acm.mipt.ru

олимпиады по программированию на Физтехе

Раздел «Язык Си» . OOP-Template :

Шаблоны

Очень часто при написании программ приходится сталкиваться со следующей проблемой. Существует множестов различных никак не связанных между собой классов, с объектами которых приходится выполнять абсолютно идентичные операции.

Например.

```
class Point{
int x,y;
 public:
  Point();
  Point(int, int);
  //... еще функции
  void print();
};

class MWeight{
   int kg; // килограммы
   int gr; // граммы.
public:
   MWeight(); // пустой мешок
   void getWeight(int p, int k);// получить вес риса
// Сложение мешка с мешком
   MWeight operator+(MWeight m1);
// Деление веса на целое число - получаем вес
   MWeight operator/(int n);
// оператор > . Если "наш" мешок больше чем параметр a, возвращает 1
// если нет - 0
   int operator>(MWeight a);
// печать
   void print();
};
```

Допустим имеются массивы объектов этих классов. Как правило, при работе с массивами возникает потребность поменять местами элементы этих массивов или просто поменять содержимое двух объектов.

Для решения этой задачи необходимо реализовать две различные функции: для работы с классом **Point** и для работы с классом **MWeight**.

```
void swapPoint(Point& a, Point& b){
    Point c;
    c = a;
    a = b;
    b = c;
};
```

```
void swapMWeight(MWeight& a, MWeight& b){
    MWeight c;
    c = a;
    a = b;
    b = c;
};
```

Как видно, все операции в этих функциях абсолютно идентичны. Различны только типы объектов, с которыми они совершаются.

Возникает желание написать каким-то образом написать одну функцию вместо двух.

Чтобы иметь такую возможность, нужно сформулировать требования к классам объектов, с которыми такие функции могут работать:

1. функция не должна зависеть от внутреннего строения объекта, с которым работает

2. интерфейс классов объектов должен предоставлять весь набор операций над объектами, необходимый для выполенения операций, используемых в функции.

Как видно, в этих функциях **swapXXX** используется только одна операция, это - присваивание. Значит во всех классах эта операция должна либо адекватно работать как операция присваивания "по умолчанию", либо переопределена для нужд класса.

в результате может получиться некоторая обобщенная функция-шаблон, которая отвечает вышеописанным требованием, и у которой параметром является **тип** или **класс**

Функции-шаблоны

Запишем функцию **swapT** как шаблон.

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Класс точек на плоскости
class Point{
  int x,y;
public:
    Point();
    Point(int x1, int y2);
// Функции класса ....
// .....
};

// Класс треугольников
class Triangle{
   Point ugol[3]; // массив точек - вершины
public:
   Triangle();
   Triangle(Point, Point, Point);
// Еще функции класса....
// .....
   void print();
};

// Описание функции-шаблона.
// слово template означает, что это шаблон.
// в угловых скобках указывается параметр шаблона и формальное имя параметра
// В данном случае параметр шаблона - имя класса, и формальное имя этого класса T
// вместо T при вызове функции C++ выяснит определит какой класс будет использоваться
// T& a и T& b - обычные параметры функции. T c - локальна япеременная класса T
// при подстановке конкретных имен объектов в качестве параметров C++  выясняет тип этих объектов,
// и формирует функцию уже для работы с конкретным классом.

template<class T> void swapT(T& a, T&b){
      T c ;
      c = a; // предполагается, что для всех объектов семейства class T
      a = b; // оператор = определенн
      b = c; // и работает правильно
};


// В классе Triangle есть нединамический массив
// C++ столь "любезен", что обеспечивает правильное копирование
// нединамических массивов  в объектах..
// Поэтому для класса Triangle тоже можно использовать стандартный оператор копирования

int main(){
   Point a(3,7), b(8,10), c(11,12); // точки
   Point d(44 ,1), f(0,0),z(-2,-1);
// Треугольники инициализируются точками
   Triangle tr1(a,b,c), tr2(d,f,z);

// меняем точки местами
// Как только функция-шаблон получает параметры, C++ сразу определяет класс или тип
// этих параметром. И функция начинает работу с переменными этого класса или типа
   swapT(a,d);
   a.print();
   d.print();
   cout<<endl;

