Michelle Novenda Hartono 莊晴雯
108062281
Department of Computer Science

In this Mini Project 2, we are required to implement the C++ Standard Template Library (STL) by ourselves, based on what we have learned in the Introduction to Programming 2 Class.

Function Implementation

ITERATOR

I2P2_iterator.h
This file contains mostly the functions that we need to implement ourselves. It has the base class of all the container-specialized iterators, which is struct iterator_impl_base. Then we have the vector_iterator, list_iterator, const_iterator, and iterator class. Inside the classes mentioned above, they each have the public functions that we need to implement, which is inherited from the iterator_impl_base. Since iterator_impl_base is a base class, we need to implement all the functions inside. We also need to declare what our vector_iterator and list_iterator stores in each class, and here I declared pointer vct and Node* lst for vector_iterator and list_iterator respectively. Then I also added the struct Node to store the Node* prev, Node* next, data, and store their initial values. In this .h file, I also added some new functions which I implement directly in the file, namely clone(), to make the vector iterator or list iterator, and node_ref() to return the position of lst.

I2P2_iterator.cpp
In this file, we implemented all the functions declared in the iterator.h file for vector_iterator, list_iterator, const_iterator, and iterator.

vector_iterator

- vector_iterator();
  Constructor, sets initial value of vct to nullptr;

- vector_iterator(pointer ptr) : vct(ptr) {}
  A constructor with parameter that sets the initial value of vct to ptr.

- iterator_impl_base &operator++();
  Moves the pointer vct to the next position. Hence, makes it vct = vct+1, ot vct++, and return *this.

- iterator_impl_base &operator--()
  Moves the pointer vct to the previous position. Hence, makes it vct = vct-1, ot vct--, and return *this.

- iterator_impl_base &operator+=(difference_type offset)
  Moves pointer vct forward with the distance that vct needs to move which is offset. Therefore, vct += offset and return *this.

- iterator_impl_base &operator-=(difference_type offset)
  Moves pointer vct backward with the distance that vct needs to move which is offset. Therefore, vct -= offset and return *this.

- bool operator==(const iterator_impl_base &rhs) const
  If the address of rhs is equal to the address of vct, return true. I used operator->() to get the address of rhs, not just the value.

- bool operator!=(const iterator_impl_base &rhs) const
  If the address of rhs is not equal to the address of vct, return true. I used operator->() to get the address of rhs, not just the value.

- bool operator<(const iterator_impl_base &rhs) const
  If the address of vct is smaller than the address of rhs, return true.

- bool operator>(const iterator_impl_base &rhs) const
  If the address of vct is greater than the address of rhs, return true.

- bool operator<=(const iterator_impl_base &rhs) const
  If the address of vct is smaller than or equal to the address of rhs, return true.

- bool operator>=(const iterator_impl_base &rhs) const
  If the address of vct is greater than or equal to the address of rhs, return true.

- difference_type operator-(const iterator_impl_base &rhs) const
  This function returns the difference of the address of vct and rhs.

- pointer operator->() const
  Returns the address of vct.

- reference operator*() const
  Returns the value of vct.

- reference operator[](difference_type offset) const
  Returns the value of vct.

list_iterator

- list_iterator()
  Constructor, sets the initial value of lst as nullptr.

- list_iterator(Node* head) : lst(head) {}
  A constructor with a parameter that sets the initial value of lst to head.

- iterator_impl_base &operator++()
  Moves one node to the next node, which implies lst = lst->next, then return *this.

- iterator_impl_base &operator--()
  Moves one node to the previous node, which implies lst = lst->prev, then return *this.

- iterator_impl_base &operator+=(difference_type offset)
  Moves one node to the next node in the distance of offset, therefore it repeats the action lst = lst->next for the amount of offset. Here, I used the for loop to implement the function.

- iterator_impl_base &operator-=(difference_type offset)
  Moves one node to the previous node in the distance of offset, therefore it repeats the action lst = lst->prev for the amount of offset. Here, I used the for loop to implement the function.

