Assignment - Interface and Implementation Using Opaque Pointers

Topics and references

- Doubly linked lists.
- Interface and implementation methodology.
- Valgrind to debug memory leaks and errors.

Learning Outcomes

- Practice interface/implementation methodology using opaque pointers to encapsulate implementation details from interface.
- Practice programming linked-lists.
- Gain practice in debugging memory leaks and errors using Valgrind.
- Read and understand code

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 that a data type is a set of values and the set of operations that can be applied on these values. In C++, built-in data types include characters, integers, and floating-point numbers. Structures and classes in C++ define new data types and can be used to form 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 the 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

In this exercise, use interface-implementation separation to complete a doubly linked list ADT. The only legal operations that clients can perform on the doubly linked list type is specified by the following interface declared in `dllist.h`:

```
dllist* construct_list(); // IMPLEMENTED
void destruct_list(dllist *ptr_dll);
size_t size(dllist const* ptr_dll);
void push_front(dllist *ptr_dll, int value); // IMPLEMENTED
void push_back(dllist *ptr_dll, int value);
void print(dllist const* ptr_dll); // IMPLEMENTED
node* find(dllist const* ptr_dll, int value);
void remove_first(dllist *ptr_dll, int value);
void insert(dllist *ptr_dll, int value, size_t position);
```

Functions marked `IMPLEMENTED` are already implemented. You've to implement the other functions in `dllist.cpp`.

How are the representation details of the doubly linked list type hidden from clients? The strategy used will be similar to the implementation of `FILE*` in the C standard library. The doubly linked list ADT is specified by type `dllist` that is not defined in interface file `dllist.h` but is instead defined in implementation file `dllist.cpp`. Encapsulation is achieved by using a feature of C/C++ that allows a structure to be declared without defining it. Such a declaration of the doubly linked list type is added to interface file `dllist.h`:

```
struct dllist;
```

This declaration, called a 'forward declaration', introduces name `dllist` into the program and indicates that `dllist` refers to a `struct` type. Since clients never see the definition of `dllist` (because it is defined in an inaccessible implementation file `dllist.cpp`), it is an incomplete type - clients know that `dllist` is a `struct` type but they don't know what members that type contains. An incomplete type can be used in only limited ways since the compiler is unable to deduce the amount of memory that must be put aside when objects of the incomplete type are defined: we can define pointers or references to such types, and we can 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 doubly linked list by a pointer to structure `dllist*` but the pointer says nothing about what structure `dllist` looks like. `dllist*` is called an opaque pointer type; 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 `dllist.cpp` has that privilege.

Implementation Details

The doubly linked list type `dllist` and the `node` type are defined in `dllist.cpp`:

```cpp
struct node {
    node *prev; // pointer to previous node
    int value; // data portion
    node *next; // pointer to next node
};
struct dllist {
    node *head, *tail;
};
```

The `struct` type called `dllist` contains the `node*` pointers plus possibly additional data members such as `size` that may be added at a later date. The use of type `dllist` is illustrated in the following code fragment that defines functions `push_front` and `print`:

```cpp
// allocate and initialize dllist object on free store
dllist* construct_list() {
    return new dllist {nullptr, nullptr};
}
// add element to front of linked list
void push_front(dllist* ptr_dll, int value) {
    if (!ptr_dll)
        return;
    ptr_dll->head = construct_node(value, nullptr, ptr_dll->head);
    node* l_pHead = ptr_dll->head;
    node* l_pNx = l_pHead->next;
    if (l_pNx)
        l_pNx->prev = l_pHead;
    if (!ptr_dll->tail)
        ptr_dll->tail = l_pHead;
}
// visit each element and print its value
void print(dllist const *ptr_dll) {
    node const *head = ptr_dll->head;
    while (head) {
        std::cout << head->value << " ";
        head = head->next;
    }
    std::cout << "\n";
}
```

Since clients cannot define objects of type `dllist`, they have to first call function `construct_list` to construct an unnamed object of type `dllist` on the free store:

```cpp
hlp2::dllist *list = hlp2::construct_list();
```

Using pointer `list`, the client can add elements to the list and then traverse each element in the list:

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

causing the following values to be written to the free store:

```
30 20 10
```

To avoid memory leaks, the memory allocated to the list elements and the list object `dllist` must be deallocated by making a call to function `destruct_list`:

```
// you must implement this function!!!
hlp2::destruct_list(list);
```

The definition of function `destruct_list` is similar to print with two notable differences:

- The code must keep a pointer `p` for the element after the element to be deleted. The element is deleted and using `p`, the traversal continues to the element pointed to by `p`.
- After deleting all the elements in the list, the object of type `dllist` that was allocated by `construct_list` must be deleted.

