

Ламбда функције

- Видели смо да се функтори могу параметризовати тј. могу имати атрибуте.
- Могу и ламбда функције.
- Ламбда израз представља две ствари:
 - Опис нове, безимене, класе (типа) која има дефинисан оператор () на одговарајући начин – зваћемо је „ламбда класа“, или „ламбда тип“ (на енглеском то зову „closure type/class“)
 - Инстанцирање те класе – зваћемо ту инстанцу „ламбда објекат“ (на енглеском то зову „closure“)
- Сваки пут када се у извршавању наиђе на место где се користи ламбда израз, направиће се нова инстанца те ламбда класе – ламбда објекат.
- Синтакса коју смо до сада видели дефинише просту ламбда класу, која нема атрибуте, тако да ће свака инстанца те класе бити увек иста.

Захватање атрибута

- Ламбда класе могу имати атрибуте. Ова два кода имају исто дејство:

```
struct L {
    L(int x) : limit(x) {}
    void operator()(int x) const { if (x < limit) cout << x; }
private:
    int limit;
};
```

```
void foo(const vector<int>& vec) {
    for (int i = 0; i < 20; ++i) {
        auto pred = L(i);
        for (int a : vec) pred(a);
    }
}
```

```
void foo(const vector<int>& vec) {
    for (int i = 0; i < 20; ++i) {
        auto pred = [i](int x) { if (x < i) cout << x; };
        for (int a : vec) pred(a);
    }
}
```

Захватање атрибута

- Још једна интересантна илустрација:

```
for (int i = 0; i < 20; ++i) {
    auto pred = [i](int x) {
        if (x < i) cout << x;
        i = 5; // ово се неће превести јер се овде мења
               // атрибут ламбда објекта (који се исто зове i)
               // а функција operator() је const
    };
    for (int a : vec) pred(a);
}
```

```
for (int i = 0; i < 20; ++i) {
    auto pred = [i](int x) mutable {
        if (x < i) cout << x;
        i = 5; // ово ће се сад превести али и даље је у питању
               // атрибут ламбда објекта тако да промена утиче
               // само на pred, не на i у горњој фор петљи
    };
    for (int a : vec) pred(a);
}
```

Захватање атрибута

- Видели смо тзв. захватање атрибута по вредности.
- Међутим, атрибут може бити и референца на нешто.
- То се зове захватање (атрибута) по референци.

```
for (int i = 0; i < 20; ++i) {
    auto pred = [&i](int x) {
        if (x < i) cout << x;
        i = 5; // ово i и даље означава атрибут ламбда објекта
              // али који је сада референца на i из фор петље
              // дакле, i у фор петљи ће бити промењено
    };
    for (int a : vec) pred(a);
}
```

Захватање атрибута

- Захватати се могу само променљиве аутоматске трајности. (То су локалне променљиве које нису декларисане са кључном речи **static**.)
- Променљиве статичке трајности (глобалне променљиве и локалне са кључном речи **static**) се не могу захватати.
- Али то је зато што нема потребе – њима приступамо без захватања, као и из било које друге функције.

```
int global;
```

```
void foo(const std::vector<int>& vec) {  
    for (int i = 0; i < 20; ++i) {  
        auto pred = [i](int x) {  
            if (x < i) cout << x << " " << global;  
        };  
        for (int a : vec) pred(a);  
    }  
}
```

Захватање атрибута

- Остаје питање захватања атрибута објекта неке класе, када се ламбда дефинише у контексту те класе, тј. унутар методе те класе.
- Ако је у питању **static** атрибут, онда је то исто променљива статичке трајности па је одговор исти као на претходном слајду.
- Остали атрибути се не могу захватати... бар не директно и експлицитно.

```
class Test {  
    int y;  
public:  
    int x;  
    void bar() {  
        // auto ttt = [x, y]() { cout << x << y; };  
        auto ttt = [this]() { cout << x << y; };  
        ttt();  
    }  
};
```

Захватање атрибута

- Али, када захватимо **this**, њега јесмо захватили по вредности (**&this** је синтаксна грешка), и кроз њега директно приступамо атрибутима спољне класе, па их можемо и мењати.
- Слично као што из методе класе не морамо стално писати **this->attr**, да би приступили атрибуту **attr**.

```
class Test {
    int y;
public:
    int x;
    void bar() {
        auto ttt = [this]() { x = 5; cout << x; };
        ttt(); // исписаће 5, и то ће бити вредност x-а на даље
    }
};
```

