Языки программирования
Лекция 4

ПМИ Семестр 2

Демяненко Я.М.

2024

Семантика перемещения

Семантика перемещения появилась в стандарте C++11.

Она нацелена на **уменьшение количества создаваемых копий** объектов при выполнении конструктора копии и операции присваивания, которые вызываются для **rvalue** выражений.

lvalue- и **rvalue** выражения

Любое выражение в C++ является или левосторонним (lvalue), или правосторонним (rvalue).

Выражение **lvalue** — это объект, который имеет имя. Все переменные являются lvalue.

А выражение **rvalue** — это временный безымянный объект, не существующий за пределами того выражения, которое его создало.

Пример. Реализовать move-семантику на примере упрощённого класса для динамического массива.

Для этого достаточно:

- определить конструктор с параметрами по умолчанию;
- определить конструктор копии;
- определить деструктор;
- перегрузить операцию сложения двух массивов одинакового размера;
- перегрузить операцию присваивания.

```
class myvector {
private:
  static int f;
  string name;
  int size;
  int * vect;
public:
  myvector(int s = 1, string nm ="noname"): size(s), name(nm){
    f++;
    cout << "constructor > " << name<<" "<<f<< endl;
    vect = new int[s];
  }
```

```
myvector(const myvector & v): size(v.size){
    f++;
    name = "Copy ( "+v.name+" )";
    cout << "copy > " << name << " " <<  f << endl;
    vect = new int[v.size];
    for (int i = 0; i<v.size; ++i)
      vect[i] = v.vect[i];
  }


~myvector(){
    f--;
    cout << "destructor > " << name << " " << f << endl;
    delete[] vect;
  }
```

```
myvector& operator= (const myvector &v){
    cout << name <<" operator= > " << v.name <<" "<< f << endl;
    if (&v != this) {
      delete[] vect;
      vect = new int[v.size];
      for (int i = 0; i<v.size; ++i)
        vect[i] = v.vect[i];
      size = v.size;
    }
    return *this;
  }
```

```
 myvector operator+(const myvector& v){
    if (size == v.size) {
      myvector v1(size, name + " + "+v.name);
      for (int i = 0; i < size; ++i)
        v1.vect[i] = vect[i] + v.vect[i];
      return v1;
    }
    return *this;//лучше использовать исключение
  }
};
```

```
#include "vect.h"
int myvector::f = 0;
int main() {

  myvector a(3,"A"), b(3,"B"),D(a);
  myvector cc (a + b);
  a = b;
  b = a + D;
  return 0;
}
```

Командная строка

```
D:\work\C++\moveconstructor\Debug>matr_vect
constructor > A 1
constructor > B 2
copy > Copy ( A ) 3
constructor > A + B 4
copy > Copy ( A + B ) 5
destructor > A + B 4
A operator= > B 4
constructor > A + Copy ( A ) 5
copy > Copy ( A + Copy ( A ) ) 6
destructor > A + Copy ( A ) 5
B operator= > Copy ( A + Copy ( A ) ) 5
destructor > Copy ( A + Copy ( A ) ) 4
destructor > Copy ( A + B ) 3
destructor > Copy ( A ) 2
destructor > B 1
destructor > A 0
```

Именно для них будут создаваться дополнительные временные объекты, которые после использования в конструкторе копии и операции присваивания тут же будут удалены.

Выражениями **rvalue** являются

a + b внутри вызова конструктора копии для объекта cc

и **a + D** в операторе присваивания.

Решение проблемы

Во многих современных компиляторах встроен механизм Return Value Optimization (RVO), решающий, в том числе, и эту проблему.

Однако автоматическая оптимизация не всегда бывает эффективной, поэтому в стандарте C++11 Бьярн Страуструп предложил вынести решение на уровень языка.

Для этого были введены **move-конструктор** и **move-operator=**

Идея move-семантики

Не удалять временный объект и **не выделять** память для полей в новом объекте, а **инициализировать** поля в создаваемом объекте **ссылками** на поля временного объекта.

Конструктор копирования выделяет новую область памяти для хранения данных, вызывая оператор **new**, а **перемещающий конструктор** — **забирает** данные **у** переданного ему **временного объекта**.

При использовании **move-семантики деструктор не** должен **освобождать** память временного объекта, поскольку ссылкой на неё инициализируется поле в другом объекте.

Для этого в **move-конструкторе** и **move-операции присваивания** поле указатель временного объекта меняет значение на **nullptr**,
а **в деструкторе** выделенная **память освобождается** только, **если** поле указатель не равен nullptr

move-конструктор

```
//move-конструктор
myvector(myvector&& v){
  name = "Move_Copy ( " + v.name+" )";
  cout << "move > " << name << " " << f << endl;
  size = v.size;
  vect = v.vect;
  // Не позволит сразу удалить временный объект
  v.vect = nullptr;
}
```

move-операция присваивания

```
//move-операция присваивания
myvector& operator=(myvector&& v){
  cout << name <<" operator-move= > "<< v.name <<" "<< f <<endl;
  if (vect != nullptr)
    delete[] vect;
  size = v.size;
  vect = v.vect;
  v.vect = nullptr;
  return *this;
}
```

