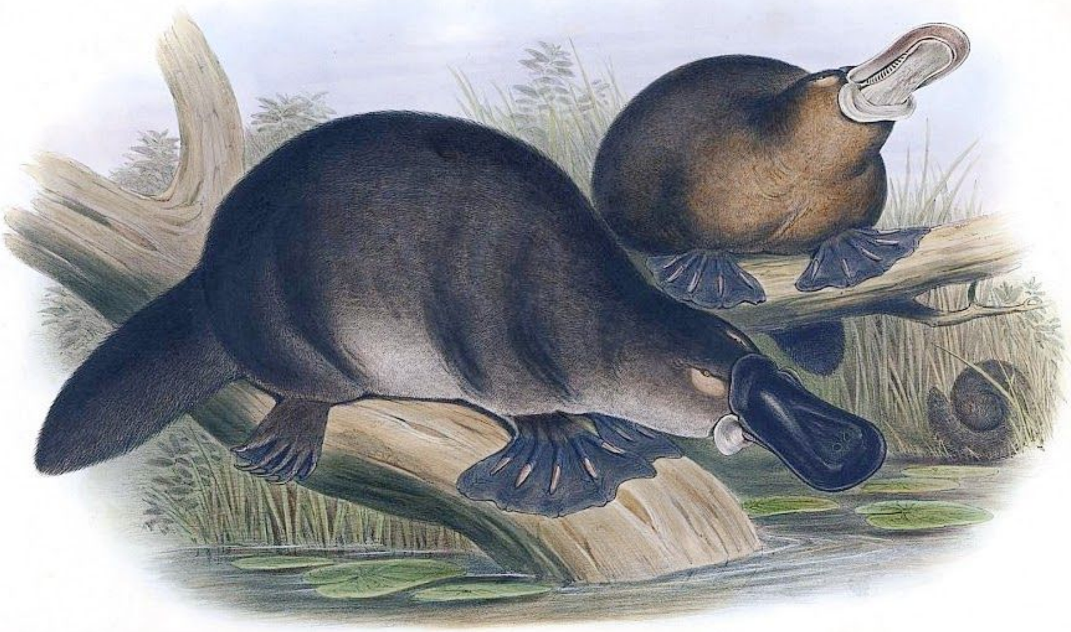


ORNITHORHYNCHUS ANATINUS.

J. Gould and H.C. Richter del. et lith.

Hullmandel & Walton Imp.

Playful Platypi

Personally, I've never played with one. But I'm thinking "Why shouldn't they be playful? They sure look like they wanna have fun."

I gotta say I've never seen one programming in C++, though. So who knows?

Some miniquests will give you some rewards for meeting a minimum requirement, and give you additional surprise points for doing something else right that wasn't explicitly asked.

In this quest, you will create an inline implementation for a class called `String_List` just like in your previous quest.

Here is what your `String_List` class should look like

```
class String_List {
private:
    struct Node {
        std::string data;
        Node *next;
        Node(std::string s = "") : data(s), next(nullptr) {}
    };
    Node *_head, *_tail, *_prev_to_current;
    size_t _size;

public:
    // TODO - You will need to fill in the inline implementations of
    // all the public methods below.
    String_List();
    ~String_List();

    size_t get_size() const;

    void clear();

    String_List *rewind();
    String_List *insert_at_current(std::string s);
    String_List *remove_at_current();
    String_List *push_front(std::string s);
    String_List *push_back(std::string s);

    // Remember that _prev_to_current can never be null because it's at
    // least equal to the list header. The current element visible to the user
    // is always the element AFTER _prev_to_current. So when you're advancing,
    // you can't advance further than the tail node (again a real non-null node).
    //
    String_List *advance_current();

    // This method is only a few lines, but please make sure you understand it
    // fully. Ask in the forums if any of it is unclear.
    //
    // The current item is the data element of whichever node is equal to
    // _prev_to_current->next.
    //
    // Ideally, we would throw an exception if _prev_to_current->next == null, which
    // should not happen, really. But since we haven't covered exceptions yet, we will
    // hack a return to a static sentinel, with the understanding that the caller
    // won't mess with the sentinel (see the spec).
    std::string get_current() const;

    // Find a specific item. Does NOT change cursor.
    //
    // The following returns a reference to the target string if found. But what will
    // you do if you didn't find the requested string? Using sentinel markers is
    // one way to handle that situation. Usually there's only one copy of the
    // sentinel that's global. We will use a local one so it's cleaner with a
    // little more risk (what's the risk?)
    std::string &find_item(std::string s) const;

    // Print up to max_lines lines starting at _prev_to_current->next. If the caller
    // wants to print from the beginning of the list, they should rewind() it first.
    // max_lines can be a local constant = 25;
    std::string to_string() const;
};
```

