Algorithms and Data Structures in C++

Complexity analysis

- Answers the question “How does the time needed for an algorithm scale with the problem size $N$?”
  - Worst case analysis: maximum time needed over all possible inputs
  - Best case analysis: minimum time needed
  - Average case analysis: average time needed
  - Amortized analysis: average over a sequence of operations

- Usually only worst-case information is given since average case is much harder to estimate.
Data structures and algorithms in the C++ standard library

Weeks 7&8

The O notation

- Is used for worst case analysis:

  An algorithm is $O(f(N))$ if there are constants $c$ and $N_0$, such that for $N \geq N_0$ the time to perform the algorithm for an input size $N$ is bounded by $t(N) < c f(N)$

- Consequences
  - $O(f(N))$ is identically the same as $O(a f(N))$
  - $O(a N^x + b N^y)$ is identically the same as $O(N^{\max(x,y)})$
  - $O(N^x)$ implies $O(N^y)$ for all $y \geq x$

The Ω and Θ notations

- $\Omega$ is used for best case analysis:

  An algorithm is $\Omega(f(N))$ if there are constants $c$ and $N_0$, such that for $N \geq N_0$ the time to perform the algorithm for an input size $N$ is bounded by $t(N) > c f(N)$

- $\Theta$ is used if worst and best case scale the same

  Data structures and algorithms in the C++ standard library An algorithm is $\Theta(f(N))$ if it is $\Omega(f(N))$ and $O(f(N))$
Data structures and algorithms in the C++ standard library

Weeks 7&8

Programming techniques for scientific simulations

---

Time assuming 1 billion operations per second

<table>
<thead>
<tr>
<th>Complexity</th>
<th>N=10</th>
<th>$10^2$</th>
<th>$10^3$</th>
<th>$10^4$</th>
<th>$10^5$</th>
<th>$10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 ns</td>
<td>1 ns</td>
<td>1 ns</td>
<td>1 ns</td>
<td>1 ns</td>
<td>1 ns</td>
</tr>
<tr>
<td>ln N</td>
<td>3 ns</td>
<td>7 ns</td>
<td>10 ns</td>
<td>13 ns</td>
<td>17 ns</td>
<td>20 ns</td>
</tr>
<tr>
<td>N</td>
<td>10 ns</td>
<td>100 ns</td>
<td>1 µs</td>
<td>10 µs</td>
<td>100 µs</td>
<td>1 ms</td>
</tr>
<tr>
<td>N log N</td>
<td>33 ns</td>
<td>664 ns</td>
<td>10 µs</td>
<td>133 µs</td>
<td>1.7 ms</td>
<td>20 ms</td>
</tr>
<tr>
<td>$N^2$</td>
<td>100 ns</td>
<td>10 µs</td>
<td>1 ms</td>
<td>100 ms</td>
<td>10 s</td>
<td>17 min</td>
</tr>
<tr>
<td>$N^3$</td>
<td>1 µs</td>
<td>1 ms</td>
<td>1 s</td>
<td>17 min</td>
<td>11.5 d</td>
<td>31 a</td>
</tr>
<tr>
<td>$2^N$</td>
<td>1 µs</td>
<td>$10^{14}$ a</td>
<td>$10^{285}$ a</td>
<td>$10^{2996}$ a</td>
<td>$10^{30086}$ a</td>
<td>$10^{301013}$ a</td>
</tr>
</tbody>
</table>

Which algorithm do you prefer?

