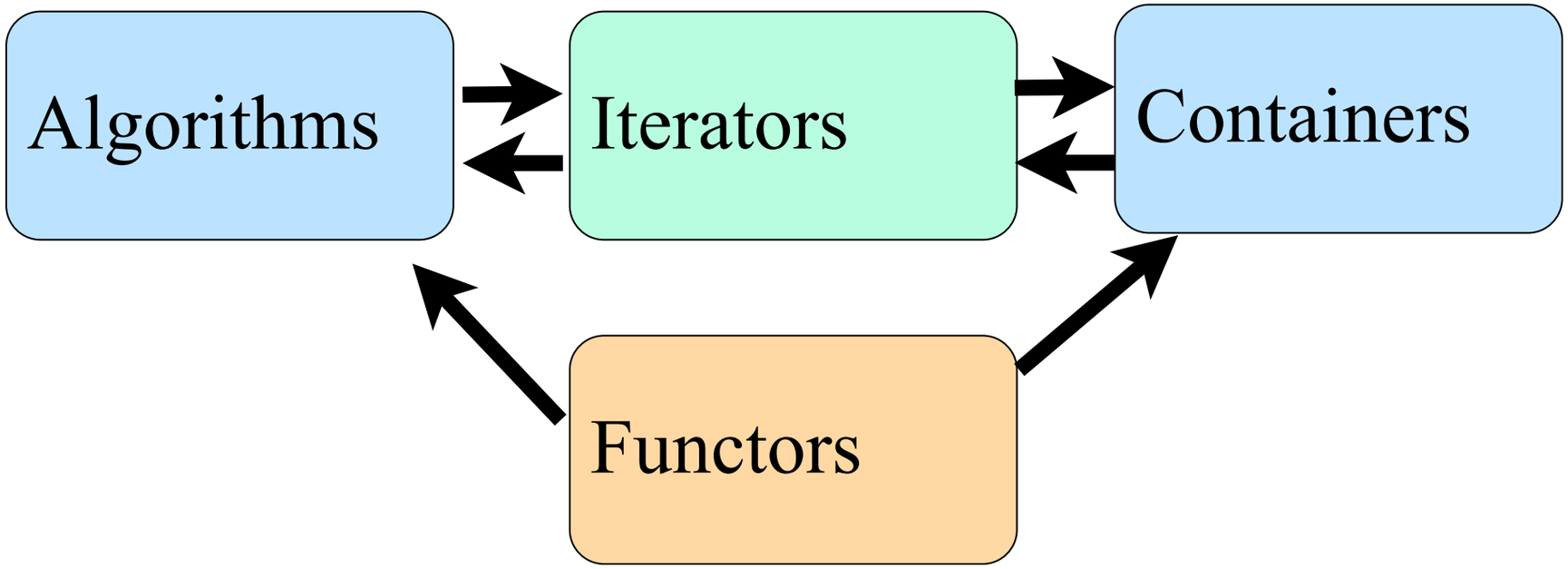
Scottsdale, AZ (SDL)
Scranton, PA (AVP)
Seattle, WA - Lake Union SPB (LKE)
Seattle, WA - Seattle/Tacoma International (SEA)
Selawik, AK (WLK)
Seward, AK (SWD)
Shageluk, AK (SHX)
Shaktoolik, AK (SKK)
Sheffield/Florence/Muscle Shoals, AL (MSL)
Sheldon Point, AK (SXP)
Sheridan, WY (SHR)
Shishmaref, AK (SHH)
Shreveport, LA (SHV)
Shungnak, AK (SHG)
Silver City, NM (SVC)
Sioux City, IA (SUX)
Sioux Falls, SD (FSD)
Sitka, AK (SIT)
Skagway, AK (SGY)
Sleetmore, AK (SLQ)
South Bend, IN (SBN)
South Naknek, AK (WSN)
Southern Pines, NC (SOP)
Spartanburg/Greenville, SC (GSP)
Spokane, WA (GEG)
Springfield, IL (SPI)
Springfield, MO (SGF)
St Petersburg/Clearwater, FL (PIE)
State College/University Park, PA (SCE)
Staunton, VA (SHD)
Steamboat Springs, CO (SBS)

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: $A_1, A_2, \ldots, A_N$ — We decide what is first, second, third, etc.

A *set* of items: $\{A_1, A_2, \ldots, A_N\}$ — We don't think of the items having an order, and there are no duplicates

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

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.

STL's ADT's

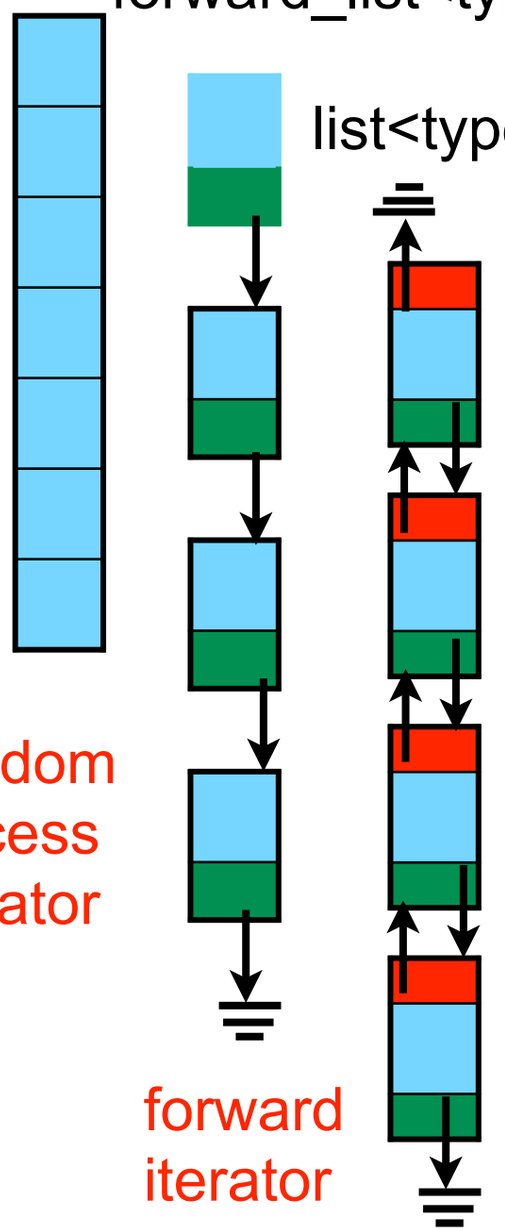
(not a complete list)

Lists
Sequence Containers

vector<type>

forward_list<type>

list<type>



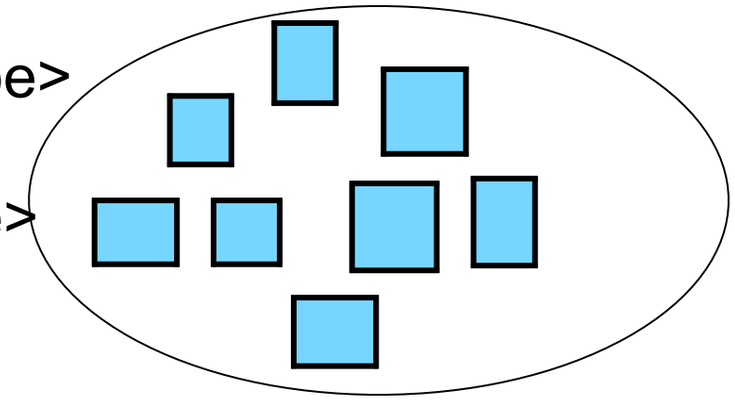
random access iterator

forward iterator

bidirectional iterator

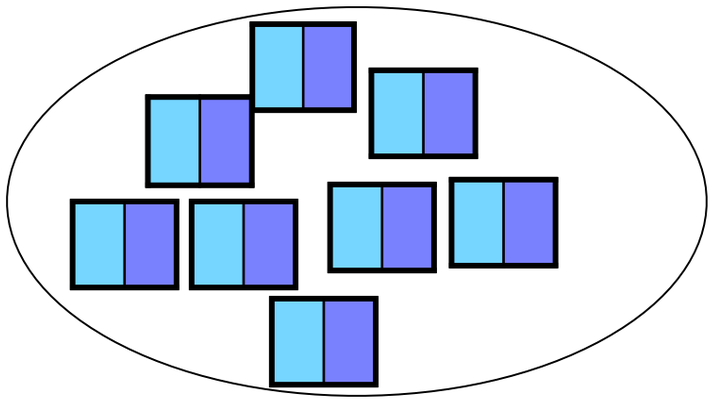
Set and Dictionary
Associate Containers

set<key>



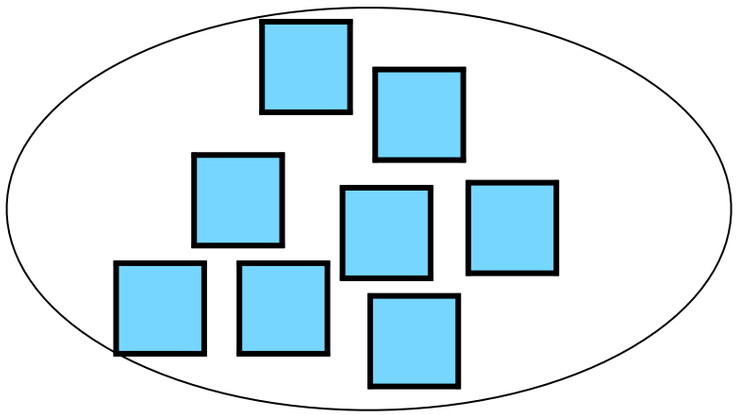
bidirectional iterator

map<key,data>



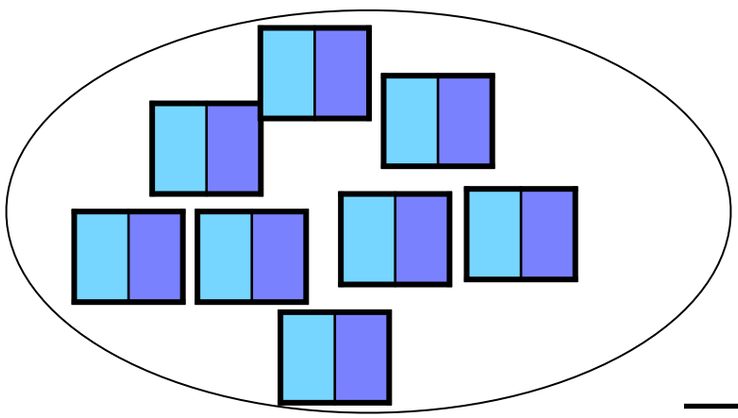
bidirectional iterator

unordered_set<key>



Forward iterator

unordered_map<key,data>

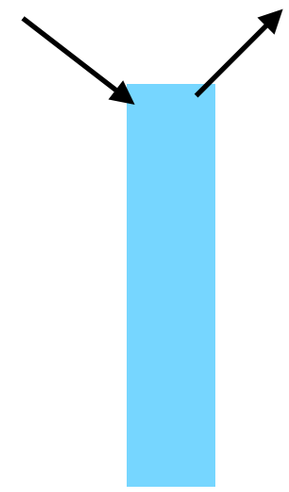


Forward iterator

Container Adapters

Stack

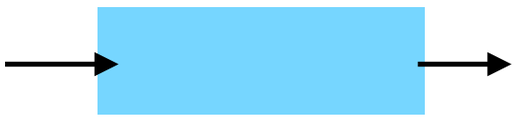
stack<type>



no iterator

Queue

queue<type>



no iterator

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 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

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()

also (except for priority queue)

- operator<, operator>, …

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

We need a way to iterate through the items

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.

