# Коллоквиум 2015: Условия, ответы и решения, критерии

Вылиток А.А.

**За каждую задачу максимум 10 баллов, минимум 0 баллов. (При вычитании баллов за ошибки по критерию задачи полагаем 0-x=0 для любого x.) В варианте 1 некоторые задачи прокомментированы более подробно.**

## Вариант 1_2015

**1**. Приведите фрагмент программы с примером ***ошибочного*** обращения к видимой, но недоступной функции.

Ответ: `class A {void f (){}; }; class B: public A { public: void g(){ f(); } };`

```
// метод f () в приватной части A  недоступен для наследника
```

Критерий: пример не удовлетворяет условию или отсутствует : -10

**2.** Что будет напечатано данной программой? Можно ли удалить хотя бы одну операцию разрешения области видимости так, чтобы печать не изменилась?

```
namespace N { int f= 2;
              int g=-2;
}
namespace M { int f= 3;
              int g=-3;
              int h=0;
}
int f=-5;
```

```
int main() { int f=1;
    cout<<::f<<N::f<<M::f<<M::h<<endl;
  using namespace N;
    cout<<f<<g<<M::h<<endl;
  using namespace M;
    cout<<f<<M::g<<h<<endl;
    return 0;
}
```

Решение

Рассмотрим первый оператор с печатью.
Выражение `::f` отсылает нас к глобальному объекту `f` типа `int` (в глобальной области видимости) со значением `-5`. Если убрать четвероточие, то будет действовать описание объекта `f` из `main()` со значением `1`, так как локальное имя скрывает глобальное в блоке (от описания до конца блока).
Выражение `N::f` означает `f` из пространства имен `N` со значением `2`; если убрать `N::`, то будет действовать локальное описание со значением `1`.
Выражение `M::f` означает `f` из пространства имен `M` со значением `3`; если убрать `M::`, то будет действовать локальное описание со значением `1`.
Выражение `M::h` означает `h` из пространства имен `M` со значением `0`; если убрать `M::`, то имя `h` будет невидимым, так как описано в области `M`, в которую `main()` (содержащая обращение к `h`) не входит.

После **using namespace** N; становятся видимыми без префикса те имена из N, которые не конфликтуют с местными (локальными) именами. Конфликт в таком случае разрешается всегда в пользу местных (локальных) – они главнее. Так, например, f по-прежнему означает локальный в main() объект со значением 1. Имя f из пространства N не становится видимым без N::, а имя g становится видимым без N::, так как ни с кем не конфликтует.

После **using namespace** M; имя h из M становится также видимым без префикса, однако теперь обращение к g без префикса становится ошибочным (неоднозначным, ambiguous), так как два одинаковых пришлых имени **g** (из **M** и из **N**) вступают в конфликт. Конфликт не решается в пользу какого-либо из них (они «равны» между собой – оба пришлые, и в результате «оба наказаны, как нашкодившие мальчишки»). Имя f по-прежнему означает локальный в main() объект со значением 1, пришлое из M имя f не становится видимым без префикса (конфликт решается в пользу местного f).

Ответ : -5230
1-20
1-30
Удалить :: нельзя.

Критерий: за каждую неверно напечатанную строчку: -3
за неправильный ответ на второй вопрос: -4

**3.** Можно ли продемонстрировать параметрический полиморфизм с помощью оператора-выражения a+(b+a); ? Обоснуйте ответ. Перечислите другие виды полиморфизма.

Ответ: другие виды полиморфизма – статический, динамический.

Параметрический продемонстрировать можно: нужно привести такое описание шаблонной операции, чтобы разные вхождения + в выражение a+(b+a) настраивались на разные типы операндов (получаем, что одна и та же операция означает разное в зависимости от контекста – это и есть проявление полиморфизма). Однако, если приведен пример с настройкой на один и тот же тип, то такой ответ тоже засчитываем без снятия баллов. Обоснование может быть схематичным, например, без реализации тел функций.

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
T1 operator+ (T1 a, T2 b) {cout<<a<<"+"<<b<<" | "; return a;
}
class A{
	char c;
	public:
	A():c('A'){}
	friend ostream & operator<<(ostream & out, const A & a);
};
ostream & operator<<(ostream & out, const A & a ){
	out<<a.c;
	return out;
}
```

```
class B{
	char c;
	public:
	B():c('B'){}
	friend ostream & operator<<(ostream & out, const B & b);
};
ostream & operator<<(ostream & out, const B & b ){
	out<<b.c;
	return out;
}

int main() {
A a; B b;
a+(b+a); // operator+ (a, operator+(b,a));
	  // Напечатается : B+A | A+B |
	return 0;
}
```

Критерий:

за каждый не названный (или не существующий ) вид полиморфизма: -2
за неверный ответ или отсутствующее обоснование про параметрический полиморфизм: -6

**4.** Данная программа выводит на печать несколько строк. Вычеркните ***фрагменты*** из функции ***main*** так, чтобы напечаталось верное утверждение об ООП и С++.