◆ When do you pick algorithm A, when algorithm B? The complexities are listed below

<table>
<thead>
<tr>
<th>Algorithm A</th>
<th>Algorithm B</th>
<th>Which do you pick?</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(ln N)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>O(ln N)</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>O(ln N)</td>
<td>1000 N</td>
<td></td>
</tr>
<tr>
<td>ln N</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>1000 ln N</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>ln N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>ln N</td>
<td>1000 ln N</td>
<td></td>
</tr>
<tr>
<td>1000 ln N</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
**Complexity: example 1**

What is the $O$, $\Omega$, and $\Theta$ complexity of the following code?

```cpp
double x;
std::cin >> x;
std::cout << std::sqrt(x);
```

**Complexity: example 2**

What is the $O$, $\Omega$, and $\Theta$ complexity of the following code?

```cpp
unsigned int n;
std::cin >> n;
for (int i=0; i<n; ++i)
    std::cout << i*i << "\n";
```
**Complexity: example 3**

What is the $O$, $\Omega$, and $\Theta$ complexity of the following code?

```cpp
unsigned int n;
std::cin >> n;
for (int i=0; i<n; ++i) {
    unsigned int sum=0;
    for (int j=0; j<i; ++j)
        sum += j;
    std::cout << sum << "\n";
}
```

**Complexity: example 4**

What is the $O$, $\Omega$, and $\Theta$ complexity of the following two segments?

- **Part 1:**
  ```cpp
  unsigned int n;
  std::cin >> n;
  double* x=new double[n]; // allocate array of n numbers
  for (int i=0; i<n; ++i)
      std::cin >> x[i];
  ```

- **Part 2:**
  ```cpp
  double y;
  std::cin >> y;
  for (int i=0; i<n; ++i)
      if (x[i]==y) {
          std::cout << i << "\n";
          break;
      }
  ```
Complexity: adding to an array (simple way)

◆ What is the complexity of adding an element to the end of an array?
  ◆ allocate a new array with N+1 entries
  ◆ copy N old entries
  ◆ delete old array
  ◆ write (N+1)-st element

◆ The complexity is O(N)

Complexity: adding to an array (clever way)

◆ What is the complexity of adding an element to the end of an array?
  ◆ allocate a new array with 2N entries, but mark only N+1 as used
  ◆ copy N old entries
  ◆ delete old array
  ◆ write (N+1)-st element

◆ The complexity is O(N), but let’s look at the next elements added:
  ◆ mark one more element as used
  ◆ write additional element

◆ The complexity here is O(1)
◆ The amortized (averaged) complexity for N elements added is

$$\frac{1}{N} (O(N) + (N - 1)O(1)) = O(1)$$
Data structures and algorithms in the C++ standard library

STL: Standard Template Library

- Most notable example of generic programming
- Widely used in practice
- Theory: Stepanov, Musser; Implementation: Stepanov, Lee

Steady Template Library

- Proposed to the ANSI/ISO C++ Standards Committee in 1994.
- After small revisions, part of the official C++ standard in 1997.

The standard C++ library

- function objects: negate, plus, multiplies, ...
  - your function ...
- predicates: less, greater, equal_to, ...
  - your predicate
- sequence algorithms: accumulate, inner_product, find, reverse, ...
  - your algorithm
- allocators: allocator
  - your allocator
- data sequences: list, vector, deque, map, set, ...
  - your container
- container adapters: stack, queue, priority_queue
  - your data structure

Programming techniques for scientific simulations
The **string** and **wstring** classes

- are very useful class to manipulate strings
  - `string` for standard ASCII strings (e.g. “English”)
  - `wstring` for wide character strings (e.g. “日本語”)
- Contains many useful functions for string manipulation
  - Adding strings
  - Counting and searching of characters
  - Finding substrings
  - Erasing substrings
  - ...
- Since this is not very important for numerical simulations I will not go into details. Please read your C++ book

The **pair** template

- `template <class T1, class T2> class pair {
  public:
    T1 first;
    T2 second;
    pair(const T1& f, const T2& s) : first(f), second(s) {}
};`
- will be useful in a number of places
# Data structures in C++

We will discuss a number of data structures and their implementation in C++:

- **Arrays:**
  - C array
  - vector
  - valarray
  - deque

- **Linked lists:**
  - list

- **Trees:**
  - map
  - set
  - multimap
  - multiset

- **Queues and stacks:**
  - queue
  - priority_queue
  - stack

---

## The array or vector data structure

- An array/vector is a consecutive range in memory

    ![Array representation]

