The C++ Standard Library (2)

Lecture 4

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

Version Control

What is Version Control?

- A database that keeps track of all changes to files over time
- Allows for collaborative development
- Allows to track who made what changes and when
- Allows to revert changes and go back to previous state

What is Git

- Distributed version control system
- Entire code and history is kept on user's machine
  - Changes can be made without internet access
  - (Except pushing and pulling from a remote server)
- One of many different version control systems
  - Subversion, Perforce, Mercurial
  - Git one of the most widely used ones

How does Git work?

- Can be complicated at first, but it is based on a few key concepts
- Based on Snapshots
  - All history is based on snapshots
  - Records what all files look like at a given point in time
  - User decides when to take snapshots and of what files
  - Can go back to any previous snapshot
    - Later snapshots are still available

How does Git work?

- Key concept: Commit
- The act of taking a snapshot
  - Verb: the user committed the code
  - Noun: the user made a new commit
- Every project is made of many commits
  - List of commits defines the timeline of changes applied to files
- Three pieces of information:
  - How did files changes from previous state
  - A reference of the commit that came before it
    - Parent commit
  - A unique hash code identifying the commit
    - Something like: `c374f26626038f020dd12f842d4dc5d67d02f59d`

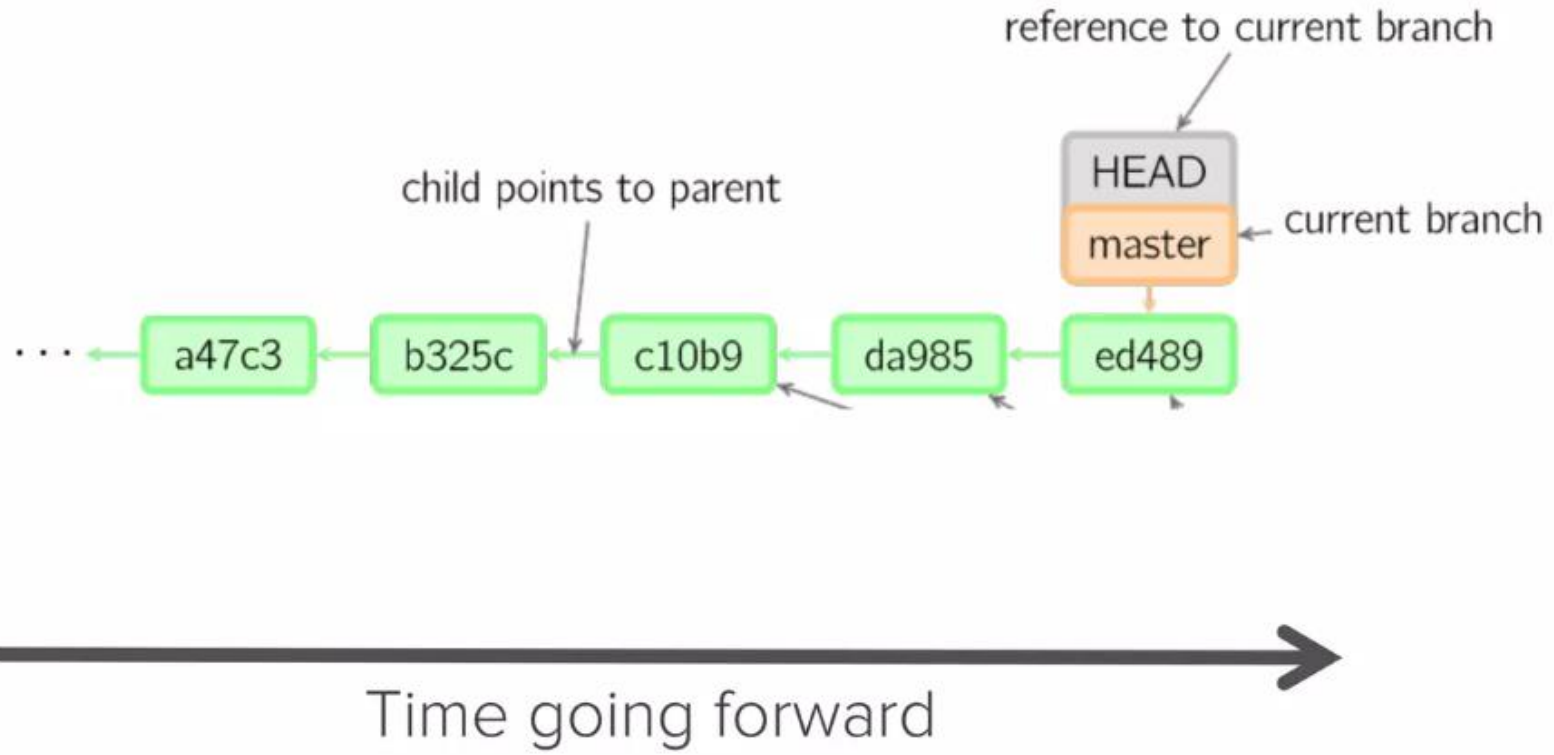
How does Git work?

- Key concept: Repositories
  - Often shortened to repo
- A collection of all files and their history for a project
  - Consists of all commits
  - Place were all the work is stored
- Can live on a local machine or on a remote server (Github)
- Copying a remote repository to your local machine is called cloning
  - Allows for teams to work collaboratively
- Downloading commits from remote repository: pulling changes
- Adding local changes to a remote repository: pushing changes

How does Git work?