Let's discuss the above fragment briefly before moving on. It's possible you haven't come across the `struct` keyword before (unless you were a C programmer). A `struct` is simply a way of grouping a bunch of other variables together and giving a name to that collection. At least that's what we're using it for here.

We're defining a struct called Node, which contains a string member called `data` and a pointer member (a memory address) called `next`.

What does `next` point to? Its value is the memory address at which *a* `Node` can be found. It might even be its own address - that is, point to itself. That's the cool thing about creating linked data structures. Not only can you jump all over the place in your heap, you can also define *self-referential structures*, which is a big deal.

If you're thinking that the third line of `Node` looks like an inline constructor, you'd be right. Although I didn't mention this before, you can think of a `struct` in C++ as simply a class in which every member is public by default. So we can define a constructor for our `Node` variables that can set default values to one or both of its members.

Moving on to the `String_List` itself, note that it has three pointer members and a `_size` member. `_size` is simple. You just use it to track the current size of your linked list so you can return this value immediately instead of traversing the list to compute it each time someone asks.

Refer to Figure 1 to see how to initialize the three pointer members.

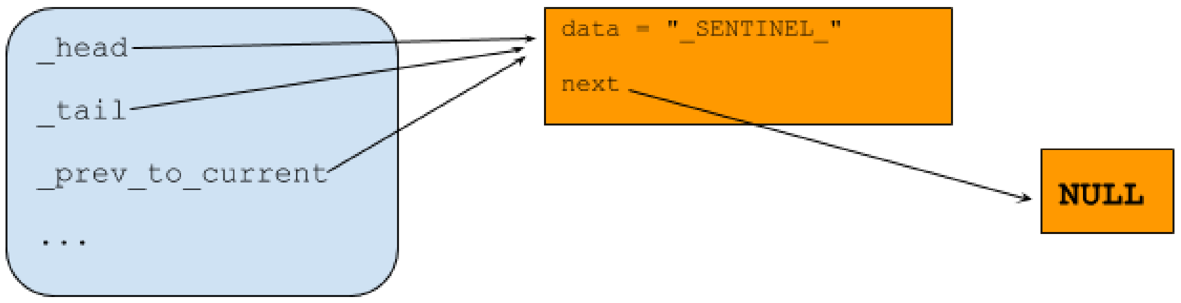


Figure 1. Pointers in an empty list

At minimum, a `String_List` should contain a single node whose data element is the string "`_SENTINEL_`". This node performs two functions.

First, it is the node that holds the payload (content) you need to return when asked for content you cannot find. It's value should be the sentinel string.[1]

Second, it makes your list manipulation significantly easier. There are ways to implement linked lists without using a header node like this. But this is the easiest and simplest way to do it for minimal overhead (one extra node). I recommend that you also (in your own time) implement the complete functionality required in this lab in a linked list that doesn't use a header node. Compare the pros and cons of either approach and decide/discuss your findings in the forums.

In our approach, we are always guaranteed to find a head node. The head node is always equal to the very first actual node of the list. This is the sentinel and will never change. The user only gets to see what `_head->next` points to and downstream from there.