деструктор

```
//изменённый деструктор
~myvector(){
  if (vect != nullptr) {
    f--;
    cout << "destructor > " << name << " " << f << endl;
    delete[] vect;
  }
}
```

myvector && v

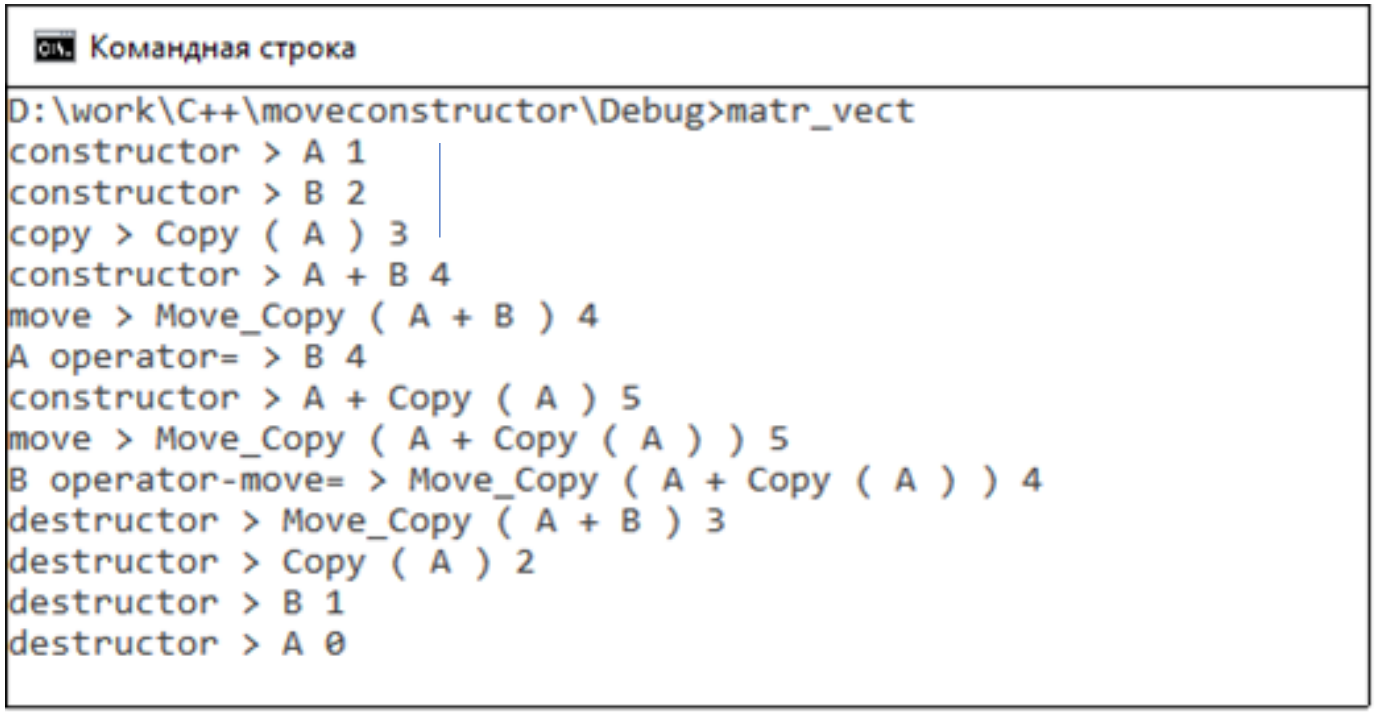
Чтобы отличать функции с перемещающей семантикой в стандарте C++11 введены **rvalue ссылки — myvector && v**.

При этом компилятор будет **использовать** функции с **перемещающей семантикой** только в случае, если параметром является **rvalue выражение** (временный объект).

Теперь, при выполнении той же самой программы, для строки `myvector cc(a+b)`; компилятор выберет move-конструктор вместо конструктора копии.

А для строки `b = a + D`; компилятор выберет move-операцию присваивания.

```
#include "vect.h"
int myvector::f = 0;
int main() {

  myvector a(3,"A"), b(3,"B"),D(a);
  myvector cc (a + b);
  a = b;
  b = a + D;
  return 0;
}
```



Командная строка

```
D:\work\C++\moveconstructor\Debug>matr_vect
constructor > A 1
constructor > B 2
copy > Copy ( A ) 3
constructor > A + B 4
copy > Copy ( A + B ) 5
destructor > A + B 4
A operator= > B 4
constructor > A + Copy ( A ) 5
copy > Copy ( A + Copy ( A ) ) 6
destructor > A + Copy ( A ) 5
B operator= > Copy ( A + Copy ( A ) ) 5
destructor > Copy ( A + Copy ( A ) ) 4
destructor > Copy ( A + B ) 3
destructor > Copy ( A ) 2
destructor > B 1
destructor > A 0
```

Командная строка

```
D:\work\C++\moveconstructor\Debug>matr_vect
constructor > A 1
constructor > B 2
copy > Copy ( A ) 3
constructor > A + B 4
move > Move_Copy ( A + B ) 4
A operator= > B 4
constructor > A + Copy ( A ) 5
move > Move_Copy ( A + Copy ( A ) ) 5
B operator-move= > Move_Copy ( A + Copy ( A ) ) 4
destructor > Move_Copy ( A + B ) 3
destructor > Copy ( A ) 2
destructor > B 1
destructor > A 0
```

Итог

Ввиду наличия большого количества стандартных классов использовать move-конструкторы и move-операции присваивания приходится редко, однако знание такого механизма необходимо.

