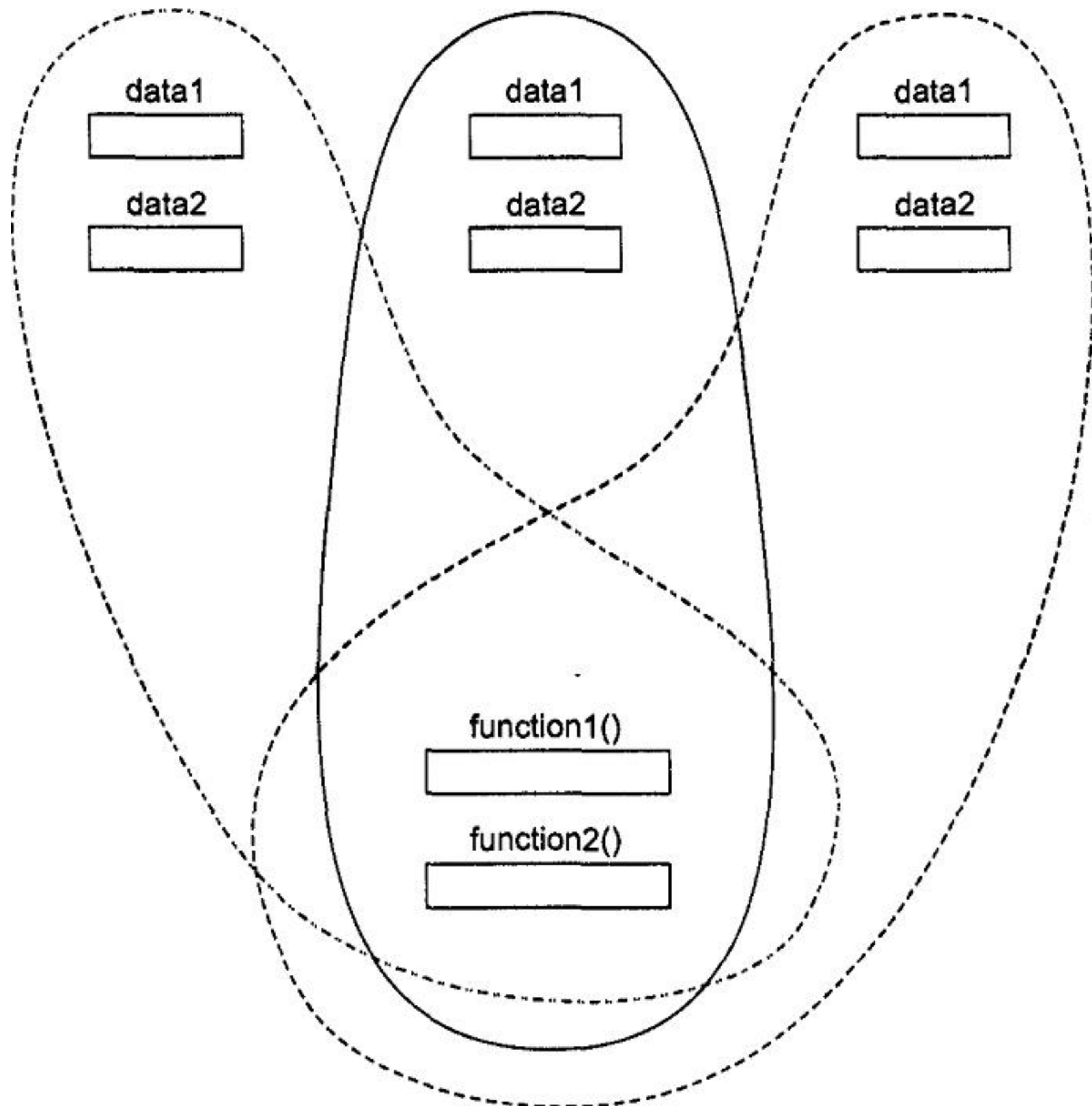
Классы, объекты и память

- Каждый объект имеет собственные независимые поля данных.
- С другой стороны, все объекты одного класса используют одни и те же методы.
- Методы класса создаются и помещаются в память компьютера всего один раз — при создании класса.


data1
data2
data1
data2
data1
data2
function1()
function2()

Статические данные класса

- Если поле данных класса описано с ключевым словом static, то значение этого поля будет одинаковым для всех объектов данного класса.
- Статические данные класса полезны в тех случаях, когда необходимо, чтобы все объекты включали в себя какое-либо одинаковое значение.
- Статическое поле по своим характеристикам схоже со статической переменной: оно видимо только внутри класса, но время его жизни совпадает со временем жизни программы.
- Таким образом, статическое поле существует даже в том случае, когда не существует ни одного объекта класса.