```
class A {
public:
   A() { cout<<"абстрактный "; }
   virtual ~A() {cout<<  endl;}
   void t(int n){cout<<"конструктор ";}
   virtual void t(){cout<<"метод ";}
};
class B : public A {
public:
    B(){}
    B(int x) {cout<<"класс ";}
    void t() { cout<<"тип ";}
    void t(int n) { cout<<"интерфейс ";}
    ~B() {cout<< "данных";}
};
```

```
int main() {
    cout<<"класс, не содержащий открытых \
           членов-данных, -- это " ;
    A * qa= new A;
    qa->t(5);
    delete qa;
    qa=new B;
    qa->A::t();
    delete qa;
    qa= new B;
    { A & ra= *qa;
      ra.t();
    }
    delete qa;
    B * qb= new B;
```

```
    qb->t(5);
    delete qb;
    B b(5);
}
```

Ответ: напечатаются строки

```
Класс, не содержащий открытых членов-данных, -- это абстрактный конструктор
абстрактный метод данных
абстрактный тип данных
абстрактный интерфейс данных
абстрактный класс данных
```

Единственный правильный вариант: *класс, в котором нет открытых членов-данных, -- это абстрактный тип данных*. Поэтому в теле main нужно оставить только:

```
cout<<  "класс, не содержащий открытых членов-данных, -- это " ;
A * qa;
qa= new B;
{ A & ra= *qa;
ra.t();
}
delete qa;
```

Остальные фрагменты текста удаляются.

Другой способ получить тот же ответ :

```
B * qb= new B;
qb->t(); // здесь было qb->t(5); -- вычеркнули  фрагмент «5»
delete qb;
```

Еще один способ :

```
A * qa;
qa=new B;
qa-> t(); // здесь было qa->A::t();-- вычеркнули фрагмент «A::»
delete qa;
```

Критерий: за неверный ответ или вычеркивание не в main: -10

**5.** Может ли одна и та же функция одновременно участвовать в перегрузке (*overloading*), скрытии (*hiding*) и замещении (*overriding*) ? Обоснуйте ответ.

***Примечание***: замещение проявляется при работе механизма виртуальных функций.

Ответ : да, в задаче 4 таковой является функция void t(); из класса B.

Критерий: -10 за неправильный или не обоснованный ответ.

**6.** Добавьте реализацию метода `sum` (из класса `C`), который вычисляет сумму всех числовых полей объекта класса `C` (каждое поле учитывается в сумме один раз). Операцию `::` можно использовать не более одного раза в каждом первичном выражении (т.е. `Y::x` возможно, `Z::Y::x` – нет).

***Примечание***: у операции `->` приоритет выше, чем у операции приведения типа `(тип) x`.

```
struct Base {  int a; };

struct B1 :  virtual  Base
{  int a; };
```

```
struct A1 : Base {  int a; };

struct B2 :  virtual Base

{  int a; };
```

```
struct A2 : Base { int a;};

struct C : A1,  A2, B1,  B2

{ int sum();  int a; };
```

Решение

Для начала выясним, сколько всего полей в объекте класса *C* ?

1) *a* из *C* (объявлено непосредственно в *C*)

2) *a* из *A1* (объявлено в *A1*)

3) *a* из *A2*

4) *a* из *B1*

5) *a* из *B2*

6) *a* из *Base*, который является частью *A1*

7) *a* из *Base*, который является частью *A2*

8) *a* из *Base*, который является общей частью *B1* и *B2* («виртуальное» наследование)

Как «добраться» в методе *sum*( ) до каждого поля *a* из каждого класса? С первыми пятью из перечисленных в списке проблем нет. Например: *A1::a* означает поле *a*, объявленное в классе *A1*.

Выражение *Base::a* в классе *C* неоднозначно, т.к. *Base* присутствует в классе *C* в трех экземплярах — непонятно, какое из трех *a* имеется ввиду.

[ Компилятор Visual C++ умеет различать их таким способом:

*A1::Base::a*, *A2::Base::a*, *B1::Base::a* . Вместо последнего можно было также написать *B2::Base::a* (оба способа именуют одно и то же поле).

Компилятор g++ считает имена *A1::Base::a*, *A2::Base::a*, *B1::Base::a* неоднозначными. Он, скорее всего, прав, поскольку грамматика C++ так устроена, что :: получается правоассоциативной операцией. Поэтому, чтобы не сталкиваться с разным поведением компиляторов, просто запрещаем в условии задачи дважды использовать :: в одном первичном выражении ]

Как быть? Надо же как-то добраться и до остальных *a* тоже.

Вспомним, что у нас есть указатель на объект *this*. Если привести его явно к указателю на *A2*, получим указатель на подобъект типа *A2* из нашего объекта, далее приводим полученный указатель к указателю на *Base* и получаем указатель на подобъект типа *Base*, причем именно тот *Base*, который находится в *A2*. Также и до остальных двух *Base*'овых полей можно добраться. Другой допустимый вариант: привести *this* сначала к указателю на *A1* (соответственно *A2*, *B1*) и использовать доступ к полю *Base:: a* через указатель, например:

( (*A1**) *this*)*->Base:: a* [ а вот так ( &(*A1*(** this*)))*->Base:: a* нельзя, поскольку у временного объекта нельзя взять адрес ]

Итого:

