Технология программирования

Объектно-ориентированное программирование

Введение

Определения понятия **объект** с разных точек зрения:

1) с точки зрения *человеческого восприятия*, объект – это осязаемый или видимый предмет, имеющий определенное поведение;

2) с точки зрения *исследователя* в некоторой предметной области объект – это опознаваемый предмет или сущность (реальная или абстрактная), имеющие важное функциональное значение в данной предметной области;

3) с точки зрения *проектирования* системы, объект – это сущность, обладающая состоянием, поведением и индивидуальностью.

4) с точки зрения *программирования*, объект – это совокупность данных (переменная), существующая в машинном представлении как единое целое, допускающее обращение по имени или указателю.

Группы объектов могут иметь схожие черты, которые составляют понятие класса. ***Класс*** – это *описание* структуры и поведения схожих объектов. Будем использовать понятия *"экземпляр класса"* и *"объект"* как синонимы.

Объявление классов и создание объектов

```
class Time
{
private:
    int hour; //объявление поля
    int minute;
    int second;
public:
    //Объявления методов
    Time(); //конструктор
    void setTime(int, int, int);
    void print24();
    void print12();
};
```

```
//Определения методов класса
Time::Time()
{
    // Инициализация поля данного класса
    hour=0;
    minute=0;
    second=0;
}
void Time::setTime(int h, int m, int s)
{
    hour=h;
    minute=m;
    second=s;
}
void Time::print24()
{
    cout<<hour<<":"<<minute<<":"<<second;
}
void Time::print12()
{
    if(hour==0||hour==12)
      cout<<12;
    else
      cout<<hour%12;
    cout<<":"<<minute<<":"<<second;
}
```

Объявление классов и создание объектов

```
//головная функция
int main()
{
   Time t; //вызов конструктора класса Time::Time()
   t.setTime(14,32,11);
   t.print24();
   t.print12();
   Time *tptr;
   tptr=new Time; //динамическое выделение памяти под
                  //объект и вызов конструктора
   (*tptr).setTime(1, 2, 3);
   delete tptr; //удаление объекта, на который
                //указывает tptr
   return 0;
}
```

Поведение объекта

Операция (действие, сообщение, метод, функция) – это определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию.

Категории операций над объектами:

1) *функция управления* – операция, которая изменят состояние объекта;

2) *функция доступа* – операция, дающая доступ для определения состояния объекта без его изменения (операция чтения);

3) *функция реализации* – операция над информацией из внешних источников, не изменяющая состояния данного объекта;

4) *вспомогательная функция* – операция, используемая вышеперечисленными операциями, и не предназначенная для самостоятельного использования другими объектами;

5) *конструктор* – операция инициализации объекта;

6) *деструктор* – операция разрушения объекта.

Поведение объекта

```
class Stack
{
private:
   int * s;  //указатель на первый элемент массива
   int top;  //указатель на вершину стека
   int size; //размер стека
public:
   void push(int i);  //управление
   int pop();         //управление
   int isEmpty();          //доступ
   int isFull();      //доступ
   void copy(Stack *st); //реализация
   Stack(int sz);     //конструктор
   ~Stack();      //деструктор
};
```

Поведение объекта

```
Stack::Stack (int sz)
{
  size=sz;
  //массив элементов
  s=new int[size];
  //несуществующий индекс массива
  top=-1;
};
Stack::~Stack()
{
  delete []s;
};
int Stack::pop()
{
  //значение на вершине стека
  int last=s[top];
  top=top-1;
  //возвращение значения
выталкиваемого
  //элемента
  return last;
};
int Stack::isEmpty()
{
  if (top>=0) return 0;
  else return 1;
};
```

```
int Stack::isFull()
{
  if (top>=size-1) return 1;
  else return 0;
};
void Stack::copy(Stack *st)
{
  while(!st->isEmpty())
    st->pop();
  for(int i=0; i<=top; i++)
    st->push(s[i]);
}
void Stack::push(int i)
{
  if(!isFull())
  {
    top=top+1;
    s[top]=i;
  }
};
```

Поведение объекта

```
void main()
{
  Stack st1(10); //создание локального объекта st1
  Stack *pst2;
  pst2=new Stack(10); //создание объекта в куче
  st1.push(77);
  pst2->push(33);
  pst2->push(st1.pop());//изменение состояния стека
  pst2->copy(&st1);
  delete pst2; //вызов деструктора Stack::~Stack()
} //здесь вызывается деструктор для локального объекта
```

Индивидуальность объекта

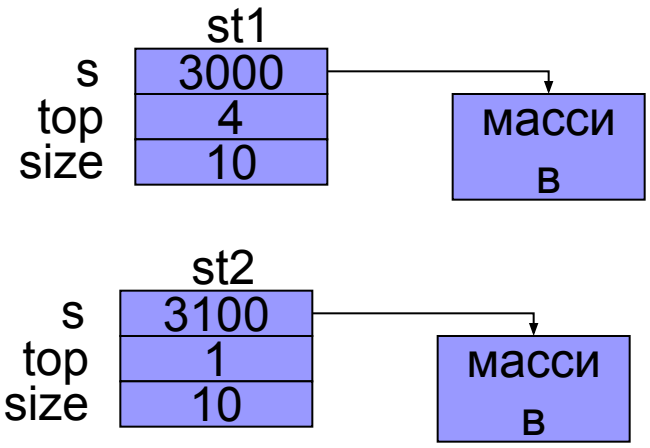
Индивидуальность – это совокупность тех свойств объекта, которые отличают его от всех других объектов.

Структурная неопределённость – это нарушение индивидуальности объектов.

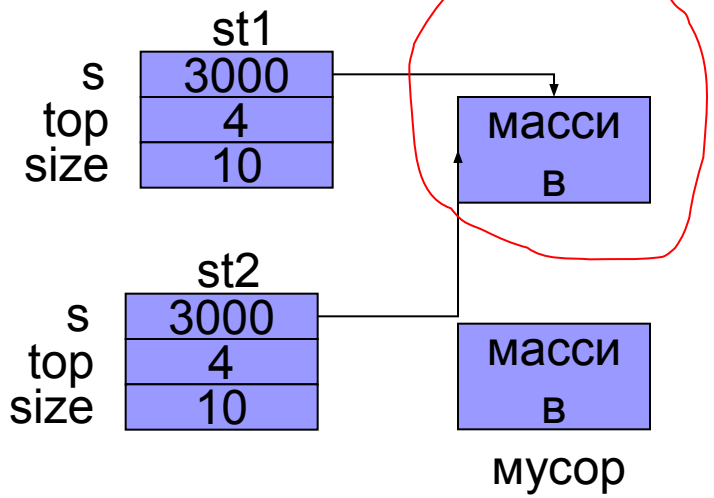
```
void main()
{
  Stack st1(10), st2(10);
  st2=st1;
} //ошибка освобождения памяти при выполнении
  //деструктора второго объекта
```

```
Stack::~Stack()
{
  delete []s;
};
```

структурная неопределенность



после st2=st1



Индивидуальность объекта

Копирующий конструктор – это конструктор, в списке формальных параметров которого стоит ссылка на объект того же класса.

```
class Stack
{
  int size;
  int top;
  int *s;
public:
  Stack(int sz)
  {
    size=sz;
    top=-1;
    s=new int[size];
  };
  // Копирующий конструктор
  Stack(const Stack &st)
  {
    size=st.size;
    top=st.top;
    s=new int[size];
    for(int i=0; i<=top; i++) //копирование массива
       s[i]=st.s[i];
  };
};
```

Индивидуальность объекта

```
//при вызове функции создается таблица адресов и
//значений. В нее помещается копия фактического
//параметра - объекта класса Stack
void f(Stack st)
{
//...
}
```

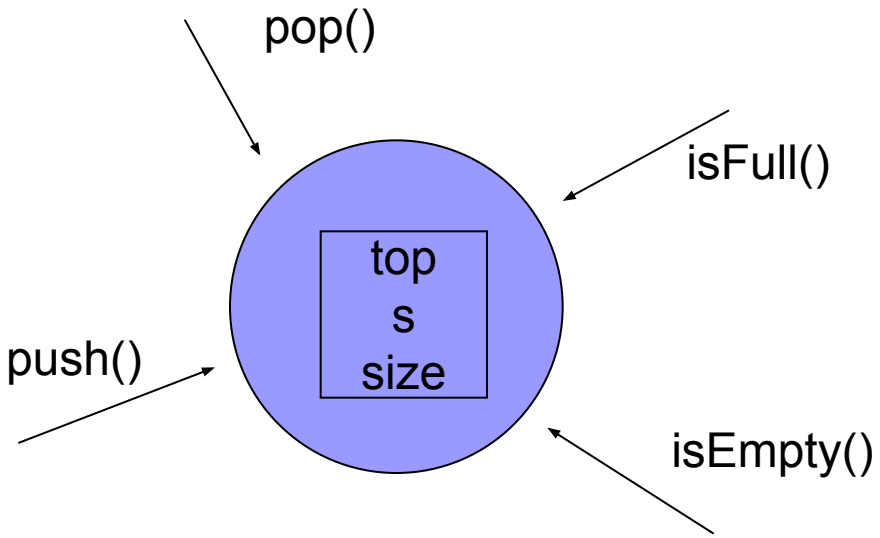
```
int main()
{
  Stack st1(100);
  //...
  Stack st2(st1); //вызов конструктора копирования
  f(st1); // передача параметра по значению
}
```

Инкапсуляция

Инкапсуляция – это объединение данных и поведения в один объект и сокрытие реализации.

