容器与算法

主讲教师： 沈艳艳

Vector: Changing Size

Changing vector size

- Fundamental problem addressed
  - We (humans) want abstractions that can change size (*e.g.*, a vector where we can change the number of elements). However, in computer memory everything must have a fixed size, so how do we create the illusion of change?
- Given

```
vector v(n);          // v.size()==n
```

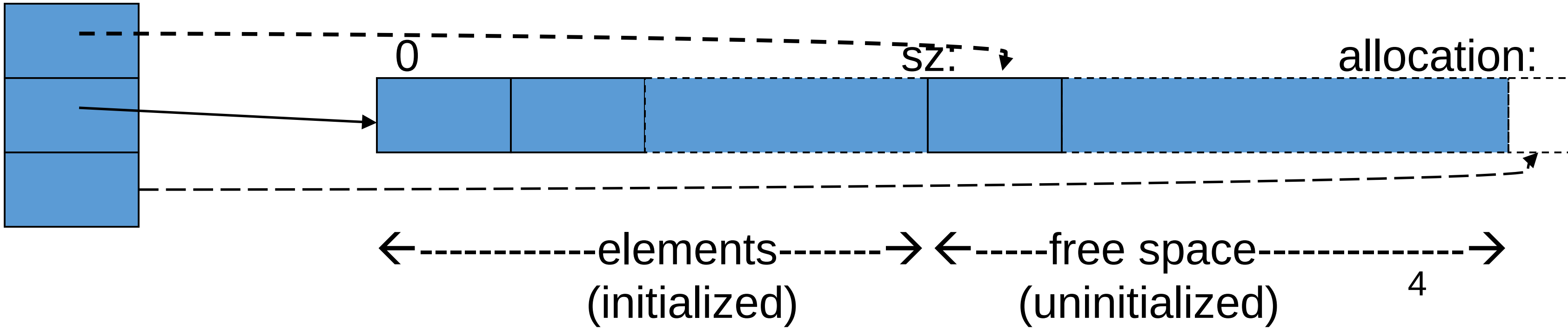
we can change its size in three ways

- Resize it
  - **v.resize(10);** ***//*** ***v*** *now has* ***10*** *elements*
- Add an element
  - **v.push_back(7);** ***//*** *add an element with the value* ***7*** *to the end of* ***v***
    // ***v.size()*** *increases by* ***1***
- Assign to it
  - **v = v2;** ***//*** ***v*** *is now a copy of* ***v2***
    // ***v.size()*** *now equals* ***v2.size()***

Representing vector

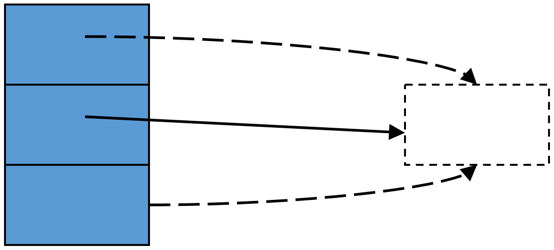
- If you **resize()** or **push_back()** once, you'll probably do it again;
  - let's prepare for that by sometimes keeping a bit of free space for future expansion

```
class vector {
  int sz;
  double* elem;
  int space;   // number of elements plus "free space"
               // (the number of "slots" for new elements)
public:
  // …
};
```

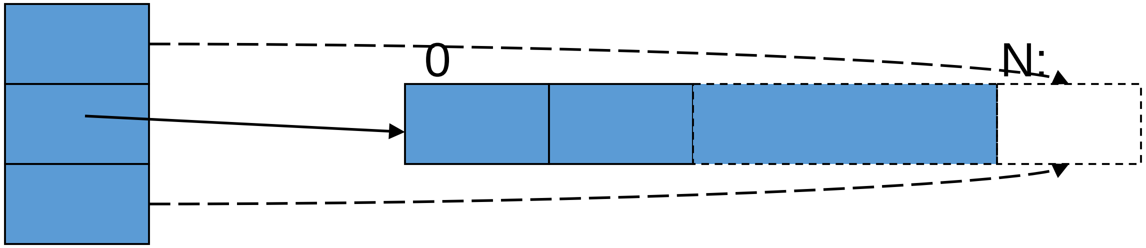


Representing vector

- An empty vector (no free store use):