- Key concept: Branches
- All commits live on a branch
  - Each branch is a sequence of commits
- There can be many branches
- The main branch is often called main or master branch

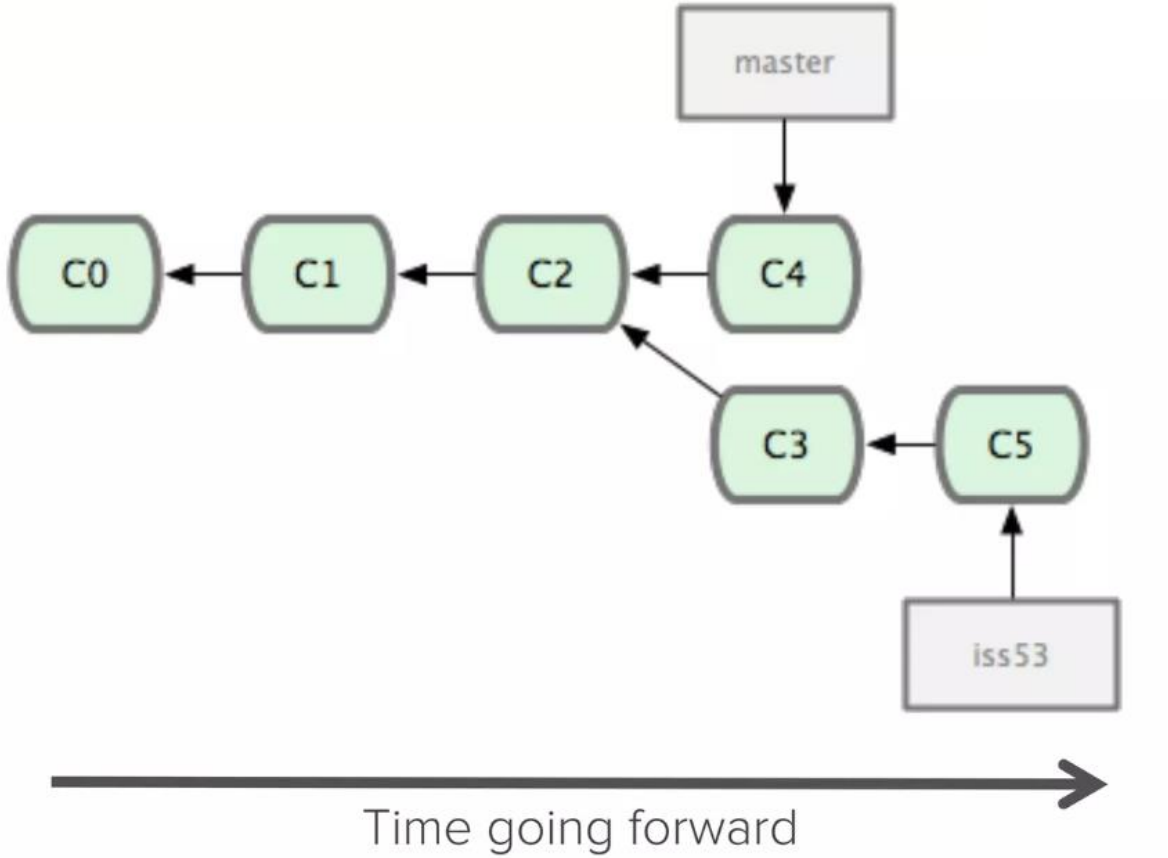
Typical Structure of a Project



Typical Structure of a Project

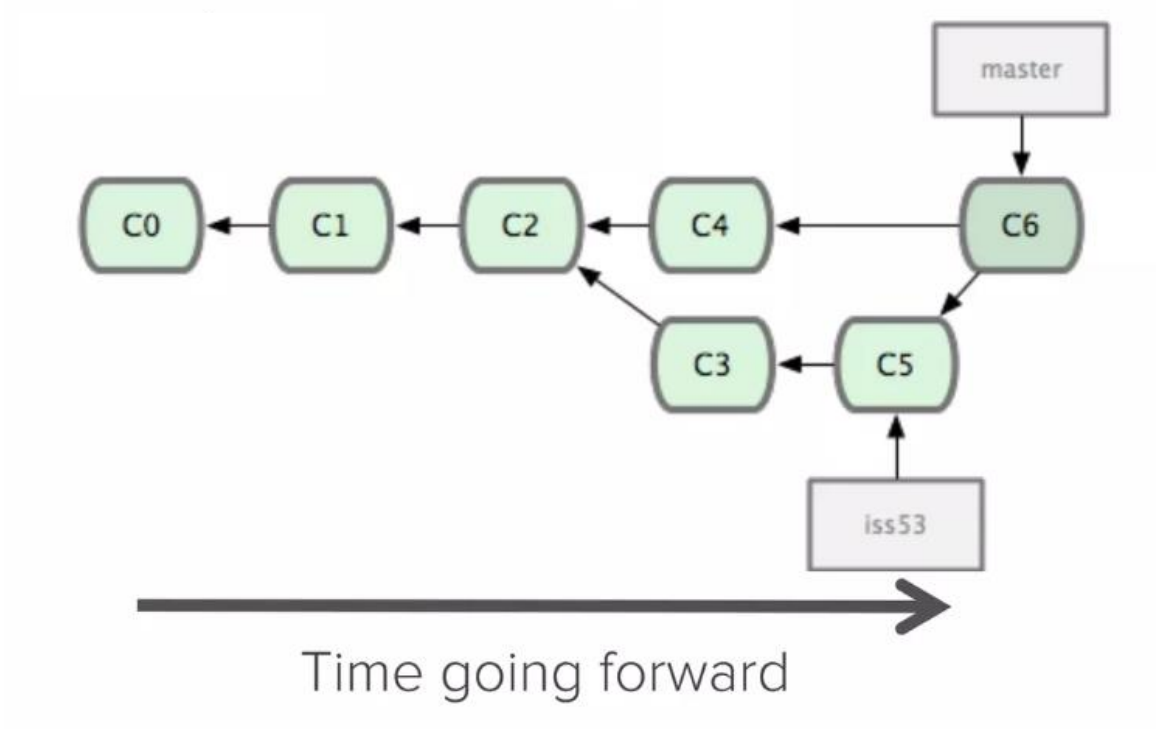
- HEAD: Reference to the most recent commit
- MASTER: The main branch in a project
  - Sometimes called ‘main’
- Key concept: branching off master branch
  - Start off a branch from a specific commit
  - Any changes to a project should start with a new branch