Модификаторы доступа:

- **private** – поля и методы доступны только внутри данного класса
- **protected** – поля и методы доступны внутри данного класса и в классах потомках
- **public** – поля и методы доступны любому клиенту класса



Обработка исключительных ситуаций

Исключительная ситуация – ошибка времени выполнения, которая приводит к невозможности (бессмысленности) продолжения выполнения реализуемого алгоритма.

```
// Пример аппаратной исключительной ситуации
// Yes with SEH Exceptions (/EHa)

void main()
{
int* p = 0;
try
{
  char k = p[3];
}
catch(...)
{
  cout << "exception";
}
}
```

Обработка исключительных ситуаций

Далее рассматриваем механизм программных исключительных ситуаций в C++

Оператор throw

```
enum
{
  ENUM_INVALID_INDEX
};
double dX = 3.0;
class MyException{
private:
  char * _message;
public:
  MyException(char* message) : _message(message){}
};

throw -1;
throw ENUM_INVALID_INDEX;
throw "Can not take square root of negative number";
throw dX;
throw MyException("Fatal Error");
```

Обработка исключительных ситуаций

Блок try и блок catch

```
try
{
    throw -1;
}
catch(int k)  // обработчик исключения
{
   cout << k << endl;
}
catch(double d)  // обработчик исключения
{
   cout << d << endl;
}
```

Когда возбуждается исключение оператором throw, выполняется переход к ближайшему обрамляющему блоку try, при этом освобождается стек. Если какой-либо из обработчиков исключения (блок catch) имеет соответствующий тип исключения, то управление передается ему и обработка исключения считается завершенной. В противном случае переход выполняется до следующего обрамляющего блока try. Если ни один обработчик не выполнился, то программа завершается с ошибкой.

Обработка исключительных ситуаций

Функция возбуждает исключение

```
double sqrte (double d)
{
  if(d < 0)
  {
    throw d;
  }
  return sqrt(d);
}

void main()
{
   try
   {
     double s = sqrte(-1.0);
     cout << s << endl;
   }
   catch(double d)
   {
     cout << "Ошибка" << endl;
   }
}
```

Обработка исключительных ситуаций

Выбор обработчика

```
double sqrte (double d)
{
  if(d < 0)
  {
    throw d;
  }
  return sqrt(d);
}

void main()
{
  try
  {
    double s = sqrte(-1.0);
    cout << s << endl;
  }
  catch(int k)
  {
    cout << "Ошибка" << endl;
  }  catch(double d)
  {
    cout << "Ошибка" << endl;
  }
}
```

Обработка исключительных ситуаций

Последовательность выбора обработчика

```
try {
    throw 'a';
}
catch(int k){
  cout << "Ошибка" << endl;
}
catch(double d){
  cout << "Ошибка" << endl;
}
catch(...){   // любые исключения
  cout << "Ошибка" << endl;
}
```

Вложенные блоки try

```
try {
  try {
    // некоторый код
  }
  catch (int n) {
    throw;
  }
}
catch (...) {
cout << "Exception occurred";
}
```

```
return m_nData[nIndex];
}
```

Обработка исключительных ситуаций

Классы исключений

```
class ArrayException
{
private:
    std::string m_strError;

    ArrayException() {}; // not meant to be called
public:
    ArrayException(std::string strError)
        : m_strError(strError)
    {
    }

    std::string GetError() { return m_strError; }
}


 int IntArray::operator[](const int nIndex)
 {
     if (nIndex < 0 || nIndex >= GetLength())
         throw ArrayException("Invalid index");

     return m_nData[nIndex];
 }
```

```
    return m_nData[nIndex];
}
```

Обработка исключительных ситуаций

Наследование классов исключений

```
class Base {
public:
    Base() {}
};
class Derived: public Base {
public:
    Derived() {}
};

int main()
{
    try
    {
        throw Derived();
    }
    catch (Derived &cDerived) // Здесь важен порядок следования обработчиков
    {
        cerr << "caught Derived";
    }
    catch (Base &cBase)
    {
        cerr << "caught Base";
    }

    return 0;
}
```

```
    return m_nData[nIndex];
}
```

Обработка исключительных ситуаций

Стандартные исключения

Стандартная библиотека C++ предлагает базовый класс exception для исключений. Для использования #include <exception>

Исключение	Описание
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typeid
ios_base::failure	thrown by functions in the iostream library

Отношение наполнения

Отношение наполнения – это отношение между классами, при котором классы формируются из общей структуры, называемой параметризованным (обобщённым) классом, путём указания фактических типов элементов параметризованного класса.
Термины "параметризованный класс", "обобщённый класс", "шаблон класса" являются синонимами.

Отношение наполнения

Необходимость:

```
class IntArray{//целочисленный массив
  int *a;
  int size;
public:
  IntArray(int sz); //создание массива
  ~IntArray();//удаление массива
};
class FloatArray{//массив вещественных чисел
  float *a;
  int size;
public:
  FloatArray(int sz);
  ~FloatArray();
};
IntArray::IntArray(int sz){
  size=sz;
  a=new int[size];
}
FloatArray::FloatArray(int sz) {
  size=sz;
  a=new float[size];}
InitArray::~InitArray(){
  delete[] a;
}
FloatArray::~FloatArray(){
  delete[] a;
}
void main(){
  IntArray intArray(10);
  FloatArray floatArray(20);
  ...
}
```

Отношение наполнения

Решение:

```
template <class T>  class Array{
  T* a;
  int size;
public:
  Array(int sz);
  ~Array();
  void initiate();//инициализация элементов
};
template <class T> Array <T>::Array(int sz){
  size=sz;
  a=new T[size];
}
template <class T> Array <T>::~Array()
  {delete[] a;}
template <class T> void Array <T>::initiate(){
  int i;
  for(i=0; i<size; i++)
    a[i]=0;
}
void main(){
  Array<int> intArray(10);//создание варианта
          //целочисленного массива
  Array<float> floatArray(20);//создание варианта
          //массива действительных чисел
  intArray.initiate();
  floatArray.initiate();
}
```

Отношение наполнения

Специализация методов шаблонного класса

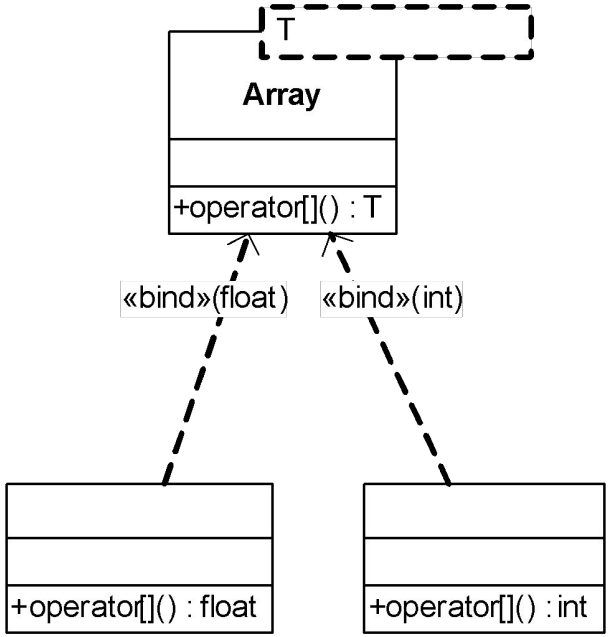
```
void Array<char>::initiate(){
  int i;
  for(i=0; i<size; i++)
     a[i]=' ';
};
void main(){
  Array <float> floatArray(20);
  Array <char> charArray(10);
  floatArray.initiate(); //инициализация нулями
  charArray.initiate();  //инициализация пробелами
};
```

Отношение наполнения

Эквивалентность типов: два конкретных класса, полученных из обобщённого класса, являются эквивалентными,
если совпадают шаблоны и фактические параметры имеют одинаковые значения.

```
template <class T, int SIZE> class Array{
  T *a;
public:
  Array();
  ~Array();
  void initiate();
};
template <class T, int SIZE> Array <T, SIZE>::Array()
  {a=new T[SIZE];};
template <class T, int SIZE> Array <T, SIZE>::~Array()
  {delete a;};
template <class T, int SIZE>
 void Array <T, SIZE>::initiate(){
  int i;
  for(i=0; i<SIZE; i++)
    a[i]=0;
}
void Array <char, 10>::initiate(){
  char blank=' ';
  for(int i=0; i<10; i++)
    a[i]=blank;
};
int main(){
  Array <float, 15> floatArray;
  Array <char, 10> charArray10;
  Array <char, 6> charArray6;
//классы объектов charArray10 и charArray6 различны
  floatArray.initiate();
  charArray10.initiate(); //инициализация пробелами
  charArray6.initiate(); //инициализация нулями
  return 0;
};
```

Зависимость




```
template<class T>
class Array
{
public:
  T operator[](int i) {…}
}

void main()
{
  Array<int> intArray;
  Array<float> floatArray;
// …
}
```

Перегрузка операторов

Ограничения:

- невозможно изменить старшинство операторов;
- запрещено изменять назначение операторов для встроенных типов;
- запрещено конструировать новые операторы.

