# 链表实验

## 1 实现单链表节点类

本小节实验可以全部写在 main.cpp 中。

链表节点类代码如下：

```
template<class T>
struct link_node{
    T data;
    link_node* next;
    link_node(T d, link_node* n):data(d), next(n){}
};
```

注意节点类的代码应该在所有函数的上方。

### 新建单链表

编辑 main 函数：

```
int main() {
    cout<<"单链表实验"<<endl;
    auto head = new link_node<int>(0, nullptr);
    return 0;
}
```

编译运行。
🥳 如果一切顺利，继续下面的操作。

### 插入元素

在 main 函数前添加函数：

```cpp
template<class T>
link_node<T>* insert_after(link_node<T>* node, T data){
    auto new_node = new link_node<T>(data, nullptr);
    new_node->next = node->next;
    node->next = new_node;
    return new_node;
}
```

该函数在 node 节点后插入数据元素 data。

修改 main 函数

```cpp
int main() {
    cout<<"单链表实验"<<endl;
    auto head = new link_node<int>(0, nullptr);
    auto p1 = insert_after(head, 1);
    cout<<"p1 所指向的元素为：  "<<p1->data<<endl;
    return 0;
}
```

编译运行，应能得到如下输出：

> 单链表实验
> p1 所指向的元素为：  1

🥳 如果一切顺利，继续下面的操作。

添加函数：

```cpp
template<class T>
void print_list(link_node<T>* head){
    auto curr_node = head->next;
    while(curr_node != nullptr){
        cout<<curr_node->data<<" ";
        curr_node = curr_node->next;
    }
    cout<<endl;
}
```

该函数将依次打印头节点指针 head 所指单链表中的所有元素。

修改 main 函数：

```cpp
int main() {
    cout<<"单链表实验"<<endl;
    auto head = new link_node<int>(0, nullptr);
    auto p1 = insert_after(head, 1);
    cout<<"p1 所指向的元素为：  "<<p1->data<<endl;

    insert_after(head, 2);
    insert_after(p1, 3);

    cout<<"表中元素为：  ";
    print_list(head);
    return 0;
}
```

编译运行，应能得到如下输出：

> 单链表实验
> p1 所指向的元素为：  1
> 表中元素为：  2 1 3

🥳 如果一切顺利，继续下面的操作。

```cpp
template<class T>
void delete_after(link_node<T>* node){
    if(node->next != nullptr){
        auto deleted_node = node->next;
        node->next = node->next->next;
        delete deleted_node;
    }
}
```

该函数删除 node 节点的后续节点。

修改 main 函数：

```cpp
int main() {
    cout<<"单链表实验"<<endl;
    auto head = new link_node<int>(0, nullptr);
    auto p1 = insert_after(head, 1);
    cout<<"p1 所指向的元素为：  "<<p1->data<<endl;

    insert_after(head, 2);
    insert_after(p1, 3);

    cout<<"表中元素为：  ";
    print_list(head);

    delete_after(p1);

    cout<<"删除p1后表中元素为：  ";
    print_list(head);
    return 0;
}
```

编译运行，应能得到如下输出：

> 单链表实验
> p1 所指向的元素为：  1
> 表中元素为：  2 1 3
> 删除p1后表中元素为：  2 1

记得，单链表在使用完毕之后必须删除整表释放内存。
添加 destroy_list 函数：

```cpp
template<class T>
void destroy_list(link_node<T>* head){
    while(head->next != nullptr){
        delete_after(head);
    }
    delete head;
}
```

在 main 函数末尾添加删除链表语句：

```cpp
int main() {
    cout<<"单链表实验"<<endl;
    auto head = new link_node<int>(0, nullptr);
    auto p1 = insert_after(head, 1);
    cout<<"p1 所指向的元素为：  "<<p1->data<<endl;

    insert_after(head, 2);
    insert_after(p1, 3);

    cout<<"表中元素为：  ";
    print_list(head);

    delete_after(p1);

    cout<<"删除p1后表中元素为：  ";
    print_list(head);
    destroy_list(head);
    return 0;
}
```

编译运行后，输出与上一次相同。

🥳 如果以上操作都一切顺利，你已掌握带头节点单链表的基本使用方法。

> **思考：** 如何判断程序是否有内存泄漏?

# 2 单链表数据操作挑战

从文件 link_list_exp.txt 中读取字符串元素，并按照原顺序存储到一个单链表中，表头指针为 head。

使用 print_list 函数打印此单链表。

delete_len_of_3 函数的功能为删除 head 所指单链表中所有字符串长度为3的数据元素。请补全这个函数。

```cpp
void delete_len_of_3(link_node<string>* head){

}
```

对 head 所指单链表执行函数 delete_len_of_3，并使用 print_list 函数打印单链表。

> **注意：**
> 想要使用如下语句插入c字符串到单链表中是不可以的哦！

```
auto head = new link_node<string>("", nullptr);
insert_after(head, "aaa");
```

请思考为什么，以及如何解决。

reverse 函数的功能是将 head 所指单链表中的元素进行逆序排列。
例如，如果表中元素为 {a,b,c}，则函数执行之后将变为 {c,b,a}。
请补全这个函数。

```
template<class T>
void reverse(link_node<T>* head){

}
```

对 head 所指单链表执行函数 reverse，并使用 print_list 函数打印单链表。

# 3 实现用于模拟 std::list 类的 simList

新建头文件 simList.h 并输入以下代码：

```
#ifndef simList_h
#define simList_h

template <class T>
class simList {
private:
    struct node{ //双链表节点
        T data;
        node *prev, *next;
        node(T d, node *p, node *n):data(d), prev(p), next(n){};
    };
    int m_size;
    node *m_head;
    node *m_tail;
public:
    class iterator{ //双链表迭代器
        friend class simList<T>;
    private:
        node *m_current;
        iterator(node *p):m_current(p){};
    public:
        T& get_data();    //获取当前迭代器所指节点中的元素
        iterator next();  //获取当前迭代器所指节点的后继节点迭代器
        iterator prev();  //获取当前迭代器所指节点的前序节点迭代器
        bool operator!=(iterator itr);
    };
    simList();
    ~simList();  //暂时注释掉此行
    iterator begin();                       //获取第一个元素的迭代器
    iterator end();                         //获取尾节点迭代器
    iterator insert(iterator itr, T item);  //在迭代器itr位置之前插入item
    iterator erase(iterator itr);           //删除迭代器指向的节点
    int size(){return m_size;}              //获取元素个数
    void push_back(T item);           //在List尾部添加数据元素item
};

#endif /* simList_h */
```

> **注意：**
> 后续在 simList.h 文件中添加的内容都必须在最后一行 #endif 的前面。

## simList 构造函数

在 simList.h 中加入如下代码：

```cpp
template <class T>
simList<T>::simList() {
    m_size = 0;
    m_head = new node(T(), nullptr, nullptr);
    m_tail = new node(T(), nullptr, nullptr);
    m_head->next = m_tail;
    m_tail->prev = m_head;
}
```

在 main.cpp 文件中输入以下代码

```cpp
#include <iostream>
#include "simList.h"
using namespace std;
int main(){
    simList<int> a;
    cout<<"测试simList: "<<endl;
    return 0;
}
```

为了能够让程序编译运行，我们先放一个空的析构函数在 simList.h 中：

```cpp
template <class T>
simList<T>::~simList(){

}
```

如果能够正确编译运行，则说明 simList 结构声明以及构造函数编写正确。
🥳 可以继续下面的操作。

# 获取链表头与尾节点

在 simList.h 中加入如下代码：

```cpp
template <class T>
typename simList<T>::iterator simList<T>::begin() {
    return simList<T>::iterator(m_head->next);
}

template <class T>
typename simList<T>::iterator simList<T>::end() {
    return simList<T>::iterator(m_tail);
}
```

这两个函数分别返回链表的第一个元素节点（begin()）和尾节点（end()）。
我们暂时不对它们进行测试，在完成插入函数后一并对它们一起进行测试。

# 插入元素函数

在 simList.h 中加入如下代码：

```
template <class T>
typename simList<T>::iterator simList<T>::insert(iterator itr, T item) {
    node *p = itr.m_current;
    node *newNode = new node(item, p->prev, p);
    p->prev->next = newNode;
    p->prev = p->prev->next;
    ++m_size;
    return iterator(p->prev);
}
```

修改 main 函数为：

```
int main(){
    simList<int> a;
    cout<<"测试simList：  "<<endl;
    a.insert(a.begin(), 1);
    a.insert(a.end(), 2);

    cout<<"表中元素个数：  "<<a.size()<<endl;
    return 0;
}
```

上述代码在链表头插入1，链表尾插入2。
编译运行后应有如下输出：

> 测试simList：
> 表中元素个数：2

🥳 如果能够正确编译运行，可以继续下面的操作。

# 完善迭代器的成员函数

## 前序迭代器

在 simList.h 中加入如下代码：

```
template <class T>
typename simList<T>::iterator simList<T>::iterator::prev() {
    return simList<T>::iterator(m_current->prev);
}
```

prev 函数获取迭代器所指节点的前序节点的迭代器。

## 后续迭代器

继续在 simList.h 中加入如下代码：

```
template <class T>
typename simList<T>::iterator simList<T>::iterator::next() {
    return simList<T>::iterator(m_current->next);
}
```

next 函数获取迭代器指针所指节点的后继节点的迭代器。

## 迭代器不等判断

继续在 simList.h 中加入如下代码：

```
template <class T>
bool simList<T>::iterator::operator!=(iterator itr) {
    return m_current != itr.m_current;
}
```

这里重载了 != 操作符，用于判断两个迭代器是否指向同一个节点。

## 获取迭代器所指节点元素

继续在 simList.h 中加入如下代码：

```
template <class T>
T& simList<T>::iterator::get_data() {
    return m_current->data;
}
```

get_data 函数获取迭代器所指节点中存储的数据元素。

修改 main 函数为：

```cpp
int main(){
    simList<int> a;
    cout<<"测试simList：<<endl;
    a.insert(a.begin(), 1);
    a.insert(a.end(), 2);
    a.insert(a.end(), 3);

    cout<<"表中元素个数：<<a.size()<<endl;

    cout<<"从头到尾依次打印表中元素：";
    for(auto iter=a.begin(); iter != a.end(); iter=iter.next()){
        cout<<iter.get_data()<<" ";
    }
    cout<<endl;

    cout<<"从尾到头依次打印表中元素：";
    auto iter=a.end().prev();
    while(iter != a.begin()){
        cout<<iter.get_data()<<" ";
        iter=iter.prev();
    }
    cout<<iter.get_data()<<endl;
    return 0;
}
```

编译运行后应有如下输出：

> 测试simList：
> 表中元素个数：3
> 从头到尾依次打印表中元素：1 2 3
> 从尾到头依次打印表中元素：3 2 1

🥳 如果能够正确编译运行，可以继续下面的操作。

## 删除元素

simList.h 中加入如下代码：

```
template <class T>
typename simList<T>::iterator simList<T>::erase(simList<T>::iterator itr) {
    node *p = itr.m_current;
    iterator re(p->next);
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    m_size--;
    return re;
}
```

修改 main 函数为：

```
int main(){
    simList<int> a;
    cout<<"测试simList：<<endl;
    a.insert(a.begin(), 1);
    a.insert(a.end(), 2);
    auto p = a.insert(a.end(), 3); //修改此行

    a.insert(a.end(), 4);
    a.erase(p);

    cout<<"表中元素个数：<<a.size()<<endl;

    cout<<"从头到尾依次打印表中元素：";
    for(auto iter=a.begin(); iter != a.end(); iter=iter.next()){
        cout<<iter.get_data()<<" ";
    }
    cout<<endl;

    cout<<"从尾到头依次打印表中元素：";
    auto iter=a.end().prev();
    while(iter != a.begin()){
        cout<<iter.get_data()<<" ";
        iter=iter.prev();
    }
    cout<<iter.get_data()<<endl;
    return 0;
}
```

编译运行后，应有如下输出：

> 测试simList：
> 表中元素个数：3
> 从头到尾依次打印表中元素：1 2 4
> 从尾到头依次打印表中元素：4 2 1

🥳 如果能够正确编译运行，可以继续下面的操作。

## simList 析构函数

由于 simList 类中大量使用了new关键字申请内存。所以一定要手动编写析构函数将所有申请的内存归还给系统。

修改 ~simList 函数为：

```
template <class T>
simList<T>::~simList(){
    while(m_size > 0){    //依次删除第一个元素
        erase(begin());
    }
    delete m_head;
    delete m_tail;
}
```

在 main 函数中，我们声明的是一个 simList 对象，因此析构函数是自动运行的，所以需要对 main 函数进行修改。

## 更方便的尾插入函数

simList.h 中加入如下代码：

```
template <class T>
void simList<T>::push_back(T item){
    insert(end(), item);
}
```

push_back 函数将数据直接插入链表尾部。
请自行修改 main 函数测试此函数是否工作正确。

至此，simList的实现全部完成！
🥳🎉

# 4 simList 的使用挑战

将文件 simList_exp.txt 中的数据按顺序读入到一个 simList<string> 中。

simList_exp.txt 文件的第一行为元素个数，之后的每一行都为一个字符串元素，共500000个元素。
simList_exp.txt 的前几行如下：

```
500000
Vz
WCthO
TygR
HWNCzDp
bIKRCE
ChaK
OkZltQD
……
```

删除 simList 中所有长度小等于3的字符串元素。
输出当前表中元素个数。

> 参考答案:
> 389036

将自己的名字插入到第 212202 位。
输出从第 212200 位到 212210 位的所有元素。

> 参考答案:
> HCOTJCpQQ VZqfAEko 张三 aYGWgzm OCTbwGOmR FOCVdruJm UcQa EMGGdmTj uHPXbUahK KaFCnlVFLH uQPIIwOk

# 5 std::list 的使用

使用 C++ 标准模板库中的 std::list 类将实验4的操作重新实现一遍。