# Ламбда функције

функцијски објекти (функтори), просте ламбда функције, захватање атрибута

# Функцијски објекти

- „Функцијски објекти“ (често и „функтори“) су променљиве („објекти“) које се синтаксно понашају као функције, односно могу тако да се употребљавају.
- То се обезбеђује преклапањем оператора ().
- Преклапање оператора облих заграда је слично преклапању оператора [], осим што код () можемо имати више параметара.

```
struct FunctionObjectShowroom{
    ...
    int operator()(int x);
    float operator()(double x);
    void operator()(int x, double y);
    int* operator()(float x, int y, long z);
    ...
};

FunctionObjectShowroom foo;
a = foo(6);
b = foo(7.6);
foo(a, b);
c = foo(7.5f, a, 46L);
```

# Функцијски објекти

- Тип функције чини информација о броју и типу параметара и типу повратне вредности. Дакле, две функције које примају једнак број параметара једнаког типа, и враћају повратну вредност једнаког типа, јесу функције које се не разликују по типу.
- Са друге стране, два функцијска објекта могу преклапати подједнак облик оператора () („примају једнак број параметара једнаког типа, и враћају повратну вредност једнаког типа“), али не морају бити истог типа.

```
int foo(double x, long y);
int bar(double a, long b);

int (*p)(double, long) = foo;
p(5.5, 7L);
p = bar;
p(5.5, 7L);
```

```
struct A {
  int operator()(double x, long y);
};
struct B {
  int operator()(double x, long y);
};
A x, y; x(5.5, 7L);
B z; z(5.5, 7L);
```

- bar и foo су истог типа.
- Функцијски објекти x и y су истог типа (A), али z је различитог типа (B).

# Функцијски објекти

- Функтори и класичне функције не могу се прослеђивати истим функцијама, јер су различитог типа.

```
void foo(int x);
void bar(int x);

void apply(void (*f)(int)) {
  ... f(elem); ...
}
apply(foo);
apply(bar);
apply(foA); // не може
```

```
struct A {
  void operator()(int x);
};
struct B {
  void operator()(int x);
};

A foA; B foB;

void apply(A f) {
  ... f(elem); ...
}

apply(foA);
apply(foB); // не може
```

# Функцијски објекти

- Али функтори и класичне функције могу инстанцирати исте шаблоне, јер се исто синтаксно понашају.

```
void foo(int x);
void bar(int x);
A foA;
B foB;

template<typename TFO>
void apply(TFO f) {
  ... f(elem); ...
}

apply(foo); // једна инстанца
apply(bar); // иста инстанца као у горњем реду
apply(foA); // друга инстанца
apply(foB); // трећа инстанца
```

```
struct A {
  void operator()(int x);
};
struct B {
  void operator()(int x);
};
```

- Због овога инстанцирање шаблона функторима може довести до бржег кода, јер компајлер тачно зна која функција ће се применити. Са обичним функцијама не зна се да ли је то foo или bar или нешто треће.

# Функцијски објекти

- Још једна погодност функцијских објеката је што се могу лако параметризовати, јер су суштински обичне класе и могу имати атрибуте.
- Илустровано на примеру инстанцирања шаблона функције find_if из стандардне библиотеке, која враћа први елемент из неког контејнера који задовољава неки услов (предикат).

```
struct Less_than
{
    int val;
    Less_than(int& x) : val(x) {}
    bool operator()(const int& x) const {
        return x < val;
    }
};

p = find_if(v.begin(), v.end(), Less_than(43));
p = find_if(v.begin(), v.end(), Less_than(76));
```

# Функцијски објекти

- Функтори се често употребљавају у раду са алгоритмима из стандардне библиотеке

```
struct Rec {
  string name;
  int age;
};
struct CmpByName {
  bool operator()(const Rec& a, const Rec& b) const
    { return a.name < b.name; }
};

struct CmpByAge {
  bool operator()(const Rec& a, const Rec& b) const
    { return a.age < b.age; }
};
vector<Rec> vr;
// ...
sort(vr.begin(), vr.end(), CmpByName()); // уреди по имену
sort(vr.begin(), vr.end(), CmpByAge()); // уреди по годинама
```