Захватање атрибута – мала новина у Це++17

- Сада је могуће захватити ***this**.
- На овај начин захватамо копију тренутног објекта

```
class Test {
    int x;
    void bar() {
        auto ttt = [*this]() {
            //x = 5; // ово сада не може
            cout << x; };
        ttt();
    }

    void foo() {
        auto ttt = [*this]() mutable {
            x = 5; // ово сада може, али неће променити this->x
            cout << x; };
        ttt();
    }
};
```


Захватање атрибута

- Додатна олакшица: могуће је навести подразумевани начин захватања.

```
struct Test {  
    int m_x, m_y;  
    void bar() {  
        int a, b;  
        ...  
        auto L1 = [a, this]() { ... cout << a << m_x; ... }  
        // је исто као да смо написали ово  
        auto L1 = [=]() { ... cout << a << m_x; ... }  
  
        auto L2 = [&a, this]() { ... a = 5; m_x = 6; ... }  
        // је исто као да смо написали ово  
        auto L2 = [&]() { ... a = 5; m_x = 6; ... }  
  
        // а може и овако: а по референци, остало по вредности  
        auto L3 = [=, &a]() { ... cout << b; ... }  
        // или обрнуто: а по вредности, остало по референци  
        auto L4 = [&, a]() { ... cout << b; ... }  
    }  
};
```

Овде сада нема =

Захватање атрибута

- Али, декларисање подразумеваног начина захватања је обично врло опасно, јер може довести до нежељеног захватања.
- Зато, врло пажљиво са тим.
- Обично је много боље експлицитно навести шта захватамо и како.

Захватање атрибута

- Ваља још напоменути неке специфичности простих ламбда класа без захваћених атрибута.
- Најпре, ламбда објекат ламбда класе без атрибута најчешће неће ни бити стварно физички створен (јер ни нема величину, а служи само да се одреди која функција треба да буде позвана).
- Такође обратити пажњу да је и ово могуће:

```
using FuncType = bool(*) (int, int);  
FuncType pf = [] (int a, int b) { return a < b; };
```

- За остале ламбде овако нешто није могуће.

Генеричке ламбде

- Као што и обичне функције и класе могу да буду шаблони (генеричке), тако и лямбда функције могу бити шаблони.
- `operator()` ће напросто бити дефинисан као шаблон функције.

```
auto addOp = [](int x, int y) { return x + y; }  
addOp(5, 6);  
addOp("djura"s, "pera"s); // не може
```

```
auto addOpGen = [](auto x, auto y) { return x + y; }  
addOpGen(5, 6);  
addOpGen("djura"s, "pera"s); // сада може оба;  
                               // две функције operator() ће бити  
                               // направљене на основу шаблона
```

Закључивање повратне вредности

- Већ смо имали питање типа повратне вредности ламбда функције.
- Одговор је био да је тип израза у **return** наредби аутоматски тип повратне вредности. (А ако нема **return** наредбе, онда је **void**)
- Али, шта ако нам је тело функције сложеније? Шта ако садржи више **return** наредби?
- Стандард каже да сви изрази у свим **return** наредбама једне ламбда функције морају имати израз истог типа. (Од Cе++14)

```
auto L1 = [](int a, int b) {  
    if (a < b) return a;  
    return b;  
} // ово је у реду
```

```
auto L2 = [](int a, int b) {  
    if (a < b) return a;  
    return 6.0;  
} // ово није
```

Закључивање повратне вредности

- Али, одређеном синтаксом могу и да искажем експлицитно ког типа треба да буде повратна вредност.
- Та синтакса се назива „пратећи повратни тип“ (енгл. „trailing return type“).

```
auto L2 = [](int a, int b) -> int {  
    if (a < b) return a;  
    return 6.0;  
} // сада је и ово ОК
```

Закључивање повратне вредности

- Закључивање типа повратне вредности ради и са обичним функцијама.
- Правила су иста:
- Ако има само један `return` – тип његовог израза је тип повратне вредности.
- Ако има више `return` наредби – типови њихових израза морају бити исти.

```
auto foo1(int a, int b) {  
    return a + b;  
}
```

```
auto foo2(int a, int b) {  
    if (a < b) return a;  
    return 6.0;  
} // ово није добро
```

```
auto foo3(int a, int b) -> int { //  
    if (a < b) return a;  
    return 6.0;  
} // ово је ОК, мада смо могли и овако: int foo3(int a, int b)
```