- Advantages
  - Fast O(1) access to arbitrary elements: `a[i]` is `*(a+i)`
  - Profits from cache effects
  - Insertion or removal at the end is O(1)
  - Searching in a sorted array is O(ln N)

- Disadvantage
  - Insertion and removal at arbitrary positions is O(N)
Slow \(O(N)\) insertion and removal in an array

- **Inserting an element**
  - Need to copy \(O(N)\) elements
    
    \[
    \begin{array}{cccccccc}
    a & b & c & d & e & f & g & h \\
    \end{array}
    \]
    
    \[
    \begin{array}{cccccccc}
    a & b & c & d & x & e & f & g & h \\
    \end{array}
    \]

- **Removing an element**
  - Also need to copy \(O(N)\) elements
    
    \[
    \begin{array}{cccccccc}
    a & b & c & x & e & f & g & h \\
    \end{array}
    \]
    
    \[
    \begin{array}{cccccccc}
    a & b & c & e & f & g & h \\
    \end{array}
    \]

Fast \(O(1)\) removal and insertion at the end of an array

- **Removing the last element**
  - Just change the size
    
    \[
    \begin{array}{cccccccc}
    a & b & c & d & e & f & e & g & h \\
    \end{array}
    \]
    
   Capacity 8, size 6:
    
    Capacity 8, size 5:

- **Inserting elements at the end**
  - Is amortized \(O(1)\)
    
    \[
    \begin{array}{cccc}
    a & b & c & d \\
    \end{array}
    \]
    
    first double the size and copy in \(O(N)\):
    
    \[
    \begin{array}{cccccccc}
    a & b & c & d & e & f & e & g & h \\
    \end{array}
    \]
    
    **then just change the size:**
    
    \[
    \begin{array}{cccccccc}
    a & b & c & d & e & f & g \\
    \end{array}
    \]
The deque data structure (double ended queue)

- Is a variant of an array, more complicated to implement
  - See a data structures book for details
- In addition to the array operations also the insertion and removal at beginning is $O(1)$
- Is needed to implement queues

The stack data structure

- Is like a pile of books
  - LIFO (last in first out): the last one in is the first one out
- Allows in $O(1)$
  - Pushing an element to the top of the stack
  - Accessing the top-most element
  - Removing the top-most element
The queue data structure

- Is like a queue in the Mensa
  - FIFO (first in first out): the first one in is the first one out

- Allows in $O(1)$
  - Pushing an element to the end of the queue
  - Accessing the first and last element
  - Removing the first element

The priority queue data structure

- Is like a queue in the Mensa, but professors are allowed to go to the head of the queue (not passing other professors though)
  - The element with highest priority (as given by the $<$ relation) is the first one out
  - If there are elements with equal priority, the first one in the queue is the first one out

- There are a number of possible implementations, look at a data structure book for details
The linked list data structure

- An linked list is a collection of objects linked by pointers into a one-dimensional sequence

![Linked List Diagram]

- **Advantages**
  - Fast \( O(1) \) insertion and removal anywhere
    - Just reconnect the pointers

- **Disadvantage**
  - Does not profit from cache effects
  - Access to an arbitrary element is \( O(N) \)
  - Searching in a list is \( O(N) \)

The tree data structures

- An array needs
  - \( O(N) \) operations for arbitrary insertions and removals
  - \( O(1) \) operations for random access
  - \( O(N) \) operations for searches
  - \( O(\log N) \) operations for searches in a sorted array

- A list needs
  - \( O(1) \) operations for arbitrary insertions and removals
  - \( O(N) \) operations for random access and searches

- What if both need to be fast? Use a tree data structure:
  - \( O(\log N) \) operations for arbitrary insertions and removals
  - \( O(\log N) \) operations for random access and searches
A node in a binary tree

- Each node is always linked to two child nodes
  - The left child is always smaller
  - The right child node is always larger

A binary tree

- Can store $N=2^{n-1}$ nodes in a tree of height $n$
  - Any access needs at most $n = O(\ln N)$ steps