Branching off master



Typical Structure of a Project

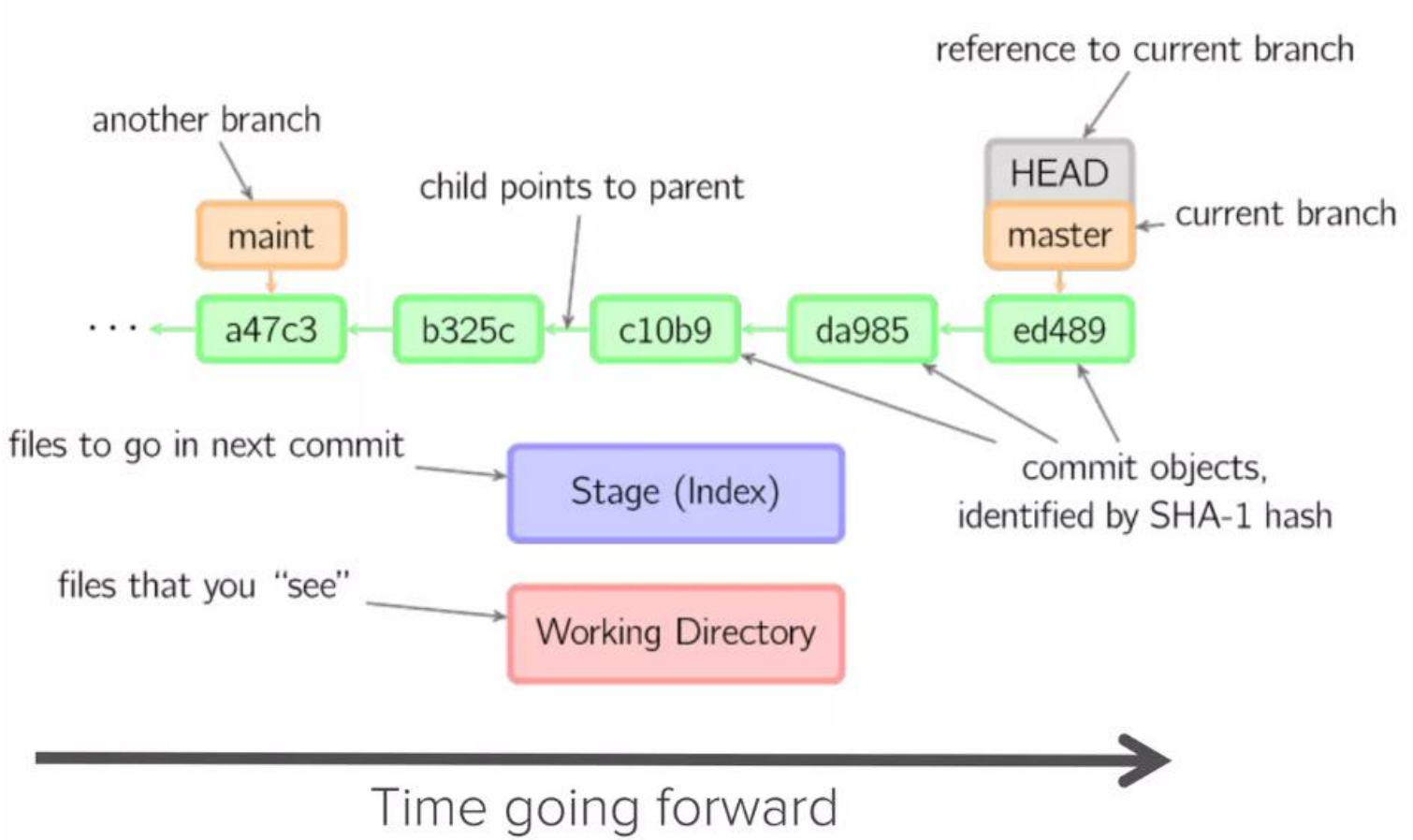
- Key concept: Merging
- Once done with a feature the branch will be merged back to master



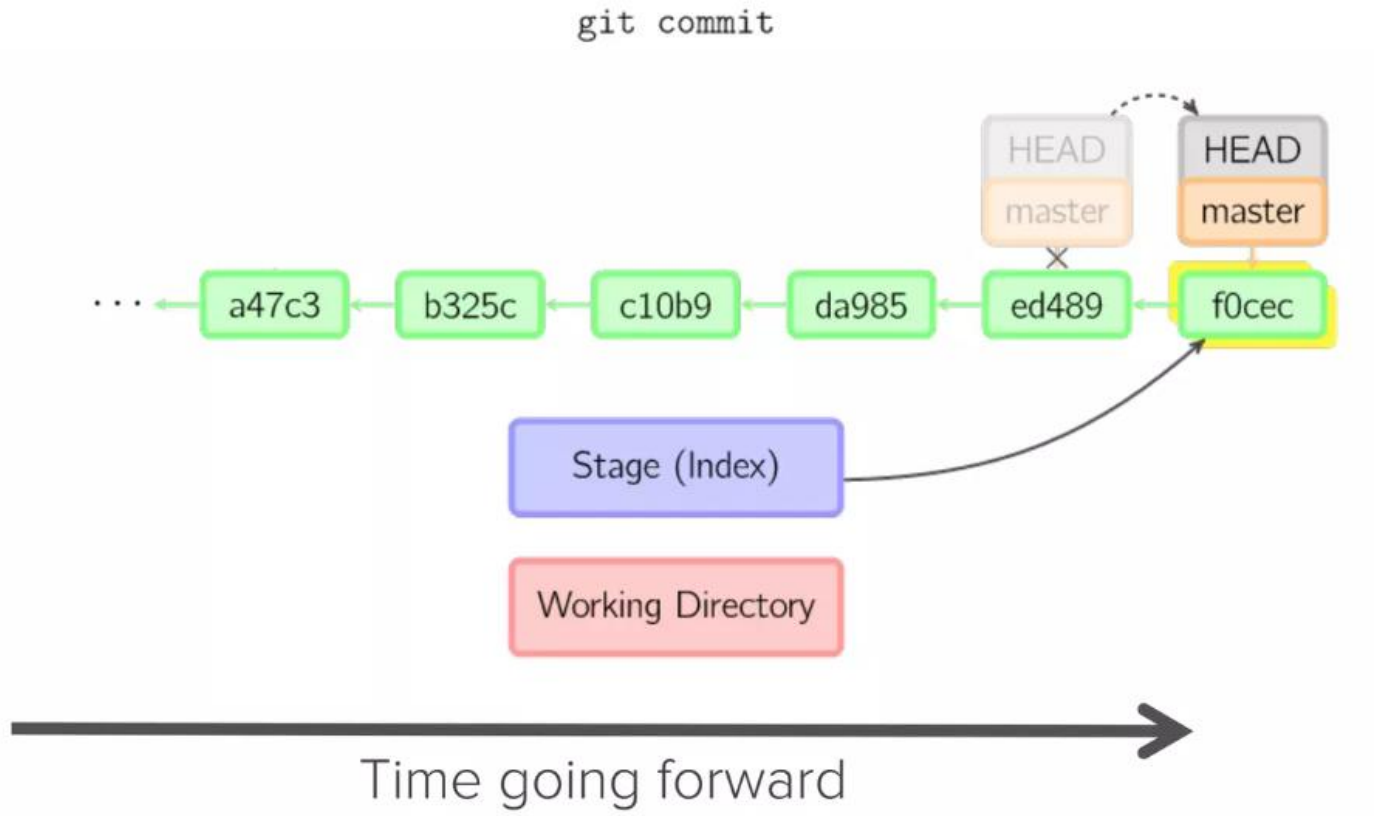
Making a Commit

- Files can be a in a lot of states and places
- Files are being edited in your local file system
  - The working directory
- A file that is ready to be committed needs to be staged (added to the index)
  - Use `'git add ...'` command to define the set of files that should be part of a commit
  - Use `'git commit ...'` command to create actual commit

