Unordered List implementation using Templates

Onat Bas, SWE501, Assn 3

Implementation

This project is intended to provide an implementation of unordered lists with templates, and it is intended to be as small as a single header file, to be portable.

List class

Lists class has some nice overloads to be easy to use. An example to these is << operator. Similar to the usage if ofstream classes, << is used in this list implementation, tis time to append (or push) elements to the list content

Memory Management

Memory allocation in List class is managed by standard C operations, malloc, free etc. Standard C operations are used because operations such as realloc guarantee that there won't be any data loss under unsuccessful or successful operations. Allocated memory is always allocated in powers of two. If the size required for the next item exceeds the limits of the current size, simply the buffer size is doubled to make up more space. Vice versa, if some elements are removed from the list and at least half of the list is empty/not used, this portion of the memory is given back to the os by simply diving the allocated mamory in half.

```
// In this example I create a new list of a custom struct called "Student"
// and add 5 students in advance and remove them one by one to
// monitor the behaviour of the memory allocation algorithm.
Onats-MacBook-Pro:asgn_idk onatbas$ ./a.out
Size of l: 0
Storage size: 1
Size of l: 5
Storage size: 8
Name of 2nd student is : (Beth Ruh).
Removed first student. Size of l: 4
Storage size: 8
Name of first student is : (James Immelt).
Removed first student. Size of l: 3
Storage size: 4
Name of first student is : (Bowden Graham).
Removed first student. Size of l: 2
Storage size: 4
Name of first student is : (Martin Wise).
Removed first student. Size of l: 1
Storage size: 2
Name of first student is : (Beth Ruh).
Removed first student. Size of l: 0
Storage size: 1
```

Conclusion

Effectiveness

Even though the memory is allocated dynamically, due the he memory being a continuous single block, user gets the benefit of having random access with this structure. Also having overloading of operator[] gives the user the feel of right usage.

Header-Only

Since the code required to use this type of class is template-based, it has to be implemented in header files, mostly. This, in this case, brings the advantage of being lightweight and high-portability. There's no linking or no dependendy.

Result

Usage:

```
$ ./sampler

usage: ./sampler "birth_count" "population_limit"
```

Sourcecode:

Main

main.cxx

```
#include "List.hxx"
#include
#include
#include

struct Student
{
        char name[20];
        int id;
        int year, month, date; //of birth

        Student& operator=(const Student& other) // copy assignment
        {
                memcpy(name, other.name, 20);
                id = other.id;
                year = other.year;
                month = other.month;
                date = other.date;
        }

};

Student MakeStudent(
        char* name,
        int id,
        int year, int month, int date)
{
        Student s;
        memcpy(s.name, name, 20);
        s.id = id;
        s.year = year;
        s.month = month;
        s.date = date;
        return s;
}
```

```
using namespace std;
int main(int argc, char const *argv[])
{
        List l;
        cout << "Size of l: " << l.getsize() << endl;
        cout << "Storage size: " << l.getbufsize() << endl;

        l << MakeStudent("John Doe", 201512345, 1990, 10, 1) <<
                MakeStudent("Beth Ruh", 120243434, 1980, 3, 27) <<
                MakeStudent("Martin Wise", 120243434, 1980, 3, 27) <<
                MakeStudent("Bowden Graham", 120243434, 1980, 3, 27) <<
                MakeStudent("James Immelt", 122448776, 1988, 8, 7);

        cout << "Size of l: " << l.getsize() << endl;
        cout << "Storage size: " << l.getbufsize() << endl;
        cout << "Name of 2nd student is : (" << l[1].name << ")." << endl;

        int i = 5;
        while (i-->0)
        {
                l.remove(0);
                cout << "Removed first student. Size of l: " << l.getsize() << endl;
                cout << "Storage size: " << l.getbufsize() << endl;
                cout << "Name of first student is : (" << l[0].name << ")." << endl;

        }
        return 0;
}
```

List.hxx

```
        #pragma once
#include

template
class List
{
public:
        List(){
                members = 0;
                size = 0;
                bufferSize = 1;
                members = (T*)malloc(16*sizeof(T));
        }

        ~List(){
                delete [] members;
        }

   T& operator[](int index)
   {
                return get(index);
   }

   T& get(int index)
   {
                if (index >= size)
                        throw "Out of boundaries";
                return members[index];
   }

   int getsize()
   {
                return size;
   }

   int getbufsize()
   {
                return bufferSize;
   }
```

```
    List& operator<<(const T &value)
    {
                this->push(value);
                return *this;
    }

    int push(const T& value)
    {
                while (size >= bufferSize)
                {
                        bufferSize *= 2;
                        members = (T*)realloc(members, bufferSize*sizeof(T));
                }

                members[size] = value;
                return size++;
    }

    void remove(int index)
    {
                if (index < 0 || index >= size || size <= 0)
                        return;


                if (--size != 0)
                        members[index] = members[size];

                bool reallocate = false;
                while (size < bufferSize/2)
                {
                        bufferSize /= 2;
                        reallocate = true;
                }

                if (reallocate)
                        members = (T*)realloc(members, bufferSize*sizeof(T));
    }


private:
        int size;
        int bufferSize;
        T* members;
};
```