

### Билет 3.13. Понятие перегрузки функций. Что является и что не является частью сигнатуры функции. Основные правила выбора наиболее подходящей функции: частное лучше общего, точное соответствие лучше приведения типа. Пример неоднозначности при обращении к функции.

**Def.** Перегрузка функций – возможность создавать несколько одноименных функций, но с различными параметрами.

**Def.** Сигнатура функции – это часть общего объявления функции, которая позволяет идентифицировать эту функцию среди других. Функция распознаётся компилятором по последовательности типов её аргументов и её имени, что и составляет сигнатуру или сигнату функции. И если функция — это метод некоторого класса, то в Signature участвует и имя класса.

В сигнатуре учитываются только типы параметров, а не типы возвращаемых значений. Перегруженные функции не могут различаться лишь типами возвращаемого значения; если списки параметров двух функций разнятся только подразумеваемыми по умолчанию значениями аргументов, то второе объявление считается повторным.

```
// объявления одной и той же функции
int max ( int *ia , int sz );
int max ( int *ia , int = 10 );
```

Ключевое слово `typedef` создает альтернативное имя для существующего типа данных, новый тип при этом не создается. Поэтому если списки параметров двух функций различаются только тем, что в одном используется `typedef`, а в другом тип, для которого `typedef` служит псевдонимом, такие списки считаются одинаковыми.

Следующие два объявления также считаются одинаковыми:

```
// объявляют одну и ту же функцию
void f( int );
void f( const int );
```

Спецификатор `const` важен только внутри определения функции: он показывает, что в теле функции запрещено изменять значение параметра. Однако аргумент, передаваемый по значению, можно использовать в теле функции как обычную инициализированную переменную: вне функции изменения не видны. Добавление спецификатора `const` к параметру, передаваемому по значению, не влияет на его интерпретацию. Функции, объявленной как `f(int)`, может быть передано любое значение типа `int`, равно как и функции `f(const int)`.

Однако, если спецификатор `const` или `volatile` применяется к параметру указательного или ссылочного типа, то при сравнении объявлений он учитывается.

```
// объявляются разные функции
void f( int* );
void f( const int* );

// и здесь объявляются разные функции
void f( int& );
```

```
void f( const int& );
```

Все перегруженные функции объявляются в одной и той же области видимости. К примеру, локально объявленная функция не перегружает, а просто скрывает глобальную.

**Def.** Разрешением перегрузки функции называется процесс выбора той функции из множества перегруженных, которую следует вызвать. Этот процесс основывается на указанных при вызове аргументах.

При разрешении перегрузки функции выполняются следующие шаги:

1. Выделяется множество перегруженных функций для данного вызова, а также свойства списка аргументов, переданных функции.
2. Выбираются те из перегруженных функций, которые могут быть вызваны с данными аргументами, с учетом их количества и типов.
3. Находится функция, которая лучше всего соответствует вызову.

Как правило при исполнении, из нескольких объявлений выбирается та функция, которая наиболее подходит. Проблема: а как решить, что подходит лучше? Правил много, рассмотрим основные принципы:

1. Ищем точное соответствие (exact matching)
2. Promotion от «меньшего» типа к «большему» (пример: short в int)
3. Конвертация из одного типа в другой (пример: int в bool).
4. Определённые пользователем преобразования типа. (Например, пользователь написал конструктор вида `String(int[])`, в этом случае при необходимости передаваемый в некоторую функцию аргумент типа `int[]` может быть преобразован к типу `String`)
5. ellipsis conversion (Функция `f(...)` имеет самый низкий приоритет, потому что это максимально общий случай. Точно так же функция `f(int a, int b)` лучше, чем `f(int a, ...)`).

**Общий принцип:** от частного к общему. Иногда приходится делать цепочки преобразований.

**Замечание:** иногда возникает неопределённость, которая может привести к СЕ. Пример:

```
1 void f(int) {
2     cout << 1;
3 }
4
5 void f(float) {
6     cout << 2;
7 }
8
9 int main() {
10     f(0.0); \\double
11 }
```

И в том, и в другом случае необходима одна конверсия. Компилятор не знает, какую выбрать.