Making a Commit



Making a Commit



The Standard Template Library

Abstract

- We will look at the C++ Standard Template Library
  - A vast collection of extremely useful containers, algorithms, and supporting data structures
- This will be a whirlwind overview over certain aspects and facilities
  - Containers, Algorithms & Iterators

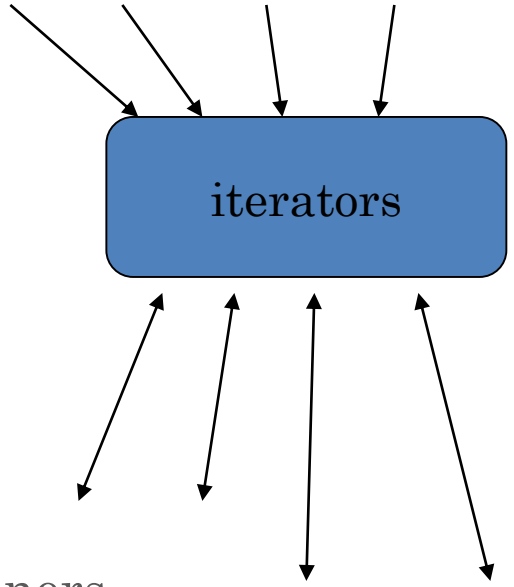
The STL (Standard Template Library)

- 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
    - Iterators know about internal container structure and data
    - Iterators expose uniform interface for algorithms

The STL

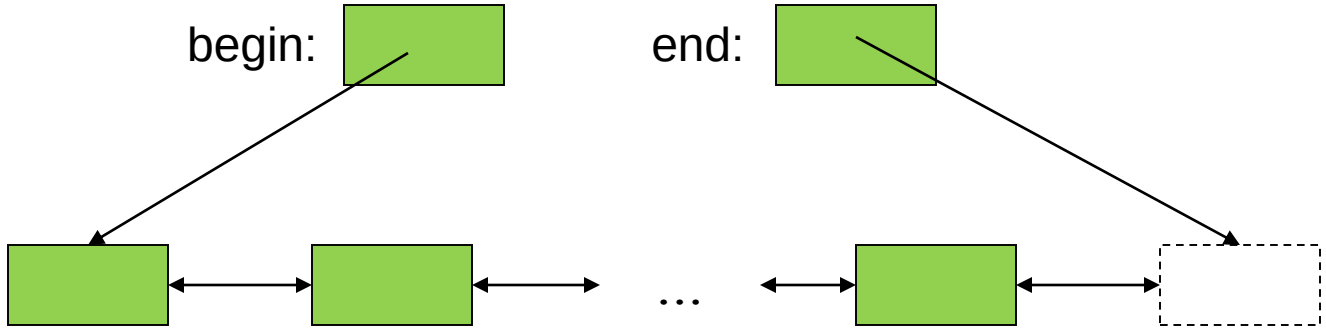
- An ISO C++ standard framework of about a dozen containers and over 100 algorithms connected by iterators
  - Other organizations provide more containers and algorithms in the style of the STL
    - Boost.org, Microsoft, SGI, …
- Probably the currently best known and most widely used example of generic programming

The STL

- If you know the basic concepts and a few examples you can use the rest
- Documentation
  - Cpp-Reference: https://en.cppreference.com
  - Draft C++ Standard: https://eel.is/c++draft/

Basic Model

- A pair of iterators defines a sequence (a range)
  - 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 of element
  - == Does this iterator point to the same element as that iterator?
- Some iterators support more operations (e.g. --, +, and [ ])

Pattern: Iterator

- **Context:**
  - 1. An object (which we'll call the *container*) contains other objects (which we'll call *elements*).
  - 2. Algorithms (that is, methods that use the container) need access to the elements.
  - 3. The container should not expose its internal structure.
  - 4. There may be multiple algorithms that need simultaneous access.
- **Solution:**
  - 1. Define an iterator class that refers to one element at a time.
  - 2. Each iterator type needs to be able to keep track of the position of the current element
  - 3. Each iterator knows how to navigate the container to find next/previous element
  - 4. There are several variations of containers
    - Each exposes its own iterator classes
    - All iterators implement common interfaces
    - The algorithm only needs to know the interface, not the concrete classes.