Отношения между классами

- Наследование
- Включение
- Подобие

Наследование (inheritance)

Наследование (inheritance) – это способность класса приобретать свойства ранее определенного класса. Один класс может наследовать структуру и поведение другого класса

В языке C++ **производный** класс наследует все члены **базового** класса, **за исключением конструкторов, деструктора, перегруженной операции присваивания и определения друзей класса**.

Таким образом, производный класс содержит в себе все данные-члены и функции-члены базового класса, добавляя к ним новые члены, определенные в нём самом.

Кроме того, производный класс может изменять (переопределять) любую наследуемую функцию-член.

Правила доступа

Раздел базового класса	Открытый (public)	Защищенный (protected)	Закрытый (private)
Доступность из функций базового класса, функций дружественных классов и из дружественных функций	Да	Да	Да
Доступность из функций классов-наследников базового	Да	Да	Нет
Доступность из других классов и функций	Да	Нет	Нет

Определение производного класса

class <имя производного класса>: { public | protected | private } <имя базового класса>

Например:

```
class Student: public Person {…};

class Ball: public Sphere {…};
```

Открытое наследование

Открытое наследование устанавливает между классами отношение «**является**», или в английской нотации «**is-a**».

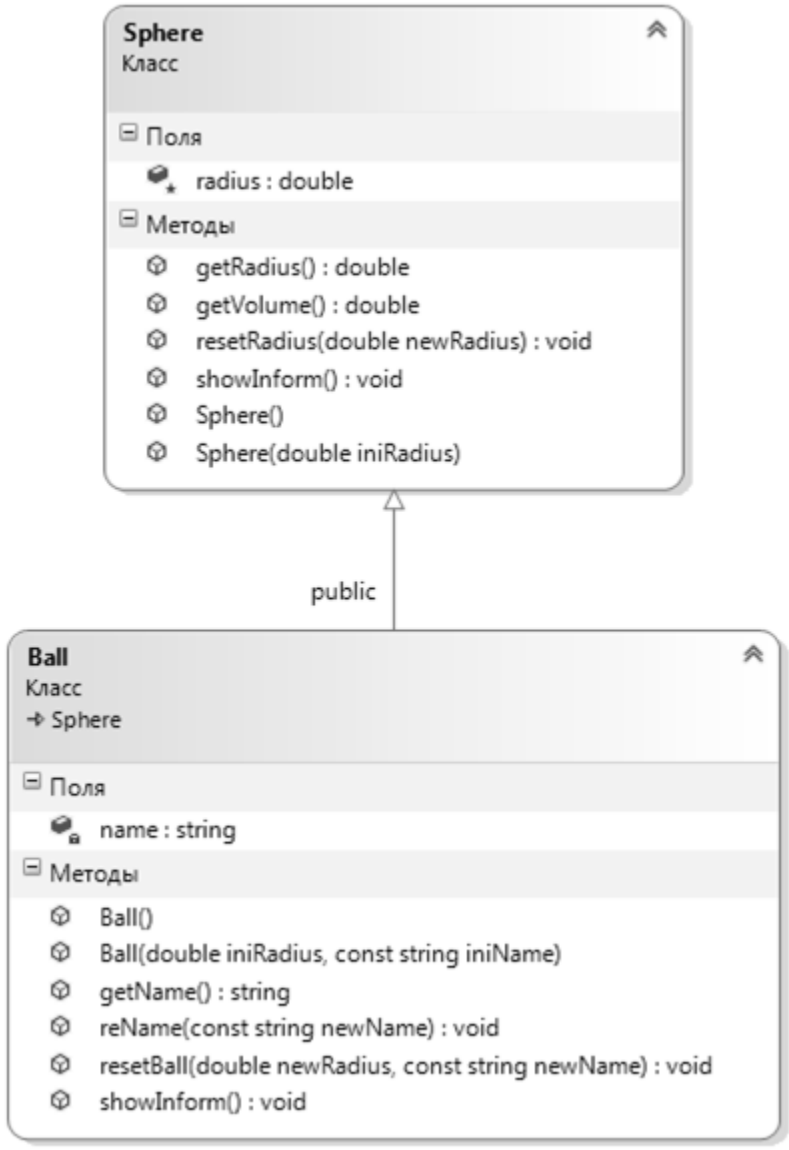
При открытом наследовании всё, что характеризует объекты класса-предка, является справедливым и для объектов класса-наследника.

Это свойство называется **совместимостью типов** объектов. Благодаря этому объект производного класса можно применять вместо объекта базового класса, но не наоборот.

Например, **объекту базового** типа можно присвоить **объект производного** типа, **указателю на объект базового** типа – **указатель на объект производного**. Объект производного типа также может быть передан в качестве параметра в функцию, вместо объекта базового типа.

