# Билет 4.5. Приватные и публичные поля и методы. Особенности выбора версии функции при перегрузке в случае выбора между приватными и публичными версиями (с примерами). Функции-друзья и классы-друзья, синтаксис объявления, пример использования: перегрузка операторов ввода-вывода. Свойства отношения дружбы в C++.

Публичные поля и методы доступны всем. Приватные поля и методы доступны только классу и его друзьям. Для понимания выбора версии функции важно запонмить золотое правило: Перегрузка выполняется до того как происходит проверка доступа. Проверка доступа — в самый последний момент.

Приватное поле, к которому нет доступа, не то же самое, что поля вообще нет. visible не равно accessible. Видимые – те, которые находит поиск имен. Доступные – те, к которым есть доступ по модификаторам доступа при наследовании.

```
1  struct Granny{
2      void f(){
3          std::cout << "Granny";
4      }
5  };
6
7  struct Mom : private Granny{
8      void f(int y){
9          std::cout << "Mom";
10     }
11 };
12
13 struct Son : Mom {
14     void f(double y){};
15 };
16
17 int main(){
18     Mom m;
19     m.f(); // output: "Mom"
20     Son s;
21     //s.f(); //CE, this func is inaccesable
22     //m.Granny::f(); // CE, as Granny inaccessible
23     s.Mom::f(0);
24 }
```

При вызове метода f от объекта типа Mom вызовется метод из функции Mom, так как этот метод, будучи названным так же, как и метод наследуемого класса, "перекрывает"функцию f из Granny, f из Granny - not visible. Поля и методы с одинаковыми именами в классе-наследнике более локальные, чем в классе родителя. Поля класса-родителя перекрываются и не видны из класса наследника.

Доступность проверяется после разрешения перегрузки и выбора версий. В данной версии бабушкина версия не видна, а мамина недоступна - CE:

```
struct Granny{
    void f(){
        std::cout << "Granny";
    }
    void g(double);
};

struct Mom : Granny{
private:
    void f(){
        std::cout << "Mom";
    }
    void g(int);
};

int main(){
    Mom m;
    m.f();
    m.g(0.0);// conversion to int, bc even if Grannies candidate is perfect, it is not visible
    m.Granny f(0.0) // will work
    std::cout << m.a;

}
```

Существует ключевое слово friend, которое позволяет объявлять “друзей” класса, то есть функции/классы/etc, которые имеют доступ ко всему(!), к чему имеет доступ наш класс (разве что кроме полей/методов класса, чьим другом является наш класс. Дружба не транзитивна). Дружба не наследуется.

```
class String {
public:
    String(const char[]);
    // позволяет определить оператор вывода в поток
    friend std::ostream& operator<<(std::ostream& out, const String& s);
    friend std::istream& operator>>(std::istream& in, const String& s);
private:
    char* buffer;
    size_t size;
};

std::ostream& operator<<(std::ostream& out, const String& s) {
    for (int i = 0; i < s.size; ++i) {
        out << s.buffer[i];
    }
    return out;
}

std::istream& operator>>(std::istream& in, BigInteger& number) {
    std::string input;
    in >> input;
    // ...длинный код преобразования строки ввода input в BigInteger...
    return in;
}
```

# Билет 4.6. Ключевое слово explicit, два возможных контекста его использования. Перегрузка операторов приведения типа для классов, пример. Определение пользовательских литеральных суффиксов для классов, пример.

Ключ-слово explicit — запрещают неявную конверсию типов. Использовать его можно в двух ситуациях: в конструкторах и операторах привидения типа.

```
struct UserId {
    int id = 0;
    explicit UserId(int x): id(x) {}

    explicit operator int() { // тип возвращаемого значения указывать не нужно
        // возвращаем число по UserId
    }
};

int main{
    UserId u = 5; //CE
    UserId u2(5);

    return 0;
}
```

Типичный пример перегрузки операторов приведения типа — operator bool() в BigInteger: при записи f(x) подстановка считается явной (контекстуальной конверсией). Тогда этот оператор просто проверяте, что return (BigInteger != 0);