//меняем местами треугольники
  swapT(tr1, tr2);
  tr1.print();
  tr2.print();

 int  k =3, m = 9;
```

```
// меняем местами целые числа
  swapT(k,m);
  cout<<k<<' '<<m<<endl;
}
```

Сразу заметим, что эта функция-шаблон будет работать и для других классов. Однако нужно учесть, что каждая функция-шаблон предъявляет особые требованияк классам, с которыми она может работать. **В данном случае необходимо иметь корректно описанный оператор копирования.**

Для функций-шаблонов могут использоваться несколько различных параметров:

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
class Point{
  int x,y;
  int dist2;
public:
    Point();
    Point(int x1, int y2);
// Здесь полезные функции класса

// Для использования в наших шаблонных функциях,
// оператор > нужен обязательно
    int operator>(const Point&);

};

class Rect{
// здесь атрибуты класса
public:
// Здесь функции

// Оператор > должен быть определен
  int operator>(const Rect&);

};

// Шаблонная функция поиска максимального элемента в массиве

// Здесь два параметра - тип элементов массива и
// n - их количество
// Функция может искать максимальный элемент если:
// определен оператор присваивания и определен оператор >
// для элемнитов всех классов, которые предполагается обрабатывать этой функцией

template<class T, int n>  T maxT(T a[n]){
        T max;
        max = a[0];
        for(int i=0; i< n; i++){
           max = (a[i] > max)? a[i] : max;
        }

        return max;
};

// Оператор > для класса Point
int Point::operator>(const Point& a){
 // Код для сравнения точек
};

// Оператор > для класса Rect
int Rect::operator>(const Rect& a){
 // Код для сравнения прямоугольников
};

int main(){
  Point a[4]; // массив точек
// Заполнение значений атрибутов точек

  Point mx; // точка на максимальном расстоянии от 0

// поиск точки на максимальном расстоянии от 0
// указываем тип элементов массива и его размер
  mx = maxT<Point,4>(a);


// массив из целых чисел
  int z[7]={3,5,1,0,7,100,12};
```

```
  int m;
// поиск максимального элемента из массива целых чисел
// указываем тип элементов массиваи его размер
  m = maxT<int,7>(z);
  cout<<m<<endl;

  Rect rz[10];
// Заполнение значений атрибутов примоугольников

  Rect rmax; // самый большой прямоугольник

// Поиск самого большого прямоугольника

  rmax = maxT<Rect,10>(rz);
}
```

Пример использования шаблооной функции cmpPT

```
#include <cstdlib>
#include <iostream>


using namespace std;
typedef int (*cmpQ)(const void*, const void*);

class Point{
   int x, y;
   public:
      Point();
      void set(int, int);
      int operator>(const Point&);
      int operator==(const Point&);
      ostream& put(ostream&);
};

Point::Point(){
   x = y = 0;
};

void Point::set(int ax, int ay){
   x = ax;
   y = ay;
};

int Point::operator>(const Point& a){
   int dist1, dist2;
   dist1 = x*x + y*y;
   dist2 = a.x * a.x + a.y * a.y;
  cout<< dist1<<' '<<dist2<<endl;
  return (dist1 - dist2 > 0)? 1 : 0;
};

int Point::operator==(const Point& a){
      return (x == a.x && y == a.y)? 1 : 0;
};

ostream& Point::put(ostream& s){
   return s <<'('<<x<<','<<y<<") ";
};

ostream& operator<<(ostream& s, Point a){
    return a.put(s);
};

template<class T> T maxT( T& a, T& b){
   return (a > b)? a : b;
};

/* Эта "обычная" функция, которую можно использовать для стандартного
   qsort и в нашей шаблонной функции maxTP
int cmpP(const void* a, const void* b){
   Point *p1 = (Point*)a, *p2 = (Point*) b;
  if((*p1) == (*p2)) return 0;
   return ((*p1) > (*p2))? 1 : -1;
};
*/

// Шаблонная функция сравнения
// При инициализации (при вызове) она "превратится" в "обычную" функцию
// типа int (*cmp)(const void*, const void*)
```

```
template<class T>int cmpPT(const void* a, const void* b){
   T *p1 = (T*)a, *p2 = (T*) b;
  if((*p1) == (*p2)) return 0;
   return ((*p1) > (*p2))? 1 : -1;
};

// шаблонный класс maxTP, который использует int (*cmp)(const void*, const void*)
template<class T> T maxTP( T* a, T* b, int (*cmp)(const void*, const void*)){
   if( cmp(a,b) > 0)
     return *a;
   return *b;
};


int main(){
   Point a, b, c;
   a.set(3, 4);
   b.set(0, 15);
   c.set(0, 5);
  Point z[5];
  z[0].set(3,4);
  z[1].set(-3,4);
  z[2].set(3,-4);
  z[3].set(0,-10);
  z[0].set(-10,-1);

// пример вызова шаблонной функции maxTP и передача
// указателя на функцию cmpPT, порождаемого шаблоном
// Так как до вызова maxTP и cmpTP не существуют, для
// инициализации в вызове явно указывается параметр - имя класса.
  cout<<maxTP<Point>(z, z + 3, cmpPT<Point>)<<endl;

// пример для массива целых чисел. Используем ту же
// шаблонную функцию.
  int v[5]={5,7,8,1,-23};
  cout<<maxTP<int>(v + 2, v + 4, cmpPT<int>)<<endl;
  return 0;
};
```

