C++11 Language Features

Alex Korban
@cpp_rocks
cpprocks.com/blog

Overview

C++11

{ Language }

Libraries

-std=c++11

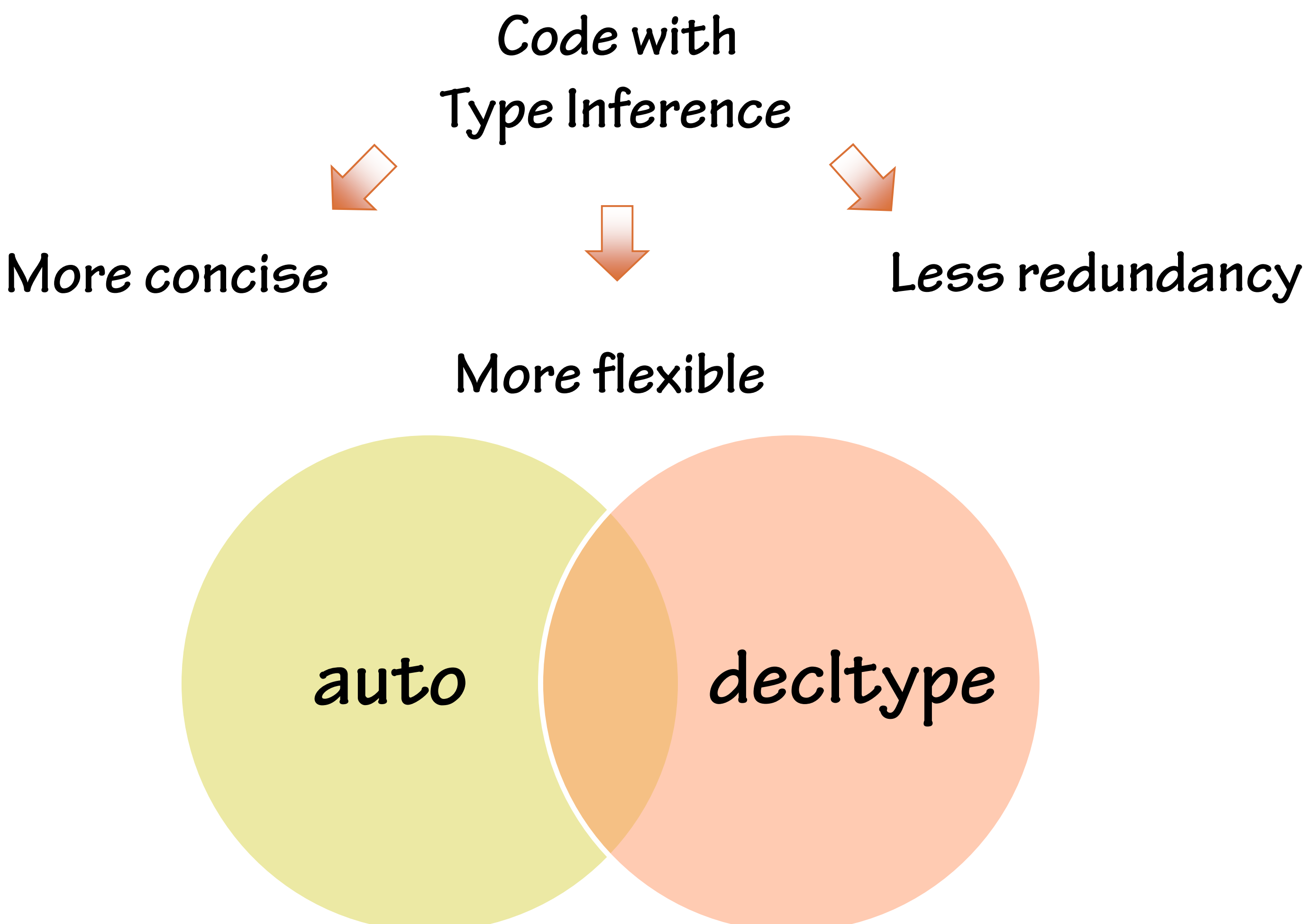
C++11 Purpose and Guiding Principles

- Standard library additions over changes to the language
- Improving abstraction mechanisms
- Increasing type safety
- Improving performance
- Zero overhead principle
- Maintaining backwards compatibility