Перегрузка литеральных суффиксов, это когда вы перегрузили 5_uid и теперь, если вы где-то в коде напишите 5_uid, то это будет UserId с id = 5; (Нижнее подчеркивание — часть синтаксиса)

```
UserId operator ""_uid(unsigned long long x) {
    return UserId(5);
}

int main() {
    UserId u = 5; //still CE
    UserId u2 = 5_uid;
}
```

# Билет 4.7. Модификатор доступа protected для полей и методов. Приватное и публичное наследование. Разница между наследованием классов и структур. Разница между видимостью и доступностью. Видимость и доступность различных членов родителя в теле наследника при приватном и публичном наследовании, явное обращение к методам родителя, использование using.

**protected** — доступ открыт классам, производным от данного, и друзьям. То есть производные классы и друзья получают свободный доступ к таким данным или методам. Все другие классы такого доступа не имеют.

- Public-наследование.При таком наследовании protected и public данные из базового класса остаются, соответственно protected и public в производном классе. Все знают о факте наследования (имеют доступ к родителю через наследника).
- Private-наследование - означает, что поля, которые достались наследнику от родителей являются private, и к ним нельзя получить доступ извне, не будучи членом класса-наследника или другом класса-наследник. Derived запретил доступ к полям, наследованным из Base, поэтому даже если какая-то функция была другом Base, эта функция не имеет доступ к полям класса Derived, унаследованным из Base. Приватность устанавливается на уровне наследника.Кроме того, из наследника нельзя обращаться к приватной части родителя. О факте наследования знает только наследник.
- Protected-наследование. Данные, которые в Base были protected и public, становятся protected. О факте наследования знают только наследники и друзья.

  Поля класса могут быть protected.

Классы по умолчанию наследуются private, структуры — public.

Видимость — это то, какие функции и переменные видны в данной области видимости. Если несколько одноименных функций/переменных есть в области видимости, то, одна из них затмевает остальные, и, как следствие, остальные не видны в области видимости. Версия сына всегда затмевает версию родителя. Доступность — то, имеет ли данная область видимости доступ к перменной или функции. Золотое правило: Доступность проверяется после видимости и выбора перегрузки. Рассмотрим на примере:

```
1  class Base {
2  public:
3      int a = 0;
4      void f() {}
5      void g(int a) {}
6  };
7
8  class Derived : public Base {
9  protected:
10     int a = 1;
11 public:
```

```
12      void f() {}
13      void g() {}
14  };
15
16  int main() {
17      Derived d; //1
18      d.f(); // Вызовется та, что из Derived
19      d.g(3); // CE, g(int) доступна, но не видна
20      d.Base::f(); // OK
21      d.Base::g(3); // OK
22      d.a; // CE, видна, но недоступна
23      d.Base::a; //OK
24      return 0;
25  }
```

Добавить в область видимости g из Base можно, дописав в public часть класса Derived строчку using Base::g, аналогично при написании в private часть он был бы видим, но недоступен. При приватном наследование Base будет вне зоны видимости, поэтому даже сама запись d.Base будет уже CE.

## Билет 4.8. Размещение объектов в памяти при наследовании. Порядок вызова конструкторов и деструкторов, а также инициализации полей при наследовании. Наследование конструкторов родителя с помощью using. Обращение к конструкторам родителя из конструкторов наследника. Множественное наследование. Проблема ромбовидного наследования, размещение объекта в памяти при таком наследовании.

Пусть класс Derived – наследник класса Base. Тогда в памяти объект типа Derived лежит так: [Base][Derived] (сначала все поля от родителя, а потом все поля от сына).

Конструкторы вызываются в порядке от самого дальнего предка до нас, деструкторы — наоборот. Важно помнить, что если у родителя нет конструктора по умолчанию, то мы должны явно инициализировать “родительскую” часть нашего класса:

```
1   class Base {
2   public:
3       int a;
4       Base(int a) : a(a) {};
5   };
6
7   class Derived : public Base {
8       int b;
9       //Derived(int b) : b(b)  нельзя писать, т.к. у Base нет конструктора по умолчанию
10      //Derived(int a, int b) : a(a), b(b)  тоже нельзя, потому что можно инициализировать только
        свои поля
11      Derived(int b) : Base(a), b(b) {};
12  };
```

К слову, так можно инициализировать ближайших предков, более дальних нельзя (при невиртуальном наследовании).