All STL containers have these operations:

- iterator begin( ) const
- iterator end( ) const

STL Container Iterators

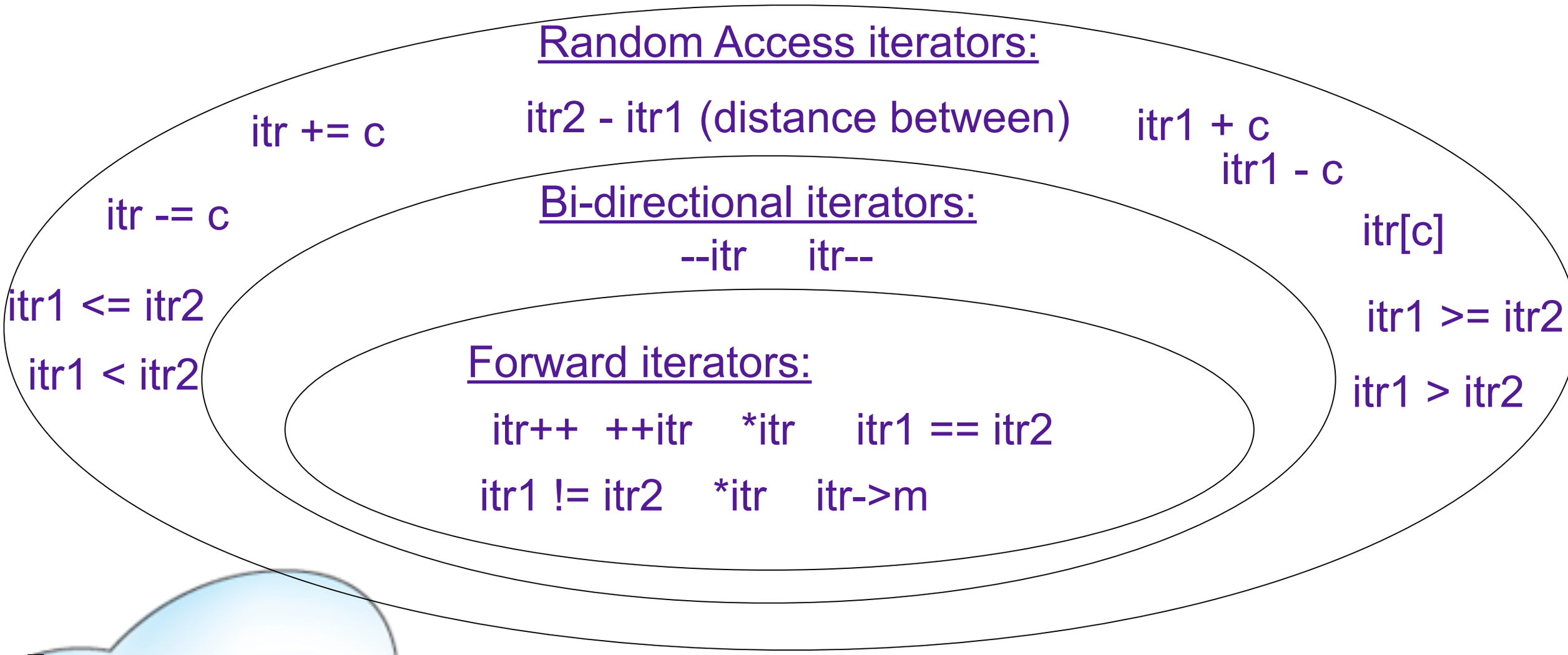
"iterators, which are a generalization of pointers"

- ++itr (or itr++) to move to next item
- *itr to dereference
- itr1 != itr2 to compare one iterator to another (or itr1 == itr2)

Some Containers have more powerful iterators

The container type determines the iterator type

Syntax is similar to pointers



To move an iterator n steps forward there is a function template called advance, advance(itr, n);
What do you think the running time of this function is?
There is function that determines the number of increments needed to get from ltr1 to ltr1, distance(ltr1, ltr2)

How to instantiate an iterator

Random Access Iterators

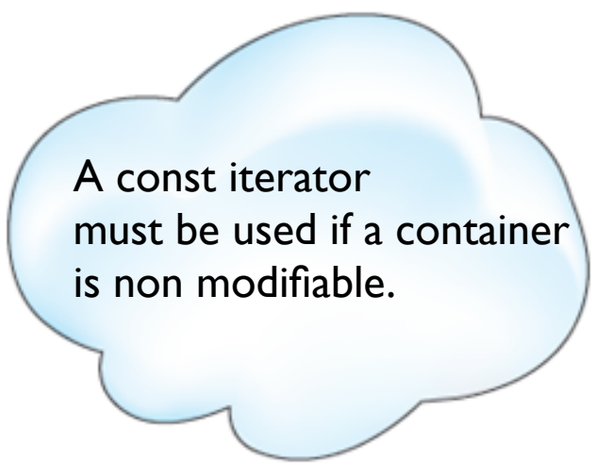
vector<T>::iterator vecItr; vector<T>::const_iterator constVecItr;

Bidirectional Iterators

list<T>::iterator listIter; list<T>::const_iterator constListItr;

map<K, V>::iterator mapItr; map<K, V>::const_iterator constMapItr;

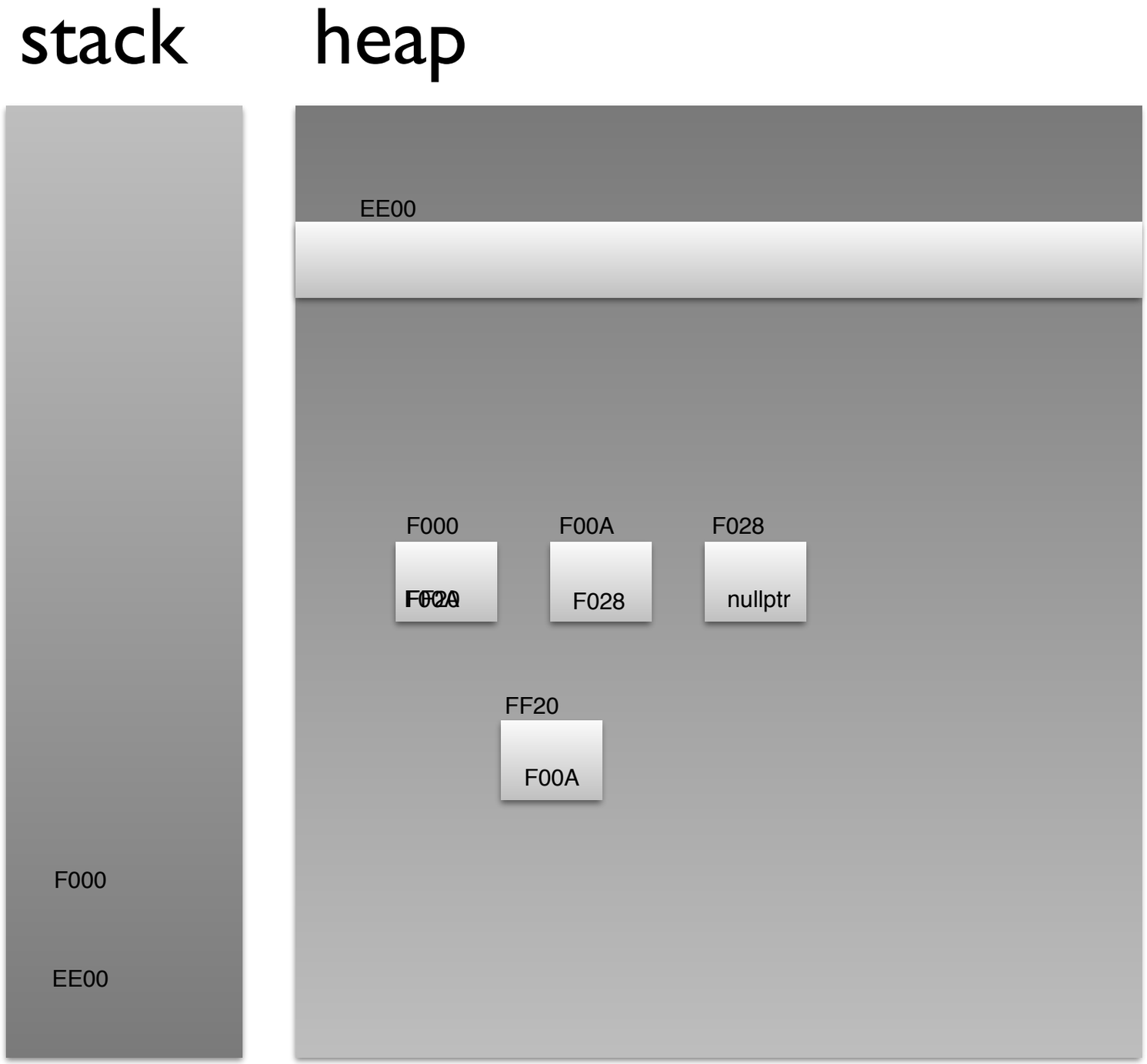
set<K>::iterator setItr; set<K>::const_iterator constSetItr;



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)

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
```

Finding an integer in a list

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

```
int 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

```
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));
```

Vectors and Strings

- Arrays are not “first class objects” – cannot do “the usual operations” such as =, ==
- STL provides vectors and strings
- 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