- A vector(n) (no free space):



vector::reserve()

- First deal with space (allocation); given space all else is easy
  - Note: **reserve()** doesn't mess with size or element values

```
void vector::reserve(int newalloc)
  // make the vector have space for newalloc elements
{
  if (newalloc<=space) return;             // never decrease allocation
  double* p = new double[newalloc];        // allocate new space
  for (int i=0; i<sz; ++i) p[i]=elem[i];   // copy old elements
  delete[ ] elem;                          // deallocate old space
  elem = p;
  space = newalloc;
}
```

vector::resize()

- Given **reserve()**, **resize()** is easy
  - **reserve()** deals with space/allocation
  - **resize()** deals with element values

```
void vector::resize(int newsize)
  // make the vector have newsize elements
  // initialize each new element with the default value 0.0
{
  reserve(newsize);             // make sure we have sufficient space
  for(int i = sz; i<newsize; ++i) elem[i] = 0;       // initialize new elements
  sz = newsize;
}
```

vector::push_back()

- Given **reserve()**, **push_back()** is easy
  - **reserve()** deals with space/allocation
  - **push_back()** just adds a value

```
void vector::push_back(double d)
  // increase vector size by one
  // initialize the new element with d
{
  if (sz==0)                    // no space: grab some
        reserve(8);
  else if (sz==space)           // no more free space: get more space
        reserve(2*space);
  elem[sz] = d;                 // add d at end
  ++sz;                         // and increase the size (sz is the number of elements)
}
```

resize() and push_back()

```
class vector {   // an almost real vector of doubles
  int sz;                          // the size
  double* elem;                    // a pointer to the elements
  int space;                       // size+free_space
public:
  // … constructors and destructors …

  double& operator[ ](int n) { return elem[n]; }      // access: return reference
  int size() const { return sz; }                      // current size

  void resize(int newsize);                            // grow
  void push_back(double d);                            // add element

  void reserve(int newalloc);                          // get more space
  int capacity() const { return space; }               // current available space
};
```