In This Module

- Type inference
- Trailing return type syntax
- Lambda expressions

Type Inference



auto

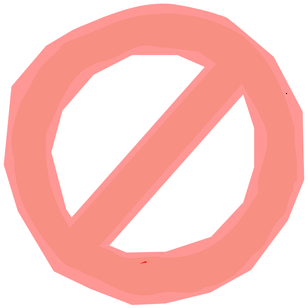
```
std::map<std::string, std::vector<int>>::const_iterator
```

x 100

```
auto a = 5;
auto plane = JetPlane("Boeing 737");
cout << plane.model();
for (auto i = plane.engines().begin(); i != plane.engines().end(); ++i)
    i->set_power_level(Engine::max_power_level);
```

Some Things Are Still Manual

```
void invalid(auto i) {}

class A
{
    auto _m;
};

int main()
{
    auto arr[10];
}
```



More Than Syntactic Sugar

```
template<typename X, typename Y>
void do_magic(const X& x, const Y& y)
{
    auto result = x * y;   // what is the type of result?
    // ...
}
```

Why Else Do We Need It?

- Don’t Repeat Yourself
- Higher level of abstraction
- Type changes are better localized
- Easier refactoring
- Simpler template code
- Declaring variables of undocumented or unnamable types

Objection, Your Honor!

Benefits > Costs

Diving In

```
auto a = 5.0, b = 10.0;

auto i = 1.0, *ptr = &a, &ref = b;

auto j = 10, str = "error";    // compile error
```

```
map<string, int> index;

auto& ref = index;
auto* ptr = &index;
const auto j = index;
const auto& cref = index;
```

Diving In

- *const* and *volatile* specifiers are removed
- arrays and functions are turned into pointers

```
const vector<int> values;
auto a = values;             // type of a is vector<int>
auto& b = values;            // type of b is const vector<int>&

volatile long clock = 0;
auto c = clock;              // c is not volatile

JetPlane fleet[10];
auto e = fleet;              // type of e is JetPlane*
auto& f = fleet;             // type of f is JetPlane(&)[10] - a reference

int func(double) { return 10; }
auto g = func;               // type of g is int(*)(double)
auto& h = func;              // type of h is int(&)(double)
```

Diving In

```
int i = 10;

auto a = i;
auto b(i);


struct Expl
{
    Expl() {}
    explicit Expl(const Expl&) {}
};

Expl e;
auto c = e; // compile error
```

A Little Bit of auto History

```
template<typename T>
void f(T t)
{}

f(expr);             // T is deduced from expr

auto var = expr;  // type of var is deduced from expr, same as above
```

decltype

```
int i = 10;
cout << typeid(decltype(i + 1.0)).name() << endl; // outputs "double"
```

```
vector<int> a;
decltype(a) b;
b.push_back(10);
decltype(a)::iterator iter = a.end();
```

```
template<typename X, typename Y>
auto multiply(X x, Y y) -> decltype(x * y)
{
    return x * y;
}
```

Side Effects

```
decltype(a++) b;


template <int I>
struct Num
{
    static const int c = I;
    decltype(I) _member;
    Num() : _member(c) {}
};

int i;
decltype(Num<1>::c, i) var = i;     // var is int&
```

decltype

```
class A
{
private:
    A();
};

cout << typeid(decltype(A())).name() << endl; // doesn't compile:
                                              // A() is private


cout << typeid(decltype(declval<A>())).name() << endl;  // OK
```

auto, decltype - How about Both at Once?

```
template<typename X, typename Y>
auto multiply(X x, Y y) -> decltype(x * y)
{
    return x * y;
}
```

```
template<typename X, typename Y>
ReturnType multiply(X x, Y y)
{
    return x * y;
}
```

```
template<typename X, typename Y>
decltype(x * y) multiply(X x, Y y) // x and y in decltype aren't in scope yet!
{
    return x * y;
}
```

Lambda Expressions

```
for_each(v.begin(), v.end(),
    [](const JetPlane &jet) { cout << jet.model() << endl; });
```

Lambda introducer

Parameter list

Body of the lambda expression

```
class lambda0
{
public:
    void operator()(const JetPlane& jet) const {
        cout << jet.model() << endl;
    }
};

for_each(v.begin(), v.end(), lambda0());
```

Why Do We Need This Thing?

