Object-Oriented Programming

Iterator

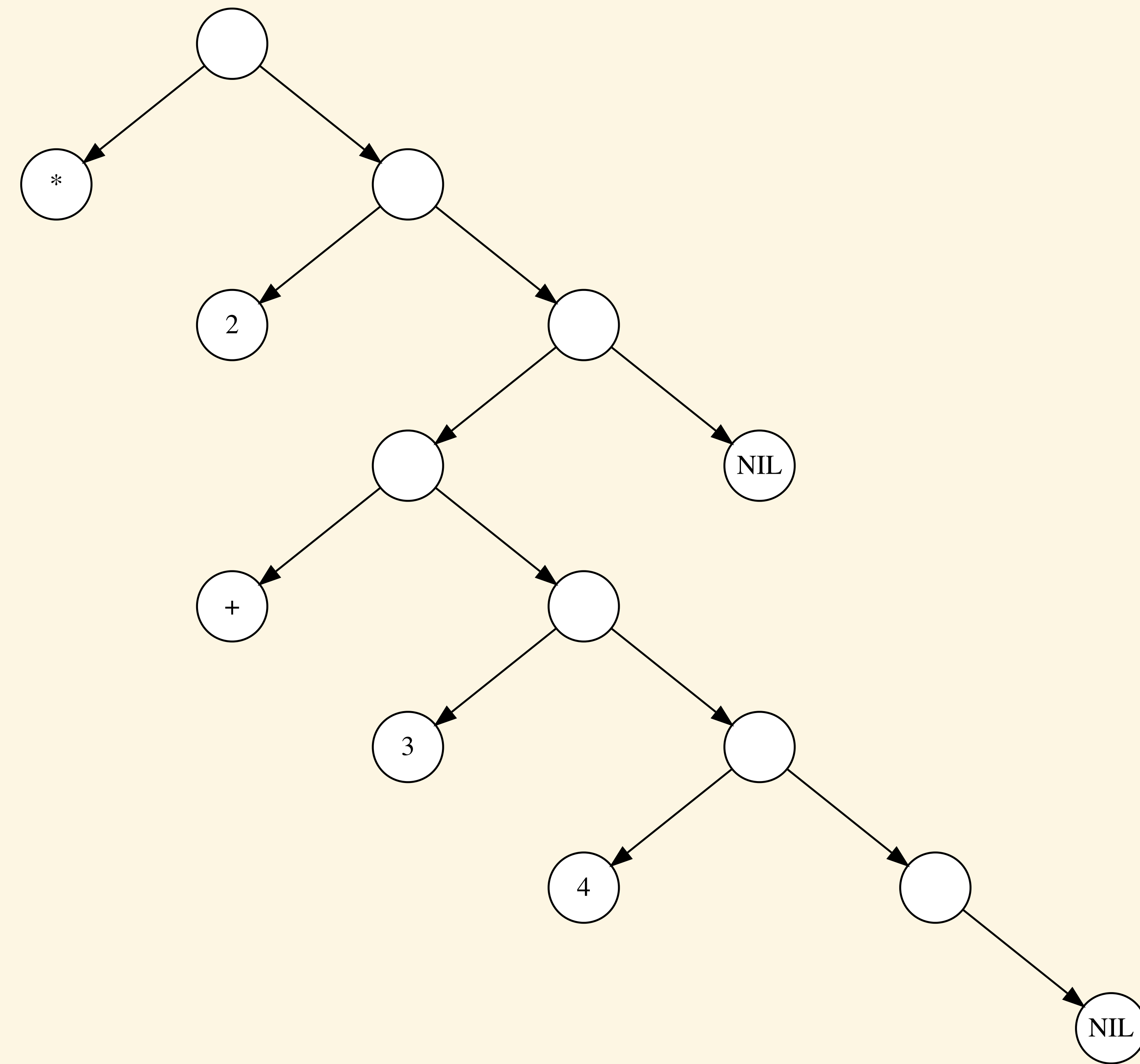
Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Credits

Based on On Iteration by Andrei Alexandrescu

Beginnings: Lisp



- Lisp is a *functional programming language*
- 2 * (3 + 4) would be

  `(* 2 (+ 3 4))`
- Primary data type is a singly-linked list (*s-expression*)
- Forward iterator only
- Indexing makes no sense

