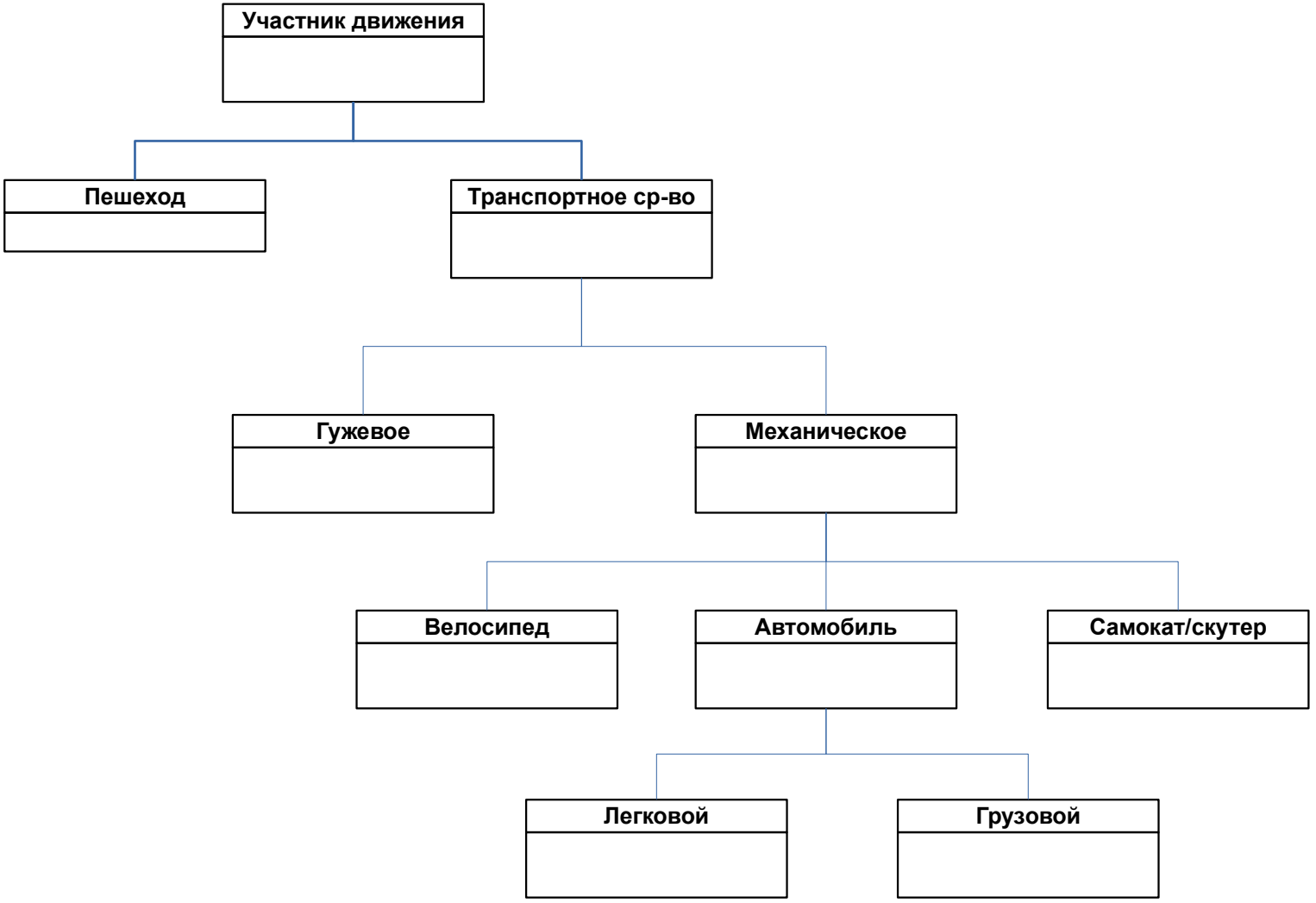
Введение в C++.
Наследование и шаблоны.

СПбГЭТУ «ЛЭТИ»
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

Иерархии классов (объектов)

Объекты реального мира образуют иерархии. В пределах иерархии существуют общие свойства, по для разных типов объектов есть различные свойства и функции (методы)



Иерархии объектов (классов)

При описании структур данных для разных ветвей иерархий существуют общие поля и методы и различающиеся поля и методы.

Для языка С приходится описывать структурные типы (шаблоны) полностью, в при операциях с ними не забывать менять типы указателей (источник ошибок).

Языки с ООП позволяют избежать таких ошибок, используя механизм ***наследования***.

Можно создавать новые структурные типы добавляя новые поля к уже известным структурным типам.

Особенности наследования

Механизм наследования позволяет реализовать переход от общего к частному.

Пример:

```
struct person {
    char name[64];
    char gender; //'m' or 'f'
    int year_of_birth;
};

struct student : person {
    char spec[16];
    int year;
    char group[8];
    float average;
};
```

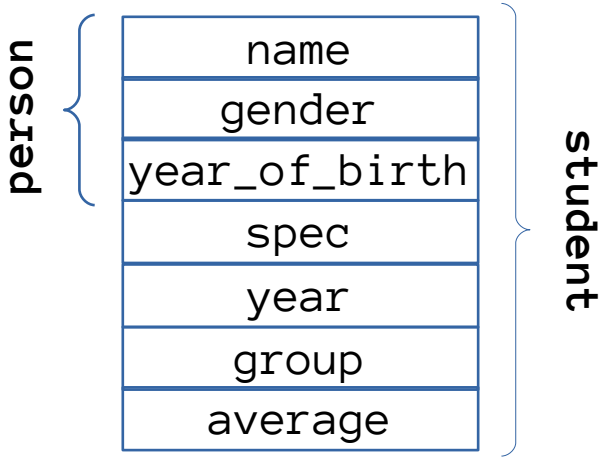
Особенности наследования

В структуре **student** добавляются уточняющие сведения (сущность «студент» — частный случай сущности «человек»).