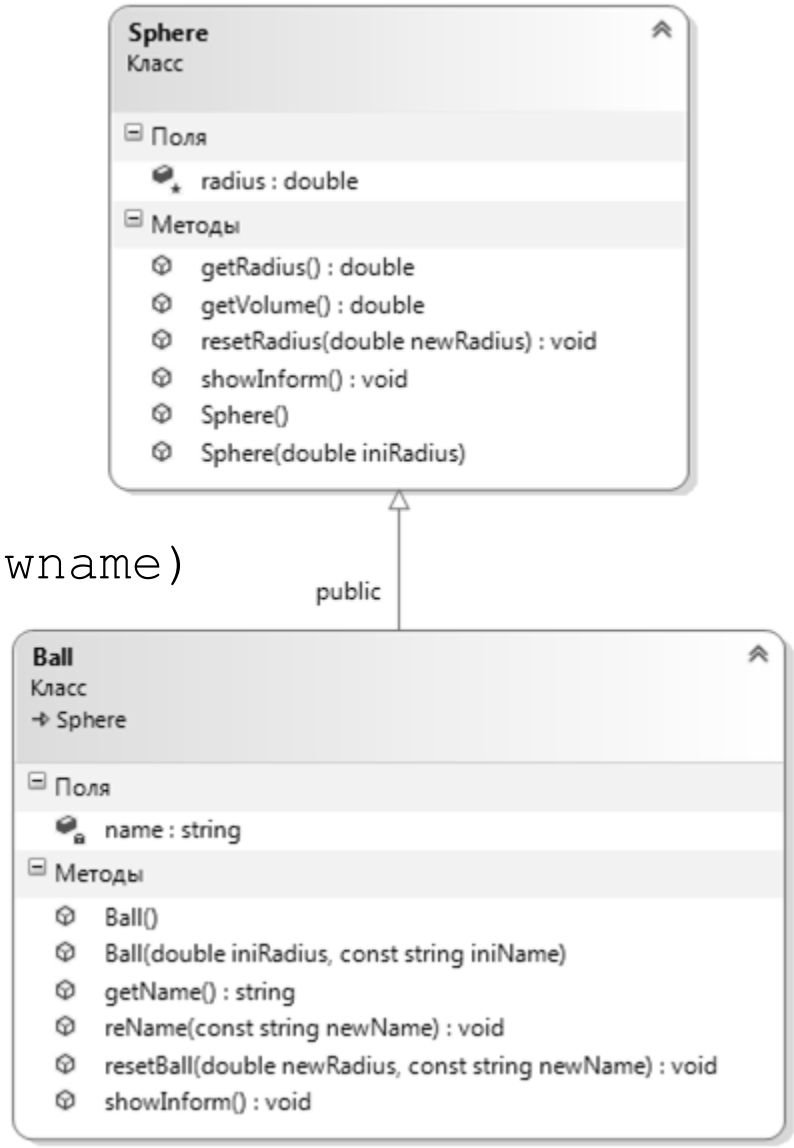
UML диаграмма иерархии классов Sphere – Ball

```
class Sphere {
  public:
    Sphere();
    Sphere(double iniRadius);
    void resetRadius(double newRadius);
    double getRadius();
    double getVolume();
    void showInform();
  protected:
    double radius;
};
```



UML диаграмма иерархии классов Sphere – Ball

```
class Ball: public Sphere{
  public:
    Ball();
    Ball(double iniRadius, const string &iniName);
    string getName();
    void reName (const string newName);
    void resetBall(double newRadius, const string &newname)
    void showInform();
  private:
    string name;
};
```



Реализация класса Sphere

```
Sphere::Sphere(): radius(1.0){ }

Sphere::Sphere(double iniRadius){
  if (iniRadius>0) radius=iniRadius; else     radius=1.0;
}
void Sphere::resetRadius(double newRadius){
  if (newRadius>0) radius=newRadius; else     radius=1.0;
}
…
double Sphere::getVolume(){
  double rad3=radius*radius*radius;
  return (4.0*3.14*rad3)/3.0;
}

void Sphere::showInform(){
  cout<<"\n Radius = "<<radius <<"\n Volume = "<<getVolume()<<endl;
}
```

Реализация класса Ball

```
Ball::Ball(): Sphere() {   setName("NoName"); }

Ball::Ball(double iniRadius, const string &iniName):
                  Sphere(iniRadius), name(iniName){}

void Ball:: reName(const string &newName){ name = newName; }

string Ball::getName() {   return name; }

void Ball::resetBall(double newRadius, const string newName) {
  radius = newRadius; // если protected
  // resetRadius(newRadius); // если private
  name = newName;
}
void Ball::showInform(){
  cout<<" Ball type - "<<name<<". Specifications:";
  Sphere::showInform(); //разрешение коллизии имён
}
```

Порядок выполнения конструкторов

Конструктор производного класса выполняется **после конструктора базового** класса.
Это правило действует и в цепочках наследования любой длины.

```
Ball::Ball(): Sphere(){   setName("NoName"); }

Ball::Ball(double iniRadius, const string iniName):
                  Sphere(iniRadius), name(iniName){}
```

Для указания, **какой именно из конструкторов** базового типа следует вызвать в каждом конкретном конструкторе класса-потомка, используется синтаксис **списка инициализаторов**.

Если конструктор базового класса отсутствует в списке инициализации, используется конструктор базового класса по умолчанию.

```
Ball::Ball(){ setName("NoName"); }
```

Конструкторы по умолчанию и наследование

Если в классе-потомке не определён ни один конструктор, то будет создан конструктор по умолчанию, который вызовет конструкторы по умолчанию всех предков.

Важно, чтобы в иерархии классов всегда были **определены конструкторы по умолчанию**.

Зоны ответственности конструкторов

Конструктор **производного** класса отвечает за инициализацию всех **элементов** данных, **добавленных** к унаследованным данным из базового класса.

