integer()  { delete _p; }
      void addl(int i) { *_p += i; }
      int get() const   { return *_p; }
private:
      int *_p;
};
```

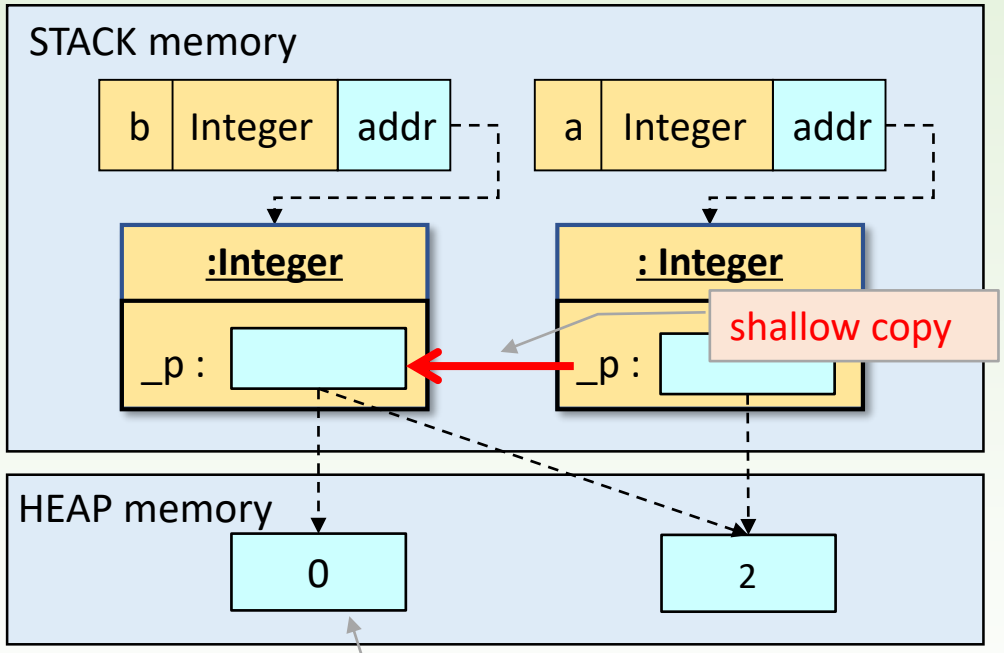
```
Integer a, b;
b = a;
```

assignment operator

```
b.add(2);
cout << a.get();
```

```
Integer b = a;
```

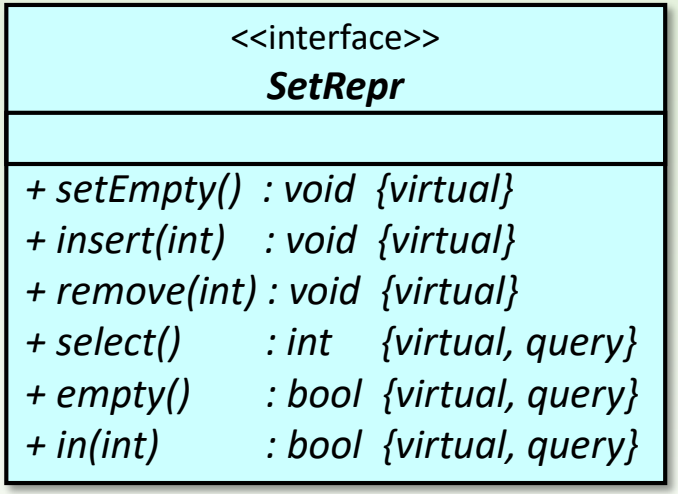
copy constructor



memory leaking

deep copy is needed

Interface of the representation



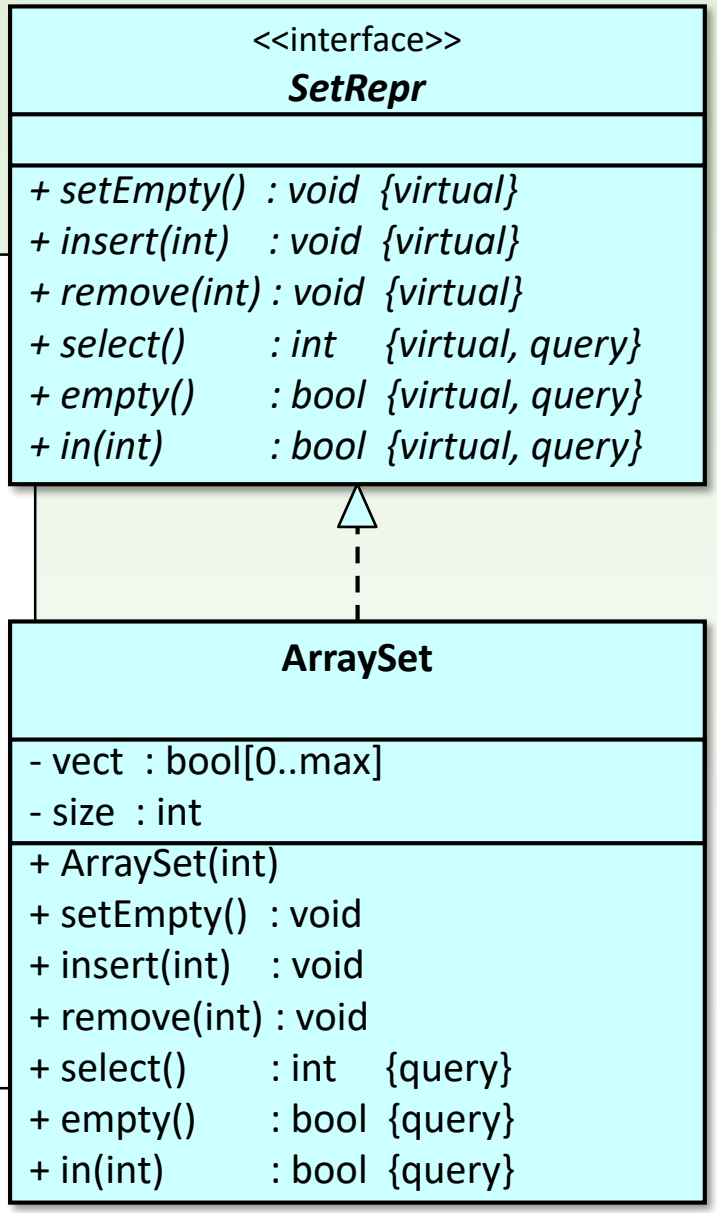

```
class SetRepr
{
public:
     virtual void setEmpty()      = 0;
     virtual void insert(int e)   = 0;
     virtual void remove(int e)   = 0;
     virtual int  select() const  = 0;
     virtual bool empty()  const  = 0;
     virtual bool in(int e)const  = 0;
     virtual ~SetRepr(){}
};
```

setrepr.h

Array representation

```
#include "setrepr.h"
#include <vector>

class ArraySet : public SetRepr{
public:
    ArraySet (int n): _vect(n+1), _size(0){
        setEmpty();
    }
    void setEmpty()      override;
    void insert(int e)   override;
    void remove(int e)   override;
    int  select() const override;
    bool empty()  const override;
    bool in(int e)const override;
private:
    std::vector<bool> _vect;
    int _size;
};
```

array_set.h



Exceptions

