String字符串的构造函数：

class String {

public:

int m\_length;

char\* m\_buff;

static int strlen(const char\* ptr);

static void strncpy(char\* dest, const char\* src, int n);

String(const char\* val = nullptr);

~String(void);

int size() { return m\_length; };

friend ostream& operator<<(ostream&, const string&);

friend string operator+(const string&, const string);

};

String::String(const char\* val) ://普通构造函数；

m\_length(strlen(val)), m\_buff(m\_length > 0 ? new char[m\_length] : nullptr) {

strncpy(m\_buff, val, m\_length);

}

int String::strlen(const char\* ptr) {

int len = 0;

while (ptr && \*ptr++ != '\0')

++len;

return len;

}

void String::strncpy(char\* dest, const char\* src, int n) {

for (int i = 0; i < n; ++i)

dest[i] = src[i];

}

ostream& operator<<(ostream& os, const String& s) {

for (int i = 0; i < s.m\_length; ++i)

os << s.m\_buff[i];

return os;

}

String operator+(const String& s1, const String& s2) {

String res;

res.m\_length = s1.m\_length + s2.m\_length;

res.m\_buff = new char[res.m\_length];

String::strncpy(res.m\_buff, s1.m\_buff, s1.m\_length);

String::strncpy(res.m\_buff + s1.m\_length, s2.m\_buff, s2.m\_length);

return res;

}

String::String(const String& rhs) ://复制构造函数

m\_length(rhs.m\_length), m\_buff(m\_length > 0 ?

new char[m\_length] : nullptr) {

strncpy(m\_buff, rhs.m\_buff, m\_length);//复制数据

}

String::String(String&& rhs) : m\_length(rhs.m\_length), m\_buff(rhs.m\_buff) {

rhs.m\_buff = nullptr;

rhs.m\_length = 0;

}//移动构造函数

String& String::operator=(String&& rhs) {

if (this != &rhs) {

delete[] m\_buff;

m\_length = rhs.m\_length;

m\_buff = rhs.m\_buff;

rhs.m\_buff = nullptr; // 置为空指针

rhs.m\_length = 0;

}

return \*this;

}

析构函数：

~String(void) {

delete[]m\_buff;

}

删除字符：

string erase(string str, int index, int length)

{

string temp;

for (int i = 0; i < index; i++){

temp.push\_back(str[i]);

}

for (int i = index + length; i < str.size(); i++){

temp.push\_back(str[i]);

}

str = temp;

return str;

}

移除全部字符：

string clear(string str){

string temp = "";

str = temp;

return str;

}

void pushback(string& str, char a){

str = str + a;

}

单链表代码：

#include<iostream>

using namespace std;

template<typename T>

class Node {

T m\_data;

Node\* m\_next = nullptr;

public:

Node(const T& val) :m\_data(val) { }

const T& data() const { return m\_data; }

T& data() { return m\_data; }

Node\* next() { return m\_next; }

};

template<typename T>

class SList {

Node<T>\* m\_head = nullptr, \* m\_tail = nullptr;

public:

SList() = default;

~SList();

void clear();

void push\_back(const T& val);

Node<T>\* insert(Node<T>\* pos, const T& val);

Node<T>\* find(const T& val);

};

template<typename T>

void SList<T>::push\_back(const T& val) {

Node<T>\* node = new Node<T>(val);

if (m\_head == nullptr)

m\_head = m\_tail = node;

else {

m\_tail->m\_next = node;

m\_tail = node;

}

}

template<typename T>

Node<T>\* SList<T>::find(const T& val) {

Node<T>\* p = m\_head;

while (p != nullptr && p->m\_data != val)

p = p->m\_next;

return p;

}

template<typename T>

Node<T>\* SList<T>::insert(Node<T>\* pos, const

T& val) {

Node<T>\* node = new Node<T>(val);

node->m\_next = pos->m\_next;

pos->m\_next = node;

if (pos == m\_tail)

m\_tail = node;

return node;

}

template<typename T>

void SList<T>::clear() {

Node<T>\* p = nullptr;

while (m\_head != nullptr) {

p = m\_head;

m\_head = m\_head->m\_next;

delete p;

}

m\_tail = nullptr;

}

template<typename T>

SList<T>::~SList() {

clear();

}

template<typename T>

ostream& operator<<(ostream&, const SList<T>&);

template<typename T>

ostream& operator<<(ostream& os, const SList<T>& list) {

Node<T>\* p = list.m\_head;

while (p != nullptr) {

os << p->data() << " ";

p = p->next();

}

return os;

};

template<typename T>

class SList; //前向声明

template<typename T>

int main() {

SList<int> l;

int val;

while (cin >> val) {

l.push\_back(val);

}

cout << l << endl;

Node<int>\* pos = l.find(20);

l.insert(pos, 25);

cout << l << endl;

}