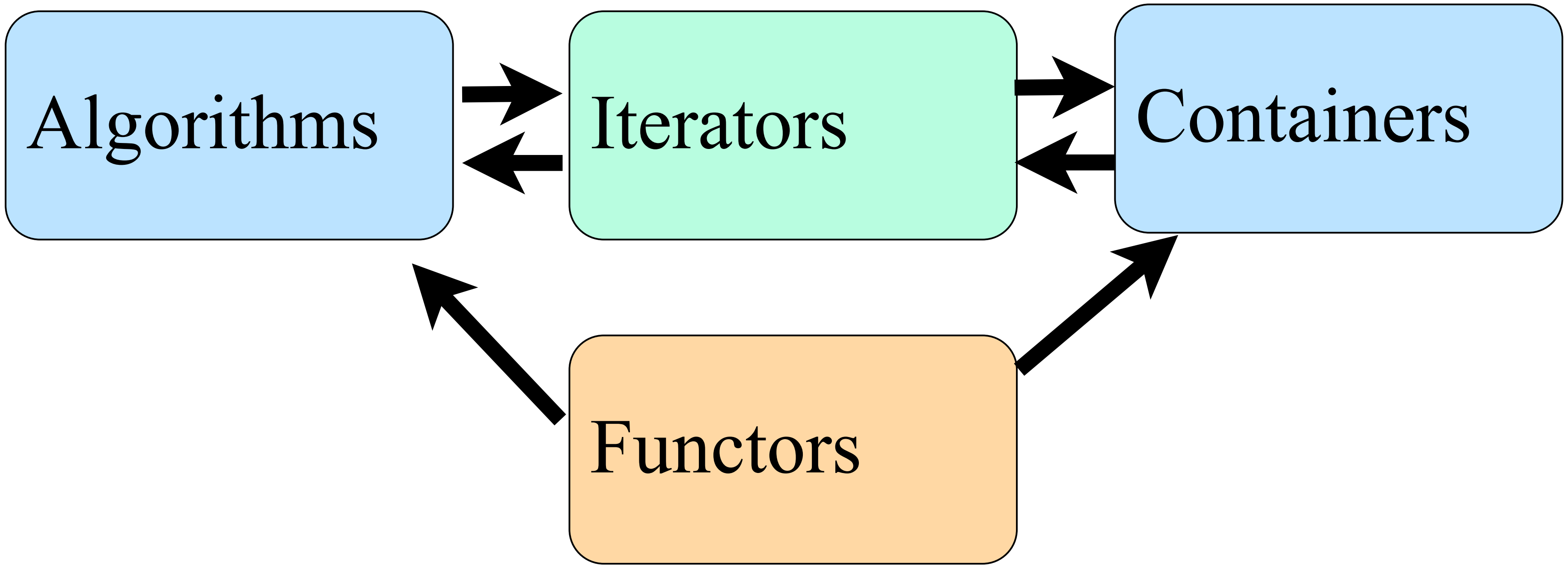
New Orleans, LA (MSY)
New Stuyahok, AK (KNW)
New York, NY - All airports (NYC)
New York, NY - Kennedy (JFK)
New York, NY - La Guardia (LGA)
Newark, NJ (EWR)
Newburgh/Stewart Field, NY (SWF)
Newport News, VA (PHF)
Newtok, AK (WWT)
Nightmute, AK (NME)
Nikolai, AK (NIB)
Nikolski, AK (IKO)
Noatak, AK (WTK)
Nome, AK (OME)
Nondalton, AK (NNL)
Noorvik, AK (ORV)
Norfolk, NE (OFK)
Norfolk, VA (ORF)
North Bend, OR (OTH)
North Platte, NE (LBF)
Northway, AK (ORT)
Nuiqsut, AK (NUI)
Nulato, AK (NUL)
Nunapitchuk, AK (NUP)
Oakland, CA (OAK)
Odessa/Midland, TX (MAF)
Ogdensburg, NY (OGS)
Oklahoma City, OK (OKC)
Omaha, NE (OMA)
Ontario, CA (ONT)
Orange County, CA (SNA)

Easy to use code written by someone else: portable, fast, well designed, documented

The interfaces to standard library facilities are defined in headers: <algorithm>, <functional>,<iterator>, <list>, <map>, queue>, <set>, <vector>, ...

STL

Standard Template Library



A C++ 11 STL reference can be found at:

http://en.cppreference.com/w/cpp

Another C++ reference can be found at:

http://www.cplusplus.com/reference/

“Mankind’s progress is
measured by the number of things
we can do without thinking”

Alfred North Whitehead

How do you organize data?

A *list* of items: $A_1, A_2, . . ., A_N$

We decide what is first, second, third, etc.

A *set* of items: $\{A_1, A_2, . . .,A_N\}$

We don’t think of the items having an order, and there are no duplicates

A *dictionary* of items: $\{(k_1,V_1),(k_2,V_2), . . .,(k_n,V_n)\}$

A set of items that map keys to values
For example:
{(apple, “the round fruit of a tree of the rose family, which typically has thin red or green skin and crisp flesh.”), (key, “a small piece of shaped metal with incisions cut to fit the wards of a particular lock, and that is inserted into a lock and turned to open or close it.”)}

{ (ORD, “Chicago, IL - O’Hare”), (JFK, “New York, NY - Kennedy”), (LGA,”New York, NY - La Guardia”), (ORD, “Chicago, IL - O’Hare”) }

A *stack* of items: Last In, First Out behavior of items

A *queue* of items: First In, First Out behavior of items

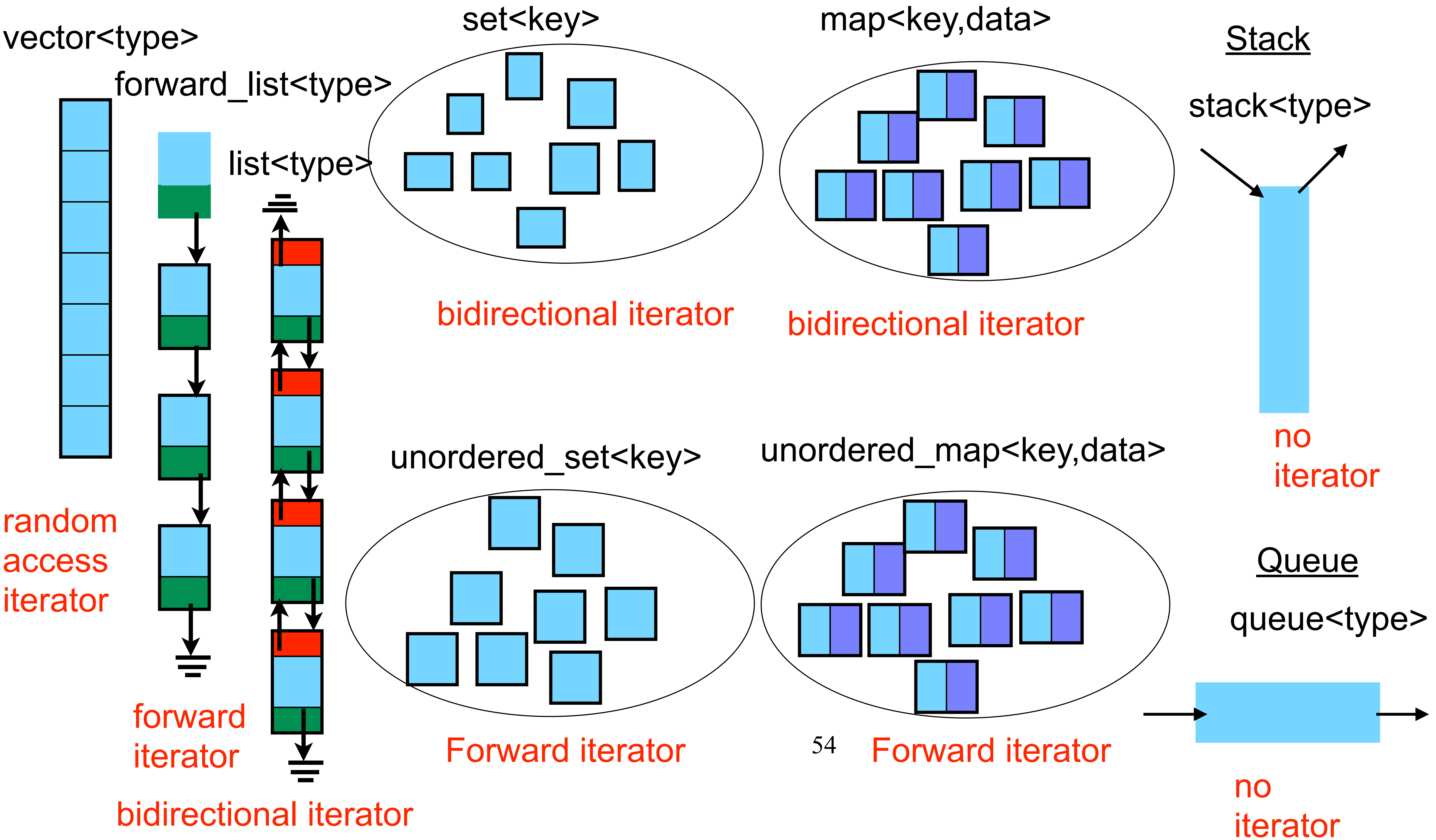
Different ADT’s have different operations we expect to perform on the data.

There are many ways we can organize the data we store in the computer

The way we organize the data in the computer affects how easy it is to insert, erase, or find an item

STL's ADT's

(not a complete list)



Lists
Sequence Containers

vector<type>

forward_list<type>

list<type>

random access iterator

forward iterator

bidirectional iterator

Set and Dictionary
Associate Containers

set<key>

bidirectional iterator

map<key,data>

bidirectional iterator

unordered_set<key>

Forward iterator

unordered_map<key,data>

Forward iterator

Container Adapters

Stack

stack<type>

no iterator

Queue

queue<type>

no iterator

We would like to write a template function that could work with more than one STL Container

How could the interface for a container help?

All containers in the STL contains:

- c.empty()
- c.clear( )
- c.size( )
- c.max_size()
- operator=
- c.swap()

Elements stored in a container need a default constructor, destructor, assignment operator. Some compilers need some overloaded operators as well

Any container adapter in the STL contains

- c.empty()
- c.clear( )
- c.size( )
- c.max_size()
- operator=
- c.swap()

We would like to write a template function that could work with more than one STL Container

How should we look at all the items in a container?

We need a way to iterate through the items that looks the same

- ++itr (or itr++) to move to next item
- *itr to dereference
- itr1 != itr2 to compare one iterator to another (or itr1 == itr2)
- c.begin() to refer to the first element
- c.end() to refer to one past the last element

Iterator Motivation

- Containers: vectors, linked lists, many other data structures hold a collection of objects
- We often want to step through a container visiting each object
- An *iterator* in C++ is an object that is used to step through a container systematically
- Common interface allows calling code to abstract away the details of the container: e.g. caller doesn’t know whether container is vector or linked list.

Code Examples for the vector and the list class

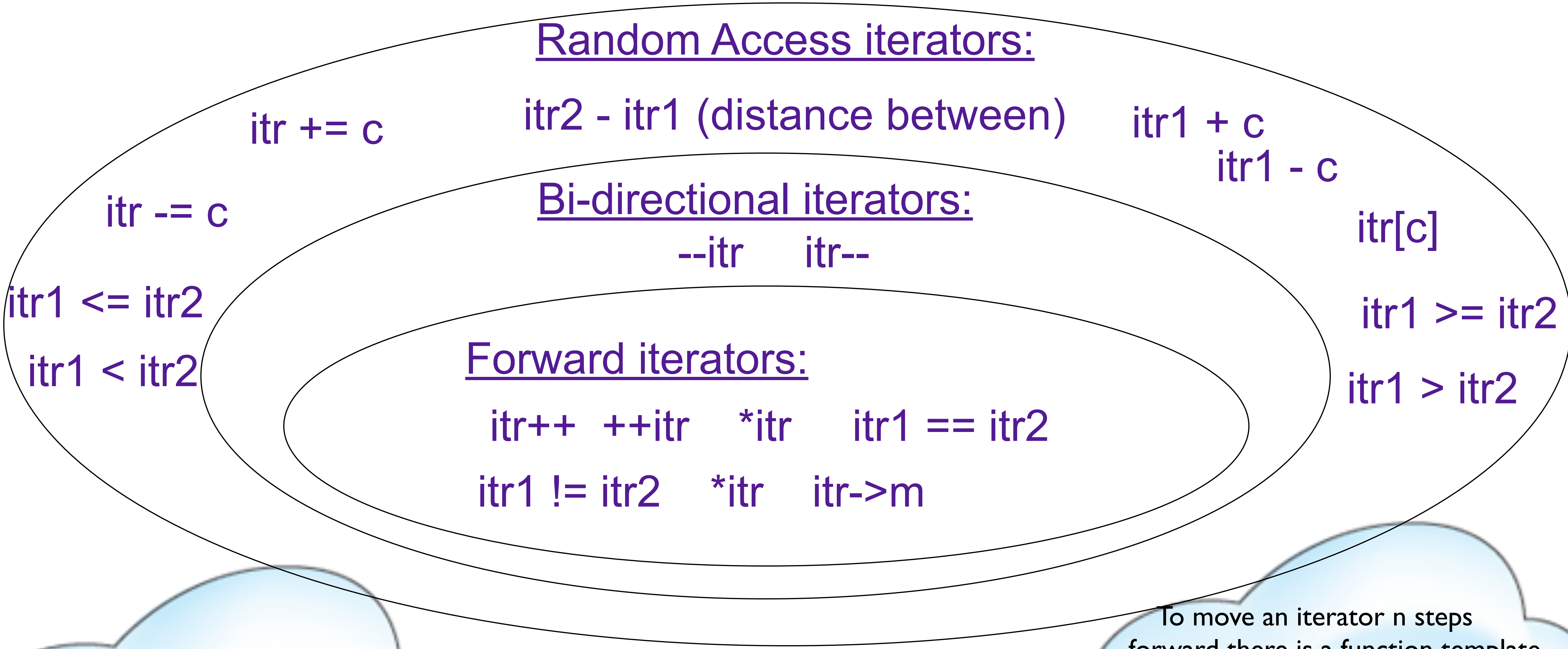
```
list<int> L;
list<int>::iterator itrL;

L.push_back(0);
L.push_front(1);
L.insert(++L.begin(), 2);
// insert(itr,x) member function
// inserts before itr

for (itrL = L.begin(); itrL != L.end(); ++itrL)
        cout << *itrL << " ";
// prints 1 2 0
```

```
vector<int> V;
vector<int>::iterator itrV;

V.push_back(1);
V.push_back(0);
V.insert(++V.begin(), 2);
// insert(itr,x) member function
// inserts x before itr

for (itrV = V.begin(); itrV != V.end(); ++itrV)
        cout << *itrV << " ";
// prints 1 2 0
```

Some Containers have more powerful iterators

The container type determines the iterator type

Syntax is similar to pointers



Random Access iterators:

itr += c

itr2 - itr1 (distance between)

itr1 + c

itr1 - c

itr -= c

itr[c]

itr1 <= itr2

itr1 < itr2

itr1 >= itr2

itr1 > itr2

Bi-directional iterators:

--itr   itr--

Forward iterators:

itr++   ++itr   *itr   itr1 == itr2

itr1 != itr2   *itr   itr->m

Generic Programming:
Essentially separating the data structure from the algorithm.
The way the STL algorithms work is by implementing algorithm based on the iterator types
The STL

To move an iterator n steps forward there is a function template called advance, advance(itr, n);
What do you think the running time o this function is?
There is function that determines the number of increments needed to get from Itr1 to Itr1, distance(Itr1, Itr2)

How to instantiate an iterator

A const iterator must be used if a contain
is non modifiable.

```
Class-name<template parameters>::iterator Itr;

Class-name<template parameters>::const_iterator VecItr;
```

For example:

```
    std::vector<int> myVector;
    std::vector<int>::iterator myVectorIterator;
```

Lets write a function that finds an item.

Should we store the items in a vector, list, set, unordered_set, map, or unordered map?

New York, NY - Kennedy (J
New York, NY - La Guardia
Newark, NJ (EWR)
Newburgh/Stewart Field, N
Newport News, VA (PHF)
Newtok, AK (WWT)
Nightmute, AK (NME)
Nikolai, AK (NIB)
Nikolski, AK (IKO)
Noatak, AK (WTK)
Nome, AK (OME)
Nondalton, AK (NNL)
Noorvik, AK (ORV)
Norfolk, NE (OFK)
Norfolk, VA (ORF)
North Bend, OR (OTH)
North Platte, NE (LBF)
Northway, AK (ORT)
Nuiqsut, AK (NUI)
Nulato, AK (NUL)
Nunapitchuk, AK (NUP)
Oakland, CA (OAK)
Odessa/Midland, TX (MAF)
Ogdensburg, NY (OGS)
Oklahoma City, OK (OKC)
Omaha, NE (OMA)
Ontario, CA (ONT)
Orange County, CA (SNA)
Orlando, FL - Herndon (OF
Orlando, FL - International
Oshkosh, WI (OSH)

Finding an item

```
vector<string>::iterator find(vector<string>::iterator start,
                         vector<string>::iterator end, string search_item)
{
    vector<string>::iterator itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
```

```
list<string>::iterator find(list<int>::iterator start,
                           list<string>::iterator end, string search_item)
{
    list<int>::iterator itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
```

```
set<string>::iterator find(set<string>::iterator start,
set<string>::iterator end,  string search_item)
{
    set<string>::iterator itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
```

```
template<class Iter, class Object>
Iter find(Iter start, Iter end, Object search_item)
{
    Iter itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
int main ()
{
    list<string>::iterator itrL;
    list<string> items1 {“Aberdeen, SD (ABR)”,”A
    itrL = find(items1.begin(), items1.end(),
           “Chicago, IL - O'Hare (ORD)”);

     vector<string>::iterator itrV;
     vector<string> items2 {“Aberdeen, SD (ABR
     itrV = find(items2.begin(), items2.end(),
                “Chicago, IL - O'Hare (ORD)”);

    set<string>::iterator itrS;
    set<string> items3 = {“Aberdeen, SD (ABR)”,
    itrS = find(items3.begin(), items3.end(),
               “Chicago, IL - O'Hare (ORD)”);
```

Finding an item

```
template<class Iter, class Object>
Iter find(Iter start, Iter end, Object search_item)
{
    Iter itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}

int main ()
{
    map<string,string>::iterator itrM;
    map<string,string> items4 = { pair<string,string>(“ABR”,”Aberdeen, SD”), pair<string,string>(“ABI”,"Abilene, TX”)
    pair<const string,string> myPair(“ORD",'“Chicago, IL - O’Hare");
    itrM = find( items4.begin(), items4.end(), myPair);
```

```
map<string,string>::iterator find(map<string, string >::iterator
start, map<string, string>::iterator end, pair<const string, string>
search_item)
{
    map<string, string>::iterator itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
```

Note: There are faster ways for finding an item in a map, set, unordered map, or unordered_set class.

We will discuss these ways later in the semester.