Beginnings: c-pointers

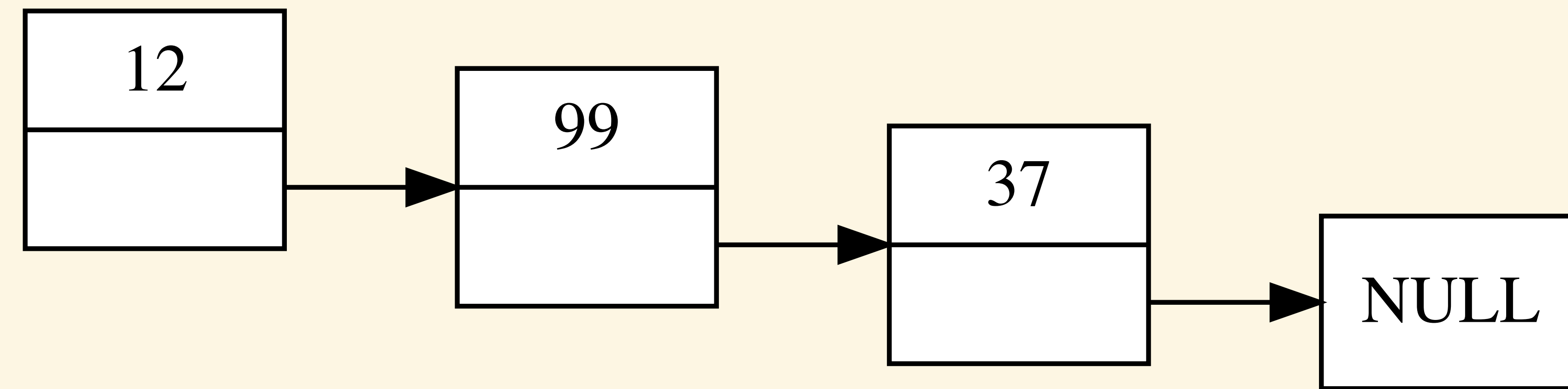
```
bool palindrome(const char* p, const char* q) {

    --q;
    while (p < q) {
        if (*p++ != *q--)
            return false;
    }

    return true;
}
```

- dereference: *p
- pre-decrement: −−q
- post-decrement: q−−
- post-increment: p++
- comparison: p < q

Beginnings: c-pointers

```
bool palindrome(const char* p, const char* q) {

    // @operator pre-decrement
    --q;
    // @operator comparison
    while (p < q) {
        // @operator dereference post-increment post-decrement
        if (*p++ != *q--)
            return false;
    }

    return true;
}
```

Beginnings: Linked Data Structure



- Linked list
- Indexing is expensive, linear time, O(n)
- Getting to the next element is cheap, constant time, O(1)

Iterator Design Pattern

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

- Gang of Four (GoF) Design Patterns: Elements of Reusable Object-Oriented Software

GoF Iterator

```
// Iterator for a container with elements of type T
interface Iterator {
    // Restart iteration
    void First();
    // Advance to next item
    void Next();
    // Are we done yet?
    bool IsDone();
    // Get current item
    T CurrentItem();
}
```

- Often, `First()` is replaced by a copy
- Forward iteration only
- Has been generalized to many related situations

C++ Library: Sequence Containers

- `std::vector`
- `std::list`
- `std::deque`
- `std::array`
- `std::forward_list`
- *`std::string`*

C++ Library: Algorithms

- `std::copy()`
- `std::sort()`
- `std::search()`
- `std::find()`
- … and lots more

C++ Iterators

Iterators tie data structures and algorithms together *

- Problem 1:
  - *m* algorithms
  - *n* containers
  - = $m \times n$ implementations
- Problem 2: User-defined containers
- Solution: Common algorithm implementation for as many containers as possible, defined so that users can create new containers
- Focus on algorithms and make containers adapt
- Inspired by c-array pointers

C-Array Operators

```
const int ARSIZE = 5;
int ar[ARSIZE] = { 5, 3, 2, 1, 6 };

// output ar with each value on its own line
for (int* p = ar; p < ar + ARSIZE; ++p)
    std::cout << *p << '\n';
```