Автоматически генерируется код для оператора присваивания "=" и оператора вычисления адреса "&"

Функция-оператор может быть или утилитой, или методом класса. При перегрузке операторов "()", "[]", "->", "=" функция-оператор должна быть методом класса

Если функция-оператор является методом класса, то самый левый операнд должен быть объектом (или ссылкой) данного класса. Если же самый левый операнд должен быть переменной встроенного типа, а правый – объектом класса, то функция-оператор может быть только утилитой.

Оператор присваивания не наследуется.

Перегрузка операторов

```
struct Complex
{
  float re,im;
};

Complex operator+(Complex c1,Complex c2)
{
  Complex c;
  c.re=c1.re+c2.re;
  c.im=c1.im+c2.im;
  return c;
};
Complex operator*(Complex c1,Complex c2)
{
  Complex c;
  c.re=c1.re*c2.re-c1.im*c2.im;
  c.im=c1.im*c2.re+c2.im*c1.re;
  return c;
};

int operator<(Complex c1, Complex c2)
{
  return (c1.re*c1.re + c1.im*c1.im) < (c2.re*c2.re +
c2.im*c2.im);
}
```

Перегрузка операторов

```
template<typename T> class Wrapper
{
private:
  T* _a;
  int _size;
public:
  Wrapper(T* a, int size)
  {
    _a = a;
    _size = size;
  }
  T getMin()
  {
    T min = _a[0];
    for(int i = 1; i < _size; i++)
    {
      if (_a[i] < min)
      {
        min = _a[i];
      }
    }
    return min;
  }
};
```

Перегрузка операторов

```
void main()
{
  Complex x1={1,2};
  Complex x2={3,4};
  Complex x3;
  x3=x1+x2;
  x1=x1+x1; //в сложении передается параметр по значению,
            //а не по ссылке
  x3=x1*x2;

  Complex mass[3] = {{1,20}, {-3,2}, {2,1}};
  Wrapper<Complex> wrapper(mass, 3);
  Complex min = wrapper.getMin();
}
```

Перегрузка операторов

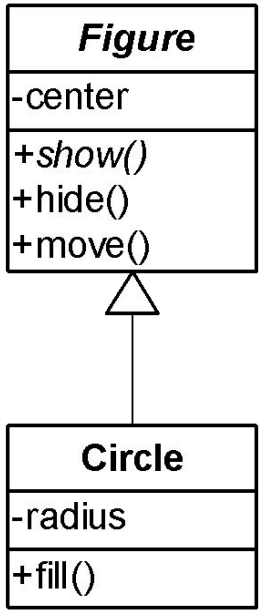
Операторы-методы класса

```
class Complex{
protected:
  float re,im;
public:
  Complex(){re=0.0;im=0.0;}
  Complex(float x,float y)
    {re=x;im=y;}
friend Complex operator+(const Complex& c1, const Complex& c2);
  int operator==(const Complex& c)const
    {return ((re==c.re)&&(im==c.im));}
  Complex operator!()const { //комплексно сопряженное число
    Complex c;
    c.re=re;
    c.im=-im;
    return c;
  }
  float& operator[](int i) { //оператор индексации
    if(i==1)return re;
    else return im;
  }
};
Complex operator+(const Complex& c1, const Complex& c2){
  Complex c;
  c.re=c1.re+c2.re;
  c.im=c1.im+c2.im;
  return c;
}
```

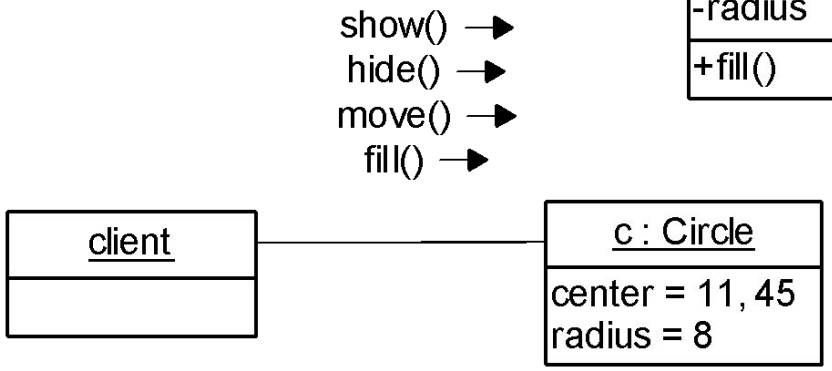
Отношение наследования между классами

Наследование – это такое отношение между классами, при котором один класс повторяет структуру и поведение другого класса (простое наследование) или нескольких других классов (множественное наследование).

```
class Figure
{
  // ...
}
class Circle : public
Figure
{
  // ...
}
```



UML – Unified Modeling Language



Отношение наследования между классами

При наследовании в подклассах можно использовать:

- добавление новых полей
- добавление новых методов
- переопределение методов суперкласса
- замещение методов суперкласса

Добавление полей и методов

```
class Circle
{
private:
  int radius;
public:
  void show();
};
class Eye : public Circle
{
private:
  int color;
public:
  void close();
...
};
```

```
int main(){
  Circle c;
  Eye e;
  e.show();
  c.show();
  e.close();
  c.close(); //ошибка, этот метод имеется
             //только в подклассе
  ...
}
```

Отношение наследования между классами

Доступ в суперклассе	Модификатор доступа при наследовании	Уровень доступа в подклассе
Открытый	public	Открытый
Закрытый	public	Недоступный
Защищённый	public	Защищённый
Открытый	private	Закрытый
Закрытый	private	Недоступный
Защищённый	private	Закрытый
Открытый	protected	Защищённый
Закрытый	protected	Недоступный
Защищённый	protected	Защищённый

Механизм доступа к полям и методам обеспечивается тем, что объявление метода содержит скрытый (необъявленный) параметр – указатель на объект, для которого выполняется метод

```
// Неформальная запись!!!
void show(Circle* this const)
{
   ...
   int r = this->radius;
   ...
}
```

Переопределение методов

Переопределение – объявление в подклассе метода с таким же именем и списком параметров, как и в суперклассе

Раннее связывание объектов и методов

```
class Circle
{
public:
  void show();
   ...
};
```

```
class Eye: public Circle
{
public:
  void show();
  ...
};
```

```
int main()
{
  Circle *pC; //указатель на объект суперкласса
  pC=new Circle; //фактический объект - класса Circle
  pC->show(); //вызов метода Circle::show()
  delete pC;
  pC=new Eye; //фактический объект - класса Eye
  pC->show(); //вызов метода Circle::show()
  delete pC;
  return 0;
}
```

