Experiment2-董皓彧

环境：

```
g++.exe (x86_64-win32-seh-rev1, Built by MinGW-Builds project) 13.2.0
Visual Stdio Code 1.86.2
```

作业仓库地址：https://github.com/FHYQ-Dong/Tsinghua-Program-Design-Assignments-2/tree/main/Experiment2

必做题

Experiment2-1

题目：

```
什么是类的构造函数？其主要用途是什么？本章学习了哪些构造函数类型？它们各有什么特点？
```

答：

```
构建对象时，需要对每个对象的内存空间进行初始化，给对象的属性赋初值。考虑到对象成员的访问属性等限定，引入类的构造函数，来完成对象的初始化工作。
1. 构造函数（默认构造函数、带不带参数（缺省值））（定义对象时自动调用）
2. 复制构造函数（使用一个已经存在的对象来构造并初始化同类的一个新对象（参数为对象的引用）
```

Experiment2-2

题目：

```
复制构造函数在什么时候会被自动调用？在哪些情况下会产生所谓的“浅拷贝”问题？
```

答：

```
1. 当用类的一个对象去初始化该类的另一个新定义的对象时
2. 如果函数的形参是类的对象，调用函数，形参和实参结合时
3. 如果函数的返回值是类的对象，函数执行完成返回对象给调用者时
当进行“赋值”时会产生“浅拷贝”的问题，如使用系统自动生成的复制构造函数、给对象赋值时等
```

选做题

Optional-Experiment2-1

题目：

> 编写一个程序，定义一个类 Person，其中包含name（字符指针）和age（整型变量）两个成员变量。之后编写 Person 类的构造函数，复制构造函数和析构函数，在主函数中实现深复制与浅复制，并分析构造函数和析构函数的调用时机

输入格式：

> 略

输出格式：

> 见代码注释

代码：

```
#include <iostream>
#include <cstring>
using std::cout, std::endl;

class Person {
    private:
        char* name;
        int age;
    public:
        Person() {
            name = nullptr;
            age = 0;
            cout << "default constructor" << endl;
        }
        Person(const char* name, int age) {
            size_t len = strlen(name);
            this->name = new char[len + 1];
            strcpy(this->name, name);
            this->age = age;
            cout << "constructor with args" << endl;
        }
        Person(const Person& p) {
            size_t len = strlen(p.name);
            this->name = new char[len + 1];
            strcpy(this->name, p.name);
            this->age = p.age;
            cout << "deep copy constructor" << endl;
        }
        ~Person() {
            // delete[] name; // cause error if name is nullptr (when shallow copy is used & the original object is destructed)
            cout << "destructor" << endl;
        }
};

int main() {
    // Constructor
    Person p1("Tom", 20), p2;
    // Shallow copy
    p2 = p1;
    // Deep copy
    Person p3 = p1;
    return 0;
}
```

输入1：

```

```

输出1：

```
constructor with args
default constructor
deep copy constructor
destructor
destructor
destructor
```