- bool operator==(const iterator_impl_base &rhs) const
  This function returns true if lst is equal to the address of rhs.

- bool operator!=(const iterator_impl_base &rhs) const
  This function returns true if lst is not equal to the address of rhs.

- bool operator<(const iterator_impl_base &rhs) const
  This function returns true if lst is smaller to the address of rhs. It stores lst in it, and ndp is the position of the node. If it is equal to ndp, it returns false immediately because the operator is only < and not <=. Then while it != nullptr, there is an if condition inside, where if it is equal to ndp, it returns true. This is because it has found the rhs. In other cases, this function returns false.

- bool operator>(const iterator_impl_base &rhs) const
  This function returns true if lst is greater to the address of rhs. It stores lst in it, and ndp is the position of the node. If it is equal to ndp, it returns false immediately because the operator is only < and not <=. Then while it != nullptr, there is an if condition inside, where if it is equal to ndp, it returns true. This is because it has found the rhs. In other cases, this function returns false.

- bool operator<=(const iterator_impl_base &rhs) const
  This function returns true if lst is smaller or equal to the address of rhs. The method of implementing this function is the same as implementing bool operator<(const iterator_impl_base &rhs) const, it is just if it is equal to ndp, the function returns true, which is inside the while loop, because the operator here is <= and it has the = sign.

- bool operator>=(const iterator_impl_base &rhs) const
  This function returns true if lst is greater or equal to the address of rhs. The method of implementing this function is the same as implementing bool operator>(const iterator_impl_base &rhs) const, it is just if it is equal to ndp, the function returns true, which is inside the while loop, because the operator here is >= and it has the = sign.

- difference_type operator-(const iterator_impl_base &rhs) const

In this function I used flag, front, back, the nodes mov_front, mov_back, and targetNode. The initial value of flag is 1, and if flag is 1, the operation inside the first if function will run. Here, while mov_front is not nullptr and if mov_front is equal to targetNode, it will break when it has found the right position, else, mov_front will keep moving next and front will increment. After this is completed, it will return front. Else, while mov_vack is not equal to nullptr, mov_back will keep moving previously until it has found the targetNode, then it will set flag to 0 so that it will not enter the if block above. Then lastly it will return back.

- pointer operator->() const
  Returns the address of lst->data.

- reference operator*() const
  Returns the value of lst->data.

- reference operator[](difference_type offset) const
  Returns the value of lst->data. If offset is smaller than 0, ptr will keep moving previously while offset increments until it reaches 0. Else, ptr will keep moving next while offset decrements until it reaches 0. In the end, this function will return ptr->data.

const_iterator and iterator both call these functions based on the type that is passed to them. If the type list is passed, then these functions will act as how the list works, same goes to vector.

- ~const_iterator()
  Destructor, deletes p_ and sets it to nullptr.

- const_iterator()
  Constructor, sets the initial value of p_ to nullptr.

- const_iterator(const const_iterator &rhs)
  To get the position so that functions can point there.

- const_iterator(iterator_impl_base *p)
  To get the position so that functions can point there.

- const_iterator &operator=(const const_iterator &rhs)
  Copy constructor, copy something on the rhs that is the same type with it and move it to the lhs.

- const_iterator &operator++()
  Adds the address of p_ to 1, then return *this. Can be implemented by using ->operator++().

- const_iterator operator++(int)

Copy the address *this to a variable named copy first, then increments the address of p_ by using ->operator++(), then returns copy.

- const_iterator &operator--()
  Subtracts the address of p_ to 1, then return *this. Can be implemented by using ->operator--().

- const_iterator operator--(int)
  Copy the address *this to a variable named copy first, then decrements the address of p_ by using ->operator--(), then returns copy.

- const_iterator &operator+=(difference_type offset)
  Adds the address of p_ to offset, then return *this. Can be implemented by using ->operator+=(offset).