`_prev_to_current` is always equal to the node immediately *before* what we call the current node (by design). You can think of it as related to the *cursor* in this linked list. It always points to a node whose next element we define as *current*. All list operations are done at the location of this cursor.[2]

[1] Maybe you object, saying that we're hard-coding the sentinel string here. What if the user of our list class wanted to store an actual data item whose value was "_SENTINEL_"? What are elegant ways to handle that situation?

[2] In a previous version of this spec, when p is a pointer, I used to say "p points to". But that made me say things like the "header points to" and this confused students who couldn't tell if I meant `_head->next` or `_head` itself. To avoid this confusion, I've decided to use the word *equals*.

I think one of the best ways you'll get to appreciate the utility of this member is to try and program this quest without using it. This member saves you from having to scan the list from the beginning just to find where "here" is. If you use this member, then "here" (the current location) is always directly in front of your `_prev_to_current` node. This will make a whole lot more sense once you've programmed this functionality both ways. I highly recommend you do so (for your own edification).

We also maintain a tail node pointer. The tail node should always equal the node in the list whose next member is equal to NULL. That is, it is the last data node. There is a subtle interplay between the tail node and the `_prev_to_current` member that will become clear as you implement the following mini quests.

Notice an interesting thing about some of the public list manipulation methods - the ones that return a pointer to the current `String_List` object. Why?

We're doing it because it allows us to program using a very nice pattern:

```
my_list
    .push_back(string_1)
   ->push_back(string_2)
   ->push_back(string_3)
   -> . . .
;
```

In fact, it would be better if you returned a reference to the current object rather than a pointer. Then you can write the even more clean and consistent pattern:

```
my_list
   .push_back(string_1)
   .push_back(string_2)
   .push_back(string_3)
   . ...
;
```

I'll leave you to experiment with the esthetics and find out more. But in this quest, you're gonna have to return a pointer to the current object (essentially **`this`**).

On to the miniquests...

In this spec, when I say the `_prev_to_curr` equals something, I mean that the contents of the pointer, `_prev_to_curr`, equals it. You can think of it as overloading our current understanding of the word equals with an added meaning.

Your first miniquest - The Constructor and Destructor

Implement:

```
String_List::String_List();
```

This should create an empty `String_List` that matches Figure 1. So it's easy rewards. Go for it.

Since your constructor is going to allocate memory on the heap, you must have a destructor or you will suffer memory leaks. Implement

```
String_List::~String_List();
```

I will not be testing your code for memory leaks in this quest, but your very first quest in CS2B involves an extension of this quest in which I WILL check for memory leaks. Make sure you have a destructor which clears out the linked list using a clear method (which you will define in one of the mini quests) and then deletes `_head` (which is what got allocated in the constructor).

Esthetics

A good rule of thumb I use when I have questions about where to deallocate memory is this: If the memory was allocated in the constructor, then it should only be deallocated in the destructor. In general, I try and match up my allocations and deallocations into symmetric locations in the causal chain leading up to some action and back.

Your second miniquest - Insert at current location

Implement:

```
String_List *String_List::insert_at_current(string s);
```

When I invoke it, I will pass it a string argument and it should insert this string at the *current location* of the String_List object. It should leave the current location unchanged. This means that your _prev_to_current member will now end up pointing to this newly inserted Node.

This is a tricky method and is worth getting correct before you move on. Refer to the picture in Figure 2 to see what needs to happen:

