The List Type

Lecture 12

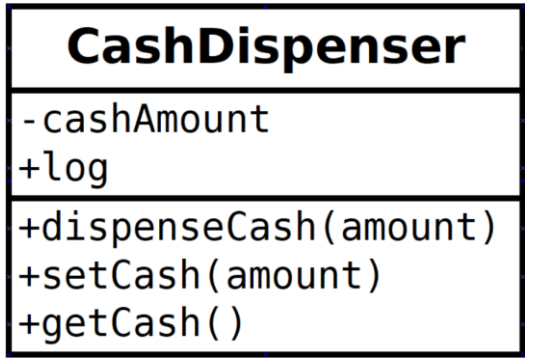
Hartmut Kaiser

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

Software Development Notes

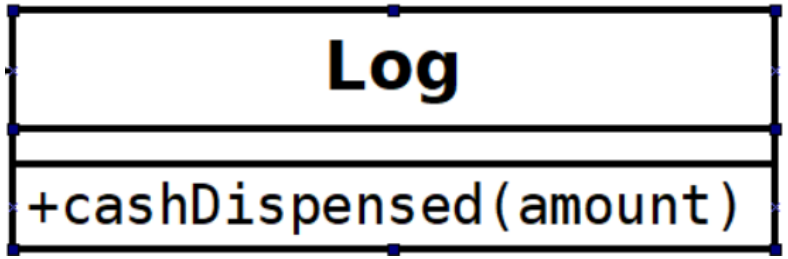
Class Diagrams

- Class diagrams are the most commonly used of the UML diagrams
  - 3 basic parts: Name, Fields (member variables), Methods
  - Visibility
    - + public
    - - private
    - # protected
      - Visible to inherited classes, but not other classes
  - Datatypes can follow member variables, but are optional
    - For example: - cashAmount : double



Related classes

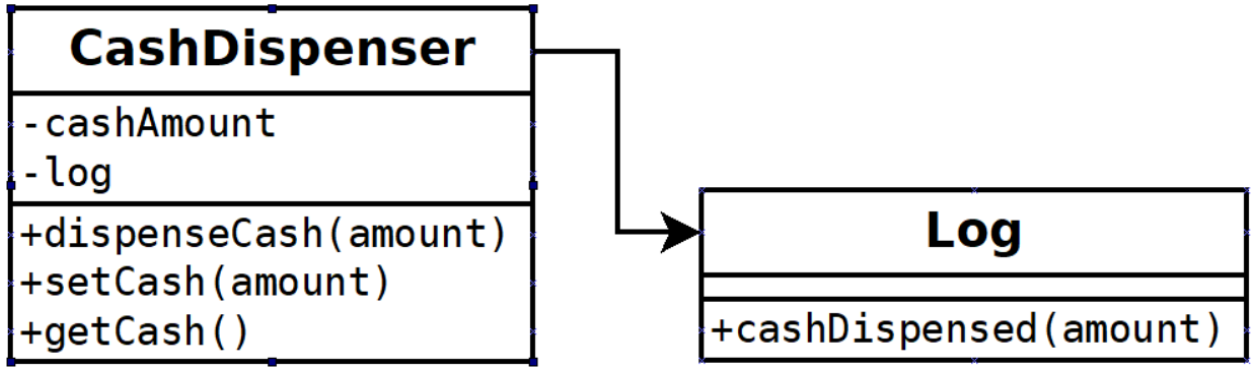
- What if we have a separate class that logs activity of a class



- We need to capture the relationship between the two classes

Association Relationship

- Association (aka delegation) is the most common connection between classes
  - Represents one class object storing a reference to another



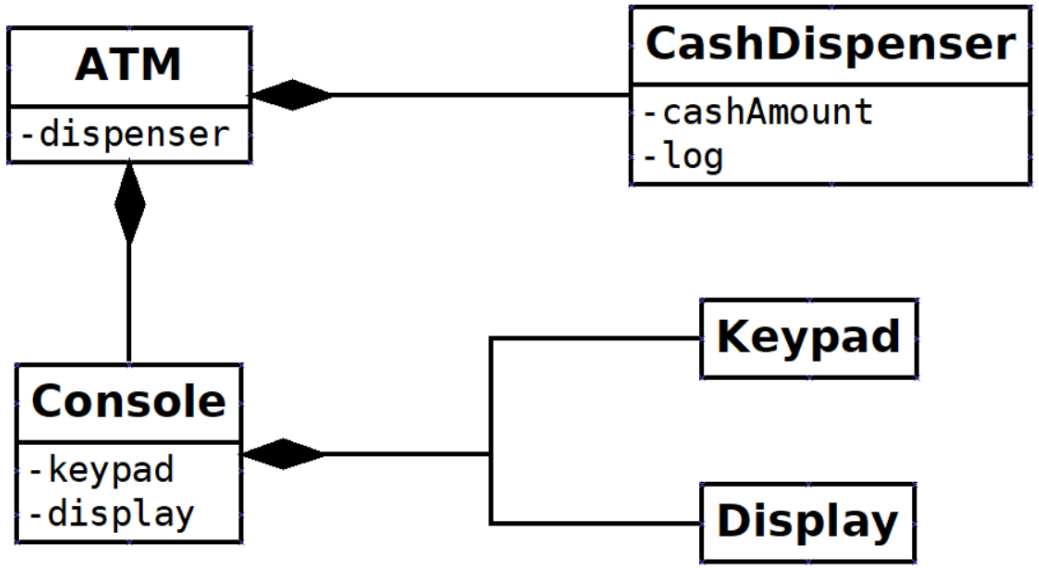
  - Represented by a solid line and a filled arrow
    - The source of the relationship (line) uses the target of the relationship (arrow)
    - For example, a CashDispenser uses a Log

Composition Relationship

- When one class is “made” of another class, the relationship is called composition
- Association vs. Composition
  - Composition is similar to association, but the relationship is stronger
  - If an object creates and “owns” an object, it is composed of that object
    - In other words, if object A cannot exist without object B, then object A is composed of object B
  - If an object is given to another object to use, they are merely associated (one delegates to the other)

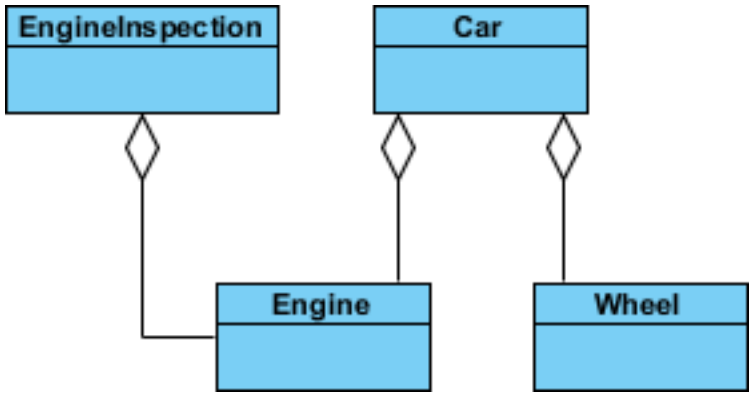
Composition Relationship

- Represented by a solid line and a filled diamond
  - The target of the relationship (diamond) is composed of the source (line) of the relationship
  - For example, an ATM is composed of a CashDispenser and a Console; a Console is composed of a Keypad and a Display