- Example: a tree of height 5 with 12 nodes
Unbalanced trees

- Trees can become unbalanced
  - Height is no longer $O(\ln N)$ but $O(N)$
  - All operations become $O(N)$

- Solutions
  - Rebalance the tree
  - Use self-balancing trees

- Look into a data structures book to learn more

Tree data structures in the C++ standard

- Fortunately the C++ standard contains a number of self-balancing tree data structures suitable for most purposes:
  - `set`
  - `multiset`
  - `map`
  - `multimap`

- But be aware that computer scientists know a large number of other types of trees and data structures
  - Read the books
  - Ask the experts
The container concept in the C++ standard

- Containers are sequences of data, in any of the data structures
  - `vector<T>` is an array of elements of type T
  - `list<T>` is a doubly linked list of elements of type T
  - `set<T>` is a tree of elements of type T
  ...

- The standard assumes the following requirements for the element `T` of a container:
  - `default constructor T()`
  - `assignment T& operator=(const T&)`
  - `copy constructor T(const T&)`
  - Note once again that assignment and copy have to produce identical copy: in the Penna model the copy constructor should not mutate!

Connecting Algorithms to Sequences

```cpp
struct node
{
    char value;
    node* next;
};

node* find( node* const s, char x )
{
    node* pos = s;
    while (pos != 0)
    {
        if ( pos->value == x )
            return pos;
        pos = pos->next;
    }
    return pos;
}
```

```cpp
int find( char const(&s)[4], char x )
{
    int pos = 0;
    while (pos != sizeof(s))
    {
        if ( s[pos] == x )
            return pos;
        ++pos;
    }
    return pos;
}
```
Data structures and algorithms in the C++ standard library

Programming techniques for scientific simulations

17
NxM Algorithm Implementations?

1. find
2. copy
3. merge
4. transform
N. accumulate

1. vector
2. list
3. deque
4. set
5. map
6. char[5]
M. foobar

F. T. S. E.

Fundamental Theorem of Software Engineering

"We can solve any problem by introducing an extra level of indirection"

--Butler Lampson

Programming techniques for scientific simulations
Iterators to the Rescue

◆ Define a common interface for
  ◆ traversal
  ◆ access
  ◆ positional comparison

◆ Containers provide iterators
◆ Algorithms operate on pairs of iterators

```cpp
template <class Iter, class T>
Iter find( Iter start, Iter finish, T x )
{
    Iter pos = start;
    for (; pos != finish; ++pos)
    {
        if ( *pos == x )
            return pos;
    }
    return pos;
}
```

```cpp
struct node_iterator
{
    // ...
    char& operator*( ) const
        { return n->value; }  
    node_iterator& operator++()
        { n = n->next; return *this; }
    private:
        node* n;
};
```

Describe Concepts for std::find

```cpp
template <class Iter, class T>
Iter find(Iter start, Iter finish, T x)
{
    Iter pos = start;
    for (; pos != finish; ++pos)
    {
        if ( *pos == x )
            return pos;
    }
    return pos;
}
```

◆ Concept Name?
◆ Valid expressions?
◆ Preconditions?
◆ Postconditions?
◆ Complexity guarantees?
◆ Associated types?
Traversing an array and a linked list

Two ways for traversing an array

1. Using an index:
   ```cpp
   T* a = new T[size];
   for (int n=0; n<size; ++n)
     cout << a[n];
   
   2. Using pointers:
   ```cpp
   for (T* p = a;
        p != a+size;
        ++p)
     cout << *p;
   ```

Traversing a linked list

```cpp
template <class T> struct node
{
  T value; // the element
  node<T>* next; // the next Node
};

template<class T> struct list
{
  node<T>* first;
};

list<T> l;
...
```

```cpp
for (mode<T>* p = l.first;
        p != 0;
        p = p->next)
  cout << p->value;
```

Generic traversal

Can we traverse a vector and a list in the same way?

1. Instead of
   ```cpp
   for (T* p = a;
        p != a+size;
        ++p)
     cout << *p;
   ```
   We want to write
   ```cpp
   for (iterator p = a.begin();
        p != a.end();
        ++p)
     cout << *p;
   ```

1. Instead of
   ```cpp
   for (node<T>* p = l.first;
        p != 0;
        p = p->next)
     cout << p->value;
   ```
   We want to write
   ```cpp
   for (iterator p = l.begin();
        p != l.end();
        ++p)
     cout << *p;
   ```
Implementing iterators for the array

