Problem 1: C++ Linked List Library (cpplist)

Your job is now to refactor your C code from the second assignment and produce a much more flexible library with similar functionality—and this time in C++. Download the zipped folder provided in the file cpplist.zip as a basis of your program and take a look at the existing code.

- **Write your implementation code** in .cpp files in the src/ directory; feel free to add more header files as needed in the include/ directory.
- **GRADER_INFO.txt** is a file for the grader (don’t edit!) containing PROG: cpplist, LANG: C++
- **As before**: you should submit a zipped folder using the same directory structure as the provided zip file. We’ve added a section in the Makefile for your convenience: if you type make zip in the same folder as your project, a zip file containing all of your code and the required headers will be constructed in the project directory and you can upload that to the grader.

We are provided a header file describing the interface for the List data structure. Look in the file list.h to find the functionality required of the other functions, which you will write.

```cpp
// Forward declaration of apply/reduce types
class ApplyFunction;
class ReduceFunction;

class List {
    // ... put whatever private data members you need here
    // can also add any private member functions you’d like

    public:
    List();
    ~List();
    size_t length() const;
    int& value( size_t pos );
    int value( size_t pos ) const;
    void append( int value );
    void deleteAll( int value );
    void insertBefore( int value, int before );
    void apply( const ApplyFunction &interface );
    int reduce( const ReduceFunction &interface ) const;
    void print() const;
};
// ... etc
```

Some questions to ask yourself for understanding:

- Why is there a function int& value( size_t pos ); as well as a function int value( size_t pos ) const?
- What is a forward declaration?
Why don’t we include the headers \texttt{apply.h} and \texttt{reduce.h} here?

You’ll find those two other header files containing definitions for your \texttt{ApplyFunction} and \texttt{ReduceFunction} classes, shown here:

```cpp
#include "list.h"

class ReduceFunction {
protected:
    virtual int function(int x, int y) const = 0;
public:
    int reduce(const List &list) const;
    virtual int identity() const = 0;
    virtual ~ReduceFunction() {};
};

// An example ReduceFunction
class SumReduce : public ReduceFunction {
    int function(int x, int y) const;
public:
    SumReduce() {};
    ~SumReduce() {};
    int identity() const { return 0; }
};
```

and then in the source code file:

```cpp
#include "list.h"
#include "reduce.h"

// This works fine, but iterating over a list like this is fairly slow. See if you can speed it up!
int ReduceFunction::reduce(const List &list) const {
    int result = identity();
    for (size_t p = 0; p < list.length(); ++p) {
        result = function(result, list.value(p));
    }
    return result;
}

int SumReduce::function(int x, int y) const {
    return x + y;
}
```
Input/Output Format

Not applicable; your library will be compiled into a testing suite, your implemented functions will be called by the program, and the behavior checked for correctness. For example, here is a potential test:

```c++
#include "list.h"

int main() {
    int N = 5;
    // you’ll need to write a copy constructor
    // to be able to do this (see Lecture 5)
    auto list = List{};

    for (int i = 0; i < N; ++i) {
        list.append(i);
    }

    list.print();
    return 0;
}
```

Upon calling this function, the code outputs

```plaintext
{ 0 -> 1 -> 2 -> 3 -> 4 }
```

or whatever your formatted output from `list.print()` is made to look like. You are strongly encouraged to write your own tests in `test.cpp` so that you can try out your implementation code before submitting it to the online grader.

Best Practices

The problem is only worth 500/1000 points when you submit; the rest of the grade will be based on how well your code follows C++ best practices and object-oriented programming principles. See a list of those [here](#). The rubric for the other 500 points is as follows.

- +500 points: Code is eminently readable, follows best practices, highly efficient, well structured, and extensible.
- +400 points: Code is easy to follow, only a few small violations of best practices, and extensible.
- +300 points: A decent refactoring effort, no egregiously bad practices, might be difficult to extend.
- +200 points: Some refactoring effort, lots of violations of best practices, not very extensible
- +100 points: Minor refactorings/improvements, little effort to follow best practices.
- +0 points: No effort to refactor or improve code (basically direct copy of HW#2)