# Ламбда функције

- Ламбда функције су форма функторског литерала и олакшавају изражавање у многим случајевима.

```
struct Rec {
  string name;
  int age;
};
vector<Rec> vr;
// ...
sort(vr.begin(), vr.end(),
  [](const Rec& a, const Rec& b){ return a.name < b.name; });
sort(vr.begin(), vr.end(),
  [](const Rec& a, const Rec& b){ return a.age < b.age; });
```

- Јасно је наведено шта су улазни параметри ламбда функције, али где се наводи тип повратне вредности?
- Закључује се на основу типа израза у return;

# Ламбда функције

- Ког типа је ламбда функција?

```
int foo(double x, long y);

typedef int (*FooType)(double, long);

FooType p1 = foo;

struct A {
  void operator()(int x);
};

A p2 = A();


? p3 = [](int a, int b){ return a < b; };
```

# Ламбда функције

- auto много помаже у том случају.

```
int foo(double x, long y);

typedef int (*FooType)(double, long);

FooType p1 = foo;

struct A {
  void operator()(int x);
};

A p2 = A();


auto p3 = [](int a, int b){ return a < b; };
```

# Ламбда функције

- Видели смо да се функтори могу параметризовати.
- Могу и ламбда функције.
- Ламбда функција представља две ствари:
  - Опис нове, безимене, класе (типа) која има дефинисан оператор () на одговарајући начин – звaћемо је „ламбда класа“, или „ламбда тип“ (на енглеском то зову „closure type/class”)
  - Инстанцирање те класе – звaћемо ту инстанцу „ламбда објекат“ (на енглеском то зову „closure“)
- Сваки пут када се у извршавању наиђе на место где се користи ламбда функција, 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.
- У таквим случајевима ламбда објекат никада ни неће бити стварно физички створен (јер ни нема величину).

# Захватање атрибута

- Али и ламбда класе могу имати атрибуте. Ова два кода имају исто дејство:

```
struct L {
  L(int x) : limit(x) {}
  void operator()(int x) { if (x < limit) cout << x << " "; }
private:
  const 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 односи на
           // атрибут ламбда објекта (који се исто зове i)
           // а тај атрибут 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 методе.

```
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 објеката 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 (и само тако и можемо да га захватимо - &this је синтаксна грешка), али 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.

# Генеричке ламбде

- Као што и обичне функције и класе могу да буду шаблони (генеричке), тако и ламбда функције могу бити шаблони.

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

auto addOpGen = [](auto x, auto y) { return x + y; }
addOpGen(5, 6);
addOpGen(“djura”, “pera”); // сада може оба
```

- Ламбда класа код генеричке ламбде је сада заправо шаблон класе.

# 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 Це++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)
```

# Употреба ламбди у СТЛ алгоритмима

- Ламбда функције омогућавају да се СТЛ алгоритми (<algorithm> и <numeric>) много једноставније користе.
- Том техником се врло сложене обраде могу изразити врло једноставно и у мало кода, а резултујући код је и даље врло ефикасан.
- Један пример смо већ видели:

```
struct Rec {
  string name;
  int age;
};
vector<Rec> vr;
// ...
sort(vr.begin(), vr.end(),
  [](const Rec& a, const Rec& b){ return a.name < b.name; });
sort(vr.begin(), vr.end(),
  [](const Rec& a, const Rec& b){ return a.age < b.age; });
```

- Али има их још...

# Употреба ламбди у СТЛ алгоритмима

- Функција accumulate.

```
template<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op) {
 while (first != last) {
     init = op(init, *first); // “init op *first”
     ++first;
 }
 return init;
}

list<double>& ld;
double product = accumulate(ld.begin(), ld.end(),
    1.0, [](auto x, auto y){ return x*y; };
double sum = accumulate(ld.begin(), ld.end(),
    0.0, [](auto x, auto y){ return x+y; };
```

# Употреба ламбди у СТЛ алгоритмима

- Функција inner_product.

```
template<class In, class In2, class T, class BinOp, class BinOp2 >
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2) {
  while (first != last) {
       init = op(init, op2(*first, *first2));
       ++first;
       ++first2;
  }
  return init;
}
```

```
vector<double>& vd; list<double> ld;
double product = inner_product(ld.begin(), ld.end(), vd.begin(),
    0.0, [](auto x, auto y){ return x+y; },
    [](auto x, auto y){ return x*y; });
```