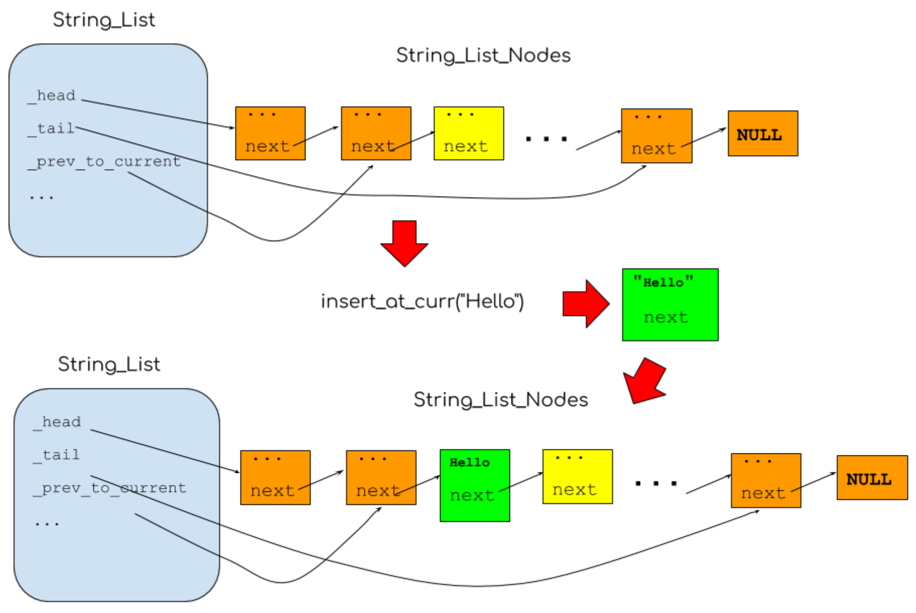


Figure 2. Insert at current location

Your third miniquest - Push back

Implement the public method: `push_back(std::string s);`

This method must insert the string `s` as a brand new data node at the end of the list. That is, this newly allocated `Node` will become the new `_tail`.

Since you have `insert_at_current()` already implemented and working now, simply use it to complete this mini quest. Save the current value of `_prev_to_current`, then set it to your `_tail`. Then insert the given string at the current position (which will now be the tail). Finally restore the value of `_prev_to_current` to your saved value.

Before you implement this, convince yourself that this works by simulating the above sequence of steps on a piece of paper with a pencil.

Your fourth miniquest - Push front

Implement:

```
String_List *String_List::push_front(std::string s);
```

This method must insert the string `s` as a brand new data node at the front of the list. That is, this newly allocated `Node` will become the node that `_head->next` points to.

Important! Remember that _prev_to_current can never be null because it's at least pointing to the list header. The current element visible to the user is always the element AFTER prev_to_current. So when you're advancing curr, and it's pointing to the element before tail, you return null (Why?)

Implement it using the same strategy as for `push_back()`

Your fifth miniquest - Advance current to next

Implement:

```
String_List *String_List::advance_current();
```

If the `_prev_to_current` is the same as the tail node, then obviously you can't advance to the next element (There is none). In that case you would return a null pointer (`nullptr`). Otherwise, make `_prev_to_current` point to whatever it's `next` member was pointing to.

Your sixth miniquest - Get current item

This method is only a few lines, but it is important to understand correctly.

Implement:

```
string String_List::get_current();
```

If the next element exists (that is, your cursor, `_prev_to_current`, is not `NULL`), then simply return its data member.

Otherwise, return the sentinel string you have remembered in your head. (Hmm... I wonder when that kind of thing might happen...)

Suppose I want to use your list, but one of my valid data items is the string "_SENTINEL_", what would I do?