```
int C::sum( ) { return a + A1::a+ A2::a+ B1::a + B2::a +
                    ( (Base*) (A1*) this)->a  +      // другой вариант ( (A1*) this)->Base:: a
                    ( (Base*) (A2*) this)->a  +
                    ( (Base*) (B1*) this)->a  ; }
```

Чтобы проверить, можно воспользоваться следующей функцией:

```
int main() {
        C c, *p=&c; c.A1::a=1; c.A2::a=1; c.a=1;c.B1::a=1; c.B2::a=1; ((A1*)p)->Base::a=1;
        ((B1*)p)->Base::a=1;  ((A2*)p)->Base::a=1;   //  ((B2*)p)->Base::a=1;
        cout<<c.sum();
        return 0;
}
```

Ответ:

```
int C::sum( ) { return a + A1::a+ A2::a+ B1::a + B2::a +
                    ( (Base*) (A1*) this)->a  +      // другой  вариант: ( (A1*) this)->Base:: a
                    ( (Base*) (A2*) this)->a  +      // другой  вариант: ( (A2*) this)->Base:: a
                                                     //  или   ( (B2*) this)->Base:: a
                    ( (Base*) (B1*) this)->a  ; }    // другой  вариант: ( (B1*) this)->Base:: a
```

Критерий: за каждое неверное слагаемое : -3

**7.** Что будет напечатано в результате выполнения программы? Компилятор ***не*** оптимизирующий.

```
class A {
    static int c;
    int n;
public:
    A():n(++c){
       cout<<" A()="<< n;
    }

    A(const A&a):n(++c){
       cout<<" A(A&)="<< n;
    }
    ~A(){
       cout<<" ~A()="<< n;
    }
};
```

```
void f(){
    try {throw A(); throw 1;}
    catch (int) {cout << "int";}
    catch (A a) {throw;}
    cout<< " f() ";
}
int A::c=0;
int main() {
    try {
        try { f(); cout<< " after_f ";}
        catch(A & a) {throw a;}
       catch(...) { cout << "inner ..."; }
    }
    catch (...) {cout << " ... ";}
}
```

Конструктор **A()** в операторе **throw A();** будет вызываться для инициализации временного объекта, с которого будет «сниматься копия» конструктором копирования для инициализации создаваемого объекта-исключения, после чего исключение «полетит», а временный объект, проинициализированный **A()**, исчезнет. Компиляторы обычно оптимизируют эту ситуацию, не делая лишнего копирования. Однако в условии явно сказано, что оптимизации нет. Поэтому ответ не совсем похож на то, что выдаст скомпилированная программа. В случае **throw a;** временный объект не создается, копия при инициализации нового объекта-исключения снимается с уже существующего объекта **a**.

Ответ: A()=1 A(A&)=2 ~A()=1 A(A&)=3 ~A()=3 A(A&)=4 ~A()=2 ... ~A()=4

Критерий: за каждую ошибку -3

**8.** Опишите прототипы двух перегруженных функций `f` из некоторой области видимости, для которых будут верны следующие обращения к ним:

```
f (-6);
f ('+', 6);
f ();
f ("abc");
```

Ответ: `f(int i=5, int j=9); f(const char * s);` //Возможны другие решения

Критерий: -3 за каждый неподходящий вызов из четырех, но не меньше 0.

Если больше двух функций описано: 0

**9.** Объясните, какой смысл имеет выражение `typeid(*p)`.

Ответ : динамическое определение типа объекта, на который указывает указатель p.

Критерий: - 10 за отсутствие или неправильный ответ.

**10.** Перепишите данный фрагмент, не используя средства из C++11:

```
long  m;
decltype(m) n = 50000;
vector<long> v;
…
for (auto p=v.begin(); p!=v.end(); ++p) {n+=*p;}
```

Ответ:

```
long  m,   n = 50000;
vector <long> v;
…
for (vector <long>::iterator p = v.begin(); p != v.end(); ++p) {n+= *p;}
```

Критерий: неправильно заменен decltype: -5

неправильно заменен auto: -5

## Вариант 2_2015

**1.** Приведите пример ***ошибочного*** обращения к доступному, но не видимому (без операции `::`) методу класса.

Ответ: `class A {public: void f (){}; }; class B: public A { public: void g(int f) {f();};`

// параметр f скрыл метод f() из A

Или `class A {public: void f (){}; }; class B: public A { int f; public: void g(){f();}`

// поле f скрыло метод f() из A

Критерий: пример не удовлетворяет условию или отсутствует : -10

**2.** Что будет напечатано данной программой? Можно ли удалить хотя бы одну операцию разрешения области видимости так, чтобы печать не изменилась?