Конструктор **базового** класса выполняет инициализацию **унаследованных** элементов данных.

```
Sphere::Sphere(double iniRadius){
      if (iniRadius>0) radius=iniRadius;  else  radius=1.0;
}
```

```
Ball::Ball(double iniRadius, const string iniName):
                        Sphere(iniRadius), name(iniName){}
```

Деструкторы

Деструктор производного класса выполняется **перед деструктором базового** класса.

В цепочках наследования произвольной длины деструкторы вызываются в **порядке обратном порядку вызова конструкторов**.

Вначале выполнится деструктор класса Ball, затем – деструктор класса Sphere.

Поскольку они явно не определены, то используются автоматические деструкторы.

Переопределение и замещающие функции, перегрузка

```
void Ball::showInform(){
  cout<<" Ball type - "<<name<<". Specifications:";
  Sphere::showInform(); //коллизия имён
}
```

При переопределении функции базового класса в производном классе списки параметров могут не совпадать.
При этом замещающая функция переопределяет исходную функцию, но с другим списком параметров.

Перегрузка при этом **не происходит**, так как она возможна только в одном пространстве имён.
Каждый класс имеет свое пространство имён.
Следовательно, производный класс вводит новое пространство имён.

Вызов функций-членов класса предка из наследника

```
void Ball::resetBall(double newRadius, const string newName) {
  resetRadius(newRadius); // public, но может быть  и protected
  name = newName;
}
```

```
void Ball::showInform(){
  cout<<" Ball type - "<<name<<". Specifications:";
  Sphere::showInform(); //public и переопределена
}
```

Вызов функций класса

Объекты производного класса могут вызывать открытые **функции**-члены **базового** класса.

```
Ball myBall(5.0, "Volleyball");
cout<<myBall.getVolume(); //функция предка
```

Указатели и ссылки

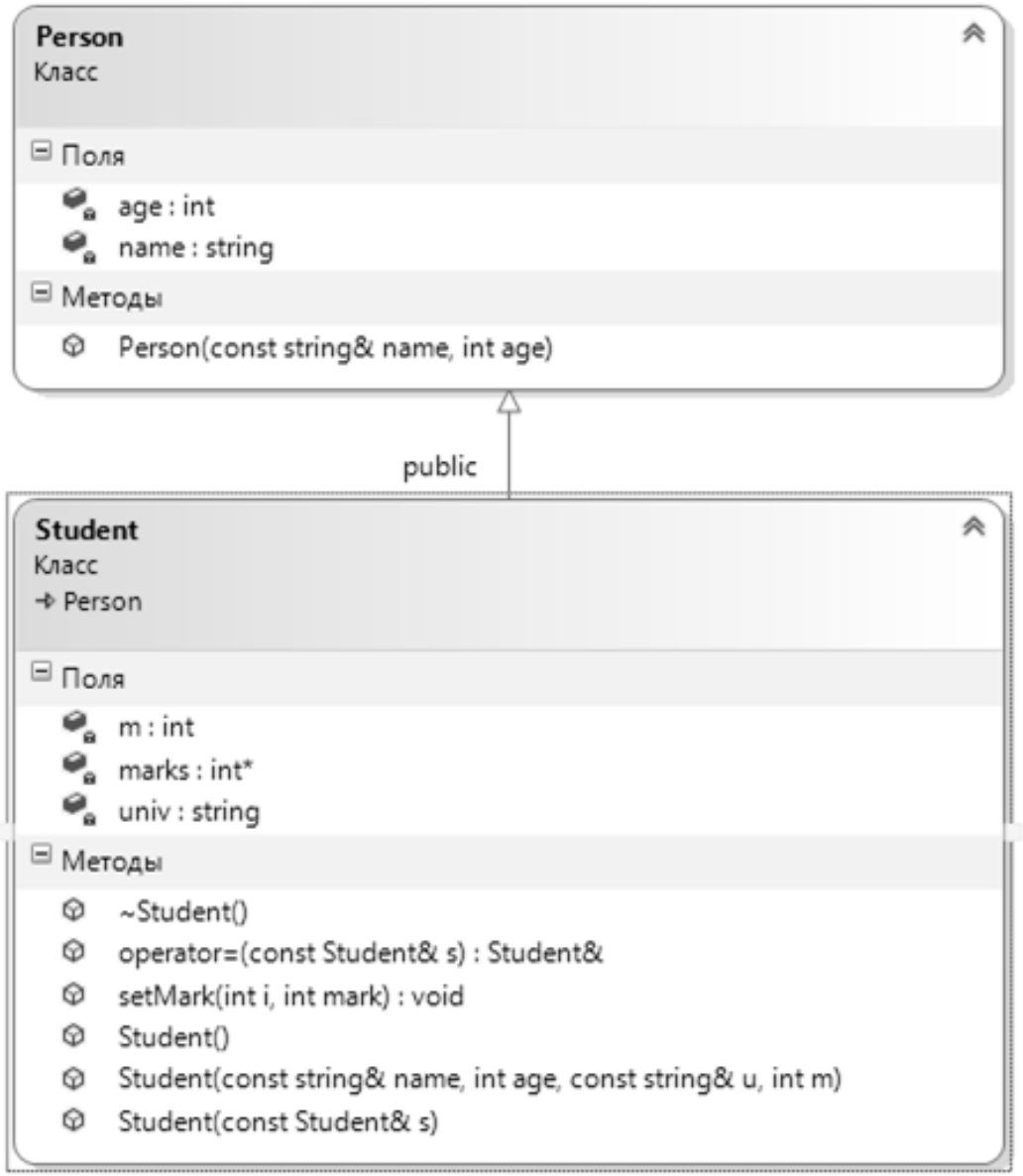
Указатель базового класса может указывать на **производный объект** без явного преобразования типов. **Ссылка** на базовый класс тоже может иметь значением адрес объекта производного класса.

```
Ball myBall(5.0, "Volleyball");
...
Sphere &pb = myBall;
Sphere *pt = &myBall;
Sphere *p = new Ball(9.0, "basketball");
pb.showInform();
cout << pb.getRadius() << endl;
pt->showInform();
cout << pt->getVolume() << endl;
p->showInform();
cout << p->getVolume() << endl;
```