## Билет 3.14. Понятие класса, полей и методов класса, модификаторы доступа `public` и `private`, отличие класса от структуры. Применение операторов точка и стрелочка. Применение двойного двоеточия. Применение ключевого слова `this`. Константные и неконстантные методы.

Существует 2 способа создать свой тип.

- Создать свой класс.

```
1 class C {  
2 };  
3  
4 int main() {  
5     C c;  
6 }
```

Пустой класс (как на примере) занимает 1 байт в памяти, так как по стандарту C++ никакой объект не может занимать 0 байт в памяти (иначе могло бы так получиться, что какие-то два объекта имеют одинаковый адрес в памяти)

- Создать свою структуру

```
1 struct C {  
2 };  
3  
4 int main() {  
5     C c;  
6 }
```

Структуру обычно используют когда не требуется внутренняя логика, нам нужно просто объединить какие-то переменные, если же появляются какие-то методы обработки - используют `class`.

У классов и структур есть свои поля и методы.

**Def.** Поля - данные которые хранятся в объекте этого типа, иначе - переменные, объявленные в теле класса.

**Def.** Методы - операции, которые над ним можно выполнять. Методы можно перегружать, как и обычные функции. Методы можно определить вне класса, если они были объявлены внутри, однако определить метод одного класса внутри другого класса нельзя - у них разные пространства.

Объекты классов бывают константными и неконстантными. Константные объекты класса могут явно вызывать только константные методы класса,

**Def.** Константный метод — это метод, который гарантирует, что не будет изменять объект или вызывать неконстантные методы класса (поскольку они могут изменить объект) - это делается с помощью квалификатора `const`. Из константного метода нельзя вызвать неконстантный.

```

1 class C {
2 private:
3     int s = 0;
4     std::string str;
5     double d = 0.0;
6 public:
7     void add_and_print(int a);
8     void add_and_print(double a) {
9         std::cout << d + a;
10    }
11 };
12 void C::add_and_print(int s) {
13     std::cout << C::s + s;
14     //std::cout << this->s + s;
15 }
16 int main() {
17     C c;
18     c.add_and_print(1);
19 }

```

Размер такого класса - сумма размеров всех полей. В целях увеличения производительности иногда к объектам добавляется padding - если в структуре сумма размеров объектов получается больше 8 байт и при этом это число не кратно 8, то компилятор дополняет до числа кратного 8.

## Инкапсуляция - первый принцип ООП

Формально: Инкапсуляция - совместное хранение полей и методов (но ограниченный доступ к ним извне).

Неформально: Объявление рядом данных и методов обработки этих данных + ограничение доступа к самим данным. Мы разрешаем доступ извне к данным только разрешенным способам их обработки, т.е. пользователь имеет доступ к ограниченному числу методов/полей.

1. private - к этим полям/методам нельзя получить доступ извне.
2. public - к этим полям/методам можно получить доступ извне.

Из main() не получится обратиться к private-полю (оно приватно в этом контексте). В классе по умолчанию все поля - private, в структуре же все поля будут public.

## Дружественные методы и классы

Иногда нужно все-таки обратиться к private-полям класса не из членов класса, для этого используется ключевое слово **friend**. Нужно внутри класса объявить функцию с этим ключевым словом. В дальнейшем если где-то в коде встретится функция с точно такой же сигнатурой и она не будет членом нашего класса, ей будет разрешен доступ к приватным полям. Если не объявляли что функция может быть методом какого-то класса, то функция с такой же сигнатурой, то в методе какого-то класса не получит доступ к полям. Другом можно объявить не только метод какого-либо класса, но и весь класс - тогда все его методы будут считаться дружественными. Дружба не взаимна и не транзитивна. friend надо использовать в исключительных случаях.

```

1 class A{
2     int s;
3     void f(int);
4 };
5 class B{
6     int t;
7 }
8 class String{
9     char* str = nullptr;
10    size_t size = 0;
11
12    friend void f(int);
13    friend void A::g(int);
14    friend class B;
15
16 };

```

Пусть мы хотим гарантировать что никто не вызовет функцию от `int`. Перегрузка выполняется до проверки доступа (найдется точное соответствие), поэтому данный код вызовет ошибку компиляции:

```

1 class C {
2 private:
3     int s = 0;
4     std::string str;
5     double d = 0.0;
6     void add_and_print(int a) {
7         std::cout << s + a;
8     }
9 public:
10    void add_and_print(double a) {
11        std::cout << d + a;
12    }
13 };
14
15 int main() {
16     C c;
17     c.add_and_print(1);
18 }
19

```

#### Указатель `this`:

Указатель на текущий объект. В контексте метода класса означает указатель на тот объект, в котором мы сейчас находимся, тот от которого вызван этот метод. `This` является скрытым первым параметром любого метода класса (кроме статических методов), а типом указателя выступает имя класса. Явно объявить, инициализировать либо изменить указатель `this` нельзя.

