```
class String
{
	friend ostream& operator<<(ostream &out, const String &s);
public:
	String(const char* str = "")
	{
		// 构造string类对象时，如果传递nullptr指针，认为程序非法，此处断言下
		if (nullptr == str)
		{
			assert(false);
			return;
		}
		_str = new char[strlen(str) + 1];
		strcpy(_str, str);
	}
	String(const String &s) : _str(new char[strlen(s._str)+1])
	{
		//_str = new char[strlen(s._str)+1];
		strcpy(_str, s._str);
	}
	String& operator=(const String &s)
	{
		if(this == &s)
			return *this;
		//异常不安全
		delete []_str;
		_str = new char[strlen(s._str)+1];
		strcpy(_str, s._str);

		return *this;
	}

	~String()
	{
		if (_str)
		{
			delete[] _str;
			_str = nullptr;
		}
	}
private:
	char* _str;
};
```

```
ostream& operator<<(ostream &out, const String &s)
{
	out<<s._str;
	return out;
}

void main()
{
	String s("Hello");
	cout<<s<<endl;
	String s1;
	s1 = s;
}
```

赋值操作符重载的深拷贝

```
class String
{
public:
	String(const char* str = "")
	{
		if (nullptr == str) //
			str = "";
		_str = new char[strlen(str) + 1];
		strcpy(_str, str);
	}
	String(const String& s)   : _str(nullptr)
	{
		//_str = new char[strlen(s._str)+1];
		//strcpy(_str, s._str);

		String strTmp(s._str);
		swap(_str, strTmp._str);
	}
	String& operator=(const String &s)
	{
		if(this != &s)
		{
			String Tmp(s._str); //
			char *tmp = _str;
			_str = Tmp._str;
			Tmp._str = tmp;
		}
		return *this;
	}
```

```
	~String()
	{
		if (_str)
		{
			delete[] _str;
			_str = nullptr;
		}
	}
private:
	char* _str;
};

void main()
{
	String s; //
}
```

string的使用

```
int main()
{
	string num("123456789");
	cout<<num<<endl;
	reverse(num.begin(), num.end());
	cout<<num<<endl;
	return 0;
}
```

完整的string的模拟实现（resize,reserve, begin, end, size, capacity, push_back, clear, c_str, [], <<）

```
//完整的string的模拟实现
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<assert.h>
using namespace std;

namespace bit
{
```

```
class string
{
	friend ostream& operator<<(ostream& out, const string& s);
public:
	string(const char* str = "") //nullptr
	{
		m_size = strlen(str);
		m_capacity = m_size; //
		m_str = new char[m_capacity + 1]; //
		strcpy(m_str, str);
	}
	string(const string& s) :m_str(nullptr), m_capacity(0), m_size(0)
	{
		string tmp(s.m_str);  //拷贝构造函数调用构造函数，类的构造函数具有类型转换的功能，能够构造tmp
		Swap(tmp); //*this  tmp  深拷贝（被拷贝的对象先拷贝到一个临时对象tmp中，
		/*tmp是一个栈上的临时对象但它拥有的数据成员在堆上，
			于是将tmp和this指针代表的要被拷贝进去的对象的数据成员进行交换，
			出swap函数的同时临时变量tmp也就被释放，同时原先要拷贝进去的数据成员也就非释放了）*/
	}
	string& operator=(const string& s)
	{
		if (this != &s)
		{
			string tmp(s);  //调用拷贝构造函数用对象s构造对象tmp
			Swap(tmp);  //然后调用Swap函数进行深拷贝
		}
		return *this;
	}
	~string()
	{
		if (m_str)
		{
			delete[]m_str;
			m_str = nullptr;
			m_capacity = m_size = 0;
		}
	}
public:
	typedef char* iterator; //迭代器  在string中迭代器是原生指针
	typedef const char* const_iterator; //
public:
	iterator begin()
	{return m_str;}
```

```
        iterator end()  //end永远迭代字符串最后一个元素的下一个位置
        {return m_str + m_size;}
        const_iterator begin()const
        {return m_str;}
        const_iterator end()const
        {return m_str + m_size;}
public:
        size_t size()const
        {return m_size;};
        size_t capacity()const
        {return m_capacity;}
public:
        void push_back(char ch)
        {
                if (m_size >= m_capacity)
                {
                        int new_capacity = (m_capacity == 0 ? 1 : m_capacity * 2);
                        reserve(new_capacity);
                }
                m_str[m_size++] = ch;
                m_str[m_size] = '\0';
        }
        void clear()
        {
                m_size = 0;
                m_str[0] = '\0';
        }
        const char* c_str()const
        {
                return m_str;
        }
public:
        char operator[](size_t i) //-1
        {
                assert(i < m_size); //15 14
                return m_str[i];
        }
        const char operator[](size_t i)const //-1
        {
                assert(i < m_size); //15 14
                return m_str[i];
        }
public:
        //abcdefghij00000
        void resize(size_t new_sz, char c = '\0')
        {
                if (new_sz > m_size)
```

```
            {
                if (new_sz > m_capacity)
                {
                    reserve(new_sz); //
                }
                memset(m_str + m_size, c, new_sz - m_size);
//memset的三个参数，1内存中的起始地址，2要被修改的值，3修改的个数
            }
            m_str[new_sz] = '\0';
            m_size = new_sz;
        }
        void reserve(size_t new_capacity)
        {
            if (new_capacity > m_capacity)
            {
                char* new_str = new char[new_capacity + 1];
                m_capacity = new_capacity;
                strcpy(new_str, m_str);
                delete[]m_str;
                m_str = new_str;
            }
        }
        void Swap(string& s)
        {
            swap(m_str, s.m_str);
            swap(m_capacity, s.m_capacity);
            swap(m_size, s.m_size);
        }
    private:
        char* m_str;
        size_t m_capacity;
        size_t m_size;
    };

};

ostream& bit::operator<<(ostream& out, const string& s)
{
    out << s.m_str;
    return out;
}

int main()
{
    bit::string s("Hello Bit.");
    bit::string s1;
    s1 = s;
```

```
    for (size_t i = 0; i < s1.size(); ++i)
        cout << s1[i];
    cout << endl;
    return 0;
}
```

在string中迭代器是原生指针

end永远迭代字符串最后一个元素的下一个位置

memset的三个参数，1内存中的起始地址，2要被修改的值，3修改的个数

拷贝构造函数考虑使用深拷贝