```
#include <exception>
#include <sstream>

class EmptySetException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Empty set";
    }
};

class IllegalElementException : public std::exception {
private:
    int _e;
public:
    IllegalElementException(int e): _e(e) {}
    const char* what() const noexcept override {
        std::ostringstream os;
        os <<  "Illegal element: " << _e;
        std::string str = os.str();
        char * msg = new char[str.size() + 1];
        std::copy(str.begin(), str.end(), msg);
        msg[str.size()] = '\0';
        return msg;
    }
};
```

setrepr.h

Methods of ArraySet

```
int ArraySet::select() const
{
    if(_size==0) throw EmptySetException();
    int e;
    for(e=0; !_vect[e]; ++e);
    return e;
}

bool ArraySet::empty() const
{
    return _size==0;
}

bool ArraySet::in(int e) const
{
    if(e<0 || e>int(vect.size())-1) throw IllegalElementException(e);
    return _vect[e];
}
```

```
Set h(100);
try {
    h.insert(101);
} catch(exception &ex){
    cout << ex.what() << endl;
}
```

exception instantiation and throw

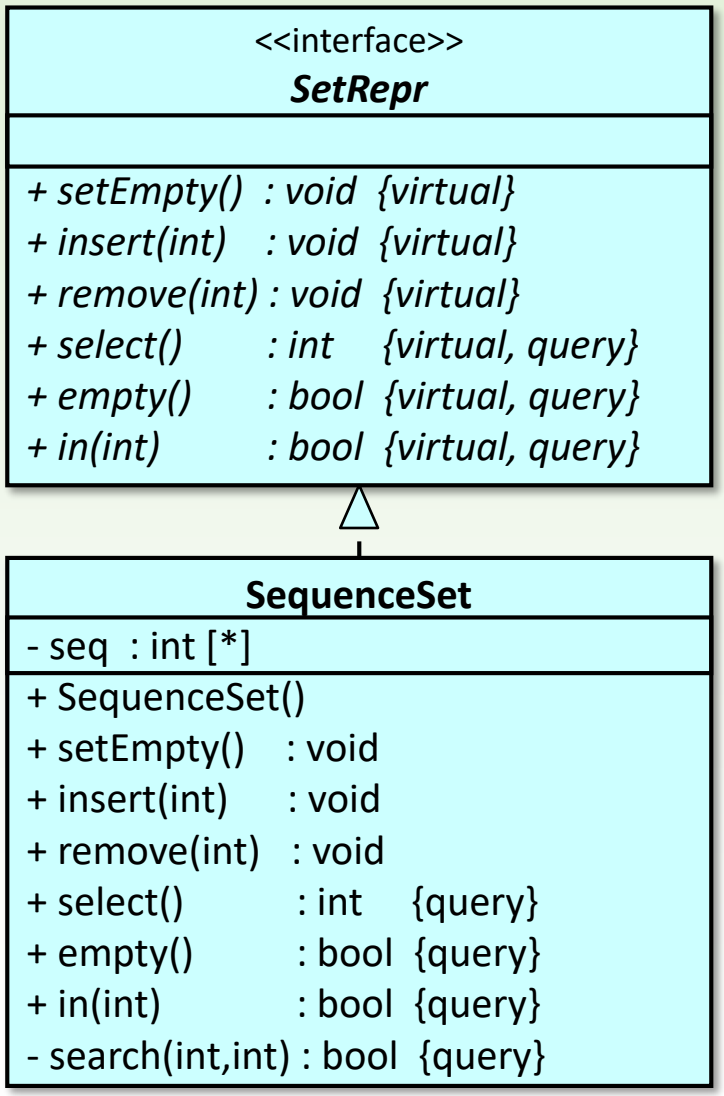
exception instantiation and throw

array_set.cpp

Sequence representation

```
#include "setrepr.h"
#include <vector>

class SequenceSet : public SetRepr{
public:
     SequenceSet (){ _seq.clear(); }
     void setEmpty()      override;
     void insert(int e)   override;
     void remove(int e)   override;
     int  select() const override;
     bool empty()  const override;
     bool in(int e)const override;
private:
     std::vector<int> _seq;
     bool search(int e,
        unsigned int &ind) const;
};
```

sequence_set.h



2nd task

Search for an item in a set of natural numbers which is greater than at least three other items in the set. (The search is obvoiusly unsuccessful if there are at last 3 items in the set.)

- A possible solution would be a linear search for an item, where a counting determines the number smaller items in the set.
- The enumeration of the items is used for both of the algorithmic patterns.

Specification

A : h:set(ℕ), l:𝕃, n:ℕ

Pre: $h = h_0$

Post: l, n = **SEARCH**$_{e \in h_0}$ (NoLess(h_0, e) ≥ 3)

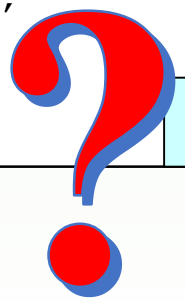
$$\text{NoLess}(h_0, e) = \sum_{\substack{u \in h_0 \\ e > u}} 1$$

standard enumeration of a set:

first()	~	-
next()	~	remove(current())
end()	~	empty()
current()	~	select()