```
namespace M {
    int f(){return 2;}
    int g=-2;
}
namespace N {
    int f(){return 3;}
    int g=-3;
    int h=0;
}
```

```
int f() {return -7;}
int main() { int g=-1;
    cout<<f()<<N::f()<<M::f()<<N::h<<endl;
  using namespace N;
    cout<<N::f()<<g<<h<<endl;
  using namespace M;
    cout<<M::f()<<M::g<<h<<endl;
    return 0;
}
```

Ответ :
```
-7320
3-10
2-20
```

`Удалить :: нельзя.`

Критерий: за каждую неверно напечатанную строчку: -3

за неправильный ответ на второй вопрос: -4

**3.** Можно ли продемонстрировать параметрический полиморфизм с помощью оператора-выражения `(b+a)+b;` ? Обоснуйте ответ. Перечислите другие виды полиморфизма.

Ответ: другие виды полиморфизма – статический, динамический.

Параметрический продемонстрировать можно: нужно привести такое описание шаблонной операции, чтобы разные вхождения + в выражение (b+a)+b настраивались на разные типы операндов (получаем, что одна и та же операция означает разное в зависимости от контекста – это и есть проявление полиморфизма). Однако, если приведен пример с настройкой на один и тот же тип, то такой ответ тоже засчитываем без снятия баллов. Обоснование может быть схематичным, например, без реализации тел функций.

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
T2 operator+ (T1 a, T2 b) {cout<<a<<"+"<<b<<" | "; return b;
}
class A{
	char c;
	public:
	A():c('A'){}
	friend ostream & operator<<(ostream & out, const A & a);
};
ostream & operator<<(ostream & out, const A & a ){
	out<<a.c;
	return out;
}

class B{
	char c;
	public:
	B():c('B'){}
	friend ostream & operator<<(ostream & out, const B & b);
};
ostream & operator<<(ostream & out, const B & b ){
	out<<b.c;
	return out;
}

int main() {
A a; B b;
(b+a)+b; // operator+ ( operator+(b,a),b);
	 // Напечатается : B+A | A+B |
	return 0;
}
```

Критерий:

за каждый не названный (или не существующий ) вид полиморфизма: -2

за неверный ответ или отсутствующее обоснование про параметрический полиморфизм: -6

**4.** Данная программа выводит на печать несколько строк. Вычеркните ***фрагменты*** из функции ***main*** так, чтобы напечаталось верное утверждение об ООП и С++.

```
    class A {
public:
    A() { cout<<"абстрактный "; }
    virtual ~A() {cout<<  endl;}
    void g(int n){cout<<"класс ";}
    virtual void g(){cout<<"метод ";}
};


class B : public A {
public: B(){}
    B(int x) {cout<<"деструктор ";}
    void g() { cout<<"конструктор ";}
    void g(int n) { cout<<"интерфейс ";}
    ~B() {cout<< "данных";}
};
```

```
int main() {
    cout<<" класс, содержащий чистую \
            виртуальную функцию, -- это ";
    A * qa= new B;
    qa->A::g();
    delete qa;
    qa= new B;
    { A & ra= *qa;
      ra.g();
    }
    delete qa;
    B * qb= new B;
    qb->g(7);
    delete qb;
    qa= new A;
    qa->g(7);
    delete qa;
    B b(7);
}
```

Ответ: напечатаются строки

```
класс, содержащий чистую     виртуальную функцию, -- это абстрактный метод данных
абстрактный конструктор данных
абстрактный интерфейс данных
абстрактный класс
абстрактный деструктор данных
```

Единственный правильный вариант: *класс, содержащий чистую виртуальную функцию, -- это абстрактный класс*. Поэтому в теле main нужно оставить только:

```
cout<<  "класс, содержащий чистую виртуальную функцию, -- это " ;
A * qa;
qa=new A;
qa-> g(7);
delete qa;
```

Критерий: за неверный ответ или вычеркивание не в main: -10

**5.** Может ли функция одновременно участвовать в скрытии (*hiding*) и замещении (*overriding*) и при этом не участвовать в перегрузке (*overloading*)? Обоснуйте ответ.

***Примечание***: замещение проявляется при работе механизма виртуальных функций.

Ответ : да, если в задаче 4 в классе B вычеркнуть g(int); то оставшаяся функция void g(); из класса B будет удовлетворять условию задачи.

Критерий: -10 за неправильный или не обоснованный ответ.

**6.** Добавьте реализацию метода `prod` (из класса `C`), который вычисляет произведение всех числовых полей объекта класса `C` (каждое поле учитывается в произведении один раз). Операцию `::` можно использовать не более одного раза в каждом первичном выражении (т.е. `Y::x` возможно, `Z::Y::x` – нет).