```cpp
template<class T>
class Array {
public:
    typedef T* iterator;
    typedef unsigned size_type;
    Array();
    Array(size_type);

    iterator begin() { return p_; }
    iterator end() { return p_+sz_; }

private:
    T* p_;
    size_type sz_;}
```

Now allows the desired syntax:
```cpp```
```cpp
for (Array<T>::iterator p = a.begin(); p !a.end(); ++p)
    cout << *p;
```

Instead of
```cpp```
```cpp
for (T* p = a.p_; p !a.p_+a.sz_; ++p)
    cout << *p;
```

Implementing iterators for the linked list

```cpp
template <class T>
struct node_iterator {
    Node<T>* p;
    node_iterator(Node<T>* q) : p(q) {}

    node_iterator<T>& operator++(){
        p=p->next;
    }

    T* operator->() { return &(p->value); }
    T& operator*() { return p->value; }

    bool operator!=(const node_iterator<T>& x) {
        return p!=x.p;
    }

    // more operators missing ... }
```

```cpp
template<class T>
class list {
    Node<T>* first;

    typedef node_iterator<T> iterator;

    iterator begin() { return iterator(first); }
    iterator end() { return iterator(0); }
```

Now also allows the desired syntax:
```cpp```
```cpp
for (List<T>::iterator p = l.begin(); p !l.end(); ++p)
    cout << *p;
```
Iterators

- have the same functionality as pointers
- including pointer arithmetic!
  - `iterator a, b; cout << b-a; // # of elements in [a,b]`
- exist in several versions
  - forward iterators ... move forward through sequence
  - backward iterators ... move backwards through sequence
  - bidirectional iterators ... can move any direction
  - input iterators ... can be read: `x=*p;`
  - output iterators ... can be written: `*p=x;`
- and all these in const versions (except output iterators)

Container requirements

- There are a number of requirements on a container that we will now discuss based on the handouts
Containers and sequences

- A container is a collection of elements in a data structure
- A sequence is a container with a linear ordering (not a tree)
  - vector
  - deque
  - list
- An associative container is based on a tree, finds element by a key
  - map
  - multimap
  - set
  - multiset
- The properties are defined on the handouts from the standard
  - A few special points mentioned on the slides

Sequence constructors

- A sequence is a linear container (vector, deque, list,...)
- Constructors
  - `container()` ... empty container
  - `container(n)` ... n elements with default value
  - `container(n,x)` ... n elements with value x
  - `container(c)` ... copy of container c
  - `container(first,last)` ... first and last are iterators
    - container with elements from the range [first,last]
- Example:
  - `std::list<double> l;
    // fill the list
    ...
    // copy list to a vector
    std::vector<double> v(l.begin(),l.end());`
Direct element access in deque and vector

- Optional element access (not implemented for all containers)
  - T& container[k] ... k-th element, no range check
  - T& container.at(k) ... k-th element, with range check
  - T& container.front() ... first element
  - T& container.back() ... last element

Inserting and removing at the beginning and end

- For all sequences: inserting/removing at end
  - container.push_back(T x) // add another element at end
  - container.pop_back() // remove last element

- For list and deque (stack, queue)
  - container.push_first(T x) // insert element at start
  - container.pop_first() // remove first element
Inserting and erasing anywhere in a sequence

- List operations (slow for vectors, deque etc.):
  - `insert (p, x)` // insert x before p
  - `insert(p, n, x)` // insert n copies of x before p
  - `insert(p, first, last)` // insert [first, last] before p
  - `erase(p)` // erase element at p
  - `erase(first, last)` // erase range [first, last]
  - `clear()` // erase all