Finding an item

```
class shorterThan
{
private:
   int length;
public:
   shorterThan(int l):length(l){}
   bool operator( )(const student & s)
     { return s.get_name().size()<length;}
};
```

```
class isUpper
{
    public:
      bool operator( )(char ch){ return (‘A’ <= ch) && (ch <= ‘Z’); }
};
```

```
list<char>::iterator find_if(list<char>::iterator itrStart,
                             list<char>::iterator itrPastEnd, isUpper pred)
{
    list<char>::iterator itr;
    for ( itr = itrStart; itr!=itrPastEnd; ++itr)
     if ( pred(*itr) )
           break;
    return itr;
}
```

```
vector<student>::iterator find_if(vector<student>::iterator itrStart,
               vector<student>::iterator itrPastEnd, shorterThan pred)
{
    vector<student>::iterator itr;
    for ( itr = itrStart; itr!=itrPastEnd; ++itr)
     if ( pred(*itr) )
           break;
    return itr;
}
```

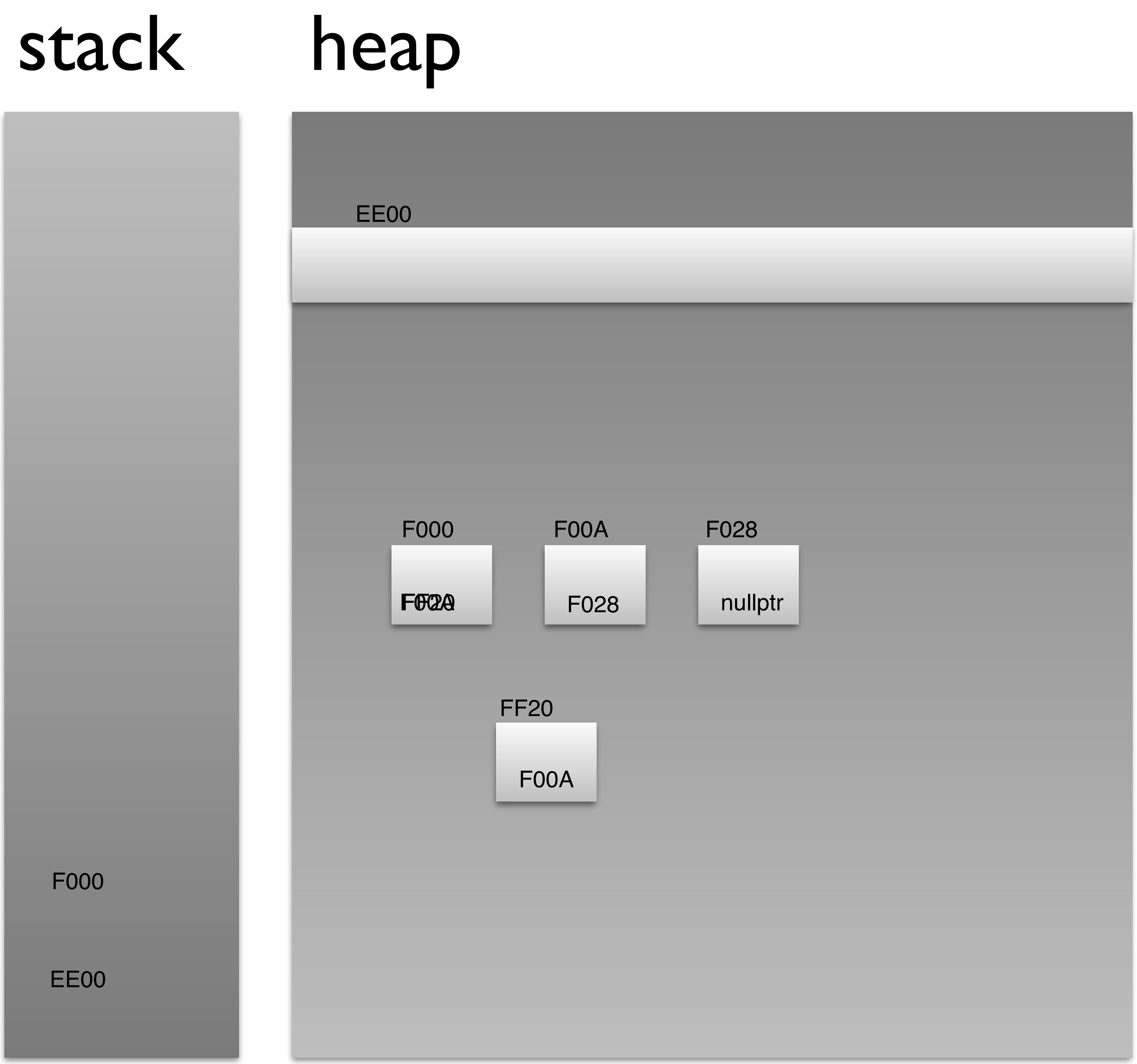
```
template<class Iter, class UnaryPred>
Iter find_if(Iter itrStart, Iter itrPastEnd, UnaryPred pred
{
    Iter itr;
    for ( itr = itrStart; itr!=itrPastEnd; ++itr)
     if ( pred(*itr) )
          break;
    return itr;
}
```

```
int main ()
{
  list<char>::iterator itrL;
  list<char> items1 {‘a’,’b’,’C’,’d’,’e’};
  itrL = find_if(items1.begin(), items1.end(), isUpper( ))


  vector<student>::iterator itrV;
  vector<student> items2;
  // code to enter the students names
  itrV = find_if(items2.begin(), items2.end(),
                    shorterThan(4));
```

Sequence containers

$A_1,A_2,A_3,\ldots,A_n$

Storing the list…



Aberdeen, SD (ABR)
Abilene, TX (ABI)
Adak Island, AK (ADK)
Akiachak, AK (KKI)
Akiak, AK (AKI)
Akron/Canton, OH (CAK)
Akuton, AK (KQA)
Alakanuk, AK (AUK)
Alamogordo, NM (ALM)
Alamosa, CO (ALS)
Albany, NY (ALB)
Albany, OR - Bus service (CVO)
Albany, OR - Bus service (QWY)
Albuquerque, NM (ABQ)
Aleknagik, AK (WKK)
Alexandria, LA (AEX)
Allakaket, AK (AET)
Allentown, PA (ABE)
Alliance, NE (AIA)
Alpena, MI (APN)
Altoona, PA (AOO)
Amarillo, TX (AMA)
Ambler, AK (ABL)
Anaktueuk, AK (AKP)
Anchorage, AK (ANC)
Angoon, AK (AGN)
Aniak, AK (ANI)
Anvik, AK (ANV)
Appleton, WI (ATW)
Arcata, CA (ACV)
Arctic Village, AK (ARC)

Vectors (and Strings)

- Arrays are not “first class objects” – cannot do “the usual operations” such as =, ==
- STL provides vectors and strings which has “the usual operations” such as =, ==
- class vector has
  - indexing v[] (starts at 0; NO range checking)
  - operator=
  - size()
  - resize() [Expensive]
  - push_back() [doubles capacity if necessary]
- use call by reference or call by const reference to pass vectors as parameters
- Implemented by wrapping the array in a class!

Thus hiding the complications from the user.

Implementation of a Vector Class

Simpler than STL implementation

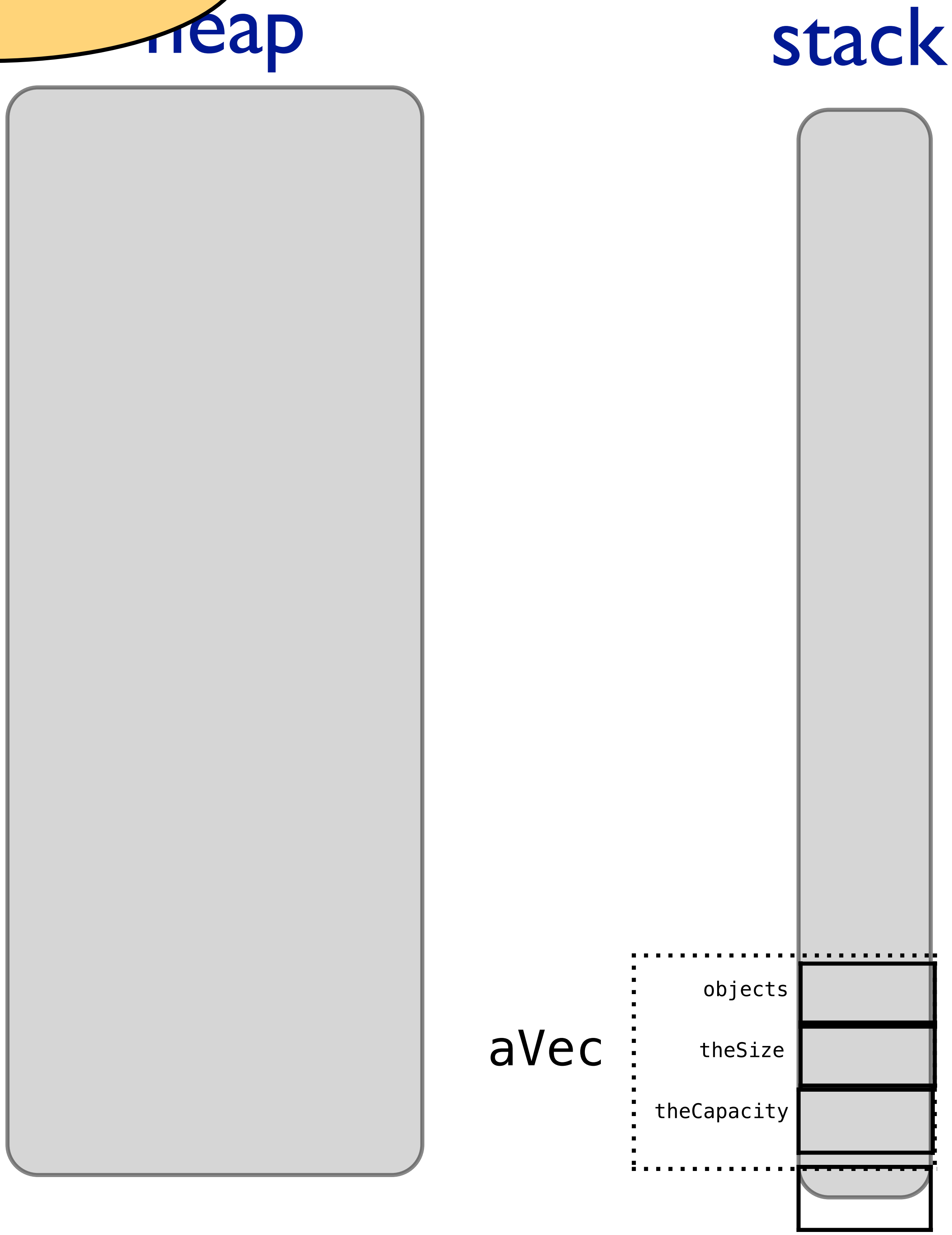
Our class is called **V**ector class to distinguish it from the STL vector class.

How would you create a vector class?

How would you create a vector class?

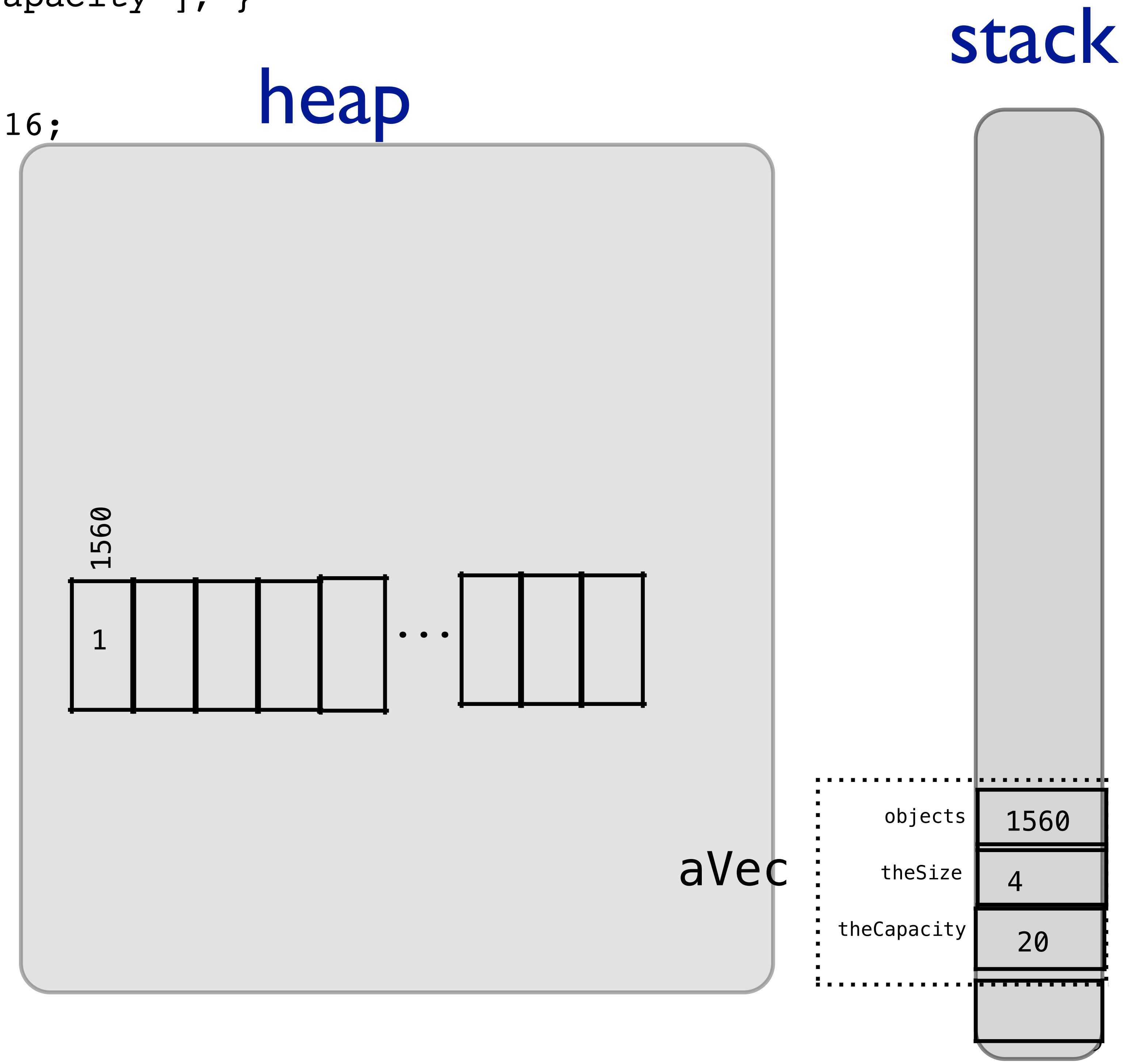
To focus on the idea/method being discussed, the other methods will be replaced by ...

```
template <class Object>
class Vector
{
  public:
   ...

  private:
    int theSize;
    int theCapacity;
    Object * objects;
};


 int main{
    Vector<int> aVec(4);

    return 0;
 }
```



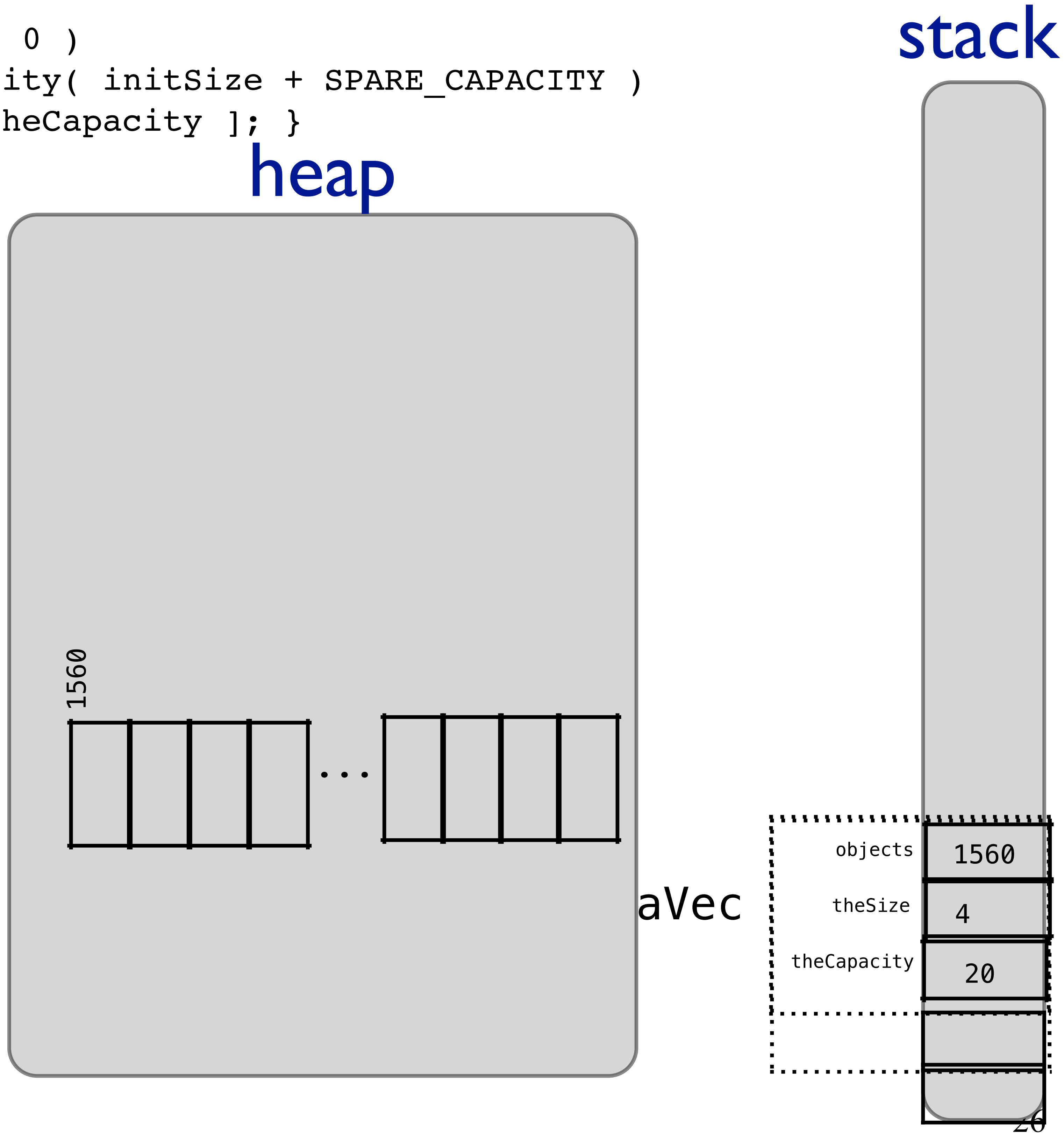
How would you create a vector class constructor?

```
template <class Object>
class Vector
{
  public:

    explicit Vector( int initSize = 0 )
    : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
          { objects = new Object[ theCapacity ]; }

    ...
    static const int SPARE_CAPACITY = 16;

  private:
     int theSize;
     int theCapacity;
     Object * objects;
};


 int main{

    Vector<int> aVec(4);
     aVec[0] = 1;
     a~~Vec[4] = 1~~;
     aVec.push_back(21)
    return 0;
 }
```



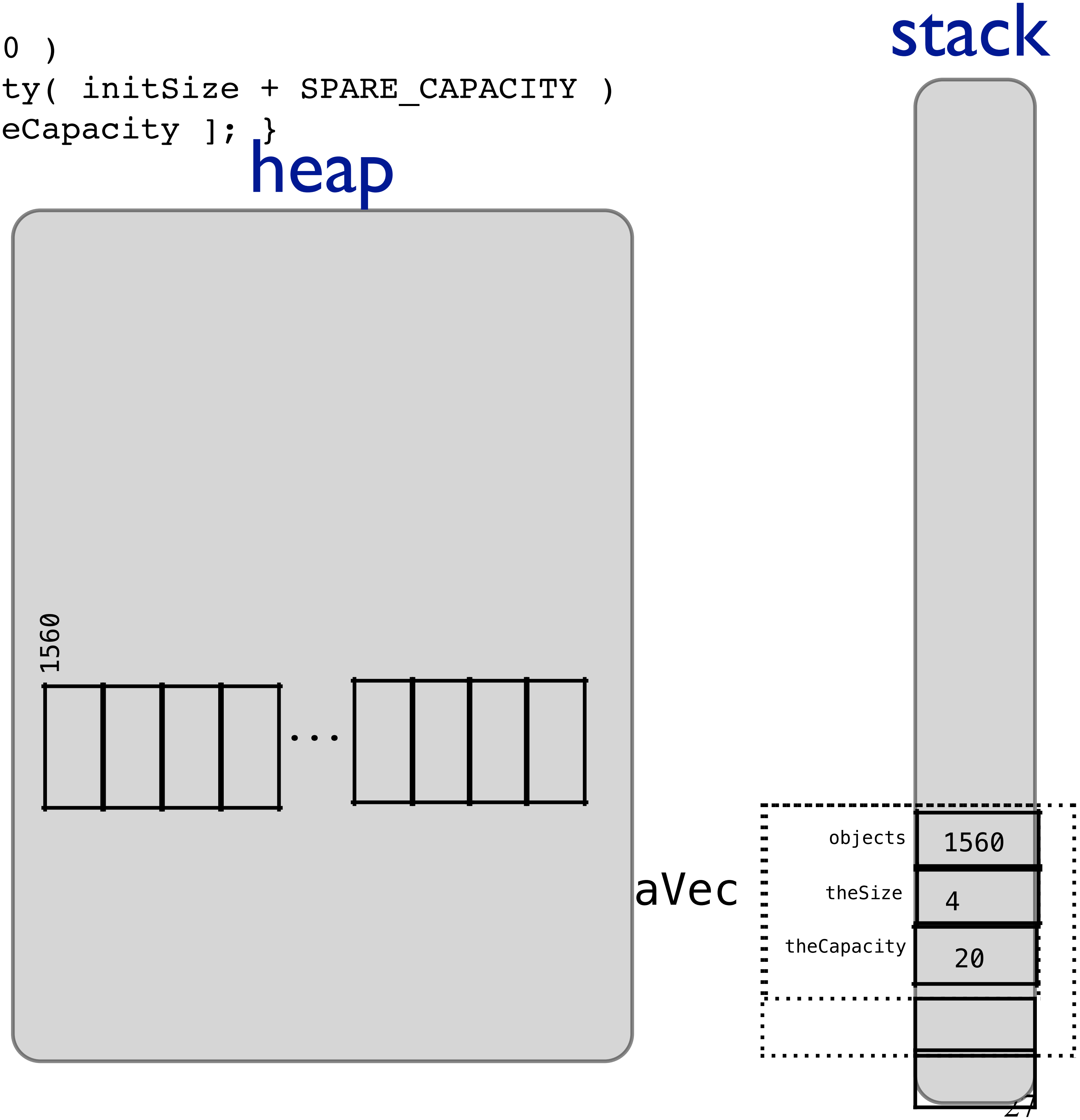
Do we need to write a destructor?

```
template <class Object>
class Vector
{
  public:
     explicit Vector( int initSize = 0 )
     : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
   ...      { objects = new Object[ theCapacity ]; }


  private:
    int theSize;
    int theCapacity;
    Object * objects;
};

void Silly()
{
   Vector<int> a(4);
   return;
}

int main{

   silly();
   return 0;
}
```