```
bool l = false;
int n;
for( ; !l && !h.empty(); h.remove(n)){
     n = h.select();
     int c = 0;
     for( ; !h.empty(); h.remove(h.select())){
          if(n > h.select()) ++c;
     }
     l = c>=3;
}
```

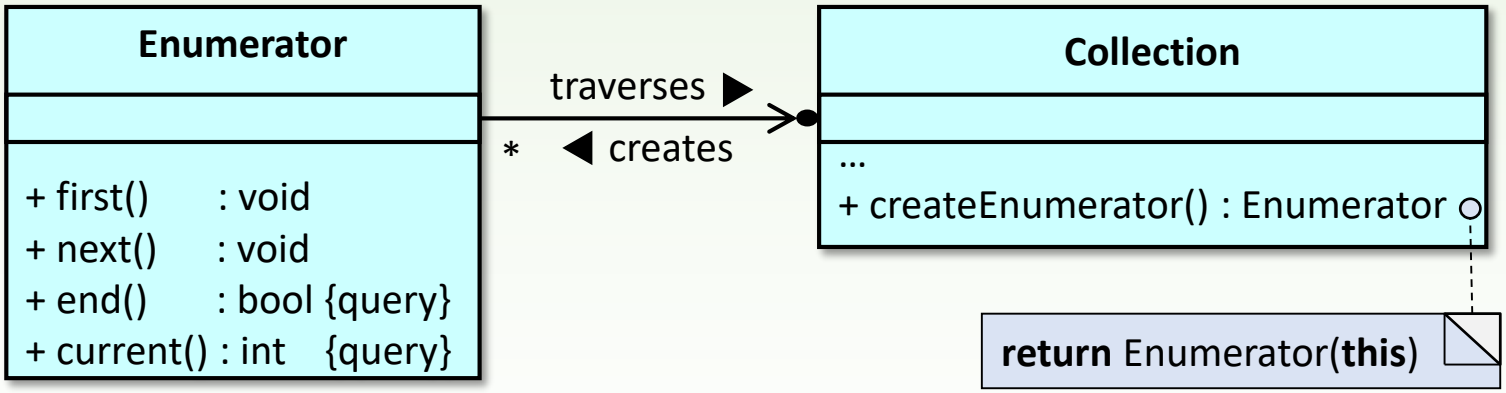
main.cpp



Not a good solution:
- the standard enumeration modifies the set, the inner loop removes all of the items
- the two enumerations are not independent, the two enumerations are interlocked

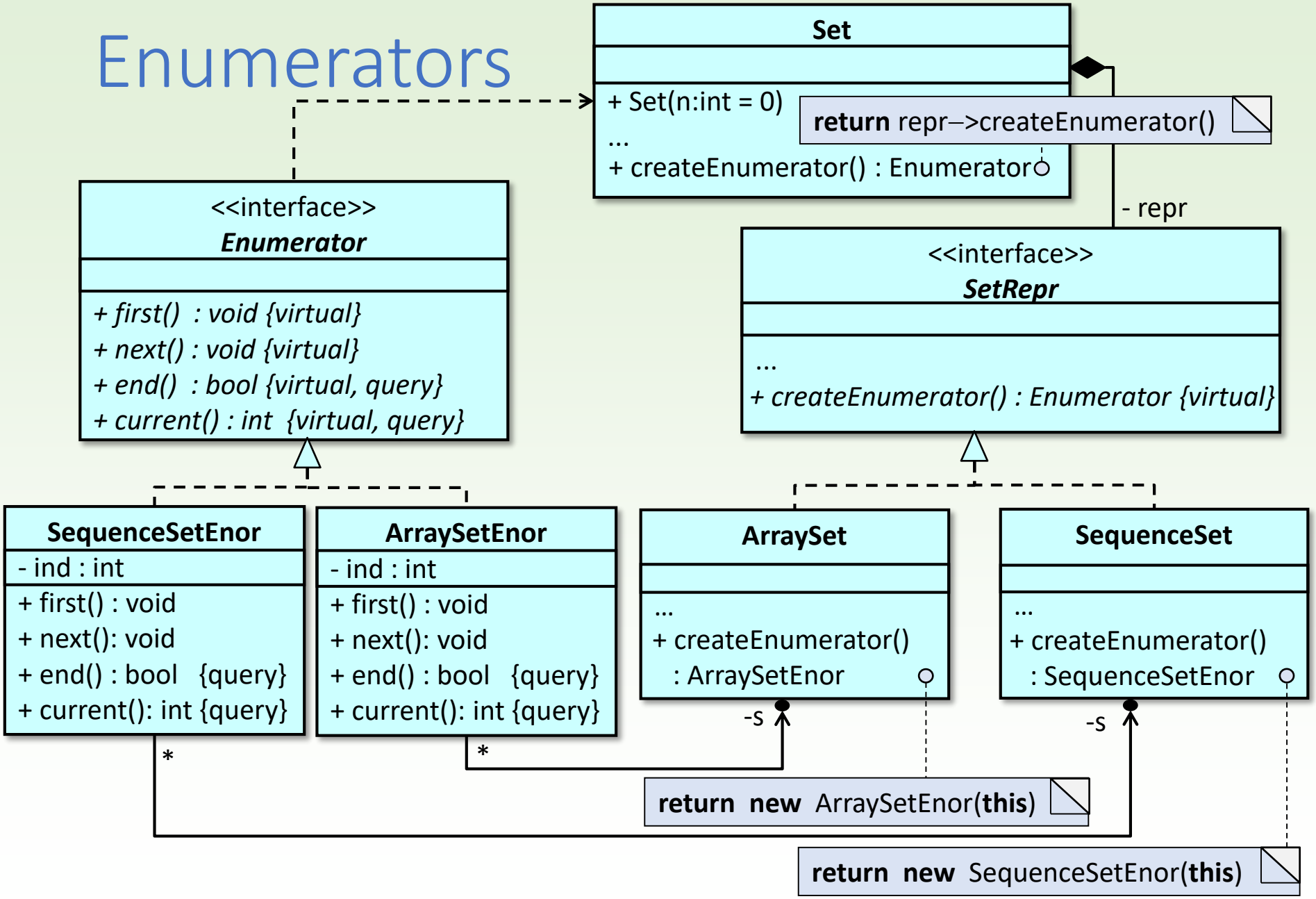
Iterator design pattern

- The enumeration (traversal) of a collection is done by an independent object (enumerator) which can access the collection (refers to it or to a constant copy). The enumerator object is created by the collection.



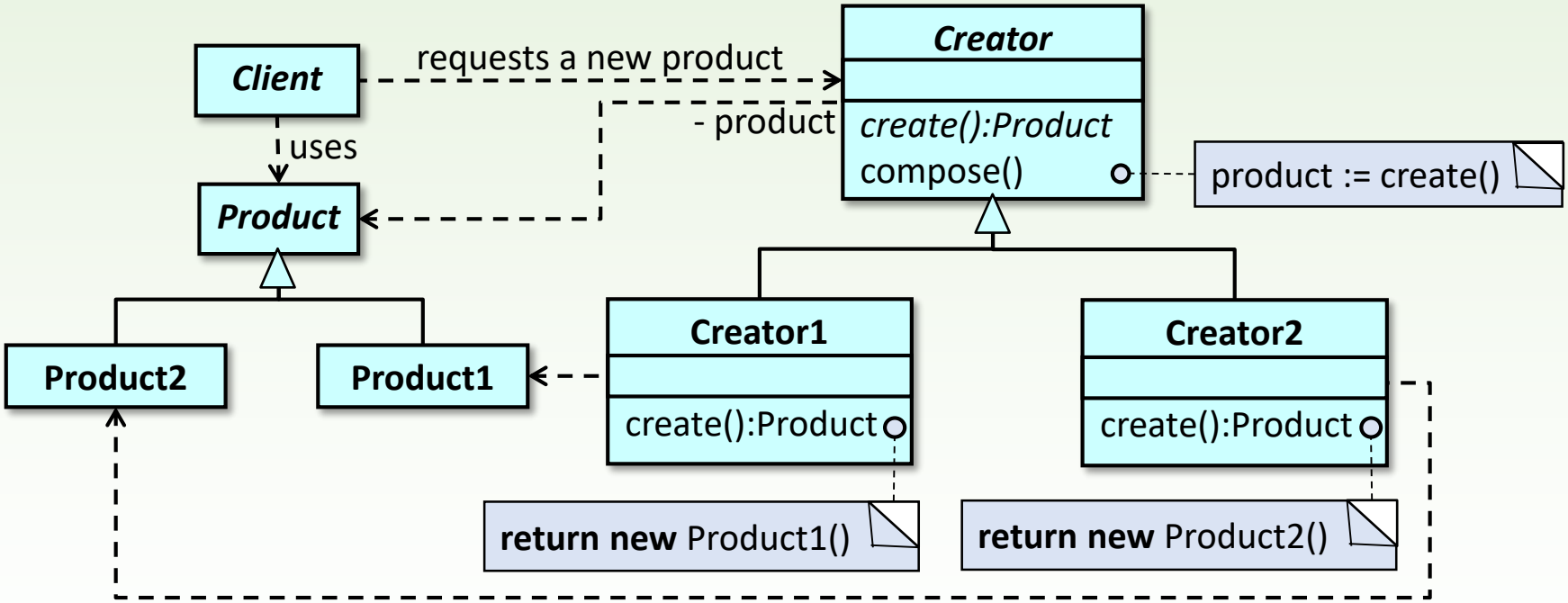
Design patterns are class diagram patterns that help object-oriented modeling. They play a significant role in reusability, modifiability, and ensuring efficiency.

Enumerators



Factory method design pattern

- The client does not know what kind of product-object it needs. It transfers the responsibility to one of the supporting subclasses.



Design patterns are class diagram patterns that help object-oriented modeling. They play a significant role in reusability, modifiability, and ensuring efficiency.

Main program

