acm.mipt.ru

олимпиады по программированию на Физтехе

Раздел «Язык Си» . OOP-Instrumental_b :

Сложные объекты.

Объекты-атрибуты. Доступ к объектам.

Объекты могут входить в состав других объектов в качестве атрибутов.

При этом часто эти объекты порождаются как динамические в процессе работы программы.

Как правило, объект, который содержит атрибут-объект может использовать все открытые функции своего арибута и его открытые атрибуты.

При этом атрибут-объект ничего "не знает" о свем хозяине, если не принять специальные меры.

Например.

```
class B;
class A{
// Функции класса A могут обращаться к функциям объектов класса B
  B b; // объект- атрибут
// порождается в момент работы конструктора объектов класса A
// так как в этот момент порождаются все атрибуты класса A
// В C++ для порождение объектов всегда происходит через вызов конструктора объекта
public:
   A();
   void print();
}
// Объекты класса B не "видят" объекты класса A
// они могут обмениваться сообщениями только с помощью других объектов
// если это запрограммировано
class B{
  int x,y;
public:
   B();
// другие функции
  void put();
};
```

Динамические объеты-атрибуты могут появляться, удаляться во время работы функций объекта-хозяина. При этом нужно не забывать, что порождение объектов должно происходить через вызов конструктора (оперетор **new**), а удаление через вызов деструктора (оператор **delete**).

Именно поэтому использование функций **malloc**, **free** и т.д. приводит к неопределенному поведению программы. Не вызывается конструктор и деструктор, и, поэтому, не выполняются действия, определенные в них: инициализация переменных, удаление динамических объектов, закрытие файлов и т.д.

Рассморим пример класса, объеты которого содержат динамический массив. Для массива должна выделяться память в нужном количестве, ее размер может меняться и необходимо удалять эту выделенную память.

Переопределение оператора копирования, Копирующий конструктор

Для копирования объектов в C++ используется стандартный оператор копирования. При работе стандартного оператора копирования из одного объекта в друой копируются:

- все значения артибутов встроенных типов в соответствии с их именами. При этом необходимо учитывать, что копируются ЗНАЧЕНИЯ указателей, а не объекты, которые расположены по этим адресам.
- все значения атрибутов, для которых определен собственный **оператор копирования** в соответствии с тем как он реализован

Если атрибутом является массив или динамический объект, создаваемый в процессе конструирования или работы функции-метода другого объекта, то для таких сложных объектов в классе должен быть реализован **оператор копирования.**

Кроме того, оператор копирования необходим, когда необходимо произвести "присваивание" объектов несовпадающих типов. Например, объекту типа "Дробь" требуется присвоить число типа **int.**

Задача Арифметика

Класс **Arithmetic** служит для работы с длинной арифметикой. Тип **unsigned char** служит для хранения числа, записанного в 100-ричной системе счисления. Каждый элемент - число, не превышающее **99.** Само число задается строкой цифр.

Для решения этой задачи нужен динамический массив (для хранения разрядов числа по основанию 100). Причем его размер зависит от количества разрядов в числе.

При этом, конечно понятно, что можно присваивать друг другу объекты одного класса. Но, для данной задачи есть необходимость присвоить объекту класса **Arithmetic** также просто целые числа или числа типа **float** с отбрасывание дробной части.

В C++ есть возможность переопределить оператор копирования и копирующий конструктор с любым типом входного параметра.

Необходимо реализовать методы класса, описанные в интерфейсе:

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

using namespace std;

class Arithmetic{
  unsigned char* digit; // для хранения числа
  int n;    // размер числа

 public:
       Arithmetic(); // конструктор

// инициализирущий конструктор
// в строке содержатся только цифры.
// Предполагается, что на строку символов до вызова
// конструктора уже выделено достаточное количество памяти и
// конец строки уже обозначен '\0'
       Arithmetic(const char*);

/*
  Деструктор. Необходим, чтобы объект
  "умер достойно".
*/
       ~Arithmetic();

// Копирующий конструктор
       Arithmetic( const Arithmetic&);

// Оператор копирования
```

```
// Оператор возвращает ССЫЛКУ, чтобы можно было
// использовать объект "как обычно" в выражениях и
// конструкциях
       Arithmetic& operator=(const Arithmetic&);