How would you create a vector class destructor?

```
template <class Object>
class Vector
{
  public:
    explicit Vector( int initSize = 0 )
    : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
          { objects = new Object[ theCapacity ]; }
    ...
     ~Vector( )
       { delete [ ] objects; }
    ...

  private:
     int theSize;
     int theCapacity;
     Object * objects;
};

void Silly()
{
    Vector<int> a(4);
    return;
}

int main{

    silly();
    return 0;
}
```



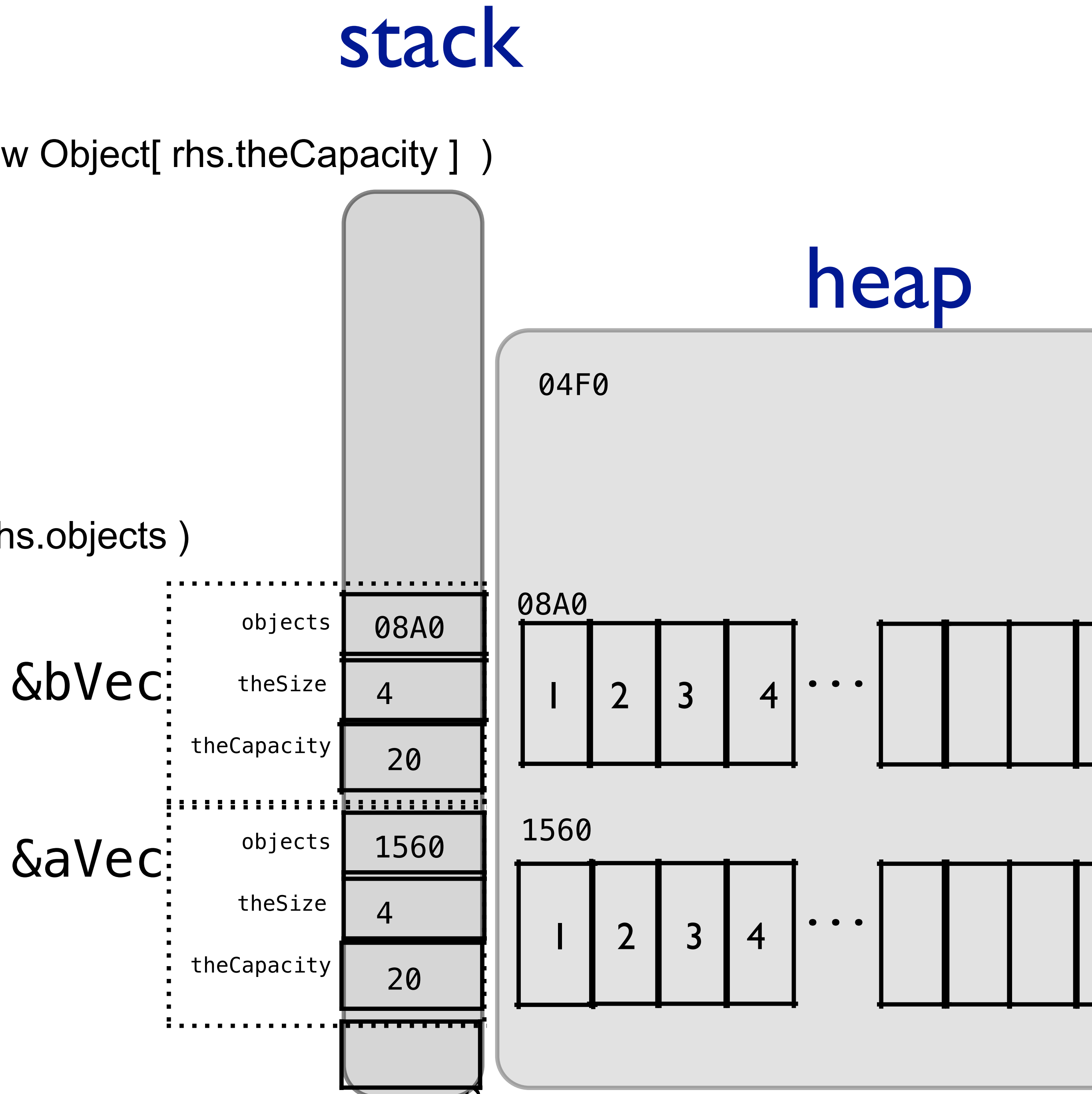
How would you create a vector class copy constructor and move constructor?

```
template <class Object>
class Vector
{
  public:
     Vector( const Vector & rhs );
     Vector( Vector && rhs );
     …
  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

```
template <class Object>
Vector<Object>::Vector( const Vector & rhs )
:theSize(rhs.theSize),theCapacity(rhs.theCapacity), objects( new Object[ rhs.theCapacity ]  )
{
   for( int k = 0; k < theSize; ++k )
      objects[ k ] = rhs.objects[ k ];
}
```

```
template <class Object>
Vector<Object>::Vector( Vector && rhs )
:theSize(rhs.theSize),theCapacity(rhs.theCapacity), objects( rhs.objects )
{
     rhs.objects = nullptr;
     rhs.theSize = 0;
     rhs.theCapacity=0;
}
```

```
int main() {
    Vector<int> avec = {1, 2, 3, 4};
    Vector<int> bVec(aVec);
    Vector<int> cVec = Vector<int>( 2 );
}
```



How would we write the method to:

```
template <class Object>
class Vector
{
  public:
  ...

  int capacity( ) const
    { return theCapacity; }
  int size( ) const
    { return theSize; }

  bool empty( ) const
     { return size( ) == 0; }

 Object & operator[]( int index )
     { return objects[ index ]; }

 const  Object & operator[]( int index ) const
      { return objects[ index ]; }



  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

determine the capacity ?

determine the size ?

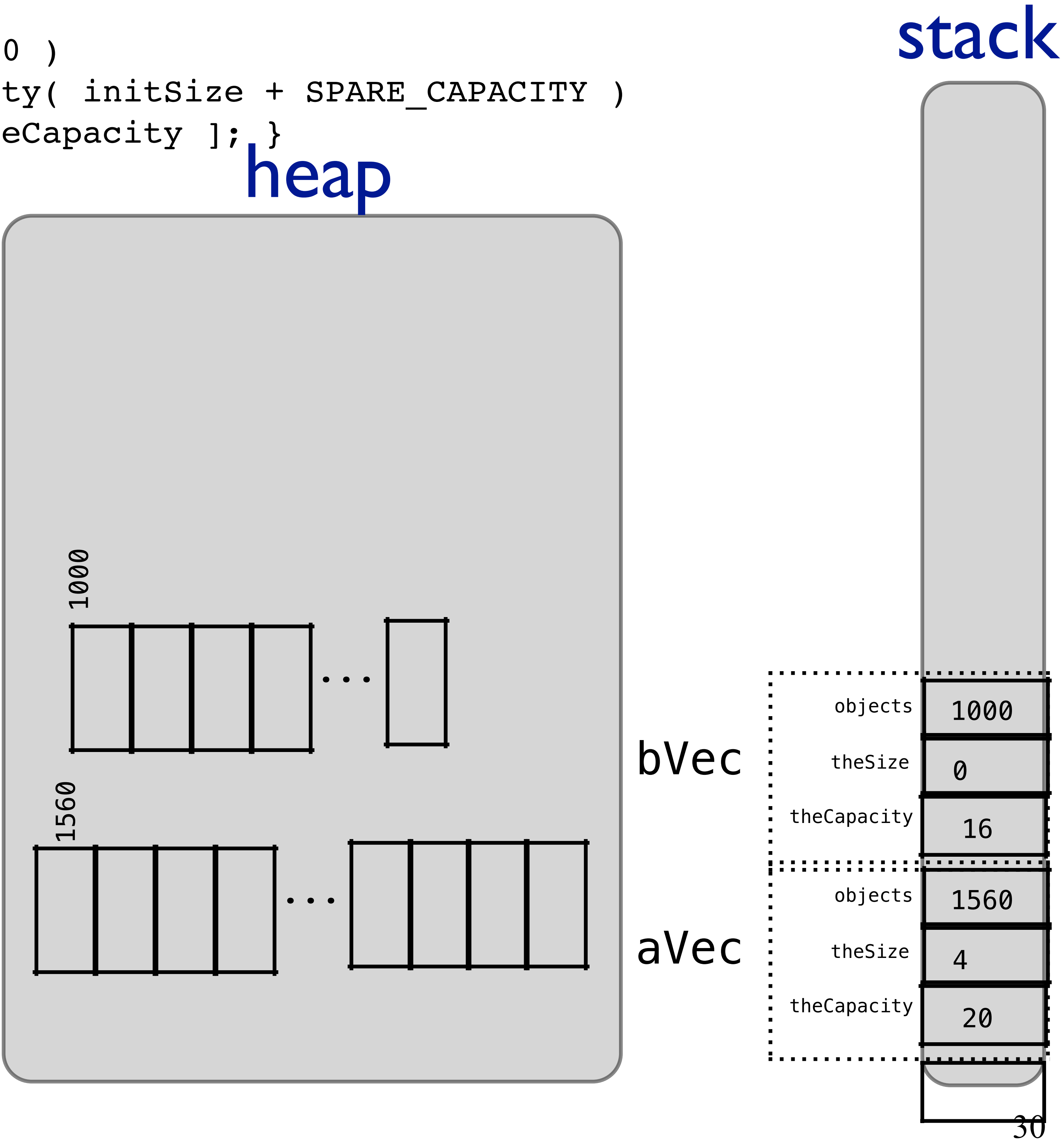
determine if the vector is empty?

return the ith item?

Do we need to create an operator=?

```
template <class Object>
class Vector
{
  public:
    explicit Vector( int initSize = 0 )
    : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
          { objects = new Object[ theCapacity ]; }

   ...
  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

```
int main{

   Vector<int> aVec(4);
   Vector<int> bVec;
   bVec = aVec;
   return 0;
}
```

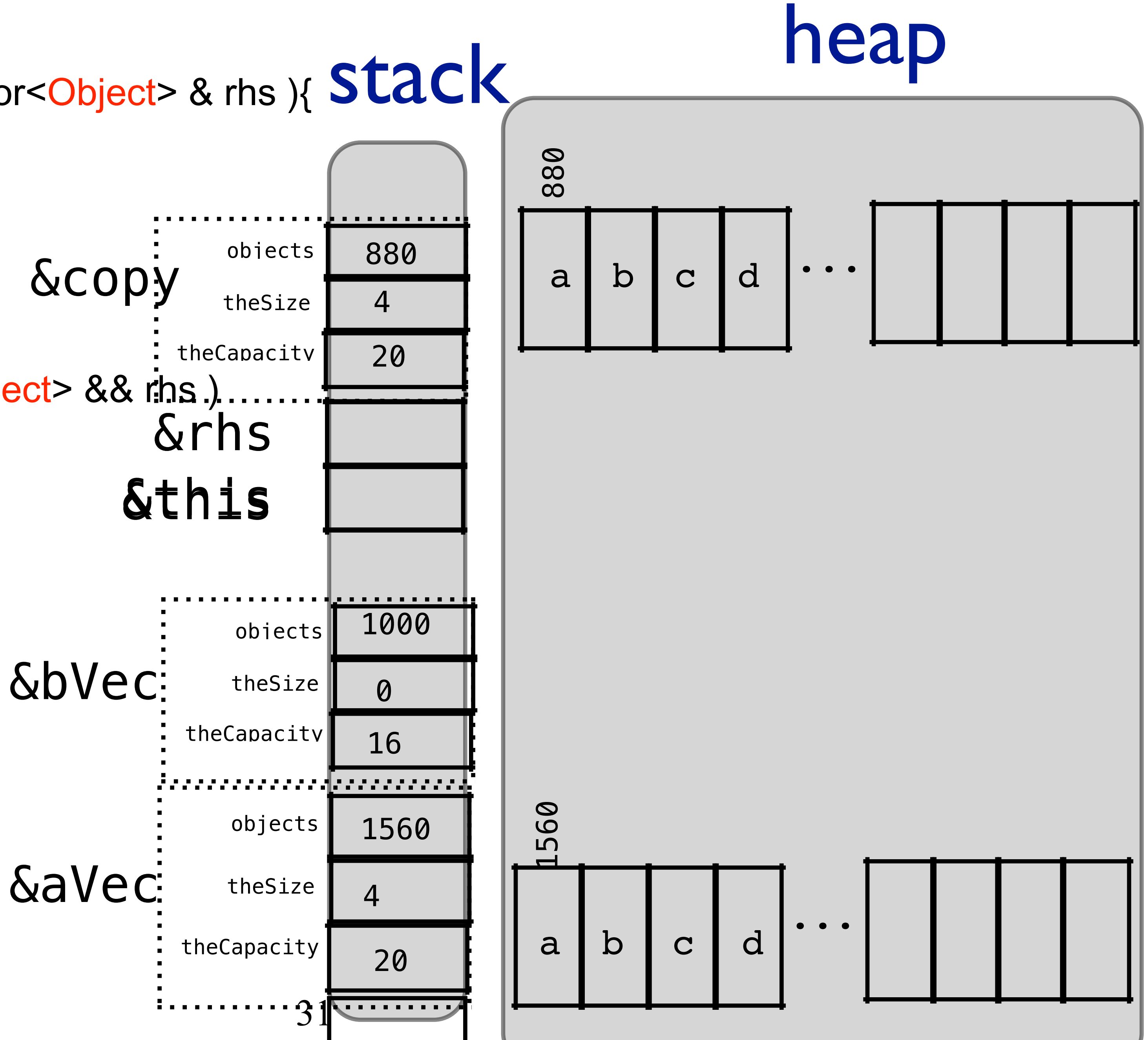


Operator= Method

```
template <class Object>
class Vector{
  public:
    ...
   Vector & operator= ( Vector && rhs );
    Vector & operator= ( const Vector & rhs );
  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

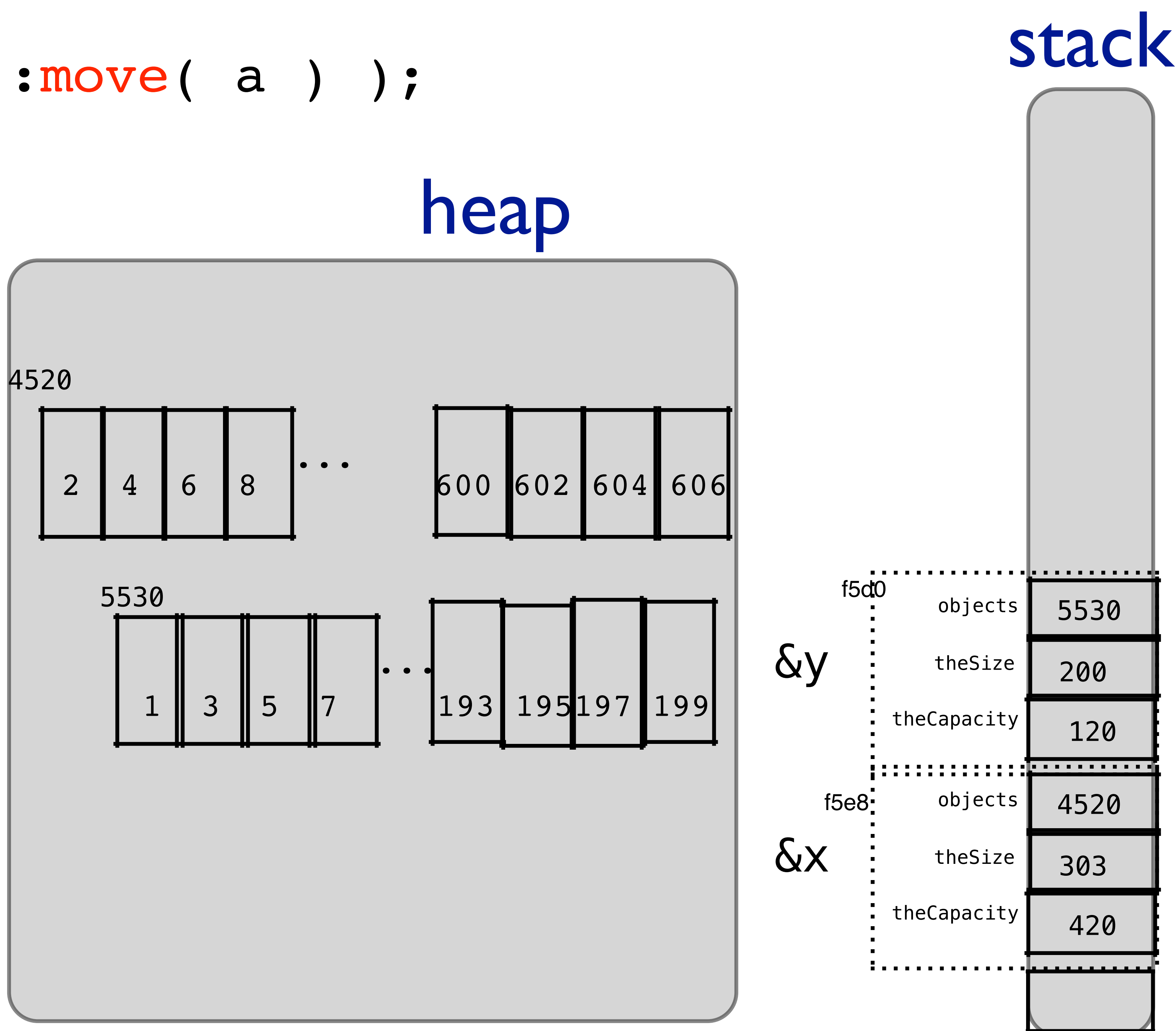
```
template <class Object>
Vector<Object> & Vector<Object>::operator=( const Vector<Object> & rhs ){
        Vector copy = rhs;
        std::swap( *this, copy );
        return *this;
}
template <class Object>
Vector<Object> & Vector<Object>::operator=( Vector<Object> && rhs )
{
        std::swap( theSize, rhs.theSize );;
        std::swap( theCapacity, rhs.theCapacity );
        std::swap( objects, rhs.objects );

        return *this;
}
```

```
Vector<char> aVec = { ’a’, ’b’, ’c’, ’d’ }
Vector<char> bVec;
 bVec = aVec;

 bVec =Vector<char>(3);
```



Using the move constructor and the Move assignment operator

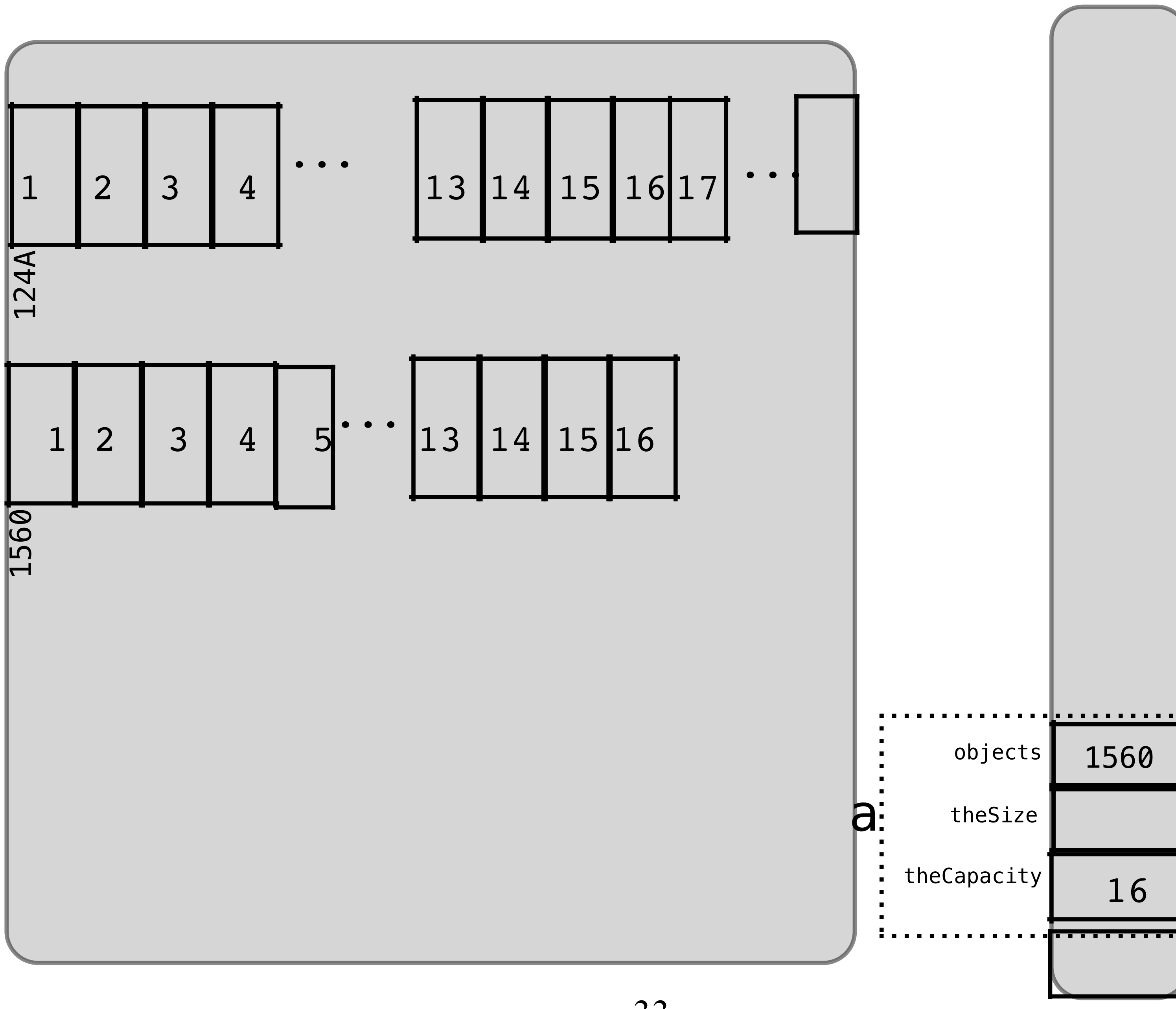
```
void swap(vector<int> & a, vector<int> & b)
{
    vector<int> tmp(std::move( a ) );
    a = std::move(b);
    b = std::move(tmp);
}

int main( )
{
   vector<int> x(303);
   vector<int> y(200);
   // code …
   swap( x, y );
```