Задача 1

Определить массивы для массива пар точек (с расстоянием между ними), для ионов (с массой иона). Написать шаблонные функцию для:
1. определения минимального элемента
2. сортировки элементов (можно использовать qsort)
Описать требования к классам элементов массивов для использования этих функций

Классы-шаблоны

Кроме шаблонных функций можно описать и шаблонные классы. Как правило, это классы-контейнеры или классы структурно похожие на контейнеры.

Рассмотрим шаблонный класс **Cbuff**. Этот шаблон позволяет работать с множеством объектов, загружаемых в кольцевой буфер. Мы предполагаем, что для всех таких классов переопределен оператор вывода * <&lt *

```
#include <cstdlib>
#include <iostream>


using namespace std;

//Класс Obj - элементы этого класса необходимо поместить в буфер
class Obj{
   int n;

public:
     Obj();
     Obj(int);
     void print();
      ostream& put(ostream&);
};
class Villager{
   string name;
   int status;
public:
      Villager(string, int);
      Villager();
```

```
      void set(string, int);
      ostream& put(ostream&);
};

// Класс-контейнер для элементов Obj
template< class T>class CBuff{

/*
  Внутренний класс Elem,  описывающий каждый узел
  кольцевого буфера
*/
  class Elem{
  public:

// Объекты, помещаемые в буфер
    T obj;

// Указатель на следующий элемент буфера
    Elem *next;

// Количество элементов в буфере
    int nm;
  };

// Два указателя на кольцевой список: на первый элемент
// и  на последний
  Elem * frst;
   Elem * last;

// Общее количество элементов
  int kolvo;

 public:
       class Iterator;
// Конструктор буфера
        CBuff();

/*
  Добавление элемента в буфер.
  Добвление будем производить после последнего элемента
*/
       void add(T);

/*
  Получить указатель на первый элемент кольцевого списка.
  Заметим, что возвращаемый тип данных (указатель на Elem)
  может быть доступен только функциям или друзьям класса CBuff
*/
       Elem* begin();

// Получить указатель на последний элемент списка
       Elem* end();

// Печать всего списка (для отладки)
        void print();
         Elem * find(const T&);
         void delete(Elem*);
      void delElem(Iterator);
/*
  Класс Iterator - внутренний класс (НЕ ОБЪЕКТ!!) класса CBuff.
  Находится в области public, значит может быть доступен для
  конструирования объекта и доступа к его методам.
  Класс Iterator:
  1. получает указатель на конкретный элемент списка,
     состоящего из узлов типа Elem
  2. перемещается к следующему элементу в списке
  3. возвращает указатель на объект Obj, помещенный в буфер
  4. позволяет выполнить операцию "разыменовывания" для
     указателей на объект
  5. позволяет выпольнить сранение ("нет, это не он") для решения
     достигнут ли нужный элемент при просмотре списка
*/
        class Iterator{

 // Указатель на элемет списка
            CBuff<T>::Elem *point;
            int idx; // для проверки конца обхода

         public:
```

```
// Конструктор
            Iterator();

// Операторприсваивание. В качестве параметра указатель на Elem
            Iterator* operator=(Elem*);

// Перемещение к следующему элементу в списке
            Iterator* operator++(int);

// Возвращение указателя на объект, помещенный в список
            T* operator->();

// Получение доступа к самому объекту. Оператор "разыменовывания"
            T operator*();

// Сравнение двух элементов ("нет, это не он")
            int operator!=(Elem*);

        };

};

//_____________________________________________________________________
// Реализация класса Obj
//_____________________________________________________________________

Obj::Obj(){
  n=0;
};
Obj::Obj(int a){
    n = a;
cout<<"obj param:"<<n<<endl;
};

void Obj::print(){
    cout<<n<<endl;
};

ostream& Obj::put(ostream& s){
   return s<<n;
};

ostream& operator<<(ostream& s, Obj b){
      return b.put(s);
};
ostream& operator<<(ostream& s, Obj* b){
      return b->put(s);
};

//_____________________________________________________
// Villager
//_____________________________________________________________________

Villager::Villager(string z, int r){
      name = z;
      status = r;
};
Villager::Villager(){

};

void Villager::set(string z, int r){
      name = z;
      status = r;
};

ostream& Villager::put(ostream& s){
   return s<<name<<'('<<status<<") ";
};


ostream& operator<<(ostream& s, Villager vl){
