Lab: Interface/Implementation Methodology with Linked Lists

Learning Outcomes

- Gain experience with interface/implementation methodology using opaque pointers to encapsulate implementation details from interface
- Gain experience with singly-linked listed data structure
- Gain experience with C++ function overloading and default parameters
- Gain practice in debugging memory leaks and errors using Valgrind
- Reading, understanding, and interpreting C++ code written by others

Overview

Modern software development consists of using software libraries or application programming interfaces (APIs) such as Google Maps API, Facebook API, Twitter API, IBM Watson API, and hundreds of other interfaces to develop new software components. Game development is almost entirely based on interfacing with APIs such as Vulkan, OpenGL, Direct3D, Autodesk Maya, UDK, Unity, Photoshop, and so on. Embedded software developers use a variety of APIs for developing software for smart cars, internet-of-things, and consumer devices such as phones, alarm systems, and refrigerators.

A software library comes in two parts: its *interface* and its *implementation*. The interface specifies *what* the library does while the implementation specifies *how* the library accomplishes the purpose advertised by its interface. Software designers develop libraries and APIs using the concepts of *abstraction* and *encapsulation*. Abstraction specifies the essential features of something that must be made visible through an interface and which information should be hidden as part of the implementation. *Encapsulation* provides techniques for packaging an interface and its implementation in such a way as to hide what should be hidden and to make visible what should be visible. Programming languages provide varying support to the concepts of abstraction and encapsulation. By design C++ provides greater support than C.

In software development, a *client* is a piece of code that uses a software component only through an interface. Indeed, clients *should* only have access to an implementation's object code and no more.

Recall a *data type* is a set of values and the set of operations that can be applied on these values. C++ provides fundamental data types for characters, integers and floating-point numbers. Using structures and classes, new data types can be defined to derive higher-level data types, such as linked lists, trees, lookup tables, and more. We *abstract* a high-level data type using an interface that identifies the legal operations on values of that type while hiding details of the type's representation in an implementation. Ideally, the operations should not reveal representation details on which clients might implicitly depend. Thus, an *abstract data type* or *ADT* is an interface created using data abstraction and encapsulation that defines a high-level data type and operations on values of that type.

Task

> ***Professional software engineers never write code involving linked lists without first reasoning about and debugging their implementations on paper with copious pictures!!!***

The aim of this lab is to apply class lectures on linked lists, abstraction, encapsulation, and interface/implementation methodology to implement a singly-linked list ADT `sllist`. Recall from class lectures that a singly-linked list is a group of dynamically allocated elements, each of type `node`, that are connected to each other through pointers. Although the ultimate goal of an ADT such as `sllist` is to allow clients to encapsulate any kind of data in a `node`, type `node` for this lab will encapsulate only an `int` value.

> ***Make sure to add compilation guards in your header file. Every public name in*** `sllist.hpp` ***and*** `sllist.cpp` ***should be scoped in namespace*** `hlp2` ***to avoid polluting the global namespace. To conserve space, example code will not be encapsulated in namespaces!!!***

The legal operations that clients can perform on `sllist` ADT is specified by the following interface declared in `sllist.hpp`:

```
// forward declaration of
struct node;
struct sllist;

// interface to individual elements of singly-linked list
int         data(node const *p);       // accessor to node's data
void        data(node *p, int newval); // mutator to node's data
node*       next(node *p);       // pointer to successor node
node const* next(node const *p); // pointer to successor node

// interface declarations for singly-linked list...
sllist* construct();
void    destruct(sllist *ptr_sll);
bool    empty(sllist const *ptr_sll);
size_t  size(sllist const *ptr_sll);
void    push_front(sllist *ptr_sll, int value);
void    push_back(sllist *ptr_sll, int value);
void    remove_first(sllist *ptr_sll, int value);
void    insert(sllist *ptr_sll, int value, size_t index);
node*   front(sllist *ptr_sll);
node const* front(sllist const *ptr_sll);
node*   find(sllist const *ptr_sll, int value);
```