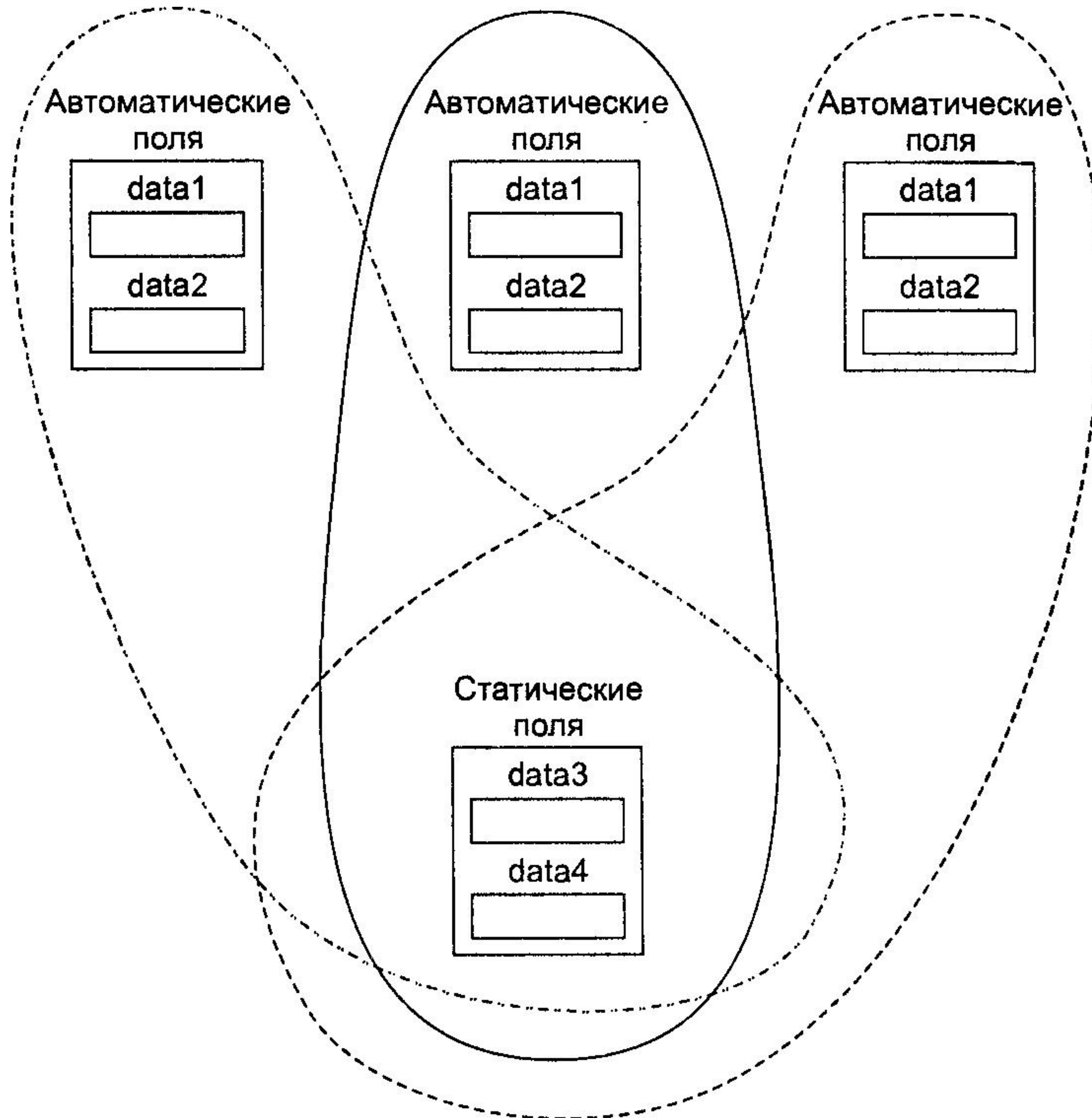
Пример использования статических полей класса

```
class foo
{
 private:
   static int count; // общее поле для всех объектов
     // (в смысле "объявления")
 public:
  foo() // инкрементирование при создании объекта
     { count++; }
  int getcount() // возвращает значение count
     { return count; }
};
```

```
int foo::count = 0; // *определение* count
//////////////////////////////////////////////////
int main()
{
•    foo f1, f2, f3; // создание трех объектов
•    // каждый объект видит одно и то же значение
•    cout << "Число объектов: " << f1.getcount() << endl;
•    cout << "Число объектов: " << f2.getcount() << endl;
•    cout << "Число объектов: " << f3.getcount() << endl;
•    return 0;
•    }
```

