COMP 2012 Object-Oriented Programming and Data Structures

Assignment 3 Smart Pointers

Page maintained by

Chun Yin CHAU
Email: cychauab@cse.ust.hk
Last Modified: 04/16/2022 14:42:05

Homepage

Course Homepage

Introduction

The main source of bugs in C++ is often believed to be the improper use of pointers. While pointers provide a very flexible way of storing and passing around references to objects, the ownership and lifetime of dynamically allocated objects on the heap becomes unclear. An object on the stack has a clearly defined scope, which also unambiguously defines its lifetime. It is obvious that the object should be deallocated once it goes out of scope, so obvious that it is handled by the compiler automatically. Dynamic memory allocation allows the programmer to control the lifetime of the object, but with power comes responsibilities. The programmer must ensure that the object is deallocated when it is no longer used, and that the object is not used after it has been deallocated.

When multiple pointers to the same object are held by different parties, it is unclear who should be responsible for deallocating the object. In fact, there is no trivial way to determine whether the object is even safe to deallocate, since there is no way to tell whether another pointer to the same address exists. The worst part is that pointers can also point to addresses on the stack, so there is no way to determine whether an object pointed to by a pointer can even be deallocated by the programmer at all.

In an attempt to solve this ownership problem of pointers in C++, we implement the concept of smart pointers, which are wrappers of ordinary pointers, with a few restrictions and some extra utilities. Together with the ordinary pointer, the smart pointer keeps track of the number of smart pointers holding the same address, and automatically deallocates the object when no instances of smart pointers hold the address. To ensure that smart pointers do not accidentally deallocate objects with other references held as ordinary pointers or references, smart pointers can only be initialized or assigned with a newly allocated address or an address held by another instance of smart pointer. The reference counters are always incremented or decremented appropriately by the constructors, destructor, assignment operator and mutators.

As an example to demonstrate the ownership problem and the application of the smart pointers we just implemented, the second half of this assignment will be an implementation of a simple representation of graphs and operations on them. Since a node can be the neighbor of, and hence have its reference held by multiple other nodes, smart pointers can be used to keep track of the number of held references of a node, and deallocate a node when no other nodes hold a reference to it.

We value academic integrity very highly. Please read the Honor Code section on our course web page 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 some sophisticated plagiarism detection software to find cheaters. It is much better than most students think. It has been proven times and times again tricks didn't work. 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 - it is much more than that. It is simply not worth to cheat at all. You would hurt your friend and yourself by doing that. It is obvious that a real friend won't ask you to get involved in a plagiarism act in any way due to the consequences. Read the Honor Code again before you even try to think about cheating.
- Serious offenders will fail the course immediately and there may be additional disciplinary actions from the department and university.

Download

Skeleton code: pa3-skeleton.zip

Please note that **you should only submit the required files**. While you may modify other files to add your own test cases, you should make sure your submitted files can be compiled with the original `main.cpp` and header files on ZINC.

If you use VS Code, you may follow the creating a project and using the terminal for custom compilation command section on our VS Code usage tutorial. That is, create a folder to hold all the extracted files in your file explorer, then open this folder in VS Code. You can then use the terminal command `g++ -std=c++11 -o programName main.cpp` to compile all sources in the folder to the program. You are also welcome to create a Makefile for it yourself. After the compilation, you can then use the command `./programName` to run the program.

End of Download

Review

We first review graphs, which is used as an example to demonstrate the ownership problem in this assignment.

A graph is a structure consisting of a set of *nodes* and a set of *edges*, which are pairs of nodes. In this assignment, we consider undirected graphs only, where an edge from a node `n1` to another node `n2` is also an edge from `n2` to `n1`. Additionally, self-edges (an edge from a node to itself) and duplicate edges are forbidden.

If an edge from `n1` to `n2` exists, we say that `n2` is a *neighbor* of `n1`. To represent a graph, we store the adjacency list of each node, which is simply a list of its neighbors. The *degree* of a node is the number of its neighbors.