   return vl.put(s);
};

ostream& operator<<(ostream& s, Villager* vl){
   return vl->put(s);
};
//_____________________________________________________________________
// Реализация класса CBuff
```

```
//_____________________________________________________________________________

// Коструктор. Предполагаем, что вначале буфер пуст.
template< class T> CBuff<T>::CBuff(){
     frst = 0;
     kolvo = 0;
};
template< class T> void CBuff<T>::delElem(CBuff<T>::Iterator it){
      cout<<"delele:" << *it <<endl;
};

// Добавление элемента
template< class T> void CBuff<T>::add(T z){
    struct Elem *new_p;
    kolvo++;

// Выделяем память под новый элемент
    new_p = new Elem;

// Заполняем ее полезной информацией
    new_p->obj = z;
    new_p->nm = kolvo;

// Проверяем, пуст буфер или нет
    if (!frst){

//  Если пуст - это элемент становится первым
//  и последним (ссылка на него же)
        frst = new_p;
        frst->next = frst;
        last = frst;

    } else{

// Если не пуст
// ему присваивается ссылка на первый элемент
         new_p->next = last->next;

// последний элемент теперь ссылается на новый
         last->next = new_p;

//  новый становится последним
         last = new_p;
    }

};

// Получение указателя на первый элемент
template <class T> typename CBuff<T>::Elem* CBuff<T>::begin(){
    kolvo = 0;
   return frst;
};

// Получение указателя на последний элемент
template <class T> typename CBuff<T>::Elem* CBuff<T>::end(){
   return last;
};


// Печать всего буфера
template <class T> void CBuff<T>::print(){
  Elem *p = frst;
// Печатаем, если список не пуст
  if (frst != 0){
//      (frst->obj).print();
         cout<<frst->obj<<' ';
      p = frst->next;

     while (p != frst){
//         (p->obj).print();
            cout<<p->obj<<' ';
         p = p->next;
     };
  }
}

//_____________________________________________________________________________
// Реализация класса CBuff::Iterator
//_____________________________________________________________________________
```

```
// Конструктор. Указатель вначале 0
template <class T> CBuff<T>::Iterator::Iterator(){
    point = 0;
};

// Оператор присваивания.
template <class T> typename CBuff<T>::Iterator* CBuff<T>::Iterator::operator=(CBuff<T>::Elem* z){
     idx = 1;
     point = z;
     return this;
};

// Оператор ++
template <class T> typename CBuff<T>::Iterator* CBuff<T>::Iterator::operator++(int){

// Если список не пуст, перемщаем указатель по списку
        if(point)
        point = point->next;
        return this;
};



// Оператор "укаатель на объект"
template <class T> T* CBuff<T>::Iterator::operator->(){

       return &(point->n);
};

template <class T> T CBuff<T>::Iterator::operator*(){
     if(point)
      return point->obj;
     else
       exit(1);
};

// Оператор "неравенство"
/*
 Так как буфер - кольцевой, то начало и конец списка совпадают
 к тому же могут быть удалены некоторые элементы и начальный
 элемент может изменится
*/
template <class T> int CBuff<T>::Iterator::operator!=(CBuff<T>::Elem* check){
 //   cout<<"check:"<<kolvo<<endl;
    bool ret;
//    cout<<"ret:"<<ret<<endl;

// Проверка в первый ли раз указатель "смотрит"
// на головной элемент
    ret = (idx);

// kolvo=0, значит перехода на следующий элемент не будет
    if(point == check)
        idx--;
    return ret;
};

// Тестирование буфера
int main(int argc, char *argv[])
{
    char c;
    CBuff<Villager> a;
      CBuff<Obj> ab;//  список пуст
    ab.add(7); // добавление элементов
    ab.add(8);
    ab.add(5);
     CBuff<Obj>::Iterator tobj;
      std::cout << "Obj:\n";
      for(tobj = ab.begin(); tobj != ab.end(); tobj++){
//         t->print();
         cout<< *tobj <<' ';
    }

    Villager t1;
      t1.set("Зюзя", 0);
    a.add(t1);
      t1.set("Кузя", 1);

     a.add(t1);
      t1.set("СеменСеменыч", 0);
```

```
      a.add(t1);
//    cout<<t1;
// Создание итератора
    CBuff<Villager>::Iterator t;
    cout<<"итератор\n";

// связывание итератора (это другой объект!!!) с буфером
    t = a.begin();
// t-> - указывает на первый элемент списка и, далее, вызов print()
//    t->print();

      cout<< *t <<endl;
     t++;
     a.delElem(t);
// Пример использования итератора в цикле
    for(t = a.begin(); t != a.end(); t++){
//        t->print();
         cout<< *t <<' ';
    }


    return 0;
}
```