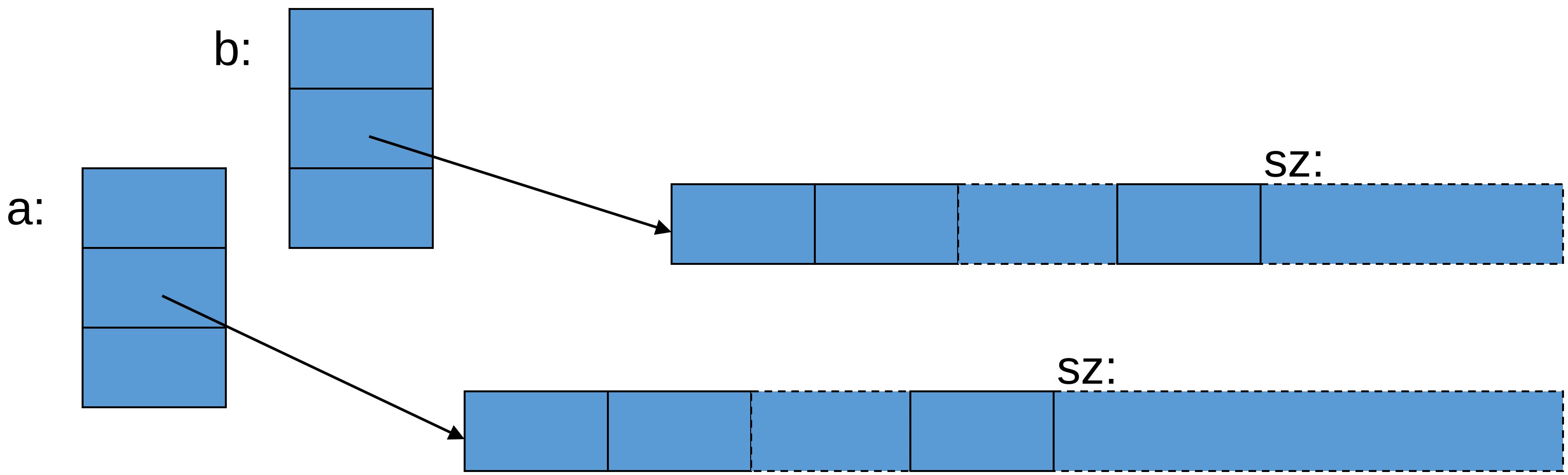
Assignment

- Copy and swap is a powerful general idea

```
vector& vector::operator=(const vector& a)
  // like copy constructor, but we must deal with old elements
  // make a copy of a then replace the current sz and elem with a's
{
  double* p = new double[a.sz];                    // allocate new space
  for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];   // copy elements
  delete[ ] elem;                                  // deallocate old space
  sz = a.sz;                                       // set new size
  elem = p;                                        // set new elements
  return *this;                                    // return a self-reference
}
```

Optimize assignment

- "Copy and swap" is the most general idea
  - but not always the most efficient
  - What if there already is sufficient space in the target vector?
    - Then just copy!
    - For example: **a = b;**



Optimized assignment

```
vector& vector::operator=(const vector& a)
{
  if (this==&a) return *this;     // self-assignment, no work needed

  if (a.sz<=space) {               // enough space, no need for new allocation
        for (int i = 0; i<a.sz; ++i) elem[i] = a.elem[i];    // copy elements
        space += sz-a.sz;                                    // increase free space
        sz = a.sz;
        return *this;
  }

  double* p = new double[a.sz];                              // copy and swap
  for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];
  delete[ ] elem;
  sz = a.sz;
  space = a.sz;
  elem = p;
  return *this;
}
```

Templates and Range Checking

Templates

- But we don’t just want vector of double
- We want vectors with element types we specify
  - **vector<double>**
  - **vector<int>**
  - **vector<Month>**
  - **vector<Record*>** **//** *vector of pointers*
  - **vector<vector<Record>>** **//** *vector of vectors*
  - **vector<char>**
- We must make the element type a parameter to **vector**
- **vector** must be able to take both built-in types and user-defined types as element types
- This is not some magic reserved for the compiler; we can define our own parameterized types, called “templates”

Templates

- The basis for generic programming in C++
  - Sometimes called “parametric polymorphism”
    - Parameterization of types (and functions) by types (and integers)
  - Unsurpassed flexibility and performance
    - Used where performance is essential (*e.g.,* hard real time and numerics)
    - Used where flexibility is essential (*e.g.,* the C++ standard library)

- Template definitions

```
template<class T, int N> class Buffer { /* … */ };
template<class T, int N> void fill(Buffer<T,N>& b) { /* … */  }
```

- Template specializations (instantiations)

```
// for a class template, you specify the template arguments:
Buffer<char,1024> buf;          // for buf, T is char and N is 1024

// for a function template, the compiler deduces the template arguments:
fill(buf);   // for fill(), T is char and N is 1024; that’s what buf has
```

Parameterize with element type

```
// an almost real vector of Ts:
template<class T> class vector {
  // …
};
vector<double> vd;                   // T is double
vector<int> vi;                      // T is int
vector<vector<int>> vvi;             // T is vector<int>
                                     //           in which T is int
vector<char> vc;                     // T is char
vector<double*> vpd;                 // T is double*
vector<vector<double>*> vvpd;        // T is vector<double>*
                                     //           in which T is double
```

Basically, **vector<double>** is

```
// an almost real vector of doubles:
class vector {
  int sz;                  // the size
  double* elem;            // a pointer to the elements
  int space;               // size+free_space
public:
  vector() : sz(0), elem(0), space(0) { }          // default constructor
  vector(const vector&);                           // copy constructor
  vector& operator=(const vector&);                // copy assignment
  ~vector() { delete[ ] elem; }                    // destructor

  double& operator[ ] (int n) { return elem[n]; }       // access: return
  reference
  int size() const { return sz; }                  // the current size

  // …
};
```

Basically, **vector<char>** is

```
// an almost real vector of chars:
class vector {
  int sz;                 // the size
  char* elem;             // a pointer to the elements
  int space;              // size+free_space
public:
  vector() : sz{0}, elem{0}, space{0} { }          // default constructor
  vector(const vector&);                           // copy constructor
  vector& operator=(const vector&);                // copy assignment
  ~vector() { delete[ ] elem; }                    // destructor

  char& operator[ ] (int n) { return elem[n]; }    // access: return reference
  int size() const { return sz; }                  // the current size

  // …
};
```