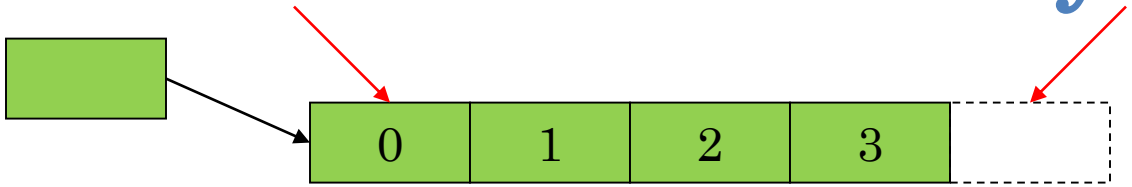
Containers

Containers

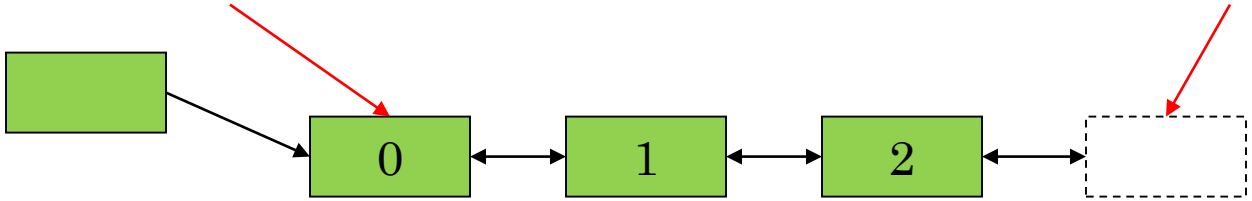
- Containers are special data structures that help storing one or more items of the same type
  - Collections of items
- Manages the storage space for its elements and provides functions to access them
- Containers replicate structures very commonly used in programming:
  - Arrays: dynamic (`std::vector`), static (`std::array`)
  - Linked lists (`std::list`, `std::forward_list`)
  - Trees (`std::set`, `std::multi_set`, `std::unordered_set`)
  - Associative arrays (`std::map`, `std::multi_map`, `std::unordered_map`)
  - Various adaptors:
    - Queues (`std::queue`, `std::priority_queue`), stacks (`std::stack`)
    - Associative interfaces for arrays (`std::flat_map`, `std::flat_set`)

Containers
(hold sequences in difference ways)

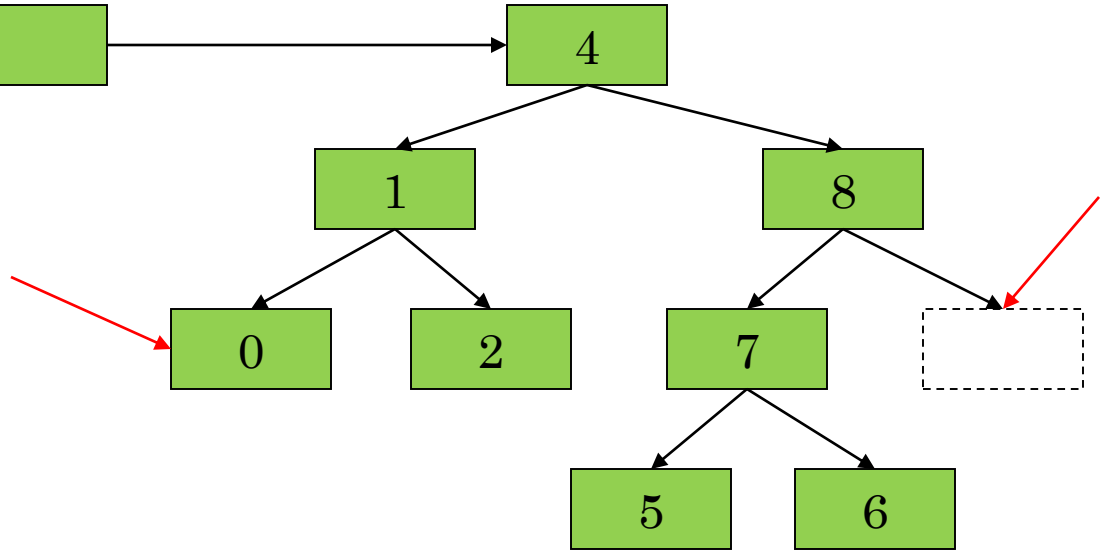
- `std::vector`



- `std::list`



- `std::set`



Dynamic arrays (std::vector)

- All containers are generic types (templates) so they can be used to store items of arbitrary types
- Dynamic arrays are the first choice container type
  - Rule: use `std::vector` by default
  - If you can't use `std::vector`, think again and change your approach such that you can use `std::vector`
- Rely on contiguous memory
  - Makes them `O(1)` in terms of access time
  - Insertion and deletion is `O(N)`, however (except at the end)

Dynamic arrays (std::vector)

```
// An array of integers, initialized with five elements:
std::vector<std::int64_t> data = {1, 2, 3, 4, 5};

// Elements can be accessed using the operator []
std::cout << "data[3] == " << data[3] << "\n";    // data[3] == 4

// An array of arrays of integers follows the same pattern
std::vector<std::vector<std::int64_t>> data2d = {{1, 2, 3}, {2, 3}, {4, 5, 6}, {7}};

// Same logic for accessing elements
std::cout << "data2d[1][0] == " << data2d[1][0] << "\n";    // data2d[1][0] == 2
std::cout << "data2d[2][2] == " << data2d[2][2] << "\n";    // data2d[2][2] == 6

// We can use element access to mutate the array
data2d[3] = {1, 2, 3, 4, 5};
std::cout << "data2d[3][2] == " << data2d[3][2] << "\n";    // data2d[3][2] == 3
data2d[1][0] = 42;
std::cout << "data2d[1][0] == " << data2d[1][0] << "\n";    // data2d[1][0] == 42
```

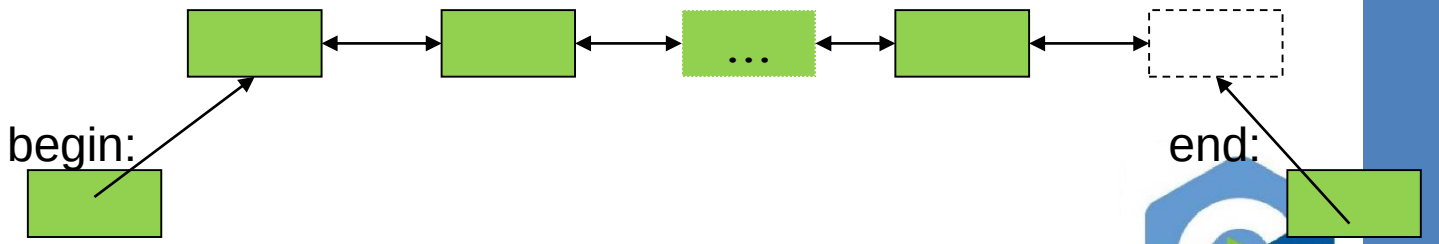
Dynamic arrays (std::vector)

- Note that accessing nonexistent elements makes the program malformed
  - Always ensure the index is valid (is in range) when accessing elements by index (i.e. `0 <= index && index < size`)
  - Various debugging tools help with identifying index-out-of range errors
    - Windows (msvc): standard library in debug builds
    - Linux, Mac (gcc, clang): address sanitizer, undefined behavior sanitizer
  - If you want to be sure, use `at()` instead `operator[]()`
    - `at()` performs index checking at runtime and throws an error if index is out of bounds