Эта программа выполнит следующее:

Число объектов: 3
Число объектов: 3
Число объектов: 3


Автоматические поля
data1
data2
Автоматические поля
data1
data2
Автоматические поля
data1
data2
Статические поля
data3
data4

Статические методы класса

```
class foo
{
 private:
   static int count; // общее поле для всех объектов
   // (в смысле "объявления")
 public:
   foo(){ incObj(); } // инкрементирование при создании    объекта
   static incObj() {return ++count;}
   int getcount(){ return count; }
};
```

- Ключевое слово *static* ставится перед типом метода. В основном используются для работы со статическими полями класса.

Ограничения на статические методы

- Имеют прямой доступ лишь к статическим данным класса
- Указатель this не виден в статических методах, поэтому в их определении нельзя обратиться к нестатическим данным класса
- Невозможно объявить в классе статическую и нестатическую версии функции
- Статические методы не могут быть виртуальными и не могут быть константными

Константные методы

- Константные методы отличаются тем, что не изменяют значений полей своего класса.
- Для того чтобы сделать функцию константной, необходимо указать ключевое слово const после прототипа функции, но до начала тела функции.
- Если объявление и определение функции разделены, то модификатор const необходимо указывать дважды — как при объявлении функции, так и при ее определении.
- Те методы, которые лишь считывают данные из поля класса, имеет смысл делать константными, поскольку у них нет необходимости изменять значения полей объектов класса.

```
class _3d
{
 double x, y, z;
public:
 _3d();
 ~_3d();
 double mod () {return sqrt (x*x + y*y +z*z);}
 double projection (_3d r) {return (x*r.x + y*r.y + z*r.z) / mod();}
 void set (double newX, double newY, double newZ)
      { x = newX; y = newY; z = newZ;}
 };
```

```
class _3d
{
 double x, y, z;
public:
 _3d();
 ~_3d();
 double mod () const  {return sqrt (x*x + y*y +z*z);}
 double projection (_3d r) const  {return (x*r.x + y*r.y + z*r.z) / mod();}
 void set (double newX, double newY, double newZ)
      { x = newX; y = newY; z = newZ;}
 };
```

Константные аргументы методов

```
class _3d
{
 double x, y, z;
public:
 _3d();
 ~_3d();
 double mod () const  {return sqrt (x*x + y*y +z*z);}
 double projection (const _3d& r) const  {return (x*r.x + y*r.y
+ z*r.z) / mod();}
 //{ r.x =0;} ошибка: нельзя изменить r
 void set (double newX, double newY, double newZ)
      { x = newX; y = newY; z = newZ;}
 };
```

Константные объекты

```
class _3d
    {
      double x, y, z;
    public:
      _3d ();
      _3d (double initX, double initY, double initZ);
      ...
       void set (double newX, double newY, double newZ)
      { x = newX; y = newY; z = newZ;}
    };


main()
    {
     _3d B (3,4,0); //создается объект B и происходит инициализация его
    элементов
     // B.x = 3.0, B.y = 4.0, B.z = 0.0
    const _3d X1 (1,0,0); //создаётся константный объект X1
     // X1.x = 1.0, X1.y = 0.0, X1.z = 0.0
    B.set(2,3,4);
    // X1.set(2,3,4); ошибка: метод set() неконстантный
    }
```