Замещение методов

Позднее связывание объектов и методов. Полиморфизм

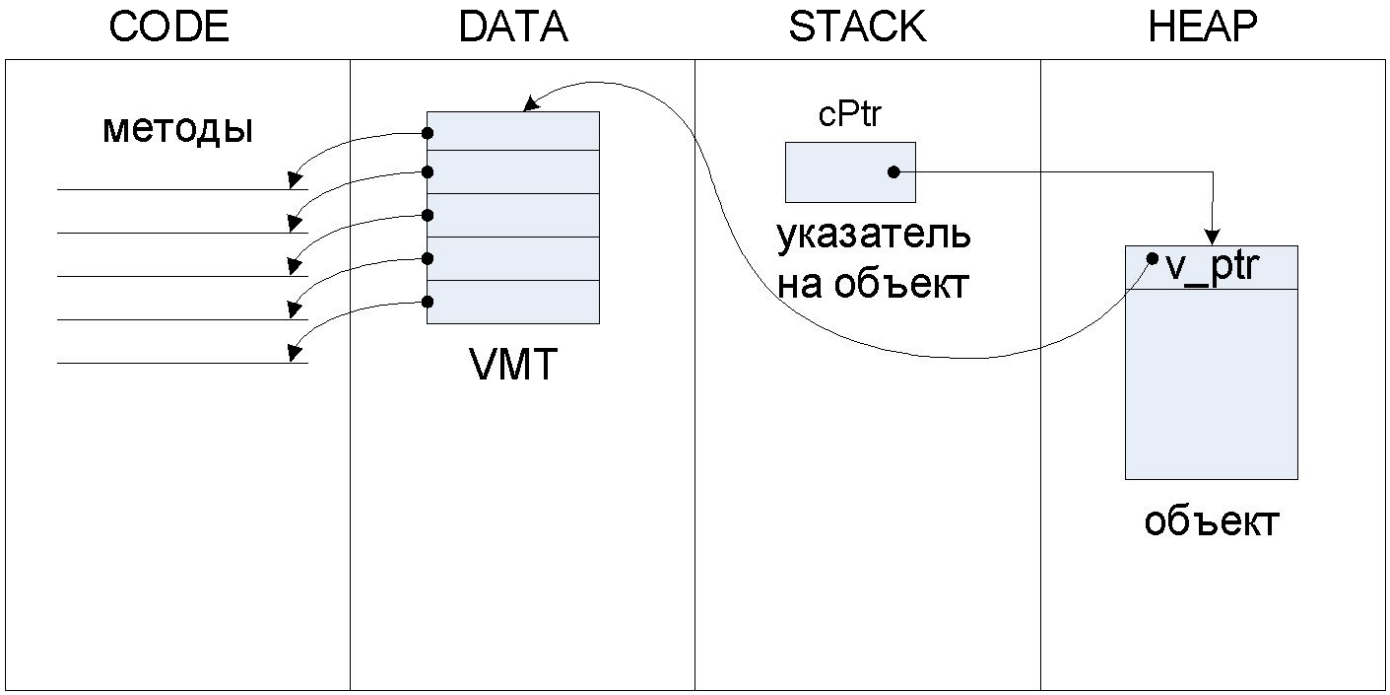
```
class Figure
{
  int x,y;
public:
  virtual void show()=0;
  virtual void hide();
  virtual void move(int newX, int
newY){
    hide();
    x=newX; y=newY;
    show();}
};
```

```
class Circle:public Figure
{
  int radius;
public:
  virtual void show();
  virtual void hide();
};
class Face: public Circle
{
  int eyeColor;
public:
  void show();
  void hide();
  virtual void
closeEyes();
};
```

```
int main()
{
  Circle *cPtr;
  cPtr=new Face; //объект класса Face
  cPtr->show(); //вызывается Face::show()
  cPtr->move(10,20);
   //Вызывается Figure::move()
   //в нем вызываются Face::show() и
   //Face::hide() в соответствии с фактическим
   //классом объекта *cPtr.
  delete cPtr;
  //...
}
```

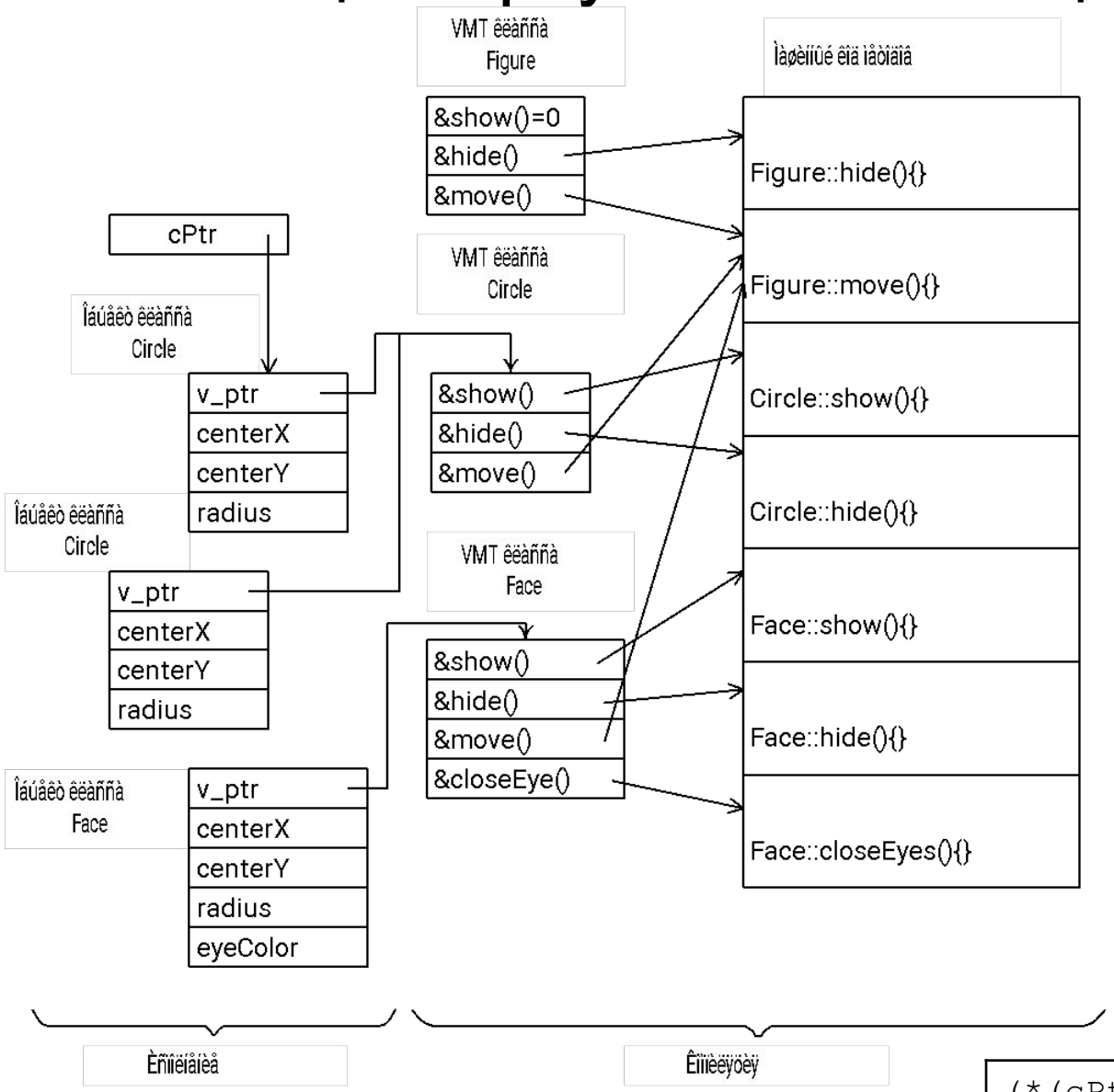
Замещение методов

Позднее связывание объектов и методов



VMT = Virtual Method Table = таблица виртуальных методов
(точнее – таблица указателей на виртуальные методы)

Таблица виртуальных методов



Конструктор заносит в поле v_ptr адрес VMT

Вызов виртуального метода

```
cPtr->move(10,20);
```

преобразуется в косвенный вызов

```
(*(cPtr->v_ptr[2]))(cPtr, 10, 20);
```

Уточнение методов

```
class Figure
{
public:
  virtual void show(){...}
};
class Circle:public Figure
{
public:
  virtual void show()
  {
    //вызов метода суперкласса
    Figure::show();
    ... //рисование окружности
  }
};
```

```
class Face:public Circle
{
public:
  virtual void show()
  {
    //вызов метода суперкласса
    Circle::show();
    ... //рисование других элементов
  }
}
```

```
void main()
{
  Face F;
  // будут вызваны три метода:
  //Figure::show(); Circle::show(); Face::show();
  F.show();
}
```

Конструкторы и деструкторы

```
class A {...}
class B:public A{...}
class C:public B{...}
main()
{
  cPtr=new C; //здесь вызываются конструкторы всех предков
            //A::A(); B::B(); C::C()
  delete cPtr; // деструкторы вызываются в обратном порядке
}
```

Проблема при использовании невиртуального деструктора

```
class B{
public:
  B();
  ~B();
};
class D:public B{
public:
  D();
  ~D();
}
void f(){
B *p=new D();//создается объект класса D
delete p; //уничтожается объект класса B
}
```

Надо так:

```
class B
{
 B();
 virtual ~B();
};
```

Перегрузка Конструкторов!!!