Vector specific operations

- Changing the size
  - `void resize(size_type)`
  - `void reserve(size_type)`
  - `size_type capacity()`

- Note:
  - `reserve` and `capacity` regard memory allocated for vector!
  - `resize` and `size` regard memory currently used for vector data

- Assignments
  - `container = c` ... copy of container c
  - `container.assign(n)` ... assign n elements the default value
  - `container.assign(n, x)` ... assign n elements the value x
  - `container.assign(first, last)` ... assign values from the range [first, last]

- Watch out: assignment does not allocate, do a resize before!
The **valarray** template

- acts like a vector but with additional (mis)features:
  - No iterators
  - No reserve
  - Resize is fast but erases contents

- for numeric operations are defined:

```cpp
std::valarray<double> x(100), y(100), z(100);
x = y + \exp(z);
```

- Be careful: it is not the fastest library!
- We will learn about faster libraries later

---

Sequence adapters: **queue** and **stack**

- are based on deques, but can also use vectors and lists
  - **stack** is first in-last out
  - **queue** is first in-first out
  - **priority_queue** prioritizes with < operator

- stack functions
  - `void push(const T& x)` ... insert at top
  - `void pop()` ... removes top
  - `T& top()`
  - `const T& top() const`

- queue functions
  - `void push(const T& x)` ... inserts at end
  - `void pop()` ... removes front
  - `T& front(), T& back()`,
    `const T& front(), const T& back()`
### list-specific functions

- The following functions exist only for `std::list`:
  - `splice`  
    - Joins lists without copying, moves elements from one to end of the other
  - `sort`  
    - Optimized sort, just relinks the list without copying elements
  - `merge`  
    - Preserves order when “splicing” sorted lists
  - `remove(T x)`
  - `remove_if(criterion)`  
    - Criterion is a function object or function, returning a `bool` and taking a `const T&` as argument, see Penna model
      - Example:
        ```cpp
        bool is_negative(const T& x) { return x<0; }
        list.remove_if(is_negative);
        ```

### The map class

- Implements associative arrays
  - `map<std::string, long> phone_book;`  
    - `phone_book[“Troyer”] = 32589;`  
    - `phone_book[“Heeb”] = 32591;`  
    - `if(phone_book[name])`  
      ```cpp
      cout << "The phone number of " << name << " is " << phone_book[name];
      ```  
    - `else`  
      ```cpp
      cout << name << "’s phone number is unknown!’;
      ```
  - Is implemented as a tree of `pairs`
  - Take care:
    - `map<T1,T2>::value_type is pair<T1,T2>`
    - `map<T1,T2>::key_type is T1`
    - `map<T1,T2>::mapped_type is T2`
    - `insert, remove, ... are sometimes at first sight confusing for a map!`
Other tree-like containers

- **multimap**
  - can contain more than one entry (e.g., phone number) per key

- **set**
  - unordered container, each entry occurs only once

- **multiset**
  - unordered container, multiple entries possible

- extensions are no problem
  - if a data structure is missing, just write your own
  - good exercise for understanding of containers

Search operations in trees

- In a map<K,V>, K is the key type and V the mapped type
  - Attention: iterators point to pairs

- In a map<T>, T is the key type and also the value_type

- Fast $O(\log N)$ searches are possible in trees:
  - `a.find(k)` returns an iterator pointing to an element with key k or `end()` if it is not found.
  - `a.count(k)` returns the number of elements with key k.
  - `a.lower_bound(k)` returns an iterator pointing to the first element with $key >= k$.
  - `a.upper_bound(k)` returns an iterator pointing to the first element with $key > k$.
  - `a.equal_range(k)` is equivalent to but faster than `std::make_pair(a.lower_bound(k), a.upper_bound(k))`
Search example in a tree

Look for all my phone numbers:

- // some typedefs
  
  typedef multimap<std::string, int> phonebook_t;
  typedef phonebook_t::const_iterator IT;
  typedef phonebook_t::value_type value_type;