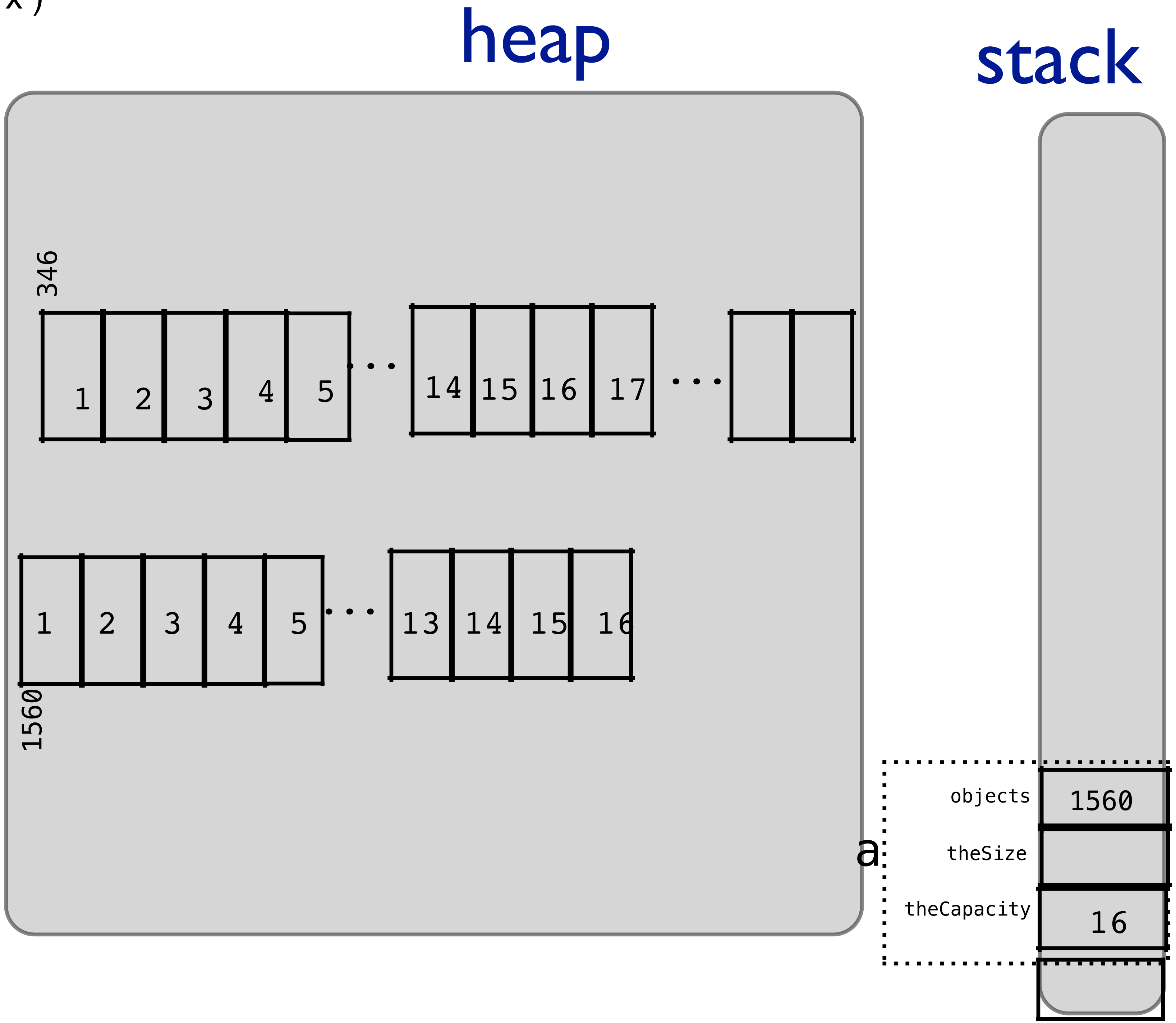
push_back and reserve methods for the Vector class

```
template <class Object>
class Vector
{
  public:

    explicit Vector( int initSize = 0 )
    : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
          { objects = new Object[ theCapacity ]; }

    ...

    void reserve( int newCapacity );

    void push_back( const Object & x );
    void push_back( Object && x );

  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

```
template <class Object>
void Vector<Object>::push_back( const Object & x )
{
    if( theSize == theCapacity )
      reserve( 2 * theCapacity + 1 );
   objects[ theSize++ ] = x;
}
int main{

   Vector<int> aVec;
   for (int i = 1; i < 18; ++i)
     aVec.push_back(i);
}
```



```
template <class Object>
void Vector<Object>::push_back( Object && x )
{
     if( theSize == theCapacity )
        reserve( 2 * theCapacity + 1 );
    objects[ theSize++ ] = std::move( x );
}
template <class Object>
void Vector<Object>::push_back( const Object & x )
{
     if( theSize == theCapacity )
        reserve( 2 * theCapacity + 1 );
    objects[ theSize++ ] = x;
}

template <class Object>
void Vector<Object>::reserve( int newCapacity )
{

    if ( newCapacity <= theCapacity ) return;
    // never decrease the capacity

    Object* p = new Object[ newCapacity];
    for( int k = 0; k < theSize; k++ )
        p[ k ] = std::move( objects[ k ] );

    delete [ ] objects;
    objects = p;
    p = nullptr;
    theCapacity = newCapacity;
}
```



Cost of using the method push_back()
Amortized Analysis

Amortized Analysis:

Used to find worst case bounds when analyzing algorithms, by looking over the entire sequence of operations, and finding the average cost of an operation. Even if a couple of operations are very expensive, if they are rare then the average cost may be much less.

Amortized Analysis shows why the vector method push_back() takes O(1) time:

First, we simplify the situation by starting with capacity = 1, and every time we resize the array, we double the size of the capacity.

When using the vector method push_back(), the number of times we double the array when adding n items is at most log(n).
The time the method push_back() takes when the array is not doubled is O(1).

If the array starts with 1 capacity,
when the array is doubled, the first time it moves 1 object,
the second time it moves 2 objects, the third time it moves 4 objects, ..., the (log(n)-1)'th time it moves 2^(log(n)-1) = n/2 objects
The sum of all the items moved is: 1 + 2 + 8 + 16 + + n/2 = O(n)

iterate |ˈitəˌrāt|

verb [with obj.]

perform or utter repeatedly.

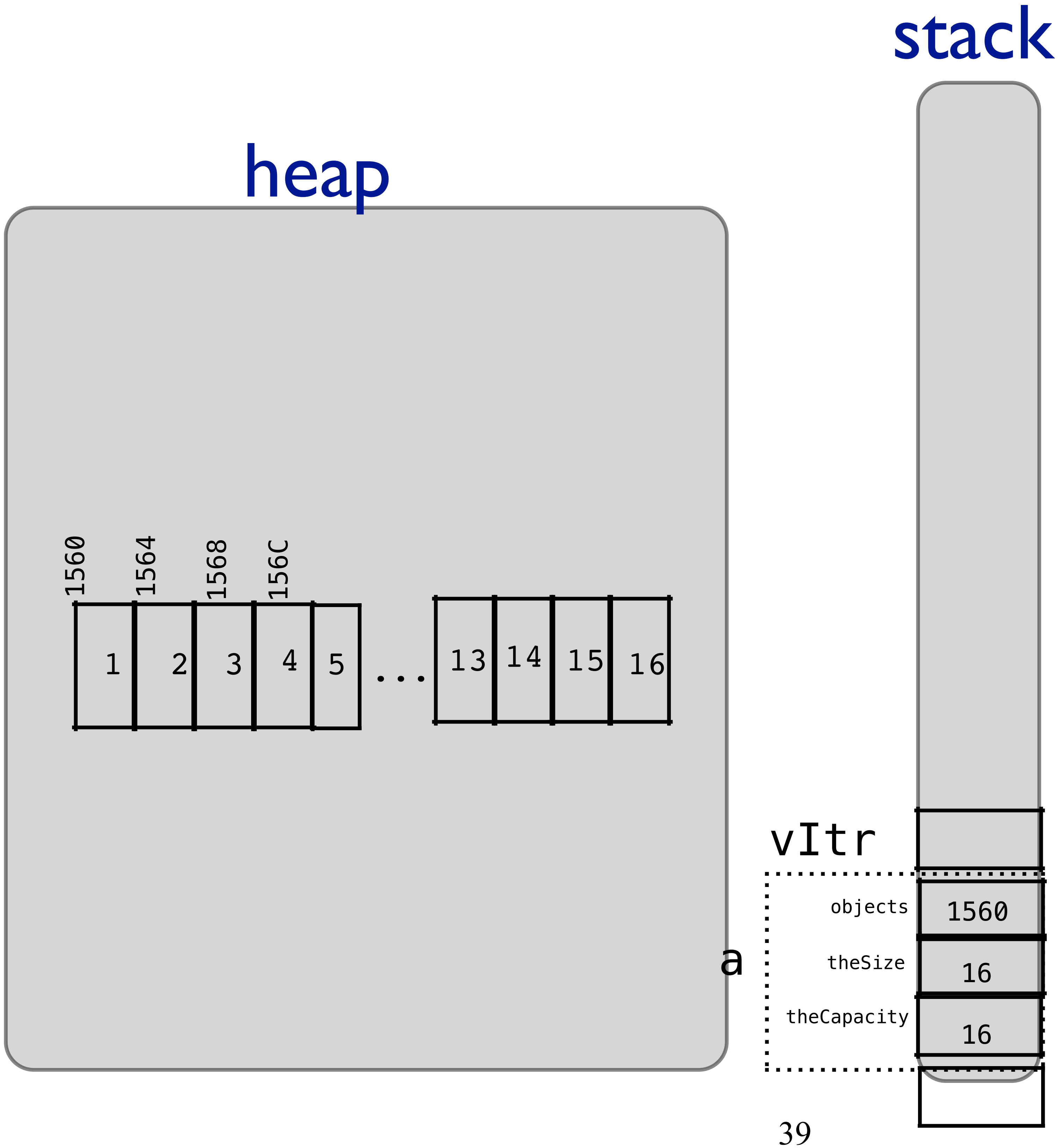
• [no obj.] make repeated use of a mathematical or computational procedure, applying it each time to the result of the previous application; perform iteration.

From the dictionary on my computer:)

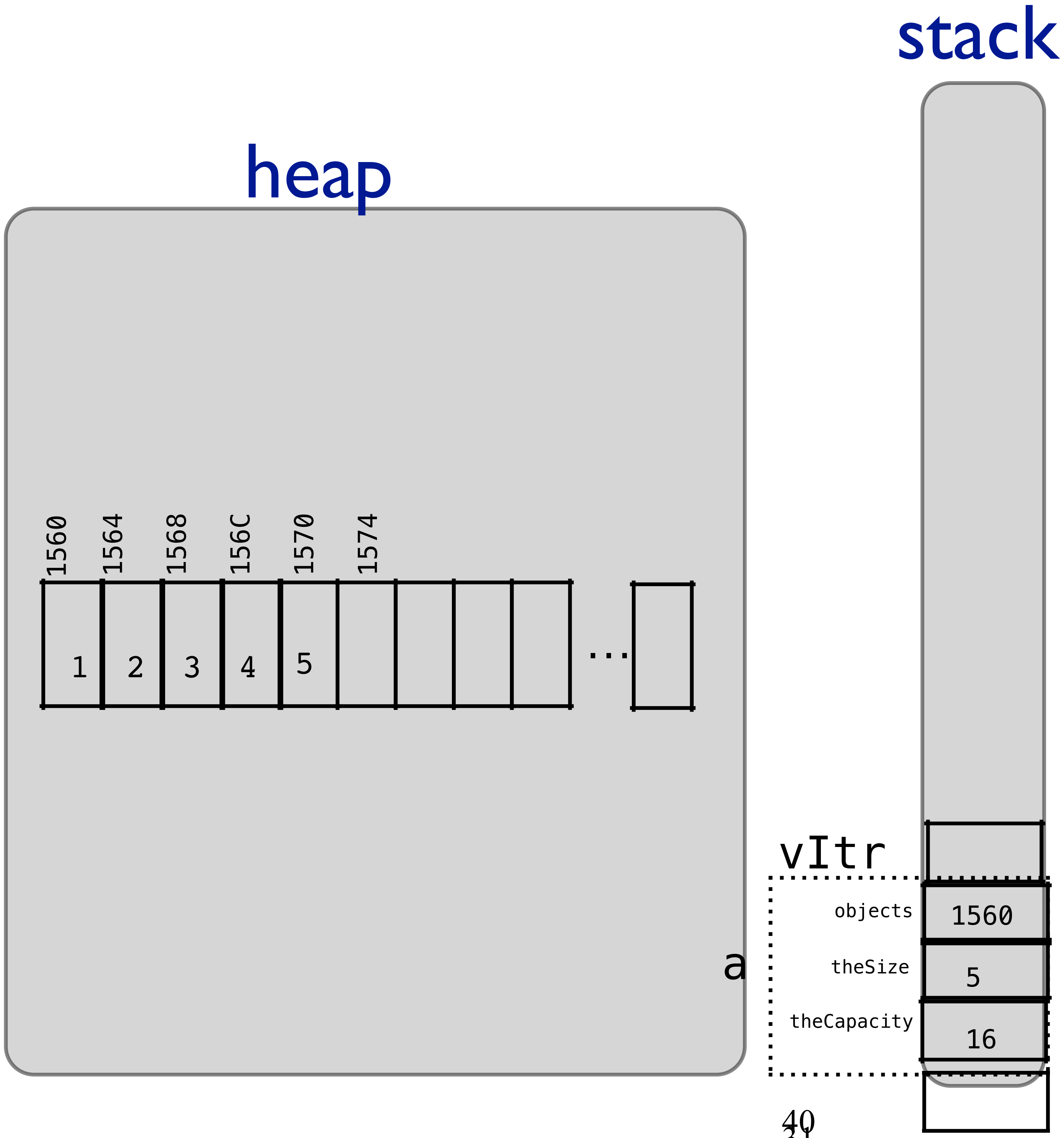
Creating a Vector Iterator

A generic Way to traverse the vector using an iterator

```
template <class Object>
class vector
{
  public:
       ...
    // Iterator: not bounds checked
     typedef Object * iterator;

     iterator begin( )
      { return &objects[ 0 ]; }
     iterator end( )
      { return &objects[ size( ) ]; }

  private:
    int theSize;
    int theCapacity;
    Object * objects;
};

int main(void)
{   vector<int> a;
    a.push_back(1);
   a.push_back(2);
     ...
    a.push_back(16);
    vector<int>::iterator vItr;
   vItr = a.begin( );
   ++vItr;
   vItr += 2;
   cout << *vItr << endl;
```

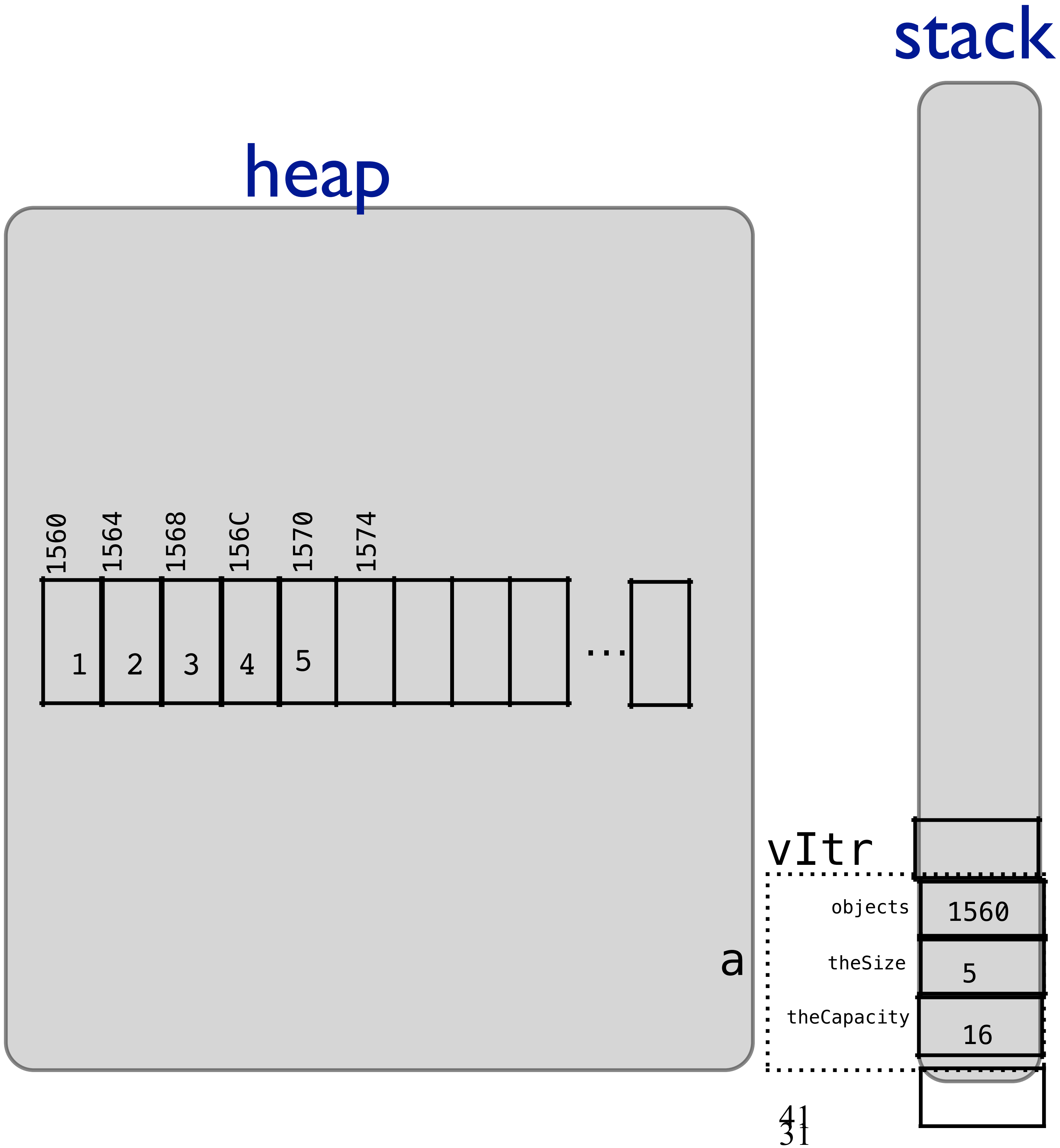


```
int main(void)
{

    Vector<int> a;

    a.push_back(1);
   a.push_back(2);
      ...
    a.push_back(5);