- const_iterator operator+(difference_type offset) const
  Copy the address *this to a variable named copy first, then adds the address of p_ with offset by using ->operator+=(offset), then returns copy.

- const_iterator &operator-=(difference_type offset)
  Subtracts the address of p_ to offset, then return *this. Can be implemented by using ->operator-=(offset).

- const_iterator operator-(difference_type offset) const
  Copy the address *this to a variable named copy first, then subtracts the address of p_ with offset by using ->operator-=(offset), then returns copy.

- difference_type operator-(const const_iterator &rhs) const
  p_ calls the operator - of each type, which can be either list or vector, depends on what is called.

- pointer operator->() const
  Returns the address of it points.

- reference operator*() const
  Returns the value of it points.

- reference operator[](difference_type offset) const
  Returns the value.

- bool operator==(const const_iterator &rhs) const
  Returns true if the address of p_ is equal to the address of rhs.p_;

- bool operator!=(const const_iterator &rhs) const
  Returns true if the address of p_ is not equal to the address of rhs.p_;

- bool operator<(const const_iterator &rhs) const
  Returns true if the address of p_ is smaller to the address of rhs.p_;

- bool operator>(const const_iterator &rhs) const
  Returns true if the address of p_ is greater to the address of rhs.p_;

- bool operator<=(const const_iterator &rhs) const
  Returns true if the address of p_ is smaller or equal to the address of rhs.p_;

- bool operator>=(const const_iterator &rhs) const
  Returns true if the address of p_ is greater or equal to the address of rhs.p_;

Iterator does not have some functions that const_iterator has. Other than that, the way of implementing these two functions are the same.

VECTOR

I2P2_Vector.h
Here, we need to declare necessary members for the Vector container, in which I declared pointer begin_, pointer end_, and pointer last_. begin_ is the first element of the vector, while last_ is the last element of the vector, end_ is the pointer to the last element in the vector.

I2P2_Vector.cpp

- ~Vector()
  Destructor, destructs the vector one by one using for loop and deletes begin_.

- Vector()
  Constructor, set all pointer's values which are begin_, end_, and last_ as nullptr.

- Vector(const Vector &rhs)
  Constructor with parameter, set all pointer's values which are begin_, end_, and last_ as nullptr. Then reserves the capacity as big as rhs.size(), then inserts rhs's value to this->begin() using for loop.

- Vector &operator=(const Vector &rhs)
  Copy constructor with parameter, first if this is equal to &rhs, it will return *this. Set all pointer's values which are begin_, end_, and last_ as nullptr. Then reserves the capacity as big as rhs.size(), then inserts rhs's value to this->begin() using for loop. In the end it returns *this.

- iterator begin()
  Returns an iterator pointing to the first element in the vector.

- const_iterator begin() const
  Returns a const_iterator pointing to the first element in the vector.

- iterator end()
  Returns an iterator pointing to the theoretical element that follows the last element in the vector.

- const_iterator end() const
  Returns a const_iterator pointing to the theoretical element that follows the last element in the vector.

- reference front()
  Returns a reference to the first element in the vector.

- const_reference front() const
  Returns a const_reference to the first element in the vector.

- reference back()
  Returns a reference to the last element in the vector.

- const_reference back() const
  Returns a const_reference to the last element in the vector.

- reference operator[](size_type pos)
  Returns the value of begin_ at the position pos.

- const_reference operator[](size_type pos) const
  Returns the value of begin_ at the position pos.

- size_type capacity() const
  Returns the vector's capacity that can be obtained by subtracting end_ with begin_;

- size_type size() const
  Return the size of the vector container or the number of elements in the vector container. The size of the vector can be obtained by subtracting last_ to begin_.

- void clear()
  Remove all the elements of the vector container. I first store the size of the vector to old_size, then destruct it one by one, also decrementing last_.