Basically, **vector<T>** is

```
// an almost real vector of Ts:
template<class T> class vector {    // read "for all types T" (just like in math)
  int sz;                // the size
  T* elem;               // a pointer to the elements
  int space;             // size+free_space
public:
  vector() : sz{0}, elem{0}, space{0};        // default constructor
  vector(const vector&);                      // copy constructor
  vector& operator=(const vector&);           // copy assignment
  vector(const vector&&);                     // move constructor
  vector& operator=(vector&&);                // move assignment

  ~vector() { delete[ ] elem; }               // destructor

  // …
};
```

Basically, **vector<T>** is

```
// an almost real vector of Ts:
template<class T> class vector {     // read "for all types T" (just like in math)
  int sz;                  // the size
  T* elem;                 // a pointer to the elements
  int space;               // size+free_space
public:
  // … constructors and destructors …

  T& operator[ ] (int n) { return elem[n]; }        // access: return reference
  int size() const { return sz; }                   // the current size

  void resize(int newsize);                         // grow
  void push_back(double d);                         // add element

  void reserve(int newalloc);                       // get more space
  int capacity() const { return space; }            // current available space

  // …
};
```

Templates

- Problems (“there is no free lunch”)
  - Poor error diagnostics
    - Often spectacularly poor (but getting better in C++11; much better in C++14)
  - Delayed error messages
    - Often at link time
  - All templates must be fully defined in each translation unit
    - So place template definitions in header files
- Recommendation
  - Use template-based libraries
    - Such as the C++ standard library
      - *E.g.,* **vector**, **sort()**
      - Soon to be described in some detail
  - Initially, write only very simple templates yourself
    - Until you get more experience

Range checking

```
// an almost real vector of Ts:

struct out_of_range { /* … */ };

template<class T> class vector {
  // …
  T& operator[ ](int n);                // access
  // …
};

template<class T>  T& vector<T>::operator[ ](int n)
{
  if (n<0 || sz<=n) throw out_of_range();
  return elem[n];
}
```

Range checking

```
void fill_vec(vector<int>& v, int n)          // initialize v with factorials
{
  for (int i=0; i<n; ++i) v.push_back(factorial(i));
}

int main()
{
  vector<int> v;
  try {
        fill_vec(v,10);
        for (int i=0; i<=v.size(); ++i)
                cout << "v[" << i << "]==" << v[i] << '\n';
  }
  catch (out_of_range) {          // we'll get here (why?)
        cout << "out of range error";
        return 1;
  }
}
```

Exception handling

- We use exceptions to report errors
- We must ensure that use of exceptions
  - Doesn’t introduce new sources of errors
  - Doesn’t complicate our code
  - Doesn’t lead to resource leaks

STL
(The containers, iterators and algorithms)

- STL – the containers and algorithms part of the C++ standard library

Common tasks

- Collect data into containers
- Organize data
  - For printing
  - For fast access
- Retrieve data items
  - By index (e.g., get the **N**th element)
  - By value (e.g., get the first element with the value **"Chocolate"**)
  - By properties (e.g., get the first elements where “**age<64**”)
- Add data
- Remove data
- Sorting and searching
- Simple numeric operations

Observation

We can (already) write programs that are very similar independent of the data type used

- Using an **int** isn't that different from using a **double**
- Using a **vector<int>** isn't that different from using a **vector<string>**

Ideals

We'd like to write common programming tasks so that we don't have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data

- Finding a value in a **vector** isn't all that different from finding a value in a **list** or an array
- Looking for a **string** ignoring case isn't all that different from looking at a **string** not ignoring case
- Graphing experimental data with exact values isn't all that different from graphing data with rounded values
- Copying a file isn't all that different from copying a vector

Ideals (continued)

- Code that's
  - Easy to read
  - Easy to modify
  - Regular
  - Short
  - Fast
- Uniform access to data
  - Independently of how it is stored
  - Independently of its type
- …

Ideals (continued)

- …
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
  - Retrieval of data
  - Addition of data
  - Deletion of data
- Standard versions of the most common algorithms
  - Copy, find, search, sort, sum, …

Examples

- Sort a vector of strings
- Find an number in a phone book, given a name
- Find the highest temperature
- Find all values larger than 800
- Find the first occurrence of the value 17
- Sort the telemetry records by unit number
- Sort the telemetry records by time stamp
- Find the first value larger than “Petersen”?
- What is the largest amount seen?
- Find the first difference between two sequences
- Compute the pairwise product of the elements of two sequences
- What are the highest temperatures for each day in a month?
- What are the top 10 best-sellers?
- What’s the entry for “C++” (say, in Google)?
- What’s the sum of the elements?