```
Set h;
// Reading data
…

bool l = false;
int n;
Enumerator* enor1 = h.createEnumerator();
for(enor1->first(); !l && !enor1->end(); enor1->next()){
     n = enor1->current();
     int c = 0;
     Enumerator* enor2 = h.createEnumerator();
     for(enor2->first(); !enor2->end(); enor2->next()){
          if(n > enor2->current()) ++c;
     }
     l = (c >= 3);
}

if (l) cout << "The number you are looking for: " << n << endl;
else   cout << "There is no such number.\n";
```

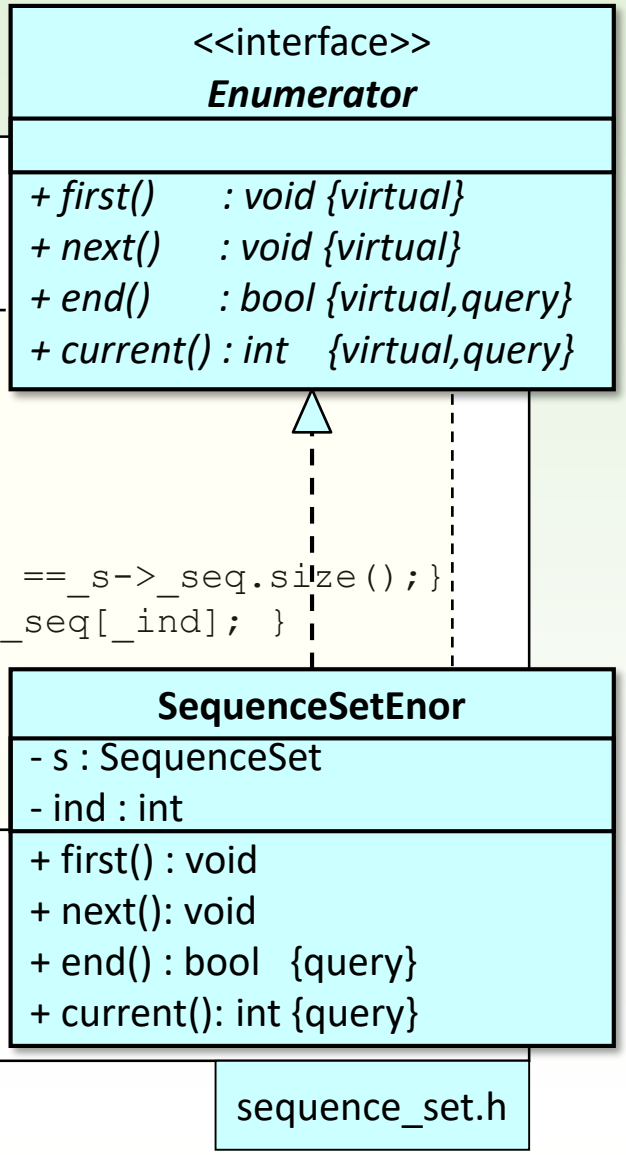
in the background:
repr–>createEnumerator()
return new SequenceSetEnor(this)

main.cpp

Enumerator of SequenceSet

```
class SequenceSet{
public:
     …
     class SequenceSetEnor : public Enumerator{
     public:
          SequenceSetEnor(SequenceSet *h): _s(h) {}
          void first()         override { _ind = 0; }
          void next()          override { ++_ind;}
          bool end()    const override { return _ind ==_s->_seq.size();}
          int current()const override { return _s->_seq[_ind]; }
     private:
          SequenceSet  *_s;
          unsigned int  _ind;
     };

     Enumerator* createEnumerator() override{
          return new SequenceSetEnor(this);
     }
};
```

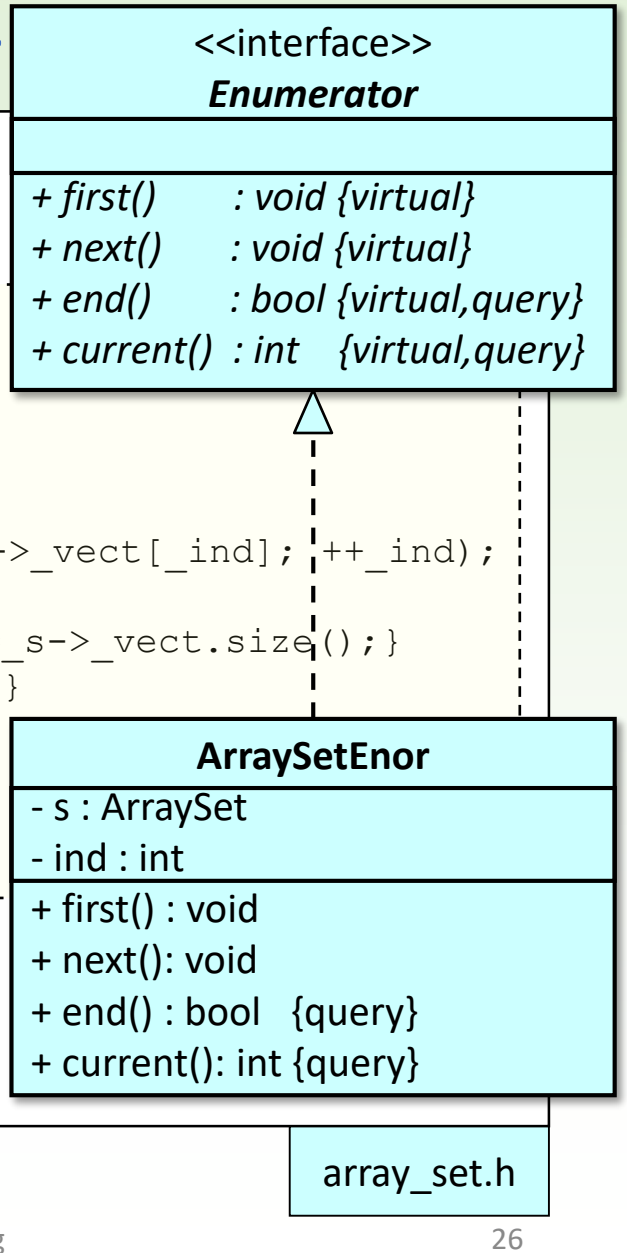
Enumeration of the items is realized by enumeration of the sequence which represents the set.



Enumerator of ArraySet

Enumeration of the items is realized by enumeration of the indexes of the array where the corresponding value is true.

