

## Lec 13

### Override - точно говорит что мы переопределяем метод

```
struct B : public A
{
    int f() override;
}
```

### Глобальный const (constexpr) - является статиком автоматически

```
const int x = 3; // static

int main()
{
}
```

Если использовать x в другом файле, то в C++ это не скомпилируется т.к. не сможет линковаться из-за того, что x - static

### Решение: extern

```
extern const int x = 3; // NOT static

int main()
{
}
```

# Приколы с auto:

auto - отбрасывает const и "указательность" при определении типа

```
const int x1 = 3;
const int &x2 = z;

int main()
{
    auto y = x1; // type(y) = int
    auto z = x2 // type(y) = int
    auto *w = &x2; // type(w) = указатель на тип, который напишет au

    auto с сохранением const, указательности и т.д.:
    decltype(auto) v = x; // type(v) = const int
}
```

## Функция auto

Использование того типа, который возвращается в return

```
auto f()
{
    return x;
}
```



```
decltype(auto) f()
{
    return x;
}
```

Можно и явно сказать чтобы возвращалась ссылка:

```
auto &f()
{
    return x;
}
```

## Лямбда функции (не функции)

- На самом деле лямбда функция - класс, который имеет уникальное неизвестное нам имя и именно по этой причине мы должны использовать `auto`.
- В нём перегруженный оператор `()`
- `()` - аргументы которые принимает функция (работает также как и в обычной функции)
- `[]` - **список переменных**, которые лямбда функция будет видеть. Это могут быть те переменные, которые видны в момент описания лямбда функции.
- Все копии в `[]` - `const` (кроме случая `mutable`)

```
void f()
{
    int x = 3;
    auto z = [] () {return 4;};
    int y = z();

    auto z = [](int a) {return a+4;};
    int y1 = z(0);
    int y2 = z(z);
}
```

## `[]` - захваты

```
void f()
{
    1.
    int x = 3;
    auto z = [x](int a) {return a+x;}; // значение x СКОПИРУЕТСЯ
    x = 0
    int y1 = z(0); // y1 == 3
}
```

2.

```
int x = 3;
auto z = [&x](int a) {return a+x;}; // будет браться ССЫЛКА на x
x = 0
int y1 = z(0); // y1 == 0
```

3. Как мы по дефолту будем захватывать

```
int x = 3;
int y = 2;
int z = 1;
int w = 500
auto z = [&, x, y, z, =w](int a) {return a+x+w+y;}; // будет бра
x = 0
int y1 = z(0); // y1 == 0
```

4. Специальное поведение **this**

[**this**] - копирование ссылки на класс  
 [**\*this**] - создаются копии всего из класса

5. Изменение переменных

```
int x = 3;
auto z = [x](int a) {x++;}; // ОШИБКА, все копии - CONST
auto z = [x](int a) mutable{x++;}; // Работает
auto z = [&x](int a) {x++;}; // Работает
```

Виктория переставила стул.

}

Struct S

```
{
    mutable int a; // способен меняться под const'ом
    int b;
    S(int x, int y) : a(x), b(y) {}
}
```

```
const S z(1, 2); // const - ничего не меняется,
// КРОМЕ: объектов с mutable
z.a = 3; // Работает (mutable)
```

## Возвращаемое значение лямбда функции

Явно указываем тип

```
int x = 3;
auto z = [x] (int a = 2) -> long long {return a + x}
x = 0;
```

Можно также и с обычными функциями (не имеет смысла)

```
auto f() -> int
{
}
}
```

Переименование захваченной функции в рамках лямбда функции

```
int x = 3;
auto z = [i = x] (int a = 2) -> long long {return a + x}
[i = x] - мы захватываем по значению x, но в рамках лямбда функц
```

## Аргументы

1. НЕ СКОМПИЛИТСЯ

```
const S x;
void f(S &a)
{
}
```

```
S g()
{
    ...
}
```

```
S x;
f(x = g());
```

2. Скомпилируется

```
const S x;
void f(const S &a) // Подключем константную ссылку к значению
{
}
```

```
S g()
{
    ...
}
```


```
S x;
f(g());
```

# Категории значений

Типы в C:

- **lvalue** (locator value)

- **rvalue** (read value)

**lvalue** — это выражение, обозначающее **объект** в памяти, у которого есть **адрес** и **идентичность**. Проще говоря — «то, что можно поместить слева от ».

