The New Old Thing

Or Fun With Lambdas

Jonathan Caves
joncaves@microsoft.com - @joncaves

Fun With Lambdas

- Sumant Tambe wrote a blog post:

  http://cpptruths.blogspot.com/2014/08/fun-with-lambdas-c14-style-part-3.html

Fun With Lambdas

```
auto list = [](auto ...v) {
   return [=](auto access) { return access(v...); };
};
auto map = [](auto func) {
   return [=](auto ...z) {
      return list(func(z)...);
   };
};
auto print = [](auto v){
   std::cout << v; return v;
};
int main(){
   list(1, 2, 3, 4)(map(print));
}
```

Fun With Lambdas

- Sumant tested his code with clang and gcc
  - clang output - “1234”

Fun With Lambdas

- Sumant tested his code with clang and gcc
  - clang output - “1234”
  - gcc output - “4321”

Fun With Lambdas

- Sumant tested his code with clang and gcc
  - clang output - “1234”
  - gcc output - “4321”
- Sumant assumed the difference was due to a bug in gcc

Fun With Lambdas

- Visual C++ matches the behavior of gcc – “4321”

Fun With Lambdas

- Visual C++ matches the behavior of gcc – “4321”
  - Except if you target ARM in which case we match clang – “1234”

Fun With Lambdas

- Visual C++ matches the behavior of gcc – “4321”
  - Except if you target ARM in which case we match clang – “1234”
- So what is the correct output?
- What is the bug?

The New Old Thing

- The issue is with this line of code:

```
return list(func(z)...);
```

- After pack expansion it is conceptually equivalent to:

```
return list(print(z1), print(z2), print(z3), print(z4));
```

- So what is wrong with that?

The New Old Thing

- ISO/IEC 9899 -3.3.2.2/p9

  “The order of evaluation of the function designator, the arguments, and subexpressions within the arguments is unspecified, but there is a sequence point before the actual call.”

- N3936 – 5.2.2/p8

  “[Note: The evaluations of the postfix expression and of the arguments are all unsequenced relative to one another. All side effects of argument evaluations are sequenced before the function is entered (see 1.9). —end note]”

A Wider Problem

- Assignment

```
std::vector<int> v;
int i = 0;
v[i] = ++i;
```

- Member access

```
f1()->mf(f2());
```

Is There A Solution?

- “Fix” the order of evaluation of sub-expressions in C++.
  - Enforce left to right evaluation
- Possible: but there are implications
  - Performance: there is a cost
  - Behavior: it may change