#### Операторы «.» и «→»:

Оператор «.» – обращение к полю или методу.

Оператор «→» – то же, что и (\*p). Обращение по разыменованному указателю объекта к его полям и методам.

## Константные и неконстантные методы.

Перегружать функции можно не только по типу параметров функции (по правому операнду), но и по квалификаторам метода (по левому операнду - тому что стоит до точки) - константный/ неконстантный метод. Конструктор, деструктор и не методы класса нельзя помечать как константные/неконстантные.

**Напоминание:** преобразование неконстантного объекта в константный разрешен, и стоит дешево, а в обратную сторону запрещен.

Следовательно, можно не писать отдельную перегрузку для неконстантного метода, если уже написан константный и они имеют одинаковую логику - при вызове такого метода от неконстантного объекта произойдет неявное преобразование.

Определение метода как константного является частью объявления. Все поля в теле этого метода считаются теперь константными (в т.ч. указатель `this`) - значит, нельзя применять неконстантные операции к полям или вызывать другие неконстантные методы из себя.

**Правило:** ставить `const` везде, где метод пригоден для константных объектов.

Кроме того, может понадобится изменить какие-то поля константного объекта - например если нужно подсчитать сколько раз обратились к методу или для кэширования. Для этого используется ключевое слово **mutable**, которое можно использовать только для полей класса. `Mutable` это своеобразный `anticonst`, поле можно будет поменять даже если находимся в константном объекте. Если в полях есть ссылка, тогда даже если метод константный, это поле не будет константным.

## Билет 3.15. Понятие конструкторов, деструкторов, пример их правильного определения для класса `String`. В каких контекстах использования от класса требуется наличие конструктора по умолчанию, в каких - конструктора копирования?

**Def.** Конструктор - метод, у которого нет возвращаемого значения, который описывает как создать объект класса с заданными параметрами. Как и любой другой метод, его можно определять вне класса. Компилятор может сам создать конструктор по умолчанию, однако в нем будут инициализированы все поля, что плохо в тех случаях, когда среди полей есть указатели.

С C++11 можно делегировать один конструктор другому - сначала выполнится один конструктор, затем другой

```
1  class String{
2      String(...): String(...) {
3          //smth that is need to be done only by second constructor//
4      }
5  };
```

**Правило:** если в классе определен хотя бы один конструктор, то определение по умолчанию уже не работает.

Если у конструктора есть один или несколько параметров по умолчанию — это по-прежнему конструктор по умолчанию. Каждый класс может иметь не более одного кон-

структора по умолчанию: либо без параметров, либо с параметрами, имеющими значения по умолчанию. Например, такой: `MyClass (int i = 0, std::string s = "")`;

В C++ конструкторы по умолчанию имеют существенное значение, поскольку они автоматически вызываются при определенных обстоятельствах, и, следовательно, при определенных условиях класс обязан иметь конструктор по умолчанию, иначе возникнет ошибка:

- Когда объект объявляется без аргументов (например, `MyClass x;`) или создается новый экземпляр в памяти (например, `new MyClass;` или `new MyClass();`).
- Когда объявлен массив объектов, например, `MyClass x[10];` или объявлен динамически, например `new MyClass [10]`. Конструктор по умолчанию инициализирует все элементы.
- Когда в классе потомке не указан явно конструктор класса родителя в списке инициализации.
- Когда конструктор класса не вызывает явно конструктор хотя бы одного из своих полей-объектов в списке инициализации.
- В стандартной библиотеке определенные контейнеры заполняют свои значения используя конструкторы по умолчанию, если значение не указано явно. Например, `vector<MyClass>(10);` заполняет вектор десятью элементами, инициализированными конструктором по умолчанию.

