Assignment: Cloning Class `std::bitset`

Topics

- Data abstraction and encapsulation using classes.
- Class templates.
- Bitwise operators

Learning Outcomes

- Demonstrate ability to apply data abstraction and encapsulation techniques using classes.
- Demonstrate ability to apply generic programming techniques using class templates.
- Practice use of bitwise operators and related conversion rules.

Task and Implementation Details

The C++ standard library defines class `bitset` to model fixed-sized arrays of bits that are useful for managing sets of flags. C and old C++ programs usually use type `unsigned long` for arrays of bits and manipulate them with bit operators `<<`, `>>`, `|`, `&`, `^`, and `~`. Class `bitset` has the advantage that `bitset`s may contain any number of bits, and provide a variety of additional operations. Bits can be manipulated individually through member functions such as:

```
// set bit at position pos to value
set(size_t pos, bool value = true);
// clear bit at position pos by setting its value to false
reset(size_t pos);
// flip bit value at position pos
flip(size_t pos);
```

Your task is to implement a clone of class `bitset`. Begin by getting familiar with the documentation describing the interface of class `bitset`. C++ Primer (login through either your DIT or SIT credentials) presents a nice overview of `bitset`, so does Microsoft's C++ standard library documentation, and if you're a glutton for reading, there's more examples here.

These are the requirements for your submission:

1. Source will be complied with C++17 standard.
2. Provide the interface in header file `bitset.h`, the implementation in file `bitset.hpp`, and then include `bitset.hpp` at the bottom of `bitset.h`.

   > ***It is not necessary for both submitted files to include any header files. Therefore don't. If you do, the grader will refuse to compile your submissions.***

3. Study the driver carefully. Even though class `bitset` provides a large interface, you're only required to define the much smaller interface required by the driver. The declarations of these interface (member) functions must match the standard's in terms of parameter types, count, and return values. The online documentation will be your source for these declarations.

4. It is not known how the standard library implements class `bitset`. However, every instance of your class must have memory size equivalent to expression `sizeof(void*)`. The data must be stored in a dynamically allocated array of the smallest possible size sufficient to contain `N` bits (counted from `0` to `N-1`).
5. Your implementation must be cross-platform: it must work even on a platform with a non-standard number of bits in a byte. This means you must use the `CHAR_BIT` macro.
6. Use bitwise operators for setting, clearing, flipping and testing individual bits in bytes.
7. If you compile with preprocessor macro definition `USE_STL_BITSET`, you can build and test the driver with the standard library's implementation of class `bitset` for the first 13 tests.
8. Pay attention to encapsulation, `const` -correctness, avoid shallow copy, and memory leaks.

Submission Details

Header and source files

You will submit `bitset.h` and `bitset.hpp`.

> ***Neither submitted files needs to include any header files. Therefore don't. If you do, the grader will refuse to compile your submissions.***

Compiling, executing, and testing

Download driver source `driver-bitset.cpp` with a variety of unit tests. Refactor a `Makefile` from previous assignments for use with this assignment.

Documentation is required

This module will use Doxygen to tag source and header files for generating html-based documentation. Every source and header file *must* begin with *file-level* documentation block. 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. The course web page provides more coverage on this topic.

Valgrind is required

Since your code is interacting directly with physical memory, things can go wrong at the slightest provocation. The range of problems that can arise when writing code dealing with low-level details has been covered in HLP1 and HLP2 lectures. You should use Valgrind to detect any potential issues that may lurk under the surface. The course web page provides more coverage on this topic.

Submission and Grading Rubrics

1. In the course web page, click on the appropriate submission page to submit necessary files.
2. F grade if your submission doesn't compile with the full suite of `g++` options or if your submission compiles but doesn't link to create an executable.
3. F grade if Valgrind reports memory errors and/or leaks.
4. For this assignment, all 21 unit tests must pass correctly to get an $A+$ grade; otherwise the grade will be F.
5. A deduction of one letter grade for each missing documentation block in your submission(s). Every source or header file you submit must have **one** file-level documentation block and

function-level documentation blocks for ***every function you're defining***. 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 E.