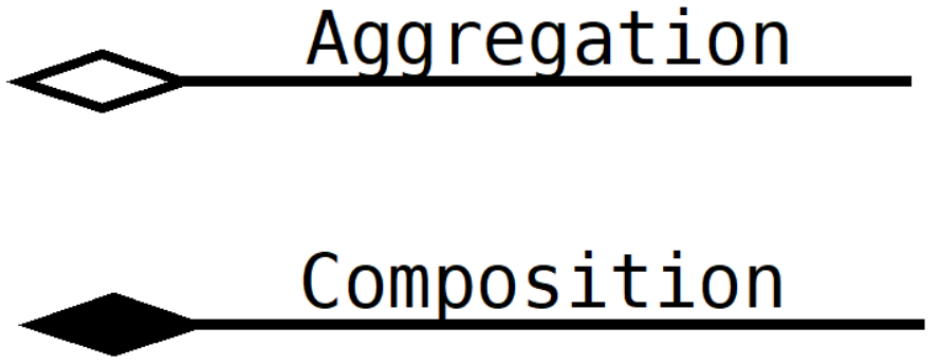
Aggregation Relationship

- Aggregation is similar to composition, but not as strong
- Aggregation implies a relationship where the child can exist independently of the parent
  - For example: Class (parent) and Student (child); delete the Class and the Students still exist
- Represented by a solid line and an unfilled diamond
  - The target of the relationship (diamond) is an aggregation of the source (line) of the relationship
  - For example, a Car contains an Engine and a Wheel; EngineInspection contains an Engine



https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

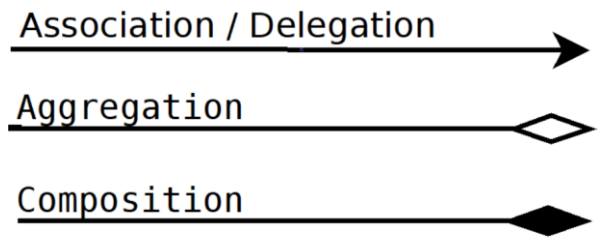
The Hollow Diamond



- Aggregation is like composition:
  - An object is made up of other objects
- But has an important difference:
  - An aggregating object does not own its members.
  - It's not responsible for creating/destroying them
  - (although it has the option)
- Fundamentally defined by destruction:
  - Composed objects go out of scope when their owner does
  - Aggregated objects do not (their life is independent)

Association vs. Aggregation vs. Composition

- Association
  - a very generic term used to represent when one class uses the functionalities provided by another class
  - Sibling-type relationship
- Composition
  - one parent class object owns another child class object and that child class object cannot meaningfully exist without the parent class object
  - Whole/part-type relationship
- Aggregation
  - If it can meaningfully exist without the parent class object
  - Container/contents-type relationship



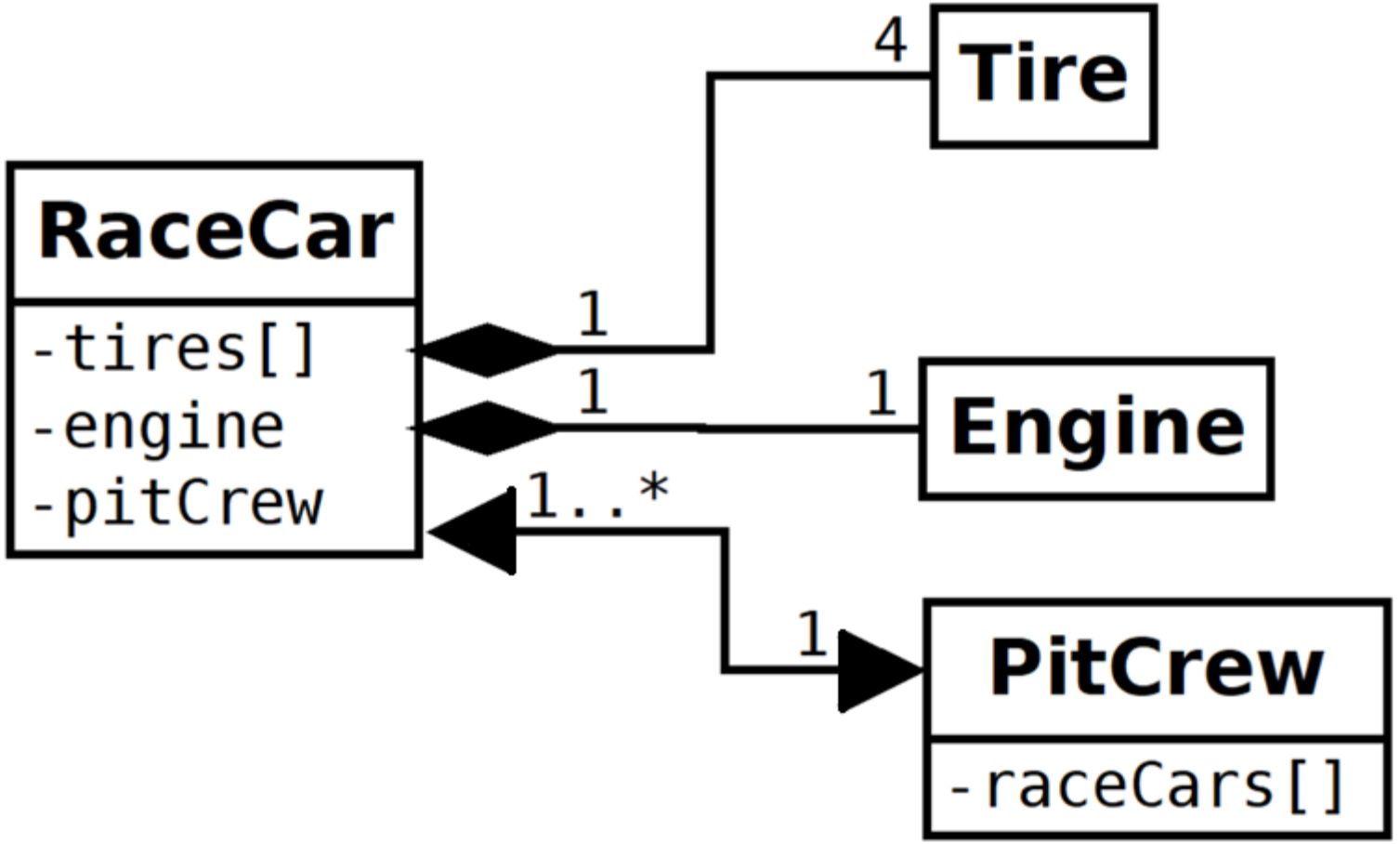
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

Class Diagrams: Multiplicity

- Number means how many instances of the class are contained/delegated to

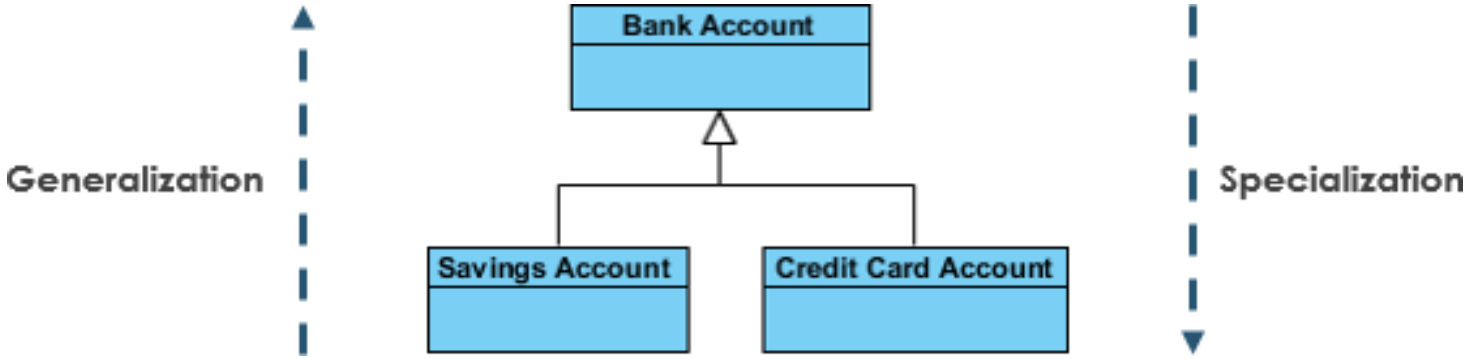
- The ".." establish a range:
  - 2..5 means at least two, but up to five
  - 1..* means at least one, but any number more
  - 0..* or just * means any number, including none