- bool empty() const
  If last_ is equal to begin_, then the list is empty.

- void erase(const_iterator pos)
  Removes an element from a particular position, in this case, pos. I first get the position of the element that is going to be deleted by subtracting it to the first element of the vector, since the position is counted from the first element of the vector, then store it in pst. Then the function erase can only be done if the position is greater or equal to 0 and is smaller than the size of the vector. Entering the for loop, I start moving values of the element after the position to be deleted until the last element is

moved. Lastly, I destruct the last element of the vector and decrement the value of last_.

- void erase(const_iterator begin, const_iterator end)
  Remove elements within a range, here, from begin to end, and can only be done if begin is not equal to end. I first get the position of the element that is going to be deleted by subtracting both begin and end to the first element of the vector, since the position is counted from the first element of the vector, then storing them to beg_pos and end_pos respectively. I also get the difference for the position of begin and end parameter by subtracting end_pos to beg_pos. If begin and end is both the beginning of the vector and the end of the vector, it immediately clears the entire vector. Else if beg_pos is greater or equal to 0, we enter the for loop, and I start moving values of beg_pos+diff_pos to beg_pos, then move to both of the next elements until it reaches right before vec_size-diff_pos. After the elements are moved, I destruct the remaining elements using a for loop and decrement the value of last_ every time an element is destructed.

- void insert(const_iterator pos, size_type count, const_reference val)
  This function inserts count number of elements with the value val and inserts it in position pos. It can only run if count is greater or equal to 0, and if the position is greater than the size, it immediately returns. Since more elements are added to the vector, I first need to do reserve in order for the capacity to be able to hold in more elements. After that, the value of last_ needs to be added to the count amount. Then there are 3 for loops, first one is to make new elements and insert the values to 0. Second for loop is to move the values to the desired positions, hence making room for the new values to be inserted. The last for loop is to add the values to be inserted in the desired position.

- void insert(const_iterator pos, const_iterator begin, const_iterator end)
  This function inserts elements within the range begin until end in position pos. It can only run if begin is not equal to end. Since more elements are added to the vector, I first need to do reserve in order for the capacity to be able to hold in more elements. After that, the value of last_ needs to be added to the amount of values to be inserted, in this case, diff_pos. Then I store the new values to be inserted in store. There are 3 for loops, the first one is to make new elements and insert the values to 0. Second for loop is to move the values to the desired positions, hence making room for the new values to be inserted. The last for loop is to add the values from store to be inserted in the desired position.

- void pop_back()
  pop_back() removes the last element of the vector. To implement this, I simply call the function erase to erase the last element of the vector.

- void pop_front()
  pop_front() removes the first element of the vector. To implement this, I simply call the function erase to erase the first element of the vector.

- void push_back(const_reference val)
  This function inserts an element val to the end of the vector. To do this, I can simply call the function insert and pass the arguments end() to indicate the position of where the element needs to be inserted, 1 as the number of element to be inserted, and val as the value of the element to be inserted.

- void push_front(const_reference val)
  This function inserts an element val to the beginning of the vector. To do this, I can simply call the function insert and pass the arguments begin() to indicate the position of where the element needs to be inserted, 1 as the number of element to be inserted, and val as the value of the element to be inserted.

- void reserve(size_type new_capacity)
  Requests that the vector capacity be at least enough to contain new_capacity elements.

- void shrink_to_fit()
  Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity. The way of implementing this function is almost the same as implementing the function reserve.

LIST

I2P2_List.h
Here, we need to declare necessary members for the List container, in which I declared Node* _head, Node* _tail, and size_type size as the list's head, tail, and size respectively since the list is a doubly linked list with two dummy values.

I2P2_List.cpp
- ~List()
  Desctructor, clears everything on the list and delete the remaining _head and _tail, and you can choose to set them to nullptr or not.