В конструкторах предпочитайте инициализацию присваиванию

- В конструкторах использование инициализации вместо присваивания для установки значений переменных-членов предохраняет от ненужной работы времени выполнения при том же объеме вводимого исходного текста.

```
class A {    string sl_, s2_;public:    A() { sl_ = "Hello, "; s2_ = "world"; }};
```

В действительности сгенерированный код конструктора выглядит так, как если бы было написали:

```
A() : sl_(), s2_() { s1 = "Hello, "; s2_ = "world"; }
```

Инициализация переменных-членов в списке инициализации дает код, более лаконичный и обычно более быстрый:

```
A() : sl_("Hello, "), s2_("world") { }
```

Определяйте и инициализируйте переменные-поля в одном порядке

- Переменные-члены всегда инициализируются в том порядке, в котором они объявлены при определении класса; порядок их упоминания в списке инициализации конструктора игнорируется. Убедитесь, что в коде конструктора указан тот же порядок, что и в определении класса.

```
class Employee {
    string email_, firstName_, lastName_;
public:
    Employee( const char* firstName, const char* lastName )  :
    firstName_(firstName), lastName_(lastName) ,
    email_(firstName_+"."+lastName_+"@company.com") {}};
```

Этот код содержит ошибку. Поскольку член email_ объявлен в определении класса до first_ и last_, он будет инициализирован первым и будет пытаться использовать еще не инициализированные поля. Более того, если определение конструктора находится в отдельном файле, то выявить такое

Пример построения классов и наследования

Класс, моделирующий построение физических пикселов на экране:

```
struct Point
    {
    int X;
    int Y;
    };
```

Пиксел на экране монитора, кроме координат своего положения, обладает еще и возможностью "светиться". Расширим структуру:

```
enum Boolean {false, true}; // false = 0, true = 1
struct Point
    {
     int X;
     int Y;
     Boolean Visible;
    };
```

```
enum Boolean {false, true}; // false = 0, true = 1
class Point
    {
    protected:
      int X;
      int Y;
      Boolean Visible;
    public:
      int GetX(void) { return X; }
      int GetY(void) { return Y; }
      Boolean isVisible (){ return Visible;}
      Point (const Point& cp); // прототип конструктора копирования
      Point (int newX =0, int new Y =0); // прототип конструктора
    };

Point :: Point (int NewX, int NewY) // конструктор
    {
     X = newX; Y = newY; Visible = false;
    }
Point :: Point (const Point& cp) // конструктор копирования
    {
     X = cp.X; Y = cp.Y; Visible = cp.Visible;
    }
```

```
Point Center(320, 120); // объект Center типа Point
Point *point_ptr; // указатель на тип Point
point_ptr = &Center; // указатель показывает на Center
```

Задание аргументов по умолчанию при описании прототипа конструктора дает возможность вызывать конструктор без аргументов или с неполным списком аргументов:

```
Point aPoint ();
Point Row[80]; // массив из объектов типа Point
Point bPoint (100);
```

```
class Point
    {
     ...
    public:
     ...
     void Show();
     void Hide();
     void MoveTo(int newX, int newY);
    };

void Point::Show()
    {
     Visible = true;
     putpixel (X,Y,getcolor());
    }
void Point::Hide()
    {
     Visible = false;
     putpixel (X,Y,getbkcolor());
    }
void Point::MoveTo (int newX, int newY)
    {
     Hide ();
     X = newX;
     Y = newY;
     Show ();
    }
```

```
main()
    {
    int graphDr = DETECT, graphMode;
     initgraph ( &graphDr, &graphMode,"");

    Point pointA (50,50);
    pointA.Show ();
    pointA.MoveTo (100,130);
    pointA.Hide ();

     closegraph();
    }
```