Optional-Experiment2-2

题目：

```
利用类（class）实现链表，要求实现链表元素查找、插入、删除、逆转、打印，并自行编写测试样例
```

输入格式：

```
略
```

输出格式：

```
见代码注释
```

代码：

```
#include <iostream>
#include <vector>

template <typename T>
inline void swap(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}

template <typename T>
class Node;
template <typename T>
class __List_iterator;
template <typename T>
class List;
```

```cpp
template <typename T>
class Node {
    public:
        T data;
        Node *next, *prev;
        Node();
        Node(const T &data);
        Node(const Node &n);
        ~Node();
};

template <typename T>
class __List_iterator {
    private:
        Node<T> *ptr;

    friend class List<T>;

    public:
        __List_iterator();
        __List_iterator(Node<T> *ptr);
        __List_iterator(const __List_iterator &it);
        ~__List_iterator();
        __List_iterator &operator++();
        __List_iterator &operator--();
        T &operator*();
        bool operator==(const __List_iterator &it);
        bool operator!=(const __List_iterator &it);
};

template <typename T>
class List {
    private:
        Node<T> *head, *tail;
        size_t sz;

    public:
        using iterator = __List_iterator<T>;
        List();
        List(const T data[], size_t size);
        List(const List &l);
        ~List();
        void push_back(const T &data);
        void push_front(const T &data);
        void insert(iterator it, const T &data);
        void pop_back();
        void pop_front();
        void erase(iterator it);
        void erase(iterator first, iterator last);
        void remove(const T &data);
        void remove_all(const T &data);
        void clear();
        void reverse();
        iterator find_the_first(const T &data);
        iterator find_the_last(const T &data);
        iterator find_the_next(const T &data, const iterator &it);
        iterator find_the_prev(const T &data, const iterator &it);
        void print();
        iterator begin();
        iterator end();
};

template <typename T>
Node<T>::Node() {
    next = prev = nullptr;
}
template <typename T>
Node<T>::Node(const T &data) {
    this->data = data;
    next = prev = nullptr;
}
template <typename T>
Node<T>::Node(const Node &n) {
    data = n.data;
    next = prev = nullptr;
}
template <typename T>
Node<T>::~Node() {
    next = prev = nullptr;
}

template <typename T>
__List_iterator<T>::__List_iterator() {
    ptr = nullptr;
}
template <typename T>
__List_iterator<T>::__List_iterator(Node<T> *ptr) {
    this->ptr = ptr;
}
template <typename T>
__List_iterator<T>::__List_iterator(const __List_iterator<T> &it) {
    ptr = it.ptr;
}
template <typename T>
__List_iterator<T>::~__List_iterator() {
    ptr = nullptr;
}
template <typename T>
__List_iterator<T>& __List_iterator<T>::operator++() {
    ptr = ptr->next;
    return *this;
}
template <typename T>
__List_iterator<T>& __List_iterator<T>::operator--() {
    ptr = ptr->prev;
    return *this;
}
template <typename T>
T& __List_iterator<T>::operator*() {
    return ptr->data;
```

```
}
template <typename T>
bool __List_iterator<T>::operator==(const __List_iterator<T> &it) {
    return ptr == it.ptr;
}
template <typename T>
bool __List_iterator<T>::operator!=(const __List_iterator<T> &it) {
    return ptr != it.ptr;
}

template <typename T>
List<T>::List() {
    head = tail = nullptr;
    sz = 0;
}
template <typename T>
List<T>::List(const T data[], size_t size) {
    head = tail = nullptr;
    sz = 0;
    for (size_t i = 0; i < size; i++) {
        push_back(data[i]);
    }
}
template <typename T>
List<T>::List(const List &l) {
    head = tail = nullptr;
    sz = 0;
    for (Node<T> *p = l.head; p != nullptr; p = p->next) {
        push_back(p->data);
    }
}
template <typename T>
List<T>::~List() {
    clear();
}
template <typename T>
void List<T>::push_back(const T &data) {
    Node<T> *p = new Node<T>(data);
    if (head == nullptr) {
        head = tail = p;
    }
    else {
        tail->next = p;
        p->prev = tail;
        tail = p;
    }
    sz++;
}
template <typename T>
void List<T>::push_front(const T &data) {
    Node<T> *p = new Node<T>(data);
    if (head == nullptr) {
        head = tail = p;
    }
    else {
        head->prev = p;
        p->next = head;
        head = p;
    }
    sz++;
}
template <typename T>
void List<T>::insert(List<T>::iterator it, const T &data) {
    Node<T> *p = new Node<T>(data);
    if (it.ptr == nullptr) {
        push_back(data);
    }
    else if (it.ptr == head) {
        push_front(data);
    }
    else {
        p->next = it.ptr;
        p->prev = it.ptr->prev;
        it.ptr->prev->next = p;
        it.ptr->prev = p;
        sz++;
    }
}
template <typename T>
void List<T>::pop_back() {
    if (tail == nullptr) {
        return;
    }
    Node<T> *p = tail;
    tail = tail->prev;
    if (tail == nullptr) {
        head = nullptr;
    }
    else {
        tail->next = nullptr;
    }
    delete p;
    sz--;
}
template <typename T>
void List<T>::pop_front() {
    if (head == nullptr) {
        return;
    }
    Node<T> *p = head;
    head = head->next;
    if (head == nullptr) {
        tail = nullptr;
    }
    else {
        head->prev = nullptr;
    }
    delete p;
    sz--;
}
```

```
template <typename T>
typename List<T>::iterator List<T>::begin() {
    return iterator(head);
}
template <typename T>
typename List<T>::iterator List<T>::end() {
    return iterator(nullptr);
}
template <typename T>
void List<T>::erase(List<T>::iterator it) {
    if (it.ptr == nullptr) {
        return;
    }
    if (it.ptr == head) {
        pop_front();
    }
    else if (it.ptr == tail) {
        pop_back();
    }
    else {
        it.ptr->prev->next = it.ptr->next;
        it.ptr->next->prev = it.ptr->prev;
        delete it.ptr;
        sz--;
    }
}
template <typename T>
void List<T>::erase(List<T>::iterator first, List<T>::iterator last) {
    while (first != last) {
        erase(first++);
    }
}
template <typename T>
void List<T>::remove(const T &data) {
    for (Node<T> *p = head; p != nullptr; p = p->next) {
        if (p->data == data) {
            erase(iterator(p));
            break;
        }
    }
}
template <typename T>
void List<T>::remove_all(const T &data) {
    for (Node<T> *p = head; p != nullptr; ) {
        Node<T> *temp = p->next;
        if (p->data == data) {
            erase(iterator(p));
        }
        p = temp;
    }
}
template <typename T>
void List<T>::clear() {
    while (head != nullptr) {
        pop_front();
    }
}
template <typename T>
void List<T>::reverse() {
    Node<T> *p = head;
    while (p != nullptr) {
        swap(p->next, p->prev);
        p = p->prev;
    }
    swap(head, tail);
}
template <typename T>
typename List<T>::iterator List<T>::find_the_first(const T &data) {
    for (Node<T> *p = head; p != nullptr; p = p->next) {
        if (p->data == data) {
            return iterator(p);
        }
    }
    return end();
}
template <typename T>
typename List<T>::iterator List<T>::find_the_last(const T &data) {
    for (Node<T> *p = tail; p != nullptr; p = p->prev) {
        if (p->data == data) {
            return iterator(p);
        }
    }
    return end();
}
template <typename T>
typename List<T>::iterator List<T>::find_the_next(const T &data, const iterator
&it) {
    for (Node<T> *p = it.ptr->next; p != nullptr; p = p->next) {
        if (p->data == data) {
            return iterator(p);
        }
    }
    return end();
}
template <typename T>
typename List<T>::iterator List<T>::find_the_prev(const T &data, const iterator
&it) {
    for (Node<T> *p = it.ptr->prev; p != nullptr; p = p->prev) {
        if (p->data == data) {
            return iterator(p);
        }
    }
    return end();
}
template <typename T>
void List<T>::print() {
    for (Node<T> *p = head; p != nullptr; p = p->next) {
        std::cout << p->data << " ";
    }
    std::cout << std::endl;
```

```
}

inline void test() {
    int a[] = {1, 2, 3, 4, 5};
    List<int> l(a, 5);
    l.print(); // l = {1, 2, 3, 4, 5}
    l.push_back(6);
    l.print(); // l = {1, 2, 3, 4, 5, 6}
    l.push_front(0);
    l.print(); // l = {0, 1, 2, 3, 4, 5, 6}
    List<int>::iterator it = l.begin();
    std::cout << *it << std::endl; // 0
    l.insert(it, -1);
    l.print(); // l = {-1, 0, 1, 2, 3, 4, 5, 6}
    l.pop_back();
    l.print(); // l = {-1, 0, 1, 2, 3, 4, 5}
    l.pop_front();
    l.print(); // l = {0, 1, 2, 3, 4, 5}
    std::cout << *l.find_the_first(3) << std::endl; // 3
    l.erase(l.find_the_first(3));
    l.print(); // l = {0, 1, 2, 4, 5}
    std::cout << (l.find_the_first(3)==l.end() ? "true" : "false") << std::endl; // true
    l.remove(4);
    l.print(); // l = {0, 1, 2, 5}
    l.insert(l.find_the_first(5), 3); l.insert(l.find_the_first(5), 3);
    l.print(); // l = {0, 1, 2, 3, 3, 5}
    l.remove_all(3);
    l.print(); // l = {0, 1, 2, 5}
    l.reverse();
    l.print(); // l = {5, 2, 1, 0}
    List<int> l2 = l;
    l2.print(); // l2 = {5, 2, 1, 0}
}

int main() {
    test();
    return 0;
}
```

输入1：

```

```

输出1：

```
1 2 3 4 5
1 2 3 4 5 6
0 1 2 3 4 5 6
0
-1 0 1 2 3 4 5 6
-1 0 1 2 3 4 5
0 1 2 3 4 5
3
```

```
0 1 2 4 5
true
0 1 2 5
0 1 2 3 3 5
0 1 2 5
5 2 1 0
5 2 1 0
```