```
class ArraySet{
public:
    …
    class ArraySetEnor : public Enumerator{
    public:
        ArraySetEnor(ArraySet *h): _s(h) {}
        void first() override { _ind = -1; next(); }
        void next()  override {
            for(++_ind;_ind<_s->_vect.size() && !_s->_vect[_ind]; ++_ind);
        }
        bool end()      const override {return _ind==_s->_vect.size();}
        int  current() const override {return _ind; }
    private:
        ArraySet  *_s;
        unsigned int _ind;
    };

    Enumerator* createEnumerator() override{
        return new ArraySetEnor(this);
    }
};
```



array_set.h

3rd task

Make the enumeration secure

- Problem: the enumeration might be wrong if the set is changed during the enumeration.
- Critical operations: setEmpty(), insert(), remove(), assignment operator, destructor.
- Solution: Do not let the critical operations to be run.

```
Set h;
 …
Enumerator * enor = h.createEnumerator();
for(enor->first(); !enor->end(); enor->next()){
     int e = enor->current();
     h.remove(e);
}
```

If sequence representation is used, deleting the current item causes an error.

main.cpp

Locking the operations

```
class UnderTraversalException : public std::exception {
public:
     const char* what() const noexcept override {
          return "Under traversal";
     }
};
```

setrepr.h

```
class Set {
public:
     …
     void Set::remove(int e)
     {
          if(_repr->getEnumCount()!=0) throw UnderTraversalException();
          _ref->remove(e);
     }
     ~Set() {
          if(_repr->getEnumCount()!=0) throw UnderTraversalException();
          delete repr;
     }
     …
};
```

The critical operation throws an exception if it is under traversal.

set.h

```
class SetRepr {
public:
     SetRepr(): _enumeratorCount(0){}
     …
     int getEnumCount() const { return _enumeratorCount; }
protected:
     int _enumeratorCount;
};
```

this is not an interface any more, but an abstract class

number of the active enumerators

setrepr.h

ArraySet

```
class ArraySet : public SetRepr {
public:
    ArraySet(int n): SetRepr(), _vect(n+1), _size(0) {
        setEmpty();
    }

    …
    class ArraySetEnor : public Enumerator{
    public:
        ArraySetEnor(ArraySet *h): _s(h)
        { ++(_s->_enumeratorCount); }
        ~ArraySetEnor() { --(_s->_enumeratorCount); }
        …
    };

    Enumerator * createEnumerator() override {
        return new ArraySetEnor(this);
    }
};
```

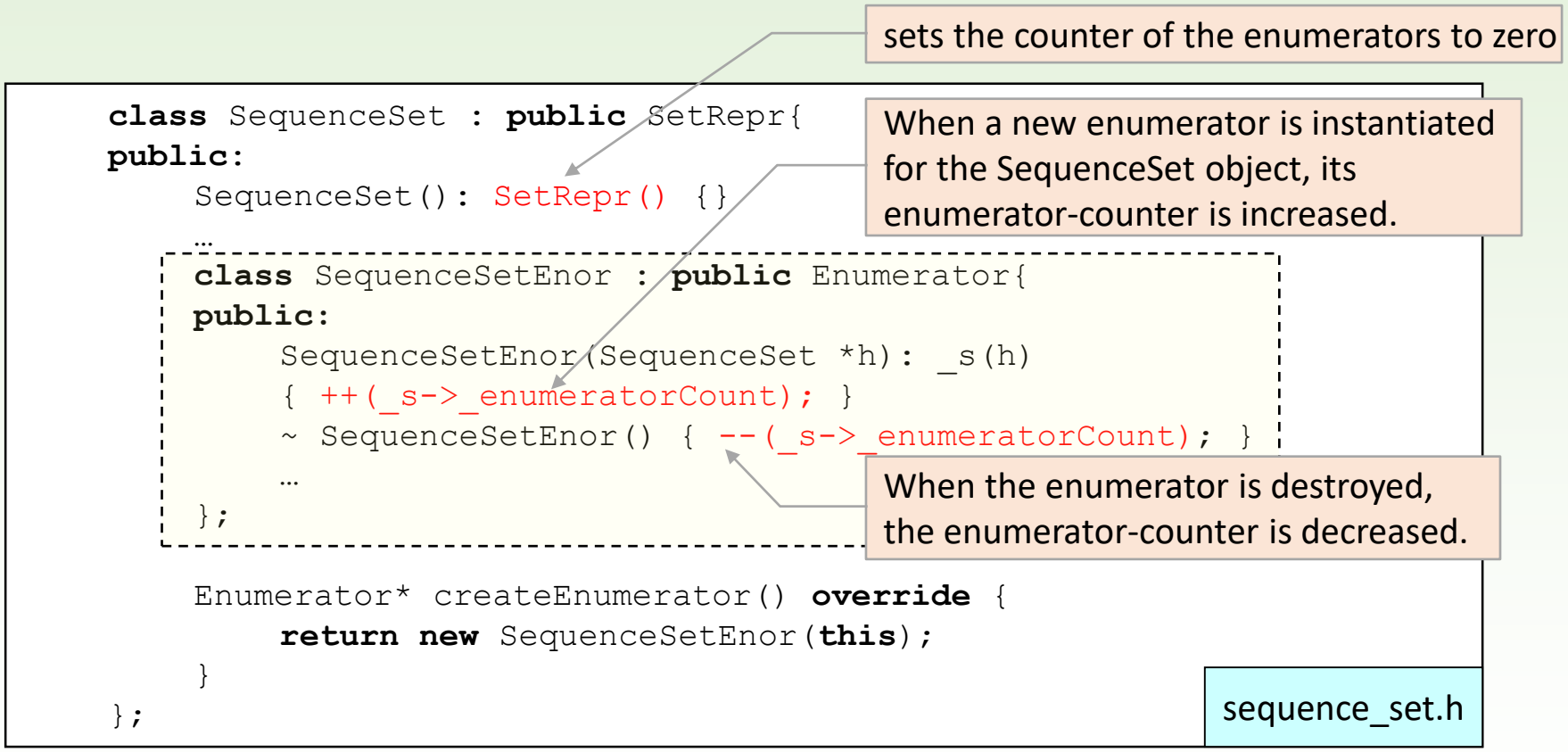
sets the counter of the enumerators to zero

When a new enumerator is instantiated for the ArraySet object, its enumerator-counter is increased.

When the enumerator is destroyed, the enumerator-counter is decreased.

array_set.h

SequenceSet




```
class SequenceSet : public SetRepr{
public:
    SequenceSet(): SetRepr() {}
    …
    class SequenceSetEnor : public Enumerator{
    public:
        SequenceSetEnor(SequenceSet *h): _s(h)
        { ++(_s->_enumeratorCount); }
        ~ SequenceSetEnor() { --(_s->_enumeratorCount); }
        …
    };