UML Cardinality/Multiplicity



Class Inheritance

- Class inheritance is the is-a relationship between classes
- Represented by a solid line and an unfilled closed arrow
  - The source of the relationship (line) is-a target (arrow) of the relationship
  - For example, a `Savings Account` is a `Bank Account`; a `Credit Card Account` is a `Bank Account`



The List Type

Abstract

- Performance of our example is fine for small inputs, but quickly degrades for a larger number of students. Why?
  - Inserting/deleting records in the middle of vectors requires moving big chunks of memory to preserve the random access property. Our program has a time complexity of $O(N^2)$.
- Different data structure is required

Using Vector Type

```
// version 3: iterators but no indexing
std::vector<student_info> extract_fails(std::vector<student_info>& students)
{
    std::vector<student_info> fail;
    auto iter = students.begin();
    while (iter != students.end()) {
        if (fail_grade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);    // watch out!
        } else
            ++iter;
    }
    return fail;
}
```

The List Type

- We rewrote code to remove reliance on indices
  - Now: change data structure allowing to efficiently delete elements from the middle of the sequence
- Common requirement, therefore: `std::list<>`
  - Vectors are optimized for fast random access
  - List are optimized for fast insert and delete at any point
    - Generally, slower than vectors
    - In heavy insertion and deletion scenarios, faster
  - Choose data structure depending on use case

The List Type

- List and vector share many common ideas
  - Can store almost any data type
  - Share almost all operations
- Converting our program to use lists is surprisingly simple
  - Main change is swapping container types
- Lists do not support indexing operations
  - But using iterators allows to get away without those

Using List Type

```
// version 4: using lists
std::list<student_info> extract_fails(std::list<student_info>& students)
{
    std::list<student_info> fail;
    auto iter = students.begin();
    while (iter != students.end()) {
        if (fail_grade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);    // watch out!
        } else
            ++iter;
    }
    return fail;
}
```

Iterator Invalidation

- For vectors, any insert/remove operation potentially invalidated all iterators.
  - Reallocation, data movement
  - Watch out when storing iterators (i.e. end())!
- For lists, no iterators are invalidated
  - Well, except for the one pointing to an erased element
  - But lists are not random access, that mean that iterators do not support random access operations

Sorting a List

- Standard sort algorithm is usable for random access iterators only
  - std::sort not usable for lists:

```
std::list<student_info> students;
student.sort(compare);
```

  - Instead of:

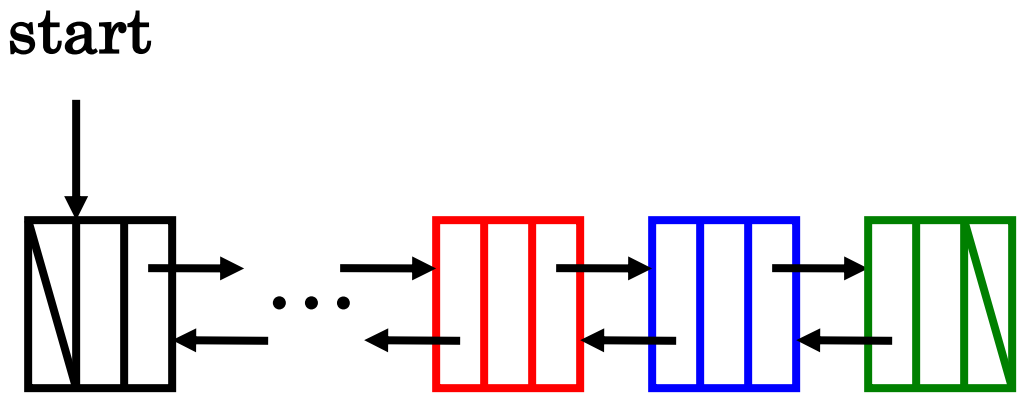
```
std::vector<student_info> students;
sort(students.begin(), students.end(), compare);
```

Performance Data

File size [records]	List [s]	Vector [s]
700	0.1	0.1
7000	0.8	6.7
70000	8.8	597.1

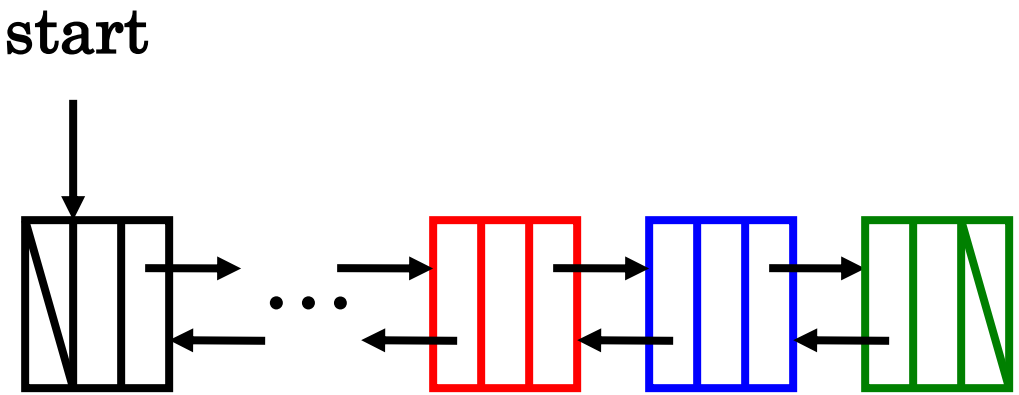
- 'Right' data structure is not a once and for all decision
  - Performance might not even matter
  - But depending on use case, performance might have a profound influence

Doubly-Linked List



- Given a pointer to a node in a doubly-linked list, we can remove the node in O(1) time.
- This isn't possible in a singly-linked list, since we must have a pointer to the node in front of the one we want to remove.

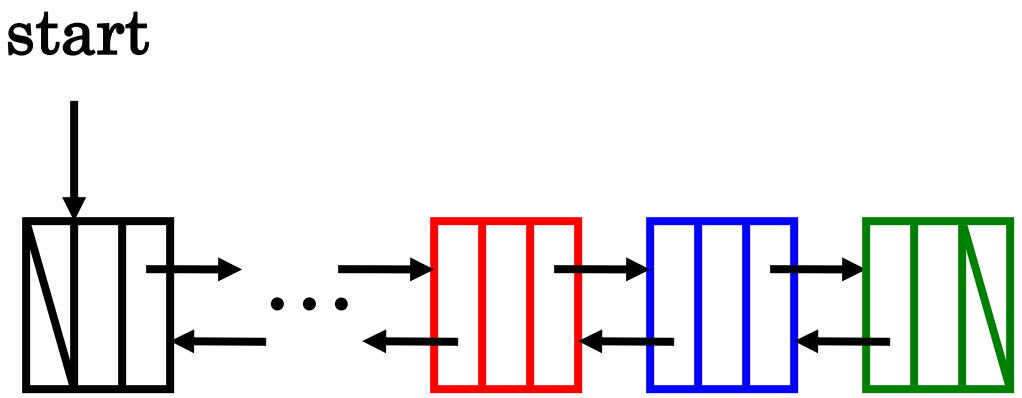
Doubly-Linked List



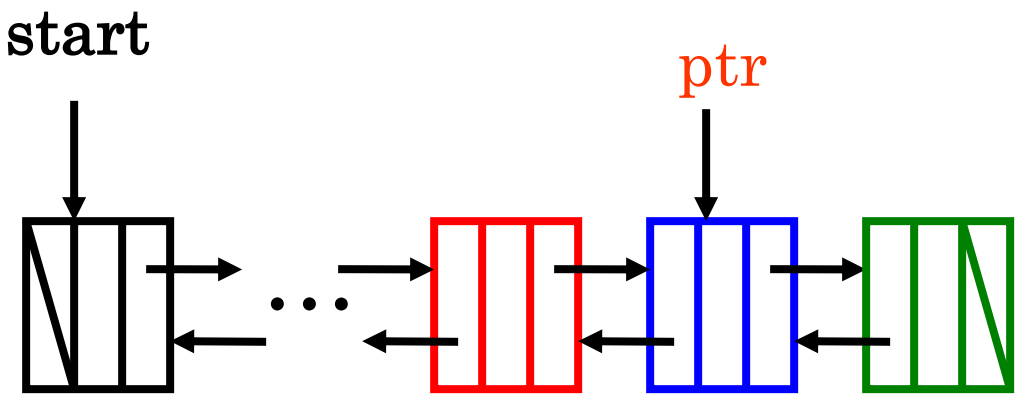

```
struct DLNode {
    DataType info;
    DLNodeIter next;
    DLNodeIter back;
};
```

Each node is made from a struct that looks something like this.