**Множественное наследование** — наследование от нескольких классов.

Нужно быть аккуратными с полями классов – они могут быть одинаковы у обоих родителей.

```
class Mother {
public:
    int a = 1;
};
class Father {
public:
    int a = 2;
};
class Son: public Mother, public Father {
    int s = 3;
};
int main() {
    Son s;
    cout << s.a;
    //CE: request for member "a"is ambiguous
}
```

**Проблема ромбовидного наследования** Рассмотрим следующий код, который при $Granny\&g = s$; выдаст неоднозначный каст. Тут две разные бабушки лежат в M и F, если обратимся к полю g у сына, будет CE. Размер сына 20, там две копии g. Сын в памяти лежит как [g][m][g][f][s]

```
struct Granny {
    int g = 0;
};
struct Mother: public Granny {
    int m = 1;
};
struct Father: public Granny {
    int f = 1;
};
struct Son: public Mother, public Father {
    int s = 3;
};
```

Еще один пример:

```
struct A {
    int a;
    int f() {};
};

struct B1 : A {};
struct B2 : A {};
struct C : B1, B2 {};

int main()
```

```
{
    C c;
    c.a; //CE
    c.f(); //тоже нельзя
    c.B1::f(); // можно
}
```

Проблема снова заключается в том, что C унаследован от B1 и B2, унаследованных от A (в памяти лежит приблизительно так: [A][B1][A][B2][C]), и получается как бы два A. То есть в выражениях (1) и (2) неизвестно, к каким именно a и f мы обращаемся, которые от того A, что от B1, или от того A, что от B2. A в выражении (3) всё хорошо, потому что однозначно. (Это ошибка “Ambiguous base class”) (Заметим, что не имеет значения, как именно был унаследован: public, private или protected) Здесь же можно сказать о том, что если у вас B унаследован от A, и C унаследован от A и B, то вы опять же не сможете обращаться к полям A через объект класса C. В таких случаях возникает “warning: inaccessible base class”.

Поскольку в C++ при инициализации объекта дочернего класса вызываются конструкторы всех родительских классов, возникает и другая проблема: конструктор базового класса бабушки будет вызван дважды.

Про **using**:

```
struct B {
    B(int = 13, int = 42);
};
struct D : B {
    using B::B;
};
```

Еще примеры.

```
struct B1 {  B1(int, ...) { } };
struct B2 {  B2(double)   { } };

int get();

struct D1 : B1 {
  using B1::B1;   // inherits B1(int, ...)
  int x;
  int y = get();
};

void test() {
  D1 d(2, 3, 4); // OK: B1 is initialized by calling B1(2, 3, 4),
                 // then d.x is default-initialized (no initialization is performed),
                 // then d.y is initialized by calling get()
  D1 e;          // Error: D1 has no default constructor
}

struct D2 : B2 {
  using B2::B2; // inherits B2(double)
  B1 b;
};
```

```
23
24 D2 f(1.0);          // error: B1 has no default constructor
```

Как и в случае использования-объявлений для любых других нестатических функций-членов, если унаследованный конструктор соответствует сигнатуре одного из конструкторов Derived, он скрывается от поиска версией, найденной в Derived. Если один из унаследованных конструкторов Base имеет сигнатуру, которая соответствует конструктору копирования / перемещения производного, это не предотвращает неявную генерацию производного конструктора копирования / перемещения (который затем скрывает унаследованную версию).

```
1 struct B1 {   B1(int); };
2 struct B2 {   B2(int); };
3
4 struct D2 : B1, B2 {
5   using B1::B1;
6   using B2::B2;
7   D2(int);    // OK: D2::D2(int) hides both B1::B1(int) and B2::B2(int)
8 };
9 D2 d2(0);    // calls D2::D2(int)
```