Отношение наполнения

Использование обобщённого класса в обычном (конкретном) классе

```
class AgrStack{
  int top;
  Array<int> arr;
public:
  AgrStack(int n):arr(n) {top=-1;}
  ~AgrStack(){;}
void init() {arr.initiate();}
};
main{
  AgrStack st(4);
  st.init();
  return 0;
}
```

Отношение наполнения

Обобщенный класс является подклассом конкретного суперкласса

```
class Base{
float x;
public:
  Base(float newX)
    {x=newX;}
};
template <class Type> class Derived:public Base{
  Type y;
public:
  Derived(float newX,Type &newY):Base(newX),y(newY){;}
};
int main(){
  char c='z';
  Derived<char> d(3.0,c);
  return 0;
}
```

Конкретный класс является подклассом обобщенного класса

```
class IntStack:public Array<int>{
  int top;
public:
  IntStack(int n): Array<int>(n){top=-1;}
  int isEmpty(){return top==-1;}
  void initiate(){
    Array<int>::initiate();
    top=-1;
  }
};
void main(){
  IntStack st(10);
  st.initiate();
}
```

Отношение наполнения

Обобщенный класс является подклассом обобщенного класса

```
template <class Type> class Stack : public Array
<Type>{
  int top;
public:
  Stack(int n): Array<Type>(n),top(-1) {;}
  int isEmpty(){return top==-1;}
  void initiate(){
    Array<Type>::initiate();
    top=-1;
  }
};
void main(){
  Stack<char> st(15);
  st.initiate();
}
```

Перегрузка операторов

```
//наследование
class Point:public Complex{
  char name;
public:
  Point(){name='U';}
  Point(float x,float y,char n):Complex(x,y), name(n) {}
};
```

```
void main(){
  Complex y1(1,2);
  Complex y2(3,4);
  Complex y3;
  y1=y1+y1;
  if(y1==y2)
    y3=y1+y2;
  y1=!y1;
  y1[1]=0.0;
  y1[2]=y1[1];

  Point z1(1,2,'D');
  Point z2(1,2,'A');
  if(z1==z2) //функция-оператор == наследуется
    ;//z1=z1+z2; //ошибка типизации (= не наследуется)
  else
    z1[1]=2*z1[2]; //функция-оператор [] наследуется
}
```

Множественное наследование

Множественное наследование: более одного родительского класса

Проблемы:

- Неопределенность наименования элементов класса
- Повторное наследование структуры и поведения

Неопределенность наименования

В разных суперклассах одного подкласса используются одинаковые имена элементов (полей или методов)

```
class Amplifier
{
protected:
  float maxPower; //максимальная мощность на нагрузке
  ...
};
class Chip
{
protected:
   float maxPower; //максимальная потребляемая мощность
   ...
};
class DCAmplifier: public Amplifier, public Chip
{
public:
  void f(float x);
  ...
};
void DCAmplifier:: f(float x)
{
maxPower=x; //неопределённость: какое из полей
            //суперклассов должно быть изменено?
...
}
```

Неопределенность наименования

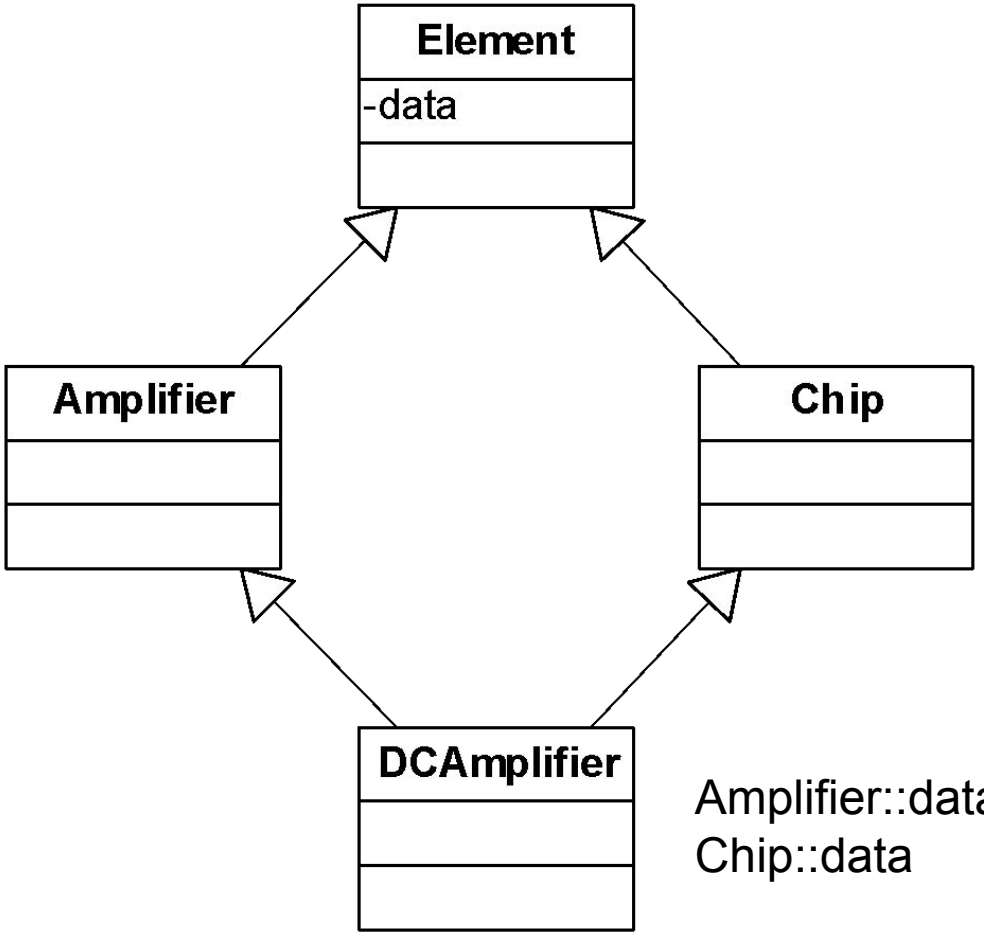
```
void printA(){
cout<<"это внешняя printA"<<endl;
};
class Base1{
public:
int a;
Base1(int m) {a=m;};
void printA(){cout<<"это Base1::a "<<a<<endl;}
};
class Base2{
public:
float a;
Base2(float r) {a=r};
void printA(){cout<<"это Base2::a "<<a<<endl;}
void print2A(){
cout<<"это 2*Base2::a "<<2*a<<endl;}
};
```

```
class Derived: public Base1, public Base2
{
public:
char a;
Derived (char l, int m, float r):
  Base1(m), Base2(r){a=l;};
void printA(){cout<<"это Derived::a "<<a<<endl;}
void printAll(){
    printA();
    Base1::printA();
    Base2::printA();
    ::printA();}
};
```

```
main(){
Base1 b1(10), *base1Ptr;
Base2 b2(2.3), *base2Ptr;
Derived d('z', 5, 7.1);

b1.printA();
b2.printA();

//вызывается Derived::PrintA()
d.printA();

//вызывается Base2::print2A(),
//так как он не переопределен
d.print2A();
d.Base1::printA();
d.Base2::printA();
base1Ptr=&d;

//вызывается Base1::printA(),
//так как методы невиртуальные
base1Ptr->printA();
base2Ptr=&d;

//вызывается Base2::printA()
  base2Ptr->printA();}
```

Повторное наследование

```
class Element {...};
class Amplifier : public Element {...};
class Chip : public Element {...};
class DCAmplifier : public Amplifier, public Chip {...};
```