класс Circle для окружности

```
class Circle: public Point
{
  int Radius; // private по умолчанию
public:
  Circle (int initX, int initY, int initR);
  void Show ();
  void Hide ();
  void Expand (int deltaR);
  void Contract (int deltaR);
  void MoveTo (int newX, int newY);
};

Circle::Circle (int initX, int initY, int initR) // конструктор
        :Point (initX, initY) // вызов конструктора базового класса
{
 Radius = initR;
}
```

```
void Circle::Show ()
    {
    Visible = true;
    circle (X,Y, Radius);
    }
void Circle::Hide ()
    {
     Visible = false;
     unsigned int tempColor = getcolor ();
     setcolor (getbkcolor());
     circle (X,Y, Radius);
     setcolor (tempColor);
    }
void Circle::Expand (int deltaR)
    {
     Hide();
     Radius += deltaR;
     Show();
    }
void Circle::Contract (int deltaR)
    {
     Expand (-deltaR);
    }
void Circle::MoveTo (int newX, int newY)
    {
     Hide ();
     X = newX;
     Y = newY;
     Show ();
    }
```

```
main()
    {
     int graphDr = DETECT, graphMode;
     initgraph ( &graphDr, &graphMode,"");

     Circle C (150,200,50); // создать
    объект окружность с центром в
    т.(150, 200) и радиуса 50
     C.Show(); // показать окружность
     getch();
     C.MoveTo (300,100); // переместить
     getch();
     C.Expand (50); // растянуть
     getch();
     C.Contract (70); // сжать
     getch();

     closegraph();
    }
```

Совместимость типов

Расширенная совместимость порожденного типа со всеми типами предка имеет три формы:

- между экземплярами объектов,
- между указателями объектов,
- между формальными и фактическими параметрами.

Однако во всех трех формах порожденные классы можно свободно использовать вместо классов предка, но не наоборот.

Например,

```
Point APoint, *ptrPoint;
Circle ACircle, *ptrCircle;
```

При наличии этих объявлений следующие присваивания являются законными:

```
APoint = ACircle;  ptrPoint = ptrCircle;
```

Обратные присваивания незаконны.

Родительскому объекту можно присваивать объект любого порожденного им класса.

void Proc (Point param) //фактические параметры могут иметь тип Point, Circle и любой другой порожденный от них тип.

Дружественные функции

Задача: лежит ли некоторый объект типа Point внутри области, занимаемой некоторым объектом типа Circle? (находится ли точка внутри окружности?)

Для решения этой задачи нужны:
- координаты точки,
- координаты центра,
- радиус окружности.

Все необходимые для этого данные лежат в областях *private* и *protected*.

Вариант решения:

```
Boolean Circle::IsInside(Point &P)
{
 if ((X-P.GetX())*(X-P.GetX())+(Y-P.GetY())*(Y-P.GetY())<= R*R) return
true;
  else return false;
}
```

```
class Point
    {
    //...
     friend Boolean IsInside (Circle &C, Point &P);
    };
class Circle: public Point
    {
    //...
     friend Boolean IsInside (Circle &C, Point &P);
    };
```

Объявление дружественной функции

friend Boolean IsInside (Circle &C, Point &P);

можно с одинаковым эффектом вставлять в любом месте в декларации класса, это может быть любой раздел (*public*, *protected* или даже *private*).

```
Boolean IsInside(Circle &C, Point &P)
    {
     if ((C.X-P.X)*(C.X-P.X)+(C.Y-P.Y)*(C.Y-P.Y)<= C.R * C.R) return true;
      else return false;
    }
```

- Функция-элемент одного класса может быть дружественной иному классу

```
class x
    {
    //…
    void f();
    };
class y
    {
    //…
    friend void x::f();
    };
```

Возможен и вариант, когда все функции одного класса - дружественны другому классу.

```
class y
    {
    //…
    friend class x;
    }
```

Переопределение операторов с помощью дружественных функций

```
class _3d
    {
    //...
     _3d operator + (_3d b);
    };

_3d _3d::operator + (_3d b)
    {
     _3d c;
     c.x = x + b.x;
     c.y = y + b.y;
     c.z = z + b.z;
     return c;
    }
```

```
class _3d
    {
    //...
     friend _3d operator + (_3d &a,
    _3d &b);
    };

_3d operator + (_3d &a, _3d &b)
    {
     _3d c;
     c.x = a.x + b.x;
     c.y = a.y + b.y;
     c.z = a.z + b.z;
     return c;
    }
```