    Vector<int>::iterator vItr = a.begin( );


    for(  ; vIter != a.end( ); ++vIter)
    {
        cout << *vIter;
    }



    int mid = (a.end( ) - a.begin( ))/2;

    cout << *(a.begin( ) + mid) << endl;
```



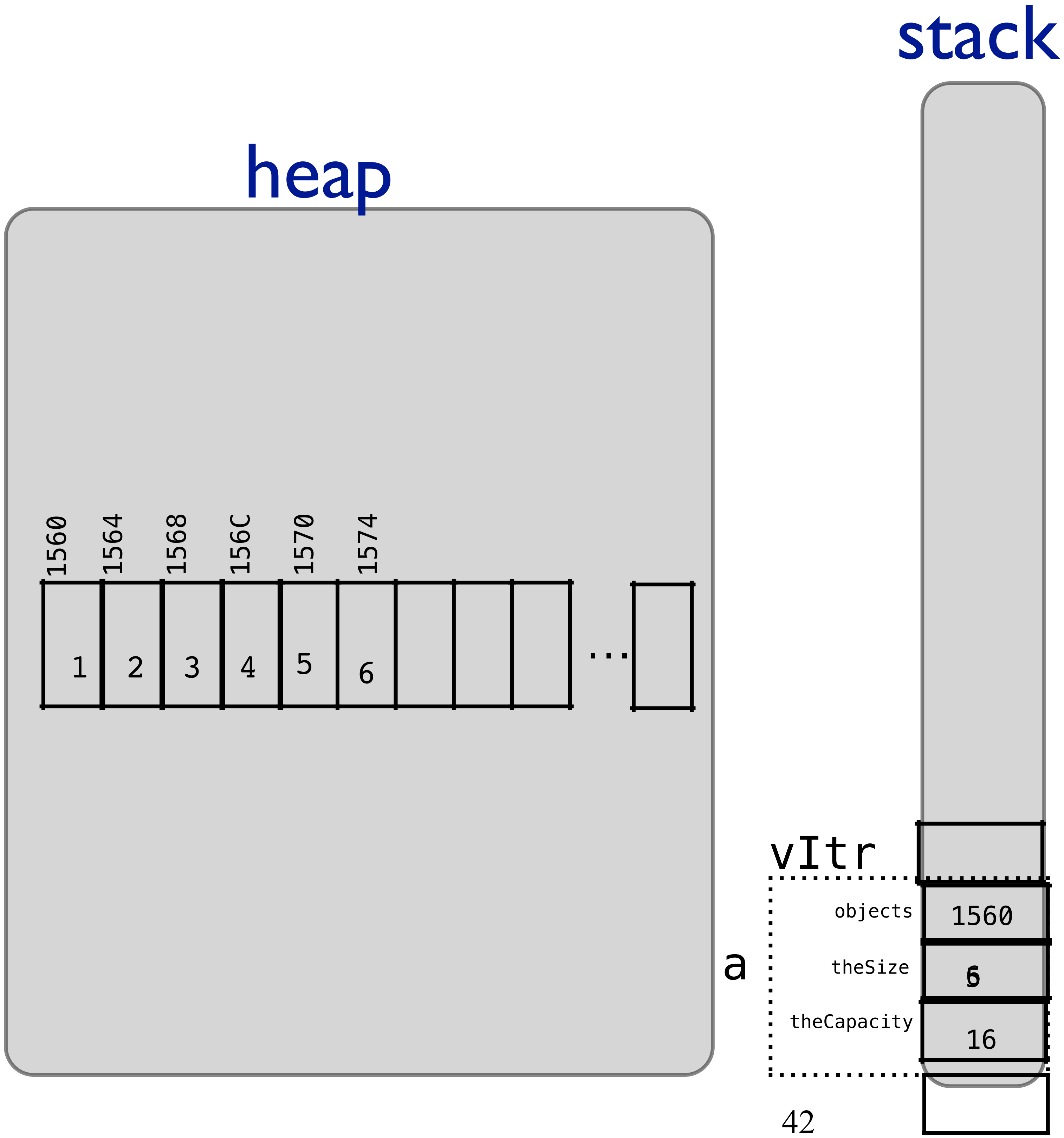
```
int main(void)
{
    Vector<int> a;
    a.push_back(1);
    a.push_back(2);
      ...
    a.push_back(5);
    Vector<int>::iterator vItr = a.end( );-1;

   for(  ; vIter != a.begin( ); --vIter)
   {
       cout << *vIter;
   }

   cout << *vIter << endl;
```

oops!!!



What is a.end( ) - a.begin( )?

```
a.push_back(6);

mid = (a.end() - a.begin())/2;
cout << *(vIter + mid) << endl;
```



const_iterator

```
template <class Object>
class Vector
{
  public:
      ...
      // Iterator: not bounds checked
         ...
       typedef Object * iterator;
       typedef const Object * const_iterator;

       iterator begin( )
         { return &objects[ 0 ]; }
      const_iterator begin( ) const
       { return &objects[ 0 ]; }

       iterator end( )
         { return &objects[ size( ) ]; }
      const_iterator end( ) const
       { return &objects[ size( ) ]; }
  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

More ways to enter the numbers 1 to 100 into a vector using an iterator

```
Vector<int> vec_of_int(100);
Vector<int>::iterator vecItr;

for ( start = 1, vecItr = vec_of_int.begin() ; vecItr != vec_of_int.end(); ++vecItr)
{
 *vecItr = start;
 start = start+1;
}
```

```
 Vector<int> vec_of_int(100);
 Vector<int>::iterator vecItr;
int start;
for ( start = 1, vecItr = vec_of_int.begin() ; vecItr != vec_of_int.end(); ++vecItr)
 *vecItr = start++;
```

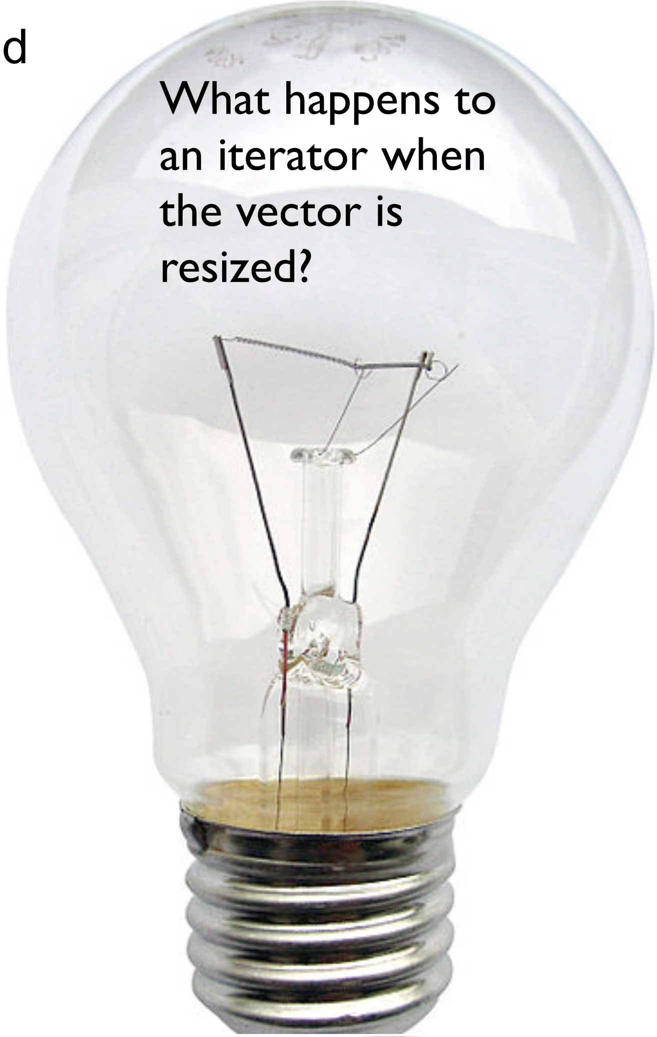
```
Vector<int> vec_of_int(100);
Vector<int>::iterator vecItr= vec_of_int.begin();
int start = 1;
while (vecItr != vec_of_int.end())
{
 *vecItr++ = start++;
}
```

#include<vector>

#include<list>

vectors - Random Access Iterator

v.push_back(value)	O(1) amortized
v.pop_back( )	O(1)
v.back( )	O(1)
v.front( )	O(1)
v[i]	O(1)
v.erase(v.begin(),v.end())	O(n)
v.erase(iterator)	O(n)
v.clear()	O(n)
v.size()	O(1)
v.insert(iterator,value)	O(n)
v.begin()	O(1)
v.end()	O(1)
v.resize(n) or v.resize(n,value)	O(n)
v.reserve(n)	O(n)
v1 = v2	O(n)
v1 = std::move(v2)	O(1)
v.capacity	O(1)



What happens to an iterator when the vector is resized?

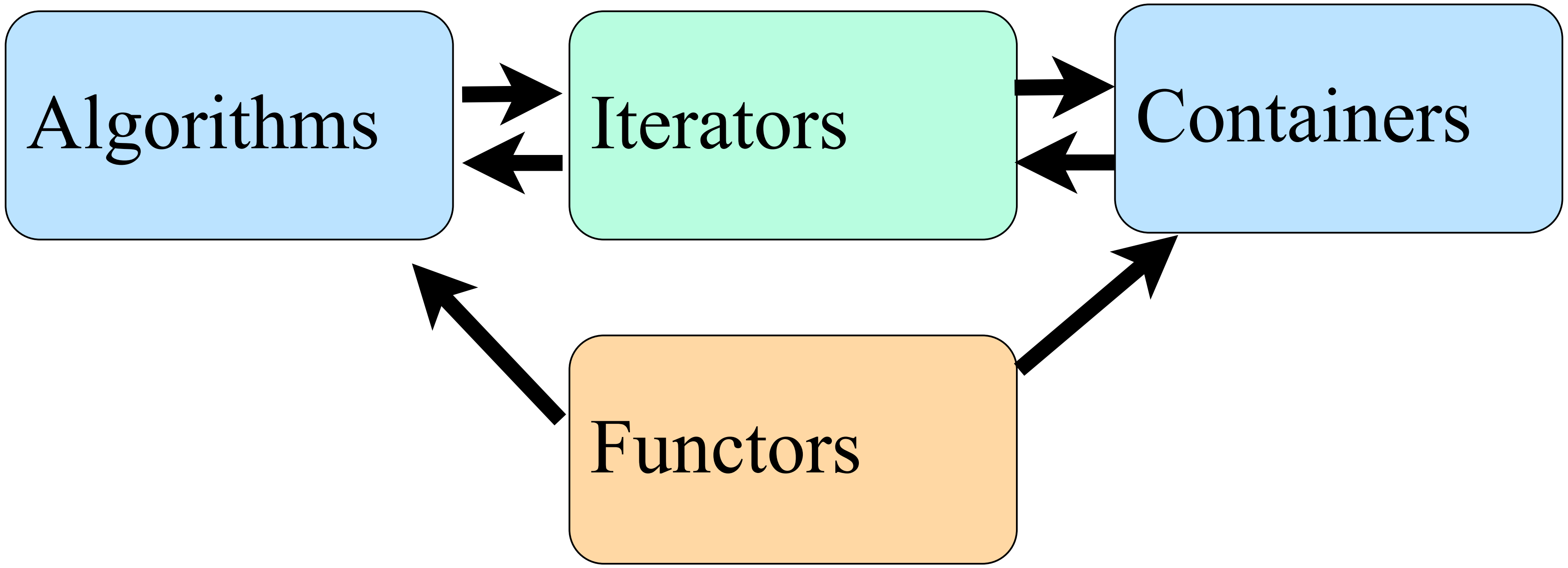
Unlike a vector, a list does not use more space than needed. A list is useful to insert and delete without moving existing elements

Note: all these times do not include constructor/destructor times which many vary according to the type

This list is not complete
Check expert-level resource for more info.

STL

Standard Template Library



Dictionary ADT

natural language dictionary
router tables
page tables
symbol tables
phone directoires
Web Pages
Student Recor

focus on data storage and retrieval

Dictionaries (ADT), SET (ADT)

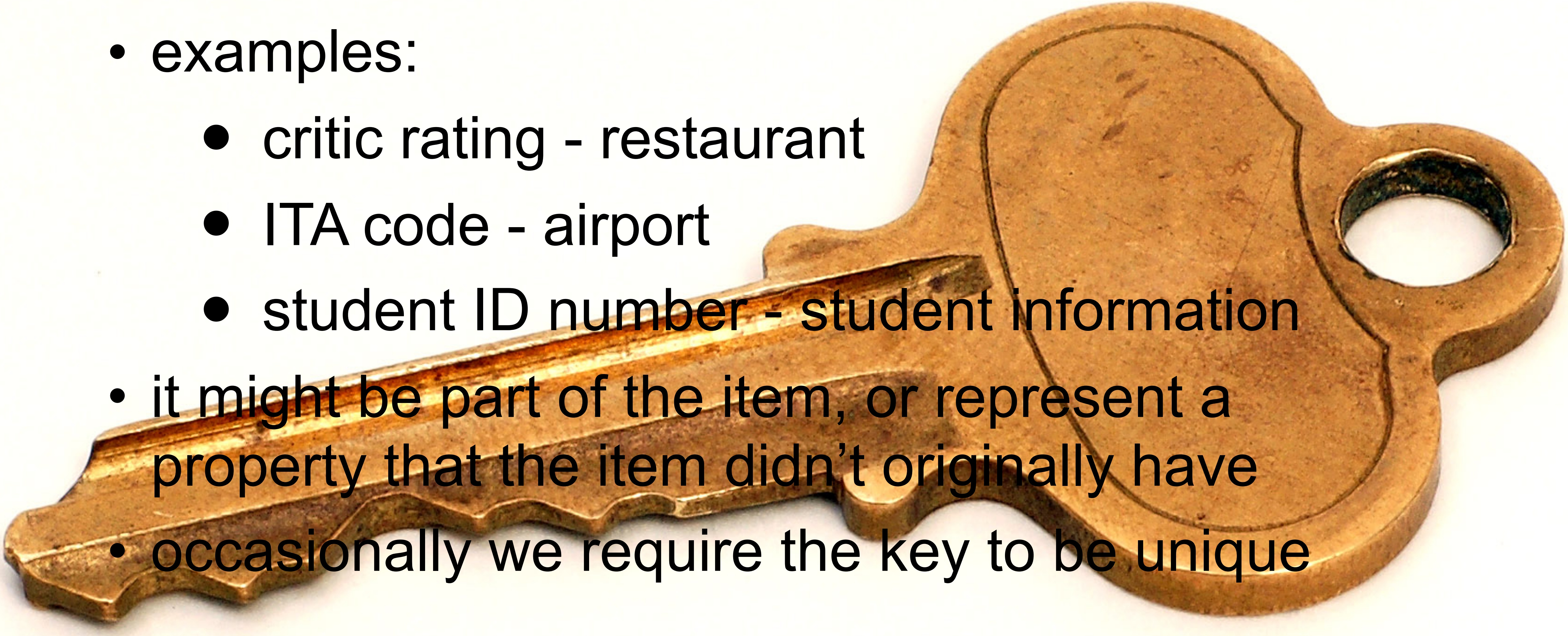
- Data structures that supports **find**, **insert**, **delete**
- Many applications
- Item referred to by a **key**. In a dictionary, keys have records associated with them
- Many choices for implementing dictionary/set

  Programmer must choose best one, based on how the program will use the dictionary

  –static versus dynamic

  –many find operations versus few find operations

keys



object assigned to *identify* an item or *rank* an item

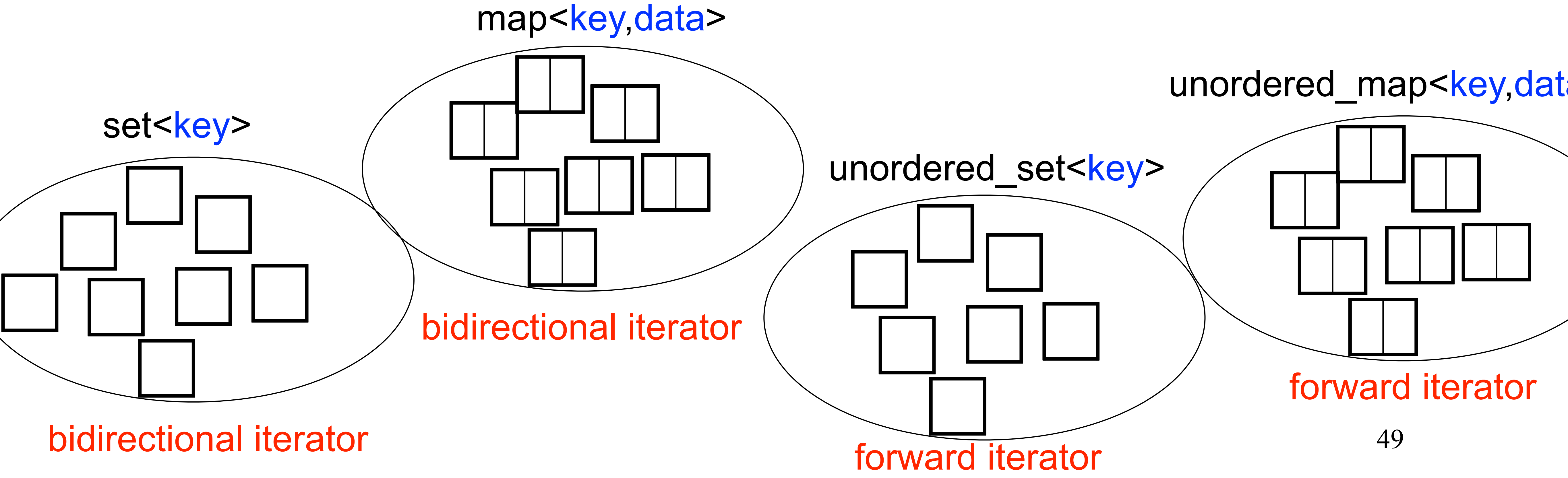
- examples:
  - critic rating - restaurant
  - ITA code - airport
  - student ID number - student information
- it might be part of the item, or represent a property that the item didn't originally have
- occasionally we require the key to be unique

keys with ≤ have a total order: **reflexive**, **antisymmetric**, **transitive**

https://en.wikipedia.org/wiki/Key_(lock)#/media/File:Standard-lock-key.jpg

Ordered/Unordered Associative containers

- Can’t insert element into particular position (order of insertion doesn't matter)
- Elements stored have key and (maybe) value, access by key
  - map, unordered_map: key, value pair. Efficient access by key
    - E.g.   *Key* is Social Security Number, *Value* is employee data
  - set, unordered_set: Elements stored by key, but no value, access by key



set<key>

bidirectional iterator

map<key,data>

bidirectional iterator

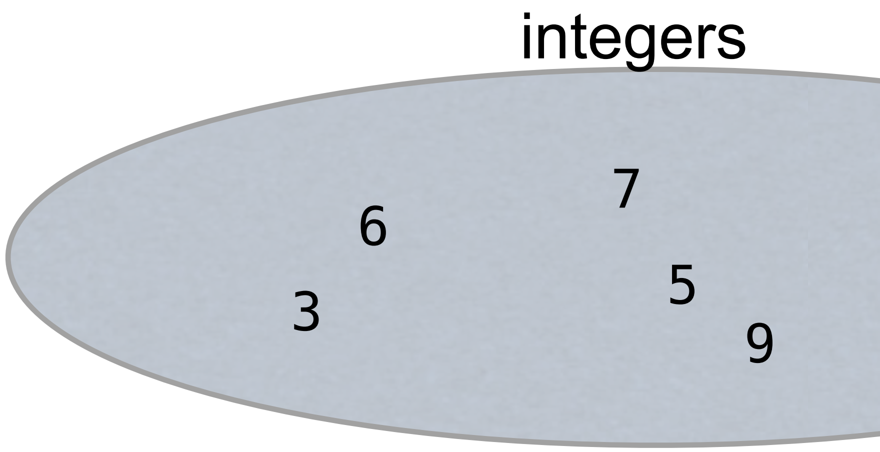
unordered_set<key>

forward iterator

unordered_map<key,dat

forward iterator

Some set and unordered_set members



pair<iterator, bool> insert(const value_type& x)

// if x is in the set, returns false

// else inserts it and returns <iterator to x, true>

size_type erase(const key_type& k);

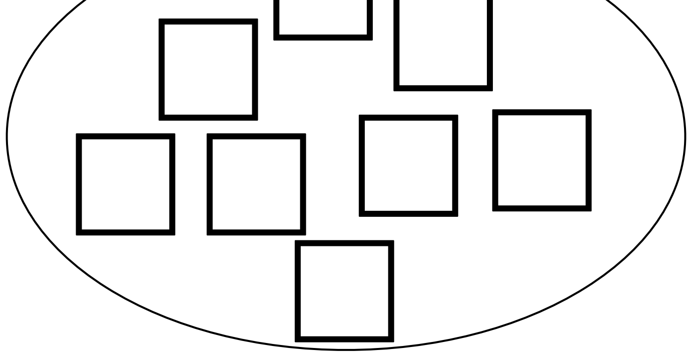
// removes element whose key is k and returns