- List()
  Constructor, make _head and _tail as a new node with the value 0. Then since this is an empty list, _head->next must link to _tail, and both _tail->next and _head->prev is nullptr, and _tail->prev links to _head. Since this is an empty list, the _size is 0.

- List(const List &rhs)
  Copy constructor, does the same thing as a constructor and here I declared now as the next of rhs._head. Then while now is not equal to rhs._tail, which means this is not an empty list, it pushes back the data of now and now keeps moving next until it does not satisfy the loop condition anymore.

- List &operator=(const List &rhs)
  Does the same thing as copy constructor, just returns *this.

- iterator begin()
  begin() returns an iterator pointing to the first element of the list. I declared the value to be returned as head_ptr. If _head->next is equal to _tail, head_ptr is equal to _tail because if it is an empty list, iterator end and begin is the same. Else, head_ptr is equal to _head->next. Then after getting head_ptr, we change the type of head_ptr to type iterator for it to be able to be returned.

- const_iterator begin() const
  The same implementation as iterator begin(), just different in the type.

- iterator end()
  end() returns an iterator pointing to the last element of the list. Tail_ptr is equal to _tail and it is returned.

- const_iterator end() const
  Same implementation as iterator end(), just different in the type.

- reference front()
  front() returns the value of the first element in the list. As long as the list is not empty, it can keep going next so it returns the data of _head->next, else, it only returns _head->data.

- const_reference front() const
  front() returns the value of the first element in the list. As long as the list is not empty, it can keep going next so it returns the data of _head->next, else, it only returns _head->data.

- reference back()
  back() returns the value of the last element in the list. As long as the list is not empty, it can keep going previously so it returns the data of_tail->prev, else, it only returns_tail>data.

- const_reference back() const
  back() returns the value of the last element in the list. As long as the list is not empty, it can keep going previously so it returns the data of_tail->prev, else, it only returns_tail>data.

- size_type size() const
  size() returns the number of element in the list, so we only need to return _size.

- void clear()
  clear() clears the entire list. It removes all the elements in the list container and it means that the size is back to 0. To clear, I need curNode to be set as _head->next, and while it has not reached _tail, I store the curNode to temp so I can keep deleting while curNode keeps moving forward while being deleted until it reaches _tail.

Following the properties of an empty linked list, _head->next must point to _tail, _tail->prev must point to _head, and the size must be to 0.

- bool empty() const
  This function tells us whether a list is empty or not. If the list is empty, which implies that _size == 0, the function returns true, else is _size != 0, the function returns false.

- void erase(const_iterator pos)
  Removes a single element from the list in the position pos. If the list is not empty, I set curNode as the position of pos. It then breaks the link of curNode from the list and re-link the list without curNode. After the action is done, it deletes curNode and reduces the _size by 1.

- void erase(const_iterator begin, const_iterator end)
  Removes a range of elements from begin to end from the list. I get the position of begin and end using the function node_ref() and declares begin_ and end_ to store both position respectively. temp here is to store the previous value of the begin_ node that is to be deleted. Then while begin_ is not equal to end_, I set a newnode to store begin_ so that begin_ can continue moving next while it is being deleted one by one and thus the _size reduces by one per deleted node. I then still needs to re-link temp->next to end_ and end_->prev to temp for the link to be linked.

- void insert(const_iterator pos, size_type count, const_reference val)
  Inserts new elements in the list before the element at position pos, for count amount of node and with the value val. So while count is greater than 0, the position is set to targetNode and I declared new_head and new_tail for storing the elements that need to be inserted. This action is supported by the for loop and after it is done storing all the new elements, we need to link the new_head and new_tail to the original list.

- void insert(const_iterator pos, const_iterator begin, const_iterator end)
  Inserts new elements in the list before the element at position pos, and the value to be inserted ranges from begin to end. This function only runs when begin is not equal to end, and I need to find the position of pos, begin, and end, then store them in targetNode, beginNode, and endNode respectively. Similar to the insert above, I need to declare new_head and new_tail and store the new values to be added here, thus adding the size. In here, the for loop is used to link the list containing the new values to the original link.