C-Array Pointers	Description	C++ Iterator term
`int* p = ar;`	Start at the beginning	*initialize*
`++p`	Advance to the next item	*increment*
`p < ar + ARSIZE`	Keep going?	*comparison*
`*p`	Get current item	*dereference*

Iterator Operations based on Pointers (C++)

Name	C/C++	C++11
dereference	`*p`	
increment	`++p`, (`p++`)	
decrement	`--p`, (`p--`)	
copy	`curp = p`	
comparison	`p != end`	
addition n	`p + n`	`std::next(p, n)`
subtraction n	`p - n`	`std::prev(p, n)`
increment n	`p += n`	`std::advance(p, n)`
decrement n	`p -= n`	
indexing	`p[n]`	
distance	`p - begin`	`std::distance(begin, p)`

Use of C++ Iterators

```
for (auto p = begin; p != end; ++p)
    std::cout << *p << '\n';
```

- `begin` is an iterator to the first element
- `end` is an iterator *past the last element*
- Uses `!=` instead of < because sequential elements may not be adjacent
- Uses `auto` to make declaration easier
- Use of `auto` also makes the code adaptable to multiple sequence containers
- For this code, the *iterator requirements* are *assignment*, *comparison*, *increment*, *dereference*

C++ Iterator Operation

```
std::vector<int> v;

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

assert(sizeof(double) == 8);
assert(sizeof(p) == 8);

void palindrome(
     std::vector<int>::const_iterator begin,
     std::vector<int>::const_iterator end);

std::sort(v.begin(), v.end());
```

- Iterators are separate objects from the container object
- We can have multiple simultaneous iterators for the same container object
- Iterators are small and typically have just a couple of fields/data members
- When IN is often passed by value and not const reference
- The C++ standard algorithms are implemented using iterators

Pre and Post

Term	C/C++	Order of Operations
pre-increment	`++p`	`p += 1;` `return p;`

post-increment	`p++`	`auto t = p;`
		`p += 1;`
		`return t;`

pre-decrement	`--p`	`p -= 1;` `return p;`

post-decrement	`p--`	`auto t = p;`
		`p -= 1;`
		`return t;`

Class Pre and Post Implementation

```
// pre-increment, ++p
Iterator& Iterator::operator++() {

    // perform operation ...

    return *this;
}

// post-increment, p++
Iterator Iterator::operator++(int) {

    Iterator copy(*this);
    operator++();
    return copy;
}
```

- Post-increment can be implemented using pre-increment
- Post-increment requires an additional copy
- Post-increment is never more efficient than pre-increment
- Post-increment is more difficult to understand than pre-increment
- Decrement has the same issues
- **Use ++p (pre-increment) instead of p++ (post-increment)**
- **Use ––p (pre-decrement) instead of p–– (post-decrement)**

