必交作业 4-10 章

周玉川 2017221302006

声明：作业为本人独立完成，如有谎言，天打雷劈。

全部代码在附件中如下图

1. 作业《面向对象程序设计 C++习题 1-3.docx》

已经提交。

2. 其他(PPT 上布置的课后作业)

- 5.7 写个简单计算器

  思路简单，a@b 获得数字 a 和 b，判断中间符号判断运算类型，注意考虑除 0 错误。

  代码

```
#include<cstdio>
#include<iostream>
using namespace std;
class keyboard
{
  public:
    std::string input(){
      char ret[50];
      int a, b;
      char op;
      cout << "Input the expression:";
      cin >> a >> op >> b;
      sprintf(ret, "%d%c%d", a, op, b);
      return ret;
    };
};
```

```
    class monitor{
      public:
        void display(std::string content) { cout << "Result:" << content <<
endl; };
    };
    class calculator{
      private:
        keyboard kbd;
        monitor mntr;

        string calculate(string expression){
          int a=0, b=0,res;
          char op;
          int i = 0;
          char ret[50];
          int num = expression[i];
          while( num <='9' && num >= '0'){
            a *= 10;
            a += num - '0';
            i++;
            num = expression[i];
          };
          op = num;
          num = expression[++i];
          while( num <='9' && num >= '0'){
            b *= 10;
            b += num - '0';
            i++;
            num = expression[i];
          };
          //cout << a << op << b;
          switch (op)
          {
          case '+':
            res = a + b;
            break;
          case '-':
            res = a - b;
            break;
          case '*':
            res = a * b;
            break;
          case '/':
            if (b == 0){
              cout << "!!!Warning:divide zero error!!!"<<endl;
              break;
            }
            res = a / b;
            break;
          default:
            break;
          }
          sprintf(ret, "%d %c %d = %d", a, op, b, res);
          return ret;
        };
      public:
        void work(){
          string exp = kbd.input();
          string content = calculate(exp);
          mntr.display(content);
        };
    };
    int main(){
      calculator t;
      t.work();
      return 0;
```

```
}
```

测试结果：

```
c++程序设计\作业\第五章\"5-7
Input the expression:12*45
12*45Result:123

c:\Users\鑫鑫玉川\Documents\Upupoo\Docker\config\课程\c++程序设计
er\config\课程\c++程序设计\作业\第五章\" && g++ 5-7.cpp -o 5-7 &&
c:\Users\鑫鑫玉川\Documents\Upupoo\Docker\config\课程\c++程序设计
er\config\课程\c++程序设计\作业\第五章\" && g++ 5-7.cpp -o 5-7 &&
c++程序设计\作业\第五章\"5-7
Input the expression:12/0
Divide zero error
Result:12 / 0 = 0

c:\Users\鑫鑫玉川\Documents\Upupoo\Docker\config\课程\c++程序设计
er\config\课程\c++程序设计\作业\第五章\" && g++ 5-7.cpp -o 5-7 &&
c++程序设计\作业\第五章\"5-7
Input the expression:1314+520
Result:1314 + 520 = 1834
```

图 5.7

- 5.8 让任务也成为类

思路：把任务封装成一个类，提供 missionStart 和 missionComplete 接口。

主函数代码 main.cpp

```
//5-8.cpp
#include "list.h"

using namespace std;
class mission
{
  private:
    List ls;
  public:
    void missionStart(){
      cout << "Mission start." << endl;
      ls.init();
      QUADPTR quad[5];
      for (int i = 0; i < 5;i++)
      {
        quad[i] = new Quadrangle();
      }
      quad[0]->tag = "Parallelogram";
      quad[1]->tag = "Rectangle";
      quad[2]->tag = "Square";
      quad[3]->tag = "Trapezoid";
      quad[4]->tag = "Diamond";
      for (int i = 0; i < 5;i++){
        ls.push_back(quad[i]);
      }
      cout << "遍历输出 List: " << endl;
      ls.itrator();
    };
    void missionComplete(){
      cout << "Mission complete." << endl;
      ls.empty();
      cout << "Tip:clear function ok." << endl;
```

```
        };
    };
    int main(){
      mission ms;
      ms.missionStart();
      ms.missionComplete();
      return 0;
}
```

测试结果：

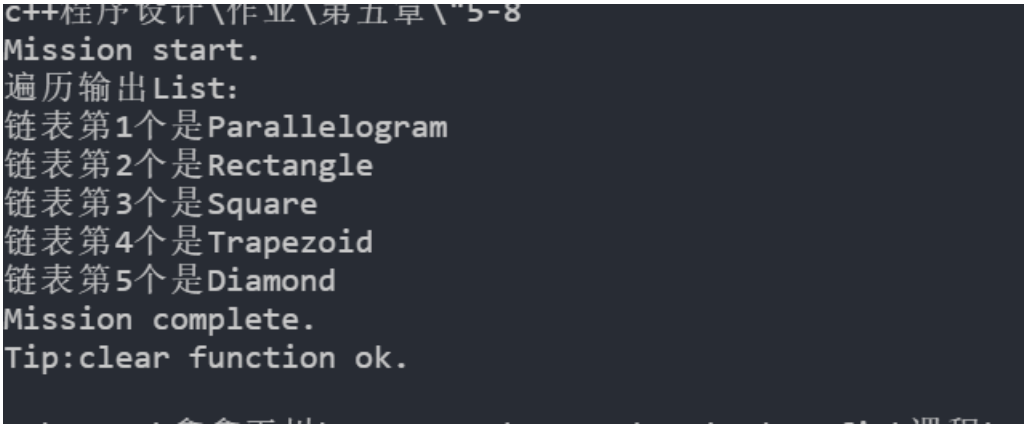

图 5-8

- 6.8 **在构造函数中初始化，在析构函数中回收内存，代理 init 和 empty 函数。**

修改后形体类：

```
    class Parallelogram
    {
    private:
      Quadrangle quad;
      int width, height;

    public:
      // 构造函数
      Parallelogram(std::string tag, int w, int h)
      {
        quad.tag = tag;
        width = w;
        height = h;
      };
      // 复制构造函数
      Parallelogram(const Parallelogram &q) : quad(q.quad), width(q.width),
height(q.height){};
      ~Parallelogram();
      double area();
      void draw(bool showResult = true);
};
```

修改后的 List 类：

```
    class List
    {
    private:
      Node *head, *tail;
      size_t len;

    public:
      // 默认构造函数
      List()
      {
        head = nullptr;
```

```
        tail = nullptr;
        len = 0;
      };
      // 复制构造函数
      List(const List &list)
      {
        if (list.head != nullptr && list.tail != nullptr)
        {
          head = new Node();
          head->quad = list.head->quad;
          head->next = nullptr;
          Node *list_beCopy_itrator = list.head, *list_copy_itrator = head, *
list_new_node;
          if (list.head != list.tail)
          {
            while (list_beCopy_itrator != list.tail)
            {
              list_new_node = new Node();
              list_beCopy_itrator = list_beCopy_itrator->next;
              list_new_node->quad = list_beCopy_itrator->quad;
              list_copy_itrator->next = list_new_node;
              list_copy_itrator = list_new_node;
            }
            tail = list_copy_itrator;
            tail->next = nullptr;
          }
          else
          {
            tail = new Node();
            tail->quad = list.tail->quad;
            tail->next = nullptr;
          }
        }
      };
      // 析构函数
      ~List()
      {
        if (tail != nullptr)
        {
          Node *before = head, *after;
          while (before != tail)
          {
            after = before->next;
            delete before;
            before = after;
          }
          delete before;
        }
      };
}
```

主函数：

```
    //6-8.cpp
    #include "list.h"

    using namespace std;
    class mission
    {
    private:
      List ls;

    public:
      void missionStart()
      {
        cout << "Mission start." << endl;
        // ls.init();不需要初始化
```

```
        QUADPTR quad[5];
        for (int i = 0; i < 5; i++)
        {
          quad[i] = new Quadrangle();
        }
        quad[0]->tag = "Parallelogram";
        quad[1]->tag = "Rectangle";
        quad[2]->tag = "Square";
        quad[3]->tag = "Trapezoid";
        quad[4]->tag = "Diamond";
        for (int i = 0; i < 5; i++)
        {
          ls.push_back(quad[i]);
        }
        cout << "遍历输出 List: " << endl;
        ls.itrator();
      };
      void missionComplete()
      {
        cout << "Mission complete." << endl;
        // ls.empty();也不需要人为回收内存
        cout << "Tip:clear function ok." << endl;
      };
    };
    int main()
    {
      mission ms;
      ms.missionStart();
      ms.missionComplete();
      return 0;
}
```

测试结果：

```
Mission start.
遍历输出List:
链表第1个是Parallelogram
链表第2个是Rectangle
链表第3个是Square
链表第4个是Trapezoid
链表第5个是Diamond
Mission complete.
Tip:clear function ok.
```

- 7.11 重载运算符

重载运算符+=，=，[],+,

```
    List &List::operator+=(QUADPTR p)
    {
      return push_back(p);
    };
    //拼接两个链表
    List &List::operator+=(const List &list)
    {
      if (list.head == nullptr)
        return *this;
      Node *t = list.head;
      while (t != list.tail)
      {
        push_back(t->quad);
        t = t->next;
      }
```

```
    return push_back(t->quad);
  };
  List &List::operator=(const List &p)
  {
    empty();          //先释放
    return *this += p; //再复制
  };
  QUADPTR List::operator[](size_t index)
  {
    if (index < 0 || index >= size())
    {
      return nullptr; //超出范围返回空
    }
    if (index == (size() - 1))
    {
      return tail->quad;
    }
    Node *t = head;
    while (index--)
    {
      t = t->next;
    }
    return t->quad;
  };

  List operator+(const List &list, QUADPTR p)
  {
    List t(list);
    return t += p;
  };
  List operator+(const List &list1, const List &list2)
  {
    List t(list1);
    return t += list2;
};
```

main.cpp 关键代码

```
cout << "原 List: " << endl;
ls.iterator();
ls += quad[4];//运算符+=
cout << "ls += quad[4]之后的 List: " << endl;
ls.iterator();
```

测试结果：

```
Mission start.
原List:
链表第1个是Parallelogram
链表第2个是Rectangle
链表第3个是Square
链表第4个是Trapezoid
ls += quad[4]之后的List:
链表第1个是Parallelogram
链表第2个是Rectangle
链表第3个是Square
链表第4个是Trapezoid
链表第5个是Diamond
Mission complete.
Tip:clear function ok.
```

图 7-11

- 8.8 实现形体类的继承，以及 Container 容器和 List 的继承
  思路：按照形体之间的关系实现继承，注意不同形体类的区别，时候增加私有变量以及重载方法。
  代码：
  Quad.h

```
    #pragma once
    #include <string>
    /*四边形*/
    class Quadrangle
    {
    protected:
      std::string tag;

    public:
      Quadrangle(std::string t) : tag(t){};
      std::string what() const { return tag; };
      double area() const;
      void draw(bool showResult = true) const;
    };

    typedef Quadrangle *QUADPTR;
    /**平行四边形*/
    class Parallelogram : public Quadrangle
    {
    protected:
      int width, height;

    public:
      // 构造函数
      Parallelogram(std::string tag, int w, int h) : Quadrangle(tag),
width(w), height(h){};
      // 复制构造函数
      Parallelogram(const Parallelogram &q) : width(q.width),
height(q.height), Quadrangle(q.tag){};
      ~Parallelogram();
      double area();
      void draw(bool showResult = true);
    };
    /*矩形*/
    class Rectangle : public Parallelogram
    {
    public:
      Rectangle(std::string tag, int w, int h) : Parallelogram(tag, w, h){};
      Rectangle(const Rectangle &q) : Parallelogram(q){};
    };
    /*正方形*/
    class Square : public Rectangle
    {
    public:
      Square(int l) : Rectangle("Square", l, l){};
      Square(const Square &q) : Rectangle(q){};
    };
    /*不规则四边形*/
    class Trapezoid : public Quadrangle
    {
    public:
      Trapezoid() : Quadrangle("Trapezoid"){};
    };
    /*梯形*/
    class Diamond : public Parallelogram
    {
```

```
    private:
      int bottomWidth;

    public:
      Diamond(int topWidth, int bottomWidth, int height)
          : Parallelogram("Diamond", topWidth, height),
bottomWidth(bottomWidth){};
      Diamond(const Diamond &q) : Parallelogram(q),
bottomWidth(q.bottomWidth){};
};
```

Container.h

```
    // container.h
    #pragma once
    #include "quad.h"
    #include <iostream>
    class Container
    {
    public:
      Container &push_back(QUADPTR p);
      size_t size() const;
      bool isEmpty() const;
      void empty();
    };
    struct Node
    {
      QUADPTR quad;
      Node *next;
    };
    class List : public Container
    {
    private:
      size_t len;
      Node *head, *tail;

    public:
      List() : len(0), head(nullptr), tail(nullptr){};
      // 复制构造函数
      List(const List &list);
      // 析构函数
      ~List();
      List &push_back(QUADPTR p);
      size_t size() const;
      bool isEmpty() const { return size() == 0; };
      void empty();
      void iterator();
    };
    bool Container::isEmpty() const
    {
      return false;
    };
    size_t Container::size() const
    {
      return 0;
    }
    void Container::empty()
    {
    }
    Container &Container::push_back(QUADPTR p)
    {
      return *this;
    }
    List &List::push_back(QUADPTR p)
    {
      Node *tmp = new Node(); // 为加入的 Node 结构体分配一个新的内存
```

```
      tmp->quad = p;          // 将申请 Node 结构体的 quad 初始化为 q
      tmp->next = nullptr;    // 将申请 Node 结构体的 next 初始化为 NULL
      if (tail == nullptr)
      {            // 判断链表是否为空
        head = tmp; //链表为空时，把 head 和 tail 都置为 tmp
        tail = tmp;
      }
      else
      {
        tail->next = tmp; // 将链表的尾部指针的 next 指向新申请的 Node 结构体
        tail = tmp;       // 将链表的尾部置为新申请的 Node 结构体
      }
      len += 1; // 加上新加入的 size
      return *this;
    }
    size_t List::size() const
    {
      return len;
    }
    void List::empty()
    {
      if (!isEmpty())
      {
        Node *before = head, *after;
        while (before != tail)
        {
          after = before->next;
          delete before;
          before = after;
        }
        delete before;
        head = nullptr;
        tail = nullptr;
      }
    };
    List::List(const List &list)
    {
      if (!list.isEmpty())
      {
        head = new Node();
        head->quad = list.head->quad;
        head->next = nullptr;
        Node *list_beCopy_iterator = list.head, *list_copy_iterator = head,
*list_new_node;
        if (list.head != list.tail)
        {
          while (list_beCopy_iterator != list.tail)
          {
            list_new_node = new Node();
            list_beCopy_iterator = list_beCopy_iterator->next;
            list_new_node->quad = list_beCopy_iterator->quad;
            list_copy_iterator->next = list_new_node;
            list_copy_iterator = list_new_node;
          }
          tail = list_copy_iterator;
          tail->next = nullptr;
        }
        else
        {
          tail = new Node();
          tail->quad = list.tail->quad;
          tail->next = nullptr;
        }
        len = list.size();
      }
```

```
      else
      {
        List();
      }
    };
    List::~List()
    {
      empty();
    };
    void List::iterator()
    {
      if (isEmpty())
      {
        std::cout << "链表是空的" << std::endl;
      }
      else
      {
        int count = 1;
        Node *tmp = head;
        while (tmp != tail && tmp != nullptr)
        {
          std::cout << "链表第" << count << "个是" << tmp->quad->what() <<
std::endl;
          count++;
          tmp = tmp->next;
        }
        std::cout << "链表第" << count << "个是" << tmp->quad->what() <<
std::endl;
      }
}
```

Misssion.h

```
    #include "container.h"
    class Mission
    {
    private:
      /* data */
    public:
      Mission(/* args */);
      ~Mission();
    };

    Mission::Mission(/* args */)
    {
      std::cout << "Mission Start" << std::endl;
      List list = List();
      QUADPTR quadrangle = new Quadrangle("Quadrangle");
      QUADPTR parallelogram = new Parallelogram("Parallelogram", 10, 20);
      QUADPTR trapezoid = new Trapezoid();
      QUADPTR square = new Square(520);
      QUADPTR diamond = new Diamond(13, 14, 52);
      list.push_back(quadrangle);
      list.push_back(parallelogram);
      list.push_back(trapezoid);
      list.push_back(square);
      list.push_back(diamond);
      std::cout << "开始遍历链表" << std::endl;
      list.iterator();
      std::cout << "链表长度为：" << list.size() << std::endl;
    }
    Mission::~Mission()
    {
      std::cout << "任务结束 Mission End" << std::endl;
}
```

实验结果图如图 8-8

```
ig\课程\c++程序设计\作业\习题汇总\8-8\"main
Mission Start
开始遍历链表
链表第1个是Quadrangle
链表第2个是Parallelogram
链表第3个是Trapezoid
链表第4个是Square
链表第5个是Diamond
链表长度为: 5
任务结束Mission End
```

图 8-8

- 9.8 把 Quadrangle 和 Container 容器改为抽象类。
  思路：把 Quadrangle 和 Container 中部分函数改为纯虚函数，在他们的派生类中重载这些函数。

  Container 中的变化，主要把 Container 中的函数设置为纯虚函数，其派生类已经重载这些函数，无需修改。

```
    class Container
    {
    public:
      virtual Container &push_back(QUADPTR p);
      virtual size_t size() const = 0;
      virtual bool isEmpty() const = 0;
      virtual void empty() = 0;
};
```

  Quad.h 中的变化,基类 Quadrangle 中的 area 和 draw 置为纯虚函数，派生类覆盖这两个函数，以形体类梯形 Diamond 为例

```
    class Quadrangle
    {
    protected:
      std::string tag;

    public:
      Quadrangle(std::string t) : tag(t){};
      std::string what() const { return tag; };
      virtual double area() const = 0; // 纯虚函数
      virtual void draw() const = 0;   // 纯虚函数
};
    /*梯形*/
    class Diamond : public Parallelogram
    {
    private:
      int bottomWidth;

    public:
      Diamond(int topWidth, int bottomWidth, int height)
          : Parallelogram("Diamond", topWidth, height),
bottomWidth(bottomWidth){};
      Diamond(const Diamond &q) : Parallelogram(q),
bottomWidth(q.bottomWidth){};
      double area() const { return (double)(width + bottomWidth) / 2 *
height; };
      void draw() const { std::cout << "draw Diamond" << std::endl; };
};
```

Mission.h 中的变化，因为基类 Quadrangle 为抽象类，无法实例化，所以去掉了对基类 Quadrangle 的实例化。

```
    #include "container.h"
    class Mission
    {
    private:
      /* data */
    public:
      Mission(/* args */);
      ~Mission();
    };

    Mission::Mission(/* args */)
    {
      std::cout << "Mission Start" << std::endl;
      List list = List();
      QUADPTR parallelogram = new Parallelogram("Parallelogram", 10, 20);
      QUADPTR trapezoid = new Trapezoid();
      QUADPTR square = new Square(520);
      QUADPTR diamond = new Diamond(13, 14, 52);
      list.push_back(parallelogram);
      list.push_back(trapezoid);
      list.push_back(square);
      list.push_back(diamond);
      std::cout << "开始遍历链表" << std::endl;
      list.iterator();
      std::cout << "链表长度为：" << list.size() << std::endl;
    }
    Mission::~Mission()
    {
      std::cout << "任务结束 Mission End" << std::endl;
}
```

实验结果图如 9-8

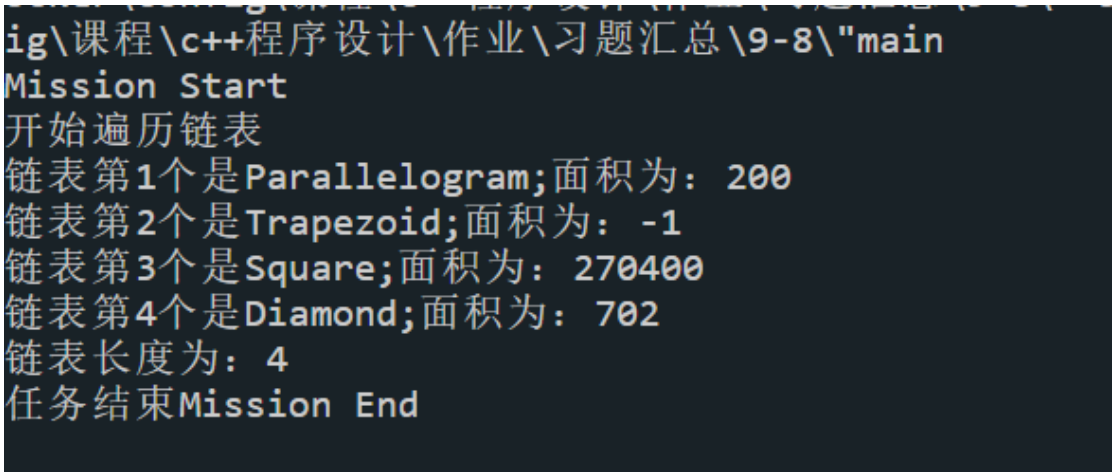

图 9-8

- 10.10 把 Container 成为类模板，其子类 List 也为模板类。
  思路，把凡是涉及到 QUADPTR 数据类型的类，结构体，方法对象等全部修改为模板。
  主要修改了 Container，List 以及 Mission 类

Container.h

```
// container.h
#pragma once
#include "quad.h"
template <typename T>
class Container
{
public:
  virtual ~Container(){};
  virtual Container &push_back(T p);
  virtual size_t size() const = 0;
  virtual bool isEmpty() const = 0;
  virtual void empty() = 0;
};
template <typename T>
struct Node
{
  T quad;
  Node *next;
};
template <typename T>
class List : public Container<T>
{
private:
  size_t len;
  Node<T> *head, *tail;
  template <typename U>
  friend List<U> operator+(const List<U> &list, U p);
  template <typename U>
  friend List<U> operator+(const List<U> &list1, const List<U> &list2);

public:
  List() : len(0), head(nullptr), tail(nullptr){};
  // 复制构造函数
  List(const List &list);
  // 析构函数
  ~List();
  List &push_back(T p);
  size_t size() const;
  bool isEmpty() const { return size() == 0; };
  void empty();
  void iterator();
};
template <typename T>
List<T> operator+(const List<T> &list, T p)
{
  List<T> t(list);
  return t.push_back(p);
};
template <typename T>
List<T> operator+(const List<T> &list1, const List<T> &list2)
{
  List<T> t(list1);
  if (list2.isEmpty())
    return t;
  Node<T> *temp = list2.head;
  while (temp != list2.tail)
  {
```

```
        t.push_back(temp->quad);
        temp = temp->next;
      }
      return t.push_back(temp->quad);
    };
    template <typename T>
    List<T> &List<T>::push_back(T p)
    {
      Node<T> *tmp = new Node<T>(); // 为加入的Node结构体分配一个新的内存
      tmp->quad = p;                // 将申请Node结构体的quad初始化为q
      tmp->next = nullptr;          // 将申请Node结构体的next初始化为NULL
      if (tail == nullptr)
      {             // 判断链表是否为空
        head = tmp; //链表为空时，把head和tail都置为tmp
        tail = tmp;
      }
      else
      {
        tail->next = tmp; // 将链表的尾部指针的next指向新申请的Node结构体
        tail = tmp;       // 将链表的尾部置为新申请的Node结构体
      }
      len += 1; // 加上新加入的size
      return *this;
    }
    template <typename T>
    size_t List<T>::size() const
    {
      return len;
    }
    template <typename T>
    void List<T>::empty()
    {
      if (!isEmpty())
      {
        Node<T> *before = head, *after;
        while (before != tail)
        {
          after = before->next;
          delete before;
          before = after;
        }
        delete before;
        head = nullptr;
        tail = nullptr;
      }
    };
    template <typename T>
    List<T>::List(const List<T> &list)
    {
      if (!list.isEmpty())
      {
        head = new Node<T>();
        head->quad = list.head->quad;
        head->next = nullptr;
        Node<T> *list_beCopy_iterator = list.head, *list_copy_iterator = head,
*list_new_node;
        if (list.head != list.tail)
        {
          while (list_beCopy_iterator != list.tail)
```

```
          {
            list_new_node = new Node<T>();
            list_beCopy_iterator = list_beCopy_iterator->next;
            list_new_node->quad = list_beCopy_iterator->quad;
            list_copy_iterator->next = list_new_node;
            list_copy_iterator = list_new_node;
          }
          tail = list_copy_iterator;
          tail->next = nullptr;
        }
        else
        {
          tail = new Node<T>();
          tail->quad = list.tail->quad;
          tail->next = nullptr;
        }
        len = list.size();
      }
      else
      {
        List();
      }
    };
    template <typename T>
    List<T>::~List()
    {
      empty();
    };
    template <typename T>
    void List<T>::iterator()
    {
      if (isEmpty())
      {
        std::cout << "链表是空的" << std::endl;
      }
      else
      {
        int count = 1;
        Node<T> *tmp = head;
        while (tmp != tail && tmp != nullptr)
        {
          std::cout << "链表第" << count << "个是" << tmp->quad->what();
          std::cout << ";面积为：" << tmp->quad->area() << std::endl;
          count++;
          tmp = tmp->next;
        }
        std::cout << "链表第" << count << "个是" << tmp->quad->what();
        std::cout << ";面积为：" << tmp->quad->area() << std::endl;
      }
}
```

Mission.h

```
    #include "container.h"
    class Mission
    {
    private:
      /* data */
    public:
      Mission(/* args */);
```

```
    ~Mission();
};

Mission::Mission(/* args */)
{
    std::cout << "Mission Start" << std::endl;
    List<QUADPTR> list1 = List<QUADPTR>();
    QUADPTR parallelogram = new Parallelogram("Parallelogram", 10, 20);
    QUADPTR trapezoid = new Trapezoid();
    QUADPTR square = new Square(520);
    QUADPTR diamond = new Diamond(13, 14, 52);
    list1.push_back(parallelogram);
    list1.push_back(trapezoid);
    list1.push_back(square);
    list1.push_back(diamond);
    std::cout << "开始遍历链表 list1" << std::endl;
    list1.iterator();
    std::cout << "链表 list1 长度为：" << list1.size() << std::endl;
    List<QUADPTR> list2 = List<QUADPTR>();
    QUADPTR parallelogram2 = new Parallelogram("Parallelogram", 5, 20);
    list2.push_back(parallelogram2);
    std::cout << "开始遍历链表 list2" << std::endl;
    list2.iterator();
    std::cout << "链表 list2 长度为：" << list2.size() << std::endl;
    std::cout << "遍历合并 list1 和 list2 的结果" << std::endl;
    List<QUADPTR> list3 = (list1 + list2);
    list3.iterator();
    std::cout << "链表 list2 长度为：" << list3.size() << std::endl;
}
Mission::~Mission()
{
    std::cout << "任务结束 Mission End" << std::endl;
}
```

实验结果如图 10-10

```
config\课程\c++程序设计\作业\习题汇总\10-10\"main
Mission Start
开始遍历链表list1
链表第1个是Parallelogram;面积为： 200
链表第2个是Trapezoid;面积为： -1
链表第3个是Square;面积为： 270400
链表第4个是Diamond;面积为： 702
链表list1长度为： 4
开始遍历链表list2
链表第1个是Parallelogram;面积为： 100
链表list2长度为： 1
遍历合并list1和list2的结果
链表第1个是Parallelogram;面积为： 200
链表第2个是Trapezoid;面积为： -1
链表第3个是Square;面积为： 270400
链表第4个是Diamond;面积为： 702
链表第5个是Parallelogram;面积为： 100
链表list2长度为： 5
任务结束Mission End
```

图 10-10

选做作业（其他班级布置的作业）

- 习题 1.4、1.5、1.6、2.5
- 习题 2.7-2.11；3.2-3.4、3.7；4.2-4.6
- 习题 5.2
- 习题 5.3-5.7
- 习题 6.1-6.2
- 习题 6.3-6.7
- 习题 7.1-7.4
- 习题 7.5，7.7，7.9
- 习题 8.1-8.5
- 习题 8.6-8.7
- 习题 9.2-9.3；9.6
- 习题 10.1-10.4
- 习题 10.5-10.7