


# Chapter VI

## Exercise 02: Mutated abomination

	Exercise : 02
Mutated abomination	
Turn-in directory : <i>ex02/</i>	
Files to turn in : <code>Makefile</code> , <code>main.cpp</code> , <code>MutantStack.{h, hpp}</code> and optional file: <code>MutantStack.hpp</code>	
Forbidden functions : <b>None</b>	

Now, time to move on more serious things. Let's develop something weird.

The `std::stack` container is very nice. Unfortunately, it is one of the only STL Containers that is NOT iterable. That's too bad.

But why would we accept this? Especially if we can take the liberty of butchering the original stack to create missing features.

To repair this injustice, you have to make the `std::stack` container iterable.

Write a **MutantStack** class. It will **be implemented in terms of** a `std::stack`. It will offer all its member functions, plus an additional feature: **iterators**.

Of course, you will write and turn in your own tests to ensure everything works as expected.

Find a test example below.

```
int main()
{
    MutantStack<int>    mstack;

    mstack.push(5);
    mstack.push(17);

    std::cout << mstack.top() << std::endl;

    mstack.pop();

    std::cout << mstack.size() << std::endl;

    mstack.push(3);
    mstack.push(5);
    mstack.push(737);
    //[...]
    mstack.push(0);

    MutantStack<int>::iterator it = mstack.begin();
    MutantStack<int>::iterator ite = mstack.end();

    ++it;
    --it;
    while (it != ite)
    {
        std::cout << *it << std::endl;
        ++it;
    }
    std::stack<int> s(mstack);
    return 0;
}
```

If you run it a first time with your `MutantStack`, and a second time replacing the `MutantStack` with, for example, a `std::list`, the two outputs should be the same. Of course, when testing another container, update the code below with the corresponding member functions (`push()` can become `push_back()`).