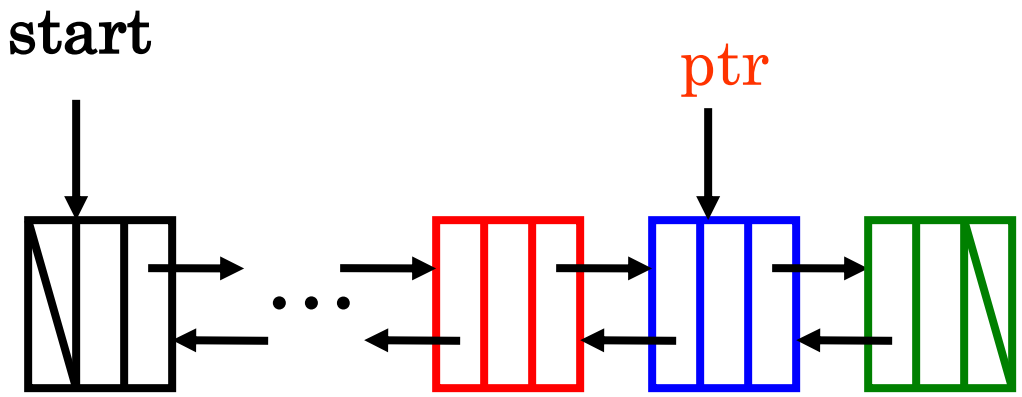
Doubly-Linked List



Doubly-Linked List

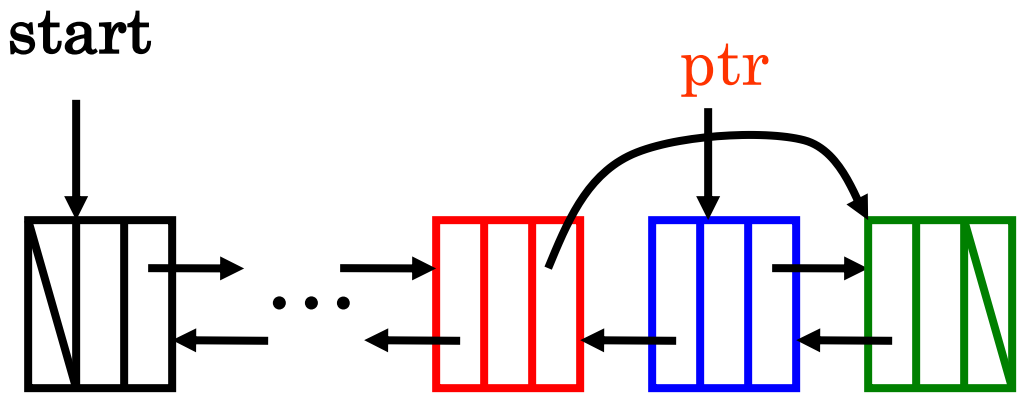


Doubly-Linked List: Delete Node



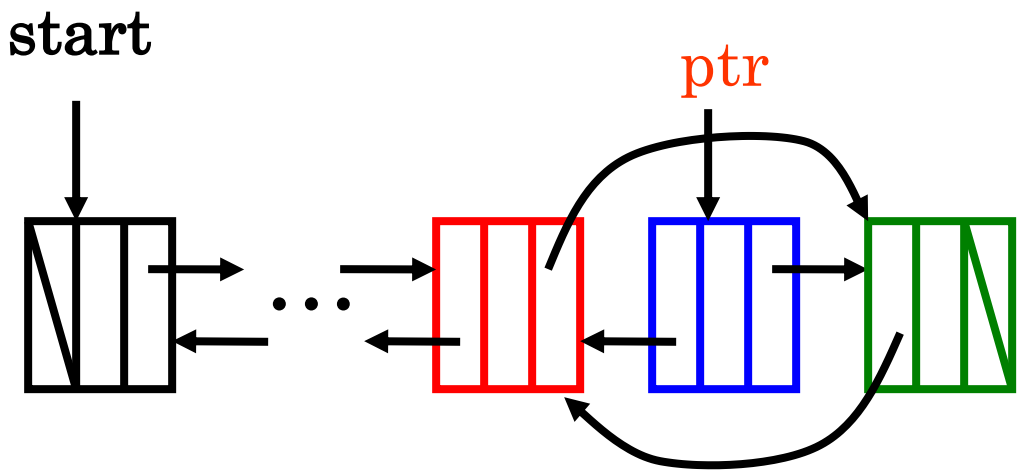

```
ptr->back->next = ptr->next;
ptr->next->back = ptr->back;
delete ptr;
```

Doubly-Linked List: Delete Node



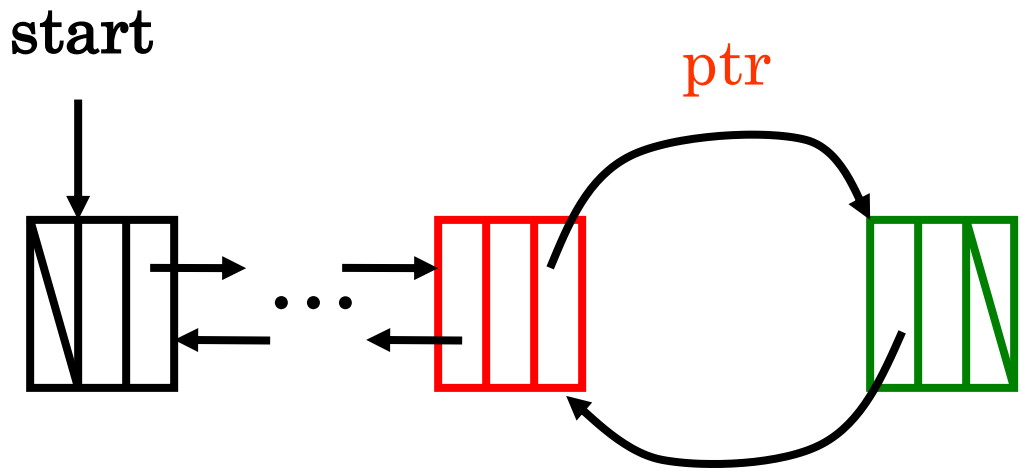

```
ptr->back->next = ptr->next;
ptr->next->back = ptr->back;
delete ptr;
```

Doubly-Linked List: Delete Node




```
ptr->back->next = ptr->next;
ptr->next->back = ptr->back;
delete ptr;
```