- Improve locality
- Reduce boilerplate
- Express intentions better

Why Do We Need This Thing?

```
auto const_val = some_default_value;
if (some_condition_is_true)
{
    // Do some operations and
    // calculate the value of const_val
    const_val = calculate();
}
const_val = 1000;  // oops, const_val
                   // can be modified later!
```

```
const auto const_val = [&] {
    auto const_val = some_default_value;
    if(some_condition_is_true)
    {
        // Do some operations and
        // calculate the value of const_val
        const_val = calculate();
    }
    return const_val;
}(); // lamdba is invoked immediately!
```

Return Type

```
[](int i) -> double { if (i > 10) return 0.0; return double(i); }
```

Lambda Parameters

- No default values for parameters
- No variable length argument lists
- No unnamed parameters

```
[](JetPlane& jet, const date_t& date) { jet.require_service(date); }
```

Lambda Body

Storing Lambdas

```
??? f = [](int i) { return i > 10; };
```

Unknowable type

```
auto f = [](int i) { return i > 10; };

f(5);   // returns false
```

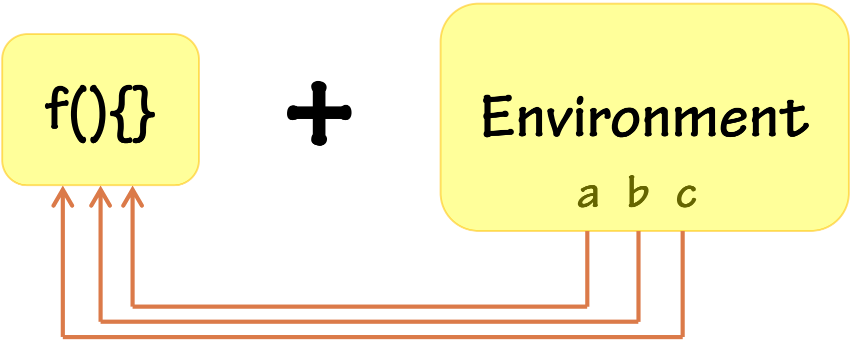
std::function to the Rescue

```
#include <functional>

class LambdaStore
{
    function<bool(double)> _stored_lambda;
public:
    function<int(int)> get_abs() const
    {
        return [](int i) { return abs(i); };
    }

    void set_lambda(const function<bool(double)>& lambda)
    {
        _stored_lambda = lambda;
    }
};

LambdaStore ls;

ls.set_lambda([](double d) { return d > 0.0; });

auto abs_lambda = ls.get_abs();
abs_lambda(-10);   // returns 10
```

References to Outside Context

```
{
    var

    lambda = [] {  ...   var   ...   }
}

lambda()
```

Closures



Capturing in C++11

```
[today](JetPlane& jet) { jet.require_service(today); }
```

```
class lambda1
{
    date_t _today;
public:
    lambda1(date_t today): _today(today) {}

    void operator()(JetPlane& jet) const
    {
        jet.require_service(_today);
    }
};
```

Capturing in C++11

```
function<bool()> g()
{
    static auto a = 5;
    static auto b = -3;
    return []() { return a + b > 0; };
}


function<bool()> f()
{
    auto a = 5;
    auto b = -3;
    // won't compile if a & b aren't captured
    return [a, b]() { return a + b > 0; };
}
```