  // the phonebook
  phonebook_t phonebook;

  // fill the phonebook
  phonebook.insert(value_type("Troyer",32589));

  // search all my phone numbers
  pair< IT, IT> range = phonebook.equal_range("Troyer");

  // print all my phone numbers
  for (IT it=range.first; it != range.second;++it)
    cout << it->second << "\n";

Almost Containers

- C-style array
- string
- valarray
- bitset

- They all provide almost all the functionality of a container
- They can be used like a container in many instances, but not all

  - int x[5] = {3,7,2,9,4};
    vector<int> v(x,x+5);
  - uses vector(first,last), pointers are also iterators!
The generic algorithms

- Implement a big number of useful algorithms
- Can be used on any container
  - rely only on existence of iterators
  - “container-free algorithms”
  - now all the fuss about containers pays off!
- Very useful
- Are an excellent example in generic programming
- We will use them now for the Penna model
  That’s why we did not ask you to code the Population class for the Penna model yet!

Example: find

- A generic function to find an element in a container:
  - list<string> fruits;
  - list<string>::const_iterator found = find(fruits.begin(),fruits.end(),"apple");
  - if (found==fruits.end()) // end means invalid iterator
    cout << “No apple in the list”;
  - else
    cout << “Found it: ” << *found << “\n”;

- find declared and implemented as
  - template <class In, class T>
    In find(In first, In last, T v) {
      while (first != last && *first != v)
        ++first;
    return first;
  }
**Example: find_if**

- Takes predicate (function object or function)
  - `bool favorite_fruits(const std::string& name)`
    ```cpp
    { return (name==“apple” || name == “orange”);}
    ```

- Can be used with `find_if` function:
  - `list<string>::const_iterator found = find_if(fruits.begin(),fruits.end(),favorite_fruits);`
  - `if (found==fruits.end())`
  - `cout << “No favorite fruits in the list”;`
  - `else`
  - `cout << “Found it: “ << *found << “\n”;`

- `find_if` declared and implemented as
  ```cpp
  template <class In, class Pred>
  In find_if(In first, In last, Pred p) {
    while (first != last && !p(*first) )
      ++first;
    return first;
  }
  ```

**Member functions as predicates**

- We want to find the first pregnant animal:
  ```cpp
  list<Animal> pop;
  find_if(pop.begin(),pop.end(),is_pregnant)
  ```

- This does not work as expected, it expects
  ```cpp
  bool is_pregnant(const Animal&);
  ```

- We want to use
  ```cpp
  bool Animal::pregnant() const
  ```

- Solution: `mem_fun_ref` function adapter
  ```cpp
  find_if(pop.begin(),pop.end(),
          mem_fun_ref(&Animal::pregnant));
  ```