***Примечание***: у операции `->` приоритет выше, чем у операции приведения типа `(тип) x`.

```
class Base
{ public: int a; };

class B1 :
  virtual public Base
{ public:  int a; };
```

```
class A1 : public Base
{ public: int a; };

class B2 :
   virtual public Base
{ public:  int a; };
```

```
class A2 : public Base
{ public: int a;};

class C : public A1, public A2,
          public B1, public B2
{ public: int prod();  int a; };
```

Ответ:

*int C::prod*( ) { *return a * A1::a* A2::a* B1::a * B2::a **

( (*Base**) (*A1**) *this*)->*a* * // *другой вариант:* ( (*A1**) *this*)->*Base:: a*

( (*Base**) (*A2**) *this*)->*a* *

( (*Base**) (*B1**) *this*)->*a* ; }

Критерий: за каждый неверный сомножитель : -3

**7.** Что будет напечатано в результате выполнения программы? Компилятор ***не*** оптимизирующий.

```
class A {
    static int c;
    int n;
public:
    A():n(++c){
       cout<<" A()="<< n;
    }

    A(const A&a):n(++c){
       cout<<" A(A&)="<< n;
    }
    ~A(){
       cout<<" ~A()="<< n;
    }
};
```

```
void f(){ try {throw A(); throw 1;}
          catch (int) {cout << "int";}
          catch (A &) {throw;}
          cout<< " f() ";
        }
int A::c=0;
int main(){
    try {
        try { f(); cout<< " after_f ";}
        catch(A  a) {throw a;}
        catch(...) { cout << "inner ..."; }
    }
    catch (...) {cout << " ... ";}
    return 0;
}
```

Ответ: A()=1 A(A&)=2 ~A()=1 A(A&)=3 A(A&)=4 ~A()=3 ~A()=2 ... ~A()=4

Критерий: за каждую ошибку -3

**8.** Опишите прототипы двух перегруженных функций h из некоторой области видимости, для которых будут верны следующие обращения к ним:

```
h (-6,"bcd");
h('-', 6);
h ();
h (3.5);
```

Ответ: `h(double d=5, int j=9); h(int i, const char * s);` // Возможны другие решения

Критерий: -3 за каждый неподходящий вызов из четырех, но не меньше 0.

Если больше двух функций описано: 0

**9.** Объясните, какой смысл имеет выражение `typeid(имя_типа)`.

Ответ : в этом случае операция typeid возвращает ссылку на объект типа type_info, представляющий тип именно с этим именем.

Критерий: - 10 за отсутствие или неправильный ответ.

**10.** Перепишите данный фрагмент, не используя средства из C++11:

```
double  t;
decltype(t) x = 50.000;
list<double> l;
…
auto p=l.rbegin();
while (p!=l.rend()) {
    x-= *p; p++;
}
```

Ответ:

```
double  t,   x = 50.000;
list <double> l;
…
list <double>::reverse_iterator p = l.rbegin();
while  (p != l.rend())   {x-= *p;  p++;}
```

Критерий: неправильно заменен decltype: -5

неправильно заменен auto: -5

## Вариант 3_2015

**1.** Что будет напечатано данной программой? Можно ли удалить хотя бы одну операцию разрешения области видимости так, чтобы печать не изменилась?

```
namespace D { int f= 2;
              int g=-2;
}
namespace C { int f= 3;
              int g=-3;
              int h=0;
}
int f=-5;
```

```
int main() { int f=1;
    cout<<::f<<D::f<<C::f<<C::h<<endl;
  using namespace D;
    cout<<f<<g<<C::h<<endl;
  using namespace C;
    cout<<f<<C::g<<h<<endl;
    return 0;
}
```

Ответ : -5230
1-20
1-30
Удалить :: нельзя.

Критерий: за каждую неверно напечатанную строчку:-3
за неправильный ответ на второй вопрос: -4

**2.** Опишите прототипы двух перегруженных функций g из некоторой области видимости, для которых будут верны следующие обращения к ним:

```
g (-6);
g ('+', 6);
g ();
g ("abc");
```

Ответ: `g(int i=5, int j=9); g(const char * s);` // Возможны другие решения

Критерий: -3 за каждый неподходящий вызов из четырех, но не меньше 0.

Если больше двух функций описано: 0

**3.** Перепишите данный фрагмент, не используя средства из C++11:

```
long  w;
decltype(w) n = 70000;
vector<long> v;
…
for (auto p=v.begin(); p!=v.end(); ++p) {n+=*p;}
```

Ответ:

```
long  w,   n = 70000;
vector <long> v;
…
for (vector <long>::iterator p = v.begin(); p != v.end(); ++p) {n+= *p;}
```

Критерий: неправильно заменен decltype: -5

неправильно заменен auto: -5

**4.** Объясните, какой смысл имеет выражение `typeid(*p)`.

Ответ : динамическое определение типа объекта, на который указывает указатель p.

