

Two C++ Tools*

Compiler Explorer and Cpp Insights

Alison Chaiken
alison@she-devel.com
Jan 23, 2020



*with a brief excursion into HW exploits

Overview

- Compiler Explorer and Cpp Insights look under the hood of C++ compilation.
 - Both kick off a compiler within the browser and show side-by-side source and output.
 - Both can be locally hosted.
- Compiler Explorer produces assembly output.
- Cpp Insights shows the output from the clang parser (specifically AST converted back to C++).

Compiler Explorer Basics

- Supports GCC and Clang plus many more.
- Multiarch including many ARM flavors.
- Arbitrary compiler options are supported.
- Settles a lot of arguments about what the compiler actually does.
- Has a [wiki](#), [FAQ](#).

CE example: the “Spectre” exploit

- Many security holes involving speculation execution by processors disclosed in recent years.
- Exploits now exist “in the wild.”
- CE illustrates how the “retpoline” fix for C++ indirect branch speculation works.

C++ Indirect Branch

Example: A common C++ indirect branch

```
class Base {  
    public:  
        virtual void Foo() = 0;  
};  
  
class Derived : public Base {  
    public:  
        void Foo() override { ... }  
};  
  
Base* obj = new Derived;  
obj->Foo();
```

The fix: “retpoline”

- *trampoline*: intermediary function that execution bounces off
- Takes advantage of the fact that in modern ISAs, “function return is itself an indirect branch. However, unlike other indirect branches, its target may be directly cached for exact future prediction at the point of function call.”[[source](#)]
- retpoline strategy: make sure that a do-nothing branch keeps the processor busy so that the desirable branch has a chance to look up the correct address.

ASM without a retpoline

```
call Derived::Derived() [complete object constructor]
mov  QWORD PTR [rbp-24], rbx
mov  rax, QWORD PTR [rbp-24]
mov  rax, QWORD PTR [rax]
mov  rdx, QWORD PTR [rax]
mov  rax, QWORD PTR [rbp-24]
mov  rdi, rax
call rdx
```

With GCC and -mindirect-branch=thunk

Demo: `-mindirect-branch=thunk`

Clear or set this option to see the code
with or w/o the retpoline.

Diff with -mindirect-branch=thunk

```
diff -u /home/alison/Peloton/Cpp-Tools/nothunk.asm /home/alison/Peloton/Cpp-Tools/thunk.asm
--- /home/alison/Peloton/Cpp-Tools/nothunk.asm 2020-01-09 05:27:00.915661235 -0800
+++ /home/alison/Peloton/Cpp-Tools/thunk.asm 2020-01-08 21:09:43.465911467 -0800
@@ -54,7 +54,7 @@
     mov     rdx, QWORD PTR [rax]
     mov     rax, QWORD PTR [rbp-24]
     mov     rdi, rax
-   call    rdx
+   call    __x86_indirect_thunk_rdx
     mov     eax, 0
     add     rsp, 24
     pop     rbx
@@ -107,3 +107,12 @@
     call    __static_initialization_and_destruction_0(int, int)
     pop     rbp
     ret
+__x86_indirect_thunk_rdx:
+   call    .LIND1
+.LIND0:
+   pause
+   lfence
+   jmp     .LIND0
+.LIND1:
+   mov     QWORD PTR [rsp], rdx
+   ret
```

Cpp Insights Basics

- Clang only.
- Support for various C++ versions.

Demo with template and lambda instantiation

How does the preprocessor resolve auto?

```
7 template <typename T>
8 void CalculateListStatistics(::std::list<T> *elemlist,
9                             ::std::map<T, int> &countmap) {
10     for (T elem : *elemlist) {
11         ::std::pair<T, int> candidate(elem, 1);
12         auto result = countmap.insert(candidate);
13         if (false == result.second) {
14             result.first->second++;
15         }
16     }
17 }
```

Maybe `std::pair<T *, bool> result; ?`

```
20 /* First instantiated from: insights.cpp:31 */
21 #ifdef INSIGHTS_USE_TEMPLATE
22 template<>
23 void CalculateListStatistics<long>(std::list<long, std::allocator<long>> & __range1 = *elemlist,
24 {
25     {
26         std::list<long, std::allocator<long>> & __range1 = *elemlist;
27         for(std::__list_iterator<long, void*> __begin0 = std::__list_iterator<long, void*>(__range1.begin(), __range1.end()),
28             __end0 = __begin0; __begin0 != __end0; ++__begin0)
29             {
30                 long elem = __begin0.operator*();
31                 ::std::pair<long, int> candidate = std::pair<long, int>(elem, 1);
32                 std::pair<std::__map_iterator<std::__tree_iterator<std::__value_type, long, std::less<long>,
33                     if(static_cast<int>(false) == static_cast<int>(result.second))
34                         result.first.operator->()->second++;
35             }
36     }
37 }
38 }
39 }
40 #endif
```