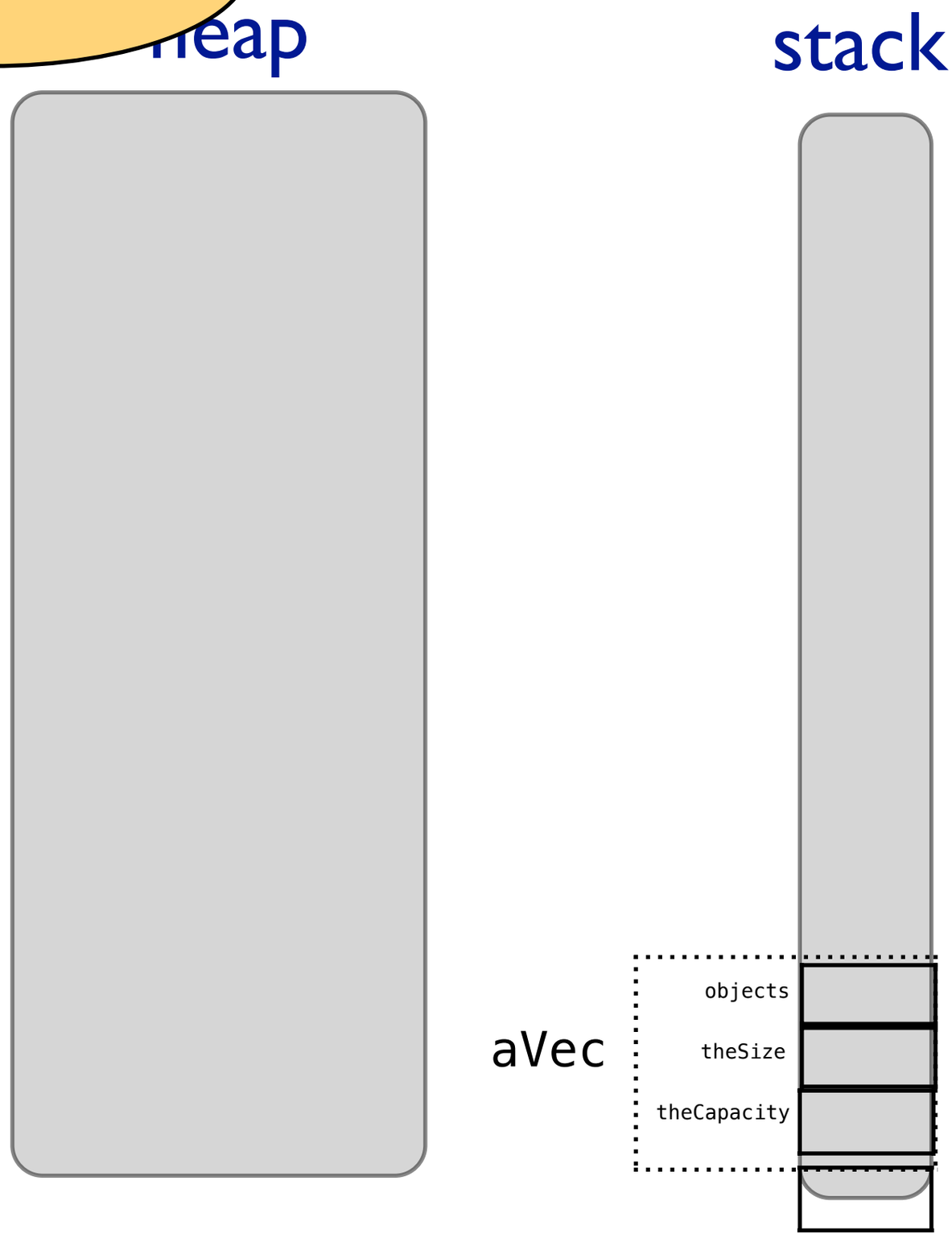
Our class is called **V**ector class to distinguish it from the STL vector class.

How would you create a vector class?

How would you create a vector class?

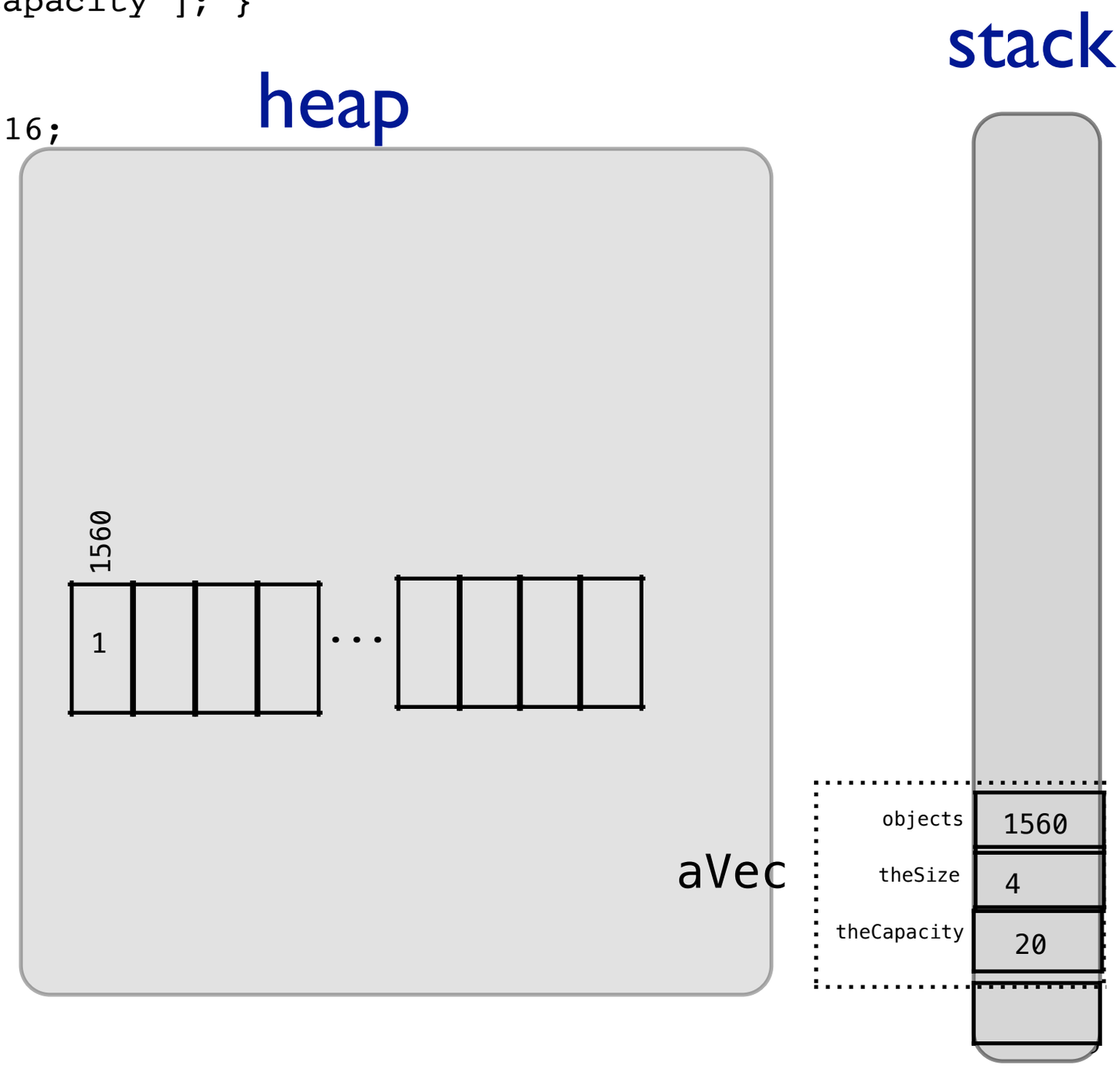
To focus on the idea/method being discussed, the other methods will be replaced by ...

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

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


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

    return 0;
 }
```



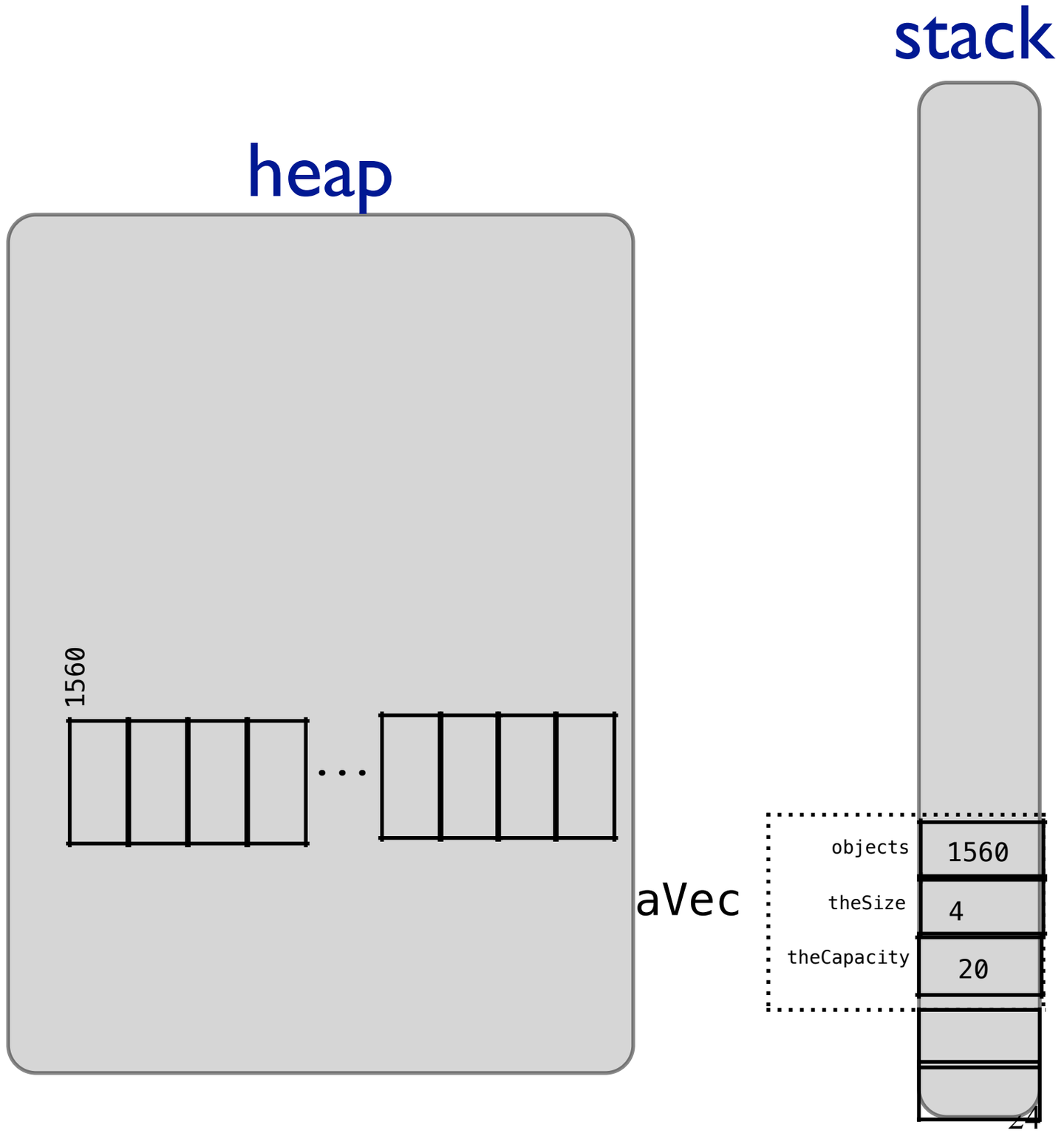
How would you create a vector class constructor?