    Enumerator* createEnumerator() override {
        return new SequenceSetEnor(this);
    }
};
```

4th task

- Create class templates, where the type of the items (in the set) can be given as an input paramterer.
- Remark: in case of array representation, only natural numbers with upper bound can be stored, the template parameter has to be **int**.

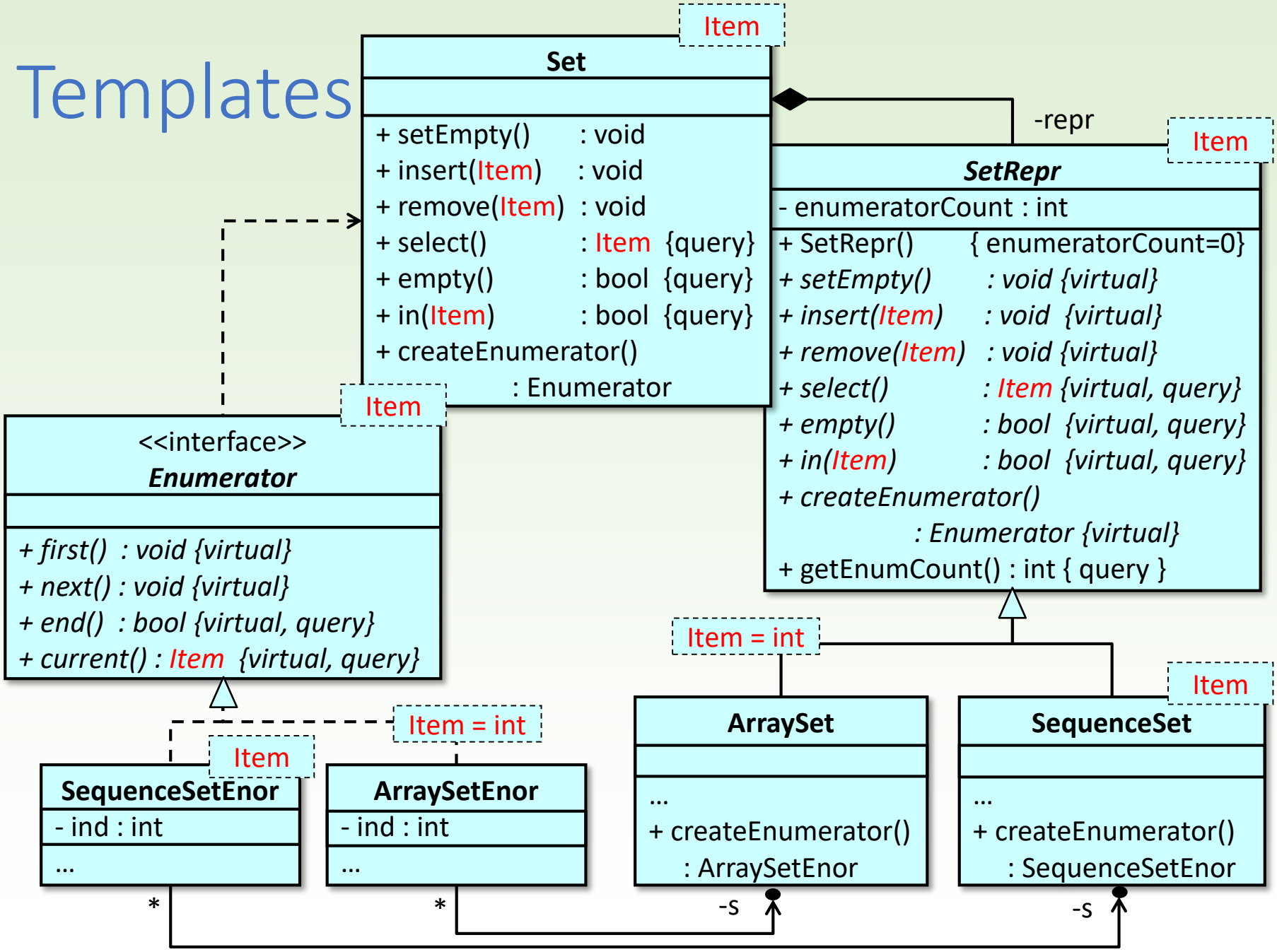
in compilation time, the class template is instantiated as a class, in runtime, the class is instantiated as an object

```
Set<int>    h1(100);
Set<int>    h2;
Set<string> h3;

h1.insert(12);
h2.insert(123);
h3.insert("apple");

Enumerator<int>    *enor1 = h1.createEnumerator();
Enumerator<string> *enor2 = h3.createEnumerator();
```

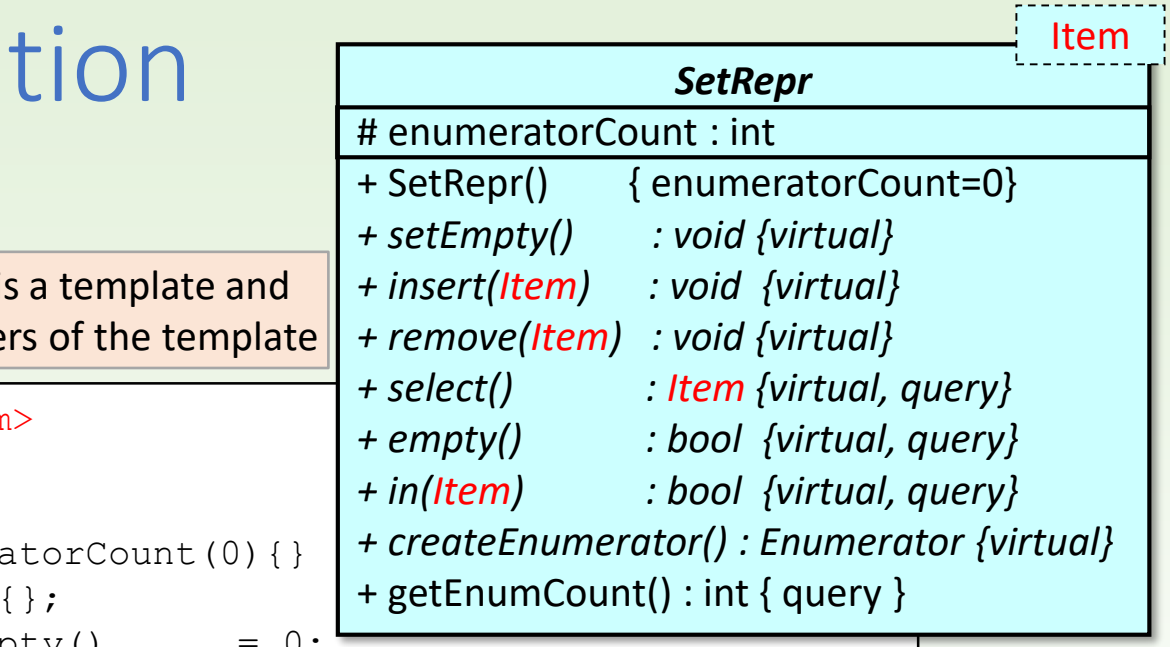
The enumerator classes are instances of class templates, too, the type parameter of which has to match the type of the set.

main.cpp

Templates


Item
Set
+ setEmpty() : void
+ insert(Item) : void
+ remove(Item) : void
+ select() : Item {query}
+ empty() : bool {query}
+ in(Item) : bool {query}
+ createEnumerator()
: Enumerator
-repr
Item
SetRepr
- enumeratorCount : int
+ SetRepr() { enumeratorCount=0}
+ setEmpty() : void {virtual}
+ insert(Item) : void {virtual}
+ remove(Item) : void {virtual}
+ select() : Item {virtual, query}
+ empty() : bool {virtual, query}
+ in(Item) : bool {virtual, query}
+ createEnumerator()
: Enumerator {virtual}
+ getEnumCount() : int { query }
Item
<<interface>>
Enumerator
+ first() : void {virtual}
+ next() : void {virtual}
+ end() : bool {virtual, query}
+ current() : Item {virtual, query}
Item = int
ArraySet
...
+ createEnumerator()
: ArraySetEnor
Item
SequenceSet
...
+ createEnumerator()
: SequenceSetEnor
Item
SequenceSetEnor
- ind : int
...
Item = int
ArraySetEnor
- ind : int
...
*
*
-s
-s

Interface-template of the representation



indicates that this is a template and gives the parameters of the template