Критерий: - 10 за отсутствие или неправильный ответ.

**5.** Что будет напечатано в результате выполнения программы? Компилятор ***не*** оптимизирующий.

```
class Y {
    static int c;
    int n;
public:
    Y():n(++c){
       cout<<" Y()="<< n;
    }
        Y(const Y&a):n(++c){
       cout<<" Y(Y&)="<< n;
    }
    ~Y(){
       cout<<" ~Y()="<< n;
    }
};
```

```
void h(){
    try {throw Y(); throw 1;}
    catch (int) {cout << "int";}
    catch (Y a) {throw;}
    cout<< " h() ";
}   int Y::c=0;
int main() {
    try {
        try { h(); cout<< " after_h ";}
        catch(Y & a) {throw a;}
       catch(...) { cout << "inner ..."; }
    }
    catch (...) {cout << " ... ";}
}
```

Ответ: `Y()=1 Y(Y&)=2 ~Y()=1  Y(Y&)=3  ~Y()=3 Y(Y&)=4 ~Y()=2 ...  ~Y()=4`

Критерий: за каждую ошибку -3

**6.** Можно ли продемонстрировать параметрический полиморфизм с помощью оператора-выражения `a*(b*a);` ? Обоснуйте ответ. Перечислите другие виды полиморфизма.

Ответ: другие виды полиморфизма – статический, динамический.

Параметрический продемонстрировать можно: нужно привести такое описание шаблонной операции, чтобы разные вхождения * в выражение a*(b*a) настраивались на разные типы операндов (получаем, что одна и та же операция означает разное в зависимости от контекста – это и есть проявление полиморфизма). Однако, если приведен пример с настройкой на один и тот же тип, то такой ответ тоже засчитываем без снятия баллов. Обоснование может быть схематичным, например, без реализации тел функций.

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
T1 operator* (T1 a, T2 b) {cout<<a<<"*"<<b<<" | "; return a;
}
class A{
   char c;
   public:
   A():c('A'){}
   friend ostream & operator<<(ostream & out, const A & a);
};
ostream & operator<<(ostream & out, const A & a ){
   out<<a.c;
   return out;
}

class B{
   char c;
   public:
   B():c('B'){}
   friend ostream & operator<<(ostream & out, const B & b);
};
ostream & operator<<(ostream & out, const B & b ){
   out<<b.c;
   return out;
}

int main() {
A a; B b;
a*(b*a); // operator* (a, operator*(b,a));
         // Напечатается : B*A | A*B |
   return 0;
}
```

Критерий: за каждый не названный (или не существующий ) вид полиморфизма: -2

за неверный ответ или отсутствующее обоснование про параметрический полиморфизм: -6

**7.** Данная программа выводит на печать несколько строк. Вычеркните ***фрагменты*** из функции ***main*** так, чтобы напечаталось верное утверждение об ООП и С++.

```
class X {
public:
   X() { cout<<"абстрактный "; }
   virtual ~X() {cout<<  endl;}
   void t(int n){cout<<"конструктор ";}
   virtual void t(){cout<<"метод ";}
};
class Y : public X {
public:
    Y(){}
    Y(int x) {cout<<"класс ";}
    void t() { cout<<"тип ";}
    void t(int n) { cout<<"интерфейс ";}
    ~Y() {cout<< "данных";}
};
```

```
int main() {
    cout<<"класс, не содержащий открытых \
          членов-данных, -- это " ;
    X * qx= new X;
    qx->t(5);
    delete qx;
    qx=new Y;
    qx->X::t();
    delete qx;
    qx= new Y;
    { X & rx= *qx;
      rx.t();
    }
    delete qx;
    Y * qy= new Y;
    qy->t(5);
    delete qy;
    Y y(5);
}
```

Ответ: напечатаются строки

```
Класс, не содержащий открытых членов-данных, -- это абстрактный конструктор
абстрактный метод данных
абстрактный тип данных
абстрактный интерфейс данных
абстрактный класс данных
```

Единственный правильный вариант: *класс, в котором нет открытых членов-данных, -- это абстрактный тип данных*. Поэтому в теле main нужно оставить только:

```
cout<<  "класс, не содержащий открытых членов-данных, -- это " ;
X * qx;
qx= new Y;
{ X & rx= *qx;
rx.t();
}
delete qx;
```

Критерий: за неверный ответ или вычеркивание не в main: -10

**8.** Может ли одна и та же функция одновременно участвовать в перегрузке (*overloading*), скрытии (*hiding*) и замещении (*overriding*) ? Обоснуйте ответ.

***Примечание***: замещение проявляется при работе механизма виртуальных функций.

Ответ : да, в задаче 7 таковой является функция void t(); из класса Y.

Критерий: -10 за неправильный или не обоснованный ответ.

**9.** Добавьте реализацию метода `sum` (из класса `C`), который вычисляет сумму всех числовых полей объекта класса `C` (каждое поле учитывается в сумме один раз). Операцию `::` можно использовать не более одного раза в каждом первичном выражении (т.е. `Y::x` возможно, `Z::Y::x` – нет).

***Примечание***: у операции `->` приоритет выше, чем у операции приведения типа `(тип) x`.