- Many other useful adapters available
  ```cpp
  Once again: please read the books before coding your own!
**push_back and back_inserter**

- **Attention:**
  - `vector<int> v,w;`  
  ```cpp
def v[k]=k; //error: v is size 0!  
  w.push_back(k); // OK: grows the array and assigns  
  ```

- **Same problem with copy:**
  - `vector<int> v(100), w(0);`  
  ```cpp
copy(v.begin(),v.end(),w.begin()); // problem: w of size 0!  
  ```

- **Solution 1:** vectors only  
  ```cpp
  w.resize(v.size()); copy(v.begin(),v.end(),w.begin());  
  ```

- **Solution 2:** elegant  
  ```cpp
  copy(v.begin(),v.end(),back_inserter(w)); // uses push_back  
  ```

- **also push_front and front_inserter for some containers**

**Penna Population**

- **easiest modeled as**  
  ```cpp
class Population : public list<Animal> {...}  
  ```

- **Removing dead:**
  ```cpp
  remove_if(mem_fun_ref(&Animal::is_dead));  
  ```

- **Removing dead, and others with probability N/N0:**  
  ```cpp
  remove_if(animal_dies(N/N0));  
  ```

- **Inserting children:**
  ```cpp
  cannot go into same container, as that might invalidate iterators:  
  vector<Animal> children;  
  for(const_iterator a=begin();a!=end();++a)  
    if(a->pregnant())  
      children.push_back(a->child());  
  copy(children.begin(),children.end(),  
    back_inserter(*this));  
  ```
The binary search

Searching using binary search in a sorted vector is $O(\ln N)$.

Binary search is recursive search in range [begin, end]
- If range is empty, return
- Otherwise test $middle = begin + (end - begin) / 2$
  - If the element in the middle is the search value, we are done
  - If it is larger, search in [begin, middle]
  - If it is smaller, search in [middle, end]

The search range is halved in every step and we thus need at most $O(\ln N)$ steps.

Example: lower_bound

template<class IT, class T>
IT lower_bound(IT first, IT last, const T& val) {
typedef typename iterator_traits<IT>::difference_type dist_t;

dist_t len = distance(first, last); // generic function for last-first

dist_t half;

IT middle;

while (len > 0) {
    half = len >> 1; // faster version of half=len/2

    middle = first;
    advance(middle, half); // generic function for middle+=half
    if (*middle < val) {
        first = middle;
        ++first;
        len = len - half - 1;
    }
    else
        len = half;
}

return first;
}
**Algorithms overview**

- **Nonmodifying**
  - `for_each`
  - `find, find_if, find_first_of`
  - `adjacent_find`
  - `count, count_if`
  - `mismatch`
  - `equal`
  - `search`
  - `find_end`
  - `search_n`

- **Modifying**
  - `transform`
  - `copy, copy_backward`
  - `swap, iter_swap, swap_ranges`
  - `replace, replace_if, replace_copy, replace_copy_if`
  - `fill, fill_n`
  - `generate, generate_n`
  - `remove, remove_if, remove_copy, remove_copy_if`
  - `unique, unique_copy`
  - `reverse, reverse_copy`
  - `rotate, rotate_copy`
  - `random_shuffle`

**Algorithms overview (continued)**

- **Sorted Sequences**
  - `sort, stable_sort`
  - `partial_sort, partial_sort_copy`
  - `nth_element`
  - `lower_bound, upper_bound`
  - `equal_range`
  - `binary_search`
  - `merge, inplace_merge`
  - `partition, stable_partition`

- **Permutations**
  - `next_permutation`
  - `prev_permutation`

- **Set Algorithms**
  - `includes`
  - `set_union`
  - `set_intersection`
  - `set_difference`
  - `set_symmetric_difference`

- **Minimum and Maximum**
  - `min`
  - `max`
  - `min_element`
  - `max_element`
  - `lexicographical_compare`
Exercise

◆ Code the population class for the Penna model based on a standard container
◆ Use function objects to determine death

◆ In the example we used a loop.
  ◆ Can you code the population class without using any loop?
  ◆ This would increase the reliability as the structure is simpler!

◆ Also add fishing in two variants:
  ◆ fish some percentage of the whole population
  ◆ fish some percentage of adults only

◆ Read Penna’s papers and simulate the Atlantic cod!

stream iterators and Shakespeare

◆ Iterators can also be used for streams and files
  ◆ istream_iterator
  ◆ ostream_iterator

◆ Now you should be able to understand Shakespeare:

```cpp
int main()
{
    vector<string> data;
    copy(istream_iterator<string>(cin),istream_iterator<string>(),
         back_inserter(data));
    sort(data.begin(), data.end());
    unique_copy(data.begin(),data.end(),ostream_iterator<string>(cout,"\n"));
}
```
Summary

◆ Please read the sections on
  ◆ containers
  ◆ iterators
  ◆ algorithms
◆ in Stroustrup or Lippman (3rd editions only!)

◆ Examples of excellent class and function designs
◆ Before writing your own functions and classes:
  Check the standard C++ library!
◆ When writing your own functions/classes:
  Try to emulate the design of the standard library
◆ Don't forget to include the required headers:
  ◆ <algorithm>, <functional>, <map>, <iterators>, ... as needed