The function `remove_first(dllist *ptr_dll, int val)` deletes the first element encountered with the same value as the second parameter. First, the function should check if the list is empty in which case, there is nothing to remove.

Second, if `head`'s value (`ptr_dll->head->value`) is the same as `val`, the first element is removed by updating as needed `head`, `tail`, and `prev` in the second element if the second element is present.

In the third scenario, suppose the second of three elements in the list has `value` equal to `val`. After `remove_first` has run, the first and third element must remain in the list. The first element's `next` must point to the last element, and the last element's `prev` must point to the first.

As for removing the tail element, this is similar to removing the head element.

`insert` is left as an exercise for you to work out.

Submission Details

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

Header file

There is no header to submit.

Source file

A partially filled implementation file is provided. Complete the necessary definitions and upload to the submission server. Remember, to frequently check your code for memory leaks and errors with Valgrind.

Compiling, executing, and testing

Download `dllist-driver.cpp`, `makefile`, and a correct input file `output.txt` for the unit tests in the driver. Run make with the default rule to bring program executable `dllist.out` up to date:

```
$ make
```

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 documentation

Every source and header file *must* begin with a *file-level* documentation block. This documentation serves the purpose of providing a reader the raison d'être of this source file at some later point of time (could be days or weeks or months or even years later). This module will use Doxygen to tag source and header files for generating html-based documentation. An introduction to Doxygen and a configuration file is provided on the module web page. Here is a sample for a C++ source file:

```
/*!*****************************************************************************
\file    scantext.cpp
\author  Prasanna Ghali
\par     DP email: pghali\@digipen.edu
\par     Course: CSD1170
\par     Section: A
\par     Programming Assignment #6
\date    11-30-2018

\brief
  This program takes strings and performs various tasks on them via several
  separate functions. The functions include:

  - mystrlen
      Calculates the length (in characters) of a given string.

  - count_tabs
      Takes a string and counts the amount of tabs within.

  - substitute_char
      Takes a string and replaces every instance of a certain character with
      another given character.

  - calculate_lengths
      Calculates the length (in characters) of a given string first with tabs,
      and again after tabs have been converted to a given amount of spaces.

  - count_words
```

```
    Takes a string and counts the amount of words inside.
********************************************************************************
***/
```

Function-level documentation

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. In team-based projects, this information is crucial for every team member to quickly grasp the details necessary to efficiently use, maintain, and debug the function. Certain details that programmers find useful include: what does the function take as input, what is the output, a sample output for some example input data, how the function implements its task, and importantly any special considerations that the author has taken into account in implementing the function. Although beginner programmers might feel that these details are unnecessary and are an overkill for assignments, they have been shown to save considerable time and effort in both academic and professional settings. Humans are prone to quickly forget details and good function-level documentation provides continuity for developers by acting as a repository for information related to the function. Otherwise, the developer will have to unnecessarily invest time in recalling and remembering undocumented details and assumptions each time the function is debugged or extended to incorporate additional features. Here is a sample for function `substitute_char`:

```
/*!******************************************************************************
****
\brief
   Replaces each instance of a given character in a string with
   other given characters.

\param string
  The string to walk through and replace characters in.

\param old_char
  The original character that will be replaced in the string.

\param new_char
  The character used to replace the old characters

\return
  The number of characters changed in the string.
********************************************************************************
***/
```

Since you are to submit `q.cpp`, add the documentation specified above to it.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit `q.cpp`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - `F` grade if your `q.cpp` doesn't compile with the full suite of `g++` options.
   - `F` grade if your `q.cpp` 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 provide a proportional grade based on how many incorrect results were generated by your submission. `A+` grade if output of function matches correct output of auto grader.
- A deduction of one letter grade for each missing documentation block in `q.cpp`. Your submission `q.cpp` must have **one** file-level documentation block and at least **one** function-level documentation block. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block may 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 may be later reduced from `A+` to `B+`. Another example: if the automatic grader gave your submission a `C` grade and the two documentation blocks are missing, your grade may be later reduced from `C` to `E`.