```
template <typename Item>
class SetRepr {
public:
     SetRepr(): _enumeratorCount(0){}
     virtual ~SetRepr(){};
     virtual void setEmpty()       = 0;
     virtual void insert(Item e)   = 0;
     virtual void remove(Item e)   = 0;
     virtual Item select()  const = 0;
     virtual bool empty()   const = 0;
     virtual bool in(Item e)const = 0;
     virtual Enumerator<Item>* createEnumerator() = 0;
     int getEnumCount() const { return _enumeratorCount; }
protected:
     int _enumeratorCount;
};
```

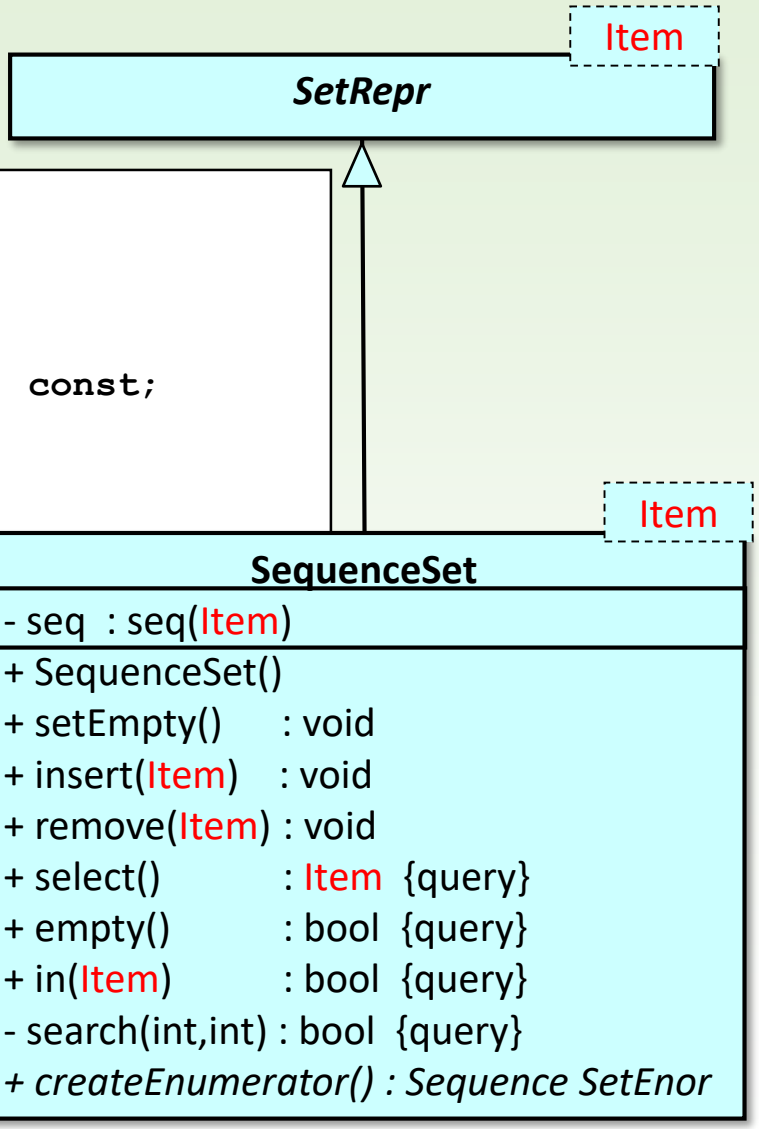
setrepr.h

Template of SequenceSet

```
template <typename Item>
class SequenceSet : public SetRepr<Item>{
private:
     std::vector<Item> _seq;
     bool search(Item e, unsigned int &ind) const;
public:
     SequenceSet (){ _seq.clear(); }
     void setEmpty()       override;
     void insert(Item e)   override;
     void remove(Item e)   override;
     Item select()  const override;
     bool empty()   const override;
     bool in(Item e)const override;
     …
};
…
```

sequence_set.hpp

Class template definition (.h) and template-method definitions (.cpp) go to the same file.



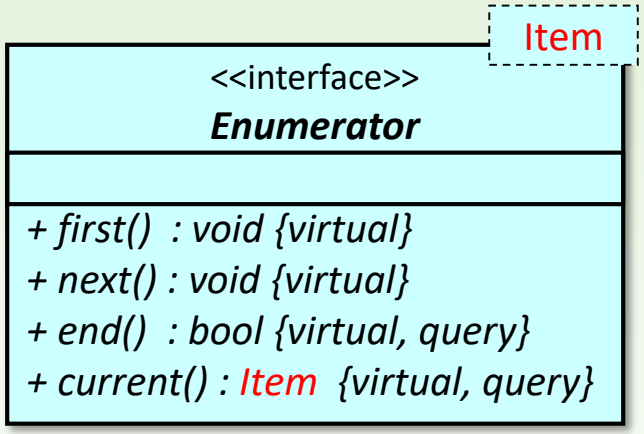
Methods of template SequenceSet

```
…
template <typename Item>
void SequenceSet<Item>::setEmpty() { _seq.clear(); }