A node `nk` is said to be *reachable* from a node `n0` if a sequence of edges `(n0, n1), (n1, n2), ..., (nk-1, nk)` exists. A *connected subgraph* of a node `n` is the set of reachable nodes from `n`.

As an example of graphs, think of the flights of an airline. The airports served by the airline are represented as nodes, and flights between two airports are represented as edges. The degree of a node (an airport) would then be the number of other airports connected by a direct flight. An airport is reachable from another airport if a sequence of flights connects

them. And the connected subgraph of an airport is the set of airport reachable by an arbitrary number of flights. Natually, the airline would prefer not to waste storage space for airports they no longer serve, so this is a perfect application of our smart pointers, which would deallocate the data of an airport when all flights to it have been discontinued (i.e. all edges of the node have been removed).

End of Review

Description

This assignment can be divided into three parts, with each depending on the previous one. The first two parts each consists of a class template, and the last part consists of some function templates that operate on the class templates. You should ensure that memory leak does not occur in **all** of the functions you implement.

Part (A) `SmartPtr<T>`

The class template `SmartPtr<T>` declared in `smartptr.h` provides an abstraction for handling memory allocation and deallocation using reference counting. It contains the following data members:

```
T* ptr;
unsigned int* count;
```

`ptr` is a pointer to type `T`. If the address stored in `ptr` is `nullptr`, we say that this `SmartPtr` is *null*. If `ptr` is not `nullptr`, it should point to a valid, allocated address. Note that a null `SmartPtr` only conceptually represents a null pointer, and the `SmartPtr` object itself is still defined. All member functions beside `operator*` can still be safely called on a null `SmartPtr`.

`count` is a pointer to type `unsigned int`, which is used to store the number of `SmartPtr` instances containing the address `ptr`. `count` should be `nullptr` if and only if `ptr` is `nullptr`.

Memory management of the object referred to by `SmartPtr`s are managed by the constructor, destructor, `set` and `unset` member functions. The only ways to initialize and assign `ptr` is by setting it to `nullptr`, an address allocated by copy-constructing an instance of type `T`, or the address contained in another instance of `SmartPtr`.

When a `SmartPtr` is created for or set to a new `T` object, `count` should be allocated to store the number 1, meaning, the address of the new `T` object is stored in only 1 `SmarPtr`. For example:

```
SmartPtr sp {42};
```

The number pointed to by `count` of `sp` is now 1 and `ptr` is allocated to store 42.

```
sp.set(0);
```

Again, the number pointed to by `count` of `sp` is set to 1 and `ptr` is allocated to store 0.

When a `SmartPtr` is copied by the copy constructor or assignment operator, the address stored in `ptr` is being copied, the number pointed to by `count` has to be incremented by 1. It means that the `T` object pointed by to `ptr` is being pointed to by one more `SmartPtr` instance.

Similarly, when a `SmartPtr` is destructed or set to refer to another object, the number pointed to by `count` has to be decremented. When the number becomes zero, it means that this instance of `SmartPtr` is the last to hold a reference to the object pointed to by `ptr`. Both `ptr` and `count` should be deallocated properly.

Member functions

`SmartPtr<T>::SmartPtr`

```
SmartPtr();
```

The default constructor initializes a null `SmartPtr`.

```
SmartPtr(const T& val);
```

The conversion constructor allocates memory for a value of type `T` by copy-constructing an instance of `T` from `val` and storing its address in `ptr`. `count` should be allocated and set to 1.

```
SmartPtr(const SmartPtr<T>& that);
```

The copy constructor does not allocate additional memory, but instead stores the same address in `ptr` as `that.ptr`. As this means one more `SmartPtr` instance is pointing to the address that `that.ptr` points to, the number pointed by `count` needs to be incremented accordingly.

`SmartPtr<T>::~SmartPtr`

```
~SmartPtr();
```

The destructor deallocates the memory pointed to by `ptr` when no other instances of `SmartPtr` hold the same address.