The STL

- Part of the ISO C++ Standard Library
- Mostly non-numerical
  - Only 4 standard algorithms specifically do computation
    - Accumulate, inner_product, partial_sum, adjacent_difference
  - Handles textual data as well as numeric data
    - E.g. string
  - Deals with organization of code and data
    - Built-in types, user-defined types, and data structures
- Optimizing disk access was among its original uses
  - Performance was always a key concern

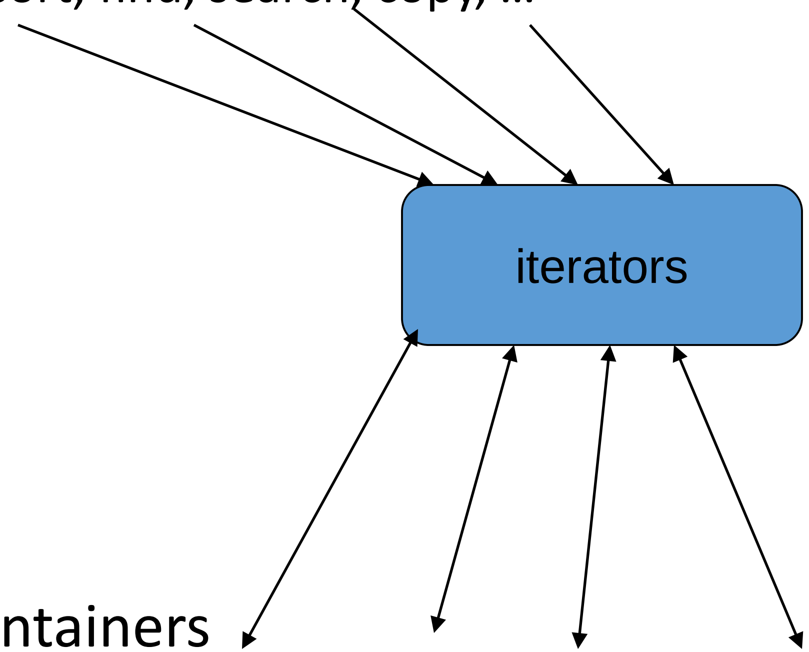
The STL

- Designed by Alex Stepanov
- General aim: The most general, most efficient, most flexible representation of concepts (ideas, algorithms)
  - Represent separate concepts separately in code
  - Combine concepts freely wherever meaningful
- General aim to make programming "like math"
  - or even "Good programming *is* math"
  - works for integers, for floating-point numbers, for polynomials, for …

Basic model

- Algorithms

  sort, find, search, copy, …



- Containers

  vector, list, map, unordered_map, …

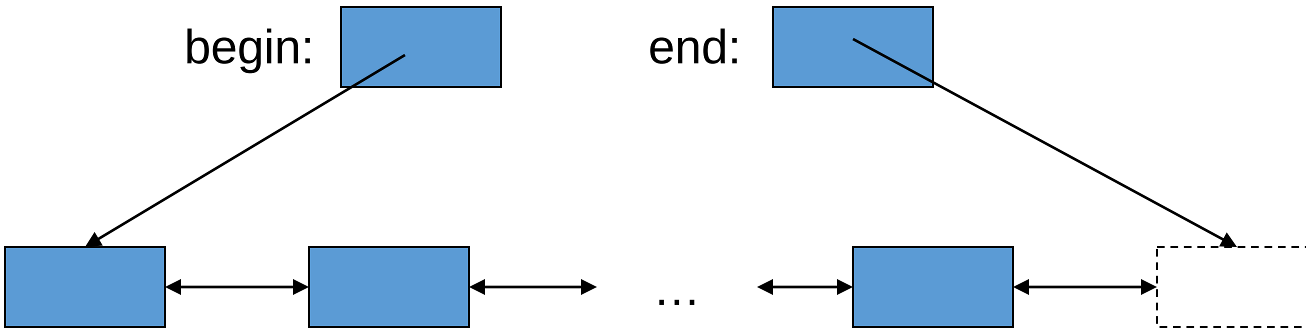
- Separation of concerns
  - Algorithms manipulate data, but don't know about containers
  - Containers store data, but don't know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types