The result of template instantiation

The answer:

```
std::pair<std::__map_iterator<std::__tree_iterator<  
std::__value_type<long, int>,  
std::__tree_node<std::__value_type<long, int>,  
void*>*, long>>, bool> result
```

Freestanding Lambda Expressions are Classes

```
class __lambda_19_16
{
public:
    inline long operator()() const
    { return (random() % static_cast<long>((ELEMNUM - 1))); }
    using retType_19_16 = auto (*)() -> long;
    inline operator retType_19_16 () const noexcept
    { return __invoke; };
private:
    static inline long __invoke()
    { return (random() % static_cast<long>((ELEMNUM - 1))); }
};

__lambda_19_16 GetRand = __lambda_19_16(__lambda_19_16{});
```

Example: macros vs. constexpr

Demo: first CppInsights,
then CompilerExplorer

Comparison: constexpr vs. C-style macro

- The input code:

```
#define CUBE(X) ((X) * (X) * (X))
```

```
constexpr Complex cubeme(const Complex &x) { return x * x * x; }
```

Calls sqrt() and cubeme() function each 1x.

▣ `constexprversion():`

```
push    rbp
mov     rbp, rsp
sub     rsp, 32
movsd  xmm0, QWORD PTR .LC2[rip]
movsd  QWORD PTR [rbp-32], xmm0
movsd  xmm0, QWORD PTR .LC3[rip]
movsd  QWORD PTR [rbp-24], xmm0
lea    rax, [rbp-32]
mov    rdi, rax
call   sqrt(Complex const&)
movq   rax, xmm0
movapd xmm0, xmm1
mov    QWORD PTR [rbp-16], rax
movsd  QWORD PTR [rbp-8], xmm0
lea    rax, [rbp-16]
mov    rdi, rax
call   cubeme(Complex const&)
mov    eax, 0
mov    edx, 0
movq   rax, xmm0
movq   rdx, xmm1
nop
leave
ret
```

constexpr code
calls operator*()
2x, for a total of 1
sqrt() and 2
operator*() calls.

cubeme(Complex const&):

```
push    rbp
mov     rbp, rsp
push    rbx
sub     rsp, 40
mov     QWORD PTR [rbp-40], rdi
mov     rdx, QWORD PTR [rbp-40]
mov     rax, QWORD PTR [rbp-40]
mov     rsi, rdx
mov     rdi, rax
call   operator*(Complex const&, Complex const&)
movq   rax, xmm0
movapd xmm0, xmm1
mov     QWORD PTR [rbp-32], rax
movsd  QWORD PTR [rbp-24], xmm0
mov     rdx, QWORD PTR [rbp-40]
lea    rax, [rbp-32]
mov     rsi, rdx
mov     rdi, rax
call   operator*(Complex const&, Complex const&)
mov     eax, 0
mov     edx, 0
movq   rax, xmm0
movq   rdx, xmm1
mov     rcx, rax
mov     rbx, rdx
movq   xmm0, rcx
movq   xmm1, rdx
add    rsp, 40
pop    rbx
pop    rbp
ret
```

C-macro code calls
sqrt() 3x and
operator*() 2x.

stupidmacro():

```
push    rbp
mov     rbp, rsp
sub     rsp, 80
movsd  xmm0, QWORD PTR .LC2[rip]
movsd  QWORD PTR [rbp-80], xmm0
movsd  xmm0, QWORD PTR .LC3[rip]
movsd  QWORD PTR [rbp-72], xmm0
lea    rax, [rbp-80]
mov    rdi, rax
call   sqrt(Complex const&)
movq   rax, xmm0
movapd xmm0, xmm1
mov    QWORD PTR [rbp-64], rax
movsd  QWORD PTR [rbp-56], xmm0
lea    rax, [rbp-80]
mov    rdi, rax
call   sqrt(Complex const&)
movq   rax, xmm0
movapd xmm0, xmm1
mov    QWORD PTR [rbp-32], rax
movsd  QWORD PTR [rbp-24], xmm0
lea    rax, [rbp-80]
mov    rdi, rax
call   sqrt(Complex const&)
movq   rax, xmm0
movapd xmm0, xmm1
mov    QWORD PTR [rbp-16], rax
movsd  QWORD PTR [rbp-8], xmm0
lea    rdx, [rbp-32]
lea    rax, [rbp-16]
mov    rsi, rdx
mov    rdi, rax
call   operator*(Complex const&, Complex const&)
movq   rax, xmm0
movapd xmm0, xmm1
mov    QWORD PTR [rbp-48], rax
movsd  QWORD PTR [rbp-40], xmm0
lea    rdx, [rbp-64]
lea    rax, [rbp-48]
mov    rsi, rdx
mov    rdi, rax
call   operator*(Complex const&, Complex const&)
```

Summary

- Compiler Explorer and Cpp Insights make differences among compilers, compiler options and arches easier to understand.
- Pasting code into them is fast and painless.