```
struct Base {  int b; };

struct B1 :  virtual  Base
{  int b; };
```

```
struct A1 : Base {  int b; };

struct B2 :  virtual Base

{  int b; };
```

```
struct A2 : Base { int b;};

struct C : A1,  A2, B1,  B2

{ int sum();  int b; };
```

Ответ:

*int C::sum*( ) { *return b + A1::b+ A2::b+ B1::b + B2::b +*

( (*Base**) (*A1**) *this*)*->b + // другой вариант:* ( (*A1**) *this*)*->Base:: b*

( (*Base**) (*A2**) *this*)*->b +*

( (*Base**) (*B1**) *this*)*->b ;* }

Критерий: за каждое неверное слагаемое : -3

**10.** Приведите фрагмент программы с примером ***ошибочного*** обращения к видимой, но недоступной функции.

Ответ: `class A {void f (){}; }; class B: public A { public: void g(){ f(); } };`

`// метод f () в приватной части A недоступен для наследника`

Критерий: пример не удовлетворяет условию или отсутствует : -10

## Вариант 4_2015

1. Что будет напечатано данной программой? Можно ли удалить хотя бы одну операцию разрешения области видимости так, чтобы печать не изменилась?

```
namespace L {
    int f(){return 2;}
    int g=-2;
}
namespace K {
    int f(){return 3;}
    int g=-3;
    int h=0;
}
int f() {return -7;}
int main() { int g=-1;
    cout<<f()<<K::f()<<L::f()<<K::h<<endl;
  using namespace K;
    cout<<K::f()<<g<<h<<endl;
  using namespace L;
    cout<<L::f()<<L::g<<h<<endl;
    return 0;
}
```

Ответ : -7320
3-10
2-20

Удалить :: нельзя.

Критерий: за каждую неверно напечатанную строчку: -3
за неправильный ответ на второй вопрос: -4

**2.** Опишите прототипы двух перегруженных функций p из некоторой области видимости, для которых будут верны следующие обращения к ним:

```
p();
p(-6,"qxq");
p(-0.5);
p('*', 9);
```

p(double d=5, int j=9); p(int i, const char * s); // Возможны другие решения

Критерий: -3 за каждый неподходящий вызов из четырех, но не меньше 0.

Если больше двух функций описано: 0

**3.** Перепишите данный фрагмент, не используя средства из C++11:

```
double  r;
decltype(r) x = 70.000;
list<double> l;
…
auto p=l.rbegin();
while (p!=l.rend()) {
    x-= *p; p++;
}
```

Ответ:

```
double  r,   x = 70.000;
list  <double> l;
…

list <double>::reverse_iterator p = l.rbegin();
while  (p != l.rend())   {x-= *p;  p++;}
```

Критерий: неправильно заменен decltype: -5

неправильно заменен auto: -5

**4.** Объясните, какой смысл имеет выражение **`typeid`**(*имя_типа*).

Ответ : в этом случае операция typeid возвращает ссылку на объект типа type_info, представляющий тип именно с этим именем.

Критерий: - 10 за отсутствие или неправильный ответ.

**5.** Что будет напечатано в результате выполнения программы? Компилятор ***не*** оптимизирующий.

```
class B {
    static int c;
    int n;
public:
    B():n(++c){
       cout<<" B()="<< n;
    }

    B(const B&a):n(++c){
       cout<<" B(B&)="<< n;
    }
    ~B(){
       cout<<" ~B()="<< n;
    }
};
```

```
void s(){ try {throw B(); throw 1;}
          catch (int) {cout << "int";}
          catch (B &) {throw;}
          cout<< " s() ";
        }
int B::c=0;
int main(){
    try {
        try { s(); cout<< " after_s ";}
        catch(B  a) {throw a;}
        catch(...) { cout << "inner ..."; }
    }
    catch (...) {cout << " ... ";}
    return 0;
}
```

Ответ: B()=1  B(B&)=2  ~B()=1  B(B&)=3 B(B&)=4  ~B()=3  ~B()=2 ...  ~B()=4

Критерий: за каждую ошибку -3

**6.** Можно ли продемонстрировать параметрический полиморфизм с помощью оператора-выражения `(b*a)*b;` ? Обоснуйте ответ. Перечислите другие виды полиморфизма.

Ответ: другие виды полиморфизма – статический, динамический.

Параметрический продемонстрировать можно: нужно привести такое описание шаблонной операции, чтобы разные вхождения * в выражение (b*a)*b настраивались на разные типы операндов (получаем, что одна и та же операция означает разное в зависимости от контекста – это и есть проявление полиморфизма). Однако, если приведен пример с настройкой на один и тот же тип, то такой ответ тоже засчитываем без снятия баллов. Обоснование может быть схематичным, например, без реализации тел функций.

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
T2 operator* (T1 a, T2 b) {cout<<a<<"*"<<b<<" | "; return b;
}
```

```
class A{
	char c;
	public:
	A():c('A'){}
	friend ostream & operator<<(ostream & out, const A & a);
};
ostream & operator<<(ostream & out, const A & a ){
	out<<a.c;
	return out;
}

class B{
	char c;
	public:
	B():c('B'){}
	friend ostream & operator<<(ostream & out, const B & b);
};
ostream & operator<<(ostream & out, const B & b ){
	out<<b.c;
	return out;
}

int main() {
A a; B b;
(b*a)*b; // operator*( operator*(b,a),b);
	  // Напечатается : B*A | A*B |
	return 0;
}
```