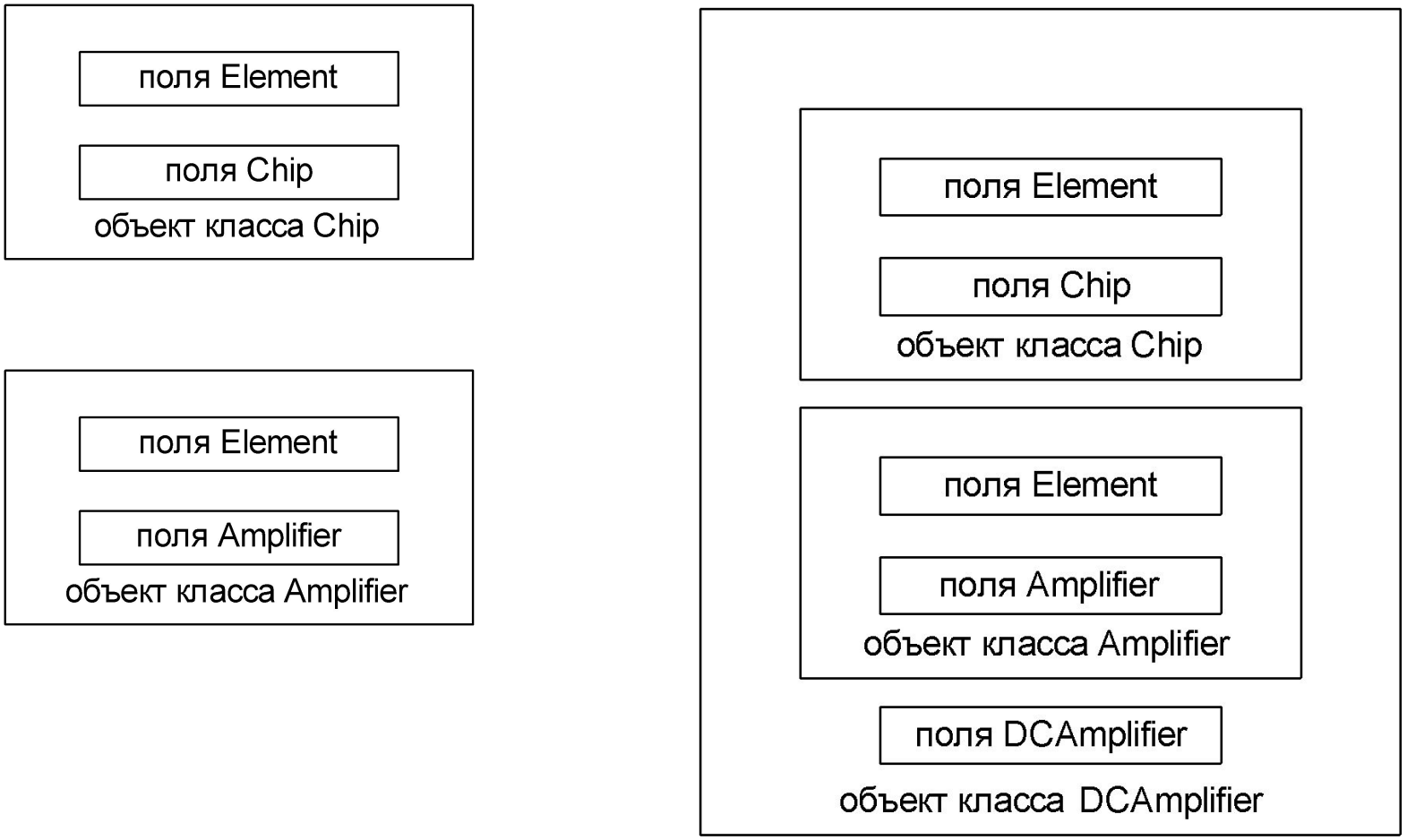
Element::data

Element::data

Amplifier::data
Chip::data

Повторное наследование

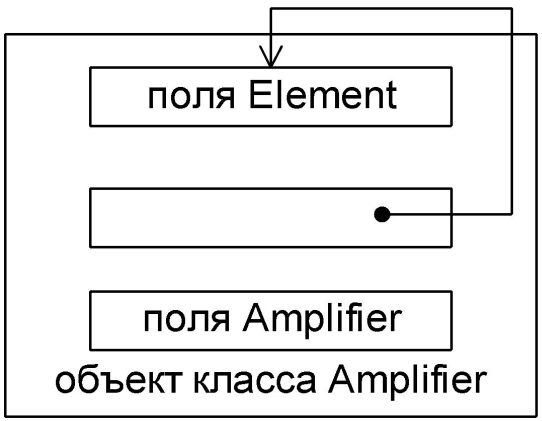
Структура объектов при повторном наследовании

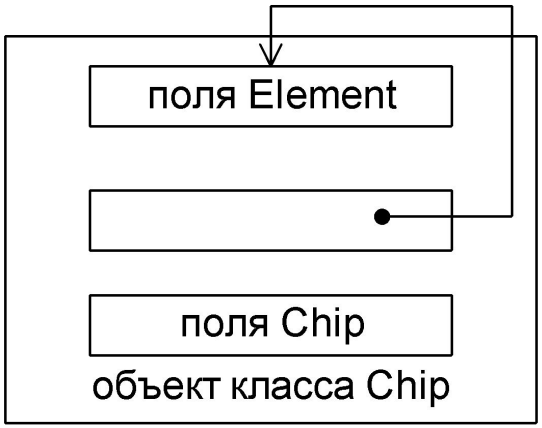


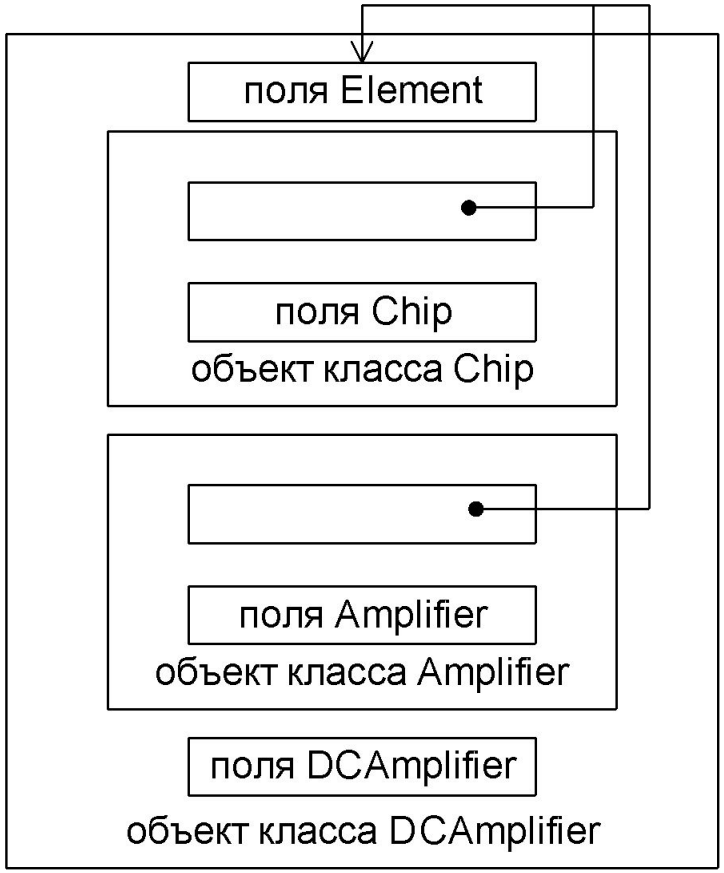
Повторное наследование

Виртуальные базовые классы

```
class Element {...};
class Amplifier : public virtual Element {...};
class Chip : public virtual Element {...};
class DCAmplifier : public Amplifier, public Chip {...};
```







Повторное наследование

Повторное наследование без виртуального наследования. Последовательность выполнения конструкторов

```
#include<iostream>
using namespace std;
class A {
protected:
  int i;
public:
  A(){i = 1; cout << "A()\n";}
  A(int k){i = 2; cout << "A(int)\n";}
  A(double d){i = 3; cout << "A(float)\n";}
};
class B: public A {
public:
  B(): A(7){cout << "B()\n";} //конструктор для A с целым параметром
};
class C: public A {
public:
  C():A(3.141){cout << "C()\n";} //конструктор для A
  //с действительным параметром
};
class D: public B, public C {
public:
  D(): B(), C() // явно указываются только конструкторы суперкласса
  {cout << "D()\n";
  cout << B::i << endl;
  cout << C::i << endl;}
};
void main(){
  D d;
}
```

Повторное наследование

Последовательность выполнения конструкторов при виртуальном наследовании

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

class A {
public:
   A(){cout << "A()\n";}
   A(int k){cout << "A(int)\n";}
   A(double d){cout << "A(double)\n";}
};
class B: virtual public A {
public:
    B(): A(7){cout << "B()\n";} //конструктор для A с целым параметром
};
class C: virtual public A {
public:
  C():A(3.141){cout << "C()\n";} //конструктор для A с действительным параметром

};
class D: public B, public C {
public:
   D(): A(3), B(), C(){cout << "D()\n";}
};
void main(){
  D d;  //вызов конструктора класса A в конструкторах
        // B и C игнорируется, вызывается A(3),
        //потом выполняется B::B(), потом C::C()
}
```

Ассоциация между классами

Отношение ассоциации определяет способы взаимодействия объектов ассоциированных классов

```
class Sensor{
public:
  int getSignal();
};
class Valve{
public:
  void switchOn(int Delta);
  void switchOff(int Delta);
};
class Controller{
public:
  void run(Sensor &s, Valve &v) {
  if(s.getSignal()>10)
     v.switchOn(5);
  else
     v.switchOff(5);
};
void main(){
  Sensor sensor;
  Valve valve;
  Controller controller;
  ...
  controller.run(sensor, valve);
```

Ассоциация (агрегация)

Агрегация по значению

Агрегация по ссылке