`SmartPtr<T>::operator=`

```
SmartPtr<T>& operator=(const SmartPtr<T>& that);
```

The assignment operator stores the same address in `ptr` as `that.ptr`. The memory originally pointed to by `ptr` is deallocated when no other instances of `SmartPtr` hold the same address. As this means one more `SmartPtr` instance is pointing to the address that `that.ptr` points to, the number pointed by `count` needs to be incremented accordingly. The returning value should allow cascading assignment, in other words, `sp1 = sp2 = sp3;` should be equivalent to `sp2 = sp3; sp1 = sp2;`.

`SmartPtr<T>::set`

```
void set(const T& val);
```

Allocates memory for a value of type `T` by copy-constructing an instance of `T` from `val` and storing its address in `ptr`. The memory originally pointed to by `ptr` is deallocated when no other instances of `SmartPtr` hold the same address. This is the first `SmartPtr` instance pointing to a `T` instance with value `val`, the number pointed by `count` should be 1.

`SmartPtr<T>::unset`

```
void unset();
```

Sets this `SmartPtr` to null. The memory originally pointed to by `ptr` is deallocated when no other instances of `SmartPtr` hold the same address.

`SmartPtr<T>::is_null`

```
bool is_null() const;
```

Returns `true` if this `SmartPtr` is null, and `false` otherwise.

`SmartPtr<T>::operator== SmartPtr<T>::operator!=`

```
bool operator==(const SmartPtr<T>& that) const;
bool operator!=(const SmartPtr<T>& that) const;
```

Compares the pointer member `ptr` of two instances of `SmartPtr` using the corresponding operators.

`SmartPtr<T>::operator*`

```
T& operator*() const;
```

Returns by reference the object pointed to by `ptr`. Whether `ptr` points to `nullptr` is not checked.

`SmartPtr<T>::operator->`

```
T* operator->() const;
```

Returns the address stored in `ptr`.

Non-member functions

`operator<<`

```
template <typename T>
std::ostream& operator<<(std::ostream& os, const SmartPtr<T>& sp);
```

If `sp` is not null, outputs to `os` the string "`SmartPtr(`", followed by the object pointed to by `sp.ptr`, then by a "`,`", then by the number of `SmartPtr` instances containing the address `sp.ptr`, finally by "`)`". The object pointed to by `sp.ptr` should be output by invoking `operator<<`. Otherwise, outputs to `os` the string "`SmartPtr()`".

For example, if `sp.ptr` points to a string "Hello World", with two instances of `SmartPtr` holding the same address, `os << sp` should output "`SmartPtr(Hello World,2)`" to `os`.

The return value should follow the convention as the standard library, where chaining multiple invocations of `operator<<` outputs to the same stream. For example, `os << sp << "\n"` would output a newline character following "`SmartPtr(Hello World,2)`" to `os`.

Implement this function template in `smartptr-output.tpp`. No other required implementations should be present in `smartptr-output.tpp`. For grading, besides the cases explicitly testing `operator<<`, your implementation in `smartptr-output.tpp` will be swapped for a standard implementation.

Part (B) `Node<T>`

The class template `Node<T>` represents a node or vertex in a graph. It contains the following data members:

```
T val;
SmartPtr<Node<T>>* out;
unsigned int capacity;
unsigned int size_p;
```

`val` is the data contained in a node, which may be used to identify nodes. (However, note that the equality of nodes are determined by reference equality, not the equality of their `val` field.) `out` is a dynamically allocated array of `SmartPtr`s to the neighbors of a node. `capacity` is the actual allocated size of the array, while `size_p` is the number of "used" cells in the array. When a neighbor is removed, the corresponding `SmartPtr` in the array is simply set to null, without modifying other `SmartPtr`s or `size_p`. When a neighbor is added, its `SmartPtr` is always appended at the end of the "used" cells, not any of the null cells resulted from removing a neighbor.

Member functions

`Node<T>::Node`

```
Node(const T& val);
```

The conversion constructor initializes a `Node` with no neighbors by copy-constructing an instance of `T` from `val`. Memory should not be allocated for the array of `SmartPtr` to neighbors.

```
Node(const Node<T>& that);
```

The copy constructor initializes a `Node` with no neighbors by copy-constructing an instance of `T` from `that.val`. Memory should not be allocated for the array of `SmartPtr` to neighbors.

