Assignment - Data Containers: Abstract Classes

Topics and references

- Abstract base classes.
- Pure virtual functions.

Learning Outcomes

- Practice object oriented design with abstract classes.
- Practice programming dynamic polymorphism with pure virtual functions.

Overview

Run-time a.k.a. dynamic polymorphism is a very useful feature of object-oriented languages. In C++, this is achieved using class inheritance and virtual functions.

A special type of virtual function is the pure virtual function that is declared with "`= 0`" right after the function signature in the declaration as shown below for `cont`. A class with such a function cannot have objects of the class created and is known as an abstract class.

The pure virtual function must be overridden in a sub-class for objects of the sub-class to be instantiated, otherwise the sub-class would also be abstract. The task here is a chance to experience programming polymorphism with an abstract class.

Task

In this exercise, define the base class `cont` of container classes `sllist` and `bits.` Use pure virtual functions in `cont` and place the definition in a file called `cont.h`.

Also update `list.cpp` from the last lab task so that the source code is compiled and run correctly by the `makefile`. Definitions of `sllist` and `bits` are below:

```
namespace hlp2 {
    class sllist : public cont
    {
    public:
        struct slnode
        {
            slnode(int = 0, slnode* = nullptr);
            virtual ~slnode() = default;
            int value;
            slnode* next;
        };
        sllist();
        sllist(sllist const&);
        virtual ~sllist();
        sllist& operator=(sllist const&);
        size_t size() const override;
```

```
        virtual void push_front(int value);
        virtual void pop_front();
        virtual void push_back(int value);
        void clear() override;
        void print() const;
        slnode* find(int value) const;
        virtual void insert(int value, size_t position);
        virtual void remove_first(int value);
    protected:
        slnode* head;
    };

    class dllist : public sllist
    {
    // dllist definition as before
    };
}
```

```
namespace hlp2 {
    class bits : public cont
    {
    public:
        bits(size_t = 100);
        bits(bits const& rhs);
        ~bits();
        bits& operator=(bits const&);
        size_t size() const override;
        void clear() override;
    private:
        size_t siz = 100;
        bool* ar = nullptr;
    };
}
```

A partial definition of `cont` is given:

```
namespace hlp2 {
    class cont
    {
    public:
        // Other declarations

        virtual size_t size() const = 0; // Pure virtual function
    };
}
```

Implementation Details

Examine `sllist`, `bits` and `list-driver.cpp` to see what should be in `cont`. Use pure virtual functions where appropriate.

Submission Details

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

Header file

Submit the `cont` definition in `cont.h`.

Source file

Update `list.cpp` from last week's lab task to work with this week's code. 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 `cont.h`, `list.h`, `bits.h`, `bits.cpp`, `list-driver.cpp`, and `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 `list.out` up to date:

```
$ make
```

You may 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
```

```
16    The number of characters changed in the string.
17  ********************************************************************************
    ***/
```

Since you are to submit `list.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 `cont.h` and `list.cpp`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - `F` grade if your `cont.h` and `list.cpp` doesn't compile with the full suite of `g++` options.
    - `F` grade if your `cont.h` and `list.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 `list.cpp`. Your submission `list.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`.