// Оператор копирования
       Arithmetic& operator=(int n);

       void print();
};
```

Заметим, что динамически выделенную память при создании или использовании объекта необходимо освобождать в момент, когда удаляем объект.

Существует встроенный деструктор, который освобождает память, занятую встроенными типами данных. Однако, если использовать встроенный деструктор, то освободится лишь память, занятая под указатель **digit**. То есть указатель прекратит свое существование, а выделенная память останется недоступной и неудаляемой. Она будет занимать память, выделенную программе, что потом может привести к нехватке памяти и ее фрагментации.

Чтобы очистить динамическую память, занятую встроенными переменными, необходимо прописать соответствующие действия в деструкторе, который тоже является программируемым методом.

Заметим еще, что числа, получившиеся в разных объектах могут иметь разную длину.

Существует встроенный оператор копирования, который копирует из одного объекта в другой встроенные значения переменных встроенных типов. То есть, если есть два объекта с указателями на динамическую область памяти, то встроенный оператор копирования скопирует адрес из указателя первого объекта в указатель другого. Таким образом, оба объекта будут указывать на одну и ту же область памяти. Это недопустимо. Копированием мы собирались получить две копии со своей собсвенной областью памяти для всех атрибутов.

```
#include <string.h>
#include <iostream>
#include "long.h"

/*
 Конструктор по умолчанию.
 Будем считать, что по-умолчанию, число - 0
*/
Arithmetic::Arithmetic(){
   n = 1;
// Динамически выделяем память под один элемент.
   digit = new unsigned char;
   digit[0] = 0;
   cout<<"Constructor\n";
};

/*
 Инициализирующий конструктор.
 число задается строкой цифр.
*/
Arithmetic::Arithmetic(const char* s){
    int i;
    int len = strlen(s);
    n = ( len % 2 )? (len >> 1) + 1: len >> 1;

// Выделение памяти под массив
    digit = new unsigned char[n];
    len--;

// Число записывает "задом-наперед" для вычислений
    for(i = 0;i < n;i++){
      digit[i] = s[ len-- ] - '0';
      if( len < 0 )
       break;
      digit[ i ] += ( s[ len-- ] - '0' ) * 10;
```

```
   }
};

void Arithmetic::print(){
   int i;
   cout<<"len="<<n<<endl;
   for(i=0;i<n; i++){
     cout<<(int)digit[i]<<' ';
   }
   cout<<endl;
};

/*
  Копирующий конструктор.
При создании объекта можно использовать данные
другого объекта.

Тогда необходимо выделить соответствующую память
и скопировать  данные.
*/
Arithmetic::Arithmetic(const Arithmetic& a){
   digit = new unsigned char[ a.n];
   n=a.n;
   memcpy(digit,a.digit, sizeof(unsigned char)*n);
};

/*
 Деструктор. Во время работы деструктора будут выполняться
инструкции, написанные в нем. То есть освободится динамическая
память, на которую указывает digit.

Деструктор вызыватся: при вызове delete, при завершении функции,
в которой этот объект используется как локальный, при вызове
деструктора объекта, в котором содержится этот объект.
*/
Arithmetic::~Arithmetic(){
  delete[] digit;
  cout<<"Dectructor!!\n";
};

/*
 Перегрузка оператора копирования.

Здесь нужно учесть, что у объекта уже выделена память
под хранение числа. Поэтому ее необходимо освободить
и выделить новую, соответствующего размера
*/
 Arithmetic& Arithmetic::operator=(const Arithmetic& a){
   delete[] digit;
    digit = new unsigned char[ a.n ];
   n = a.n;
   cout<<"размер: "<<sizeof( unsigned char)*n<<endl;
   memcpy( digit, a.digit, sizeof( unsigned char)*n );
    return *this;
};

/*
 Присваивание объекта "несоответствующего" типа.
*/
 Arithmetic& Arithmetic::operator=(int a){
// На самом деле это число может содержать больше
// разрядов по основанию 100
// Исправить код самостоятельно
   delete[] digit;
    digit = new unsigned char;
   n = 1;
   digit[0] = a;
    return *this;
};
```

Пример использования методов класса **Arithmetic**

```
#include "long.h"


int main(){

// Динамический объект
  Arithmetic *a = new Arithmetic();

// Инициализирующий конструктор.
  Arithmetic b( "12345" );

// Копирующий конструктор
  Arithmetic c( b );
  a->print();

/*
  Удаление динамического объекта.
Вызывается деструктор этого объекта.
*/
  delete a;
  b.print();
  c.print();
  Arithmetic d;

// Оператор копирования
  d = c;
  d.print();

  c = 5;
  c.print();
/*
  При завершении функции main вызовутся деструкторы
всех локальных объектов.
*/
 return 0;
}
```

Задачи

Задача 1

Для класса **Arithmetic** реализовать методы, объявленные в интерфейсе:

```
class Arithmetic{
  char* digit; // для хранения числа
  int n;    // размер числа

 public:
       Arithmetic(); // конструктор

// инициализирущий конструктор
       Arithmetic(unsigned char*);

/*
  Деструктор. Необходим, чтобы объект
  "умер достойно".
*/
       ~Arithmetic();

// Копирующий конструктор
       Arithmetic( const Arithmetic&);
// Инициализирующий коструктор (числом)
       Arithmetic( int);
// Оператор копирования
```

```
       Arithmetic& operator=(const Arithmetic&);
// Копирование числа
       Arithmetic& operator=(int);
       Arithmetic& operator=(float);
/*
 Оператор сложения. Учесть, что перед операцией
числа могут быть разного размера
*/

    Arithmetic operator+(const Arithmetic&);

    Arithmetic& operator+=(const Arithmetic&);

// Сложение с числом.
    Arithmetic operator+(int);
// Инкремент

   Arithmetic& operator++(int); // постфиксный
   Arithmetic& operator++();    // префиксный
// Вычитание
   Arithmetic operator-(const Arithmetic&);
   Arithmetic operator-(int);
// Декремент

   Arithmetic& operator--(int); // постфиксный
   Arithmetic& operator--();    // префиксный

// Сложная задача
//  Умножение на число
   Arithmetic operator*(int);

// Очень сложная задача
//  Деление на число
   Arithmetic operator/(int);
    void print();
};
```

Задачи

Задача 2

Реализовать следующие операторы для работы с классом "Дроби":
1. полноценные операции сложения, вычитания, умножения, деления, инвертирования дробей (например, сложение и дробью, и с числом)
2. операции сравнения дробей: равенство, больше, меньше (так же и с дробью, и с числом). Если истина, возвращаем 1, ложь - 0

Задача 3 Интервалы времени

Написать два класса:

```
class TimeInterval{
 CTime begin; // начало промежутка
 CTime end;  // конец промежутка
 public:
  TimeInteraval();
  TimeInterval(CTime, CTime);
// Сравнение двух промежутков времени:
// если не пересекаются, возвращаем 0
// если пересекаются
// если полностью совпадают, возвращаем 10
// если первый полностью входит во второй – 12
// если второй входит в первый – 13
// если начало первого раньше – 14
//  если начало второго раньше – 15
```

```
// если конец первого позже – 16
//  если конец второго позже – 17
  int operator>(const TimeInterval);
 // печать в поток
  ostream& put(ostream&);
  friend class timeLine;
};

class TimeLine{
// список интервалов.
// все интервалы должны быть упорядочены по времени
//  и не пересекаться.
// Если два интервала пересекаются, то они преобразуется в один интервал,
// началом которого выбирается самое ранне время, а концом – самое позднее
// если пересекаются более двух интервалов, то началом выбирается
// самое раннее время, а концом самое  позднее
// Например: были интервалы (10:00-12:00, 13:00-13:20, 14:05-14:10)
// Добавляем интервал (11:00-14:00)
// В результате получает такой список: (10:00-14:00, 14:05-14:10)
  TimeInterval *timeLn;
  int n;
public:
   TimeLine();
// Добавить интервал. Правила образования интервалов как
// описано выше
   TimeLine& operator+=(TimeInterval);
// Удалить интервал.
// Если этот интервал не пересекается ни с каким,
// то список не изменяется.
// Если пересекается, то из списка удаляется пересекаемая часть
// Например: список: (10:00-14:00, 14:05-14:10), удаляем интервал (11:00-12:00)
// получаем (10:00-11:00, 12:00-14:00, 14:05-14:10)
   TimeLine& operator-=(TimeInterval);
// Добавить еще один список. Интервалы при этом
// получаются по правилам, описанным выше
   TimeLine& operator+(const TimeLine&);
//  удалить список интервалов. Правила
// образования новых интервалов описаны выше
   TimeLine& operator-(const TimeLine&);
// печать списка на поток.
   ostream& put(ostream&);
};
```

При работе с многомерными массивами часто используется конструкция подобная приведенной ниже:

```
 int **a;
 a = new int [10];
 ...
 for(i = 0; i < 10; i++){
   a[i] = new int[10];
   ......
 }
 int k = a[3][4];
```

Несмотря на то, что к элементам такого двумерного массива удобно обращаться, используя естественную нумерацию строк и столбцов, его использование может стать не очень эффективным.

Если обработке подлежат матрицы или изображения большого размера, то для более эффективного использования оперативной памяти для них нужно выделять непрерывную область памяти.

Но тогда получается, что приходится работать с одномерным массивом. При этом пересчитыать индекся не удобно.

Рассмотрим одну из реализаций класса для двумерной матрицы.

Будем считать, что данные размещаются в файле в следующем формате:

1. первая строка содержит два целых числа - размер матрицы (первое ***n*** - по вертикали, второе ***m*** - по горизонтали)
2. далее идут ***n x m*** чисел, записанных через разделитель

Например:

3	3	
1	2	3
4	5	6
7	8	9

Класс **Matr1** может быть реализован, например, таким способом:

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

using namespace std;

/*
   Двумерная матрица. Для размещения выделяется непрерывная область
   динамической памяти.
   Чтобы можно было обращаться к элементам матрицы с использованием
   двумерных индексов a[i][j] переопределяем оператор [].

*/
class Matr1{
  int *a;  // это указатель на область данных
  int m, n;  // размер n - по вертикали, m - по горизонтали
public:

    Matr1();
/*
  Деструктор нужен обязательно так как при удалении объекта
  матрицы необходимо очищать и динамическую память тоже
*/
    ~Matr1();
// чтение данных из потока
    istream& get(istream&);
// запись данных в поток
    ostream&  put(ostream&);
// переопределение оператора [].
// возвращает *int, чтобы можно было еще раз применить []
    int *operator[](int);
};
// Конструктор.
Matr1::Matr1(){
   a = 0;
   n = m = 0;
};
// Деструктор. Освобождаем выделенную область
Matr1::~Matr1(){
 if (a)
   delete[] a;
};

// Выделение динамической памяти и чтение данных из потока
istream& Matr1::get(istream& s){
     s>>n>>m;
     int i,j;

// проверка, если память уже использовалась,
// удаляем ее и выделяем вновь другую
     if(a) delete[] a;
     a = new int[ n * m ];
// чтение из потока
    for( i = 0; i < n * m; i++){
      s>>a[i];
```

```
    }
   return s;
};
// Запись в поток "красиво"
ostream& Matr1::put(ostream& s){
      int x,y;
     for( y = 0; y < n; y++){
       for(x = 0; x < m; x++)
        s<<a[ y * m + x]<<' ';
       s<<endl;
     }
   return s;
};
// возвращение указателя на новую "строку"
int* Matr1::operator[](int t){
     return a + m * t;
};
// переопределение оператора ввода для Matr1
istream& operator>>(istream &s,Matr1& m){
     return m.get(s);
};
// переопределение оператора вывода для Matr1
ostream& operator<<(ostream& s, Matr1& m){
     return m.put(s);
};

// Тестирование
int main(){
// Объект Matr1
   Matr1 m ;
// Данные в файле
   ifstream f("m.dat");
// Получение данных из файла
   f>>m;
// печать матрицы (отладка)
   cout<<m;
// применение оператора [] к матрице
    int *t = m[2];
   cout<<"указатель "<<t<<" значение:"<<t[0]<<endl;
// двойное применение операторов []: сначала к матрице,
// а потом сразу к указателю:
   int k = m[2][1];
   cout<<"breck:"<<k<<endl;
    return 0;
}
```

Задача 4.

Для работы с <<псевдографикой>> используется <<изображение>>: **n** строк по **m** символов в каждой. При этом белый цвет - <<.>>, черный - <<*>>.

Реализовать класс **Pict:**

```
class Pict{
// Указатель на память под картинку
  char *pic;
// Размер
  int n,m;
public:
   Pict(int, int);
// Деструктор
   ~Pict();
// Копирующий конструктор
   Pict(const Pict&);
/* заполнение данными
*/
   void getData(ifstream& f);
//Обращение к строке.
```

```
   char* operator[](int);
// "переворот" зеркально (вертикально)
   Pict* operator!();
// копирование
   Pict* operator=(const Pict&);
// сравнение
   int operator==(const Pict&);
   void print();
};
```

Массивы указателей на функции.

Предположим, что имеется ряд объектов, которые должны выполнять различные функции в зависимости от своего состояния. Кроме этой "мелочи", объекты в остальном совершенно похожи. Поэтому нет смысла описывать несколько разных классов для таких объектов.

Проблему можно решить, созданием массива ссылок на функции одного интерфейса. Различаться будут только имена функций.