How are the representation details of the singly-linked list ADT hidden from clients? The strategy used will be similar to the implementation of `FILE*` presented in `<cstdio>` by the C standard library. The C standard library provides programmers a variety of functions such as `fopen`, `fclose`, `fread`, `fwrite`, and so on to read from and write to files. Initially, function `fopen` must be called to return a value of type `FILE*` that indirectly provides access to a hidden object of type `struct FILE`. However, clients have no idea and no reason to care how the `struct FILE` object is implemented and how file I/O functions manage a file stream by manipulating the state of such objects. Instead, a forward declaration to `FILE` and declarations to file I/O functions are presented in `<stdio.h>`. `struct FILE` and file I/O functions are defined in a source file and clients are then given access to these implementations via an object file [or more traditionally a library file which is a collection of object files].

Likewise, types `node` and `sllist` will not be defined in interface file `sllist.hpp` but will instead be defined in implementation file `sllist.cpp`. Encapsulation is achieved by using a feature of C/C++ that allows a structure to be declared without defining it. Types `node` and `sllist` are declared in interface file `sllist.hpp`, like so:

```
struct node;
struct sllist;
```

These declarations, called *forward declarations*, introduce names `node` and `sllist` as `struct` types into any source file that includes `sllist.hpp`. Since clients never see definitions of types `node` and `sllist` [because they're defined in an inaccessible implementation file `sllist.cpp`], they're *incomplete* types. That is, clients know that `node` and `sllist` are `struct` types but they don't know what members are defined by these types. An incomplete type can be used only in limited ways since the compiler will be unable to deduce the amount of memory that must be reserved when objects of the incomplete type are defined. All clients can do is define pointers or references to such types, and declare [but not define] functions that use an incomplete type as a parameter or a return type. The implementation is hidden because clients can represent a singly-linked list by a value of type `sllist*` or an element in the list by a value of type `node*`. But the pointers say nothing about the data members that comprise structure `sllist` or structure `node`. Hence, `node*` and `sllist*` are called *opaque pointer* types; clients can manipulate such pointers freely, but they can't dereference them; that is, they can't look at the internals of the structure pointed to by them. Only the implementation in `sllist.cpp` has that privilege. This is exactly how the C standard library abstracts and encapsulates the behavior of file streams by only presenting type `FILE*`.

Implementation details

> ***Your implementation can use any header from the C++ standard library except the interface declared in headers `<list>` and `<forward_list>`. Breaking this criterion will be considered a contravention of the Academic Integrity Policy.***

Implementation of `node`

Type `node` is defined in `sllist.cpp`:

```
struct node {
  int value;  // data portion
  node *next; // pointer portion
};
```

The interface declared in `sllist.hpp` is trivially implemented in `sllist.cpp`:

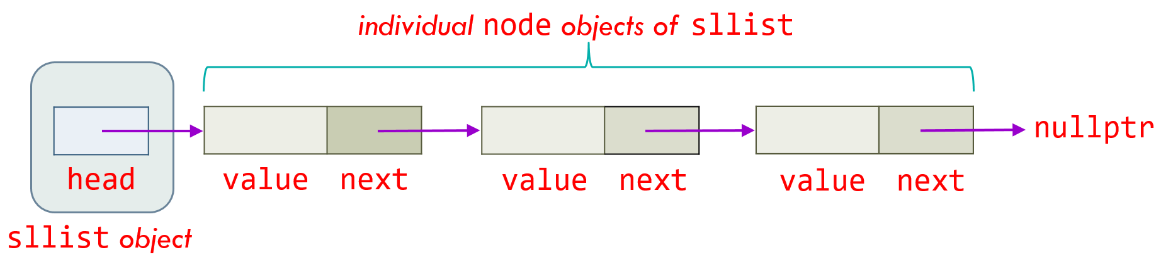
```
int         data(node const *p)        { return p->value; }
void        data(node *p, int newval) { p->value = newval; }
node*       next(node *p)              { return p->next; }
node const* next(node const *p)        { return p->next; }
```

Implementation of `sllist`

Singly-linked list type `sllist` is defined in `sllist.cpp`:

```cpp
struct sllist {
  node *head;
};
```

In class lecture, a plain `node*` was used to point to the first linked list element. In this exercise, we define a `struct` type to contain the `node*` plus additional data members that will be added in future exercises.