The STL

- An ISO C++ standard framework of about 10 containers and about 60 algorithms connected by iterators
  - Other organizations provide more containers and algorithms in the style of the STL
    - Boost.org, Microsoft, SGI, …
- Probably the currently best known and most widely used example of generic programming

Basic model

- A pair of iterators defines a sequence
  - The beginning (points to the first element – if any)
  - The end (points to the one-beyond-the-last element)

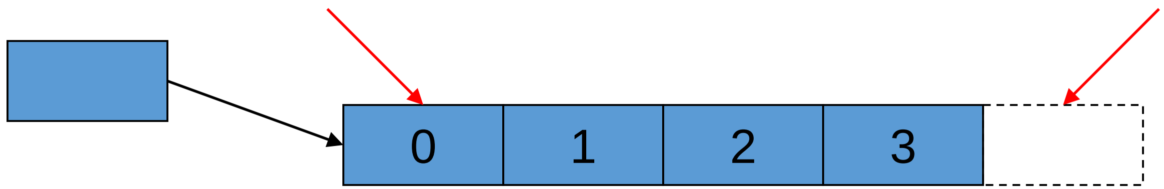


- An iterator is a type that supports the "iterator operations"
  - ++ Go to next element
  - * Get value
  - == Does this iterator point to the same element as that iterator?
- Some iterators support more operations (e.g. --, +, and [ ])

Containers

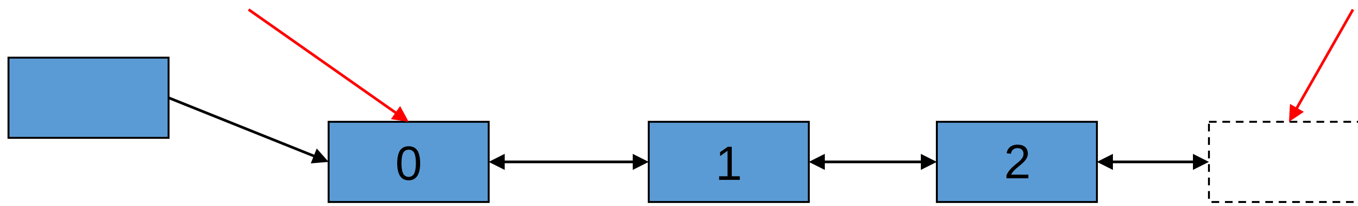
(hold sequences in difference ways)

