```
/* ---------------------------------------------------------------
// String.h
// ------------------------------------------------------------ */
#ifndef GROUP_STRING
#define GROUP_STRING
#include <iostream>
#include <string.h>

class String{
	friend std::istream &operator>> (std::istream &is, String &s);
public:
	String();
	String(const String &s);
	String(const char* s);
	~String();
	size_t size() const;
	const char* c_str() const;
	const char &operator[] (const size_t i) const;
	char &operator[] (const size_t i);
	String &operator+= (const String &s);
	String &operator+= (const char* s);
	String &operator+= (char);
	void clear();
	String &operator= (String s);
	String &operator= (char* s);
	String &operator= (char s);
	String &swap (String &rhs);
	String &swap (char* &rhs);
	void append (const String &s);
	void append (const char* s);
	void append (char s);
private:
	size_t size_ = 0;
	char* str_ = nullptr;
};
bool operator== (const String &lhs, const String &rhs);
bool operator== (const char* lhs, const String &rhs);
bool operator== (const String &lhs, const char* rhs);
bool operator!= (const String& lhs, const String& rhs);
bool operator!= (const char* lhs, const String &rhs);
bool operator!= (const String &lhs, const char* rhs);
bool operator<    (const String& lhs, const String& rhs);
bool operator< (const char* lhs, const String &rhs);
bool operator< (const String &lhs, const char* rhs);
bool operator<= (const String& lhs, const String& rhs);
bool operator<= (const char* lhs, const String &rhs);
bool operator<= (const String &lhs, const char* rhs);
bool operator>    (const String& lhs, const String& rhs);
bool operator> (const char* lhs, const String &rhs);
bool operator> (const String &lhs, const char* rhs);
bool operator>= (const String& lhs, const String& rhs);
bool operator>= (const char* lhs, const String &rhs);
bool operator>= (const String &lhs, const char* rhs);
std::ostream &operator<< (std::ostream &os, const String &s);
String operator+ (const String &lhs, const String &rhs);
String operator+ (const char* lhs, const String &rhs);
String operator+ (const String &lhs, const char* rhs);
String operator+ (const String &lhs, char rhs);
String operator+ (char lhs, const String &rhs);
```

```
void swap (String &lhs, String &rhs);
void swap (char* &lhs, String &rhs);
void swap (String &lhs, char* &rhs);
#endif

/* -----------------------------------------------------------
// String.cpp
// ---------------------------------------------------------- */
#include "String.h"
String::String() {
	size_ = 0;
	str_ = new char[1];
	str_[0] = '\0';
}
String::String (const String &s) {
	size_ = s.size();
	str_ = new char[size() + 1];
	for (size_t i = 0; i < size(); ++i) {
		str_[i] = s.str_[i];
	}
	str_[size()] = '\0';
}
String::String (const char* s) {
	size_ = strlen(s);
	str_ = new char[size() + 1];
	for (size_t i = 0; i < size(); ++i) {
		str_[i] = s[i];
	}
	str_[size()] = '\0';
}
String::~String() {
	if(str_ != NULL) {
		delete[] str_;
	}
}
size_t String::size() const {
	return size_;
};
const char* String::c_str() const {
	return &str_[0];
}
const char &String::operator[] (const size_t i) const {
	return str_[i];
}
char &String::operator[] (const size_t i) {
	return const_cast<char &>( static_cast<const String &>(*this)[i] );
}
String &String::operator+= (const String &s) {
	this->append(s);
	return *this;
}
String &String::operator+= (const char* s) {
	this->append(s);
	return *this;
}
String &String::operator+= (char s) {
	this->append(s);
	return *this;
}
void String::clear() {
        size_ = 0;
        str_[0] = '\0';
}
String &String::operator= (String s) {
        this->swap(s);
       return *this;
}
String &String::operator= (char* s) {
        this->swap(s);
       return *this;
}
String &String::operator= (char s) {
        char *oldStr = str_;
        str_ = new char[2];
        str_[0] = s;
        str_[1] = '\0';
        delete[] oldStr;
        return *this;
}
String &String::swap (String &rhs) {
        size_t tempSize = rhs.size();
        rhs.size_ = size();
        size_ = tempSize;
        char* strTemp = rhs.str_;
        rhs.str_ = str_;
        str_ = strTemp;
        return *this;
}
String &String::swap (char* &rhs) {
        size_ = strlen(rhs);
        char* strTemp = rhs;
        rhs = str_;
        str_ = strTemp;
        return *this;
}
void String::append (const String &s) {
        size_t newSize = size() + s.size();
        size_t oldSize = size();
        char* oldData = str_;
        size_ = newSize;
        str_ = new char[newSize + 1];
        for (size_t i = 0; i < newSize; ++i) {
                if (i < oldSize) {
                        str_[i] = oldData[i];
                } else {
                        str_[i] = s.str_[i - oldSize];
                }
        }
        str_[newSize] = '\0';
        delete[] oldData;
}
void String::append (const char* s) {
        size_t newSize = size() + strlen(s);
        size_t oldSize = size();
        char* oldData = str_;
        size_ = newSize;
        str_ = new char[newSize + 1];
        for (size_t i = 0; i < newSize; ++i) {
                if (i < oldSize) {
                        str_[i] = oldData[i];
```

```
		} else {
			str_[i] = s[i - oldSize];
		}
	}
	str_[newSize] = '\0';
	delete[] oldData;
}
void String::append (char s) {
	size_t newSize = size() + 1;
	size_t oldSize = size();
	char* oldData = str_;
	size_ = newSize;
	str_ = new char[newSize + 1];
	for (size_t i = 0; i < newSize; ++i) {
		if (i < oldSize) {
			str_[i] = oldData[i];
		} else {
			str_[i] = s;
		}
	}
	str_[newSize] = '\0';
	delete[] oldData;
}
bool operator== (const String &lhs, const String &rhs) {
	bool result = 1;
	const char *lstr = lhs.c_str();
	const char *rstr = rhs.c_str();
	size_t shorter = (lhs.size() < rhs.size()) ? lhs.size() : rhs.size();
	for (size_t i = 0; i < shorter; ++i) {
		if ((int)lstr[i] != (int)rstr[i] || lhs.size() != rhs.size()) {
			result = 0;
			break;
		}
	}
	return result;
}
bool operator== (const char* lhs, const String &rhs) {
	bool result = 1;
	const char *rstr = rhs.c_str();
	size_t shorter = (strlen(lhs) < rhs.size()) ? strlen(lhs) : rhs.size();
	for (size_t i = 0; i < shorter; ++i) {
		if ((int)lhs[i] != (int)rstr[i] || strlen(lhs) != rhs.size()) {
			result = 0;
			break;
		}
	}
	return result;
}
bool operator== (const String &lhs, const char* rhs) {
	bool result = 1;
	const char *lstr = lhs.c_str();
	size_t shorter = (lhs.size() < strlen(rhs)) ? lhs.size() : strlen(rhs);
	for (size_t i = 0; i < shorter; ++i) {
		if ((int)lstr[i] != (int)rhs[i] || lhs.size() != strlen(rhs)) {
			result = 0;
			break;
		}
	}
	return result;
}
```

```
bool operator!= (const String &lhs, const String &rhs) {
	return !(lhs == rhs);
}
bool operator!= (const char* lhs, const String &rhs) {
	return !(lhs == rhs);
}
bool operator!= (const String &lhs, const char* rhs) {
	return !(lhs == rhs);
}
bool LexicographicCmp(const char* lhs, const char* rhs, const size_t cmpSize, bool
rightLonger) { //return 1: right string larger
	bool rightLarger = 0;
	for (size_t i = 0; i < cmpSize; ++i) {
		if ((int)lhs[i] < (int)rhs[i]) {
			rightLarger = 1;
			break;
		}
		if ((int)rhs[i] < (int)lhs[i]) {
			rightLarger = 0;
			break;
		}
		if (!rightLarger && rightLonger && ((i+1) == cmpSize)) {
			rightLarger = 1;
		}
	}
	return rightLarger;
}
bool operator< (const String &lhs, const String &rhs) {
	const char *lstr = lhs.c_str();
	const char *rstr = rhs.c_str();
	size_t cmpSize = (lhs.size() < rhs.size()) ? lhs.size() : rhs.size();
	bool rightLonger = (lhs.size() < rhs.size()) ? 1 : 0;
	return LexicographicCmp(lstr, rstr, cmpSize, rightLonger);
}
bool operator< (const char* lhs, const String &rhs) {
	const char *rstr = rhs.c_str();
	size_t cmpSize = (strlen(lhs) < rhs.size()) ? strlen(lhs) : rhs.size();
	bool rightLonger = (strlen(lhs) < rhs.size()) ? 1 : 0;
	return LexicographicCmp(lhs, rstr, cmpSize, rightLonger);
}
bool operator< (const String &lhs, const char* rhs) {
	const char *lstr = lhs.c_str();
	size_t cmpSize = (lhs.size() < strlen(rhs)) ? lhs.size() : strlen(rhs);
	bool rightLonger = (lhs.size() < strlen(rhs)) ? 1 : 0;
	return LexicographicCmp(lstr, rhs, cmpSize, rightLonger);
}
bool operator<= (const String &lhs, const String &rhs) {
	return ((lhs == rhs) || (lhs < rhs));
}
bool operator<= (const char* lhs, const String &rhs) {
	return ((lhs == rhs) || (lhs < rhs));
}
bool operator<= (const String &lhs, const char* rhs) {
	return ((lhs == rhs) || (lhs < rhs));
}
bool operator> (const String &lhs, const String &rhs) {
	const char *lstr = lhs.c_str();
	const char *rstr = rhs.c_str();
	size_t cmpSize = (lhs.size() < rhs.size()) ? lhs.size() : rhs.size();
	bool rightLonger = (rhs.size() < lhs.size()) ? 1 : 0;
	return LexicographicCmp(rstr, lstr, cmpSize, rightLonger);
}
bool operator> (const char* lhs, const String &rhs) {
	const char *rstr = rhs.c_str();
	size_t cmpSize = (strlen(lhs) < rhs.size()) ? strlen(lhs) : rhs.size();
	bool rightLonger = (rhs.size() < strlen(lhs)) ? 1 : 0;
	return LexicographicCmp(rstr, lhs, cmpSize, rightLonger);
}
bool operator> (const String &lhs, const char* rhs) {
	const char *lstr = lhs.c_str();
	size_t cmpSize = (lhs.size() < strlen(rhs)) ? lhs.size() : strlen(rhs);
	bool rightLonger = (strlen(rhs) < lhs.size()) ? 1 : 0;
	return LexicographicCmp(rhs, lstr, cmpSize, rightLonger);
}
bool operator>= (const String &lhs, const String &rhs) {
	return ((lhs == rhs) || (lhs > rhs));
}
bool operator>= (const char* lhs, const String &rhs) {
	return ((lhs == rhs) || (lhs > rhs));
}
bool operator>= (const String &lhs, const char* rhs) {
	return ((lhs == rhs) || (lhs > rhs));
}
std::ostream &operator<< (std::ostream &os, const String &s) {
	return os << s.c_str();
}
std::istream &operator>> (std::istream &is, String &s) {
	is >> s.str_;
	s.size_ = strlen(s.c_str());
	return is;
}

String operator+ (const String &lhs, const String &rhs) {
	String str = String(lhs);
	str.append(rhs);
	return str;
}
String operator+ (const char* lhs, const String &rhs) {
	String str = String(lhs);
	str.append(rhs);
	return str;
}
String operator+ (const String &lhs, const char* rhs) {
	String str = String(lhs);
	str.append(rhs);
	return str;
}
String operator+ (const String &lhs, char rhs) {
	String str = String(lhs);
	str.append(rhs);
	return str;
}
String operator+ (char lhs, const String &rhs) {
	String str;
	str = lhs;
	str.append(rhs);
	return str;
}
void swap (String &lhs, String &rhs) {
	lhs.swap(rhs);
```

```
}
void swap (char* &lhs, String &rhs) {
	rhs.swap(lhs);
}
void swap (String &lhs, char* &rhs) {
	lhs.swap(rhs);
}
```