```
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;
    aVec[4] = 1;
   return 0;
}
```



Do we need to write a destructor?

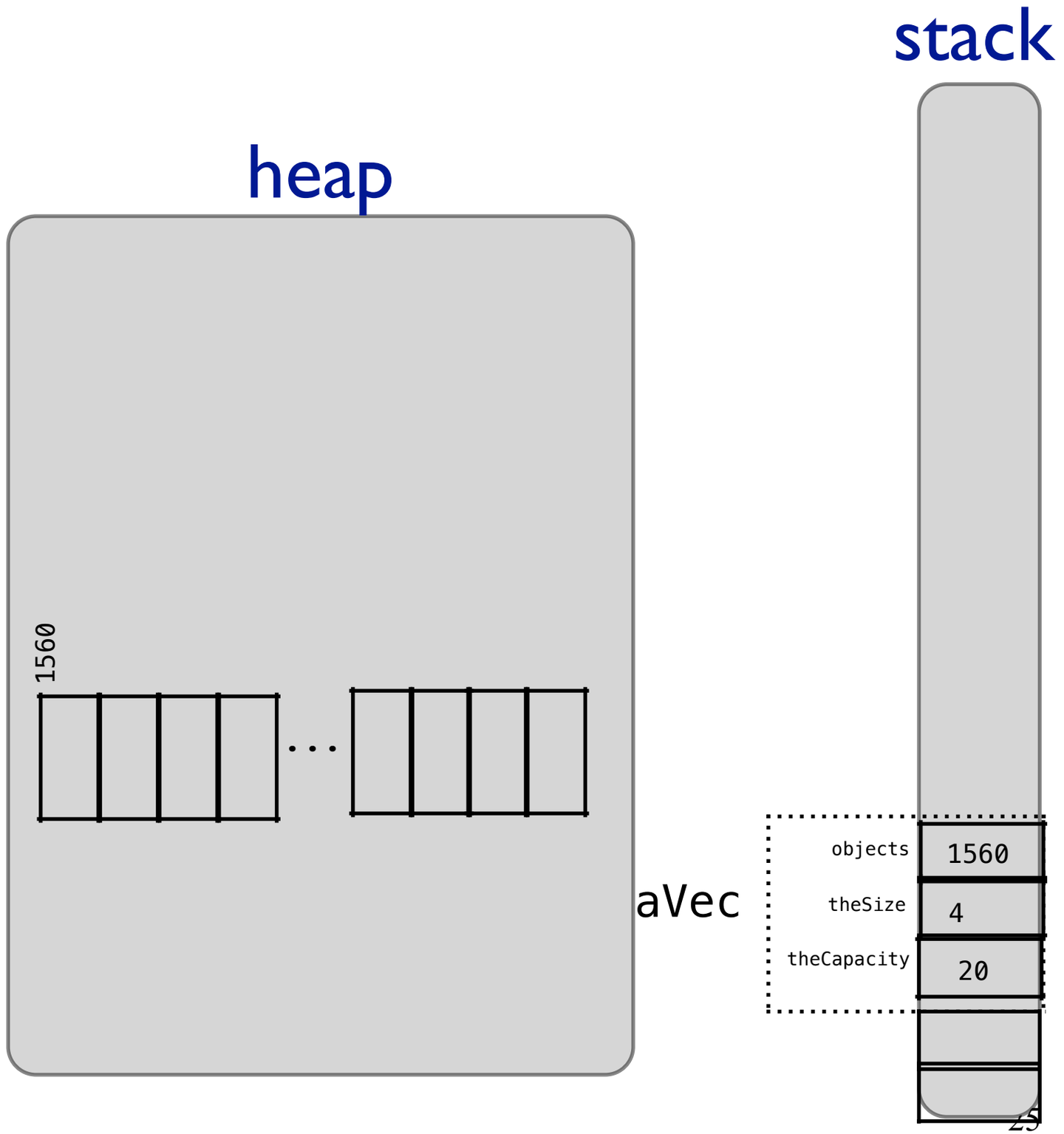
```
template <class Object>
class Vector
{
  public:


    ...


  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:

    ...
    ~Vector( )
      { delete [ ] objects; }
    ...

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

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

int main{

   silly();
   return 0;
}
```



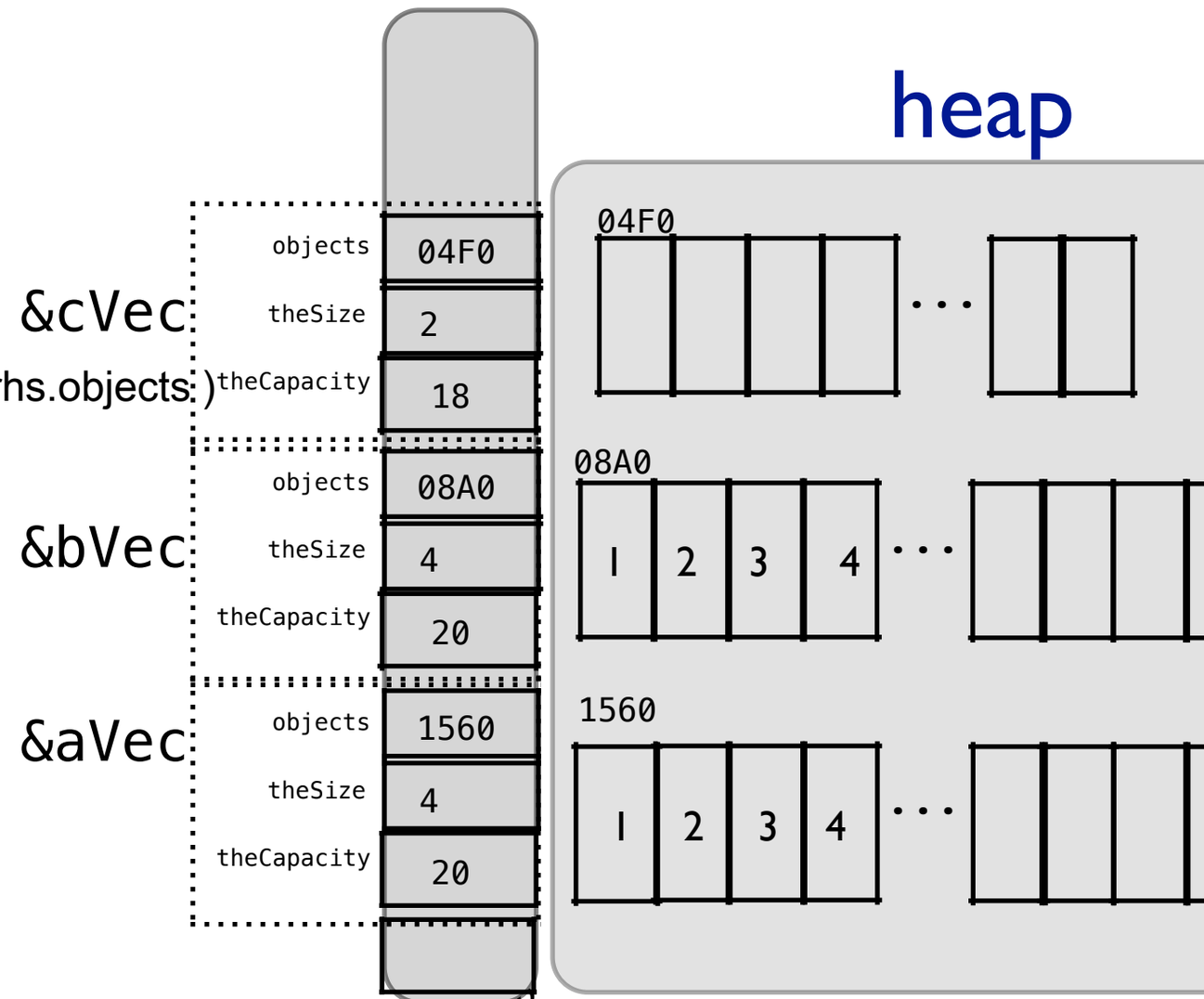
How would you create a vector class copy constructor and move constructor?

```
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 ?

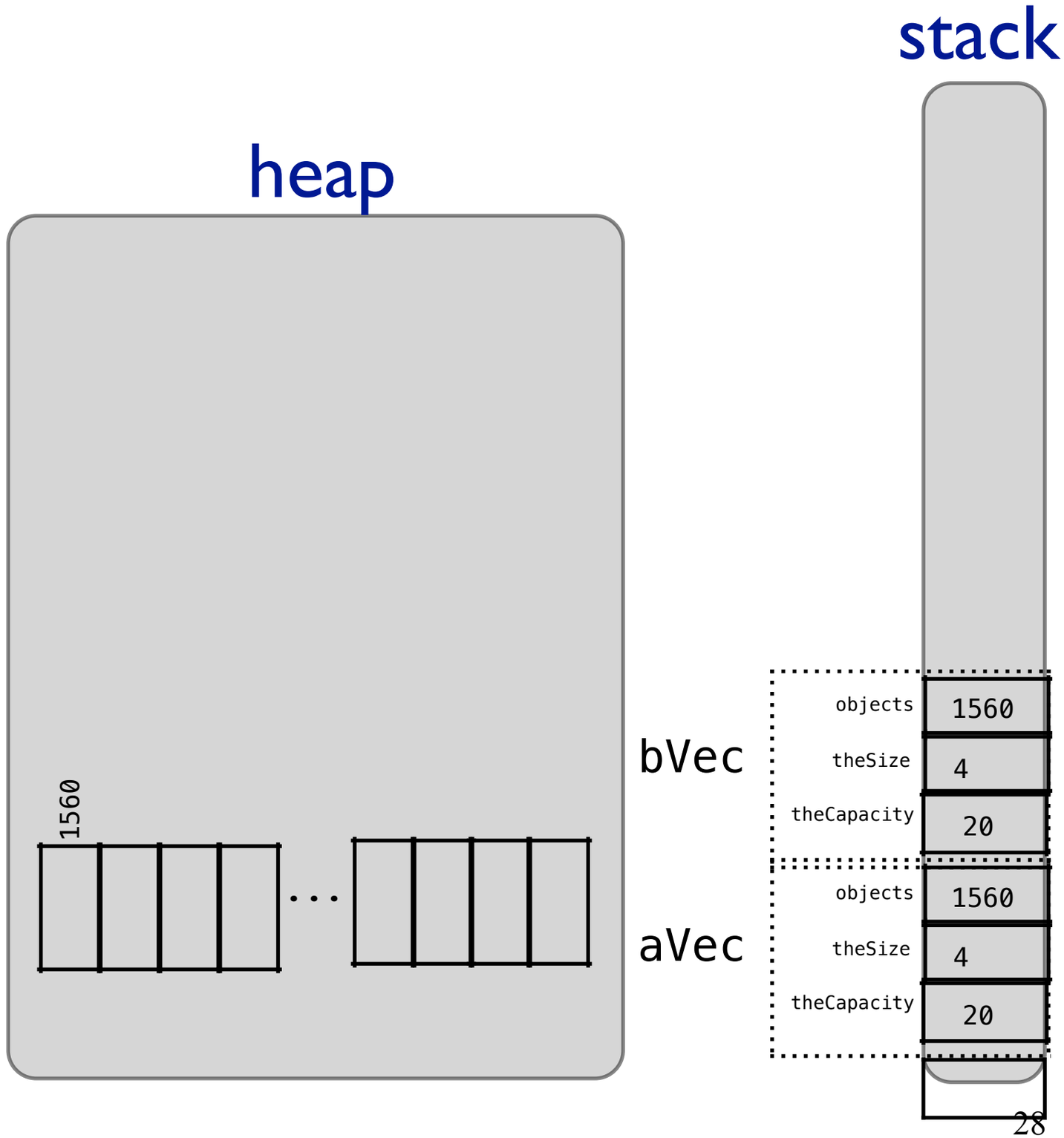
determine the size ?

determine if the vector is empty?

return the i^{th} item?

Do we need to create an operator=?

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

    ...

