

## Билет 4.4. Статические переменные, особенности и примеры их использования. Статические поля и методы классов, статические константы, их особенности и примеры использования. Понятие статической памяти, разница между статической и стековой памятью.

Статические переменные: Использование ключевого слова `static` с локальными переменными изменяет их свойство продолжительности жизни с автоматического на статическое (или «фиксированное»). Статическая переменная (или «переменная со статической продолжительностью жизни») сохраняет свое значение даже после выхода из блока, в котором она определена. То есть она создается (и инициализируется) только один раз, а затем сохраняется на протяжении выполнения всей программы.

```
1 #include <iostream>
2
3 void incrementAndPrint()
4 {
5     static int s_value = 1; \\ переменная s_value является статической
6     ++s_value;
7     std::cout << s_value << std::endl;
8 } \\ переменная s_value не уничтожается здесь, но становится недоступной
9
10 int main()
11 {
12     incrementAndPrint(); \\ cout << 2
13     incrementAndPrint(); \\ cout << 3
14     incrementAndPrint(); \\ cout << 4
15 }
```

Статические поля и методы: поля и методы обозначенные словом `static`, являются общими для всего класса, а не отдельного объекта, при том выделяются в статической памяти.

```
1 struct A {
2     static int a;
3 };
4
5 int A::a = 1;
6
7 int main() {
8     A a;
9     std::cout << a.a++ << A::a++ << A().a++; \\ 123
10 }
```

Для метода аналогично. Примером их использования может быть счетчик количества объектов данного класса. Или синглтон. Из статических методов нельзя обращаться к нестатическим полям класса. Статический переменные можно инициализировать внутри класса, только если они константные и целочисленные. Выделяются статические переменные при запуске программы в статической памяти.

Статическая память: память выделяющаяся в начале работы программы, которая содержит статические переменные и поля, а также глобальные переменные. При запуске программы упрощенно компьютер выделяет память:

[DATA | TEXT | STACK - 4(8)MB]

На стеке выделяется обычно 4 или 8 мегабайт. DATA хранит статические переменные, там выделяется статическая память. TEXT содержит код программы. Стэк хранит локальные переменные, а также указатели на запускаемые функции, то есть рекурсия происходит на стеке. Потому может произойти Stack Overflow — переполнение стека, например, при рекурсивном запуске функции из самой себя много раз подряд (Segmentation Fault).

**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 запомнить 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(); // CE, Granny's version will be taken
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 - СЕ:

```
1 struct Granny{
2     void f(){
3         std::cout << "Granny";
4     }
5     void g(double);
6 };
7
8 struct Mom : Granny{
9 private:
10    void f(){
11        std::cout << "Mom";
12    }
13    void g(int);
14 };
15
16 int main(){
17     Mom m;
18     m.f();
19     m.g(0.0); // conversion to int, bc even if Grannies candidate is perfect, it is not visible
20     m.Granny f(0.0) // will work
21     std::cout << m.a;
22
23 }
```

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.

```
1 class String {
2 public:
3     String(const char[]);
4     // позволяет определить оператор вывода в поток
5     friend std::ostream& operator<<(std::ostream& out, const String& s);
6     friend std::istream& operator>>(std::istream& in, const String& s);
7 private:
8     char* buffer;
9     size_t size;
10 };
11
12 std::ostream& operator<<(std::ostream& out, const String& s) {
13     for (int i = 0; i < s.size; ++i) {
14         out << s.buffer[i];
15     }
16     return out;
```

```

17 }
18
19 std::istream& operator>>(std::istream& in, BigInteger& number) {
20     std::string input;
21     in >> input;
22     // ...длинный код преобразования строки ввода input в BigInteger...
23     return in;
24 }

```

## 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 приведения типа.

```

1 struct UserId {
2     int id = 0;
3     explicit UserId(int x): id(x) {}
4
5     explicit operator int() { // тип возвращаемого значения указывать не нужно
6         // возвращаем число по UserId
7     }
8 };
9
10 int main{
11     UserId u = 5; //CE
12     UserId u2(5);
13
14     return 0;
15 }

```

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

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

```

1 UserId operator ""_uid(unsigned long long x) {
2     return UserId(5);
3 }
4
5 int main() {
6     UserId u = 5; //still CE
7     UserId u2 = 5_uid;
8 }

```

## 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 переменной 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 наследовании 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 неvirtуальном наследовании).

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

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

```
1 class Mother {
2 public:
3     int a = 1;
4 };
5 class Father {
6 public:
7     int a = 2;
8 };
9 class Son: public Mother, public Father {
10     int s = 3;
11 };
12 int main() {
13     Son s;
14     cout << s.a;
15     //CE: request for member "a" is ambiguous
16 }
```

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

```
1 struct Granny {
2     int g = 0;
3 };
4 struct Mother: public Granny {
5     int m = 1;
6 };
7 struct Father: public Granny {
8     int f = 1;
9 };
10 struct Son: public Mother, public Father {
11     int s = 3;
12 };
```

Еще один пример:

```
1 struct A {
2     int a;
3     int f() {};
4 };
5
6 struct B1 : A {};
7 struct B2 : A {};
8 struct C : B1, B2 {};
9
10 int main()
```

```

11 {
12     C c;
13     c.a; //CE
14     c.f(); //тоже нельзя
15     c.B1::f(); // можно
16 }

```

Проблема снова заключается в том, что С унаследован от В1 и В2, унаследованных от А (в памяти лежит приблизительно так: [А][В1][А][В2][С]), и получается как бы два А. То есть в выражениях (1) и (2) неизвестно, к каким именно а и f мы обращаемся, которые от того А, что от В1, или от того А, что от В2. А 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 В унаследован от А, и С унаследован от А и В, то вы опять же не сможете обращаться к полям А через объект класса С. В таких случаях возникает “warning: inaccessible base class”.

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

Про **using**:

```

1 struct B {
2     B(int = 13, int = 42);
3 };
4 struct D : B {
5     using B::B;
6 };

```

Еще примеры.

```

1 struct B1 { B1(int, ...) { } };
2 struct B2 { B2(double) { } };
3
4 int get();
5
6 struct D1 : B1 {
7     using B1::B1; // inherits B1(int, ...)
8     int x;
9     int y = get();
10 };
11
12 void test() {
13     D1 d(2, 3, 4); // OK: B1 is initialized by calling B1(2, 3, 4),
14                     // then d.x is default-initialized (no initialization is performed),
15                     // then d.y is initialized by calling get()
16     D1 e;          // Error: D1 has no default constructor
17 }
18
19 struct D2 : B2 {
20     using B2::B2; // inherits B2(double)
21     B1 b;
22 };

```



```
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)
```