Assignment: Doubly Linked List - Operator Overload

Topics and references

- Operator overload.
- Friend functions.

Learning Outcomes

- Practice operator overloading.
- Learn to work with friend functions.
- Practice programming linked-lists.

Overview

Operator overloading is a major feature of C++ that is handy and makes for easier manipulation of objects and code readability. This assignment is a chance to practice programming member and non-member operator overloads.

Task

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

```
namespace hlp2 {
    class dllist
    {
    public:
        struct node
        {
            node* prev;
            int value;  // data portion
            node* next;
        };

        dllist();

        // Copy ctor
        dllist(dllist const&);

        ~dllist();

        // Copy assignment
        dllist& operator=(dllist const&);

        // If data is the same and in the same order, return true
        friend bool operator==(dllist const&, dllist const&);

        // Add the data of the right dllist to the end of the left dllist,
        // preserving the order of the data
        friend dllist operator+(dllist const&, dllist const&);

        // Add the data of the right dllist to the end of this dllist,
        // preserving theorder of the data and returning a reference to self
        dllist& operator+=(dllist const&);

        // Return a dllist with data in same order but negated
        dllist const operator-() const;

        // Pop back node and return reference to self
        dllist& operator--();

        // Copy self, pop back node, and return copy
        dllist const operator--(int);

        // Return data of node using input parameter as index where 0 is the
        // head
        int& operator[](size_t);
        int const& operator[](size_t) const;

        // Print using print()
        friend std::ostream& operator<<(std::ostream&, dllist const&);

        // Return the count of elements
        size_t size() const;

        // Add a new value to the beginning
        void push_front(int value);

        // Remove the front node
        void pop_front();

        // Add a new value to the end
        void push_back(int value);

        // Print the contents - IMPLEMENTED
        void print() const;

        // Find first occurrence
        node* find(int value) const;

        // Insert value in linked list at index.
        void insert(int value, size_t position);

        // Remove the first element in list with value
        void remove_first(int value);

    private:
        node* head;
        node* tail;
    };

    // If data of the dllists is different in terms of value, sequence, or
    // number of items, return false
    bool operator!=(dllist const&, dllist const&);
}
```

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

Implementation Details

Implement the operators in `dllist` so they behave as specified above.

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 output 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
```

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 `dllist.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 `dllist.cpp`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - `F` grade if your `dllist.cpp` doesn't compile with the full suite of `g++` options.
    - `F` grade if your `dllist.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 `dllist.cpp`. Your submission `dllist.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`.