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



int main{

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

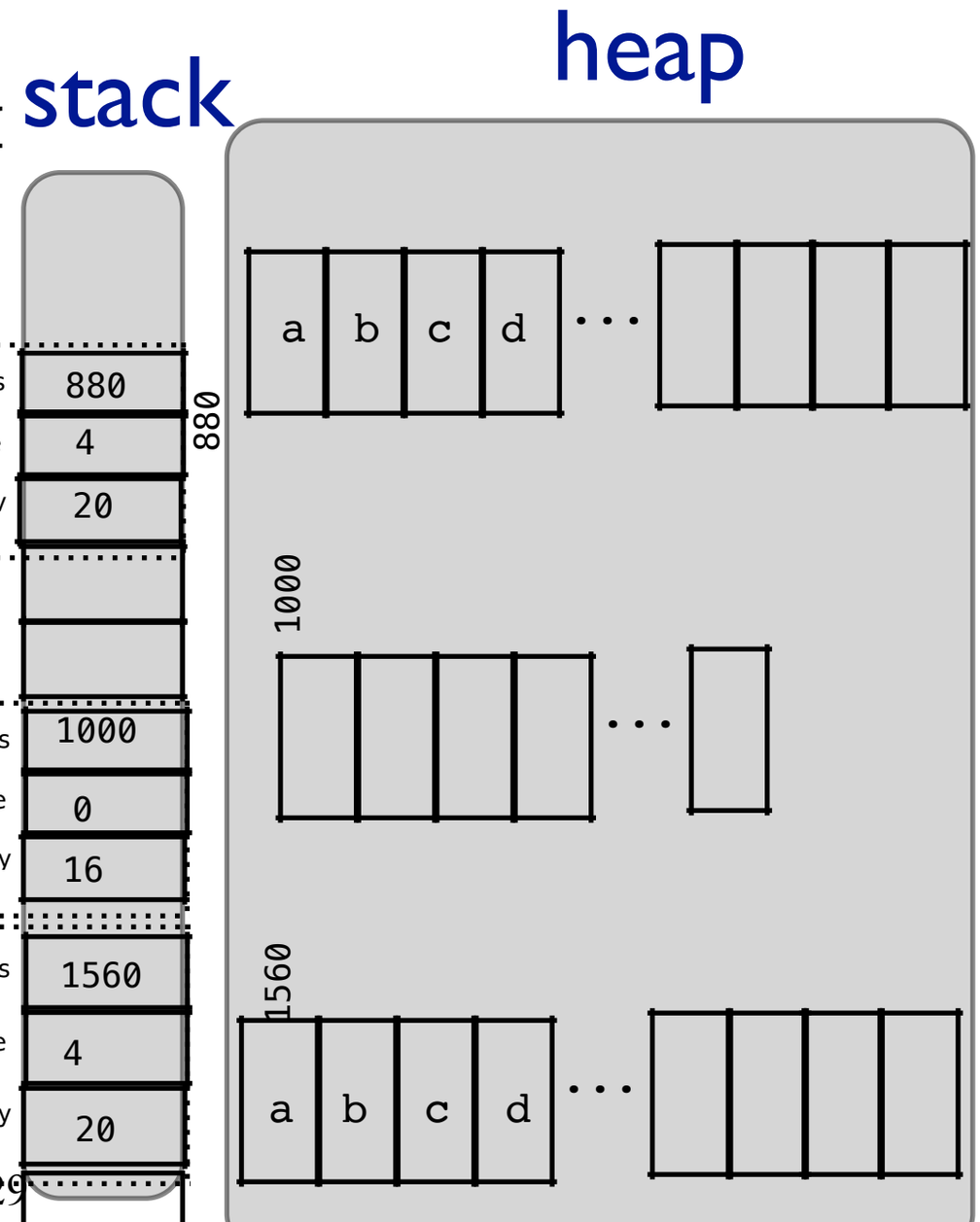


Operator= Method

```
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;
}
```

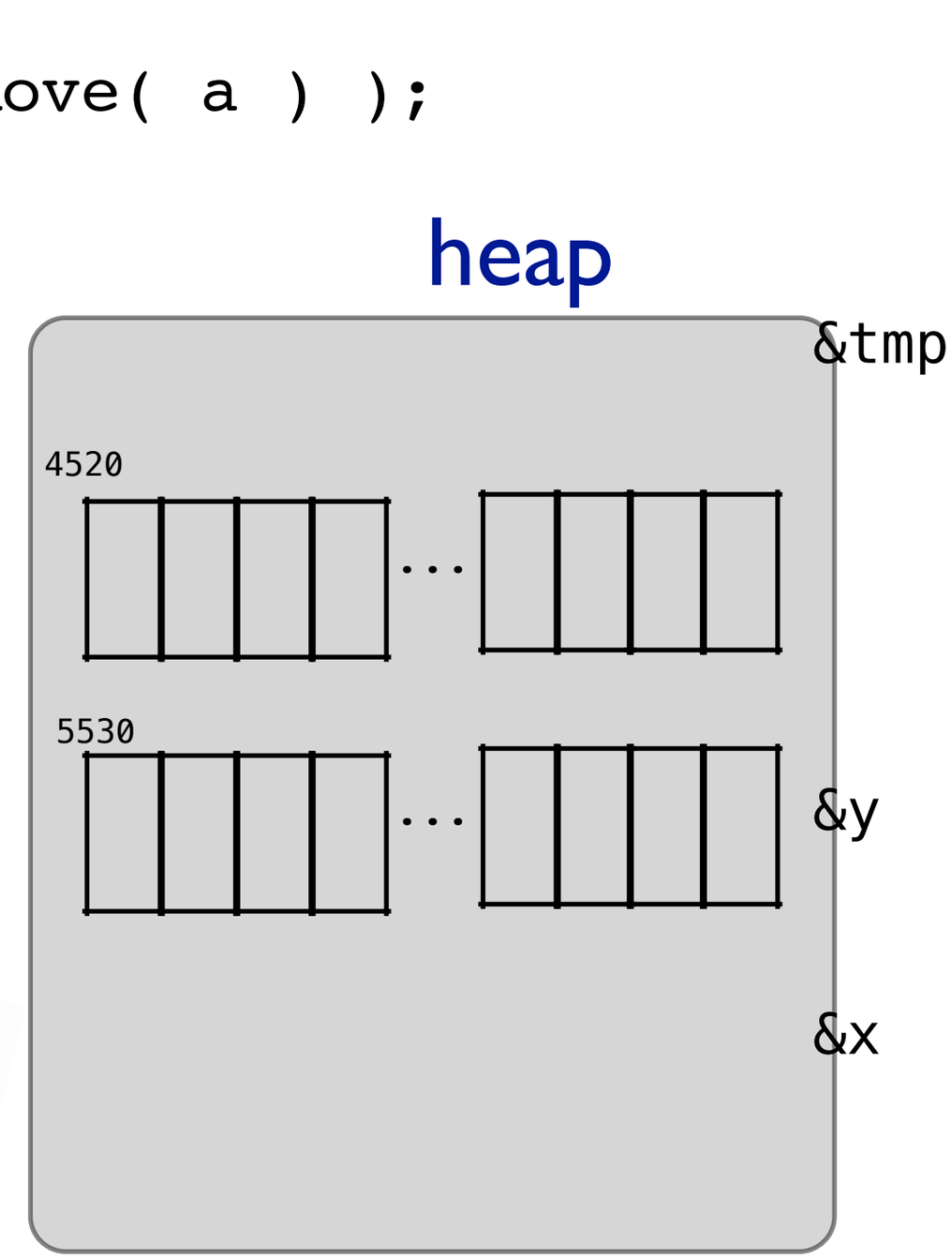
```
Vector<char> aVec = { 'a', 'b', 'c', 'd' }
Vector<char> bVec;
 bVec = aVec;
 bVec =Vector<char>(3);
```



Using the move constructor and the Move assignment operator

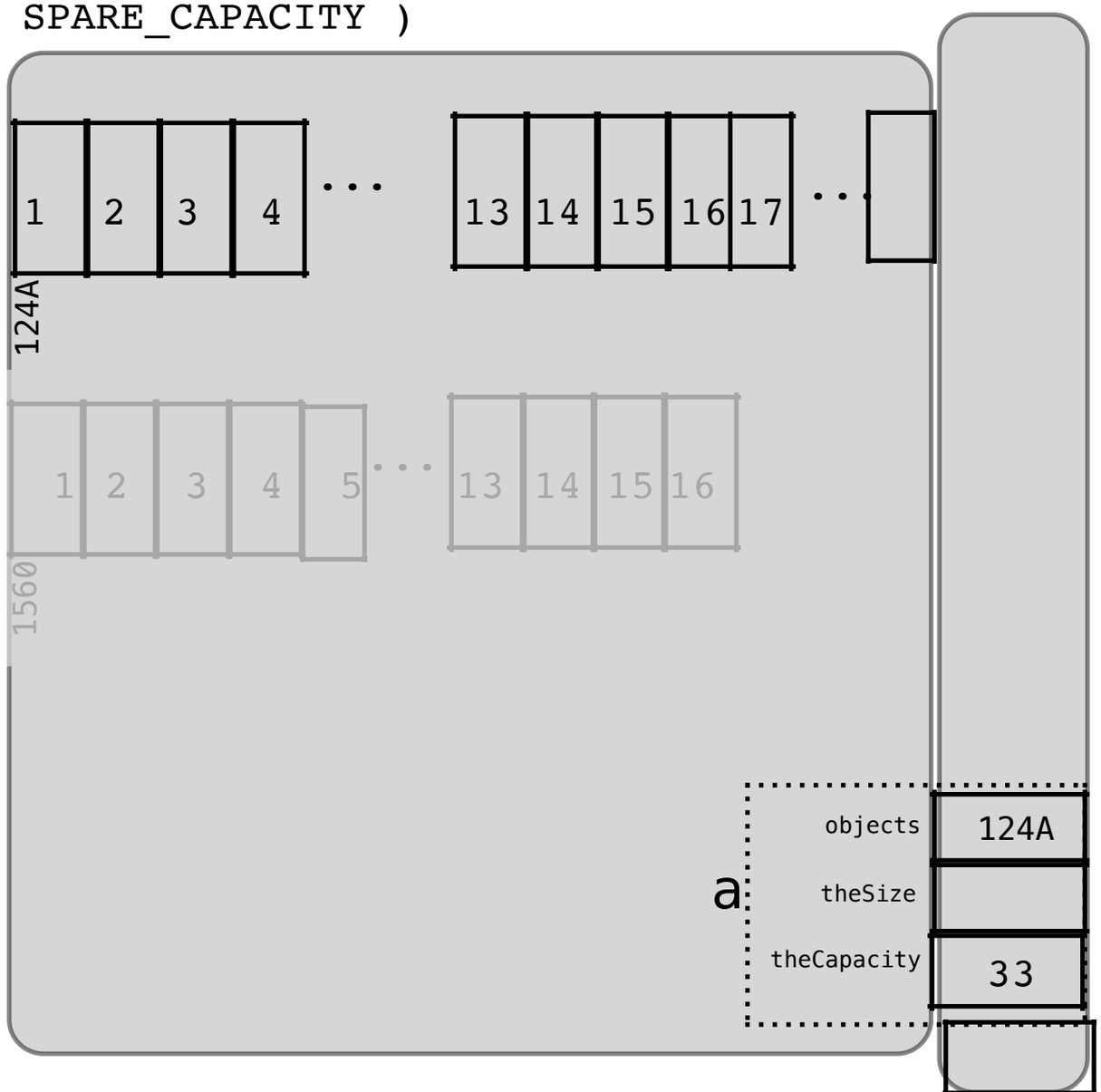
```
void swap(vector<int> & a, vector<int> & b)
{
    vector<int> tmp(std::move( a ) );
    a = std::move(b);
    b = std::move(tmp);
}

int main( )
{
  vector<int> x(400);
  vector<int> y(400);
  // code …
  swap( x, y );
```

If the type of the object you want to move the resources from doesn't support moving the resources, you will copy the object