Указатель или ссылка **базового** типа позволяет вызывать только **функции базового** класса, поэтому воспользоваться **pb, pt** или **p** для вызова функции getName() **производного** класса **нельзя**.

Вызов функции **showInform()** через указатели **p, pt** или ссылку **pb** на базовый класс обратится к **реализации** этой функции **в** классе **Sphere**, игнорируя ее переопределение в классе Ball

UML диаграмма иерархии классов Person – Student



Класс Person

```
class Person {
private:
  string name;
  int age;
public:
  Person(const string& name = "noname", int age = 18): name(name), age(age) {}
  friend ostream & operator<< (ostream & os, const Person & p) {
    os << p.name << " " << p.age << endl;
    return os;
  }
};
```

Класс Student – конструкторы

```
class Student: public Person {
private:
  string univ;     // Университет
  int m;
  int* marks;      // Оценки
public:
  Student() : univ("МГУ"), m(3)   {
    marks = new int[m];
    for (int i = 0; i<m; ++i)
      marks[i] = 0;
  }

  Student(const string& name, int age, const string& u, int m):
    Person(name, age), univ(u), m(m)   {
    marks = new int[m];
    for (int i = 0; i<m; ++i)
      marks[i] = 0;
  }
```

Класс Student – конструкторы, деструктор

```
Student(const Student& s): Person(s), univ(s.univ), m(s.m){
    marks = new int[s.m];
    for (int i =0; i<m; ++i)
      marks[i] = s.marks[i];
  }

  ~Student() { delete[] marks; }
```

Класс Student – функции и операции

```
Student& operator=(const Student& s) {
    if (&s != this) {
      delete[] marks;
      Person::operator=(s);
      univ = s.univ;
      m = s.m;
      marks = new int[m];
      for (int i = 0; i<m; ++i)
        marks[i] = s.marks[i];
    }
    return *this;
  };
```

Класс Student – функции и операции

```
void setMark(int i, int mark) {
    if (i < 0 || i >= m)
      throw - 1;
    marks[i] = mark;
  }

  friend ostream & operator<< (ostream & os, const Student & s) {
    os<<(Person)s;
    os << s.univ << endl;
    for (int i = 0; i < s.m; ++i)
      os << s.marks[i] << " ";
    os << endl;
    return  os;
  }
};
```

```
int main() {
  Student s("Pinokkio", 18, "ЮФУ", 3);
  s.setMark(0, 5);
  s.setMark(1, 4);
  cout << s;
  Student s1(s);
  cout << s1;
  Student s2;
  cout << s2;
  s2 = s1;
  cout << s2;
  system("PAUSE");
  return 0;
}
```

Наличие **динамически выделяемой памяти** int* marks в классе Student создаёт дополнительные проблемы. В этом случае нельзя использовать создаваемые по умолчанию функции: **конструктор без параметров, конструктор копии, деструктор и операцию присваивания**.
Необходимо **предусмотреть их реализацию** в классе.

Приведение типов upcast

Приведение типа-наследника к базовому типу

```
Student(const Student& s): Person(s), univ(s.univ), m(s.m) {
  marks = new int[s.m];
  for (int i =0; i<m; ++i)
    marks[i] = s.marks[i];
}
```

Фактически const Person & присваиваем переменную типа const Student &

Преобразование в иерархии «предок-потомок»

Переменной типа предок можно присвоить переменную типа потомок, но не наоборот.

```
Person p("Иванов", 20);
Student s("Петров", 19, "ЮФУ", 3);
p = s;
```

При присваивании объекта производного класса переменной базового класса происходит **копирование только полей базового** класса, остальная часть информации объекта производного класса будет утеряна.

```
s = p; // ошибка компиляции
```

Попытка присвоить объекту класса наследника объект класса предка приведёт к ошибке компиляции.

Преобразование в иерархии «предок-потомок»

При работе с указателями и ссылками на объекты предка и наследника действует аналогичное правило преобразования типов

```
Person* pp = &p;
Student* ss = &s;
pp = ss;
ss = pp; // ошибка компиляции

Person& rp = s;
Student& rs = p; // ошибка компиляции

Student(const Student& s): Person(s), univ(s.univ), m(s.m)   {
    marks = new int[s.m];
    for (int i =0; i<m; ++i)
      marks[i] = s.marks[i];
  }
```

Операция присваивания наследника подробнее

Операция присваивания не наследуется

Поэтому, чтобы выполнить присваивание для private полей, унаследованных от базового класса Person, необходимо выполнить операцию присваивания, определённую в классе Person.

```
Person::operator=(s);
```

```
Student& operator=(const Student& s) {
    if (&s != this) {
      delete[] marks;
      Person::operator=(s);
      univ = s.univ;
      m = s.m;
      marks = new int[m];
      for (int i = 0; i<m; ++i)
        marks[i] = s.marks[i];
    }
    return *this;
};
```