Algorithms and Iterators

The simplest Algorithm: `std::find`

```
// Find the first element that is equal to a value
template <typename In, typename T> requires(std::input_iterator<In> && ...)
In find(In first, In last, T const& val)
{
    while (first != last && *first != val)
        ++first;
    return first;
}

// find an int in a given vector
void f(std::vector<int> const& v, int x) {
    auto p = std::find(v.begin(), v.end(), x);
    if (p != v.end()) {
        // we found x
        std::cout << *p << "\n";
    }
}
```

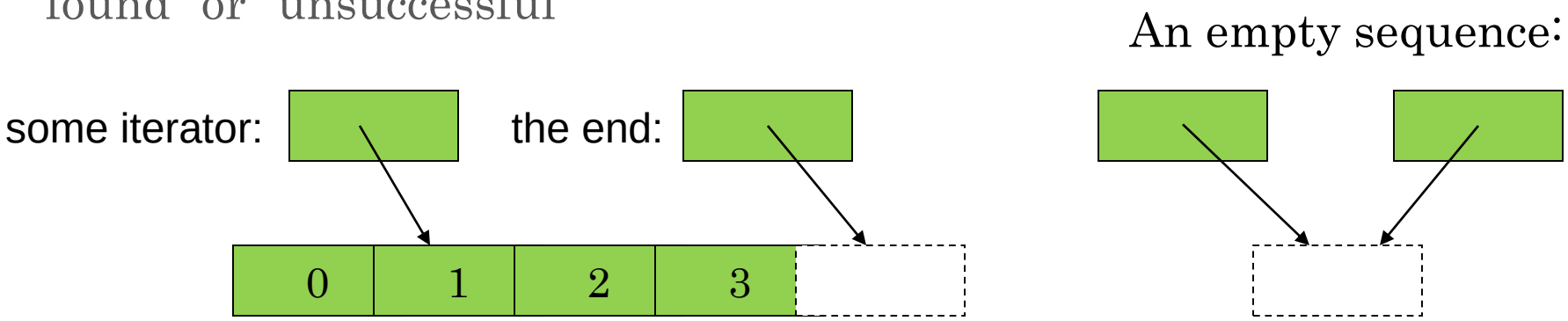


std::find: Generic for both, element type and container type

```cpp
// works for list of strings
void f(std::list<std::string> const& l, std::string x)
{
    auto p = std::find(l.begin(), l.end(), x);        // <-- here
    if (p != l.end())    // did we find x?
    {
        std::cout << "Found x: " << *p << "\n";
    }
}

// works of set of doubles
void f(std::set<double> const& s, double x)
{
    auto p = std::find(s.begin(), s.end(), x);        // <-- here
    if (p != s.end())    // did we find x?
    {
        std::cout << "Found x: " << *p << "\n";
    }
}
```

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"



Simple algorithm: `find_if`

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

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

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

Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a `bool`
- A function:

```
bool odd(int i) { return i % 2; }   // % is the remainder
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?
```

- A lambda function:

```
auto odd = [](int i) { return i % 2; }
odd(7);     // call odd: is 7 odd?
```

Function objects

- A concrete example using state:

```
template <typename T>
struct less_than {
    T val;    // value to compare with

    less_than(T x) : val(x) {}

    bool operator()(T const& x) const {
        return x < val;
    }
};

// find x < 43 in std::vector<int>:
p = find_if(v.begin(), v.end(), less_than(43));

// find x < “perfection” in std::list<std::string>:
q = find_if(ls.begin(), ls.end(), less_than("perfection"));
```

Function objects

- A very efficient technique
  - inlining very easy
    - and effective with current compilers
  - Faster than equivalent function
    - And sometimes you can't write an equivalent function
- The main method of policy parameterization in the STL
- Key to emulating functional programming techniques in C++
  - 'Using code as data'

Lambda

- A concrete example:

```
// find x < 43 in std::vector<int>:
p = std::find_if(v.begin(), v.end(), [](auto x) { return x < 43; });

// find x < “perfection” in std::list<std::string>:
q = std::find_if(ls.begin(), ls.end(), [](auto x) { return x < "perfection"; });
```

Policy Parameterization

- Whenever you have a useful algorithm, you eventually want to parameterize it by a "policy".
  - For example, we need to parameterize sort by the comparison criteria

```
struct record {
    std::string name;    // standard string for ease of use
    char addr[24];       // old C-style string to match database layout
    // ...
};

std::vector<record> vr = {
    {"John Doe", "42 Main Street"},
    {"Donald Duck", "123 3rd Avenue"},
};

std::sort(vr.begin(), vr.end(), cmp_by_name());    // sort by name
std::sort(vr.begin(), vr.end(), cmp_by_addr());    // sort by addr
```

Comparisons

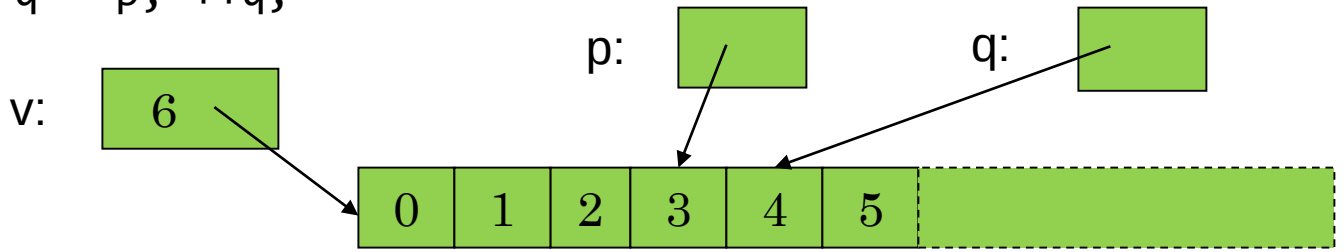
```
// Different comparisons for record objects:
struct cmp_by_name {
    bool operator()(record const& a, record const& b) const
    {
        return a.name < b.name; // look at the name field of Rec
    }
};

struct cmp_by_addr {
    bool operator()(record const& a, record const& b) const
    {
        return 0 < std::strncmp(a.addr, b.addr, 24);    // correct?
    }
};
// note how the comparison function objects are used to hide ugly
// and error-prone code
```