`Node<T>::~Node`

```
~Node();
```

The destructor deallocates any memory allocated for a node.

`Node<T>::operator*`

```
T& operator*();
const T& operator*() const;
```

The indirection operator returns by reference the member `val`.

`Node<T>::degree`

```
unsigned int degree() const;
```

Returns the number of neighbors of a node.

`Node<T>::size`

```
unsigned int size() const;
```

Returns the number of "used" cells in the array of `SmartPtr` to neighbors, i.e. `size_p`.

`Node<T>::operator[]`

```
SmartPtr<Node<T>> operator[](unsigned int i) const;
```

The subscript operator returns the `SmartPtr` at index `i` of the array of neighbors. The index includes any null `SmartPtr`s resulting the removal of neighbors. The array bounds are unchecked.

`Node<T>::add`

```
void add(SmartPtr<Node<T>> n);
```

Adds `n` to the array of neighbors by appending it to the end of the "used" cells. Does nothing if `n` is null, or if `n` is non-null and already a neighbor, or if `n` points to this node itself. If the array of neighbors is full, expand its capacity to double its current capacity or `init_capacity`, whichever is larger.

`Node<T>::remove`

```
void remove(SmartPtr<Node<T>> n);
```

Removes `n` from the array of neighbors. Does nothing if `n` is null, or if `n` is not a neighbor. Do not modify any other `SmartPtr`s or `size_p`.

`Node<T>::exists`

```
bool exists(SmartPtr<Node<T>> n) const;
```

Returns `true` if `n` is non-null and is a neighbor of this node. Returns `false` otherwise.

`Node<T>::find`

```
SmartPtr<Node<T>> find(T val) const;
```

Returns a `SmartPtr` to the first neighbor containing a value equal to `val`. Returns a null `SmartPtr` if none of the neighbors contain a value equal to `val`.

Non-member functions

`operator<<`

```
template <typename T>
std::ostream& operator<<(std::ostream& os, const Node<T>& n);
```

Outputs to `os` the string "`Node(`", followed by the value `val`, then by a "`,{`", then by a list of values `val` its neighbors, separated by commas, finally by "`})`". The values `val` should be output by invoking `operator<<`.

For example, if `n.val` is the integer 0, and `n` has three neighbors containing 1, 2 and 3 respectively, `os << n` should output "`Node(0,{1,2,3})`" to `os`.

The return value should follow the convention as the standard library, where chaining multiple invocations of `operator<<` outputs to the same stream.

Implement this function template in `graph-output.tpp`. No other required implementations should be present in `graph-output.tpp`. For grading, besides the cases explicitly testing `operator<<`, your implementation in `graph-output.tpp` will be swapped for a standard implementation.

Note: on the array `out` and related operations

In case you are confused by the above description, here's an example to illustrate the effect of the operations on the array `out` and the data members. Take `init_capacity = 4` in this example. `x` denotes a null `SmartPtr`.

Upon initialization, `out` should point to `nullptr`.

```
Node n {0};

       v size_p = capacity = 0
out[i] |
    i   0  1  2  3  4  5  6  7
```

As the first node is added as a neighbor, `out` is expanded to `init_capacity`. When each node is added, the element at index `size_p` is assigned the value of the corresponding `SmartPtr`, then `size_p` is incremented.

```
n.add(n1);
n.add(n2);
n.add(SmartPtr<Node<int>>{});
n.add(n3);

       size_p = 3  4 = capacity
                v  v
out[i] |n1|n2|n3| x|
     i   0  1  2  3  4  5  6  7
```

When a neighbor is removed, the corresponding element is `out` is set to a null `SmartPtr`. The rest of the elements are not moved to fill the gap, and `size_p` is unchanged.

```
n.remove(n2);

       size_p = 3  4 = capacity
                v  v
out[i] |n1| x|n3| x|
     i   0  1  2  3  4  5  6  7
```

When more neighbors are added after a neighbor has been removed, their entries are still appended to the end of the "used" cells in the array, not used to fill the gaps. When the array is full, expand its capacity to twice is original capacity.

```
n.add(n4);
n.add(n2);

            size_p = 5        8 = capacity
                     v        v
out[i] |n1| x|n3|n4|n2| x| x| x|
     i   0  1  2  3  4  5  6  7
```

Part (C) Graph operations

The following are function templates that operate on `SmartPtr`s to `Node`s. Graphs should be constructed using the following functions instead of directly calling the member functions of `Node`.