Capturing by Reference

```
JetPlane jet;
vector<Person> passengers;

for_each(passengers.begin(), passengers.end(),
    [&jet](const Person& p) { jet.load_passenger(p); });


class lambda1
{
    JetPlane& _jet;
public:
    lambda1(JetPlane& jet): _jet(jet) {}
    void operator()(Person& p) const { _jet.load_passenger(p); }
};



int a, b, c, d;
[a, &b, c, &d]() {};
```

Default Capture Modes

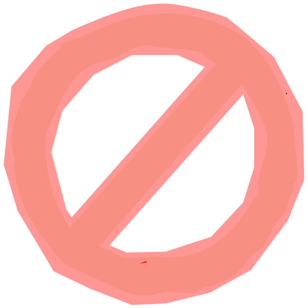
```cpp
int a, b, c, d;

[=]() { return (a > b) && (c < d); };


[&]() { a = b = c = d = 10; };


// override default capture by value
[=, &a]() { a = 20; };

// override default capture by reference
[&, d]() { d = 20; };  // doesn't compile because d is captured by value



[&a, &b, &c, x, y, &z]            [&, x, y]
```

Capturing Class Members

```
class JetPlane
{
    const int _min_fuel_level;
    vector<Tank> _tanks;

public:
    bool is_fuel_level_safe()
    {
        return all_of(_tanks.begin(), _tanks.end(),
            [this](Tank& t) { return t.fuel_level() > _min_fuel_level; });
    }

    bool is_fuel_level_critical()
    {
        return any_of(_tanks.begin(), _tanks.end(),
            [=](Tank& t) { return t.fuel_level() <= _min_fuel_level; });
    }
};
```

Limitations of Capturing

```
#include <iostream>

auto x = 12;
auto f = [&x]() { return x; };

int main()
{
    std::cout << f() << std::endl;
}
```



Mutable Lambdas

```
vector<pair<int, int>> flight_hours;
// ... flight hours are populated with values ...

auto running_total = 100; // from previous month

for_each(flight_hours.begin(), flight_hours.end(),
    [running_total](pair<int, int>& x) mutable
        { running_total += x.first; x.second = running_total; });
```

```
template <class Func>
void by_const_ref(const Func& f) { f(); }

by_const_ref([] {});            // OK

by_const_ref([]() mutable {}); // OK

string s("executing mutable lambda");
by_const_ref([s]() mutable { cout << s << endl; });  // error
```

Conversion to Function Pointers

```
typedef int (*Func)();

Func f = [] { return 10; };

f(); // invoke lambda via function pointer
```

Nested Lambdas

```
auto mode = public_announcement;
vector<Cabin> cabins;

for_each(cabins.begin(), cabins.end(),
    [=](Cabin& cabin)
    {
        for_each(cabin.seat_screens().begin(), cabin.seat_screens().end(),
            [=](SeatScreen& seat_screen) { seat_screen.set_mode(mode); });
    });
```

Lambdas and Recursion

```
function<int(int)> fibonacci = [&](int n) -> int
{
    if (n < 1)
        return -1;
    else if (n == 1 || n == 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
};
```

How Not to Shoot Yourself in the Foot

```
function<int()> f;

{
    auto i = 5;
    f = [&i] { return i; };
}

f();  // undefined because i is out of scope
```

```
function<int()> f;

{
    auto p = new int(10);
    f = [=] { return *p; };
    delete p;
}

f();  // undefined behavior because p has been deleted
```

How Not to Shoot Yourself in the Foot

```
function<int()> f;

class Plane
{
    int _capacity;
public:
    Plane(int capacity): _capacity(capacity) {}
    function<int()> get_lambda() const
    {
        return [=] { return _capacity; };
    }
};

{
    Plane plane(10);
    f = plane.get_lambda();
}

f();  // undefined behavior because plane is out of scope now
```

Rules of Thumb for Lambdas

- Write short and clear lambdas
- If it's becoming long, you might need a function object
- Don't duplicate code across lambda expressions

Lambda Syntax in All Its Glory

```
[capture_block](parameter_list) mutable exception_spec -> return_type { body }
```

Summary

- Type inference: auto and decltype
- Trailing return type syntax
- Lambda expressions