Doubly-Linked List: Delete Node

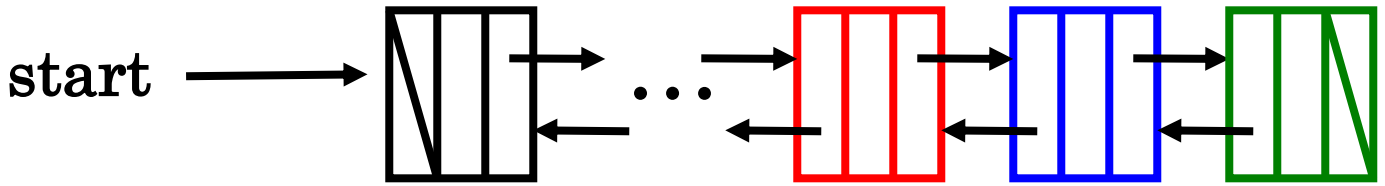



```
ptr->back->next = ptr->next;
ptr->next->back = ptr->back;
delete ptr;
```

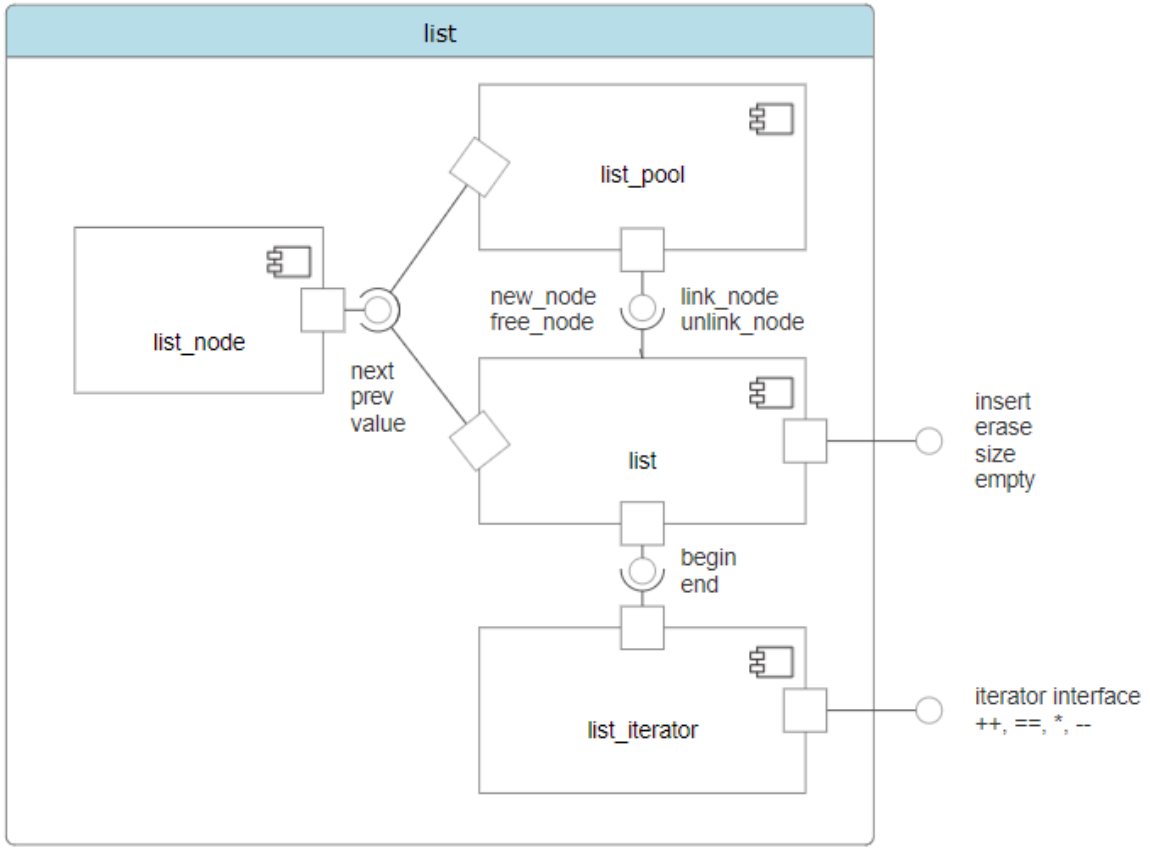
Creating Our Own `list` Type

Creating Our Own list Type

- Use `std::vector` whenever possible
  - If it's not possible think again and make it possible
- This is because of the architecture of modern computers
  - Working with data that is consecutive in memory is much faster than 'chasing pointers'
- Standard lists are implemented as interlinked nodes
  - Each element stored in a list is allocated separately
  - Nodes are not consecutive in memory, but may be randomly dispersed



Our `list` Type, Component Diagram

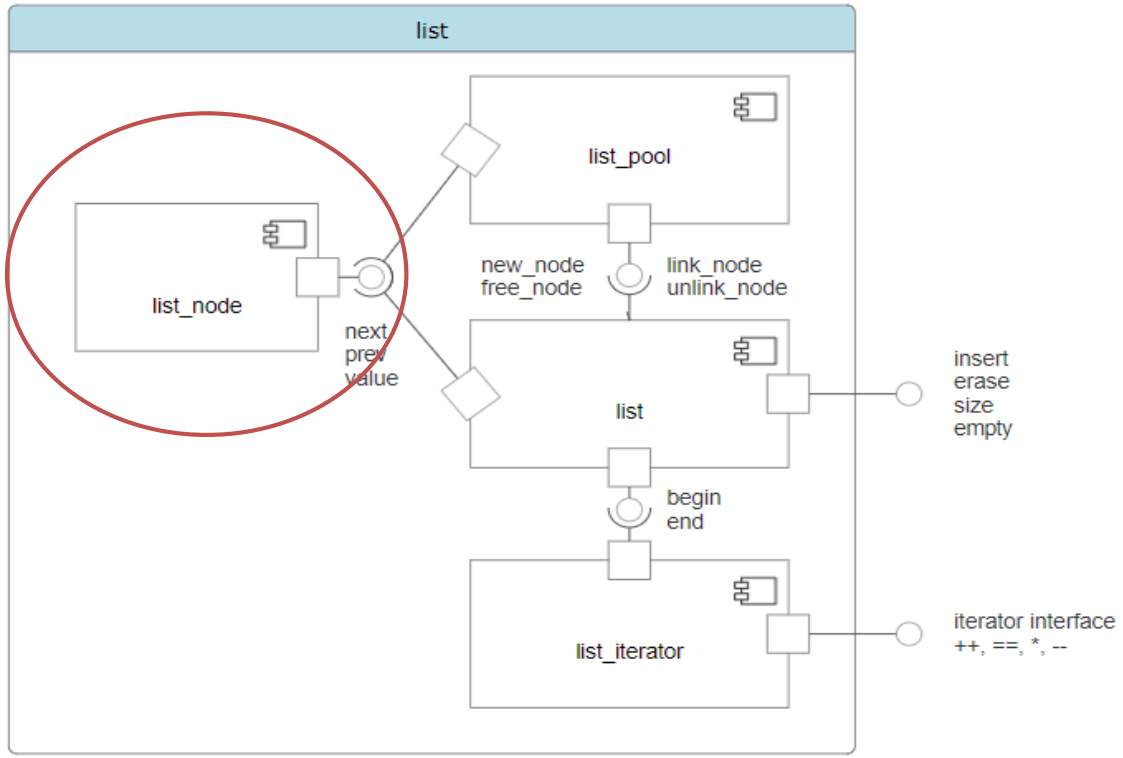


Creating Our Own list Type

- We will build an extremely fast `list<T>` implementation
  - Avoid spreading list nodes around in memory
  - Pooling all list nodes in one place (using `std::vector`)
- Combine the best of two worlds
  - Fast insert/delete in the middle
  - Fast traversal as nodes are consecutive in memory
- We will build a `list_pool<T>` user defined type that
  - Stores and recycles instances of a `list_node<T>` user defined type using a `std::vector<list_node<T>>`
  - Refers to the node instances by their index into the `std::vector`
  - Manages a free list of nodes that have been 'erased'

Creating the `list_node` Type

- A `list_node<T>` holds the node value and the indices of the next and previous nodes (or zero if there is none)
- Both `T` and `list_node<T>` are `SemiRegular`