Your seventh miniquest - Remove current item

Implement:

```
String_List *String_List::remove_at_current();
```

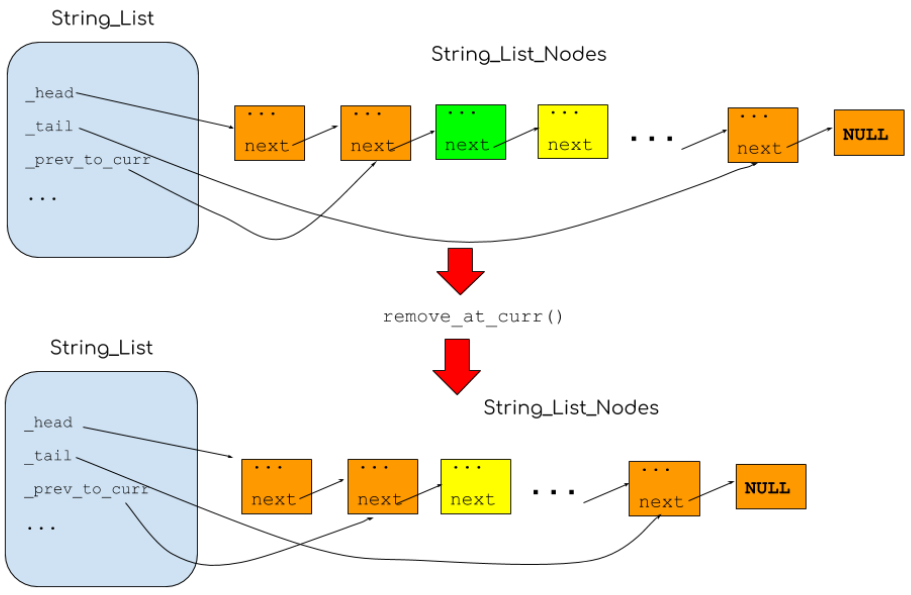


Figure 3. Remove at current location

Again, this is a tricky method and important to get right. Refer to the picture in Figure 3 to decide exactly what needs to happen in your code.

Your eighth miniquest - Get size

Implement:

```
size_t String_List::get_size() const;
```

I can't believe you're gonna get a reward for just reporting the value of size!

Still, I guess this is your last lab this quarter and it's time for a treat.

Your ninth miniquest - Rewind

Implement:

```
String_List *String_List::rewind();
```

This should reset `_prev_to_current` back to the `_head` node.

Your tenth miniquest - Clear

Implement:

```
void String_List::clear();
```

Yet another tricky method to get right. When invoked, it must

1. Iteratively (not recursively) delete all the non-null nodes in the chain starting at _head->next (Talking points: What might the problem be with a recursive delete instead?)
2. Reset _prev_to_current and _tail back to _head
3. Set the value of _head->next to nullptr

Note that you are not deleting the memory associated with the _head node itself. That is the job of the destructor. What the constructor created, only the destructor should undo.

Your eleventh miniquest - Find an item

Implement:

```
string& String_List::find_item(std::string s) const;
```

Note an important aspect of the signature. It returns a reference to a string. Not a copy of a string. If the requested string exists in the list, then what gets returned is a reference to the actual data element. That means that if I assign something to this reference, it will change the contents of the list node that contains that string.

It's important to understand exactly what that means. Please do discuss it in the forums and help each other out whenever possible. Ask me if you're stuck.

What will this method return if the requested string is not found in the list? In that case, return a reference to a static string constant defined within this method. It's value should be "`_SENTINEL_`" Implement it the same way you did for `get_current()`.

Your twelfth miniquest - Stringify

Implement:

```
      string String_List::to_string() const;
```

This method must return a string representation of the list of strings in the following exact format. As usual, I've colored in the spaces. Each gray rectangle stands for exactly one space.

```
# String_List - <N> entries total. Starting at cursor:
<string 1>
<string 2>
<...>
```

The parts in red above (also in angle brackets) must be replaced by you with the appropriate values. By *current elem*, I mean the element that `_prev_to_current` points to.

You would print a maximum of 25 elements this way. If the list has more than 25 strings, you must print first 25 and then print a single line of 3 periods (`...`) in place of the remaining elements.

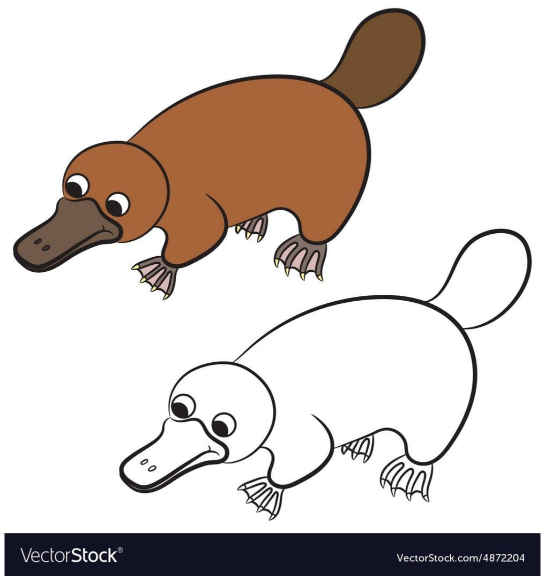
There should be one (and only one) newline after the last line (which may be ellipses).