Операция вывода в поток наследника подробнее

Поскольку операция вывода в поток является внешней, то к ней невозможно применить разрешение области видимости.
Поэтому используется явное приведение типа.

```
os<<(Person)s;
```

```
friend ostream & operator<< (ostream & os, const Student & s) {
    os<<(Person)s;
    os << s.univ << endl;
    for (int i = 0; i < s.m; ++i)
      os << s.marks[i] << " ";
    os << endl;
    return  os;
  }
```

Приведение типов downcast

Приведение базового типа к типу-наследнику

В этом случае необходимо использовать явное приведение типов с помощью шаблонной функции

static_cast<тип_наследника>

Приведение типов downcast

Добавим в класс Student функцию get_univ(), которая возвращает название учебного заведения, где учится студент.
Такой функции нет, и не может быть в Person.

```
class Student : public Person {
  …
public:
    …
  string get_univ() const  {
    return univ;
  }
};
```

Теперь рассмотрим ситуацию, когда нам может понадобиться приведение базового типа к типу наследника

```
Person *p = new Student("Петров", 19, "ЮФУ", 3);
p->get_univ();    // ошибка компиляции
```

При вызове p->get_univ()произойдёт ошибка компиляции, так как в Person не определена функция get_univ(). Для корректного вызова указатель p нужно привести к типу *Student

```
// Современный стиль:
static_cast<Student*>(p)->get_univ();

// Старый стиль:
((Student*)p)->get_univ;
```

Аналогичная ситуация возникает и при использовании ссылок:

```
Person & rp = *new Student("Петров", 20, "ЮФУ", 3);
static_cast<Student &>(rp).get_univ();
delete &rp;
```

Ограничения преобразования downcast

Преобразование downcast в C++ возможно только для **указателей или ссылок** на объекты.

Корректное преобразование downcast возможно только **как обратное** преобразование к upcast.

Отношение включения

Каждый ресурс, под который выделяется память в конструкторе, обычно стремятся обернуть объектом класса, контролирующим этот ресурс, что упрощает код.

В этом случае между классами возникает не отношение наследование, а отношение включения («has-a»). Оно означает, что один класс содержит в качестве члена объект другого/других классов.

Модифицировать класс Student из иерархии Person-Student, используя для хранения оценок разработанный ранее класс Array

```
class Student: public Person {
private:
  string univ;     // Университет
  Array marks;      // Оценки
public:
  Student() : marks(3), univ("МГУ") {      }
  Student(const string& name, int age, const string& u, int m):
                              Person(name, age), univ(u), marks(m) { }
```

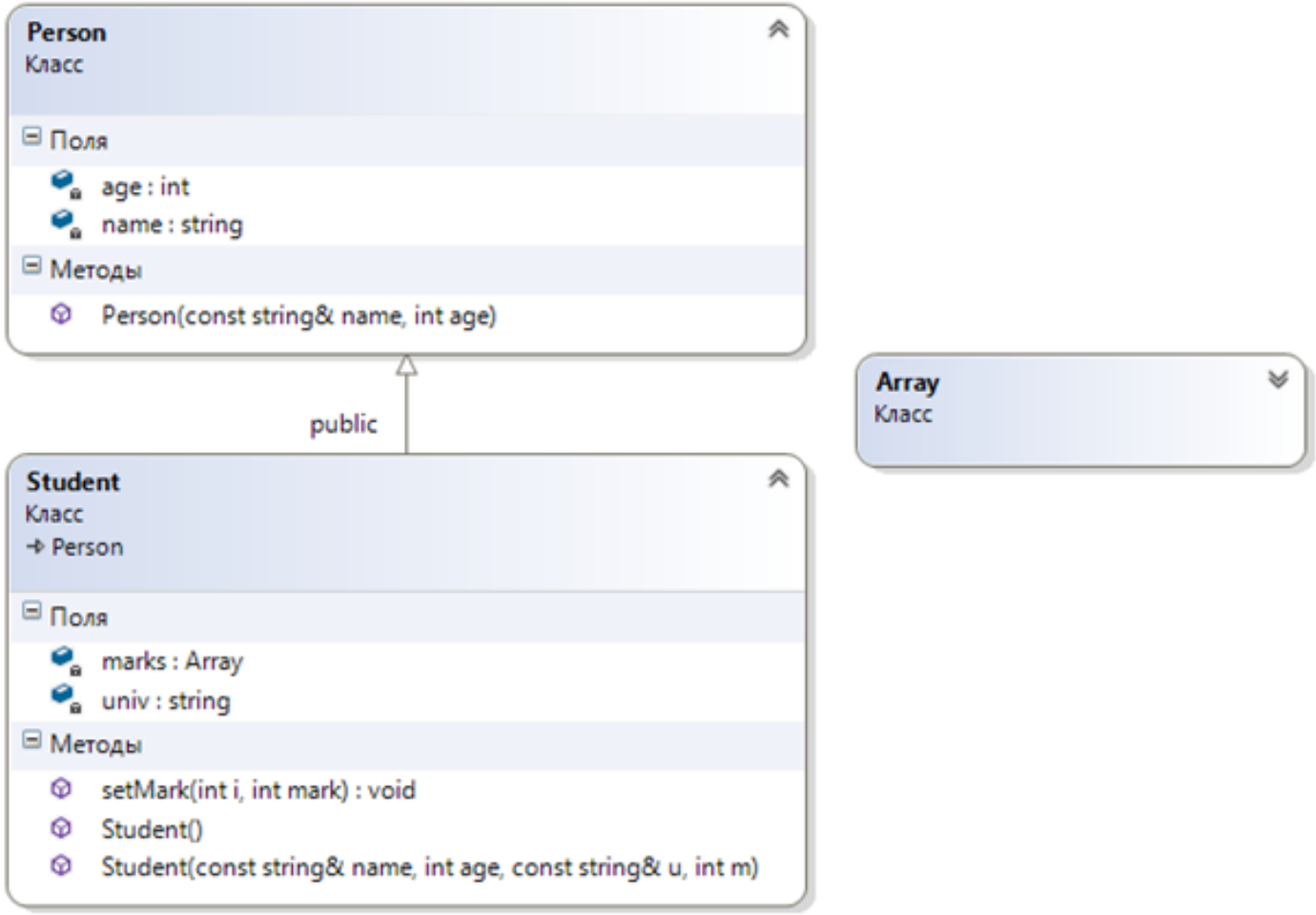
Модифицировать класс Student из иерархии Person-Student, используя для хранения оценок разработанный ранее класс Array

```
void setMark(int i, int mark) {
    if (i < 0 || i >= marks.length())
       throw - 1;
    marks[i] = mark;
  }

  friend ostream & operator<< (ostream & os, const Student & s)   {
    os<<(Person)s;
    os << s.univ << endl;
    for (int i = 0; i < s.marks.length(); ++i)
            os << s.marks[i] << " ";      //??
    os << endl;
    return  os;
  }
};
```