```
class Mouth{
  float capacity;
public:
  Mouth(float c);
  ~Mouth();
};
class Eye{
  float radius;
  int color;
public:
  Eye(float r, int c);
  ~Eye();
};
Mouth::Mouth(float c): capacity(c) {;}
Mouth::~Mouth(){;}
Eye::Eye(float r, int c): radius(r), color(c) {;}
Eye::~Eye(){;}
class Face{
  Eye leftEye;
  Eye rightEye;
  Mouth mouth;
public:
  Face(float eRadius, int eColor, float
mCapacity);
  ~Face();
};
Face::Face(float eRadius, int eColor,
float mCapacity): leftEye(eRadius, eColor),
rightEye(eRadius, eColor), mouth(mCapacity) {
  ...
  leftEye=rightEye;
  ...
}
Face::~Face(){;}
```

```
int main(){
  Face face(5, 1, 10); //здесь
вызываются конструктор агрегирующего
объекта и конструкторы агрегируемых
объектов
} //здесь автоматически вызываются
деструкторы агрегируемых объектов и
деструктор агрегирующего объекта
```

Ассоциация

Ситуации, в которых возникает отношение ассоциации классов:

1) методу данного класса передается объект используемого класса, или указатель на такой объект в качестве параметра;

2) поле данного класса является объектом используемого класса или указателем на такой объект (агрегация);

3) в методе данного класса создаются локальные объекты используемого класса;

4) метод данного класса выполняет действия над глобальным объектом используемого класса.

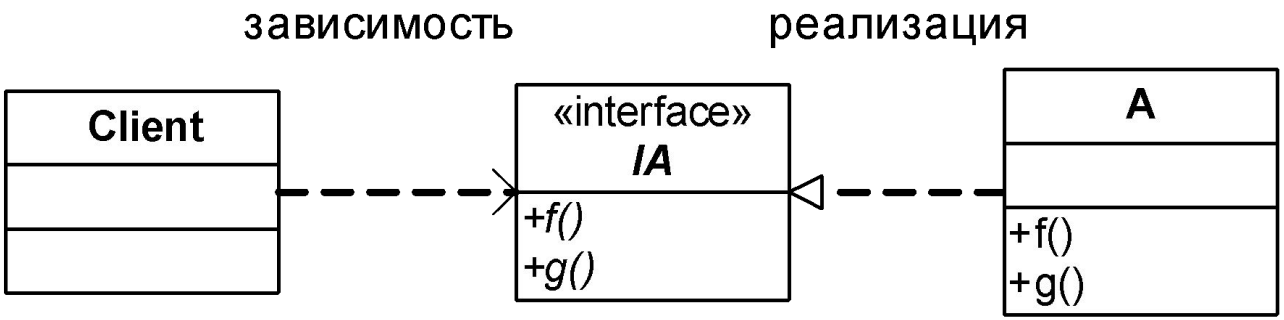
Ассоциация

Пример

```
#include <iostream.h>
class Chip{
public:
  void initiate();//привести в начальное состояние
  ...
};
class Signal{
public:
  float getValue(){...};//получить значение сигнала
};
class Beep{
public:
  Beep(int seconds){...}
};
class Device{
private:
  Chip chip; //объект - поле класса
public
  void output(Signal &s){ //вывод значения сигнала
    cout<<s.getValue(); //доступ к глобальной переменной
  }
  void reset(){
    Beep *pbeep; //создание локального указателя на объект
    pbeep=new Beep(10);
    delete pbeep;
    chip.initiate();
  }
};
```

```
int main(){
  Device divice;
  Signal signal;
  ...
  device.output(signal);
  ...
}
```

Зависимость



Дружественное отношение

```
class Device; //предварительное объявление класса
class Chip {
friend class Device; //объявление дружественного класса
private:
  void f(); //закрытый метод
  ...
};
class Device{
...
Chip chip;
public:
   void g();
   ...
};
...
void Device::g(){
  chip.f(); // закрытая функция здесь доступна
}
```

Потоковый ввод/вывод

1) *iostream* – основная библиотека со стандартными потоками ввода/вывода.

2) *iomanip* – библиотека с манипуляторами для форматного ввода/вывода.

3) *fstream* – библиотека файловой системы ввода/вывода.

4) *strstream* – библиотека внутреннего форматного ввода/вывода для выполнения операций над строками символов.

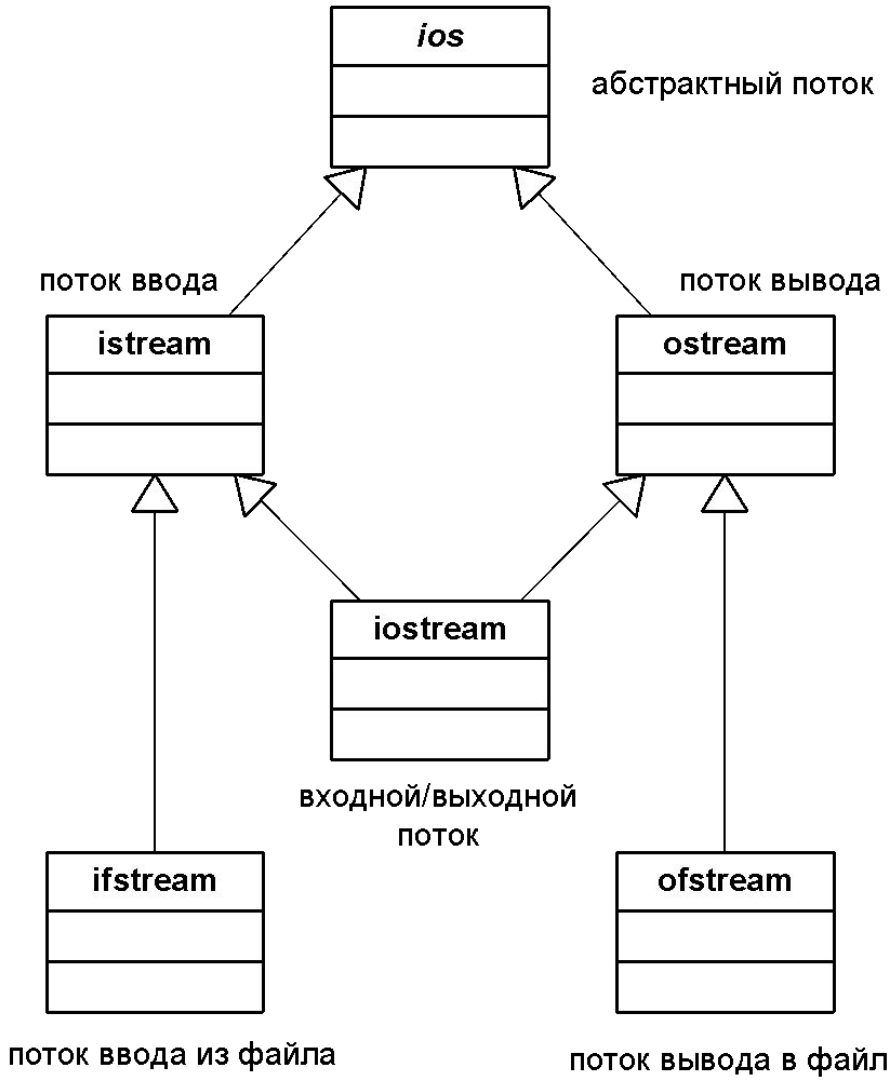
cin – поток, связанный со стандартным входным устройством;

cout – поток, связанный со стандартным выходным устройством;

cerr – поток, связанный со стандартным устройством вывода сообщений об ошибке;

clog – поток, связанный со стандартным устройством системного журнала.