Creating the `list_node` Type

- A `list_node<T>` holds the node value and the indices of the next and previous nodes (or zero if there is none)
- Both `T` and `list_node<T>` are `SemiRegular`

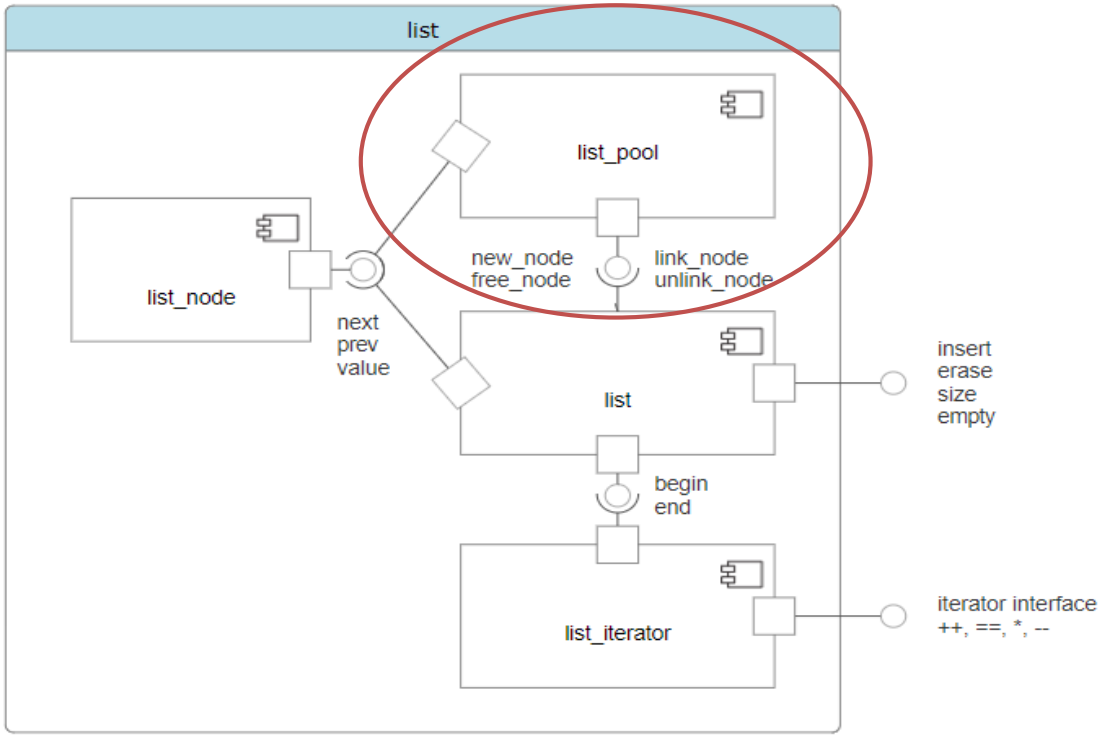
```
template <std::semiregular T>
struct list_node
{
    T value;                // the node value
    std::size_t prev = 0;   // pool index of previous node
    std::size_t next = 0;   // pool index of next node
};
```

Creating the `list_pool` Type

- A list pool is an object with many list nodes inside
- Our lists will not be full STL containers
  - Simplified implementation to avoid having to talk about pointers
  - Elements in a container should go away when the container goes away (but they don't)
  - Still a reasonable implementation
- Elements in our list will exist until
  - Either the list goes out of scope
  - A list node is explicitly erased and then reused (a new value is added to the list)
- The number of nodes allocated by the `list_pool` will never shrink, only grow

Creating a List Pool

- A `list_pool<T>` holds the list of nodes and the starting index of the free list (or zero if it is empty)
- The `list_pool<T>` is `SemiRegular`



Creating a List Pool

- A `list_pool<T>` holds the list of nodes and the starting index of the free list (or zero if it is empty)
- The `list_pool<T>` is `SemiRegular`

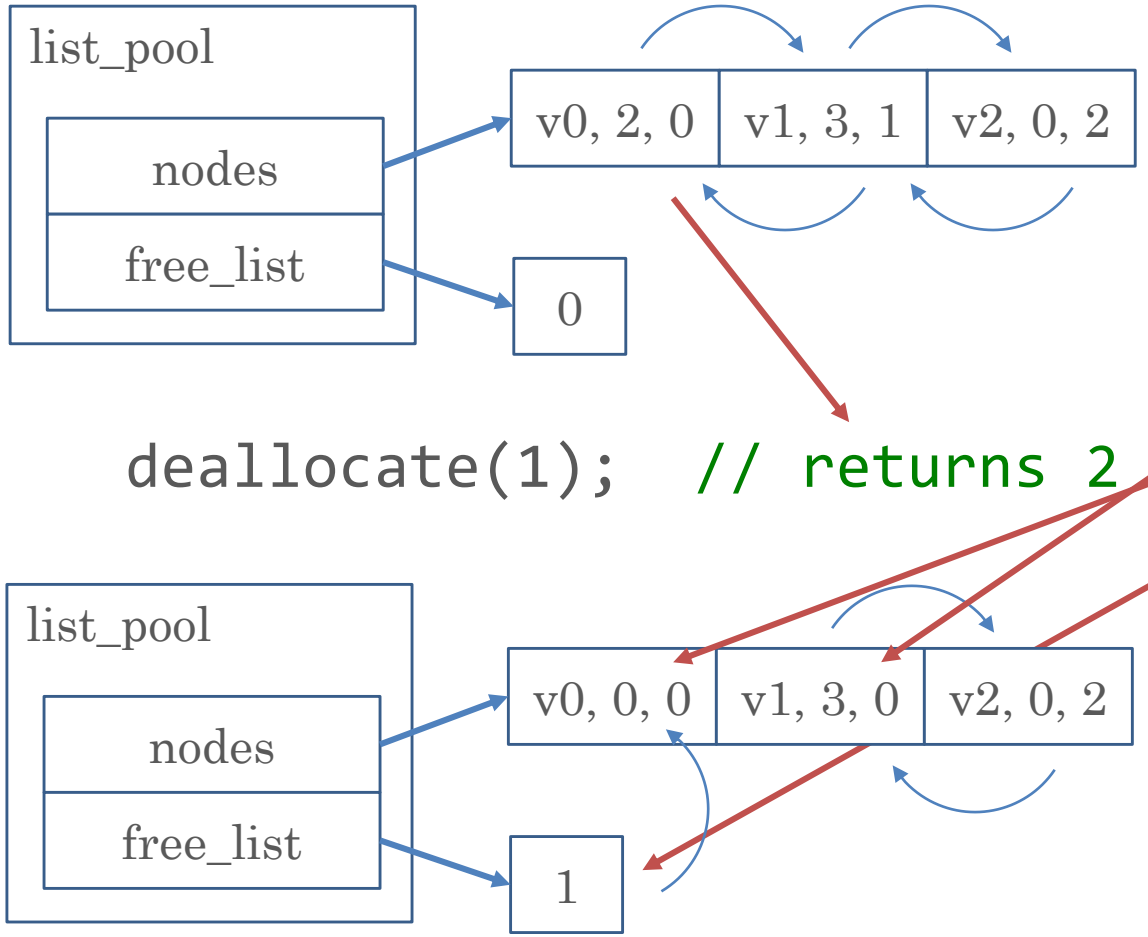
```
template <std::semiregular T>
struct list_pool
{
    std::vector<list_node<T>> pool;     // list of nodes
    std::size_t free_list = 0;          // index of first free (usable) node (if any)

    list_node<T>& node(std::size_t pos) { return pool[pos - 1]; } // access given node
    T& value(std::size_t pos) { return node(pos).value; }         // access node value
    std::size_t& next(std::size_t pos) { return node(pos).next; } // access next node
    std::size_t& prev(std::size_t pos) { return node(pos).prev; } // access prev node
};
```

