System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block G: Advanced C++

12. Other Language Improvements

Eugene Zouev

Structured bindings

Conditions in if/switch, for/while

Structured binding

```
auto [ x, y, z ] = expression ;

auto [ x, y, z ] { expression } ;

auto [ x, y, z ] ( expression );
```

Simplified

Since C++17

1. Introduces variables from brackets to the current scope.
2. Binds them to <u>subobjects</u> or <u>elements</u> of the object from *expression*.

Structured binding

Examples

```
int a[2] = { 1, 2 };

auto [x,y] = a;

auto& [xr, yr] = a;
```

A temporary array e is created. Array a gets copied to e.

x refers to e[0], and y refers to e[1].

xr refers to a[0], and yr refers to a[1].

Structured binding

Examples

```
struct S {
   int x;
   const double y;
};
S f();

const auto [x, y] = f();
```

x is of type const int;

y is of type const double

```
std::tuple<int,int&> f();

auto [x, y] = f();

const auto [z, w] = f();
```

x is of type int;

y is of type int&

z is of type const int;

w is of type int&

Structured binding: References

- ISO Standard, Section 11.5
- http://en.cppreference.com/w/cpp/language/structured_binding

The notion of “condition”

The Notion of "Condition"

Canonical form

```
if ( condition )
    statement
else
    statement
```

What is "condition"?

- Expression, contextually convertible to bool
- Declaration of a <u>single non-array variable with an initializer</u>

Since C++03

Syntax rule (simplified):

condition:
    expression
    decl-specifier-seq declarator = initializer-clause
    decl-specifier-seq declarator braced-init-list

The Notion of "Condition"

Example

```
if ( int x = f() )
{
    int a;
    cout << x;
}
else
{
    int b;
    y = x + 1;
}
```

The scope of a;
a & x are visible

The scope of b;
b & x are visible,
but not a.

The scope of x

Hint: the declaration in the condition is not necessarily of the boolean type!

A question: is it really useful?

- Really, the value of x **is definitely true** in the then-part, and **definitely false** in the else-part...

The Notion of "Condition"

The *newest* form

```
if ( init-statement_opt condition )
    statement
else
    statement
```

Since C++17

What is "init-statement"?

- ***An expression-statement*** (i.e., *expression* with **;**)
- ***A simple-declaration*** (i.e., several declarations with initializers)

The Notion of "Condition"

Example

This is *simple-declaration* This is *condition*

```
if ( int a = f(), b = f2(); a && b )
{
   // Do something with a and b
}
```

Of course, this is the same as: ☺☺

```
{
   int a = f(), b = f2();
   if ( a && b )
   {
      // Do something with a and b
   }
}
```

"Condition" in `while`

Canonical form

```
while ( condition )
   statement
```

What is "condition"?

- **Expression, contextually convertible to bool**
- **Declaration of a single non-array variable with an initializer**

Since C++03

If *condition* is a declaration such as `T t = x`, the declared variable is only in scope in the body of the loop, and is **destroyed** and **recreated** on every iteration

"Condition" in for

Canonical form

```
for ( init-statement condition_opt ; expression_opt )
    statement
```

Since C++03

What is "init-statement"?

- ***An expression-statement*** (i.e., *expression* with **;**)
- ***A simple-declaration*** (i.e., several declarations with initializers)

Schematic example

```
for ( int x=f1(), x2=f2(); x+y<100; x++,y++ )
    Loop body with x and y
```

The Notion of “Condition”: References

- ISO Standard, Sections 6.4, 6.5
- https://stackoverflow.com/questions/7836867/c-variable-declaration-in-if-expression
- http://en.cppreference.com/w/cpp/language/if
- http://en.cppreference.com/w/cpp/language/while

For-range

for-range

For's advanced form

```
for ( range-declaration : range-expression )
    loop-statement
```

Since C++11

Range-declaration:

A declaration of a named variable, whose type is the type of the element of the sequence represented by *range_expression*, or a reference to that type. Typically, **auto** specifier is used for automatic type deduction

Range-expression

Any expression that represents a suitable sequence (either an array or an object for which begin and end member functions or free functions are defined) or a ***braced list***.