```
istream cin;
ostream cout;
ostream cerr;
ostream clog;
```



Потоковый ввод/вывод

Потоковый вывод

```
ostream& ostream::operator<<(const int& val){...}
ostream& ostream::operator<<(int val){...}
ostream& ostream::operator<<(char* val){...}
```

```
cout<<121<<0<<13; //равносильно последовательному
                  //выполнению cout<<121; cout<<0; cout<<13;
```

```
char *string="test"; //строка оканчивающаяся нулём
cout<<string; //выводятся символы "test",
              //начиная с указателя string
```

```
double f=sqrt(2.0);
cout<<f<<endl;                 //результат 1.414214
cout.precision(3);             //метод для потока
cout<<f<<endl;                 //результат 1.414
cout<<setprecision(2)<<f<<endl; //результат 1.41
```

Потоковый ввод/вывод

Потоковый ввод

```
istream& istream::operator>>(int&);
istream& istream::operator>>(char&);
```

```
cin>>x>>y; //равносильно последовательному выполнению
          //операторов cin>>x; cin>>y;
```

```
while(cin>>x)  //цикл ввода данных
  {if(x>y)...}
```

```
while((c=cin.get())!=EOF){...} //здесь EOF - символ
                               //конца файла
```

```
char buffer[1024];
cin>>buffer; //чтение
cout<<buffer; //вывод
```

Потоковый ввод/вывод

Перегрузка операторов ввода/вывода (только утилиты)

```
class Complex{
friend ostream &operator<<(ostream&, const Complex&);
friend istream &operator>>(istream&, Complex&);
private:
  float re;
  float im;
};
ostream &operator<<(ostream &output, const Complex &comp){
  output<<"re="<<comp.re<<" im="<<comp.im;
  return output;
}
istream &operator>>(istream &input, Complex &comp){
  input>>comp.re;
  input>>comp.im;
  return input;
}
void main(){
  Complex A, B;
  cin>>A>>B;
  cout<<A<<B;
  ...
}
```

Потоковый ввод/вывод

Файлы последовательного доступа: вывод

```
#include "stdafx.h"
#include<iostream>
#include<fstream>

using namespace std;

void main()
{
  ofstream outputFile("x.dat", ios::out);
  if(!outputFile)
  {
    cerr<<"ошибка"<<endl;
  }
  else
  {
    int index;
    float fNum;
    while(cin>>index>>fNum)
    {
      outputFile<<index<<' '<<fNum<<endl;
    }
  }
}
```

Некоторые режимы открытия файлов:
ios::out – файл для вывода;
ios::in – файл для ввода;
ios::app – добавлять данные только в конец файла;
ios::noreplace – если файл существует, то генерировать ошибку файловой операции;
ios::nocreate – если файл не существует, то не создавать новый;
ios::trunc – очистить файл.

```
class ios
{
  public:
  enum
  {
    out,
    in,
    app,
// . . .
  };
};
```

Потоковый ввод/вывод

Файлы последовательного доступа: ввод

```
#include "stdafx.h"
#include<iostream>
#include<fstream>

using namespace std;

void main()
{
  int index;
  float fNum;
  ifstream inputFile("x.dat", ios::in);
  while(inputFile >> index >> fNum)
  {
      cout<<index<<' '<<fNum<<endl;
  }
}
```

Константные методы, константные поля и объекты-константы

Константные методы

```
class Time
{
private:
   int hour;
public:
   void setTime(int hr);
   int getHour() const;
   Time(int hr);
};
void Time::setTime(int hr)
{
  hour=hr;
}
int Time::getHour() const
{
  return hour;
}
Time::Time(int hr)
{
  hour=hr;
}
void main()
{
  const Time noon(24);
  //noon.setTime(0); //ошибка
  cout<<noon.getHour(); //правильно
}
```

Константными объявляются методы, которые не должны изменять поля объекта и не вызывают не константные методы

```
class Time
{
  int getHour() const;
  int getHour();
}
 Внутреннее представление:
int getHour(const Time * const this)
int getHour(Time * const this)
```

Константные методы, константные поля и объекты-константы

Константные поля

```
class Inc{
private:
  int value;
public:
  // Костантное поле
  const int increment;
  Inc(int v, int i);
  void addIncrement();
};

Inc::Inc(int v, int i):increment(i) {
  value=v;
}
void Inc::addIncrement(){
  value=value+increment;
//  increment = 2; //ошибка
}

void main(){
  Inc counter(10, 2);
  counter.addIncrement();
  int i = counter.increment;
  //counter.increment = 2; // ошибка
}
```

Если поле объекта не должно изменяться на всём протяжении существования объекта, то оно может быть объявлено константным.

Статические методы, статические поля

```
class Student
{
private:
  // поле объекта
  int id;
  // статическое поле
  static int counter;
public:
  Student()
  {
    counter=counter+1;
    id=counter;
  }
  //статический метод
  static int howMany()
  {
    //используется статическое поле
    return counter;
  }

  void writeInfo()
  {
    cout<<"id: "<<id;
    cout<<" total: "<<counter<<endl;
  }
};
// инициализация статического поля
int Student::counter=0;
```

```
void main()
{
  cout<<Student::howMany()<<endl;
  Student st1, st2, st3;
  cout<<Student::howMany()<<endl;
  cout<<st2.howMany()<<endl;
  st2.writeInfo();
}
```

Особенности использования конструктора копирования

Конструктор копирования по умолчанию вызывает конструктор копирования родительского класса. Если для дочернего класса явно задан конструктор копирования, то его реализация должна явным образом вызывать конструктор копирования родительского класса.

Конструктор копирования по умолчанию агрегирующего объекта вызывает конструкторы копирования агрегируемых объектов. Если в агрегирующем классе явно задан конструктор копирования, то он должен содержать инициализаторы для копирования полей.

```
// Базовый класс с конструктором копирования
class Base
{
public:
  char name;
  int id;
  // Конструктор без параметров
  Base(){name='U', id=0;}
  // Конструктор копирования
  Base(const Base& sb) : name(sb.name),
id(sb.id) {;}
};
```

```
// Производный класс с конструктором
копирования
class Array:public Base{
public:
  int size;
  float *ptr;
  Array(int sz){
    size=sz;
    ptr=new float[size];
  }
  // Конструктор копирования
  Array(const Array& sa);
};

// Если явно не вызывать конструктор
копирования предка,
// то копирование будет неправильным, так как
будет вызван конструктор
// по умолчанию (без параметров)
Array::Array(const Array& sa):Base(sa){
  size=sa.size;
  ptr=new float[size];
  for(int i=0; i<size; i++){
    ptr[i]=sa.ptr[i];
  }
}
```

Особенности использования конструктора копирования

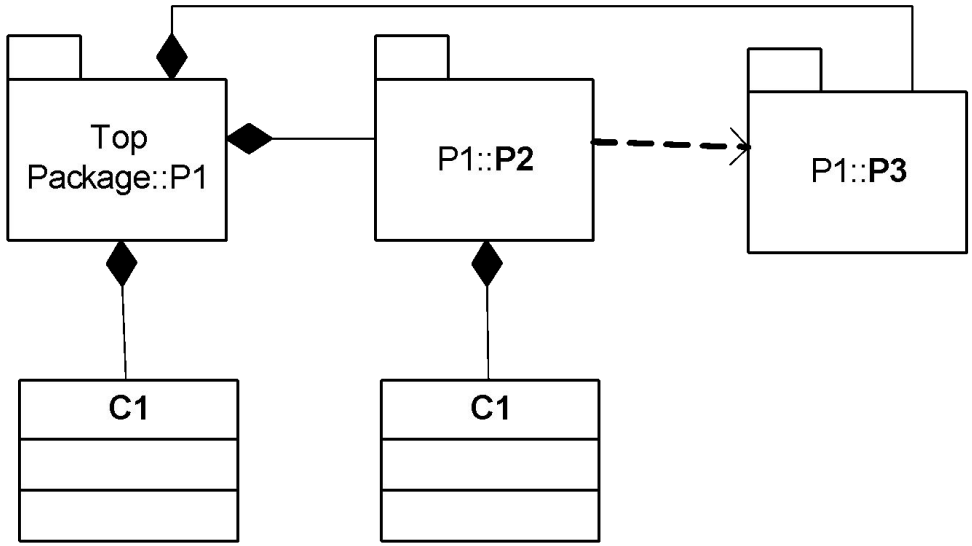
```
// Агрегирующий класс без конструктора
копирования
class Aggregate{
public:
  Array fArr;
  Array sArr;
  Aggregate():fArr(3), sArr(4){;}
  // нет явно заданного конструктора
копирования
  // в агрегирующем классе
};

// Агрегирующий класс с конструктором
копирования
class AggregateWithCopy{
public:
  Array fArr;
  Array sArr;
  AggregateWithCopy():fArr(3), sArr(4){;}
  // Конструктор копирования агрегирующего
класса
  AggregateWithCopy(const AggregateWithCopy &
nta):
    fArr(nta.fArr), sArr(nta.sArr){;}
};
```

```
// Функция-утилита.
// Параметр, передаваемый по значению - объект
void func(Array param)
{
  for(int i=0; i<param.size;i++)
    param.ptr[i]=0;
}
```

```
void main(){
  Base b1;
  b1.name = 'Z';
  Base b2(b1);

  Array x1(3);
  x1.name = 'Z';
  Array x2(x1);

  Aggregate t1;
  t1.fArr.name = 'Z';
  // Другая форма вызова конструктора
копирования
  Aggregate t2=t1;

  AggregateWithCopy n1;
  n1.fArr.name = 'Z';
  AggregateWithCopy n2=n1;

  func(x2);
}
```

Пространство имен

Пакет – механизм общего назначения для распределения программных элементов по группам с установлением владельца, а также средства для предотвращения конфликтов имен




```
namespace P1
{
  class C1 {}

  namespace P2
  {
    class C1 {}
  }
}

int main()
{
  P1::C1   x1;
  P1::P2::C1 x2;
}
```