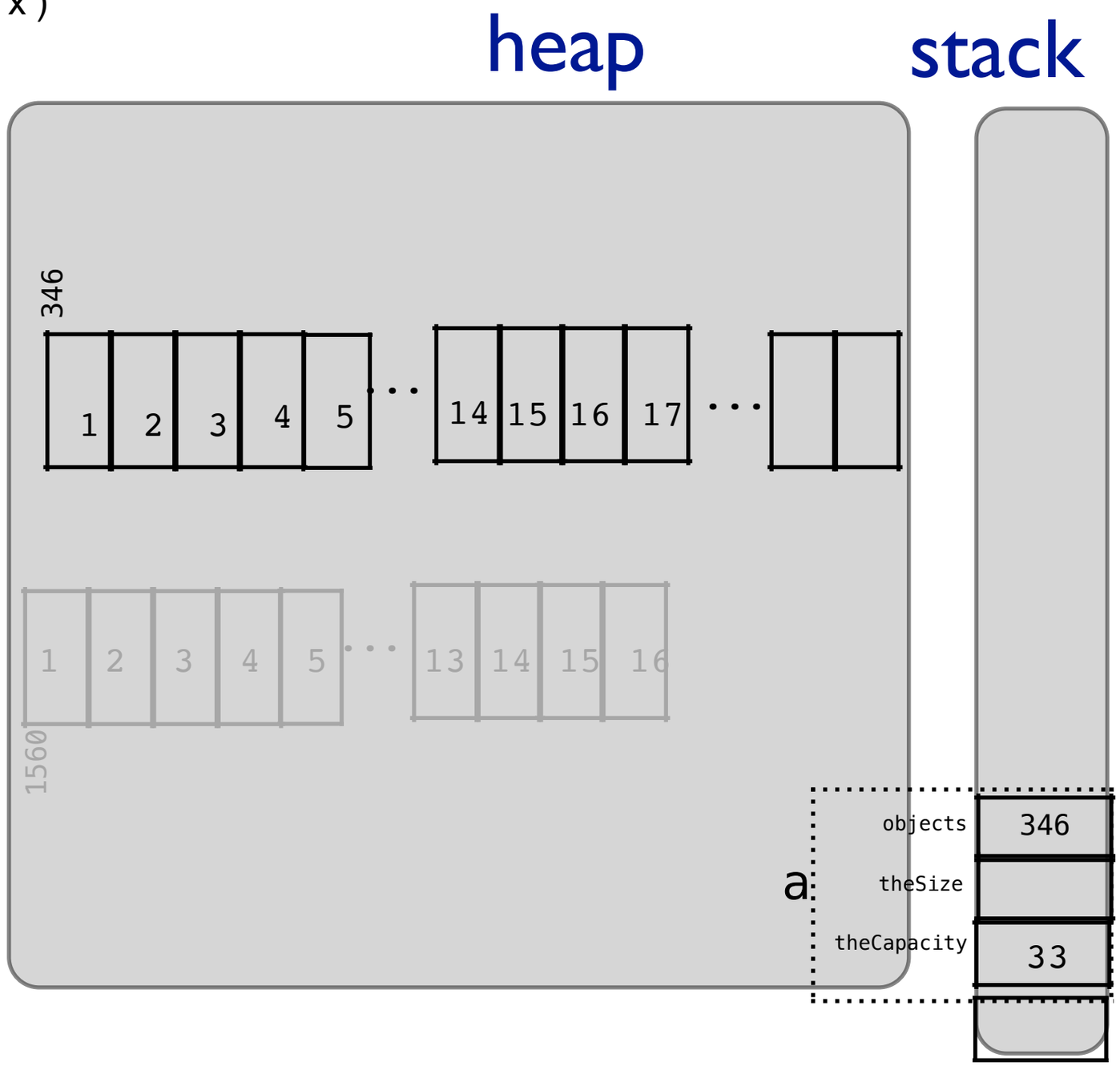


push_back and reserve methods for the Vector class

```
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;
};


 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)

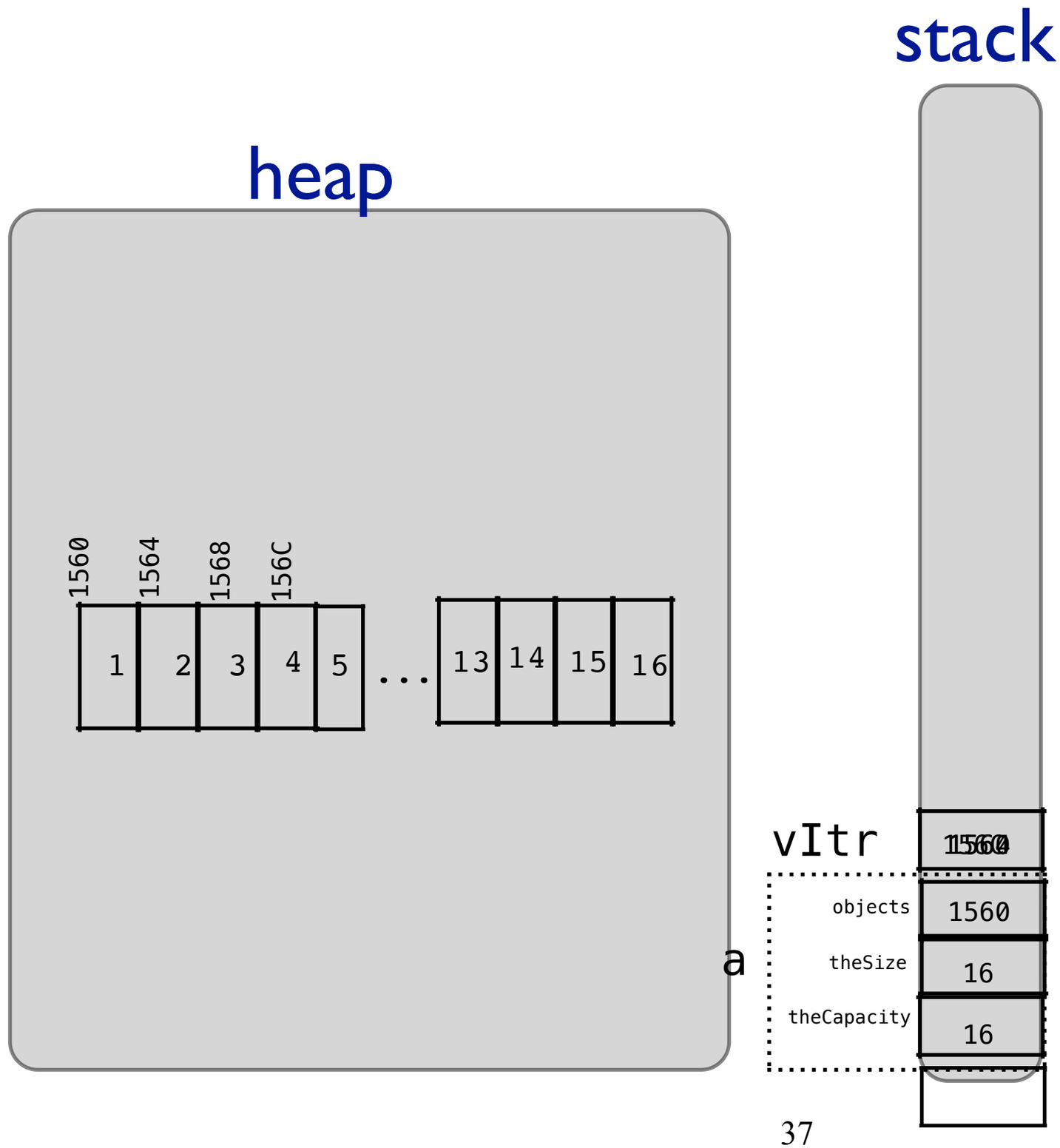
iterate |ˈitəˌrāt|
verb [with obj.]
perform or utter repeatedly.
• [no obj.] make repeated use of a mathematical or computational procedure, applying it each time to the result of the previous application; perform iteration.

From the dictionary on my computer:)

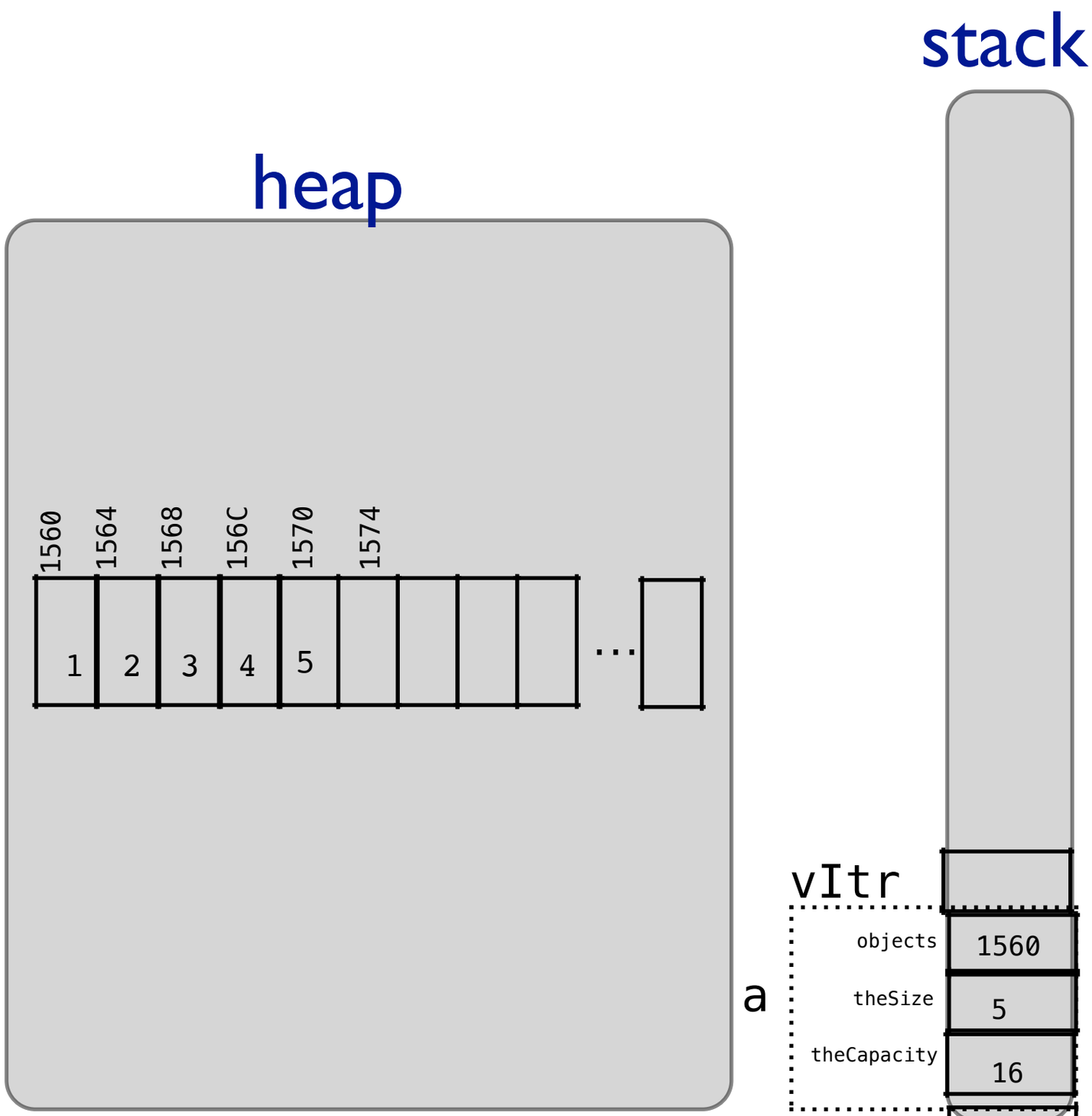
Creating a Vector Iterator

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;
```

