Struct point

{ int x,y}

Напишем функцию:

Void print(point x) …

Если это файлы, то тоже не возникает проблем с соединением их посредством подключения хедера.

Struct point

{

Int x,y;

Void print();

}

В си это недопустимо, а в с++ возможно. Из минусов необходимо где-то реализовывать функцию.

Void point::print()

{..}

**Ввели понятие Классы**

Class point

{

Int x,y;//x и y не доступны в остальной программе

Public;//обеспечивает доступ до всех полей – всем владельцам обкта, ещё есть privat и protected, по одной штуке модифактору доступа приват и паблик;

Void print()// стараться называть функцию в классе – методом.

{…}

Void setx(int \_x)//\_x – не хороший стиль, потому что это сливается с подчёркиванием общего текста

{x=\_x;}

Bool sety(int \_y)// без типа данных

{

If ((\_y>10) || (\_y<-10)) {return 1;}

Y=\_y;

Return 0;

}

Int getx() {return x;}

Int gety() {return y;}

}

У этого подхода есть минус: принудительная инициализация значения.

Point a,b;

a.setx(3);

b.sety(5);

int k=a.getx();

k=b.get.x();

k будет равно 5.

Class x

{

Int k;

Public;

Void set(int a)

{

this->k = a;

}

}

Неявно создаётся конструкция типа:

X \* const this;

Константный указатель, который нельзя менять. Эту конструкцию нельзя писать, она прописывается «виртуально». Оно используется там, где при создание класса необходим указатель на себя.

Class q

{

Q \*p;

Q \*s;

Public;

…

Void append(q \*t)

{

t->s=s;

t->p=this;

s->p=p;// вроде было так((

}

}

Class c1

{

int k;

public

void set(int x) {k=x;};

int get() const {return k;}; //обеспечивает физическую неизменность класса

}

Неизменность класса нужна для безопасности класса. Есть ещё логическая неизменность – не меняется часть данных в соответствии с логикой программы.

Void m(c1 & t1,const c1 & t2)

{

T1.set(5);

Int i=t1.get();

T2.set(3);// тут возникнет проблема, потому что стоит const.

Int j=t2.get();

}

Class c2

{

Int x;

Public;

Void s1(int i) const

{ x=i;}; // работать не будет((

Void s2(int i) const

{ ((c2\*)this)->x=i;}// а это работает)

}

Struct p {int a;};

Class c3

{

P\* x;

Public;

Void s(int i) const

{ x->a=i;}

}

Остаётся проблема инициализации переменных.

Class c3

{

Int a,b,c;

Public;

С3(int \_a,int \_b,int \_c) {a=\_a …};

C3(int x) {a=x,b=x;c=x+3;}

C3() {a=0;b=0;c=0;}

С3(const c3 & t)// const обеспечивает неизменность входной переменной

{ }

}

C3 t1=c3(5,n,3);// допустима такая форма записи

C3 t2(3);// тоже работает

C3 t3;// конструктор по умолчанию

C3 t4=t1;// конструктор копирования, предоставляемый компилятором (?)

С3\* t5=new с3(5)//new – оператор, который создает класс и возвращает указатель на него

C3 \* t6=t5;// не вызывается конструктор по умолчанию, так как мы работаем с указателями и это не работает)

C3 \* t7 = new c3(t5);

Здесь достаточно free для освобождения памяти.

Struct p{ int a;};

Class c4

{

P \* x;

Public

C4(int i)

{ };

}

C4 a;// будет создан кусок памяти с адресом x, который будет указывать на область памяти a

Деструктор

~C4()(Delete x;}// при выходе из области видимости, автоматически освободится память от x

Приватные конструкторы нужны, когда мы внутри объекта создаем копию объекта.

Class c5

{

Int a;

Public

Void m()

{ a++;}

}

C5 t;

t.m();//произойдёт обращение к методу

надо после класса написать «подстановку»

inline void c5::m()

{a++;}

Методы в 1 строчку лучше делать inline, остальные пихать в класс.

Class point

{

Friend void move();

}

Class vector

{

Friend void move();

}

Оба класса можно создать. В обоих классах есть приватные поля и они друг друга не видят.

Void move(point & , vector & )

При использования friend приватные поля используемых классов становятся доступны для использования