std::vector

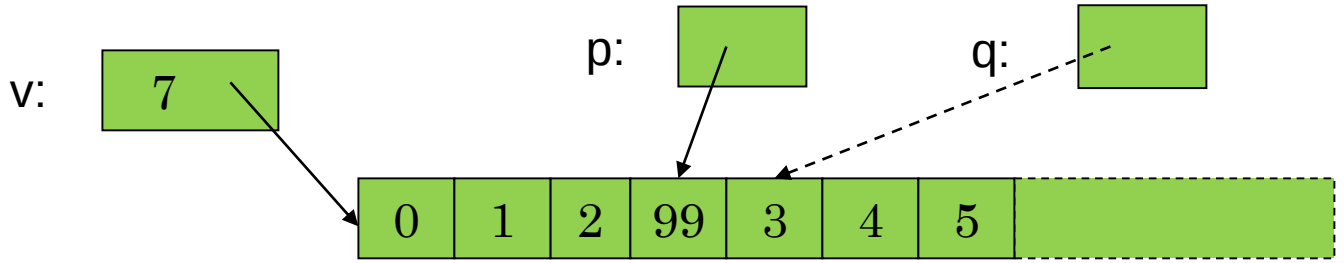
```
template <typename T>
class vector {
    T* elements;
public:
    // ...
    using iterator = ...;    // the type of an iterator is implementation defined
                             // and it (usefully) varies (e.g. range checked iterators)
    // a vector iterator could be a pointer to an element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

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

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

insert() into std::vector

```
std::vector<int>::iterator p = v.begin(); ++p; ++p; ++p;
std::vector<int>::iterator q = p; ++q;
```

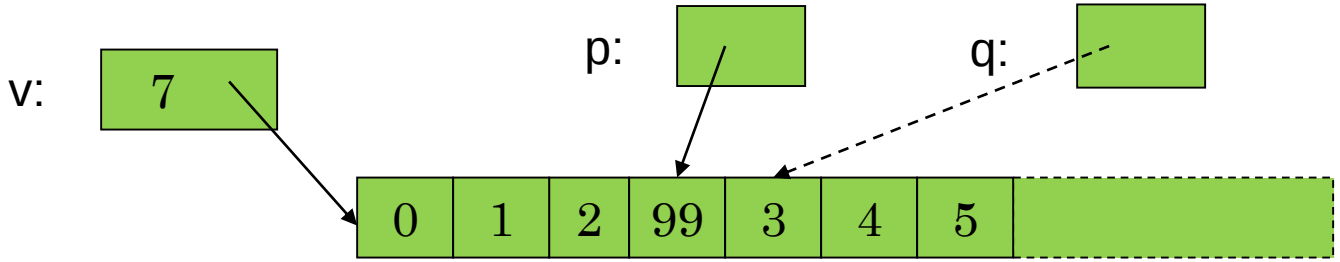



```
p = v.insert(p, 99);  // leaves p pointing at the inserted element
```

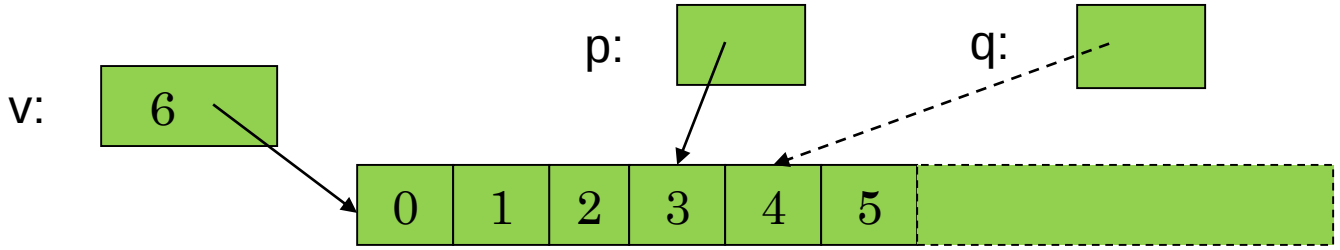


- Note: q is invalid after the **insert()**
- Note: Some elements moved; all elements could have moved

erase() from std::vector

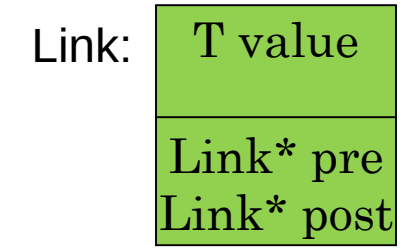



```
// leaves p pointing at the element after the erased one
p = v.erase(p);
```



- vector elements move when you insert() or erase()
- Iterators into a vector are invalidated by insert() and erase()

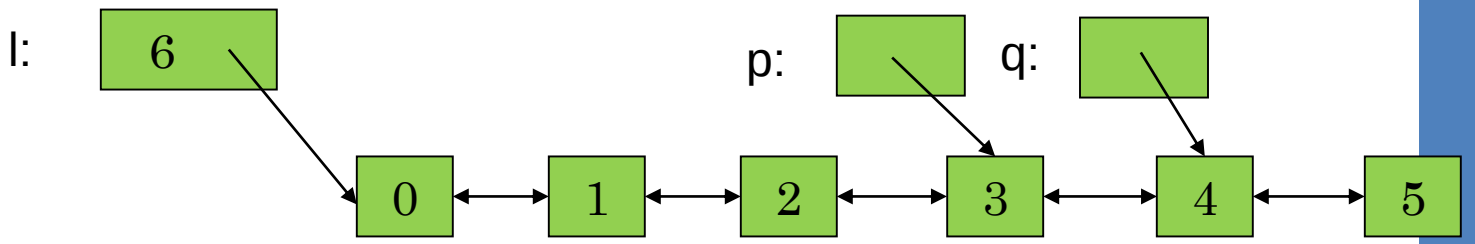
std::list



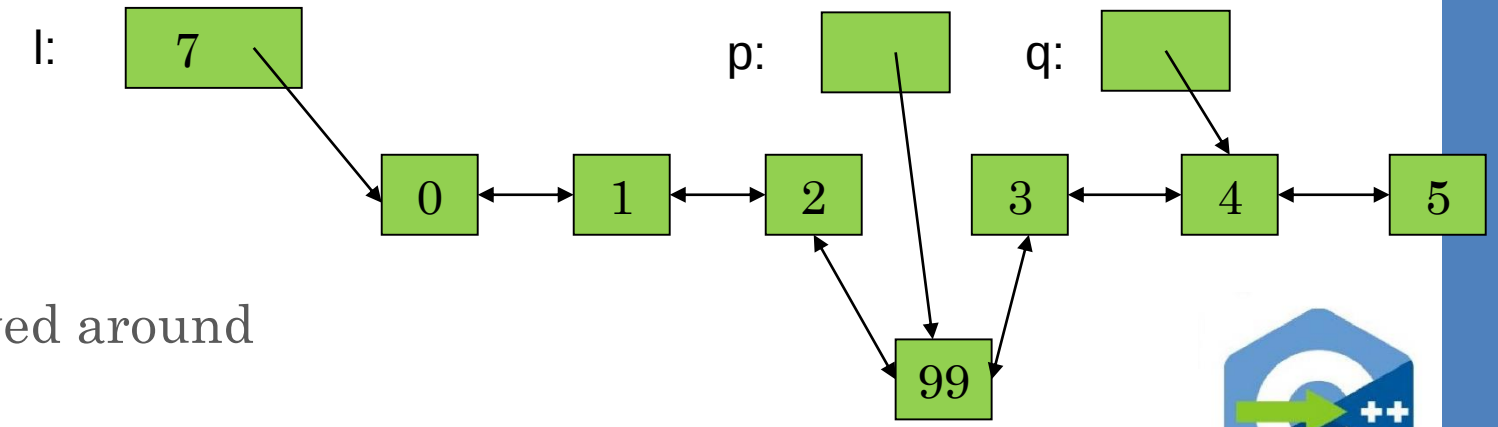