Сущность «студент» имеет все свойства сущности «человек» + добавочные свойства.

Структура **person** — базовая, родительская («объект-предок»)

Структура **student** — унаследованная, порождённая («объект-потомок»).



Первые три поля в структуре типа **student** располагается с теми же смещениями от начала и имеют такие же размеры, что и в структуре типа **person**.

Указатели на структуры типа **student** могут быть неявно приведены к указателям на структуры типа **person** (но не наоборот!!).

Пример: **U19.cpp**

C++ разрешает неявное преобразование адреса переменной порожденного типа в адрес переменной родительского типа (как для указателей, так и для ссылок).

Защита при наследовании

По умолчанию для структур все поля являются открытыми (**`public`**), для классов — закрытыми (**`private`**).

Наследование возможно как для структур, так и для классов.

Классы можно наследовать от структур (и наоборот).

При построении класса-потомка на основе базового класса (предка) различают открытое (**`public`**) и закрытое (**`private`**) наследование.

```
Class B : public A {
// поля и методы
}

Class C : private A {
// поля и методы
}
```

Защита при наследовании

В первом случае все открытые поля и методы класса `A` будут доступны для объектом класса `B` отовсюду (могут быть вызваны в любом месте модуля)

Во втором случае все открытые поля и методы класса `A` будут доступны только из методов класса `C` (при описании класса `C`).

Если какие-то поля и методы класса должны быть доступны ***только*** для его потомков, то при описании класса может использоваться режим **`protected`**.

Поля и методы в секции **`protected`** будут доступны в самом базовом классе и в его потомках (при их описании).

Конструкторы и деструкторы при наследовании

При создании объекта «порожденного» класса создается также и объект «базового» класса (*должен отработать конструктор «базового» класса*).
При уничтожении объекта «порожденного» класса должен быть уничтожен также и объект «базового» класса (*должен отработать деструктор «базового» класса*).
При этом в теле конструктора и деструктора объекта-потомка должны быть доступны все части объекта, для которого они вызываются.
Т.е. конструктор «предка» должен отработать раньше, а деструктор «предка» - позже, чем конструктор и деструктор «потомка».

Компилятор автоматически вставляет в код деструктора «потомка» вызов деструктора для «предка», причем сначала работает деструктор для «потомка», потом — для «предка».

С конструкторами есть особенности в случае, если конструктор требует инициализации значений полей.

Конструкторы и деструкторы при наследовании

Пусть `A` — базовый класс, конструктору которого требуется два параметра типа `int`, `B` — класс, унаследованный от `A`, имеющий конструктор по умолчанию и имеющий поле `i` типа `int`.

```
Class A {
// ...
public:
   A(int p, int q) { /* ... */ }
   // ...
};

Class B : public A {
   int i;
public:
   B();
   // ...
};
```

Тогда конструктор для B реализуется как

```
B::B() : A(2,3), i(4) {/* ... */}
```

Назначение шаблонов

Если требуется использовать одинаковые (или почти одинаковые) фрагменты кода, то есть несколько путей.

1. «Лобовой» - так и писать (почти) повторяющиеся фрагменты кода. Плохой вариант, т.к. нужно при любом изменении отслеживать все фрагменты.

2. Сделать функции. Подходит, если сохраняется набор параметров (количество и типы, с поправкой на наличие вариадических функций в Си).

3. Если различаются типы параметров и результатов, то в Си можно написать макрос. Это сложно и трудно выявлять ошибки (*компилятор показывает ошибки не в коде макроса, а в коде функции, при обращении к макросу*). *Использование макропроцессора в Си небезопасно, его следует минимизировать.*

4. В C++ есть механизм шаблонов для функций и классов. ***Шаблон — это «заготовка» кода, которая при конкретизации некоторых параметров превращается в код функции или класса*** (при разных значениях параметров можно получить разные функции или классы).

Создание шаблонов

Пусть имеется функция сортировки массива целых чисел (пример **`U20.cpp`**)

Если требуется сделать функцию сортировки массива вещественных чисел, то она будет отличаться только типом параметра с адресом массива и типом локальной переменной **`tmp`**.

То же самое — с функцией генерации массива. Отличаются типы указателей и элементов массива.

Шаблоны для функций и их использование — примеры **`U21.cpp`**, **`U22.cpp`**.

Для функции вывода шаблон создать можно, но вывод будет некорректным (форматы разные). ***Как учесть разные типы чисел?***

Для классов шаблоны создаются аналогично. Пример — шаблон класса массива (пример **`U23.cpp`**).

В методах и полях можно использовать шаблонный тип данных.

Полиморфизм

Полиморфи́зм — возможность объектов с одинаковой спецификацией иметь различную реализацию.
Полиморфизм состоит в том, что операция или действие, обозначаемое одним и тем же символом, корректно выполняется для операндов или параметров различных типов.

Статический полиморфизм реализуется при переопределении операций и перегрузке функций.

Полиморфизм ***адресов*** реализуется при наследовании.

Параметрический полиморфизм реализуется при создании и использовании шаблонов.

Что не рассматривалось

- Исключения
- Виртуальные функции и динамический полиморфизм
- Дружественные функции и классы
- Статические поля и методы
- Подробности про переопределение операций