// number of elements removed (0 or 1)

void erase(iterator pos);

iterator find(const key_type& k) const ;

```
unordered_set<int> setOfIntegers;
unordered_set<int>::iterator itrS;

setOfIntegers.insert(5);
setOfIntegers.insert(5);
setOfIntegers.erase(5);
setOfIntegers.insert(3);
setOfIntegers.insert(6);
setOfIntegers.insert(7);
setOfIntegers.insert(9);

itrS = setOfIntegers.find(6);
if (itrS == setOfIntegers.end())
    cout << ”6 not in set”;
else
    cout << ”6 in set”;
```



set<key, key compare>

bidirectional iterator

- Unique key (no duplicates)
- Supports insertion, deletion, and find in O(log n) time
- range [first, last) is sorted
- How's it implemented?
  - vector or linked list can't meet all the time bounds
  - Answer: …

unordered_set<key, key compare>

forward iterator

- Unique key (no duplicates)
- Supports insertion, deletion, and find in O(1) time on average
- range [first, last) is unsorted
- How's it implemented?
  - vector or linked list can't meet all the time bounds
  - Answer: …

Pair

```
template<class Type1, class Type2>
struct pair
{
public:
     Type1 first;
     Type2 second;
     pair (const Type1 & f=Type1(),const Type2 & s = Type2())
     : first(f), second( s){}
};

 pair<string, string> airportCode1;
 airportCode1.first = “ABR”;
 airportCode1.second= “Aberdeen, SD”;
 cout << airportCode1.first<< airportCode1.second;

 pair<string, string> airportCode2;
 airportCode2.first = “ABI”;
 airportCode2.second= “Abilene, TX”;
 cout << airportCode2.first<< airportCode2.second;
```

Aberdeen, SD (ABR)
Abilene, TX (ABI)
Adak Island, AK (ADK)
Akiachak, AK (KKI)
Akiak, AK (AKI)
Akron/Canton, OH (CAK)
Akuton, AK (KQA)
Alakanuk, AK (AUK)
Alamogordo, NM (ALM)
Alamosa, CO (ALS)
Albany, NY (ALB)
Albany, OR - Bus service (CVO)
Albany, OR - Bus service (QWY)
Albuquerque, NM (ABQ)
Aleknagik, AK (WKK)
Alexandria, LA (AEX)
Allakaket, AK (AET)
Allentown, PA (ABE)
Alliance, NE (AIA)
Alpena, MI (APN)
Altoona, PA (AOO)
Amarillo, TX (AMA)
Ambler, AK (ABL)
Anaktueuk, AK (AKP)
Anchorage, AK (ANC)
Angoon, AK (AGN)
Aniak, AK (ANI)
Anvik, AK (ANV)
Appleton, WI (ATW)
Arcata, CA (ACV)

Important map/Unordered_map Member functions

pair<iterator, bool> insert(**const** value_type& x)

Inserts x into the map

- Won’t insert if there’s already an element with that key in the map
- Return value.second indicates whether insertion was successful

iterator find(**const** key_type& k)

Finds an element whose key is k

- Returns end( ) if not found
- Caller should check whether returned iterator is valid

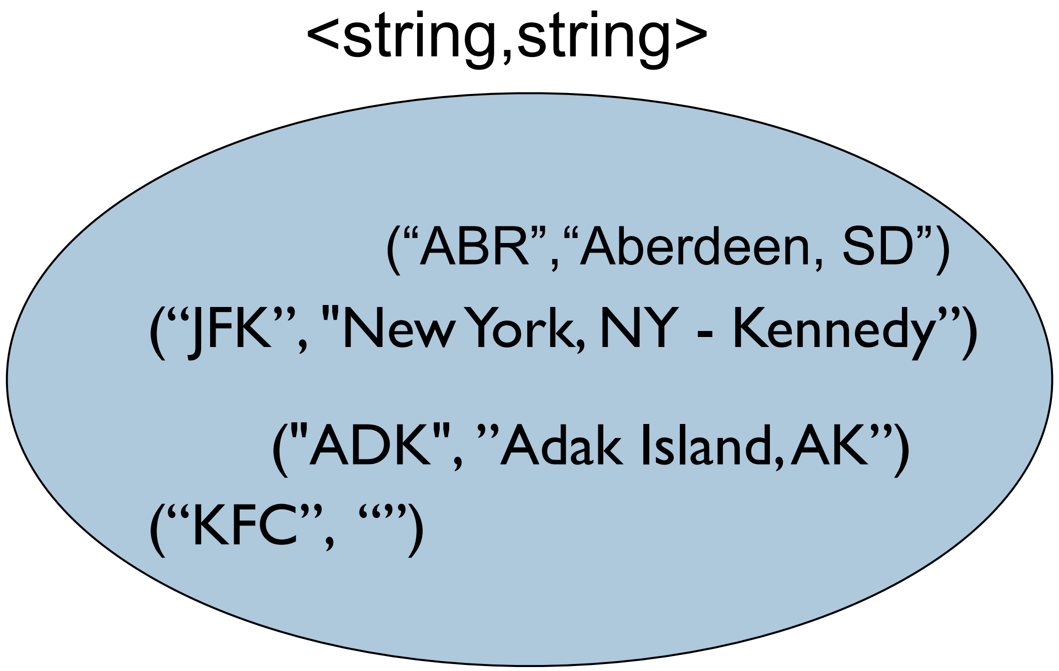
void erase(iterator pos)

Erases the element pointed to by pos

size_type erase(**const** key_type& k)

Erases the element whose key is k

```
~~unordered~~_map<string,string> mymap;
~~unordered~~_map<string,string>::iterator mItr;

pair<string, string> airportCode1;
airportCode1.first = “ABR”;
airportCode1.second= “Aberdeen, SD”;
mymap.insert(airportCode1);
mymap.insert(pair<string, string>("ADK", ”Adak Island, AK”))

mItr = mymap.find(”ABR");
if ( mItr == mymap.end( ))
    cout << ”ABR is not in the map”;
else
    mymap.erase( mItr );

mymap[“JFK”] = “New York, NY - Kennedy”;
if ( mymap[“KFC”] != “Lexington, KY” )
    cout << “I just added KFC to mymap!”;
```



<string,string>

(“ABR”,“Aberdeen, SD”)

(“JFK”, "New York, NY - Kennedy”)

("ADK", ”Adak Island, AK”)

(“KFC”, “”)

map::operator[ ]
unordered_map::operator[ ]

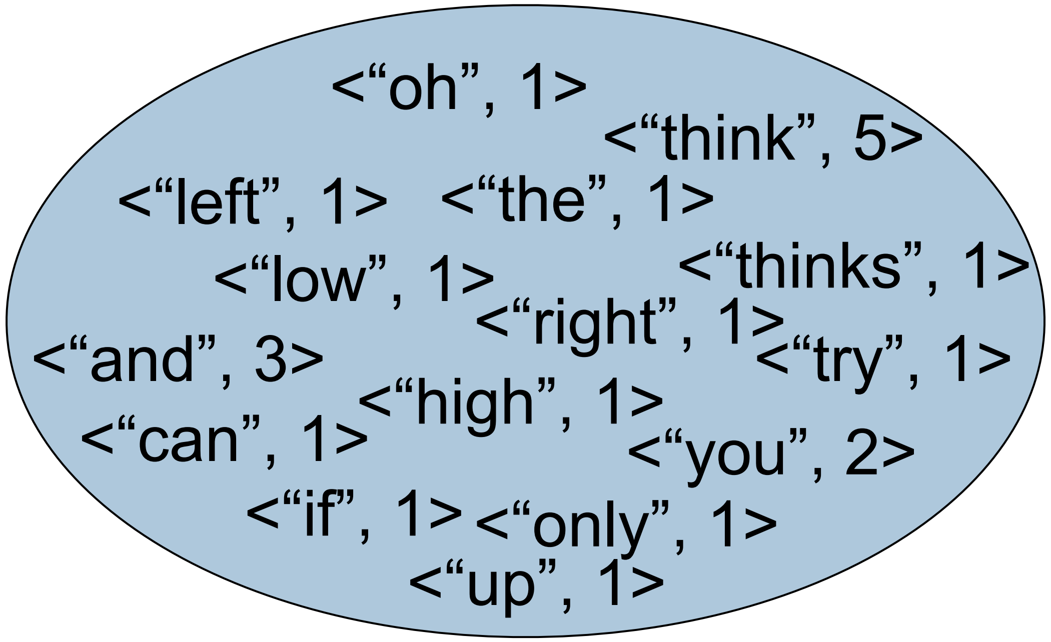
- data_type& operator[ ](const key_type& k)
  Returns a reference to the object that is associated with a particular key. If the map does not already contain such an object, operator[ ] inserts the default object data_type().
  - m[k] is equivalent to the "simple" ☺ (according to STL docs) expression

    (*((m.insert(value_type(k, data_type()))).first)).second
  - Notation suggests array indexed by key values (but that's not how it's implemented)
  - If side effect of adding new object when key is not found is not wanted, instead use:

```
it = m.find(k);
if (it != m.end())
  { // access or update it->second};
else
  {//handle case where k is not found}
```

  - Similar situation if update of data for an existing key is not wanted

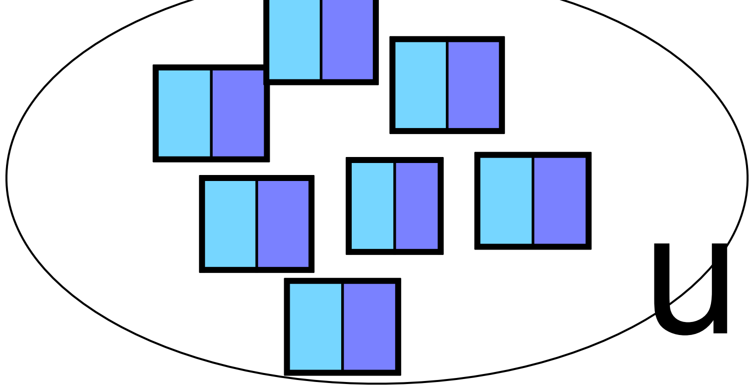
Quote by Dr. Seuss: “think left and think right and think low and think high oh the thinks you can think up if only you try”



<“oh”, 1>
<“think”, 5>
<“left”, 1>
<“the”, 1>
<“low”, 1>
<“thinks”, 1>
<“right”, 1>
<“and”, 3>
<“try”, 1>
<“high”, 1>
<“can”, 1>
<“you”, 2>
<“if”, 1>
<“only”, 1>
<“up”, 1>

```
// Word frequencies -- using map
// Fred Swartz 2001-12-11
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
       string word;
       map<string, int> freq;
       // map of words and their frequencies

       // input buffer for words.
       //--- Read words/tokens from input stream
      while (cin >> word)
              { freq[word]++; }
      //--- Write the count and the word.
      map<string, int>::const_iterator iter;
      for (iter = freq.begin(); iter ! = freq.end(); ++iter)
                     { cout << iter->second << " " << iter->first << endl; }
      return 0;
}
```

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

int main()
{
       unordered_map<string, int> freq;
       string word;
       // map of words and their frequencies

       // input buffer for words.
       //--- Read words/tokens from input stream
      while (cin >> word)
              { freq[word]++; }
      //--- Write the count and the word.
      unordered_map<string, int>::const_iterator iter;
      for (iter = freq.begin(); iter ! = freq.end(); ++iter)
                     { cout << iter->second << " " << iter->first << endl; }
      return 0;
}
```



map

map<key,value, key_compare>

bidirectional iterator

- Unique keys (no duplicates)
- Supports insertion, deletion, and find in O(log n) time
- range [first, last) is sorted by key
- How’s it implemented?
  - vector or linked list can’t meet all the time bounds
  - Answer: …

unordered_map

unordered_map<key,value, key_compare>

forward iterator

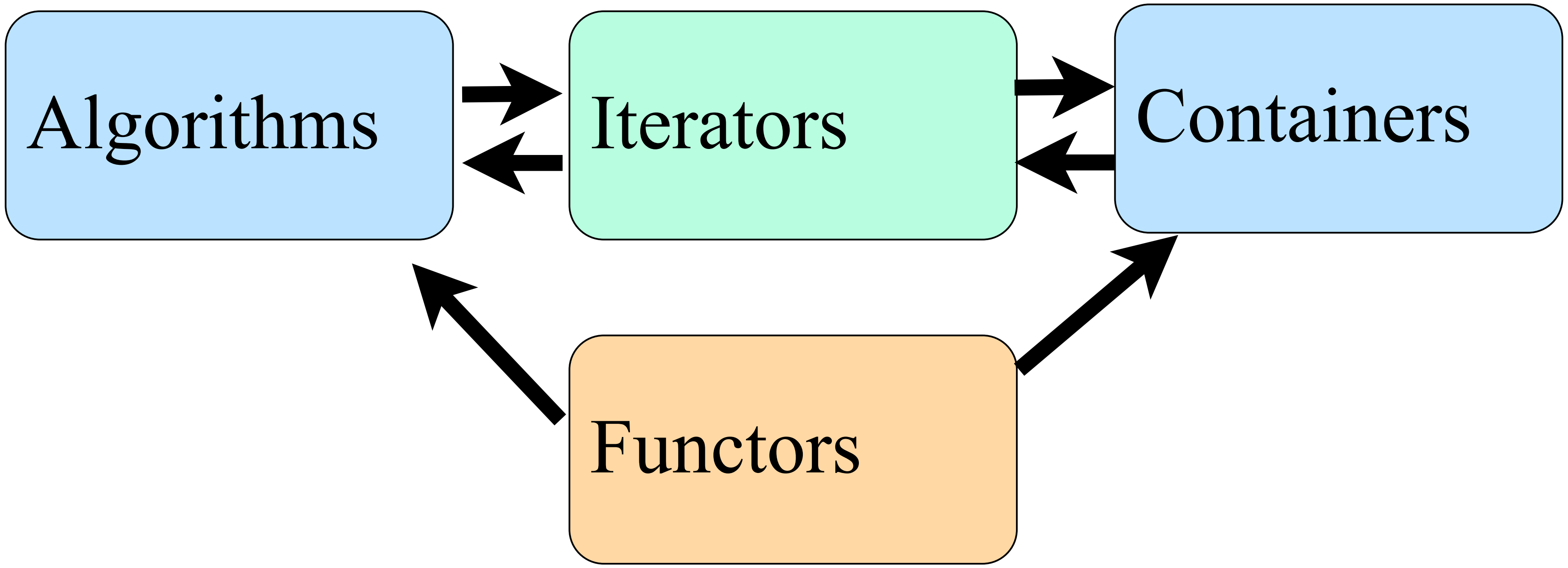
- Unique keys (no duplicates)
- Supports insertion, deletion, and find in O(1) time on average
- range [first, last) is unsorted
- How’s it implemented?
  - vector or linked list can’t meet all the time bounds
  - Answer: …

data structure	build	insert	find
vector	O(n)	O(1) Into the back	O(n)
sorted vector	O(n log n)	O(n)	O(log n)
set or map	O(n log n)	O(log n)	O(log n)
unordered_set unordered_map	O(n) ave. O(n^2) worst	O(1) ave O(n) worst	O(1) ave O(n) worst

Easy to use code written by someone else.

STL

Standard Template Library



Algorithms

Iterators

Containers

Functors

A C++ 11 STL reference can be found at:

http://en.cppreference.com/w/cpp

Another C++ reference can be found at:

http://www.cplusplus.com/reference/

Motivation for the STL Algorithms

Computers do very simple operations: add, shift by two, etc.
Compilers take the C++ language into machine code.
But we still have to write a large number of steps compared to typical pseudo code
STL algorithm take one more step to being able to write high level commands.

Find the average exams score

```
ifstream input(“exam1.txt”);
vector<double> exam_scores;

int score;
while ( input >> score )
    exam_scores.push_back(score);



// Compute the average
double total = 0;
for (vector<double>::iterator itr = exam_scores.begin(), itr != exam_scores.end(); ++itr)
     total += *itr;
cout << “Average score for exam 1 is “ << total/exam_scores.size();
```

To use accumulate
you need to add
#include <numeric>

Instead Use a STL Algorithm

Find the average exams score.

```
ifstream input(“exam1.txt”);
vector<double> exam_scores;

int score;
while ( input >> score )
    exam_scores.push_back(score);



// Compute the average
cout << accumulate(exam_scores.begin(), exam_scores.end(), 0.0)/exam_scores.size();
```

#include<algorithm>
#include<numeric>

STL Algorithms

Algorithms typically have requirements for the type of iterator passed as a parameter. Bidirectional iterators can be used whenever forward iterators are required/used. Random access iterators can be used whenever bidirectional iterators are required/used.

- Iterator-based template functior
- Types of algorithms: non-modifying sequence operators, mutating sequence operators, sorting etc, and numeric operation.

Soooo simple!

You can and will write these yourself!

Reasons to use the STL Algorithms

- speed
- correct
- clarity

Typical STL Algorithm

```
template <class ForwardItr, class T>
T accumulate (ForwardItr first, ForwardItr last, T init )
{
    while (first!=last)
        init = init + *first++;

    return init;
}
```

- Range accessed in find is [first, last)
  - round parenthesis means boundary not included

To use accumulate
you need to add
#include <numeric>

Using the STL Algorithm

Find the average exams score.

```
template <class ForwardItr, class T>
T accumulate (ForwardItr first, ForwardItr last, T init )
{
    while (first!=last)
        init = init + *first++;

    return init;
}
```

```
ifstream input(“exam1.txt”);
vector<double> exam_scores;

int score;
while ( input >> score )
    exam_scores.push_back(score);
```

```
// Compute the average
cout << accumulate(exam_scores.begin(), exam_scores.end(), 0.0)/exam_scores.size();
```

accumulate with a specified binary operation to compute the result

```
template <class ForwardIterator, class T, class BinaryOperation>
  T accumulate (ForwardIterator first, ForwardIterator last, T init, BinaryOperation binary_op)
{
   while (first!=last) {
       init=binary_op(init,*first);
       ++first;
    }
   return init;
}
```

To use accumulate
you need to add
#include <numeric>

Find the average gpa

vector<student> class_list;

```
template <class ForwardIterator, class T, class BinaryOperation>
   T accumulate (ForwardIterator first, ForwardIterator last, T init, BinaryOperation binary_op)
{
    while (first!=last) {
        init=binary_op(init,*first);
        ++first;
     }
    return init;
}
```

```
//create a functor!
class add_gpa
{
public:
    double operator( )(double total, const student & s) { return total + s.get_gpa(); }
};
```

```
// Compute the average
cout << accumulate(class.begin(), class.end(), 0.0, add_gpa() )/class.size();
```

```
#include<functional>
```

STL function objects

STL Function Objects

STL function objects are *classes* that contain an operator( )

- Generator function objects don't take a parameter they return a value (e.g. *rand*, the random number generator functor.)
- Unary function objects take one parameter
- Binary function objects take two parameters

A special kind of functor is a predicate functor: function that returns a bool

- Examples of binary predicate objects in the STL

  less encapsulates operator<

  greater encapsulates operator>

  equal_to encapsulates operator==

  not_equal_to encapsulates operator!=

  greater_equal encapsulates operator>=

  less_equal encapsulates operator<=

#include<functional>

STL function object examples less

```
template <class Object>
class less
{ public:
     bool operator()(const Object& lhs, const Object& rhs) const
        {return lhs < rhs;}
};
```

#include<functio

STL function object example

```
template <class Object>
class less
{ public:
  bool operator()(const Object& lhs, const Object& rhs)const
      {return lhs < rhs;}
};
```

```
// less example
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

int main () {
  int foo[]={10,20,5,15,25};
  int bar[]={15,10,20};
  sort (foo, foo+5, less<int>() );   // 5 10 15 20 25
  sort (bar, bar+3, less<int>() );  //   10 15 20
  return 0;
}
```

code modified from http://www.cplusplus.com/reference/functional/less/

#include<functional>

greater_equal

```
template <class T>
class greater_equal
{
  public:
      bool operator() (const T& lhs, const T& rhs) const
        {return lhs >= rhs;}
};
```

#include<functional>

minus

```
template <class T>
class minus
{
    public:
        T operator() (const T& lhs, const T& rhs) const
            {return lhs-rhs;}
};
```

lower_bound

- lower_bound does binary search on range [first,last)
  - container must be sorted
  - need random access iterator for runtime O(log n)
  - Code (Figure 7.9, p. 244) for random access iterator (STL code is slightly different)
  - computation of middle iterator uses iterator subtraction
  - returns iterator to leftmost element in [first,last) containing element >= x (if none exists, returns last)

STL Style Binary Search Algorithm

```
template<class RandomIterator, class Object, class Compare>
RandomIterator lower_bound( const RandomIterator begin, const RandomIterator end,
  const Object & x, const Compare lessThan)
{
     RandomIterator low=begin;
     RandomIterator mid;
     RandomIterator high = end;
   while (low < high)
        {
          mid = low + (high - low) /2;
          if(lessThan(*mid, x))
               low = mid +1;
          else
               high=mid;