Типы в C++: **Const & подключается ко всему**

- **glvalue** — выражение, которое имеет определённый адрес в памяти
  - **lvalue** — «локаторное» значение: именованные объекты (переменные, элементы массивов и т.п.), можно брать адрес и присваивать **Сюда подключаются &**
  - **xvalue** — «истекающее» glvalue: временный объект с доступным адресом, чьи ресурсы можно переместить (std::move)
- **rvalue** — временное значение без постоянного адреса **Сюда подключаются &&**
  - **xvalue** — «истекающее» rvalue: временный объект, ресурсы которого можно перенести
  - **prvalue** — «чистое» rvalue: литералы и результаты вычислений, из которых нельзя перемещать ресурсы

lvalue - существует и не собирается умирать

xvalue - существует и скоро умрёт

prvalue - конкретные константы и значения по типу 1, 2, "a"...

Если возможно подключиться к конструктору **копирования** или **конструктору перемещения** - автоматом будет подключение к **конструктору перемещения**

## Пример xvalue

```

S g()
{
    S a;
    return a; Когда функция закончит своё действие, то вернёт a и по
}

```

Хавают:

```

lvalue: <type> &obj
lvalue, xvalue, prvalue: const <type> &obj
rvalue: <type> &&obj
в функции: prvalue --> xvalue.

```

## Конструктор перемещения

- **Копирование**

Создаёт **новые** копии всех ресурсов (буферы, указатели) из **other** в **this**.  
После вызова у вас будут два **независимых** экземпляра с одинаковыми данными.

- **Перемещение**

«**Перебрасывает**» (steal) ресурсы из **other** в **this** без аллокаций/копирований.

В результате **this** получает указатели **other**, а **other** остаётся в валидном, но «опустошённом» состоянии (например, с **nullptr**).

```

const char *p;
S();
S (const S & a)
{
    char x= new char[st...];
    strcpy(x, a.p);
    p = x;
}

S (S &&a)
{
    p = a.p;
}

```

```
a.p = NULL; // Если не обнулить, то деструктор a - уничтожит зн
}
```

## Оператор перемещения

Нужно для того, чтобы значения умирающего не СКОПИРОВАТЬ, а именно переместить (видимо чтобы потом не писать `a.p = NULL` в конце)

```
S &operator=(S &&x);
```

## Объявление xvalue переменных

`move` == `cast`

при помощи `std::move` мы понимаем что полезная часть жизни `b` - кончилась и дальше нам `b` - не понадобится => он является xvalue

```
S a, b, c;
a = std::move(b); // Вызовется не оператор копирования, а оператор п
```