Модифицировать класс Student из иерархии Person-Student, используя для хранения оценок разработанный ранее класс Array

```
void setMark(int i, int mark) {
    if (i < 0 || i >= marks.length())
       throw - 1;
    marks[i] = mark;
  }

  friend ostream & operator<< (ostream & os, const Student & s)   {
    os<<(Person)s;
    os << s.univ << endl;
    os << marks << endl;
    return  os;
  }
};
```

UML диаграмма классов



Порядок вызова конструкторов и деструкторов

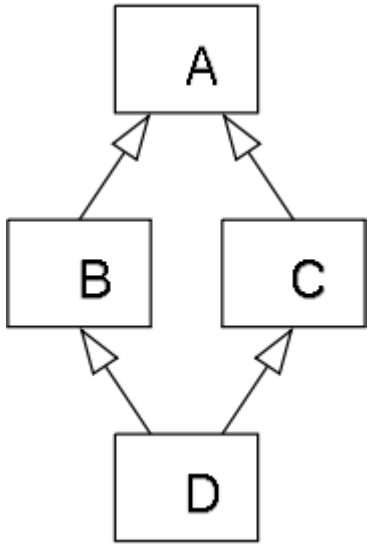
1. Вызов конструктора базового класса
2. Вызов конструктора полей
3. Вызов конструктора основного объекта
4. Вызов деструктора основного класса
5. Вызов деструкторов полей
6. Вызов деструктора предка

Множественное наследование. Проблемы

```
class orient {
public:
    void print() { std::cout << "orient" << std::endl;
}
};

class point {
public:
    void print() { std::cout << "point" << std::endl; }
};

class vector : public point, public orient {};

int main() {
    vector v;
    v.print();
}
```

"vector::print" не является однозначным
неоднозначный уровень доступа "print"

Проблема ромба



Проблема ромба (Diamond problem) — классическая проблема в языках, которые поддерживают возможность множественного наследования.

В C ++ решение проблемы оставлено на усмотрение программиста.

Ромбовидная проблема

Ромбовидная проблема — прежде всего проблема дизайна, и она должна быть предусмотрена на этапе проектирования.

На этапе разработки ее можно разрешить следующим образом:

- вызвать метод конкретного суперкласса;
- обратиться к объекту подкласса как к объекту определенного суперкласса;
- переопределить проблематичный метод в последнем дочернем классе.

```
class Device {
    public:
        void turn_on() { cout << "Device is on." << endl; }
};

class Computer: public Device {};

class Monitor: public Device {};

class Laptop: public Computer, public Monitor {
    /*
    public:
        void turn_on() {
            cout << "Laptop is on." << endl;
        }
    // uncommenting this function will resolve diamond problem
    */
};
```

```
int main() {
    Laptop Laptop_instance;

    // calling method of specific superclass
    Laptop_instance.Monitor::turn_on();

    // treating Laptop instance as Monitor instance via static cast
    static_cast<Monitor&>( Laptop_instance ).turn_on();
    return 0;
}
```

Проблема ромба: Конструкторы и деструкторы

```
class Device {
    public:
        Device() { cout << "Device constructor called" << endl; }
};

class Computer: public Device {
    public:
        Computer() { cout << "Computer constructor called" << endl; }
};

class Monitor: public Device {
    public:
        Monitor() {  cout << "Monitor constructor called" << endl;  }
};

class Laptop: public Computer, public Monitor {};
```

Проблема ромба: Конструкторы и деструкторы

```
int main() {
    Laptop Laptop_instance;
    return 0;
}
```

Решение

Следует далее