**Def.** Конструктором копирования (англ. copy constructor) называется специальный конструктор, применяемый для создания нового объекта как копии уже существующего. Такой конструктор принимает как минимум один аргумент: ссылку на копируемый объект.

Обычно компилятор автоматически создает конструктор копирования для каждого класса (известные как неявные конструкторы копирования, то есть конструкторы копирования, заданные неявным образом), но в некоторых случаях программист создает конструктор копирования, называемый в таком случае явным конструктором копирования (или «конструктором копирования, заданным явным образом»). В подобных случаях компилятор не создает неявные конструкторы.

Конструктор копирования по умолчанию просто копирует все поля (**shallow copy**) - в т.ч. и указатели, а значит может возникнуть UB - указатели просто перекопировались, но указывают на одну область памяти. Чтобы запретить копирование, можно либо сделать конструктор приватным, либо использовать **delete**.

```
String(const String& s);
```

Если в классе есть поля которые запрещают себя копировать, то дефолтный конструктор копирования не сможет сгенерироваться - выдаст СЕ.

Существует четыре случая вызова конструктора копирования:

- Когда объект является возвращаемым значением
- Когда объект передается (функции) по значению в качестве аргумента

- Когда объект конструируется на основе другого объекта (того же класса)
- Когда компилятор генерирует временный объект (как в первом и втором случаях выше; как явное преобразование и т. д.)

Конструктор нельзя вызывать как метод от другого конструктора - будет вызвано не от нашей строки, а от копии, которая затем удалится.

**Def.** Деструктор - метод, который вызывается непосредственно перед тем, как объект будет уничтожен. Как и любой другой метод, его можно определять вне класса.

У деструктора нет параметров, его нельзя перегрузить. Деструктор вызывается когда объект выходит из области видимости. Если в конструкторе не было никаких нетривиальных действий, например выделение памяти или закрытия потоков, то деструктор можно оставить пустым, обнулять какие-то переменные или указатели в деструкторе не нужно, так как после вызова деструктора компилятор снимет эти переменные со стека.

Реализация нескольких конструкторов и деструктора:

```

1 class String{
2 private:
3     char* str = nullptr;
4     size_t sz = 0;
5     size_t cpct = 1; \\capacity
6 public:
7     Конструкторы\\
8     String() {}
9     String(size_t size_, char s = '\\0');
10    String(const String& other_str);
11    Деструктор\\
12    String::~~String(){
13        delete [] str;
14    }
15 };
16
17 String::String(size_t size_, char symbol = '\\0') : size(size_),
18             cpct(size * 2 + 1), str(new char[cpct]){
19     memset(str, symbol, size_);
20 }
21
22 String::String(const String& other_str) : size(other_str.size),
23             cpct(size * 2 + 1), str(new char[cpct]){
24     memcpy(str, other_str.str, size);
25 }

```

**Билет 3.16. Понятие перегрузки операторов, синтаксис перегрузки оператора присваивания. В каких контекстах использования от класса требуется наличие оператора присваивания?**

**Правило трех:** Если в классе потребовалось реализовать нетривиальный деструктор, или нетривиальный конструктор копирования или нетривиальный оператор присваивания,



ния, то в классе нужно реализовать все три.

Нередко трансформируется в правило пяти:

**Правило пяти:** Если в классе потребовалось реализовать одного из этих пяти пунктов, стоит реализовывать все.

- Деструктор
- Конструктор копирования
- Оператор присваивания копированием
- Конструктор перемещения
- Оператор присваивания перемещением

Дефолтный оператор присваивания тоже работает по принципу shallow copy.

Если в классе есть поля, которые не допускают присваивание (например ссылки), то генерация дефолтного оператора присваивания не определена и будет СЕ (но (!) дефолтный конструктор копирования будет работать).

Операция присваивания копированием отличается от конструктора копирования тем, что должна очищать члены-данные цели присваивания (и правильно обрабатывать самоприсваивание), тогда как конструктор копирования присваивает значения неинициализированным членам-данным.

Про присваивание перемещением:

Определение конструктора перемещения и оператора присваивания перемещением выполняется аналогично определению конструктора копирования и оператора присваивания копированием. Однако, в то время как функции с копированием принимают в качестве параметра константную ссылку l-value, функции с перемещением принимают в качестве параметра неконстантную ссылку r-value.