`new_node`

```
template <typename T>
SmartPtr<Node<T>> new_node(const T& val);
```

Returns a `SmartPtr` to a new instance of `Node`. The `val` member of `Node` should be copy-constructed (twice) from `val`.

`remove_node`

```
template <typename T>
void remove_node(SmartPtr<Node<T>> n);
```

Removes the node `n` from a graph by removing all edges between `n` and each of its neighbors, which would deallocate their memory if no other `SmartPtr`s hold a reference to them. Does nothing if `n` is null.

`add_edge`

```
template <typename T>
void add_edge(SmartPtr<Node<T>> n1, SmartPtr<Node<T>> n2);
```

Adds an edge connecting `n1` and `n2` by adding each of the nodes to the neighbor list of the other node. Does nothing if either `n1` or `n2` is null, or if an edge between `n1` and `n2` already exists, or if `n1` and `n2` refers to the same node.

`remove_edge`

```
template <typename T>
void remove_edge(SmartPtr<Node<T>> n1, SmartPtr<Node<T>> n2);
```

Removes the edge between `n1` and `n2` by removing each of the nodes from the neighbor list of the other node, which would deallocate their memory if no other `SmartPtr`s hold a reference to them. Does nothing if either `n1` or `n2` is null, or if an edge between `n1` and `n2` does not exist.

`remove_graph`

```
template <typename T>
void remove_graph(SmartPtr<Node<T>> root);
```

Removes every node from the connected subgraph of `root` by removing all edges between each pair of nodes, which would deallocate their memory if no other `SmartPtr`s hold a reference to them. Does nothing if `root` is null.

`find`

```
template <typename T>
SmartPtr<Node<T>> find(SmartPtr<Node<T>> root, T val);
```

Finds a node whose `val` field is equal to `val`. Returns a null `SmartPtr` if none of the nodes have a `val` field is equal to `val`, or if `root` is null.

For grading, you can assume that there is at most one node in the graph whose `val` field is equal to `val`, but the rest of the nodes may have equal `val` fields.

`reachable`

```
template <typename T>
bool reachable(SmartPtr<Node<T>> root, SmartPtr<Node<T>> dest);
```

Returns `true` if `dest` is reachable from `root`, i.e. `dest` is in the connected subgraph of `root`, and `false` otherwise. Returns `false` if either `root` or `dest` is null.

End of Description

Tasks

Your task is to implement the missing member functions of the two classes and function templates in `smartptr.tpp`, `smartptr-output.tpp`, `graph.tpp` and `graph-output.tpp`.

End of Tasks

Sample Output and Grading

Your finished program should produce the same output as our sample output for all given test cases. User input, if any, is omitted in the 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. Be reminded to remove any debugging message that you might have added before submitting your code.

There are 27 given test cases of which the code can be found in the given main function. These 27 test cases are first run without any memory leak checking (they are numbered #1 - #27 on ZINC). Then, the same 27 test cases will be run again, in the same order, with memory leak checking (those will be numbered #101 - #127 on ZINC). For example, test case #108 on ZINC is actually the given test case #8 (in the given main function) run with memory leak checking.

Each of the test cases run without memory leak checking (i.e., #1 - #27 on ZINC) is worth 1 mark. The second run of each test case with memory leak checking (i.e., #101 - #127 on ZINC) is worth 0.5 mark. The maximum score you can get on ZINC, before the deadline, will therefore be 27*(1+0.5) = 40.5.

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 "Your Output" tab in the test case details popup) for errors such as memory leak. 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 0 mark for the test case if there is any error there. 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 leak yourself using the same options, you may follow our Checking for memory leak yourself guide.