Creating a List Pool: `deallocate`

- Deallocation simply adds the node to the free list

```
std::size_t deallocate(std::size_t pos)    // list_pool member function
{
    std::size_t n = next(pos);
    prev(n) = 0;                    // remove backlink
    next(pos) = free_list;          // append current freelist
    free_list = pos;                // include deallocated node in freelist
    return n;                       // return next node (or zero, if none)
}
```

Creating a List Pool: deallocate

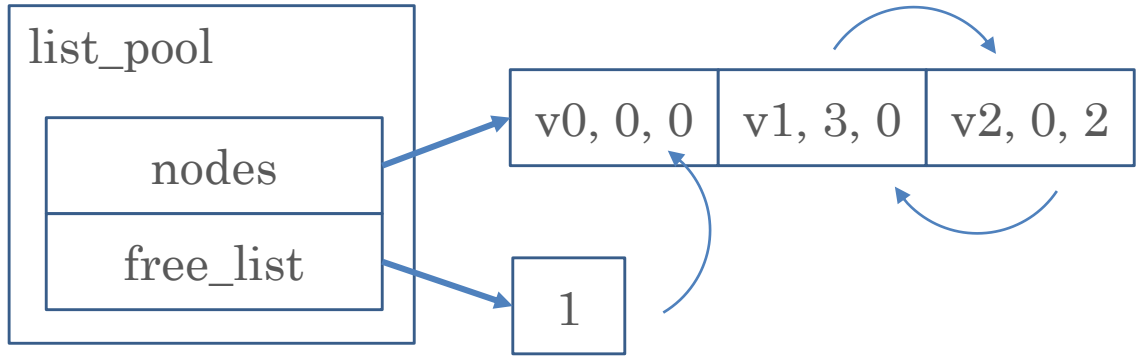



```
std::size_t
deallocate(std::size_t x) {
    std::size_t n = next(x);
    prev(n) = 0;
    next(x) = free_list;
    free_list = x;
    return n;
}
```

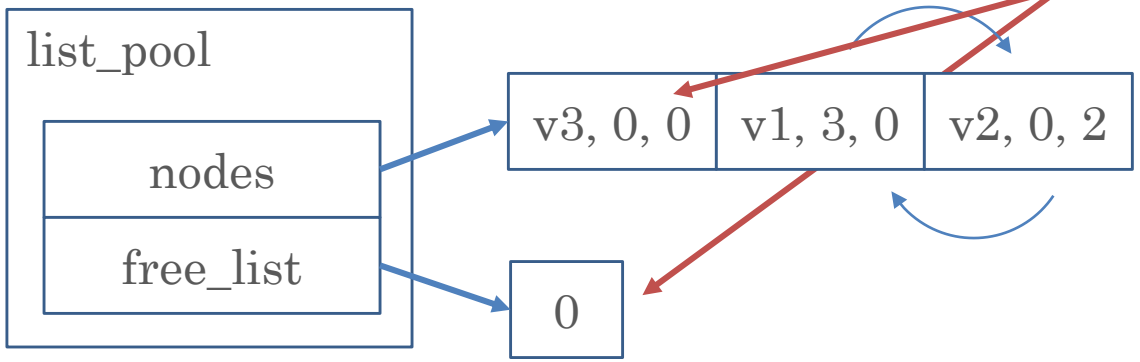
Creating a List Pool: `allocate`

```
std::size_t allocate(T const& val) {      // list_pool member function
    std::size_t node;
    if (free_list == 0) {
        pool.push_back(list_node<T>());   // create new node
        node = pool.size();
    }
    else {
        node = free_list;                 // get node from free list
        free_list = next(free_list);
    }

    value(node) = val;
    next(node) = 0;
    prev(node) = 0;
    return node;                          // new node index
}
```

Creating a List Pool: `allocate`




```
allocate(v3); // returns 1
```

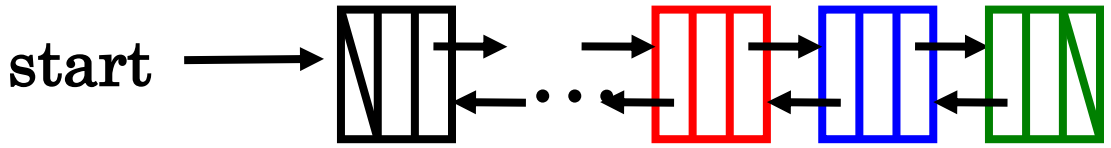



```
std::size_t
allocate(T const& val) {
    std::size_t node = free_list;
    free_list = next(free_list);

    value(node) = val; // v3
    next(node) = 0;
    prev(node) = 0;

    return node;
}
```

Creating a List Pool: `link`

- Linking a new node into list:

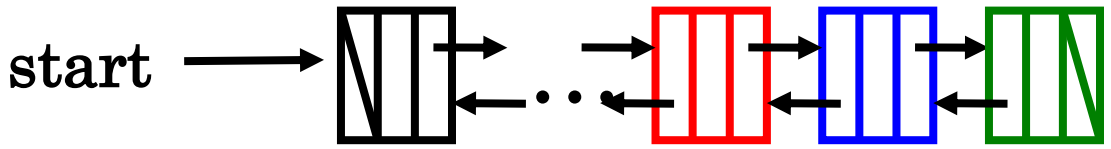
```
std::size_t link(std::size_t pos, std::size_t new_node) {
    prev(new_node) = pos;
    if (pos != 0) {
        next(new_node) = next(pos);
        if (next(pos) != 0)
            prev(next(pos)) = new_node;
        next(pos) = new_node;
    }
    return new_node;
}
```



Creating a List Pool: unlink

- Unlinking an existing node from list:

```
std::size_t unlink(std::size_t pos) {

    if (prev(pos) != 0)
        next(prev(pos)) = next(pos);
    if (next(pos) != 0)
        prev(next(pos)) = prev(pos);

    return next(pos);
}
```

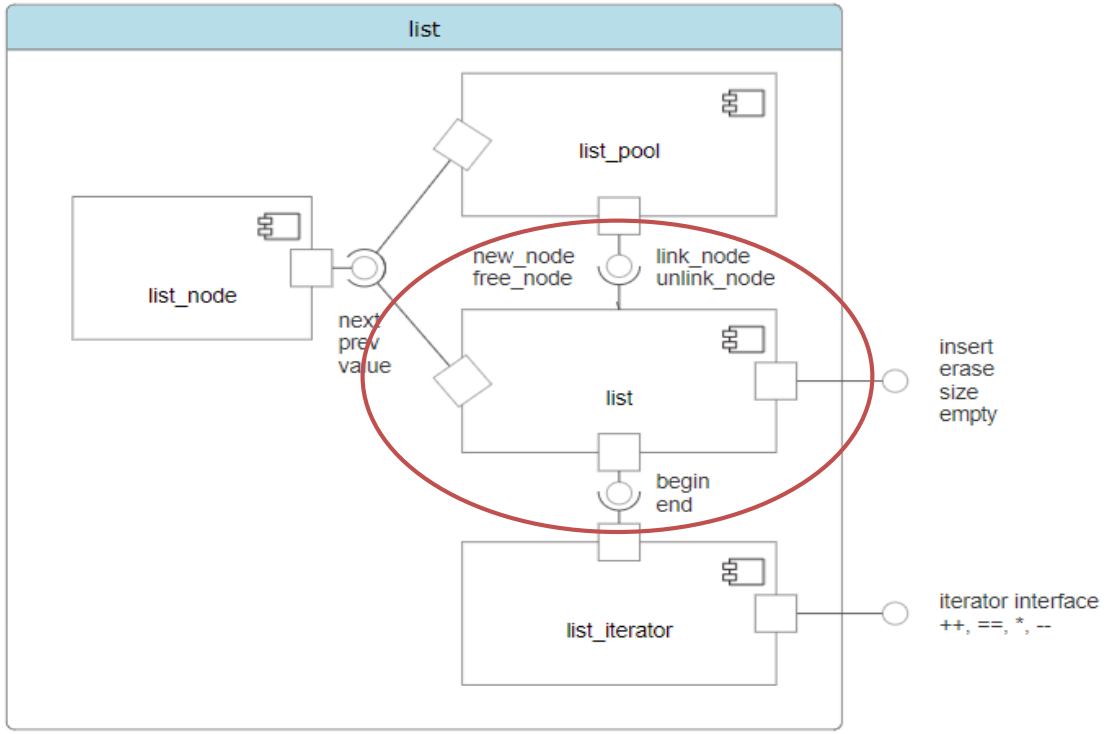


Creating a List Pool: `insert` and `erase`

```
std::size_t insert(T const& val, std::size_t pos)
{
    return link(pos, allocate(val));
}

std::size_t erase(std::size_t pos)
{
    std::size_t node = unlink(pos);
    deallocate(pos);
    return node;
}
```