`construct()`, `push_front()`, and `size()`

Interface functions `construct`, `push_front`, and `size` are declared in `sllist.hpp`

```cpp
sllist* construct();
void    push_front(sllist *ptr_sll, int value);
size_t  size(sllist const *ptr_sll);
```

What is the purpose of `construct()`? Since clients cannot define objects of type `sllist`, they have to first call `construct()` to create an unnamed object of type `sllist` on the free store:

```cpp
hlp2::sllist *ptr {hlp2::construct()};
```

The purpose of `push_front()` is to insert a new `node` object to the front of the linked list. Using the pointer `ptr` returned by `construct()`, `push_front()` can be used to add elements to the front of the list:

```cpp
hlp2::push_front(ptr, 10);
hlp2::push_front(ptr, 20);
hlp2::push_front(ptr, 30);
```

As expected, `size()` returns the number of elements in the list and be used by clients to determine the number of elements contained in the list:

```cpp
std::cout << hlp2::size(ptr) << "\n";
```

Functions `construct`, `push_front`, and `size` are defined in `sllist.cpp`:

```cpp
// declaration in anonymous namespace of private function that creates
// a node on heap and initializes it with data and pointer to successor
hlp2::node* create_node(int value, hlp2::node *next = nullptr);

```

```
// return initialized sllist object allocated on free store
sllist* construct() {
  return new sllist {nullptr};
}

// add element to front of linked list
void push_front(sllist *ptr_sll, int value) {
  ptr_sll->head = create_node(value, ptr_sll->head);
}

// return number of elements in linked list container
size_t size(sllist const *ptr_sll) {
  size_t cnt {};
  for (node *head = ptr_sll->head; head; head = next(head)) {
    ++cnt;
  }
  return cnt;
}

// define this private function in anonymous namespace!!!
hlp2::node* create_node(int value, hlp2::node* next) {
  return new hlp2::node {value, next};
}
```

Since multiple functions will create `node`s, it is good practice to encapsulate the creation process in a private function `create_node`. This functional decomposition approach provides one well-tested repository for creating `node`s and is an important element in reducing code complexity.

`destruct()`

To avoid memory leaks, the memory allocated to list elements and list object `sllist` must be deallocated by making a call to `destruct()`:

```
hlp2::destruct(ptr); // implement this!!!
```

The definition of `destruct()` is similar to `size()` with two differences:

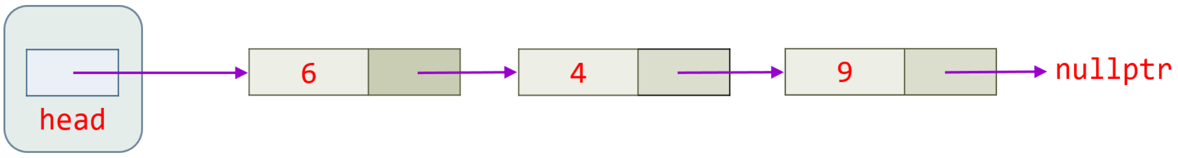
- The code must keep a pointer `p` to the successor of the element being deleted. The element is deleted and traversal continues to the successor node pointed to by `p`.
- After deleting all elements in the list, the object of type `sllist` previously allocated by `construct()` must be deleted.