Пример

```
#include <iostream>
#include <cstdlib>
using namespace std;

// int (*step)(int, int) - описание укзателя на функцию,
// которая получает два параметра int и возвращает int
// Описываем тип данных step - указатель на вышеупомянутые функции (такого формата)
typedef   int (*step)(int, int);

// Описание функции go1 (подходит по формату, что делает сейчас не важно)
int go1(int x, int y){
    if (x>0 && x<20 && y >0 && y <20){
         cout<<"Ходи go1!!"<<endl;
         return 1;
    }
      return 0;
};

// Описание функции go2 ( тоже подходит по формату, делает другое)
int go2(int x, int y){
    if ((x==-1 || x==1) && (y >0 && y <20)){
         cout<<"Ходи go2!!"<<endl;
         return 1;
    }
      return 0;
};

int  main(){

// описание массива указателей на функции
// step объявили раньше через typedef
   step go[2];

// присваивание элементу массива указателя на функцию go1
// имя функции – это указатель на нее
   go[0] = go1;
// присваивание элементу массива указателя на функцию go2
// имя функции – это указатель на нее
   go[1] = go2;

        int res;
// Вызов первой функции (параметры нужны!!)
   res = go[0](10,15);
// Вызов второй функции (параметры нужны!!)
   go[1](-1,4);
```

```
    return 0;
}
```

Задача 5. Игровое поле.

Для описания игрового поля необходимо реализовать следующие классы:

Класс описания различных фигур. У нас будут стена и фигура. Стена просто занимает клетку, а фигура может ходить по заданным правилам.

```
class GameItem{
  char what; //обозначение типа фигуры
// описание указателя на функцию "хода"
  typedef   int (*step)(int, int);
// описание указателя на функцию отображения:
// стена - ||, а фигура - V
  typedef ostream& (*show)(ostream&);

// массив функций "хода"
   step go[2];
// массив функций отображения
   show look[2];

  public:
   GameItem();
// инициализирующий конструктор: 0 - стена, 1 - фигура
   GameItem(char);
// установить значение для фишки
        void setMe(char);
// поверить возможность сделать ход на данную клеточку,
// по правилам для этой фишки
   int canGo(int, int);

// функция вывода в поток
   ostream& showMe(ostream&);
};

// переопределение оператора вывода в поток для ЛЮБОГО потока
ostream& operator<<(ostream&, GameItem& );
```

Класс игрового поля. Предполагается, что дано поле размером NxM?. При этом точка (0,0) находится в середине поля. Отсчет производится от нее в положительную и отрицательную стороны.

```
class GamePole{
// Объявление Cell как синоним для указателя на GameItem
   typedef GameItem* Cell;
// предполагается динамический массив для указателей на GameItem
// так как сами фишки во время игры не изменяются, можно хранить только две
// и пользоваться указателями на эти две фишки
   Cell *pole;
// массив всевозможных фишек для игры
   GameItem gi1[2];
// размер поля
   int n,m;
// можно добавить своих атрибутов для удобства программирования
   public:
// Конструктор для поля. Параметры - размеры
     GamePole(int, int);
// Деструктор необходим для удаления динамических объектов
        ~GamePole();
// получить данные из открытого файла
     int getData(ifstream&);
// оператор [] - вохвращает указатель на ряд  относительно точки (0,0)
     Cell* operator[](int);
// проверка можно ли переместить фишку с клетки (a,b) на клетку (c,d)
     int canGo(int a, int b, int c, int d);
// вывод поля в поток
```

```
     ostream& showMe(ostream&);
};

// переопределение оператора вывода для поля.
ostream& operator<<(ostream&, GamePole& );
```

Данные находятся в файле в формате **:** в каждой строке – данные об одной фишке**.** первыое целое число тип фишки**,** далее два числа – координаты фишки относительно центра

```
1 2 2
0 0 1
0 0 2
1 0 0
```

Пример записи реализации оператора **[].**

```
GamePole::Cell* GamePole::operator[](int k){
// Здесь нужно правильно вычислить ряд и вернуть СЕРЕДИНУ ряда
// В этом примере вычисляется "чего-нибудь"
      int xc = m/2;
      int yc = n/2;
      int ct = yc + xc;
    return pole + yc - k;
};
```

Пример использования реализованного оператора **[]** для данного объекта**.**

```
int GamePole::canGo(int x1, int y1, int x2, int y2){
// Разыменовываем указатель на текущий объект
// После этого можно применять к нему []
    if((*this)[y1][x1]==0)
         return 0;
// Помним, что в поле содержится УКАЗАТЕЛЬ на фишку
      return ((*this)[y1][x1])->canGo(x2 - x1, y2 -y1);
};
```

В этом файле описаны две стенки на клетках (0,1) и (0,2) и две фишки на клетках (2,2) и (0,0) -- TatyanaOvsyannikova2011 - 17 Feb 2016