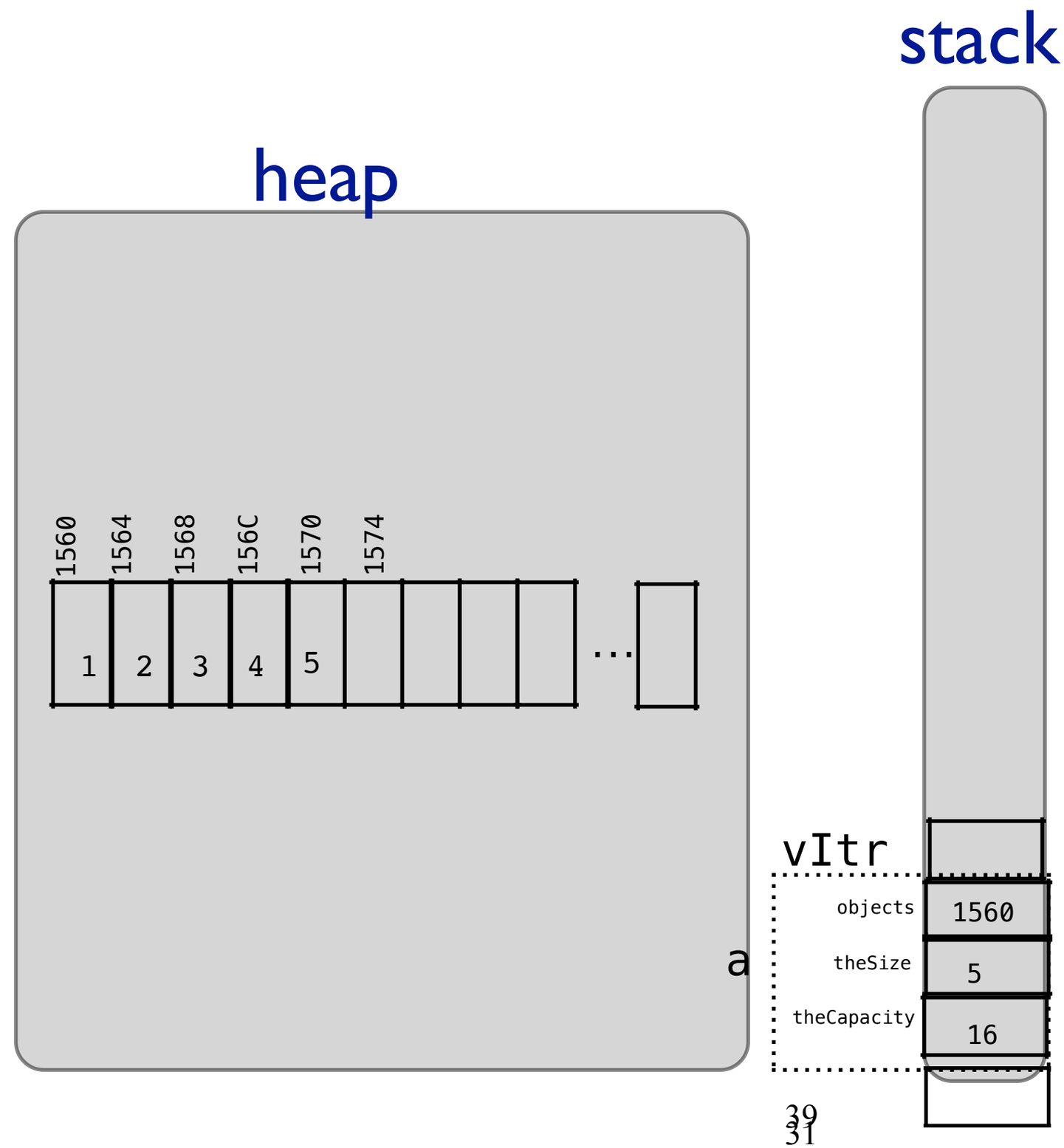


```
int main(void)
{   vector<int> a;
    a.push_back(1);
    a.push_back(2);
       ...
    a.push_back(5);
    vector<int>::iterator vItr = a.begin( );

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



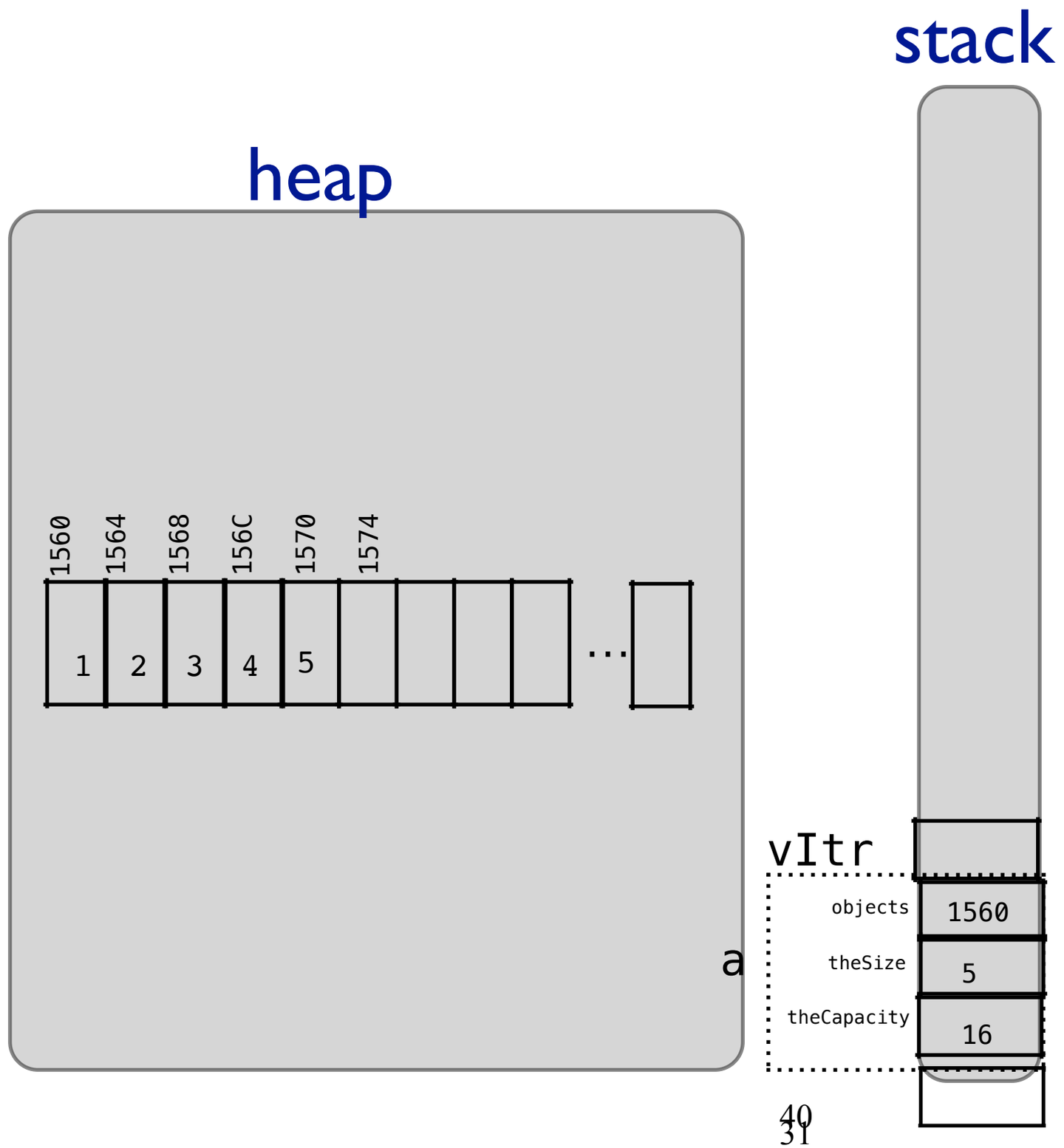
```
int main(void)
{
    Vector<int> a;

    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;
```



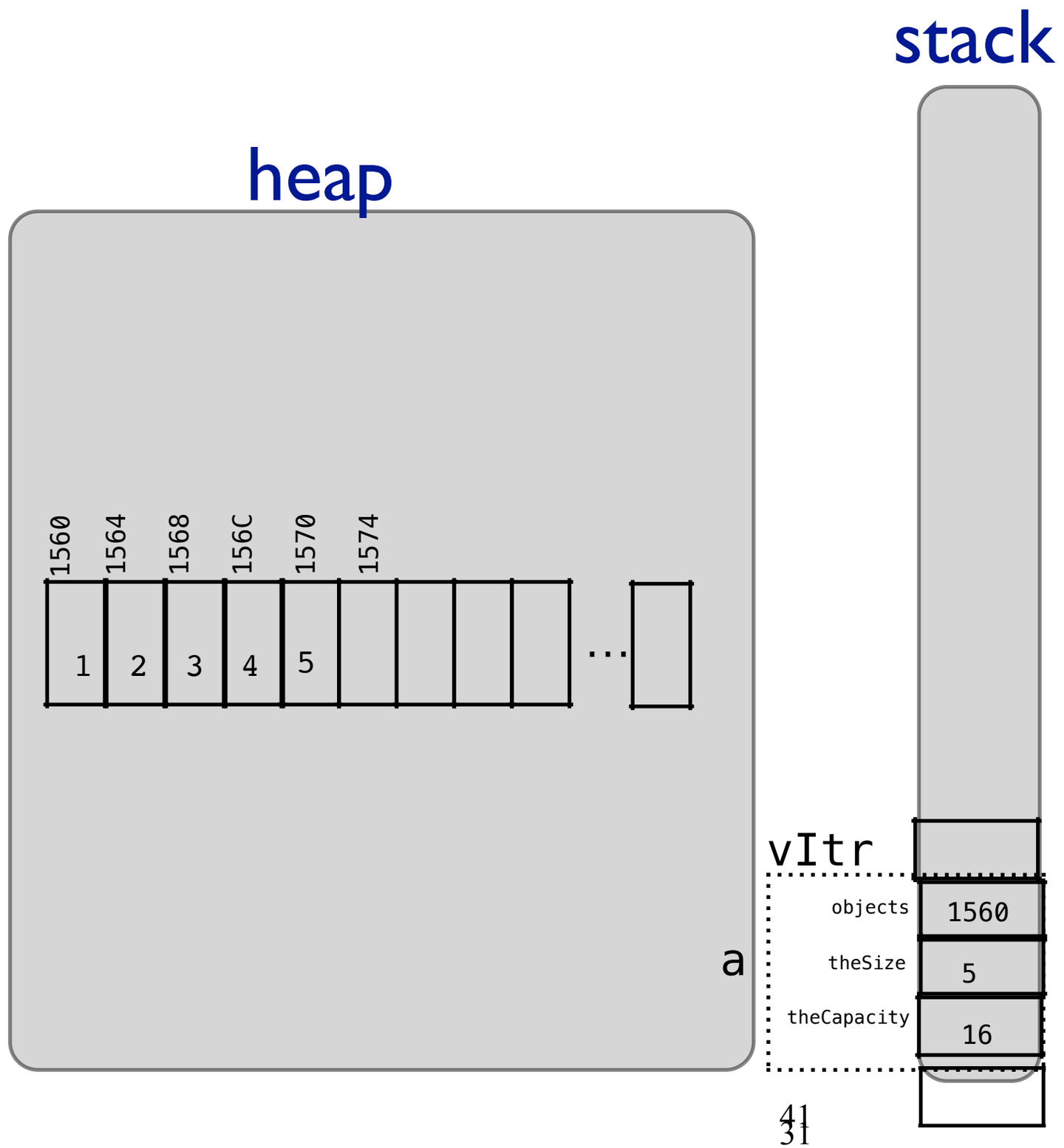
```
int main(void)
{
    Vector<int> a;