`empty()`

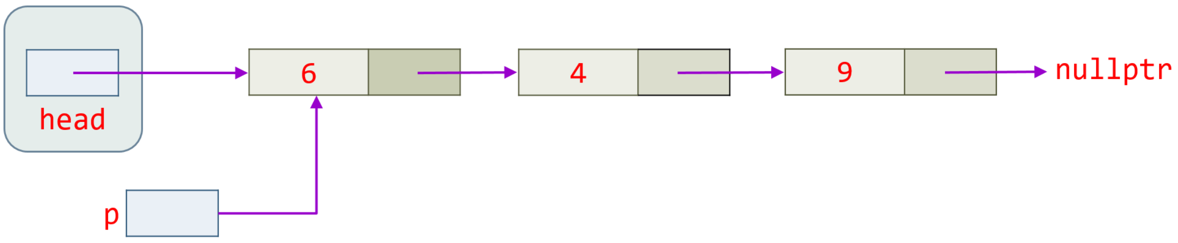
This function declared as `bool empty(sllist const *ptr_sll);` returns `true` if the list pointed to by `ptr_sll` has no elements. The function can be trivially defined by checking data member `head` of the `sllist` object pointed to by `ptr_sll` or by determining there are zero elements by calling companion function `size`.

`remove_first()`

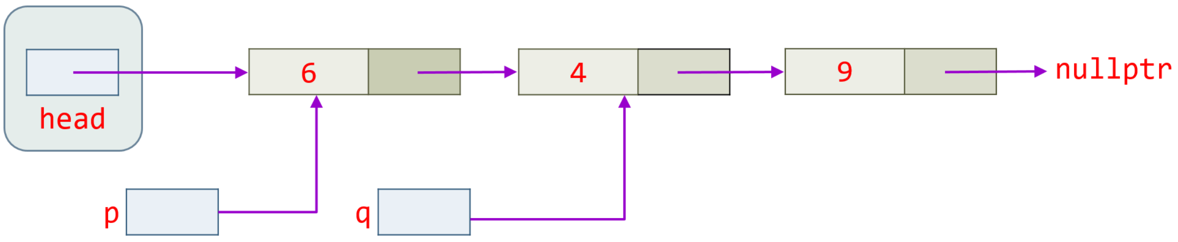
Function `remove_first(sllist *ptr_sll, int val)` deletes the *first* element encountered with the same value as second parameter `val`. Suppose the second argument in a call to `remove_first()` for linked list pictured below is `4`. After this call to `remove_first()` returns, the element with value `6` must point to the element with value `9` followed by deleting the element with value `4`. This requires `remove_first` to identify both the predecessor and successor elements to the element with value `4`.



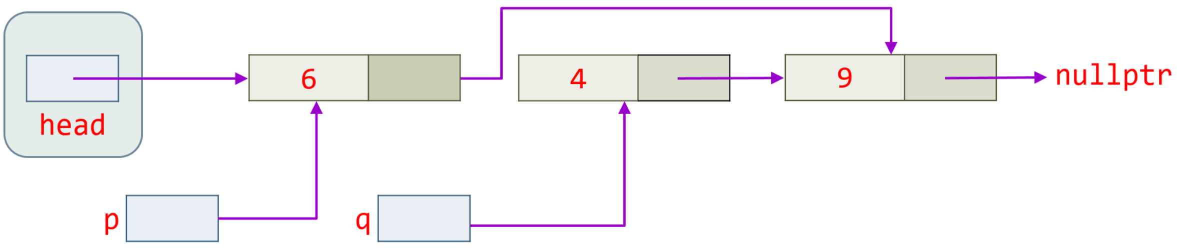
How to implement `remove_first`? First, the function creates a new pointer `p` that points to the same memory address as `head`.