Теперь всё просто! Вместо выполнения глубокого копирования исходного объекта в неявный объект, мы просто перемещаем (воруем) ресурсы исходного объекта. Под этим подразумевается поверхностное копирование указателя на исходный объект в неявный (временный) объект, а затем присваивание исходному указателю значения null (точнее nullptr) и в конце удаление неявного объекта.

Вот пример реализации оператора присваивания для String.

```
1 String& String::operator =(String other_str){
2     if (this == &s) return *this; //self-assignment
3     swap(other_str);
4     return *this;
5 }
```

Нельзя создавать новые операторы путем перегрузки, можно лишь перегружать существующие, и то не все. Путем перегрузки нельзя поменять приоритет операторов.

### Замечание про оператор присаивания.

Если определять оператор `+` внутри класса, то будет вот такая проблема:

Следующий вызов

```
1 Complex c(2.0);  
2 1.0 + c;
```

выдаст ошибку, так как мы определили оператор «`+`» только тогда, когда левым операндом является объект класса (`*this`).

При определении оператора вне функции и левый, и правый операнд будут равноправны, и компилятор сможет делать каст как левого, так и правого операнда - соответственно в оператор надо передавать два параметра. Тогда корректный код выглядит так:

```
1 struct Complex{  
2     double re = 0.0;  
3     double im = 0.0;  
4  
5     Complex (const Complex&){}  
6  
7     Complex& operator +=(const Complex& z) {  
8         re += z.re;  
9         im += z.im;  
10        return *this;  
11    }  
12 }  
13  
14 Complex operator +(const Complex& a, const Complex& b) {  
15     Complex copy = a; //Copy constructor definitely called  
16     return copy += b;  
17     //copy += b;  
18     //return copy;  
19 }
```

## Билет 3.17. Условия генерации компилятором конструкторов и оператора присваивания. Использование слов `default` и `delete` для их запрета или принудительной генерации компилятором.

Ключевое слово **default** введено в C++ 11. Его использование указывает компилятору самостоятельно генерировать соответствующую функцию класса, если таковая не объявлена в классе. Это слово может быть использовано только с конструкторами (по умолчанию, копирования, перемещения), с операторами присваивания, с деструктором. То есть с теми методами, которые компилятор вообще умеет генерировать.

С C++11 можно определять конструктор по умолчанию следующим образом (если поля проинициализированы и среди полей нет ссылок):

```
1 String() = default;
```

Ключевое слово **delete** используется для того, чтобы запретить приведение типов или генерацию каких-либо методов. Собственно, это неплохо работает, поскольку сначала производится перегрузка, а потом проверка доступа.

С C++11 можно запретить какой-либо конструктор - т.е. нельзя будет вызвать конструктор с некоторыми параметрами:

```
1  class String{  
2      String(int n, char c) = delete;  
3  };
```

Оператор присваивания по умолчанию почленно присваивает поля. Генерируется только от `const type&`.

Конструктор копирования по умолчанию просто копирует все поля, не может быть сгенерирован, если какое-то из полей не допускает копирования.

Генерация:

- Конструктор по умолчанию генерируется автоматически, если нет объявленного пользователем конструктора.
- Конструктор копирования генерируется автоматически, если нет объявленного пользователем конструктора перемещения или оператора присваивания перемещением (поскольку в C++03 нет конструкторов перемещения или операторов присваивания перемещением, это упрощается до "always" в C++03).
- Оператор присваивания копированием генерируется автоматически, если нет объявленного пользователем конструктора перемещения или оператора присваивания перемещением.
- Деструктор генерируется автоматически, если нет объявленного пользователем деструктора.

**C++11 и позже:**

- Конструктор перемещения генерируется автоматически, если нет объявленного пользователем конструктора копирования, оператора присваивания копированием или деструктора, а также если сгенерированный конструктор перемещения действителен.
- Оператор присваивания перемещением генерируется автоматически, если нет объявленного пользователем конструктора копирования, оператора присваивания копированием или деструктора, и если сгенерированный оператор присваивания перемещением действителен (например, если ему не нужно присваивать постоянные члены).