- void pop_back()
  Removes the last element of the list and reduces the size of the list by 1. The function runs if the list is not empty. Since it removes the last element, we deal with _tail.

- void pop_front()
  Removes the first element of the list and reduces the size of the list by 1. The function runs if the list is not empty. Since it removes the first element, we deal with_head.

- void push_back(const_reference val)
Adds a new element val at the end of the list. I first declared a newnode with value 0, then set the value to val. This function deals with _tail since an element is added at the end of the list. I need to connect _tail->prev->next to newnode, then newnode->prev to _tail->prev, newnode->next to _tail and lastly _tail->prev to newnode in order for the list to be linked. Do not forget that we need to increase the size by one.

- void push_front(const_reference val)
Adds a new element val at the beginning of the list. I first declared a newnode with value 0, then set the value to val. This function deals with _head since an element is added at the beginning of the list. I need to connect _head->next->prev to newnode, then newnode->next to_head->next, newnode->next to _head and lastly _head->next to newnode in order for the list to be linked. Do not forget that we need to increase the size by one.

Time Complexity

Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	Back: O(1) or O(n) Other: O(n)	O(1)	Back: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	No
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes

- Insertion: Vector will be slower since when inserting an element to a vector, we need to move the other elements to make room for the elements to be inserted. As for list, we only need to link and re-link the list to insert new nodes.

- Access: Access in vector is constant time, in which no matter how many elements we iterate through, it takes the same time, example: begin_[2]. That is why, it only takes O(1). As for linked list, access in here is in linear time, in which as number of elements we have to search rises, so does the time, because we need to iterate one by one, we cannot directly jump to the item that we want to access.

- Erase: Vector takes more time in erasing because after erasing the elements, we still need to move other elements in the vector to the correct place. As for linked list, we can just detach the link, delete the detached link and then re-link the linked list again.

- Find: Both vector and linked list took the same amount of time to find an element because we don't know where the element we want to find is located, so we need to iterate one by one to find the element/s desired.

Hierarchy Relationship

```cpp
struct dynamic_size_container : container_base {
  virtual iterator begin() = 0;
  virtual const_iterator begin() const = 0;
  virtual iterator end() = 0;
  virtual const_iterator end() const = 0;
  virtual void clear() = 0;
  virtual void erase(const_iterator pos) = 0;
  virtual void erase(const_iterator begin, const_iterator end) = 0;
  // The following need only be defined for vector
  virtual void reserve(size_type new_capacity) {}
  virtual void shrink_to_fit() {}
  virtual size_type capacity() const { return size(); }
};

struct ordered_container : dynamic_size_container {
  virtual reference back() = 0;
  virtual const_reference back() const = 0;
  virtual reference front() = 0;
  virtual const_reference front() const = 0;
  virtual void insert(const_iterator pos, size_type count, const_reference val) = 0;
  virtual void insert(const_iterator pos, const_iterator begin, const_iterator end) = 0;
  virtual void pop_back() = 0;
  virtual void pop_front() = 0;
  virtual void push_back(const_reference val) = 0;
  virtual void push_front(const_reference val) = 0;
};

struct randomaccess_container : ordered_container {
  virtual reference operator[](size_type pos) = 0;
  virtual const_reference operator[](size_type pos) const = 0;
};
```

```cpp
class List : public ordered_container {
 protected:
  Node* _head;
  Node* _tail;
  size_type _size;
  // you may want to declare necessary members for your List container here

  /* The following are standard methods from the STL
   * If you are not sure what they do
   * look them up and implement your own version */
 public:
  ~List();
  List();
  List(const List &rhs);
  List &operator=(const List &rhs);
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;
  size_type size() const;
  void clear();
  bool empty() const;
  void erase(const_iterator pos);
  void erase(const_iterator begin, const_iterator end);
  void insert(const_iterator pos, size_type count, const_reference val);
  void insert(const_iterator pos, const_iterator begin, const_iterator end);
  void pop_back();
  void pop_front();
  void push_back(const_reference val);
  void push_front(const_reference val);
};
```

```cpp
class Vector : public randomaccess_container {
 private:
    pointer begin_;
    pointer end_;
    pointer last_;


  /* The following are standard methods from the STL
   * If you are not sure what they do
   * look them up and implement your own version */
 public:
  ~Vector();
  Vector();
  Vector(const Vector &rhs);
  Vector &operator=(const Vector &rhs);
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;
  reference operator[](size_type pos);
  const_reference operator[](size_type pos) const;
  size_type capacity() const;
  size_type size() const;
  void clear();
  bool empty() const;
  void erase(const_iterator pos);
  void erase(const_iterator begin, const_iterator end);
  void insert(const_iterator pos, size_type count, const_reference val);
  void insert(const_iterator pos, const_iterator begin, const_iterator end);
  void pop_back();
  void pop_front();
  void push_back(const_reference val);
  void push_front(const_reference val);
  void reserve(size_type new_capacity);
  void shrink_to_fit();
};
```

As you can see, the class List inherits the ordered_container, you can refer to the line class List : public ordered_container
Hence, the class List can use all the functions inside the order container. While the class Vector, it inherits the randomaccess_container, as seen on the line class Vector : public randomaccess_container
Therefore, the class Vector can use all the functions inside the randomaccess_container. Inside the randomaccess_container, there are two more functions namely reference operator[](size_type pos) and const_reference operator[](size_type pos) const, which can be used by Vector.

```cpp
struct iterator_impl_base
{
  virtual reference operator*() const = 0;
  virtual reference operator[](difference_type offset) const = 0;
  virtual pointer operator->() const = 0;
  virtual difference_type operator-(const iterator_impl_base &rhs) const = 0;
  virtual iterator_impl_base &operator++() = 0;
  virtual iterator_impl_base &operator--() = 0;
  virtual iterator_impl_base &operator+=(difference_type offset) = 0;
  virtual iterator_impl_base &operator-=(difference_type offset) = 0;
  virtual bool operator==(const iterator_impl_base &rhs) const = 0;
  virtual bool operator!=(const iterator_impl_base &rhs) const = 0;
  virtual bool operator<(const iterator_impl_base &rhs) const = 0;
  virtual bool operator>(const iterator_impl_base &rhs) const = 0;
  virtual bool operator<=(const iterator_impl_base &rhs) const = 0;
  virtual bool operator>=(const iterator_impl_base &rhs) const = 0;
  virtual iterator_impl_base* clone() const = 0;
  virtual Node* node_ref() const = 0;
};
```

```cpp
class vector_iterator : public iterator_impl_base {
 protected:
  pointer vct;

 public:
  vector_iterator();
  vector_iterator(pointer ptr) : vct(ptr) {}
  iterator_impl_base &operator++();
  iterator_impl_base &operator--();
  iterator_impl_base &operator+=(difference_type offset);
  iterator_impl_base &operator-=(difference_type offset);
  bool operator==(const iterator_impl_base &rhs) const;
  bool operator!=(const iterator_impl_base &rhs) const;
  bool operator<(const iterator_impl_base &rhs) const;
  bool operator>(const iterator_impl_base &rhs) const;
  bool operator<=(const iterator_impl_base &rhs) const;
  bool operator>=(const iterator_impl_base &rhs) const;
  difference_type operator-(const iterator_impl_base &rhs) const;
  pointer operator->() const;
  reference operator*() const;
  reference operator[](difference_type offset) const;
  iterator_impl_base* clone() const
  {
    return new vector_iterator(vct);
  }
  Node* node_ref() const
  {
    return nullptr;
  };
};
```

```cpp
class list_iterator : public iterator_impl_base {
 protected:
  Node* lst;

 public:
  list_iterator();
  list_iterator(Node* head) : lst(head) {}
  iterator_impl_base &operator++();
  iterator_impl_base &operator--();
  iterator_impl_base &operator+=(difference_type offset);
  iterator_impl_base &operator-=(difference_type offset);
  bool operator==(const iterator_impl_base &rhs) const;
  bool operator!=(const iterator_impl_base &rhs) const;
  bool operator<(const iterator_impl_base &rhs) const;
  bool operator>(const iterator_impl_base &rhs) const;
  bool operator<=(const iterator_impl_base &rhs) const;
  bool operator>=(const iterator_impl_base &rhs) const;
  difference_type operator-(const iterator_impl_base &rhs) const;
  pointer operator->() const;
  reference operator*() const;
  reference operator[](difference_type offset) const;
  iterator_impl_base* clone() const
  {
    return new list_iterator(lst);
  }
  Node* node_ref() const
  {
    return lst;
  }
};
```