template <typename Item>
void SequenceSet<Item>::insert(int e)
{
    unsigned int ind;
    if(!search(e,ind)) _seq.push_back(e);
}
template <typename Item>
void SequenceSet<Item>::remove(int e)
{
    unsigned int ind;
    if(search(e,ind)){
        _seq[ind] = _seq[_seq.size()-1];
        _seq.pop_back();
    }
}
template <typename Item>
int SequenceSet<Item>::select() const
{
    if(_seq.size()==0) throw EmptySetException();
    return _seq[0];
}
```

Not just a class, but a function may be template, too.
Specifically, methods of a class template are templates, too.

sequence_set.hpp

Interface-template of the enumerator



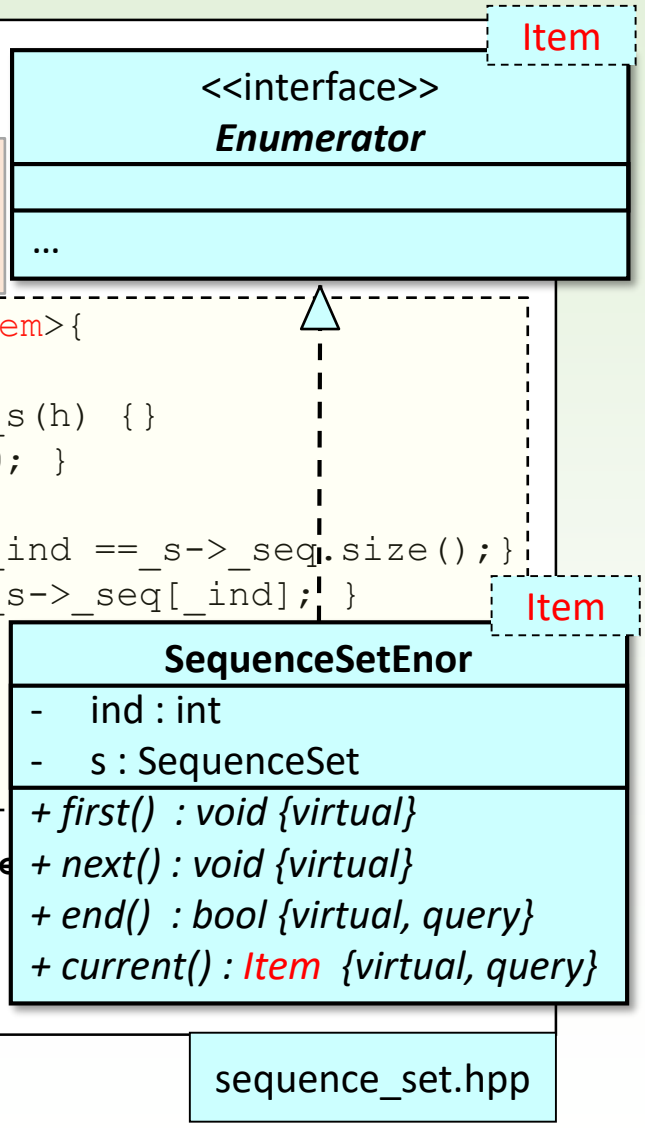

```
template <typename Item>
class Enumerator {
public:
     virtual void first() = 0;
     virtual void next()  = 0;
     virtual bool end()       const = 0;
     virtual Item current() const = 0;
     virtual ~Enumerator(){};
};
```

enumerator.h

Template of SequenceSetEnor

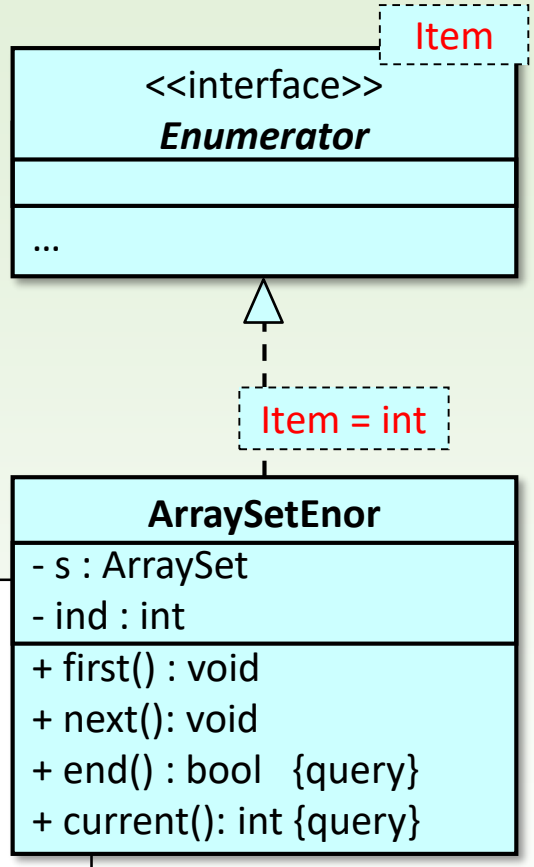
```
template <typename Item>
class SequenceSet : public SetRepr<Item>{
public:
    …
    class SequenceSetEnor : public Enumerator<Item>{
    public:
        SequenceSetEnor(SequenceSet<Item> *h): _s(h) {}
        void first()          override { _ind = 0; }
        void next()           override { ++_ind;}
        bool end()      const override { return _ind ==_s->_seq.size();}
        Item current() const override { return _s->_seq[_ind]; }
    private:
        SequenceSet<Item>  *_s;
        unsigned int  _ind;
    };

    Enumerator<Item>* createEnumerator() override
        return new SequenceSetEnor<Item> (this);
    }
};
```

because of the embedding, this is a template with parameter *Item*: indicating that this is a template is not necessary



sequence_set.hpp

Enumerator of ArraySet



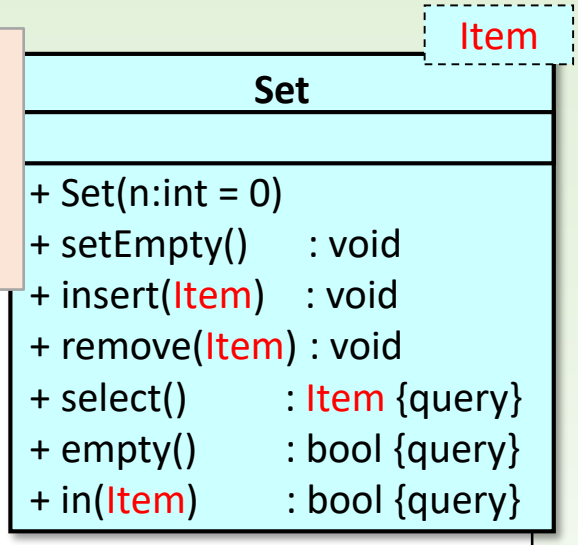
ArraySet does not change, except that in the inheritance, the parent requires **int** as template parameter

```
class ArraySet : public SetRepr<int>{
public:
    class ArraySetEnor : public Enumerator<int>{
        …
    };

    Enumerator<int>* createEnumerator() override{
        return new ArraySetEnor(this);
    }
};
```

array_set.hpp

Class template Set

Compilation error:
This assignment may cause error if *Item* is not int. Though, this assignment is needed only if Item=int.




```
template <typename Item>
class Set {
public:
     Set(int n = 0) {
          if (0 == n) _repr = new SequenceSet<Item>;
          else        _repr = new ArraySet(n);
     }
     ~Set() { delete _repr; }
     void setEmpty()       { _repr->setEmpty(); }
     void insert(Item e)   { _repr->insert(e); }
     void remove(Item e)   { _repr->remove(e); }
     Item select()  const { return _repr->select(); }
     bool empty()   const { return _repr->empty(); }
     bool in(Item e)const { return _repr->in(e); }

     Enumerator<Item>* createEnumerator() { _repr->createEnumerator(); }
private:
     SetRepr<Item> *_repr;

     Set(const Set& h) ;
     Set& operator=(const Set& h);
};
```

set.h

Instantiation depending on a parameter instead of a conditional

```
template <typename Item>
class Set {
public:
     Set(int n = 0) { _repr = createSetRepr<Item>(n); }
     …
private:
     SetRepr<Item>* _repr;

     static SetRepr<Item>* createSetRepr(int n) {
          return new SequenceSet<Item>;
     }
};
```

a factory design pattern template instantiates the representation

General creator template for class Set<Item>

set.h

```
template<>
inline SetRepr<int>* Set<int>::createSetRep(int n) {
     if (n > 0) return new ArraySet(n);
     else       return new SequenceSet<int>;
}
```

Special creator template for class Set<int>

set.h