Container Support for Iterators: std::vector

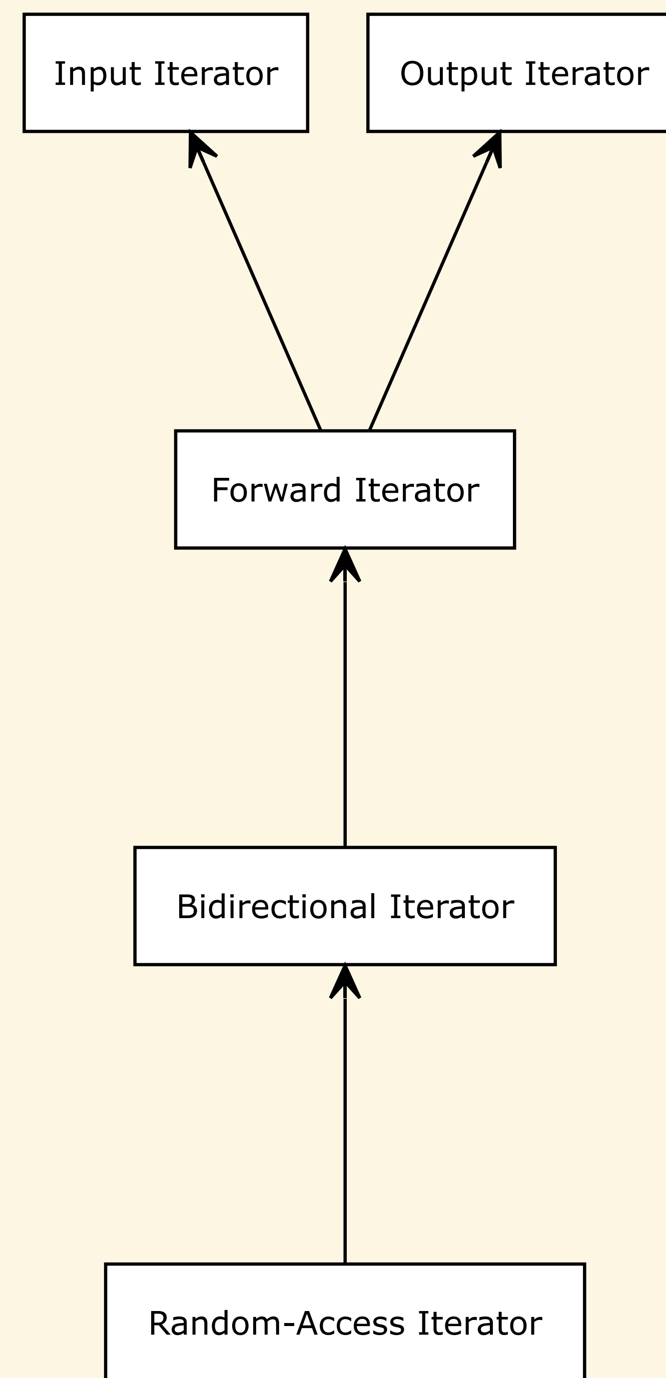
Description	Method
Iterator to first element	**`v.begin()`**
Iterator *past the end* of the vector	**`v.end()`**
const Iterator form of `v.begin()`	`v.cbegin()`
const Iterator form of `v.end()`	`v.cend()`
Reverse iterator to the element preceding the first element	`v.rbegin()`
Reverse iterator to the last element in the vector	`v.rend()`
const Iterator form of `v.rbegin()`	`v.crbegin()`
const Iterator form of `v.rend()`	`v.crend()`

Common Interface

```
std::vector<int> d{ 1, 2, 3 };
for (auto p = d.begin(); p != d.end(); ++p)
    std::cout << *p << '\n';

std::list<int> d{ 1, 2, 3 };
for (auto p = d.begin(); p != d.end(); ++p)
    std::cout << *p << '\n';

std::deque<int> d{ 1, 2, 3 };
for (auto p = d.begin(); p != d.end(); ++p)
    std::cout << *p << '\n';

std::string d = "abc";
for (auto p = d.begin(); p != d.end(); ++p)
    std::cout << *p << '\n';
```

- Consistent usage helps in the comprehension of the code
- Can change the data structure without changing the algorithm
- Easy to learn new data structures
- Can create user-defined data structures that work with the existing algorithms
- Can create new algorithms that work with existing containers

C++ Iterator Hierarchy



- *input iterators* - one-pass input from streams
- *output iterators* - one-pass output to streams
- *forward iterators* - model access to singly-linked lists
- *bidirectional iterators* - model access to double-linked lists
- *random-access iterators* model array access

Container Iterator Operators

Category	*p	++p	p++	curp = p	p != end	--p	p--	p[5]	p += 5	p - begin
forward	✔	✔	✔	✔	✔					
bidirectional	✔	✔	✔	✔	✔	✔	✔			
random-access	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Container Iterator Operators

- All iterator operations are constant time O(1)

- Pointers in C are random-access iterators

C++ Container Types

- *sequence container*: `std::array`, `std::vector`, `std::deque`, `std::list`, `std::forward_list`
- *associative container*: `std::set`, `std::map`, `std::multiset`, `std::multimap`
- *unordered associative Container*: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset`, `std::unordered_multimap`
- Special Containers: `std::string`
- Container Adapters: `std::stack`, `std::queue`, `std::priority_queue` do not provide iterators