Review

Now I recommend that you rewind and review the specs one more time making notes along the way. It will help you when you start cutting code.

You may be thinking, "Whew! That's a lot." But really most of these are simple one-liners. If you find yourself coding more than 20 lines for any one of these methods (including comments), then you must seriously look for other signs of over-engineering in your code.

If you're up to it, consider this your real challenge (optional): The complete functionality described in this spec must be implemented in your `String_List` class in under 200 clearly readable lines including comments.



Starter code

You will submit one file: `String_List.h`. Here is `your` starter code

```
// Student ID:
// TODO - Type in your student ID after the colon above.
//  String_List.h
//
#ifndef String_List_h
#define String_List_h

#include <iostream>
#include <sstream>

// Important implementation note: With the exception of to_string(),
// and clear(), all list methods below should operate in a constant amount
// of time regardless of the size of the String_List instance.
//
// The semantics of _prev_to_current is such that it always points to the
// node *BEFORE* the current one. This makes the manipulations easy because
// we can only look forward (and not back) in singly linked lists.
//
// I've included some method header comments below where there's likely to
// be confusion.
//
class String_List {
private:
    struct Node {
        std::string data;
        Node *next;
        Node(std::string s = "") : data(s), next(nullptr) {}
    };

    Node *_head, *_tail, *_prev_to_current;
    size_t _size;

public:
    String_List() {
        // TODO - Your code here
    }

    ~String_List() {
        // TODO - Your code here
    }

    String_List *insert_at_current(std::string s) {
        // TODO - Your code here
    }

    String_List *push_back(std::string s) {
        // TODO - Your code here
    }

    String_List *push_front(std::string s) {
        // TODO - Your code here
    }

    String_List *advance_current() {
        // TODO - Your code here
    }
    std::string get_current() const {
        // TODO - Your code here
    }

    String_List *remove_at_current() {
        // TODO - Your code here
    }

    size_t get_size() const {
        // TODO - Your code here
    }

    String_List *rewind() {
        // TODO - Your code here
    }

    void clear() {
        // TODO - Your code here
    }

    // Find a specific item. Does NOT change cursor.
    //
    // The following returns a reference to the target string if found. But what will
    // you do if you didn't find the requested string? Using sentinel markers is
    // one way to handle that situation. Usually there's only one copy of the
    // sentinel that's global. We will use a local one so it's cleaner with a
    // little more risk (what's the risk?)
    //
    std::string& find_item(std::string s) const {
        // TODO - Your code here
    }

    // Print up to max_lines lines starting at _prev_to_current->next. If the caller
    // wants to print from the beginning of the list, they should rewind() it first.
    //
    std::string to_string() const {
        // TODO - Your code here
    }

    friend class Tests; // Don't remove this line
};

#endif /* String_List_h */
```

Testing your own code

You should test your methods using your own `main()` function in which you try and call your methods in many different ways and cross-check their return value against your expected results. But when you submit you must NOT submit your `main()`. I will use my own and invoke your methods in many creative ways. Hopefully you've thought of all of them.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
1. Enter the secret password for this quest in the box.
2. Drag and drop your `String_List.h` file into the button and press it. (Make sure you're not submitting a `main()` function)
3. Wait for me to complete my tests and report back (usually a minute or less).

Points and Extra Credit Opportunities

Extra credit awaits well-thought out and helpful discussions.

May the best coders win. That may just be all of you.

If everything went according to plan, this should be your last quest of CS2A. Hopefully you aced all of them and have unlocked the password for the first quest in CS2B.

Enjoy your break, and ...

Happy Hacking,

&