for-range: examples

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
   vector<int> v = {0, 1, 2, 3, 4, 5};

   for (const int& i : v)
      cout << i << ' ';

   for (auto i : v)
      cout << i << ' ';

   for (int n : {0, 1, 2, 3, 4, 5})
      cout << n << ' ';

   int a[] = {0, 1, 2, 3, 4, 5};
   for (int n : a)
      cout << n << ' ';

   for (int n : a)
      cout << 1 << ' ';
}
```

access by const reference

access by value, the type of i is int

the initializer may be a *braced-init-list*

the initializer may be a usual array

No need for any loop ☺☺

for-range: informal semantics

```
for ( range-declaration : range-expression )
    loop-statement
```

range-expression is evaluated to determine the **sequence** or **range** to iterate. Each element of the sequence, in turn, is **dereferenced** and **assigned** to the variable with the type and name given in *range-declaration*.

```
{
  auto && __range = range_expression;
  auto __begin = begin_expr ;
  auto __end = end_expr ;
  for ( ; __begin != __end; ++__begin)
  {
    range_declaration = *__begin;
    loop_statement
  }
}
```

If *range-expression* is an array:
begin_expr is *__range*
end_expr is *__range+__bound* (array size)

If *range-expression* is an object of a class type `C`:
begin_expr is *__range*.begin()
end_expr is *__range*.end()

- The assumption is that class `C` contains member functions `begin()` & `end()`.

Since C++17

`for`-range: references

- ISO Standard, Section 6.5.4
- http://en.cppreference.com/w/cpp/language/range-for

Initialization semantics

Four initialization forms

```
int x(0);           // initializer in parentheses
int y = 0;          // initializer after '='

int z { 0 };        // initializer in braces
int t = { 0 };      // initializer in braces
                    // with '='
```

Since C++11

Here, "=" doesn't denote assignment, but **initialization**!!!

```
class C { ... };

C c1;      // default constructor
C c2 = c1; // copy constructor
C1 = c2;   // assignment via operator=()
```

Uniform initialization

The idea was to define a syntax construct that could represent **all possible kinds** of initialization.

The syntax construct is **braced initialization** (to be more precise, *braced-init-list*).

More things become possible with `{ }`...

```
std::vector<int> v1(1,2,3,4,5,6);  // error

std::vector<int> v2{1,2,3,4,5,6};  // OK!
```

```
class C { ... };
C c1();    // not an object but function declaration
C c2{};    // OK: object declaration ☺
```

Uniform initialization

More things become possible with { }...

Default member initialization

```
class C {
   ...
  private:
   int x { 0 };   // OK
   int y = 0;     // OK
   int x(0);      // Error
};
```

No narrowing conversions

```
double x, y, z;
 ...
int sum2(x+y+z);     // OK
int sum3 = x+y+z;    // OK

int sum1 { x+y+z }; // Error
```

Data loss

More careful checks

Uniform initialization: semantics

The common rule is that the construct like

```
{ v1, v2, v3, … }
```

is considered as the value of type

```
initializer_list<T>
```

```
int z { 77 };
int t = { 77 };
```

Both `z` and `t` get the single value of `77`

```
auto z { 77 };
auto t = { 77 };
```

Here, type of `z` and `t` is deduced as `initializer_list<int>` with the single value of `77` in it!..

Uniform initialization: semantics

```
class C {
  public:
    C(int i, bool b);    // 1
    C(int i, double d);  // 2
    ...
};

C c1(10,true);  // #1
C c2{10,true};  // the same

C c3(10,5.3);   // #2
C c4{10,5.3};   // the same
```

Here, parentheses & braces
have the same semantics…

Uniform initialization: semantics

…but if we **add one more constructor**, the situation changes…

```
class C {
  public:
    C(int i, bool b);     // 1
    C(int i, double d);   // 2
    ...
    C(initializer_list<double> il);  // 3
};

C c1(10,true);  // #1
C c2{10,true};  // 10 & true get converted both to
                // double, and constructor #3 is invoked!

C c3(10,5.3);   // #2
C c4{10,5.3};   // 10 gets converted to double, and
                // constructor #3 is invoked!
```

Uniform initialization: references

- ISO Standard, Section 11.6.4
- Scott Meyers, Effective Modern C++, O'Reily.