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

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

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

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

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



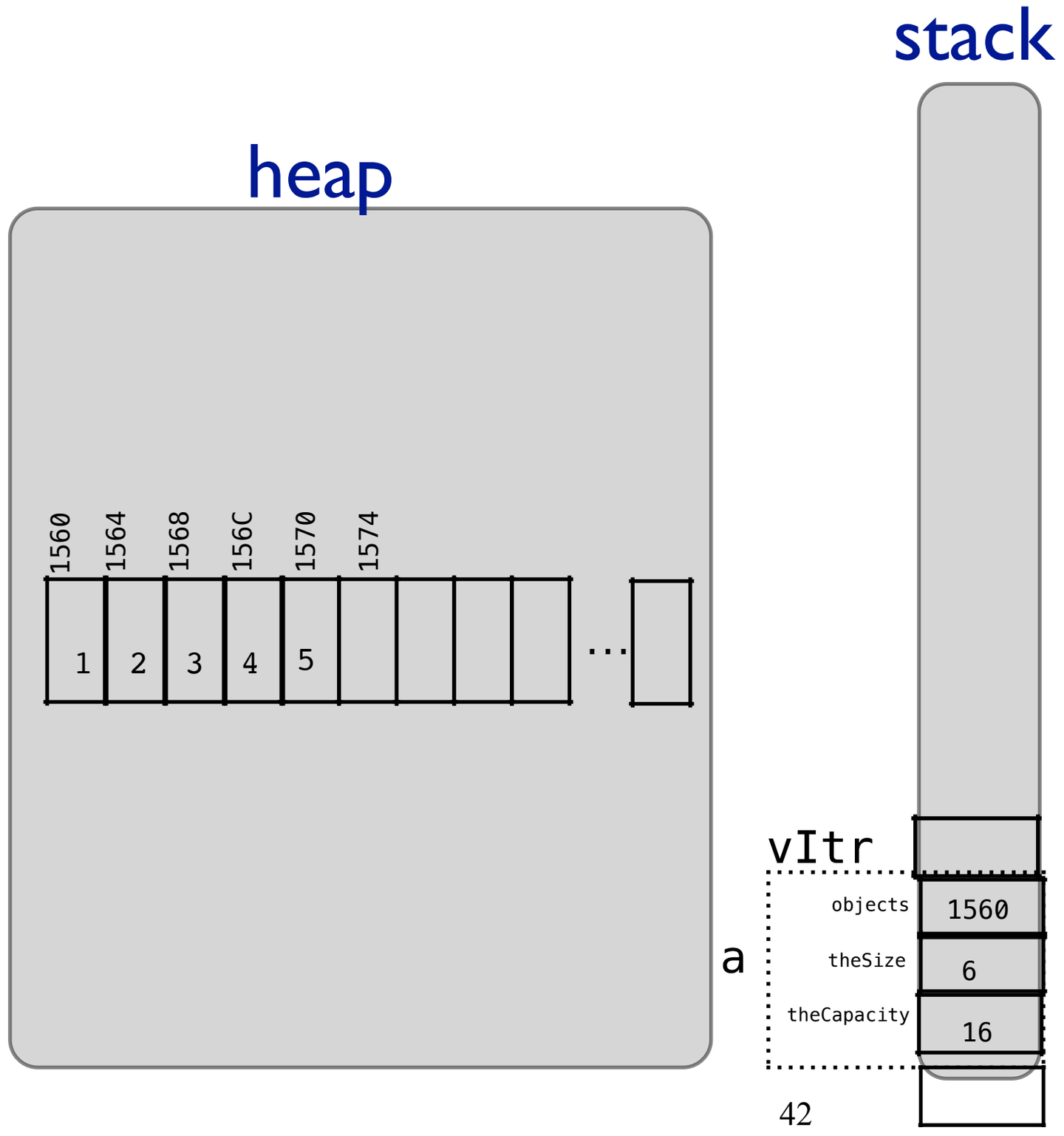
```
int main(void)
{
    Vector<int> a;
    a.push_back(1);
    a.push_back(2);
      ...
    a.push_back(5);
    Vector<int>::iterator vItr = a.end( );

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

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

oops!!!



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

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

mid = (a.end() - a.begin() -1)/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;
}
```

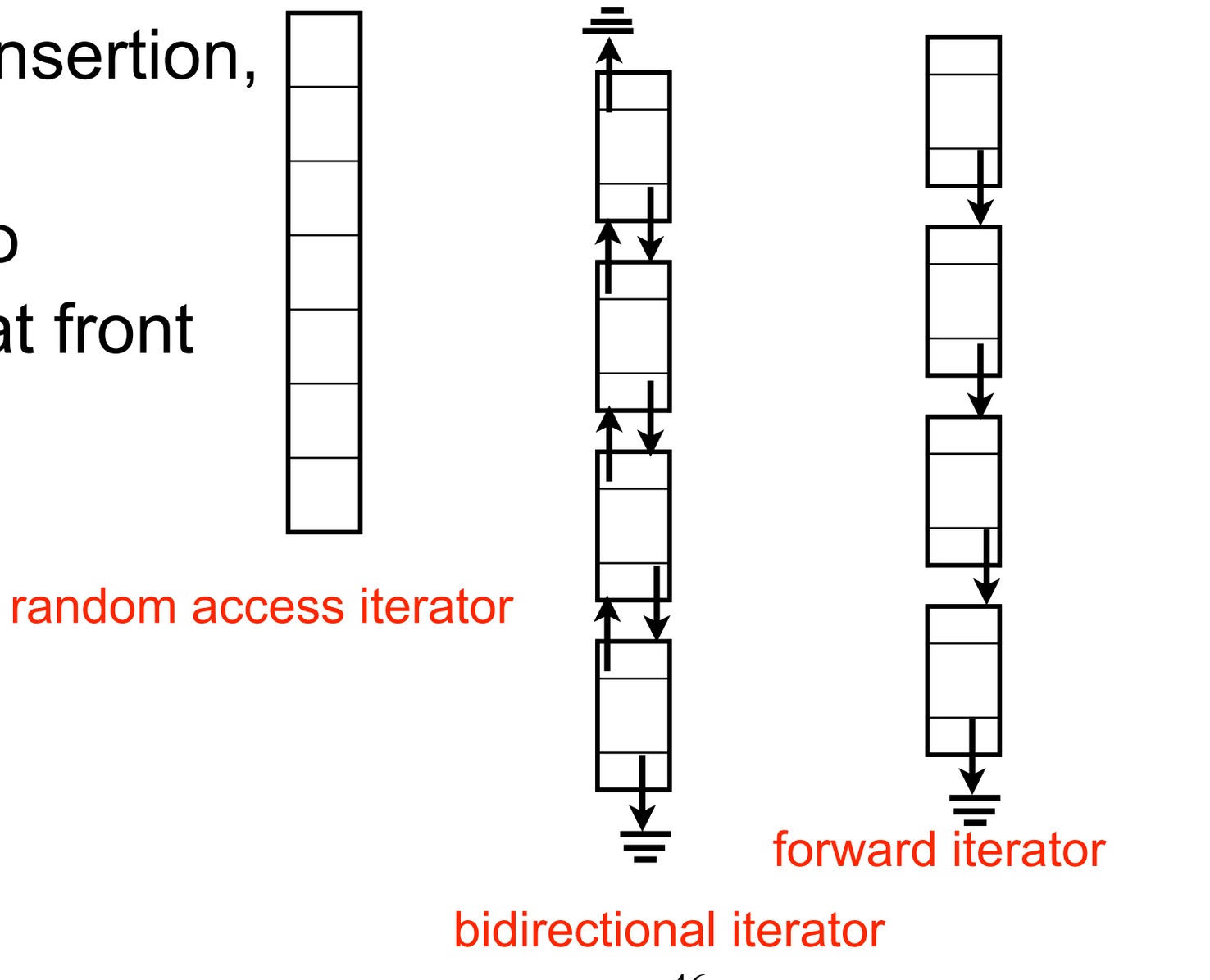
```
  Vector<int> vec_of_int(100);
  Vector<int>::iterator vecItr;

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++;
}
```

Additional Information

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>

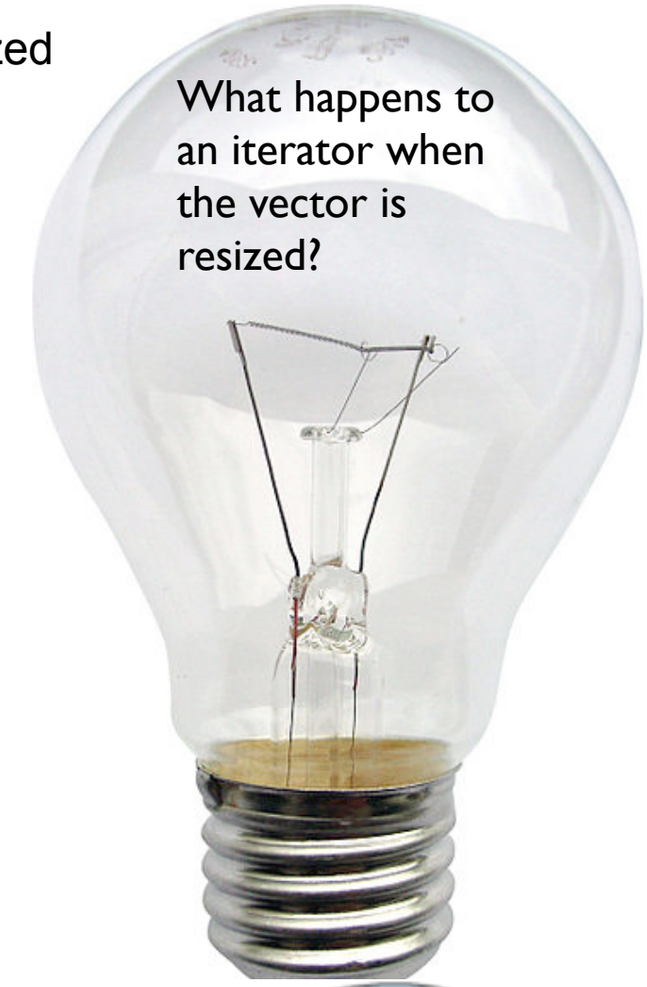
#include<list>

Some methods in the vector and list classes

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

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)



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(n)
- 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