After the deadline

We will have 21 additional test cases which won't be revealed to you before the deadline. Together with the 27 given test cases, there will then be 48 test cases used to give you the final assignment grade. All 48 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 48*(1+0.5) = 72. Details will be provided in the marking scheme which will be released after the deadline.

End of Sample Output and Grading

Submission and Deadline

Deadline: 23:59:00 on 24 Apr 2022 (Sunday)

Submit a single zip file `pa3.zip` containing the following files only: `smartptr.tpp`, `smartptr-output.tpp`, `graph.tpp` and `graph-output.tpp`. Submit the file to ZINC. ZINC usage instructions can be found here.

Notes:

- You may submit your file multiple times, but only the last submission will be graded. **You do NOT get to choose which version we grade.** If you submit after the deadline, late penalty will be applied according to the submission time of your last submission.
- Submit early to avoid any last-minute problem. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day especially in the last few hours, so you should expect you would get the grading result report not-very-quickly. However, as long as your submission is successful, we would grade your latest submission with all test cases after the deadline.
- In the grading report, pay attention to various errors reported. For example, **under the "make" section, if you see a red cross, click on the STDERR tab to see the compilation errors.** You must fix those before you can see any program output for the test cases below.
- Make sure you submit the correct file yourself. You can download your own file back from ZINC to verify. Again, **we only grade what you uploaded last to ZINC**.

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online auto-grader ZINC. If we cannot even compile your work, it won't be graded. Therefore, for parts that you cannot finish, just put in dummy implementation so that your whole program can be compiled for ZINC to grade the other parts that you have done. Empty implementations can be like:

```
int SomeClass::SomeFunctionICannotFinishRightNow()
{
    return 0;
}

void SomeClass::SomeFunctionICannotFinishRightNowButIWantOtherPartsG
{
}
```

Late submission policy

There will be a penalty of -1% (out of a maximum 100%) for every minute you are late. For instance, since the deadline of the assignment is 23:59:00 on 24 Apr, if you submit your solution at 1:00:00 on 25 Apr, there will be a penalty of -61% for your assignment. However, the lowest grade you may get from an assignment is zero: any negative score after the deduction due to a late penalty (and any other penalties) will be reset to zero.

End of Submission and Deadline

Frequently Asked Questions

General

Q: My code doesn't work / there is an error, here is the code, 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 on your own and we should not finish the tasks for you. We might provide some very general hints to you, but we shall not fix the problem or debug for you.

Q: Can I add extra helper functions?

A: You may do so in the files that you are allowed to modify and submit. That implies you cannot add new member functions to any given class.

Q: Can I include additional libraries?

A: No. Everything you need is already included - there is no need for you to add any include statement (under our official environment).

Q: Can I use global variable or static variable such as "static int x"?

A: No.

Q: Can I use "auto"?

A: No.

Q: Can I use function X or class Y in this assignment?

A: In general if it is not forbidden in the description and the previous FAQs, and you can use it without including any additional library on ZINC, then you can use it. We suggest quickly testing it on ZINC (to see if a basic usage of it compiles there) before committing to using it as library inclusion requirement may differ on different environments.

Q: My program gives the correct output on my computer, but it gives a different one on ZINC. What may be the cause?

A: Usually inconsistent strange result (on different machines/platforms, or even different runs on the same machine) is due to relying on uninitialized hence garbage values, missing return statements, accessing out-of-bound array elements, improper use of dynamic memory, or relying on library functions that might be implemented differently on different platforms (such as pow() in cmath).
You may find a list of common causes and tips on debugging in the notes here.
In this particular PA, it is probably related to misuse of dynamic memory. Good luck with bug hunting!

Specification clarification

Q: Should I increment the `count` when I copy construct or copy assign a `SmartPtr`?

A: Since the `ptr` member of the newly constructed / assigned `SmartPtr` is `nullptr`, its `count` member should also be `nullptr` according to the description. There is no need to count the number of `SmartPtr`s containing the address `nullptr`.

End of Frequently Asked Questions