- **vector**



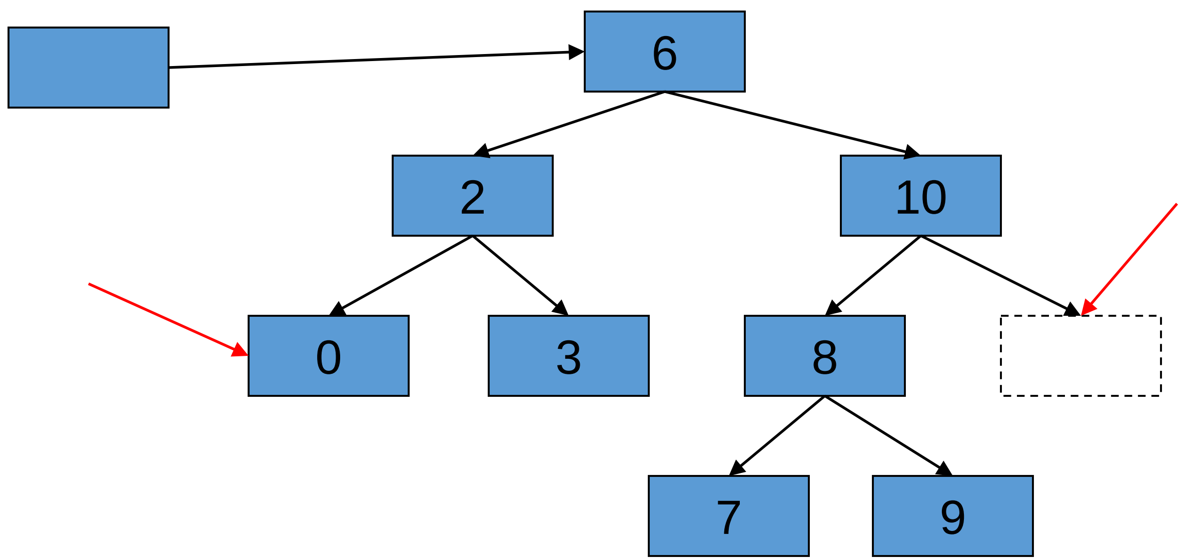
- **list**

(doubly linked)

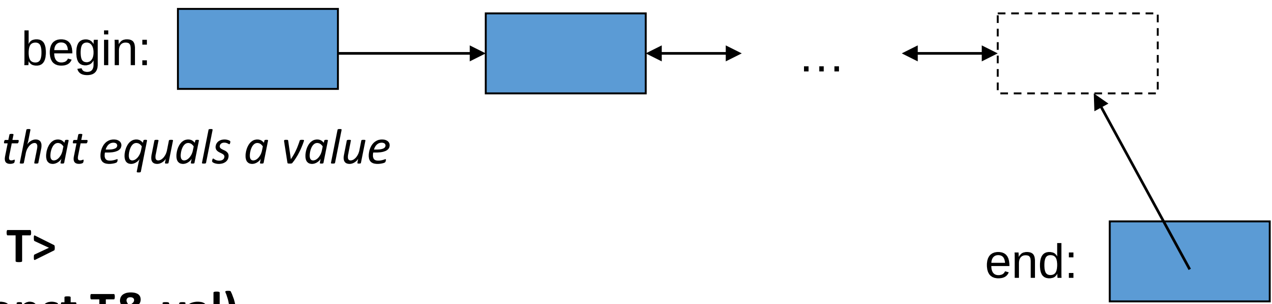


- **set**

(a kind of tree)



The simplest algorithm: **find()**




```
// Find the first element that equals a value

template<class In, class T>
In find(In first, In last, const T& val)
{
  while (first!=last && *first != val) ++first;
  return first;
}

void f(vector<int>& v, int x)// find an int in a vector
{
  vector<int>::iterator p = find(v.begin(),v.end(),x);
  if (p!=v.end()) { /* we found  x */ }
  // …
}
```

We can ignore (“abstract away”) the differences between containers

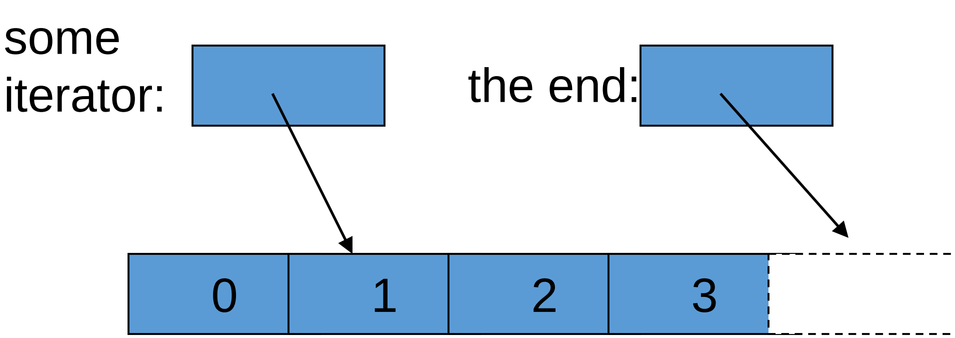
find()