Критерий:

за каждый не названный (или не существующий ) вид полиморфизма: -2

за неверный ответ или отсутствующее обоснование про параметрический полиморфизм: -6

**7.** Данная программа выводит на печать несколько строк. Вычеркните ***фрагменты*** из функции ***main*** так, чтобы напечаталось верное утверждение об ООП и C++.

```
class Y {
public:
    Y() { cout<<"абстрактный "; }
    virtual ~Y() {cout<<  endl;}
    void g(int n){cout<<"класс ";}
    virtual void g(){cout<<"метод ";}
};


class Z : public Y {
public: Z(){}
```

```
int main() {
    cout<<" класс, содержащий чистую \
            виртуальную функцию, -- это ";
    Y * qy= new Z;
    qy->Y::g();
    delete qy;
    qy= new Z;
    { Y & ry= *qy;
      ry.g();
    }
```

```
    Z(int x) {cout<<"деструктор ";}
    void g() { cout<<"конструктор ";}
    void g(int n) { cout<<"интерфейс ";}
    ~Z() {cout<< "данных";}
};
```

```
    delete qy;
    Z * qz= new Z;
    qz->g(7);
    delete qz;
    qy= new Y;
    qy->g(7);
    delete qy;
    Z z(7);
}
```

Ответ: напечатаются строки

```
класс, содержащий чистую  виртуальную функцию, -- это абстрактный метод данных
абстрактный конструктор данных
абстрактный интерфейс данных
абстрактный класс
абстрактный деструктор данных
```

Единственный правильный вариант: *класс, содержащий чистую виртуальную функцию, -- это абстрактный класс*. Поэтому в теле main нужно оставить только:

```
cout<<  "класс, содержащий чистую виртуальную функцию, - это " ;
Y * qy;
qy=new Y;
qy-> g(7);
delete qy;
```

Критерий: за неверный ответ или вычеркивание не в main: -10

**8.** Может ли функция одновременно участвовать в скрытии (*hiding*) и замещении (*overriding*) и при этом не участвовать в перегрузке (*overloading*)? Обоснуйте ответ.

***Примечание***: замещение проявляется при работе механизма виртуальных функций.

Ответ : да, если в задаче 7 в классе Z вычеркнуть g(int); то оставшаяся функция void g(); из класса Z будет удовлетворять условию задачи.

Критерий: -10 за неправильный или не обоснованный ответ.

**9.** Добавьте реализацию метода `prod` (из класса `C`), который вычисляет произведение всех числовых полей объекта класса `C` (каждое поле учитывается в произведении один раз). Операцию `::` можно использовать не более одного раза в каждом первичном выражении (т.е. `Y::x` возможно, `Z::Y::x` – нет).

***Примечание***: у операции `->` приоритет выше, чем у операции приведения типа `(тип) x`.

```
class Base
{ public: int b; };

class B1 :
  virtual public Base
{ public:  int b; };
```

```
class A1 : public Base
{ public: int b; };

class B2 :
   virtual public Base
{ public:  int b; };
```

```
class A2 : public Base
{ public: int b;};

class C : public A1, public A2,
          public B1, public B2
{ public: int prod();  int b; };
```

Ответ:

*int C::prod*( ) { *return b * A1::b* A2::b* B1::b * B2::b **

( (*Base**) (*A1**) *this*)->*b* * // *другой вариант:* ( (*A1**) *this*)->*Base:: b*

( (*Base**) (*A2**) *this*)->*b* *

( (*Base**) (*B1**) *this*)->*b* ; }

Критерий: за каждый неверный сомножитель : -3

**10.** Приведите пример ***ошибочного*** обращения к доступному, но не видимому (без операции `::`) методу класса.

Ответ: `class A {public: void f (){}; }; class B: public A { public: void g(int f) {f();};`

// параметр f скрыл метод f() из A

Или `class A {public: void f (){}; } ; class B: public A { int f; public: void g(){f();};`

// поле f скрыло метод f() из A

Критерий: пример не удовлетворяет условию или отсутствует: -10