```
template <typename T>
class list {
    Link* elements;
public:
    // ...
    using iterator = ...;    // the type of an iterator is implementation defined
                             // and it (usefully) varies (e.g. range checked iterators)
        // a list iterator could be a pointer to a link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

    iterator begin();    // points to first element
    const_iterator begin() const;
    iterator end();      // points 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
};
```

insert() into std::list

```
std::list<int>::iterator p = l.begin(); ++p; ++p; ++p;
std::list<int>::iterator q = p; ++q;
```

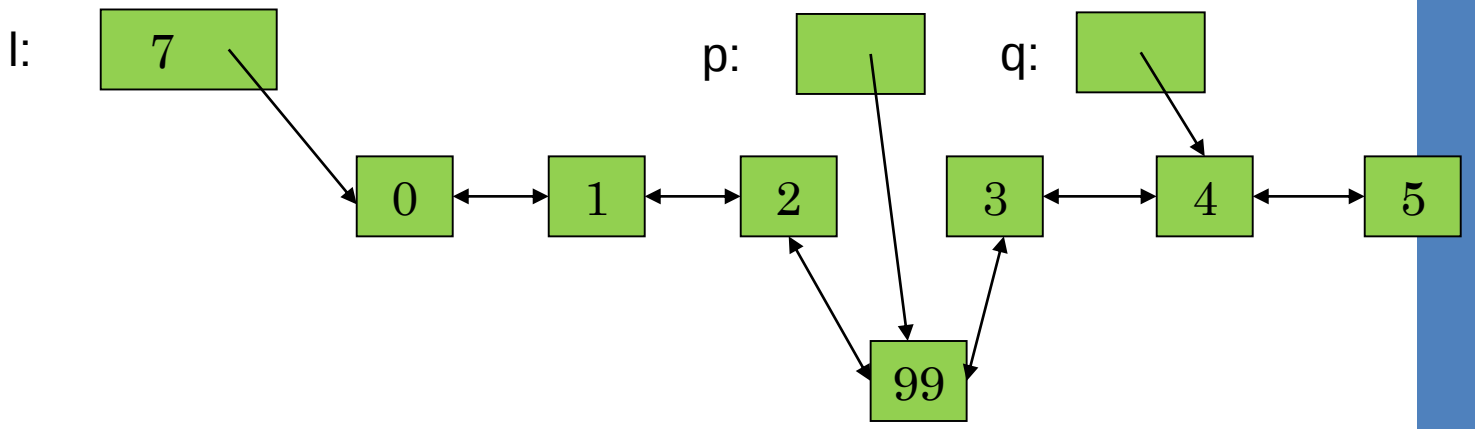



```
p = l.insert(p, 99);     // leaves p pointing at the inserted element
```

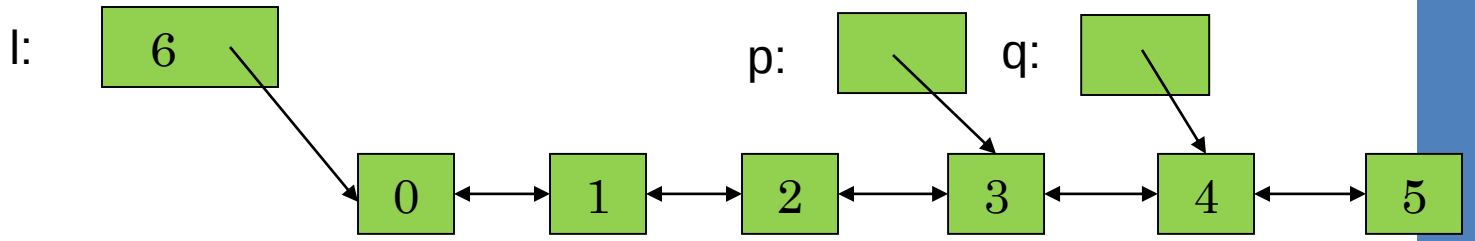


- Note: q is unaffected
- Note: No elements moved around

erase() from std::list




```
// leaves p pointing at the element after the erased one
p = l.erase(p);
```



- Note: list elements do not move when you insert() or erase()

std::array

```
template <typename T, size_t N>
class array {
    T elements[N];
public:
    // ...
    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 one beyond the last element
    const_iterator end() const;

    // no erase
    // no insert
};
```

Ways of traversing a container

```
for (int i = 0; i < v.size(); ++i)    // why int?
    // do something with v[i]

// longer but always correct
for (std::vector<int>::size_type i = 0; i < v.size(); ++i)
    // do something with v[i]

for (std::vector<int>::iterator p = v.begin(); p != v.end(); ++p)
    // do something with *p

for (auto p = v.begin(); p != v.end(); ++p)
    // do something with *p

for (auto const& e : v)
    // do something with e
```

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP