COMP 2012H Honors Object-Oriented Programming and Data Structures

Assignment 9 Multi-type map using Skiplist

Page maintained by

Brian Chung
twchungab@connect.ust.hk
Last Modified:
10/31/2022 11:02:57

Homepage

Course Homepage

Honor Code

We value academic integrity very highly. Please read the Honor Code section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there will be additional disciplinary actions from the department and university, upto and including expulsion.

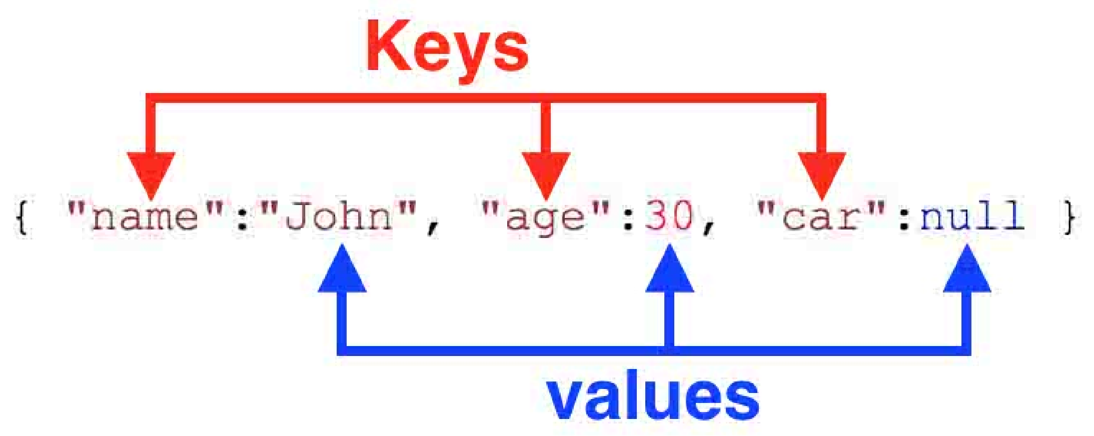
End of Honor Code

Objectives & Intended Learning Outcomes

The objective of this assignment is to familiarise you with templates, structs, pointers, operators overloading, and dynamic binding. After this assignment you should be able to:

1. Use templates to generalize data structures.
2. Understanding the principle of Skiplist and implement it.
3. Use `typeinfo` library and templates to manipulate with types.
4. Implement a C++ `auto`-like class.

End of Objectives & Intended Learning Outcomes



An example of a multi-type map: JSON. It can be considered as a map where its values can be `int`, `string`, etc...

Source: https://www.shapediver.com/blog/json-objects-explained

Introduction

The goal of this assignment is to implement:

- (I) A map data structure by using Skiplist, and

- (II) A C++ auto-like class, i.e. a wrapper class that could store any type of data, just like the Object class in Java.

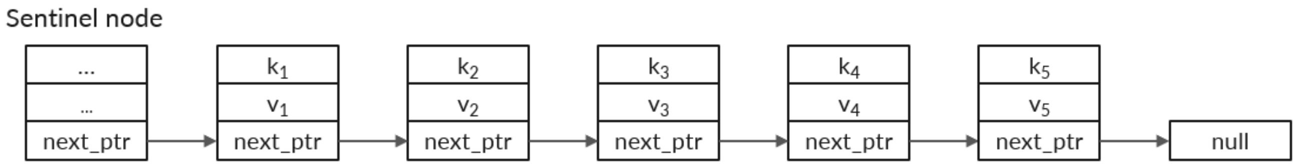
If you have implemented these 2 parts successfully, combining them together should give you a multi-type map. You may have questions like: What is a "map"? What is a "Skiplist"? Don't worry! All of the them will be mentioned below.

Map

A map (or dictionary) is a data structure that consists of a list of `(key, value)` pairs. Usually, we may represent a map like this: `{"key1": value1, "key2": value2, ...}`. For example, the following represents a map of student IDs:

- {"Sam": 12345678, "Tom": 11223344}
  - The key `"Sam"` is having a value `12345678`
  - The key `"Tom"` is having a value `11223344`

Using the example above, if we query the map with the key `"Sam"`, then the map will return its value `12345678` to us. For the implementation, one may use linked-list with sentinel node (if you forgot what is a sentinel node, you may go back to the PA5 Description Page to have a review) to implement the map.

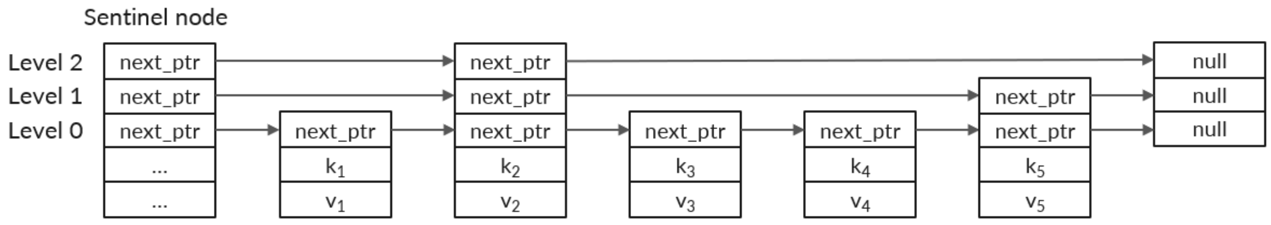


A map implemented using linked-list. The "..." in the sentinel node means we don't care about the value of it.

As can be seen from the above, when using linked-list to implement a map, a `Node` is having data memebers `key`, `value`, and `next_ptr`. Also, it is ovbious that the insert, delete and query operation in this kind of map is literally the same as the operations in a linked-list.

Skiplist

However, one of the disadvantages of linked-list is that it will need to traverse the whole list if the key being queried is at the end of the list. This leads to slow operations if the given list is large!
The main reason that it is slow because there is only 1 linked-list to traverse. Therefore, why not try to make more linked-lists instead of just 1? This is basically the idea of Skiplist!



An example of a Skiplist.

Here we will briefly talk about the structure of Skiplist.

From the perspective of the whole Skiplist:

- All the nodes in the Skiplist need to be **sorted by the key**, this implies the keys in the Skiplist need to be comparable.
  - Using the Skiplist example above, it implies `k1 < k2 < k3 < k4 < k5`.
- There are **multiple levels** of linked-lists, where the number of levels in the Skiplist are probabilistic, and it is based during the creation of the nodes.
  - Details will be given from the perspective of a node.
- The Level 0 linked-list must be the oridinary linked-list that links all the nodes in the Skiplist together.
- While the other levels of linked-lists may skip some of the nodes in the Skiplist.
  - Thats the word "skip" in Skiplist means.
  - You can think of these extra linked-lists are express lane for searching due to the fact that the nodes are sorted.

From the perspective of a node:

- Instead of having only one `Node*`, each `Node` is going to have an **array of** `Node*` to store the array of `next_ptr`.
- The `Node*` array must have at least length of 1, as each `Node` must be linked through the Level 0 linked-list.
- Except the sentinel node, the length of the `Node*` array will be determined during the insertion of the `Node` and will not be changed after the `Node` is being inserted to the Skiplist.
- Continue from the point above, if the `Node` is being linked by Level `i`, it will be linked by Level `i+1` too with a fixed probability `p`.
    - In other words, if a `Node` is being linked by Level `i`, then it must also be linked by Level `i-1, i-2, ..., 1, 0` too.
- Therefore, a `Node*` array with length `i` means that Node is being linked by Level `i-1, i-2, ..., 1, 0` linked-lists.

Basic Operations of the Skiplist map

The routine of the operations in the Skiplist will be described here. You may also want read this supplementary slides which contain concrete examples.

Search (Assuming the key exists)

1. Starting from the highest level of the Skiplist.
2. Traverse the current level until
    - The required `Node` with the key is found.
        - Found! Return the `Node` value.
    - The current `Node` key is larger than the desired key **OR** the traversal reaches the end.
        - Go back to the previous `Node` on the current level.
3. Go down by 1 level, repeat Step 2 until the desired key is found.

The case where the key does not exist in the map is not mentioned here. This is left as an exercise. ;)

Insert

- A new `Node` will be created and its `Node*` array length will be decided as follows:
    1. Initiate the length to 1.
    2. `true` with a probability `p`, `false` otherwise.
    3. Add the length by 1 if the result in Step 2 is `true`, and repeat Step 2. Otherwise, stop the iteration.
    - This algorithm has been implemented to you in the skeleton code already so you don't have to implement it yourself.
- Now, just insert the new created `Node` to the Skiplist like what you normally do with oridinary linked-list.

Note:

- You will need to find a siutable position to insert the `Node` as to be recalled that the keys in the Skiplist needs to be kept sorted.
- The sentinel node `Node*` array may need to be expanded if the `Node*` array of the inserting `Node` is larger.

Delete

- Very simple and easy, just remove the node on all the linked-list levels like what you normally do on oridinary linked-list.

Note:

- You may also need to shrink the sentinel node `Node*` array if necessary, same idea as inserting.

End of Introduction

Description

Please read the FAQ and Change Log sections regularly, and do check it one day before the deadline to make sure you don't miss any clarification, even if you have already submitted your work by then. You can also raise questions on Piazza and remember to use the "pa9" tag.

Code structure

The skeleton code structure is as follows:

```
PA9
├── skiplist.hpp
├── object.hpp
└── main.cpp
```

The `skiplist.hpp` contains the Skiplist class definition and the class function declarations.

The `object.hpp` contains the object class (i.e. the C++ `auto`-like class) definition and the class function declarations.

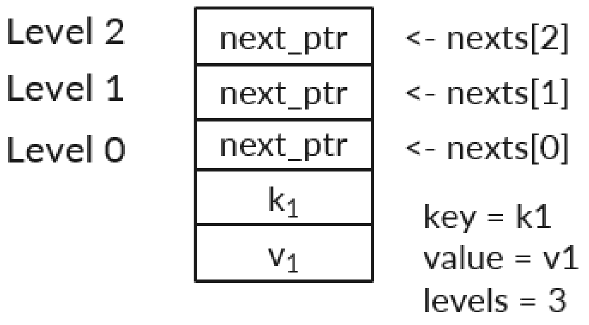
The `main.cpp` contains some test cases for you to test.

Definition of Skiplist map

The following is the `Node` class definition in this assignment, which is defined in `skiplist.hpp`.

```
template <typename K, typename V>
struct Node {
    K key; //The key of the Node
    V value; //The value of the Node
    Node** nexts; //Storing an array of Node* next_ptr
    int levels; //Length of Node** nexts
};
```

Here is an example of a `Node` with data members shown:



The following is the `Skiplist` class definition in this assignment, it is also defined in `skiplist.hpp`.

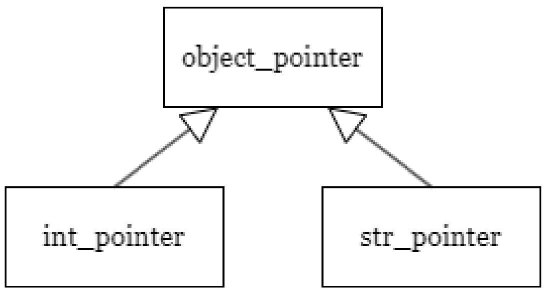
```
template <typename K, typename V>
class Skiplist {
    double prob; //the prob `p` to add a new level for a node
    Node<K, V>* head; //Node* that points to the head (sentinel node) of the Skiplist
};
```

Implementation of `object` class

In order to understand how an `object` class can be implemented, lets consider a simplier example. How can we create a class that could store a value of `int` or `string`? Polymorphism and dynamic binding may help us here! For example, one can implement it like this:

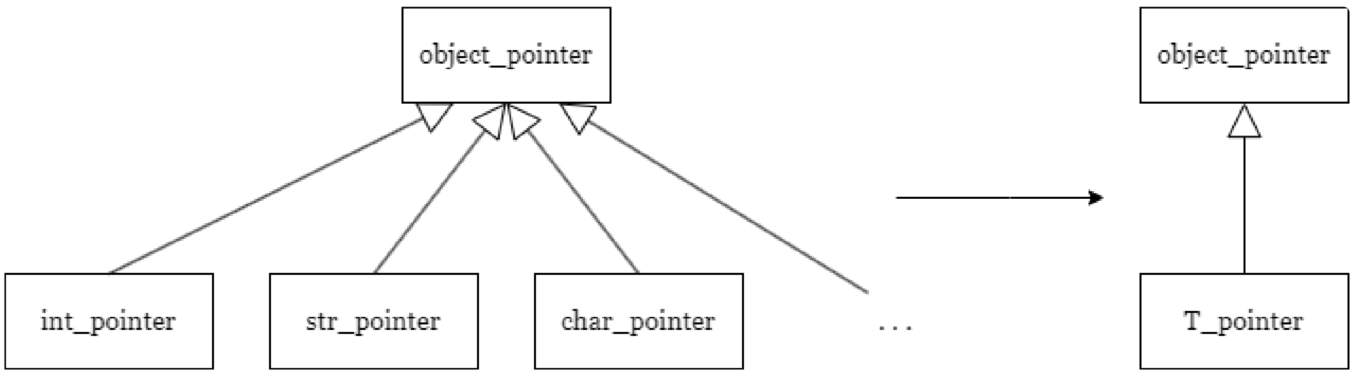
```
//Constructors or member functions definition are not shown
class object_pointer { /*...*/ };
class int_pointer : public object_pointer {int value; /*...*/};
class str_pointer : public object_pointer {string value; /*...*/};

//This pointer may now point to an object that is either containing a int value or a string v
object_pointer* int_or_str;

int_or_str = new int_pointer(1); //It now stores an int
int_or_str->type() //"It's a int"

int_or_str = new string_pointer(string("2012H")); //It now stores a string
int_or_str->type() //"It's a string"
```

The following diagram shows the relationship between classes of the mentioned example:



What should be the next step to create the `object_pointer` class? It is impossible for us to create a `type_pointer` class for each type! The answer is obvious though, we can make use of **generalization**.

By generalizing `int_pointer`, `str_pointer`, etc... into a class named `T_pointer`, we will be able to implement the `object_pointer` class successfully.



The idea of how the `object_pointer` class works by using polymorphism and generalization.

As a result, we can create the `object` class which is just a wrapper to store a `object_pointer*`. The `object` class definition can be found inside `object.hpp`:

```
class object {
    object_pointer* obj_ptr;
};
```

The definition of `object_pointer` and `T_pointer` are also included in `object.hpp`, but they are incomplete. You will have to complete them as one of the tasks in this assignment.

Additional Remarks

- You are required to implement
  - The Part I tasks in `skiplist.tpp`,
  - The Part II tasks in `object.hpp` and `object.tpp`,

  The `object.hpp` file has already been given to you but you will have to create `skiplist.tpp` and `object.tpp` on your own, **and submit these three files only to ZINC**.
- You are ONLY allowed to modify the structs `object_pointer` and `T_pointer`, and uncommenting function declarations of bonus tasks (if applicable) inside `object.hpp`. **Any modifications beyond inside `object.hpp`, or modifications inside `skiplist.hpp` are strictly forbidden.**

- You are NOT allowed to include any additional libraries. All the required libraries are already included in the skeleton code.
- You are NOT allowed to use the `auto` or `decltype` keywords.
- You are NOT allowed to define any static or global variables, or additional classes or structs. There is no need of it in this assignment.

End of Description

Tasks

This section describes the headers or functions that you will need to implement. It is not required to implement Part 1 before Part 2.

Part I: Skiplist

Implement the following functions of `Node` and `Skiplist` class inside the file `skiplist.tpp`. You may want to implement the functions in order as the functions you implemented before may help you in the latter functions.

```
Node<K, V>()
```

Description - The default constructor of `Node`. This will be used to create the sentinel node.

- As all nodes must have Level 0, you should allocate a dynamic `Node*` array of size 1 and have `nullptr` point by it.
- Don't forget to initialize the member variable `levels`.
- There is no need to initialize member variables `key` or `value` as we don't care about them in the sentinel node. You may assume `K` and `V` is going have default construtor.

```
Node<K, V>(K key, V value, int levels)
```

Description - The other constructor of `Node`. This will be used to create a regular node.

- Allocate a dynamic `Node*` array of size `levels` and have `nullptr` point by it.
- Don't forget to initialize the remaining member variables.

Parameters

- `key` - the key of the node.
- `value` - the value of the node.
- `levels` - the number of levels of the node, which should be larger than 0.

```
~Node<K, V>()
```

Description - The destructor of `Node`. Remember to deallocate any allocated memory.

```
Skiplist<K, V>(double prob = 0.25)
```

Description - The conversion constructor of `Skiplist`. This creates an empty Skiplist.

- An empty Skiplist means having just a sentinel node.
- Initialize the member variable `prob` too.

Parameters

- `prob` - the probability `p` to add a new level for a node, used in Skiplist inserting, defaults to `0.25`.

```
~Skiplist<K, V>()
```

Description - The deconstructor of `Skiplist`. Remember to deallocate any allocated memory.

```
Skiplist<K, V>& operator=(const Skiplist<K, V>& other)
```

Description - The assignment operator of `Skiplist`. You should do a deep copy of the whole Skiplist.

Parameters

- `other` - the `Skiplist` object to be copied.

Return value - A reference to the target of assignment.

```
Skiplist<K, V>(const Skiplist<K, V>& other)
```

Description - The deep copy constructor of `Skiplist`.

Parameters

- `other` - the `Skiplist` object to be copied.

Notes

- You may make use of the `operator=` function if you want.

```
bool get(const K& get_key, V& rtn_value) const
```

Description - The search function of `Skiplist`. Gets the value associated with the key `get_key`, and the result is returned by reference with `rtn_value`.

Parameters

- `get_key` - the target key to be found.
- `rtn_value` - a reference parameter which is used to return the value associated with `get_key`. Should not be touched if the target key is not found.

Return value - `true` if the target key is found, `false` otherwise.

Notes

- For example, if the Skiplist is having the content `{"Ben": 111, "Mary": 123}`, then
  - `get("Mary", my_var)` should return `true`, with `my_var` being `123` after the function returns,
  - `get("John", my_var)` should return `false`.

```
void update(const K& update_key, const V& update_value)
```

Description - The update function of `Skiplist`. Updates the value associated with the key `get_key` to `update_value`. In exact, your function should:

- If the node with key `update_key` is found, just replace the associated value of that node to `update_value`.
- Otherwise, if the node doesn't exist in the Skiplist, you should create a new node and insert it to the Skiplist.

Parameters

- `update_key` - the target key to be updated.
- `update_value` - the new value to be associated with the key `update_key`.

Notes

- In the scenario where a node needs to be created, you should determine its node level by calling the `getRandomLevels()` function once. As mentioned in Basic Skiplist Operations, this function has already been implemented to you.
- Be reminded that you shall keep the Skiplist sorted.
- You should find the node to be updated, or the correct position to insert the new node by a similar approach on how you search for a node in Skiplist.
- Be reminded that the sentinel node level should also be extended when necessary.
- For example, if the Skiplist is having the content `{"Ben": 111, "Mary": 123}`, then
    - `update("Mary", 999)` should update the Skiplist to `{"Ben": 111, "Mary": 999}`,
    - `update("John", 567)` should update the Skiplist to `{"Ben": 111, "John": 567,`

```
bool remove(const K& remove_key)
```

Description - The delete function of `Skiplist`. This removes the node with key `remove_key` from the Skiplist. No action is needed if the key doesn't exist in the Skiplist.

Parameters

- `remove_key` - the key to be removed.

Return value - `true` if the key `remove_key` exists in the Skiplist originally, `false` otherwise.

Notes

- Again, you should find the node to be removed, by a similar approach on how you search for a node in Skiplist.
- Be reminded that the sentinel node level should also be shrinked when necessary.
- For example, if the Skiplist is having the content `{"Ben": 111, "Mary": 123}`, then
    - `remove("Mary")` should update the Skiplist to `{"Ben": 111}`, and return `true`,
    - `remove("John")` should remain the Skiplist unchanged, and return `false`.

```
int size() const
```

Description - Returns the size of the Skiplist.

Return value - The number of key-value pairs in the Skiplist.

Notes

- For example,
    - An empty Skiplist returns `0`,
    - A Skiplist being `{"Ben": 111, "Mary": 123}` returns `2`.

```
bool empty() const
```

Description - Returns whether the Skiplist is empty.

Return value - `true` if the Skiplist is empty, `false` otherwise.

```
void print() const
```

Description - Prints the Skiplist map out in the form of key-value pairs. (i.e. `{k1: v1, k2: v2, ...}`)

- An `endl` should be printed at the end.

Notes

- For example,
    - An empty Skiplist prints `{}`,

    - A Skiplist being `{"Ben": 111, "Mary": 123}` will print `{Ben: 111, Mary: 123}`.

```
template <typename T>
Skiplist<K, T> map(T (*f)(V value)) const
```

Description - This function maps all the value of type `V` in a Skiplist, to another value of type `T` using the function `f`.

Parameters

- `f` - the mapping function that takes a paramater `value`, which returns a new value based on the original value.

Return value - The new mapped Skiplist.

Notes

- The parameter `f` here is called **function pointer**, it is just C++ supports us to **pass function** as a parameter to another function. You can just use `f` like how you use a function normally.
- Only the key-value pairs of the returned Skiplist will be checked. This means the returned Skiplist is not required to be structurally the same as the original Skiplist.
- You may want to make use of functions that you have implemented already.
- For example, if we define the function

  ```
  double square(int x) {return x*x;}
  ```

  and if we have a `Skiplist<string, int>` being `{"a": 3, "b": 6}`:
    - `map(square)` will returns a `Skiplist<string, double>` being `{"a": 9, "b": 36}`.

```
Skiplist<K, V> filter(bool (*f)(V value)) const
```

Description - This function filters all the values in a Skiplist, using the given function `f`.

Parameters

- `f` - the filter function that takes a paramater `value`, which will be filtered out from the Skiplist if `f` returns `false`.

Return value - The new filtered Skiplist.

Notes

- Only the key-value pairs of the returned Skiplist will be checked. This means the returned Skiplist is not required to be structurally the same as the original Skiplist.
- You may want to make use of functions that you have implemented already.
- For example, if we define the function

  ```
  bool positive(double x) {return x > 0;}
  ```

  and if we have a `Skiplist<string, double>` being `{"a": 3.1, "b": -6.0}`:
    - `filter(positive)` will returns a `Skiplist<string, double>` being `{"a": 3.1}`.

```
Skiplist<K, V> operator+(const Skiplist& other) const
```

Description - The `+` operator merges two Skiplists together. You can just think this function as "batch" update.

Parameters

- `other` - the other Skiplist to be merged with the current Skiplist.

Return value - The merged Skiplist.

Notes

- Only the key-value pairs of the returned Skiplist will be checked. This means the returned Skiplist is not required to be structurally the same as the original Skiplist.
- The order of the key-value pairs in the map should be the same as in the Skiplist.
- You may want to make use of functions that you have implemented already.
- For example, if we have a Skiplist `a` being `{"Ben": 111, "Mary": 123}`:
  - Let Skiplist `b` being `{"Alex": 111, "John": 345}`, then `a + b` returns a Skiplist being `{"Alex": 111, "Ben": 111, "John": 345, "Mary": 123}`
  - Let Skiplist `c` being `{"Alex": 111, "Mary": 345}`, then `a + c` returns a Skiplist being `{"Alex": 111, "Ben": 111, "Mary": 345}`

Part II: Object

Header of `object.hpp`

Before you start to implement functions for `object` class, you will have to complete the header `object.hpp` first. Here is the skeleton code provided to you for structs `object_pointer` and `T_pointer`.

```
struct object_pointer {

};

struct T_pointer {
    T value;
    T_pointer(T value);
    const std::type_info& type() const;
    object_pointer* copy() const;
};
```

As can be seen, they are incomplete. You task here is to use the ideas mentioned in the implementation of object class, which are **dynamic binding and generalization** to complete the two structs.

Important Notes

- You are ONLY allowed to modify the struct `object_pointer` and `T_pointer`.
- You are NOT allowed to add/remove/rename any data members to both structs.
- During the grading, we will use your `object.hpp` header file to compile the program. Please make sure you have performed this task correctly. It should be fine as long as your `object` class member functions are working properly, as only the `object` class member functions will be tested.

Member functions of `T_pointer`

Implement the member functions of `T_pointer` inside `object.tpp`. You should implement these functions first before you move on implementing the functions for `object` class. If you are implementing correctly, they should all just be in few lines.

```
T_pointer(T value)
```

Description - Conversion constructor for `T_pointer`, uses to initialize `value`.

Parameters

- `value` - the value to be stored in `T_pointer`

```
const std::type_info& type() const
```

Description - This function returns the `type_info` of `T`, using the `typeinfo` library.

Return value - The `typeid` of type `T`.

```
object_pointer* copy() const
```

Description - This function returns a deep copy of the current `T_pointer` object.

Return value - A `object_pointer*` pointing to a new `T_pointer` object that is deep copied from the current object.

Member functions of `object`

Implement the member functions of `object` inside `object.tpp`. If you are implementing correctly, they should all just be in few lines.

```
template <typename T>
object(const T& value)
```

Description - Conversion constructor for `object`.

Parameters

- `value` - the value to be stored to the `object` object.

```
~object()
```

Description - Destructor for `object`. Remember to deallocate any allocated object.

```
object& operator=(const object& other)
```

Description - The assignment operator of `object`. You should do a deep copy of the `object` object.

Parameters

- `other` - the `object` object to be copied.

Return value - A reference to the target of assignment.

```
object(const object& other)
```

Description - The deep copy constructor of `object`.

Parameters

- `other` - the `object` object to be copied.

Notes

- You may make use of the `operator=` function if you want.

```
const std::type_info& type() const
```

Description - This function gets the type of the current object.

Return value - the `type_info` of the type that the `object` object is storing.

Notes

- For example,

  ```
  object a = 1;
  a.type() //Type info of int

  object b = 1.5;
  a.type() //Type info of double
  ```

```
template <typename T>
T cast_back() const
```

Description - This function casts the value that the current object is storing back to type `T`.

Return value - the value that the `object` object is storing in type `T`.

Notes

- You should check whether the type `T` is the same as the type that the current object is storing first.
- If it is the same, you should achieve this by using `dynamic_cast`.
- Otherwise, you should throw an error, identicating the type is not the same.
  - Throwing error is a way to stop the execution of the program, when the user does something that should not be done.
  - You can throw the error easily by just writing this line:

    ```
    throw std::runtime_error("Object casting failed!");
    ```

- For example,

  ```
  object a = 1; //Storing int
  int b = a.cast_back<int>(); //OK!!

  object b = 1.5; //Storing double
  int c = b.cast_back<int>(); //Error thrown, as the type double != int
  ```

Bonus

This part is the bonus part of this assignment. **It is an optional task and you can choose to or not to finish it.** For the bonus, implement the following member functions of `object` inside `object.tpp`. The respective function declarations has been given to you inside `object.hpp` but they are commented. **You will need to uncomment the respective declaration in the header if you have finished it.**

Hint

- You may have to modify the struct `object_pointer` and `T_pointer` too. Again, be reminded that
  - You are ONLY allowed to modify the struct `object_pointer` and `T_pointer`.
  - You are NOT allowed to add data member to both structs.
  - During the grading, we will use your `object.hpp` header file to compile the program. Please make sure your program works for other tasks after you have implemented the bonus tasks.
- Be reminded you are free to modify your `.tpp` file too, as long as you don't violate the rules mentioned.

```
object operator+(const object& other) const
```

Description - This function overloads the `+` operator to perform addition. Error will be thrown if the two objects are storing different types of values.

Return value - An `object` object which is the sum of two objects.

Notes

- You may assume all the value types that `object` stores have `operator+` defined.
- You should first check whether the type of this object is storing is the same as the type that the other object is storing.
- If it is not the same, you should throw an error, identicating the type is not the same.
  - Throw the error by writing this line:

    ```
    throw std::runtime_error("Types needed to be the same when performing +");
    ```

- For example,

  ```
  object a = 1; //Storing int
  object b = 2; //Storing another int
  object c = a + b; //OK!! `c` now stores an int of 3

  object d = 1; //Storing int
  object e = 1.5; //Storing double
  object f = d + e; //Error thrown, as type int != double
  ```

```
bool operator==(const object& other) const
```

Description - This function overloads the `==` operator to perform comparison.

Return value - `true` if the two objects are storing the same type and value, `false` otherwise.

Notes

- You may assume all the value types that `object` stores have `operator==` defined.
- For example,

  ```
  object a = 1; //Storing int
  object b = 1; //Storing another int
  bool c = (a == b); //True, as both storing int and 1 == 1

  object d = 2; //Storing int
  object e = 2.0; //Storing double
  bool f = (d == e); //False, as one is double and one is int
  ```

```
friend std::ostream& operator<<(std::ostream &os, const object& obj)
```

Description - This function overloads the `<<` operator, this outputs the value that the object is storing to the output stream `os`.

Parameters

- `os` - the output stream.
- `obj` - the `object` object to be output to `os`.

Return value - A reference to the parameter `os`.

Notes

- You may assume all the value types that `object` stores have `operator<<` defined.
- For example,

```
object a = string("123");
cout << a; //Outputs 123
```

End of Tasks

Resources & Sample I/O

- Skeleton code: PA9_Skeleton.zip
- Demo program: No demo programs will be provided as the Skiplist and object class are both "Abstract Data Type (ADT)". 19 of known test cases are all already provided in `main.cpp`. You can just compile your program to try it out.
- Sample outputs: You can download the sample outputs for the known 19 test cases here. Any user input is omitted in the output files. Please note that the sample output, naturally, does not show all possible cases. It is part of the assessment for you to design your own test cases to test your program.

End of Resources & Sample I/O

Submission & Grading

Deadline: 12 November 2022 Saturday HKT 23:59.
Submit a zip of three files `skiplist.tpp`, `object.hpp` and `object.tpp` to ZINC.

Grading Scheme

There are 19 given test cases which the code can be found in the given main function in `main.cpp`. These 19 test cases are first run without memory leak checking (numbered #1 - #19 on ZINC). Then, the same 19 test cases will be rerun, in the same order, with memory leak checking (those will be numbered #20 - #38 on ZINC). For example, test case #21 on ZINC is actually the given test case 2 (in the given main function) run with memory leak checking.

Each test case run without memory leak checking (i.e., #1 - #19 on ZINC) is worth 1 mark. The second run of each test case with memory leak checking (i.e., #20 - #38 on ZINC) is worth 0.5 mark. The maximum score you can get on ZINC before the deadline, will be 19*(1+0.5) = 28.5.

About memory leak and other potential errors
Memory leak checking is done via the `-fsanitize=address,leak,undefined` option (related documentation here) of a recent g++ compiler on Linux (it won't work on Windows for the versions we have tested). Check the "Errors" tab (next to the "Your Output" tab in the test case details popup) for errors such as memory leaks. Other errors/bugs such as out-of-bounds, use-after-free bugs, and some undefined-behavior-related bugs may also be detected. You will get a 0 mark for the test case if there is any error. Note that if your program has no errors detected by the sanitizers, then the "Errors" tab may not appear. If you wish to check for memory leaks yourself using the same options, you may follow the Checking for memory leak yourself guide.

Test cases summary
We will have 59+6 additional test cases which won't be revealed to you before the deadline. Together with the 19 given test cases, there will then be 78+6 test cases used to give you the final assignment grade. All 78+6 test cases will be run two times as well: once without memory leak checking and once with memory leak checking. The assignment total will therefore be 78*(1+0.5) = 117, and an additional of 6*(1+0.5) = 9 marks for bonus.

The following table shows the summary of the test cases.

Task Function		Number of test cases before deadline (Known test cases)	Number of test cases after deadline (Known+hidden test cases)
Part I	`Node()`	1	1
	`Node(K, V, int)`	1	2
	`Skiplist()`	1	2
	`Skiplist(const Skiplist&)`	1	4
	`get(const K&, V&)`	1	9
	`update(const K&, const V&)`	1	7
	`remove(const K&)`	1	10
	`size()`	1	3
	`empty()`	1	2
	`print()`	1	3
	`map(T (*f)(V))`	1	3
	`filter(bool (*f)(V))`	1	3
	`operator=(const Skiplist&)`	1	4
	`operator+(const Skiplist&)`	1	3
Part II	`object(T value)`	1	5
	`object(const object&)`	1	3
	`operator=(const object&)`	1	5
	`type()`	1	4
	`cast_back()`	1	5
Bonus	`operator+(const object&)`	0	2
	`operator==(const object&)`	0	2
	`operator<<(std::ostream&, const object&)`	0	2

End of Submission & Grading

Frequently Asked Questions

Q: My code doesn't work, here it is, can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it by yourself and we should never finish the tasks for you. We are happy to help with explanations and advice, but we are **not allowed** to directly debug for you.

Q: Am I allowed to create helper functions?
A: Yes.

Q: When I am testing with my code, `object a = "test"` gives me a compilation error. Is it normal?
A: Yes. Be reminded that the syntax of double quote is just a syntax-sugar for a char array. This happens because the `object` class we created in this assignment actually doesn't support storing array types. Therefore, you don't have to worry about it as there won't be test cases that construct `object` from an array.
In case you wonder the solution to it, one can use `std::decay`. For details, you may refer to this link.

End of FAQ

Change Log

- **10/31 10:49**
  1. Fixed `print_full()` inside the skeleton `skiplist.hpp` so that the program output matches with the sample output. Piazza@345

End of Change Log