Intermediate Level C++

C++11: New Features

Yong Zhang

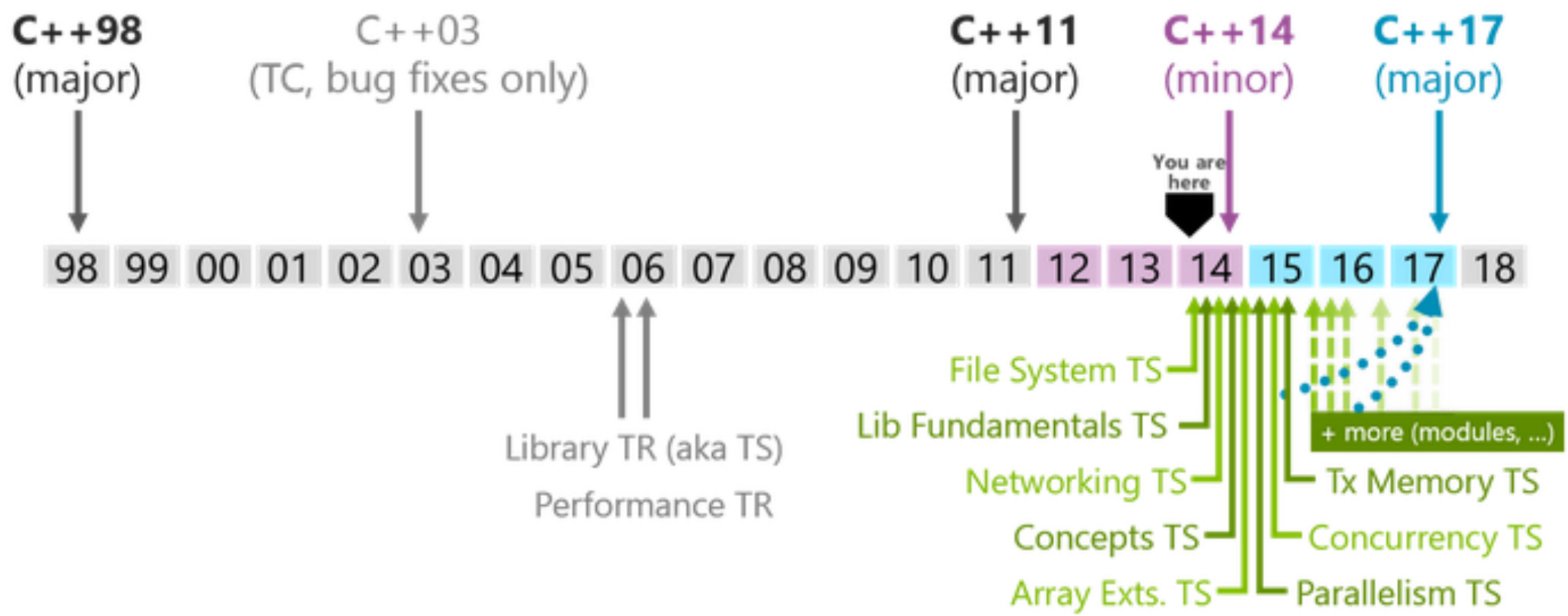
Objectives

In this chapter you will learn:

- What is new in Visual Studio 2013
- Type inference using auto and decltype
- Trailing return types
- Lambda expressions
- Uniform Initialization

C++ Current Status

- http://isocpp.org/std/status



What is new in Visual Studio 2013

- **Auto completion of braces, parentheses, quotes etc**
- **Switch between header and cpp file**
- **Intellisense**
- **Code Formatting**
- **Demo: WhatIsNewInVS2013**

auto

- **Type inference: automatic deduction of the data type of an expression.**
- **C++11 allows type inference using two ways:**
  - auto (declare variables)

```
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);
```

```
void invalid(auto i) {}

class A
{
    auto _m;
};

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

  - Decltype (general d

Type Inference

- **Type inference: automatic deduction of the data type of an expression.**
- **C++11 allows type inference using two ways:**

auto and decltype

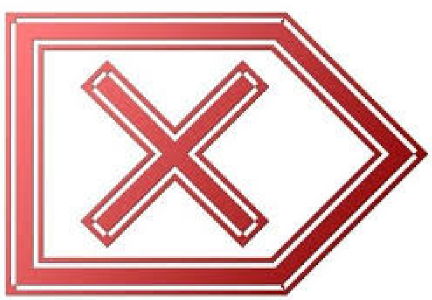
```
for (std::vector<int>::const_iterator itr = myvec.cbegin(); itr != myvec.cend(); ++itr)
```

```
for (auto itr = myvec.cbegin(); itr != myvec.cend(); ++itr)
```

auto

- **auto: define a variable with an explicit initialization**

```
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);
```



```
void invalid(auto i) {}

class A
{
    auto _m;
};

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

auto (continued)

- **auto: can be used when it is hard to specify the type explicitly:**

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

- **auto: can be used to declare multiple variables in the same statement:**

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

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

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

auto (continued)

- **auto: can be used with const and volatile qualifiers:**

```
map<string, int> index;

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

```
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)
```

decltype

- **decltype: determine the type of an expression at compile time:**

```
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;
}

decltype(a++) b;
```

Trailing Return Type

- **Trailing return type: return type is specified after the parameters:**

```
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

- **Lambda expressions are used to define anonymous functions in place right where you need them.**
- **Benefits of using lambda expressions**
  - Improve locality
  - Reduce boilerplate
  - Express intentions better

```
[capture](parameters) -> return_type { function_body }
```

```
[](int x, int y) -> int { return x + y; }
```

Lambda Expressions (continued)

- **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: just like a normal function**

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

Lambda Expressions (continued)

- **Store Lambdas in a variable**

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

Unknowable type

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

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

Lambda Expressions (continued)

- **Referring to use external variables**

```
{
    var

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

lambda()
```

- **Free variables:**
  - variables referred to in a function that isn't either local or an argument
- **Closures**
  - a function combined with a referencing environment for the non-local variables of a function
  - a function is closed over its free variables, hence the term closure
- **Capturing:**
  - The referencing environment binds the non-local variables to corresponding local variables in the function when the closure is created

Lambda Expressions (continued)

- **Capturing by value:**

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

- **Capturing by reference:**

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

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

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

Lambda Expressions (continued)

- **Default Capture Mode:**

```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
```

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

Lambda Expressions (continued)

- **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; });
    }
};
```

Uniform Initialization

- **Many ways for Initialization**
  - to initialize built in types:

```
int a = 2;
int b(2);
```

  - to initialize C-style arrays:

```
int list[4] = {2, 4, 6, 8};
char letters[5] = {'a', 'e', 'i', 'o', 'u'};
double numbers[3] = {3.45, 2.39, 9.1};
int table[3][2] = {{2, 5} , {3,1} , {4,9}};
```

  - to initialize an object:

```
Foo f = 3;
Employee newHire(John, today + 1, salary);
Employee CEO();
Employee someone;
```

Uniform Initialization (continued)

- **Uniform Initialization – braces are always okay**

```
int a{2};
Employee CEO{};
Employee newHire {John,today+1,salary}
vector<int> v {1,2,3,4};
vector<Employee> staff {CEO,newHire};


vector<Employee> company { CEO,
                           newHire,
                           {Mary, today+1, salary}
                         };
```

- **Benefits: consistent and easy to read and understand**
- **Demo: UniformInitialization Project**