## Пример вызова конструктор копирования

```
S(S && a) : z(a)
{
    p = a.p;
    a.p = NULL;
}
```

## Пример вызова конструктор перемещения

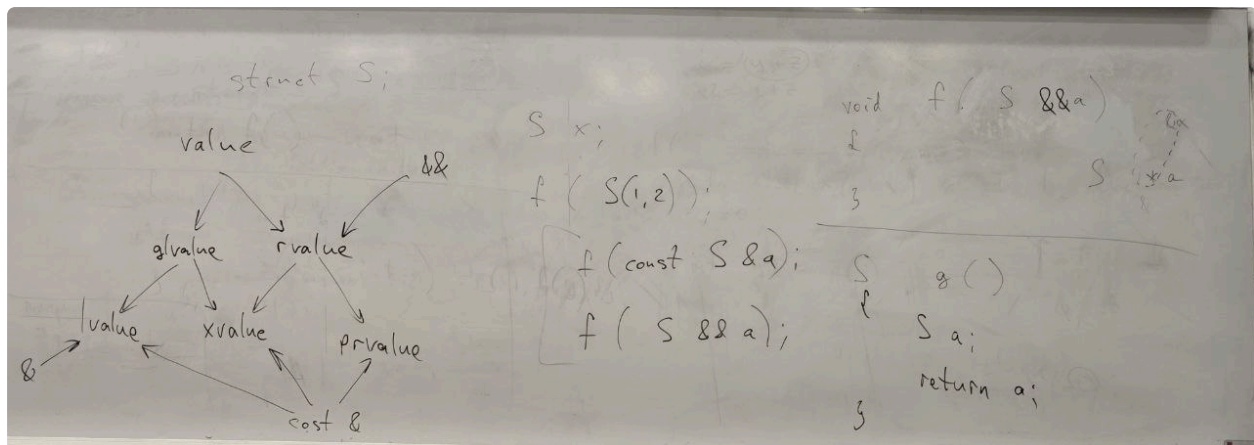
```
S(S &&a) : z(std::move(a))
{
```



```

p = a.p;
a.p = NULL;
}

```



## Шаблоны (такого нет в С) - правило автогенерации классов и функции

- Шаблон в C++ не создаёт функцию во время объявления, он создаёт их при вызове. По сути на шаблоне строится функция
- Шаблоны лучше создавать в .h файле
- Шаблоны лучше всего писать до конца, а не просто объявлять шаблон и не писать как он работает. (Всё же это может понадобится, если реализация шаблона находится в другом .h файле, в таком случае мы делаем ручное **инстанцирование** )
- 

Отличие от дженериков:

Механизм реализации	Type erasure: вся информация о типе стирается на этапе компиляции, генерируется один байт-код	Code generation: для каждого набора параметров генерируется отдельный код
Проверка типов	На этапе компиляции, но с приведениями в байт-коде (unchecked casts)	Строгая компиляция: генерируется и проверяется каждый экземпляр шаблона

Посмотрим на abs:

## КОПИПАСТА

```
int abs(int x)
{
    return x < 0? -x : x;
}

long abs(long x)
{
    return x < 0? -x : x;
}
```

## КАК НАДО:

```
template <typename T> T abs(T x)
{
    return x < 0? -x : x;
}

int x;
x = abs(x); // Только в этот момент по шаблону создастся функция с T

S y;
y = abs(y); // сработает, если определены: оператор сравнения, унарн

template <typename T, typename X> T abs(X x)
{
    return x < 0? -x : x;
}

long x;
x = abs<int, long> (x); // abs будет работать от лонга
float f;
x = abs(f); // Не скомпилился

template <typename T>
T max(T a, T b)
{
    return a > b ? a : b;
}

int x;
x = max<int>(3, 5); //Скомпилился

long x;
x = max(3, x); // НЕ скомпилился
```

```
x = <long>max(3, x); // Скомпилируется

template <typename T>
T1 max(T1 a, T2 b)
{
    return a > b ? a : b;
}
long x;
x = max(3, x); // скомпилируется, НО вернёт int
```

## Аргумент шаблона - кроме typedef ещё и целые типы + ещё что-то

```
template <typename T1, int i> //int i - обязательно константа времени
T1 max(T a, T b)
{
    T x[i];
    return a > b ? a : b;
}
```

```
template <typename T>
class S
{
    T x;
    T f();
}
```

Обычно при создании такого класса нужно приписывать:

```
S<int> a;
```

## Специализация шаблона

Я хочу чтобы функция max для конкретного типа выглядела так.  
 В данном случае полная специализация т.к. <> - пустая  
 Частичная специализация <> - здесь есть какие-то аргументы (это запр  
 templtae<> float max<float>(float a, float b)  
 {

```
}
```

## ЧТО ТАКОЕ AUTO

```
void f(auto x)
{

}
```

==

```
template <typename T> void f(T x)
```