Creating the `list` Type

- With the existing facilities, implementing a list interface is straightforward

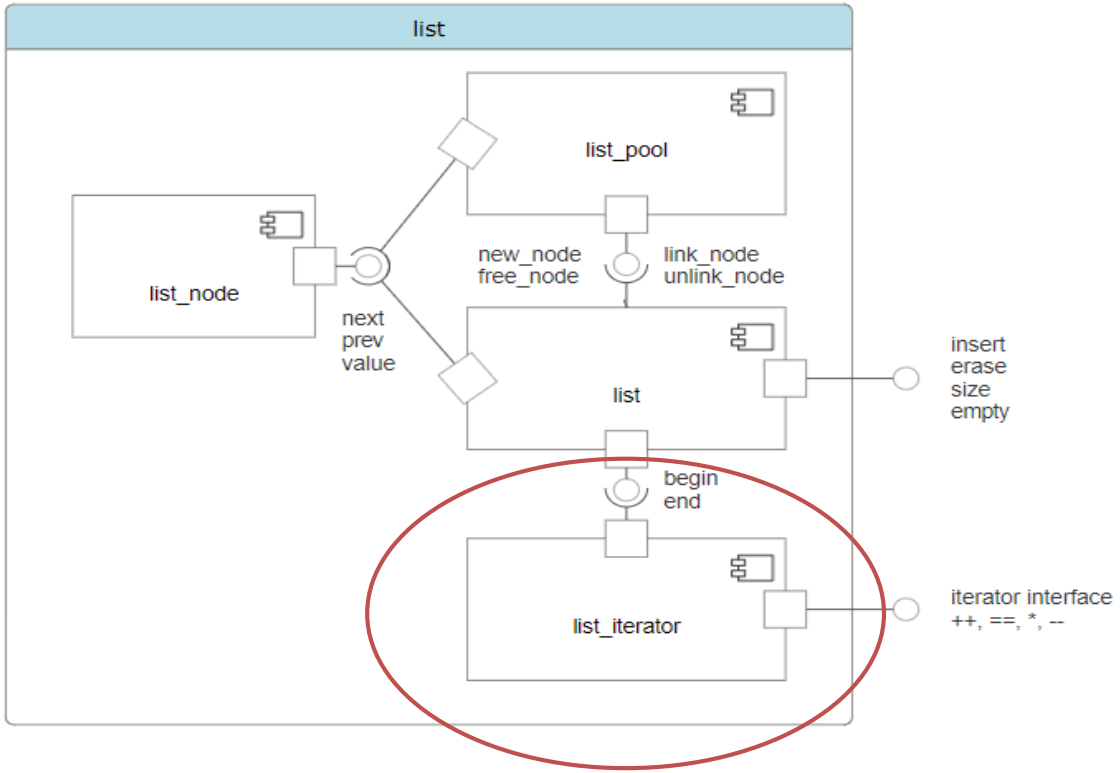


Creating the `list` Type

```
template <std::semiregular T>
class list {
    list_pool<T> pool;          // list_node's are stored here
    std::size_t front = 0;      // index of first list node
    std::size_t back = 0;       // index of last list node
public:
    void push_back(T const& val)  {
        back = pool.insert(val, back);
        if (front == 0)
            front = back;
    }

    list_iterator<T> erase(list_iterator<T> it);
    list_iterator<T> begin();
    list_iterator<T> end();
};
```

Creating the `list_iterator` Type

- With the existing facilities, implementing a list interface is straightforward



Iterators

- Iterators are special types
  - Identify a container and an element in the container
  - Let us examine the value stored in that element
  - Provide operations for moving between elements in the container
  - Restrict the available operations in ways that correspond to what the container can handle efficiently

Iterator Types

- Every standard container, such as `std::vector`, defines two associated iterator types:

```
container_type::iterator
container_type::const_iterator
```

  - Where container_type is the container (`std::vector<student_info>`)
  - Use `iterator` to modify the element, `const_iterator` otherwise (read only access)
- Note, that we don‘t actually see the actual type, we just know what we can do with it.
  - Abstraction is selective ignorance!

Iterator Types

- Every `container_type::iterator` is convertible to the corresponding `container_type::const_iterator`
  - students.begin() returns an iterator, but we assign it to a const_iterator
- Opposite is not true! Why?

Iterator Operations

- Containers do not only expose their (specific) iterator types, but also actual iterators:

```
students.begin(), students.end()
```

  - begin(): 'points' to the first element
  - end(): 'points' to the element after the last one

- Iterators can be *compared*:

```
iter != students.end()
```

  - Tests, whether both iterators refer to the same element

- Iterators can be *incremented*:

```
++iter
```

  - Make the iterator 'point' (refer) to the next element

Iterator Operations

- Iterators can be *dereferenced*:

```
*iter
```

  - Evaluates to the element the iterator refers to

- In order to access a member of the element the iterator refers to, we write:

```
(*iter).name
```

  - (why not: `*iter.name` ?)

- Syntactic sugar, 100% equivalent:

```
iter->name
```

Iterator Operations

- Some iterators can get a number added

```
students.erase(students.begin() + i);
```

  - Overloaded operator+, makes the iterator refer to the ‘i’ –s element after begin
  - Equivalent to invoking ++ ‘i’ times
  - Defined only for iterators from *random access* containers
    - `std::vector`, `std::string` are random access (indexing is possible)
    - Will result in compilation error for sequential (non-random access) containers

Creating the `list_iterator` Type

- With that knowledge, let's implement `list_iterator<T>`

```
template <std::semiregular T>
class list_iterator {
    list_pool<T>& pool;
    std::size_t pos;

public:
    list_iterator(list_pool<T>& p, std::size_t curr) : pool(p), pos(curr)  {}

    T& operator*() { return pool.value(pos); }              // dereference

    list_iterator& operator++() { pos = pool.next(pos); return *this; }    // prefix++
    list_iterator operator++(int) {                                        // postfix++
        auto prev_pos = pos;
        pos = pool.next(pos);
        return list_iterator(pool, prev_pos); }
};
```

Creating the `list_iterator` Type

- With that knowledge, let's implement `list_iterator<T>`:

```
friend bool operator==(
    list_iterator const& lhs, list_iterator const& rhs)
{
    return lhs.pos == rhs.pos;
}
friend bool operator!=(
    list_iterator const& lhs, list_iterator const& rhs)
{
    return !(lhs == rhs);
}
```

Using Our List Type

```
// version 5: using our lists
list<student_info> extract_fails(list<student_info>& students)
{
    list<student_info> fail;
    auto iter = students.begin();
    while (iter != students.end()) {
        if (fail_grade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);    // watch out!
        } else
            ++iter;
    }
    return fail;
}
```

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP