**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。

1. **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：

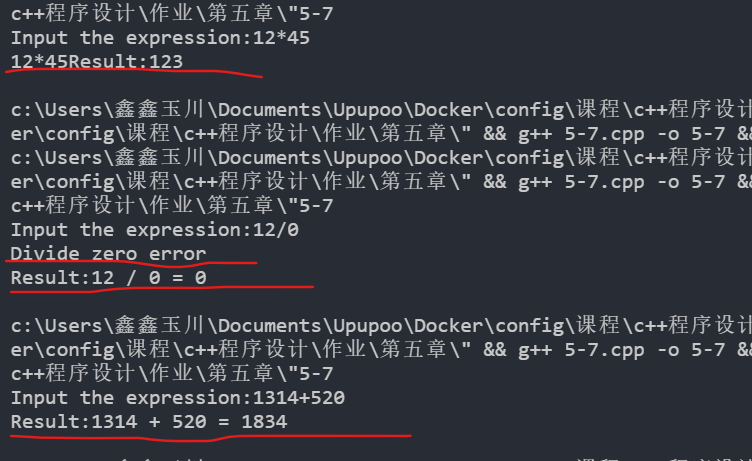


图5.7

* 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：

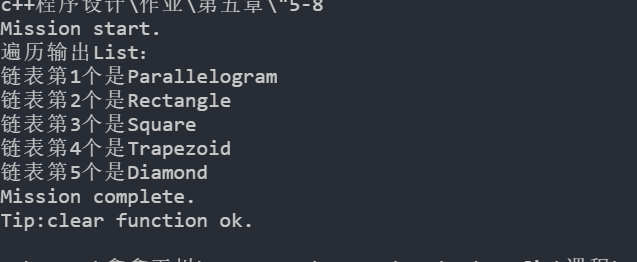


图5-8

* 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);  }; |

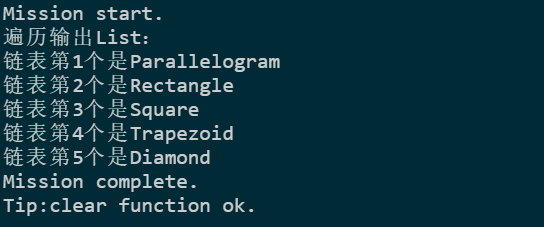
修改后的List类：

|  |
| --- |
| 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：



* 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：

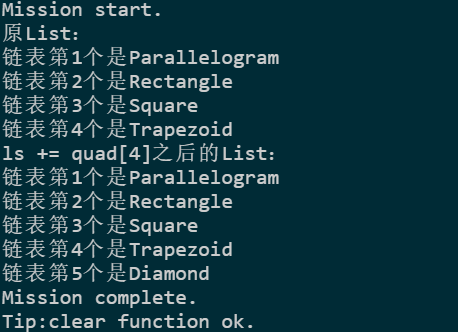


图7-11

* **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**

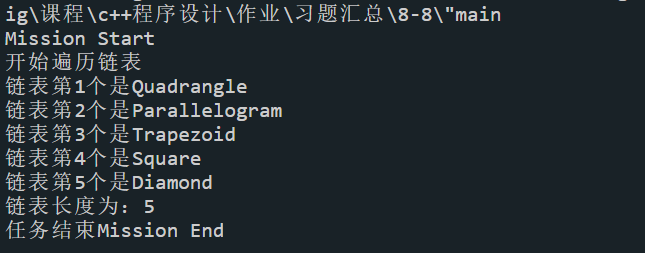


图8-8

* **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;**  }; |

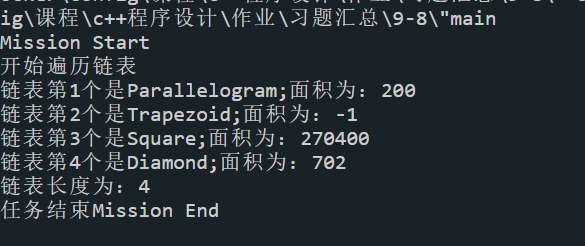
**Quad.h中的变化,基类Quadrangle中的area和draw置为纯虚函数，派生类覆盖这两个函数，以形体类梯形Diamond为例**

|  |
| --- |
| 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**

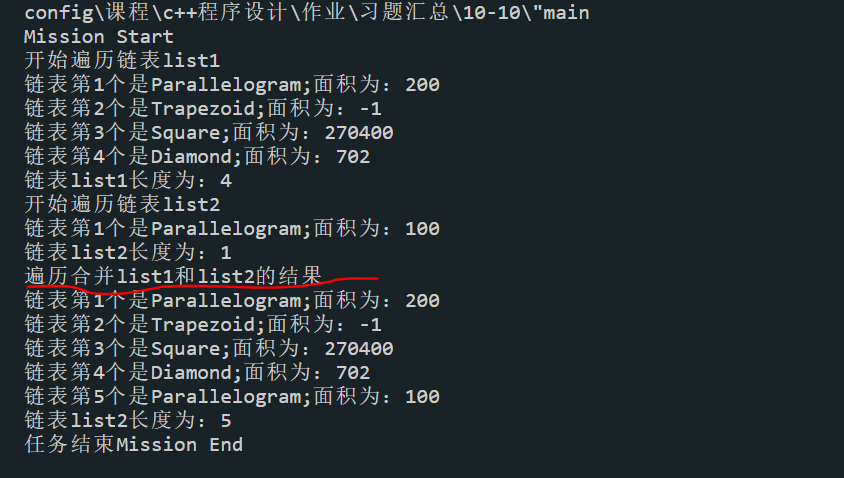
**Container.h**

|  |
| --- |
| // 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**



**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