generic for both element type and container type

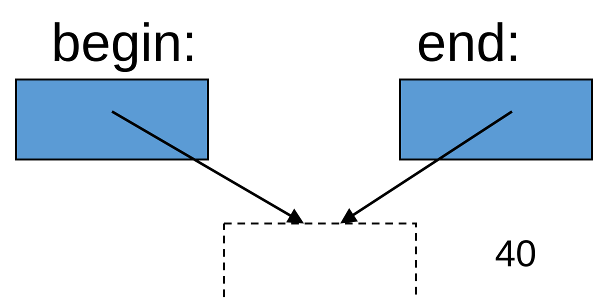
```
void f(vector<int>& v, int x)          // works for vector of ints
{
   vector<int>::iterator p = find(v.begin(),v.end(),x);
   if (p!=v.end()) { /* we found  x */ }
   // …
}

void f(list<string>& v, string x)      // works for list of strings
{
   list<string>::iterator p = find(v.begin(),v.end(),x);
   if (p!=v.end()) { /* we found  x */ }
   // …
}

void f(set<double>& v, double x)            // works for set of doubles
{
   set<double>::iterator p = find(v.begin(),v.end(),x);
   if (p!=v.end()) { /* we found  x */ }
   // …
}
```

Algorithms and iterators

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is "one past the last element"
  - ***not*** "the last element"
  - That's necessary to elegantly represent an empty sequence
  - One-past-the-last-element isn't an element
    - You can compare an iterator pointing to it
    - You can't dereference it (read its value)
- Returning the end of the sequence is the standard idiom for "not found" or "unsuccessful"



An empty sequence:



Simple algorithm: **find_if()**

- Find the first element that matches a criterion (predicate)
  - Here, a predicate takes one argument and returns a **bool**

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
  while (first!=last && !pred(*first)) ++first;
  return first;
}

void f(vector<int>& v)
{
  vector<int>::iterator p = find_if(v.begin(),v.end,Odd());
  if (p!=v.end()) { /* we found an odd number */ }
  // …
}
```

A predicate

Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a **bool**
- For example
  - A function

    bool odd(int i) { return i%2; } // ***%*** *is the remainder (modulo) operator*

    odd(7); **//** *call* ***odd****: is 7 odd?*

  - A function object

    struct Odd {

    bool operator()(int i) const { return i%2; }

    };

    Odd odd; **//** *make an object* ***odd*** *of type* ***Odd***

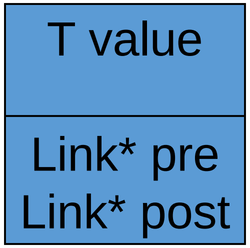
    odd(7); **//** *call* ***odd****: is 7 odd?*

vector

```
template<class T> class vector {
  T* elements;
  // …
  using value_type = T;
  using iterator = ???;        // the type of an iterator is implementation defined
                               // and it (usefully) varies (e.g. range checked iterators)
                               // a vector iterator could be a pointer to an element
  using const_iterator = ???;

  iterator begin();            // points to first element
  const_iterator begin() const;
  iterator end();              // points to one beyond the last element
  const_iterator end() const;

  iterator erase(iterator p);                 // remove element pointed to by p
  iterator insert(iterator p, const T& v);    // insert a new element v before p
};
```

list

Link:




```
template<class T> class list {
   Link* elements;
   // …
   using value_type = T;
   using iterator = ???;          // the type of an iterator is implementation defined
                                  // and it (usefully) varies (e.g. range checked iterators)
                                  // a list iterator could be a pointer to a link node
   using const_iterator = ???;

   iterator begin();              // points to first element
   const_iterator begin() const;
   iterator end();                // points to one beyond the last element
   const_iterator end() const;

   iterator erase(iterator p);                 // remove element pointed to by p
   iterator insert(iterator p, const T& v);    // insert a new element v before p
};
```

Vector vs. List

- By default, use a **vector**
  - You need a reason not to
  - You can “grow” a vector (e.g., using **push_back()**)
  - You can **insert()** and **erase()** in a vector
  - Vector elements are compactly stored and contiguous
  - For small vectors of small elements all operations are fast
    - compared to lists
- If you don’t want elements to move, use a **list**
  - You can “grow” a list (e.g., using **push_back()** and **push_front()**)
  - You can **insert()** and **erase()** in a list
  - List elements are separately allocated
- Note that there are more containers, e.g.,
  - map
  - unordered_map

Some useful standard headers

- **<iostream>** I/O streams, cout, cin, …
- **<fstream>** file streams
- **<algorithm>** sort, copy, …
- **<numeric>** accumulate, inner_product, …
- **<functional>** function objects
- **<string>**
- **<vector>**
- **<map>**
- **<unordered_map>** hash table
- **<list>**
- **<set>**