        }

      return low;
}
```

Pair of iterators define search space

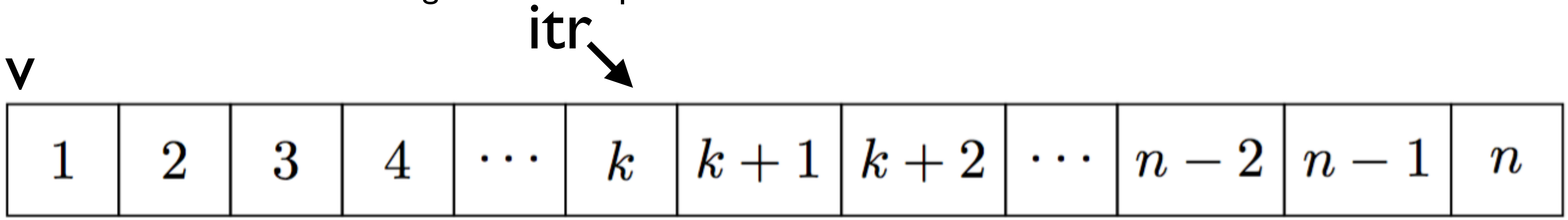
```
template<class Object>
class less
{
public:
     bool operator()(const Object& x,const Object& y) const  { return (x < y); }
};
```

Running Time?

O(log(n)) where n is the number of items in [first, last)

Additional Information

The following shows how the vector changes after the erase method is called. Assume the following vector v contains the numbers 1 through n. And itr points to the number k.

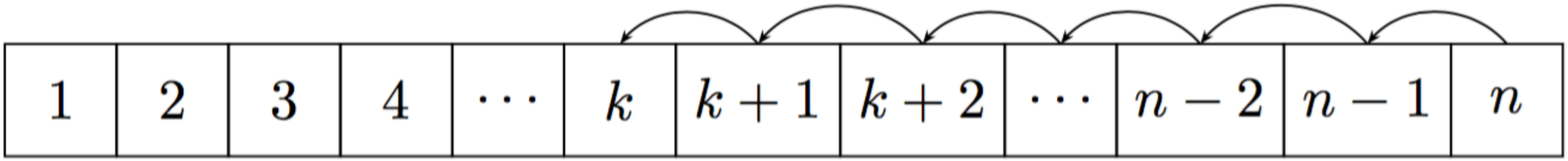


v

itr

1	2	3	4	$\cdots$	k	$k+1$	$k+2$	$\cdots$	$n-2$	$n-1$	n

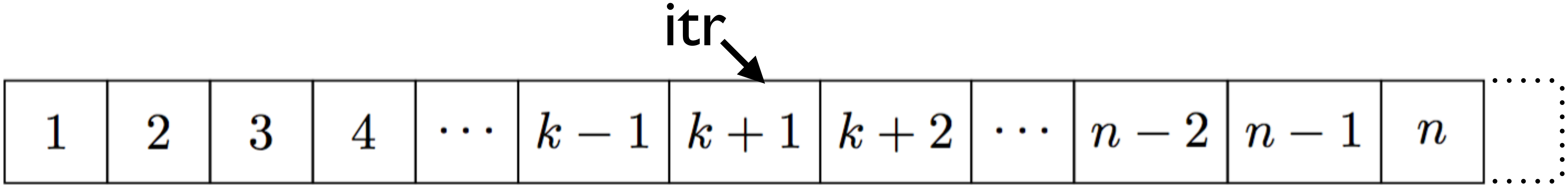
The method v.erase(itr) moves the item in location (itr + 1) into the location pointed to by (itr), then moves the item in location (itr+2) into the location (itr + 1), etc.



1	2	3	4	$\cdots$	k	$k+1$	$k+2$	$\cdots$	$n-2$	$n-1$	n

In this example, the method returns an iterator to the number (k+1).

After calling v.erase(itr) the size of vector has decreased.



itr

1	2	3	4	$\cdots$	$k-1$	$k+1$	$k+2$	$\cdots$	$n-2$	$n-1$	n

STL Containers

Any container in the STL contains:

- c.empty()
- c.clear( )
- c.size( )
- c.max_size()
- operator=
- c.swap()
- c.erase()
- operator<, operator>, …
- c.insert(iterator,value) // inserts before iterator where applicable
- c.begin( ) //returns an iterator to the first element
- c.end( ) //returns an iterator to one past the last element

Any container adapter in the STL contains:

- c.empty()
- c.clear( )
- c.size( )
- c.max_size()
- operator=
- c.swap()

also (except for priority queue)

- operator<, operator>, …

Elements stored in a container need a default constructor, destructor, assignment operator. Some compilers need some overloaded operators as well

STL Iterators

- Designed to act like pointers to arrays
- Iterators refer to a specific type of container

```
vector<int> v1(3);
vector<int>::iterator vecIntItr;
vector<string> v2(3); // cannot use vecIntItr with v2
vector<string>::iterator vecSItr2;
list<int> l; //cannot use vecIntItr or vecSItr2 with l
list<int>::iterator listItr;
```

- begin( ) and end( ) are member functions of every container

  begin( ) returns iterator accessing first item

  end( ) returns iterator accessing one position PAST last item

- Designed to be fast (consequently no error checking...)

How to instantiate an iterator

A const iterator
must be used if a contain
is non modifiable.

Random Access Iterators

```
vector<T>::iterator vecItr;                vector<T>::const_iterator constVecItr;
```

Bidirectional Iterators

```
list<T>::iterator listIter;         list<T>::const_iterator constListItr;
map<K, V>::iterator mapItr;   map<K, V>::const_iterator constMapItr;
set<K>::iterator setItr;           set<K>::const_iterator constSetItr;
```

Forward Iterators

```
unordered_set<K>::iterator setItr;      unordered_set<K>::const_iterator constSetItr;
unordered_map<K>::iterator setItr;    unordered_map<K>::const_iterator constSetItr;
```

Finding an item

```
vector<int>::iterator find(vector<int>::iterator start,
                           vector<int>::iterator end, int search_item)
{
    vector<int>::iterator itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
```

```
list<int>::iterator find(list<int>::iterator start, list<int>::iterator end,
                         int search_item)
{
    list<int>::iterator itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
```

```
template<class Iter, class Object>
Iter find(Iter start, Iter end, Object search_item)
{
    Iter itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
nt main ()

 list<int>::iterator itrL;
 list<int> items1 {0,1,2,3,4,5};
 itrL = find(items1.begin(), items1.end(), 2);


 vector<int>::iterator itrV;
 vector<int> items2 {0,1,2,3,4,5};
 itrV = find(items2.begin(), items2.end(), 2);
```

Finding an item

```
map<int,char>::iterator find(map<int, char >::iterator start,
map<int, char>::iterator end, pair<const int, char> search_item)
{
    map<int,char>::iterator itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
```

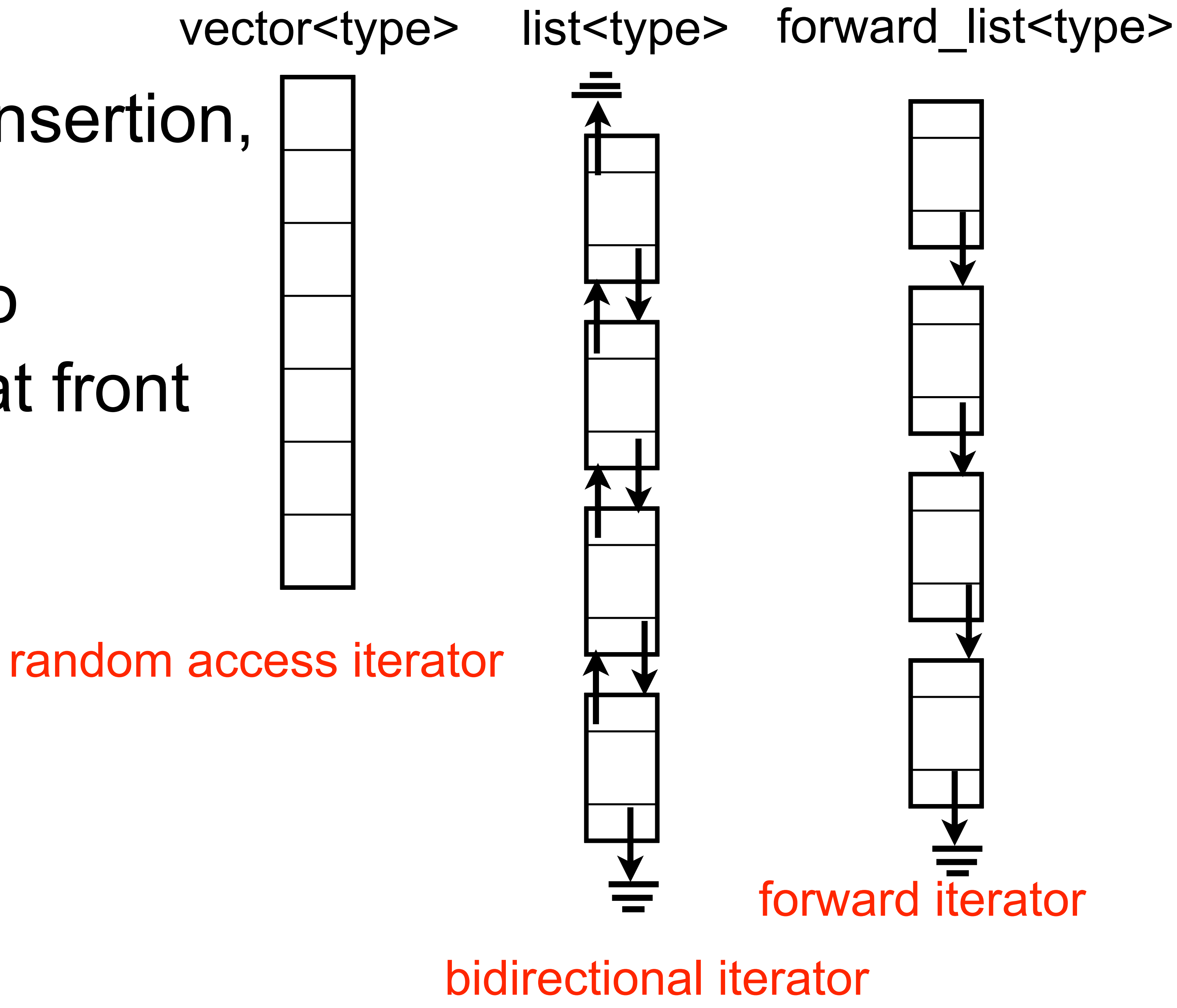
```
set<int>::iterator find(set<int>::iterator start, set<int>::iterator end,
                          int search_item)
{
    set<int>::iterator itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
```

```
template<class Iter, class Object>
Iter find(Iter start, Iter end, Object search_item)
{
    Iter itr;
    for ( itr = start; itr!=end; ++itr)
     if (*itr == search_item)
          break;
    return itr;
}
```

```
int main ()
{
  set<int>::iterator ItrS;
  set<int> items3 = {0,1,2,3,4,5};
  itrS = find(items3.begin(), items3.end(), 2);

  map<int,char>::iterator ItrS;
  map<int,char> items4 = {(0,'a'),(1,'b'),(2,'c)};
  pair<const int,char> myPair(2,'b');

  itrS = find( items4.begin(), items4.end(), myPai
```

Sequence containers $A_1,A_2,A_3,\ldots,A_n$

- vector: Efficient indexed access v[i], insertion/deletion at end
- list, forward_list: Efficient insertion, or deletion at any position
- deque: Like vector, but also efficient insertion/deletion at front

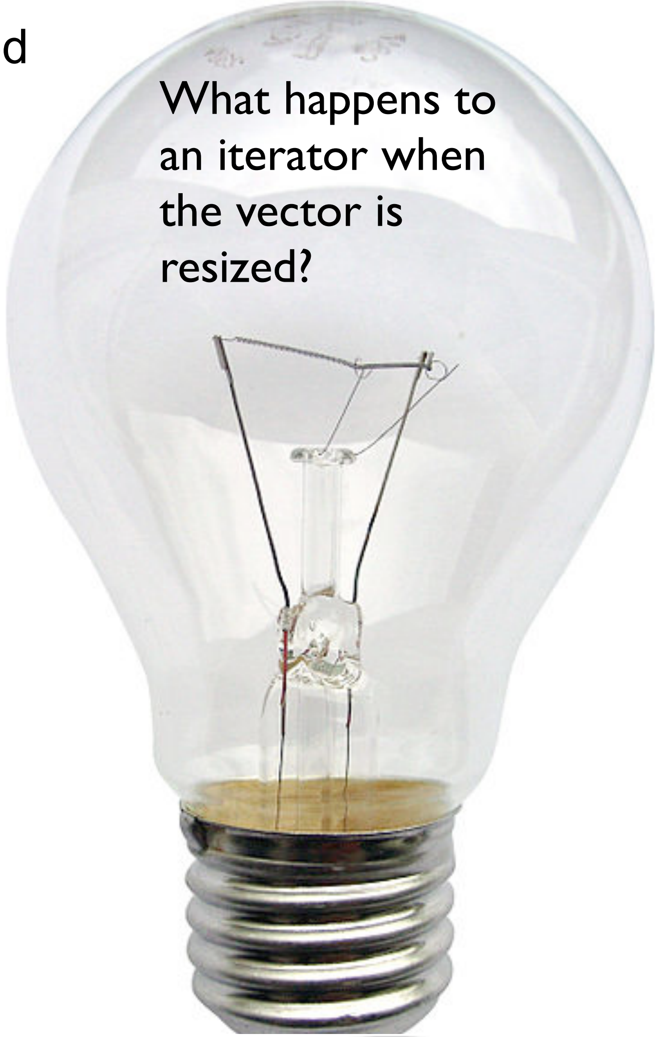


#include<vector>

#include<list>

vectors - Random Access Iterator

v.push_back(value)	O(1) amortized
v.pop_back( )	O(1)
v.back( )	O(1)
v.front( )	O(1)
v[i]	O(1)
v.erase(v.begin(),v.end())	O(n)
v.erase(iterator)	O(n)
v.clear()	O(n)
v.size()	O(1)
v.insert(iterator,value)	O(n)
v.begin()	O(1)
v.end()	O(1)
v.resize(n) or v.resize(n,value)	O(n)
v.reserve(n)	O(n)
v1 = v2	O(n)
v.capacity	O(1)



What happens to an iterator when the vector is resized?

Unlike a vector, a list does not use more space than needed. A list is useful to insert and delete without moving existing elements

list - Bidirectional Iterators

l.push_back(value)	O(1)
l.pop_back( )	O(1)
l.push_front(value)	O(1)
l.pop_front( )	O(1)
l.front()	O(1)
l.back()	O(1)
l.erase(v.begin( ),v.end( ))	O(n)
l.erase(iterator)	O(1)
l.clear( )	O(n)
l.size( )	O(1)
l.insert(iterator,value) //inserts before iterator	O(1)
l.begin( )	O(1)
l.end( )	O(1)
l.resize(n) or l.resize(n,value)	
l1 = l2	O(n)
l.sort( ) & l.sort(comparator)	O(n log(n))

This list is not complete
Check expert-level
resource for more info.

Note: all these times do not include constructor/destructor times which many vary according to the type

Functor Example

From http://www.stroustrup.com/bs_faq2.html#this

```
class Sum {
     int val;
public:
     Sum(int i) :val(i) { }
     operator int() const { return val; }      // extract value

     int operator()(int i) { return val+=i; }  // application
};

void f(vector<int> v)
{
     Sum s = 0;    // initial value 0
     s = for_each(v.begin(), v.end(), s); // gather the sum of all elements
     cout << "the sum is " << s << "\n";

     // or even:
     cout << "the sum is " << for_each(v.begin(), v.end(), Sum(0)) << "\n";
}
```

functor
Capable of maintaining a state.
The state can be examined
from the outside (static variables
cannot be examined from the
outside.)

Simple to adapt to a new search criteria!

```
template<class InputIterator, class UnaryPredicate>
  InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred)
{
  while (first!=last) {
    if (pred(*first)) return first;
    ++first;
  }
  return last;
}
```

```
class gpa_between
{
public:
   gpa_between(double l, double u):lower(l),upper(u){};
   bool operator()(student& record){return ((lower<= record.get_gpa()) &&
(record.get_gpa() <= upper)); }
private:
   double lower;
   double upper;
};
```

```
int main ()
{
  vector<student> classList;
  vector<student>::iterator itr;
  //some code to fill the vector, etc
  itr = find_if(classList.begin(), classList.end(), gpa_between(3.0,4.0));
  cout << endl<< (*itr).get_name()<< endl;
```

find_if

```
template<class InputIterator, class UnaryPredicate>
  InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred)
{
  while (first!=last) {
    if (pred(*first)) return first;
    ++first;
  }
  return last;
}
```

```
class gpaIs
{
public:
   gpaIs(const double value):value(value){}
   bool operator()(student& rhs){return ( value == rhs.get_gpa() );}
private:
   double value;
};
```

classList

George	Thomas	Adam	William	Abagail		
2.2	3.3	2.3	3.8	4		

```
int main ()
{
 vector<student> classList;
 vector<student>::iterator itr;
 gpaIs gpaIs3p3(3.3);
 //some code to fill the vector, etc
 itr = find_if(classList.begin(), classList.end() gpaIs3p3);
```

for_each

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function fn)
{
  while (first!=last) {
    fn (*first);
    ++first;
  }
  return fn;
}
```

Code using a STL algorithm for_each and a non-STL functor

Conversion operator is a member function. It cannot modify the member variables. Note that the syntax is odd. It has no return type:

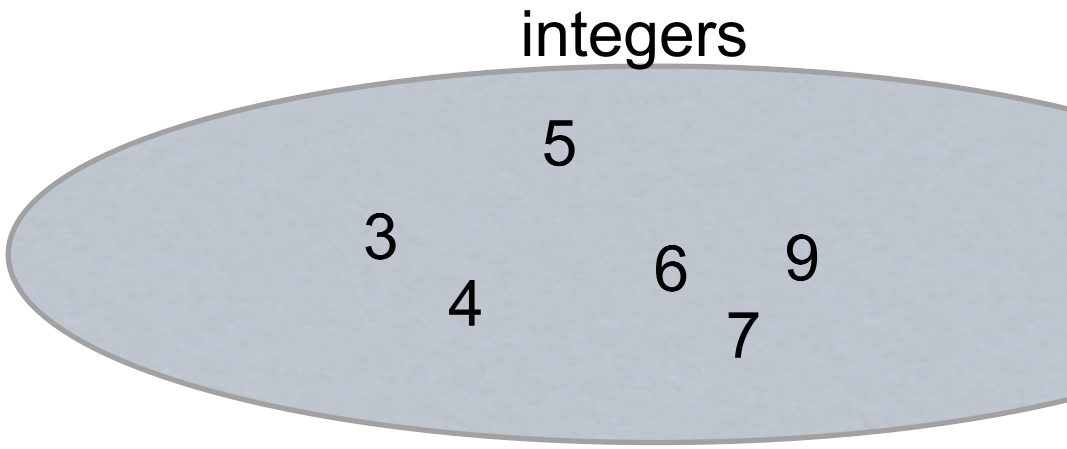
    operator type()const;

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function fn)
{
  while (first!=last) {
    fn (*first);
    ++first;
  }
  return fn;
}

    class Sum {
           int val;
    public:
           Sum(int i) :val(i) { }
           operator int() const { return val; }            // extract value

           int operator()(int i) { return val+=i; }   // application
    };

   void f(vector<int> v)
   {
          Sum s = 0;     // initial value 0
          s = for_each(v.begin(), v.end(), s);    // gather the sum of all elements
          cout << "the sum is " << s << "\n";