Задачи

Задача 1.0--

Дописать функции **find()** и **delete()**

Задача 1.1

Описать шаблон функции **delElem(Iterator)**

Задача 1.2 (Купец)

Купец нанял N работников (не менее 1 и не более 100). Все работники делали одинаковую работу. По окончании работ купец расплатился с работниками монетами. Но он дал каждому монет не поровну, а как-нибудь: кому-то много, а кому-то мог вообще не заплатить.

Работники возмутились и стали требовать справедливости.

Купец обещал уладить дело.

Он поставил всех работников в круг. Затем у каждой пары работников он брал деньги, складывал и делил пополам. Купец начинал круг с пары 1-2, затем 2-3, ... и заканчивал круг N-1

Если оставалась лишняя монета, купец забирал ее себе, а половинки раздавал рабочим.

После каждого такого "уравнивания" все проверяли поровну ли у них монет. Если у всех было поровну, процесс заканчивался.

Рабочие, получив РАВНОЕ количество монет уходили домой, а купец получал еще немножечко денег.

Используя шаблон **CBuff** написать программу, которая вычисляет сколько монет получил купец и сколько каждый работник.

Задача 1.3 (Новость)

Каждый житель некоторого острова мог быть либо рыцарем, либо лжецом. Рыцари говорят правду. Лжецы лгут.

Рыцарь может произнести ложь только в том случае, если он не знает, что это ложь. Когда какой-нибудь рыцарь произносит ложь, он тут же погибает. Лжец может произнести правду лишь в том случае, если не знает, что это правда. Если же лжец произносит правду, то с этого момента он становится рыцарем.

Никто из жителей не разговаривал сам с собой. Если на острове один житель, он молчит.

Дома всех жителей были построены так, что образовывали окружность, поэтому каждый житель этого острова имел двух соседей: справа и слева.

Если же справа/слева от жителя А сосед умирал, то правый/левый сосед умершего становился правым/левым соседом жителя А.

Например:

C->B->A->K->O

А имеет соседей В - левый и К - правый.

Если у А умрет сосед В, то С станет левым соседом А.

Каждый день один раз в день житель заходил к одному своему соседу слева и делился с ним своей новостью.

Однажды царь сообщил одному из своих придворных фразу, ложность которой определялась, тем, кто был царь. Эту фразу жители острова передавали друг другу как новость. Каждый, услышавший

фразу, считает ее истинной.

Утром новость попадала к царю, если на острове было хотя бы 2 жителя.

В тайной канцелярии города известно кто из жителей был лжец, а кто рыцарь; а так же у кого какие соседи справа.

Новость ходила по острову **N** дней.

Используя класс **Village** и шаблон класса **CBuff** напсатть программу для моделирования жизни на острове в течение этих **N** дней.

Данные о жителях вводить из файла!!! -- TatyanaOvsyannikova2011 - 11 Nov 2015