Here, the class vector_iterator and list_iterator both inherit the same base class which is iterator_impl_base, and is also an abstract class. An abstract class needs all the functions to be implemented. Therefore, both vector_iterator and list_iterator class must implement all the functions declared inside the iterator_impls_base class.

```cpp
class const_iterator {
 public:
  using difference_type = I2P2::difference_type;
  using value_type = I2P2::value_type;
  using pointer = I2P2::const_pointer;
  using reference = I2P2::const_reference;
  using iterator_category = std::random_access_iterator_tag;

 protected:
  iterator_impl_base *p_;

 public:
  ~const_iterator();
  const_iterator();
  const_iterator(const const_iterator &rhs);
  const_iterator(iterator_impl_base *p);
  const_iterator &operator=(const const_iterator &rhs);
  const_iterator &operator++();
  const_iterator operator++(int);
  const_iterator &operator--();
  const_iterator operator--(int);
  const_iterator &operator+=(difference_type offset);
  const_iterator operator+(difference_type offset) const;
  const_iterator &operator-=(difference_type offset);
  const_iterator operator-(difference_type offset) const;
  difference_type operator-(const const_iterator &rhs) const;
  pointer operator->() const;
  reference operator*() const;
  reference operator[](difference_type offset) const;
  bool operator==(const const_iterator &rhs) const;
  bool operator!=(const const_iterator &rhs) const;
  bool operator<(const const_iterator &rhs) const;
  bool operator>(const const_iterator &rhs) const;
  bool operator<=(const const_iterator &rhs) const;
  bool operator>=(const const_iterator &rhs) const;
  /* This class holds an iterator_impl_base
   * and you may want to have some ways to
   * invoke a container-specialized method from here
   * for insert/erase methods (look at their parameters if you are not sure) */
  Node* node_ref() const
  {
    return p_->node_ref();
  }
};
```

```cpp
class iterator : public const_iterator {
 public:
  using difference_type = I2P2::difference_type;
  using value_type = I2P2::value_type;
  using pointer = I2P2::pointer;
  using reference = I2P2::reference;
  using iterator_category = std::random_access_iterator_tag;

 public:
  iterator();
  iterator(iterator_impl_base *p);
  iterator(const iterator &rhs);
  iterator &operator++();
  iterator operator++(int);
  iterator &operator--();
  iterator operator--(int);
  iterator &operator+=(difference_type offset);
  iterator operator+(difference_type offset) const;
  iterator &operator-=(difference_type offset);
  iterator operator-(difference_type offset) const;
  difference_type operator-(const iterator &rhs) const;
  pointer operator->() const;
  reference operator*() const;
  reference operator[](difference_type offset) const;
  Node* node_ref() const
  {
    return p_->node_ref();
  }
};
```

Here, the class iterator inherits the class const_iterator. Iterator has all the functions that a const_iterator class has, even though not all functions are there. This is allowed since const_iterator is not an abstract class.