          // or even:
          cout << "the sum is " << for_each(v.begin(), v.end(), Sum(0)) << "\n";
   }
```

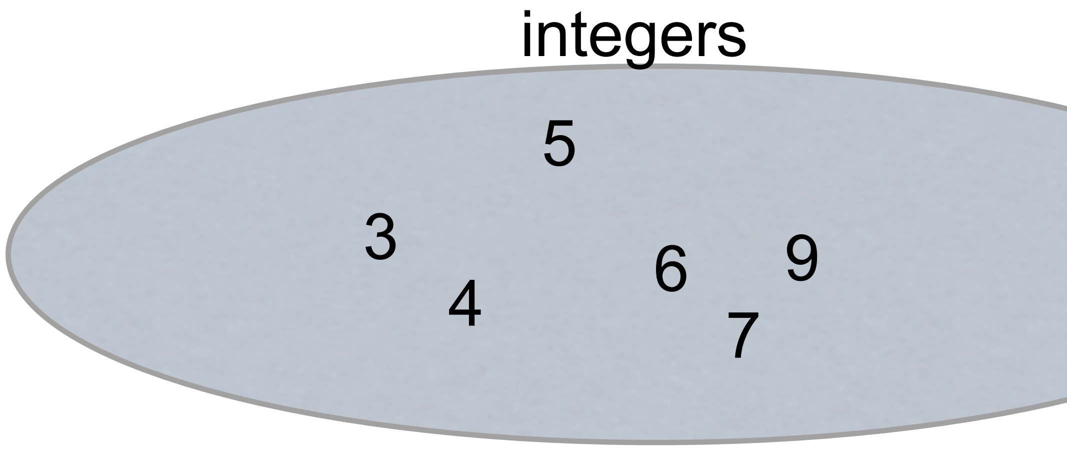
<u>functor</u>
Capable of maintaining a state. The state can be examined from the outside (static variables cannot be examined from the outside.)

Modified code from http://www.stroustrup.com/bs_faq2.html#this

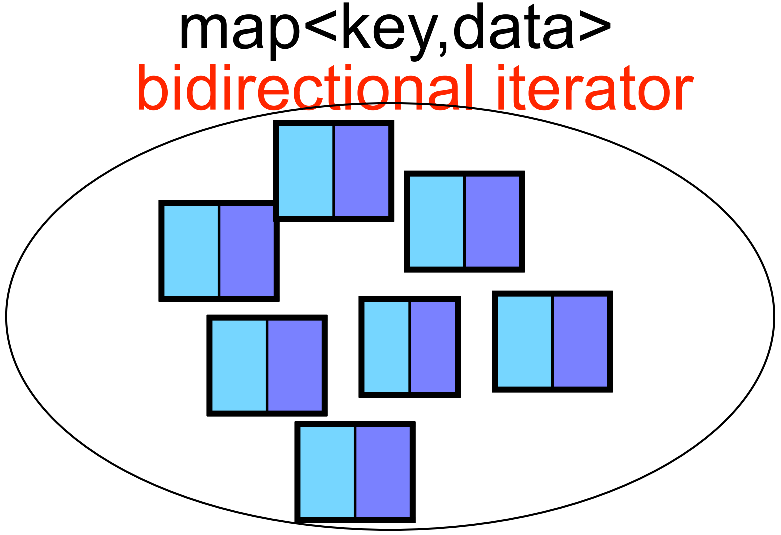
```
#include<unordered_set>

unordered_set<int> setOfIntegers;
unordered_set<int>::iterator itrS;
          for(int i=0; i<10; ++i)
            setOfIntegers.insert(rand()%10);

           cout << setOfIntegers.size() << "items inserted into the set" << endl;
           for(itrS=setOfIntegers.begin(); itrS!=setOfIntegers.end(); ++itrS)
             cout << *itrS << " ";
           cout << endl;


          itrS= setOfIntegers.find(3);// if 3 is found returns an iterator to 3
          if (itrS != setOfIntegers.end()) // if 3 isn’t found returns an iterator
          {                                 // to end( )
            setOfIntegers.erase(3);
            for(itrS=setOfIntegers.begin(); itrS!=setOfIntegers.end(); ++itrS)
             cout << *itrS << " ";
            cout << endl;
          }

           setOfIntegers.erase(13);//returns 0 since 13 did not exist,
```

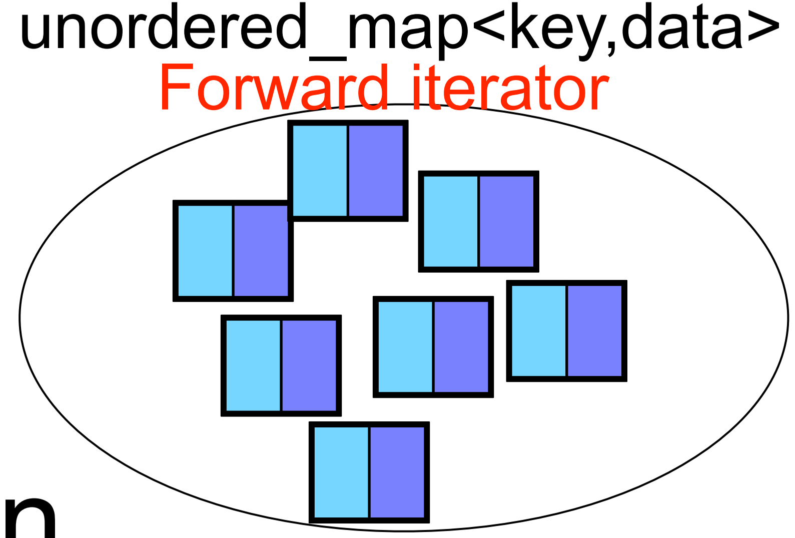


```
#include<set>
```



```
set<int> setOfIntegers;
set<int>::iterator itrS;
for(int i=0; i<10; ++i)
  setOfIntegers.insert(rand()%10);

cout << setOfIntegers.size() << "items inserted into the set" << endl;
for(itrS=setOfIntegers.begin(); itrS!=setOfIntegers.end(); ++itrS)
  cout << *itrS << " ";
cout << endl;


itrS= setOfIntegers.find(3);// if 3 is found returns an iterator to 3
if (itrS != setOfIntegers.end()) // if 3 isn’t found returns an iterator
{                                // to end( )
  setOfIntegers.erase(3);
  for(itrS=setOfIntegers.begin(); itrS!=setOfIntegers.end(); ++itrS)
   cout << *itrS << " ";
  cout << endl;
}

 setOfIntegers.erase(13);//returns 0 since 13 did not exist,
```





Some Types used in

map< key_type, data_type, key_compare>
unordered_map< key_type, data_type, key_compare>

- key_type : The map's key type (Key). Cannot be changed
- data_type : The type of object associated with the keys (Data). Can be changed
- value_type : The type of object,

  pair<const key_type, data_type>, stored in the map.
- key_compare : function object that compares two keys for ordering (Compare)
- const and non-const iterators
  - *it is not mutable, but it->second is mutable

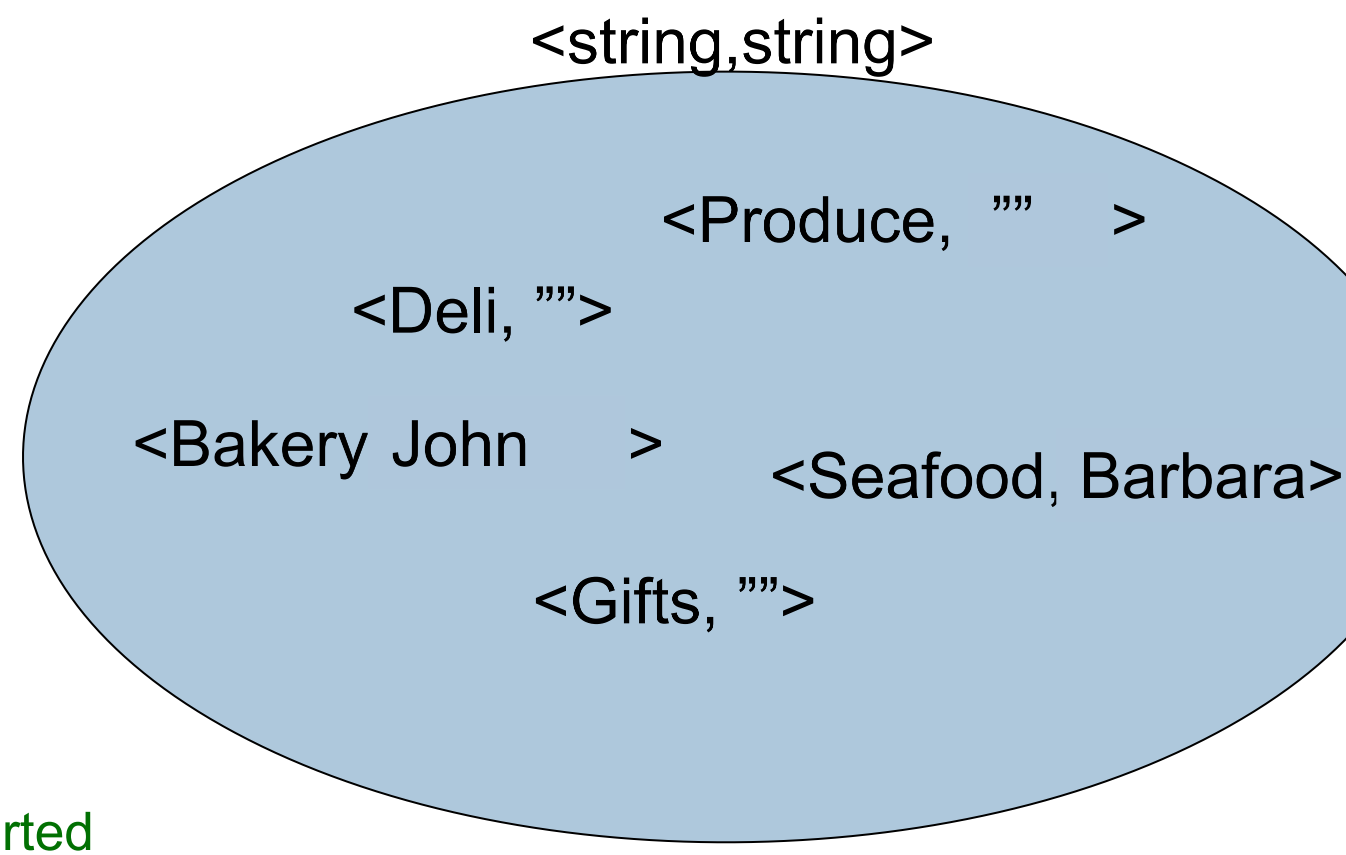
unordered_map example

```
// unordered_map::insert
#include <iostream>
#include <string>
#include <unordered_map>

int main ()
{
  std::unordered_map<std::string,double>
              myrecipe,
              mypantry = {{"milk",2.0},{"flour",1.5}};

  std::pair<std::string,double> myshopping ("baking powder",0.3);

  myrecipe.insert (myshopping);                        // copy insertion
  myrecipe.insert (std::make_pair<std::string,double>("eggs",6.0)); // move insertion
  myrecipe.insert (mypantry.begin(), mypantry.end());  // range insertion
  myrecipe.insert ( {{"sugar",0.8},{"salt",0.1}} );    // initializer list insertion
```

From http://www.cplusplus.com/reference/unordered_map/unordered_map/insert/

~~unordered~~_map example

```
#include <iostream>
#include <string>
#include ~~<unordered~~_map>

using namespace std;

int main ()
{
~~unordered~~_map<string,string> mymap;

  mymap["Bakery"]="Barbara";  // new element inserted
  mymap["Seafood"]="Lisa";    // new element inserted
  mymap["Produce"]="John";    // new element inserted

  string name = mymap["Bakery"];   // existing element accessed (read)
  mymap["Seafood"] = name;              // existing element accessed (written)

  mymap["Bakery"] = mymap["Produce"];   // existing elements accessed (read/written)

  name = mymap["Deli"];      // non-existing element: new element "Deli" inserted!

  mymap["Produce"] = mymap["Gifts"];    // new element "Gifts" inserted, "Produce" written
}
```



<string,string>

<Produce, ””   >

<Deli, ””>

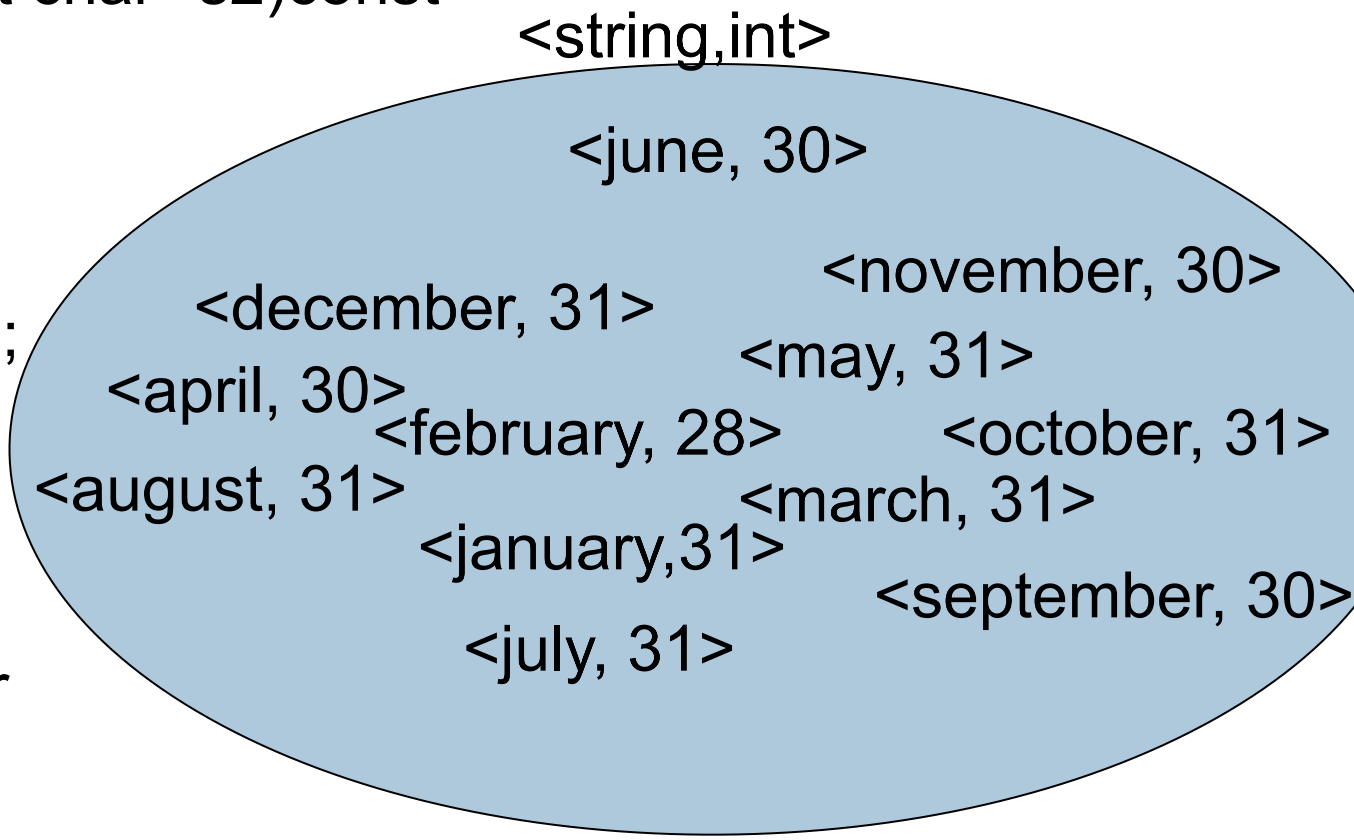
<Bakery John   >

<Seafood, Barbara>

<Gifts, ””>

Modified from http://www.cplusplus.com/reference/unordered_map/unordered_map/operator[]/

```
          // Example from SGI STL documentation
          struct ltstr{
           bool operator()(const char* s1,const char* s2)const
             {    return strcmp(s1, s2) < 0;  }
           };
          int main()  {
unordered_map<const char*, int, ltstr> months;
           months["january"] = 31;
           months["february"] = 28;
           . . .
unordered_map<const char*, int, ltstr>::iterator
             cur  = months.find("june");
unordered_map<const char*, int, ltstr>::iterator prev = cur;
unordered_map<const char*, int, ltstr>::iterator next = cur;
           ++next;
           --prev;
           cout << "Previous (in alphabetical order) is " <<    (*prev).first << endl;
           cout << "Next (in  alphabetical order) is " << (*next).first << endl;
          }
```



<string,int>

<june, 30>

<november, 30>

<december, 31>

<may, 31>

<april, 30>

<february, 28>

<october, 31>

<august, 31>

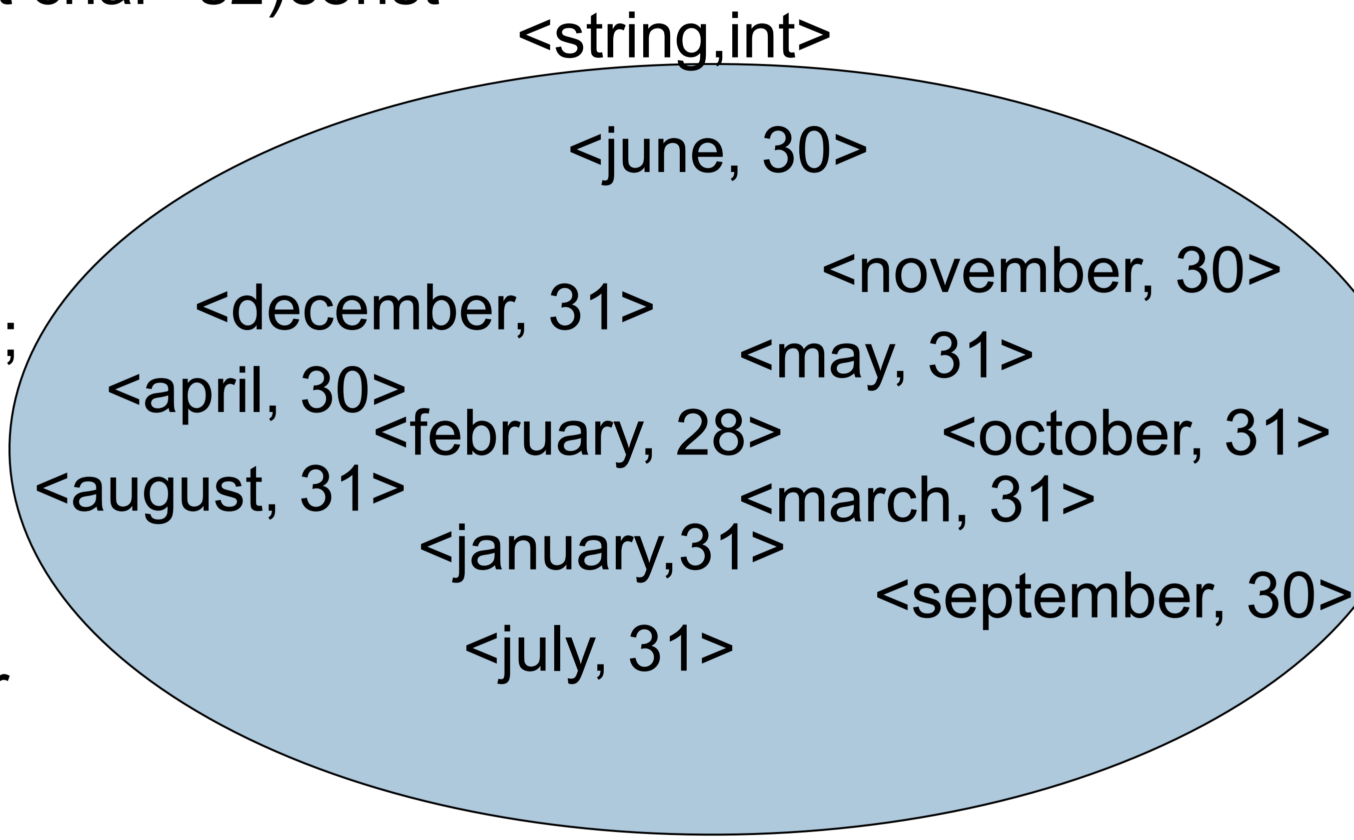
<march, 31>

<january,31>

<september, 30>

<july, 31>

```
// Example from SGI STL documentation
struct ltstr{
 bool operator()(const char* s1,const char* s2)const
   {    return strcmp(s1, s2) < 0;  }
 };
int main()  {
  map<const char*, int, ltstr> months;
  months["january"] = 31;
  months["february"] = 28;
  . . .
  map<const char*, int, ltstr>::iterator
    cur  = months.find("june");
  map<const char*, int, ltstr>::iterator prev = cur;
  map<const char*, int, ltstr>::iterator next = cur;
  ++next;
  --prev;
  cout << "Previous (in alphabetical order) is " <<    (*prev).first << endl;
  cout << "Next (in  alphabetical order) is " << (*next).first << endl;
}
```



#include <set>

#include <unordered_set>

set

Bidirectional Iterator

s.find(key)	O(log(n))
s.lower_bound(key)	O(log(n))
s.upper_bound(key)	O(log(n))
s.size()	O(1)
s.empty()	O(1)
s.insert(k)	O(log(n))
s.begin()	O(1)
s.end()	O(1)
s.erase(iterator) & s.erase(key)	O(1) amortized & O(log(n))
s.clear()	O(n)

unordered_set

Forward Iterator

	average case	worst case
s.find(key)	O(1)	O(n)
s.size()	O(1)	O(1)
s.empty()	O(1)	O(1)
s.insert(k)	O(1)	O(n)
s.begin()	O(1)	O(1)
s.end()	O(1)	O(1)
s.erase(iterator) & s.erase(key)	O(1), & O(1)	O(n), &O(n)
s.clear()	O(n)	O(n)

Note: all these times do not include constructor/destructor times which many vary according to the type

#include<map>

Great to use when you need to access the elements by key.

map - Bidirectional Iterator

m.insert(pair)	O(log(n))
m.find(key)	O(log(n))
m.size()	O(1)
m.begin()	O(1)
m.end()	O(1)
m.lower_bound(key)	O(log(n))
m.upper_bound(key)	O(log(n))
m[key]	O(log(n))
m.clear()	O(n)
m.erase(key) & m.erase(iterator)	O(log(n)) & O(1) amortized

This list is not complete. Check expert-level resource for more info.

#include<unordered_map>

unordered_map - Forward Iterator

u.insert(pair)	O(1) ave, 0(n) worst case
u.find(key)	O(1) ave, 0(n) worst case
u.size()	O(1)
u.begin()	O(1)
u.end()	O(1)
m[key]	O(1) ave, 0(n) worst case
m.clear()	O(n)
m.erase(key) & m.erase(iterator)	