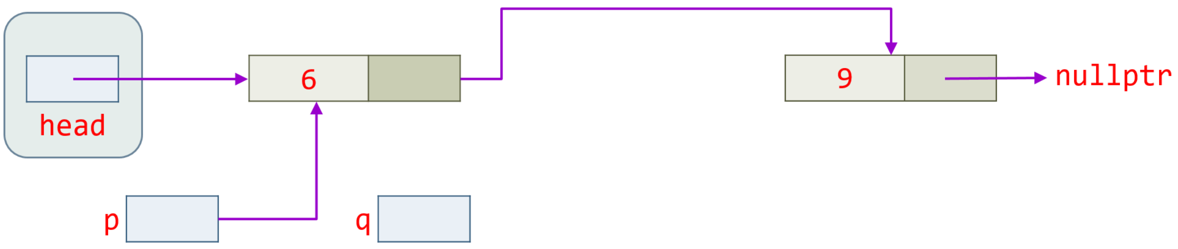
If pointer `p` advances to element with value `4`, it will be impossible to locate the *one-before-removal-element* with value `6`. The trick is to use a second pointer `q` so that `p` and `q` point to consecutive elements with `p` always pointing to the element *before* the element pointed to by `q`. Hence, when `q` points to the removal element, `p` will point to the *one-before-removal-element*.



Once `q` points to the removal element, modify `p->next` to bypass the removal element:



Delete the memory pointed to by `q`:



Edge cases involving empty lists, removing the first element or last element in the list are left to the reader as an exercise.

`push_back()`

Function `push_back(sllist *ptr_sll, int val)` adds a new element with value `val` to the *end* of the list container pointed to by `ptr_sll`, after its current last node. Use the definition of companion function `size` to reach the last element in the list and the definition of private function `create_node` as a guide to implementing `push_back()`.

`front()`

The interface declares two overloads of `front()`

```
node*    front(sllist *ptr_sll);
node const* front(sllist const *ptr_sll);
```

Both these overloads return a pointer to the first element of the list or `nullptr` if the list has zero elements.

`find()`

Function `find` declared as `node* find(sllist *ptr_sll, int value);` returns a pointer to the *first* element in the list pointed to by parameter `ptr_sll` whose data is equal to the second parameter `value`. The implementation of this function is left to the reader as an exercise.

`insert()`

Function `insert` declared as `void insert(sllist *ptr_sll, int value, size_t position);` inserts a new element encapsulating data equal to parameter `value` into the list pointed to by parameter `ptr_sll` at an index specified by parameter `position`. Assume zero-based indexing with the first list element at index `0`, the second element at index `1`, and so on. If `ptr_sll` points to a list with four elements, their indices will range from `0` to `3`. A call `insert(ptr_sll, 11, 2)` will then insert a new element with value `11` after the element with index `1` and before the element with current index `2`. Now, the list will have five elements with indices ranging from `0` to `4`. A subsequent call `insert(ptr_sll, 25, 5)` will therefore insert a new element at index `5` after the current last element [whose index is `4`]. Now the list has six elements with indices ranging from `0` to `5`. A subsequent call `insert(ptr_sll, -5, 30)` will add a new element after the current last element in the list [since the third argument in the call specifies an index `30` that doesn't exist in the current list].

> ***Allocation and deallocation of heap memory when implementing linked lists is a complex process. Use Valgrind frequently and often to check for both memory leaks and memory errors.***

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Header and source files

Submit interface file `sllist.hpp` and implementation file `sllist.cpp`.

Compiling, executing, and testing

Download `sllist-driver.cpp`, `makefile`, and correct output file `output.txt` generated by the unit tests. Run `make` with the default rule [`$ make`] to bring program executable `sllist.out` up to date. Or, directly test your implementation by running `make` with target `test`: `$ make test`. If the `diff` command in the `test` rule is not silent, then one or more of your function definitions is incorrect and will require further work.

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - F grade if your submission doesn't compile with the full suite of `g++` options.
   - F grade if your submission doesn't link to create an executable.
   - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will assign 50% of the grade based on the input and output files given to you. The remaining 50% of the grade will be awarded based on the additional tests implemented by the auto grader.
   - The auto grade will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if your output matches correct output of auto grader.
   - A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.