„final“ in C++ 11

Detlef Wilkening

11.1.2018

- **Seit C++ 11 gibt es das neue *Schlüsselwort* "final"**
  - Okay, kein richtiges Schlüsselwort – aber dazu später noch mehr
- **Man kann nun Klassen und virtuelle Funktionen "final" deklarieren**
  - Damit sind sie „final“, daher:
    - man kann sich von der Klasse nicht mehr ableiten
    - man kann die Funktion nicht mehr überschreiben

- **Seit C++ 11 gibt es das neue *Schlüsselwort* "final"**
  - Okay, kein richtiges Schlüsselwort – aber dazu später noch mehr
- **Man kann nun Klassen und virtuelle Funktionen "final" deklarieren**
  - Damit sind sie „final“, daher:
    - man kann sich von der Klasse nicht mehr ableiten
    - man kann die Funktion nicht mehr überschreiben
- **Fertig**

- **Seit C++ 11 gibt es das neue *Schlüsselwort* "final"**
  - Okay, kein richtiges Schlüsselwort – aber dazu später noch mehr
- **Man kann nun Klassen und virtuelle Funktionen "final" deklarieren**
  - Damit sind sie „final“, daher:
    - man kann sich von der Klasse nicht mehr ableiten
    - man kann die Funktion nicht mehr überschreiben
- **Fertig**
- **Okay, schauen wir es uns noch etwas genauer an**
  - Aber *"viel mehr"* gibt es dazu eigentlich nicht zu sagen

```
struct A final
{
};

struct B : A       // Error
{
};
```

```
struct A
{
};

struct B final : A
{
};

struct C : B       // Error
{
};
```

```
struct A
{
   virtual void f() final;
};

struct B : A
{
   void f() override;  // Error
};
```

```
struct A
{
   virtual void f();
};

struct B : A
{
   void f() final;
};

struct C : B
{
   void f() override;  // Error
};
```

- **Nur virtuelle Funktionen dürfen „final“ deklariert werden**

```
struct A
{
   void f() final;     // Error
};
```

```
struct A
{
   void f();
};

struct B : A
{
   void f() final;     // Error
};
```

- **C++ Core Guidelines**
  - C.128: Virtual functions should specify exactly one of virtual, override, or final

```
struct A
{
   virtual void f() { cout << 'A'; }
};

struct B : A
{
   void f() override { cout << 'B'; }
};

struct C : B
{
   void f() final { cout << 'C'; }
};
```

```
int main()
{
   A* p = nullptr;

   A a;
   p = &a;
   p->f();      // => A

   B b;
   p = &b;
   p->f();      // => B

   C c;
   p = &c;
   p->f();      // => C
}
```

- **Was soll „final“ aber?**
- **Immerhin erstellt man Klassen und virtuelle Funktionen um sich davon abzuleiten bzw. sie zu überschreiben!**

- **Was soll „final“ aber?**
- **Immerhin erstellt man Klassen und virtuelle Funktionen um sich davon abzuleiten bzw. sie zu überschreiben!**

- **=>**

- **1. Man kann Nicht-Basisklassen noch expliziter ausdrücken**
- **2. Man kann innerhalb eines Moduls ableiten oder überschreiben, ohne dass dies Teil der Schnittstelle wird**
- **3. Performance – Compiler kann devirtualisieren**

- **1. Man kann Nicht-Basisklassen noch expliziter ausdrücken**
  - Basis-Klassen:
    - Klasse enthält virtuelle Funktionen
      - Typischerweise u.a. einen virtuellen Destruktor
    - Klasse enthält nur Typen, die über Template-Argumente definiert werden
      - Beispiel std::unary_function (deprecated in C++11, removed in C++17)

```
struct A
{
   virtual ~A() = default;
   virtual void f() = 0;
};

template <class Arg, class Result> struct unary_function
{
   typedef Arg argument_type;
   typedef Result result_type;
};
```

- **1. Man kann Nicht-Basisklassen noch expliziter ausdrücken**
  - Eigentlich sollte sich niemand von Klassen ableiten, die klar nicht als Basisklasse gedacht sind, aber…
  - Mit „final“ ist dies noch klarer
  - Und der Compiler meckert, wenn es jemand doch macht…

```
struct A final       // Ganz klar, man soll und kann sich nicht ableiten
{
   void f();
};
```

- **2. Man kann innerhalb eines Moduls ableiten oder überschreiben, ohne dass dies Teil der Schnittstelle wird**
  - Verhindert das Ableiten von einer modul-internen Klasse, die direkt genutzt werden soll, aber nicht zum Ableiten frei gegeben ist

```
namespace module
{
   struct A
   {
      virtual void f() { cout << 'A'; }
   };

   struct B final : A
   {
      void f() override { cout << 'B'; }
   };
}
struct C : module::B          // Error - verhindert das Ableiten
{
};
```

- **2. Man kann innerhalb eines Moduls ableiten oder überschreiben, ohne dass dies Teil der Schnittstelle wird**
  - Verhindert das Überschreiben von modul-internen Funktionen, das die Funktionalität des Moduls brechen könnte, ohne das man sich nicht ableiten oder Funktionen überschreiben kann

```
namespace module
{
   struct A
   {
      virtual void fintern() = 0;
   };
   struct B : A
   {
      void fintern() final
         { cout << 'I'; }
      virtual void fextern()
        { fintern(); }
   };
}
```

```
namespace application
{
  struct C : module::B
  {
    void fintern() override; // Error

    void fextern() override
    {
      // okay
    }
  };
}
```

- **Beispiel**
  - Bibliothek für eine grafische Oberfläche „Fly“
  - Views haben Funktionen für GetFocus und LooseFocus
    - Diese Funktionen dürfen / sollen überschrieben werden
    - Wenn der Nutzer darauf reagieren möchte
  - Intern müssen die Focus-Events des Systems behandelt werden
    - Windows WM_SETFOCUS und WM_KILLFOCUS müssen gehandelt werden
    - Hier soll der Anwender aber nicht eingreifen dürfen

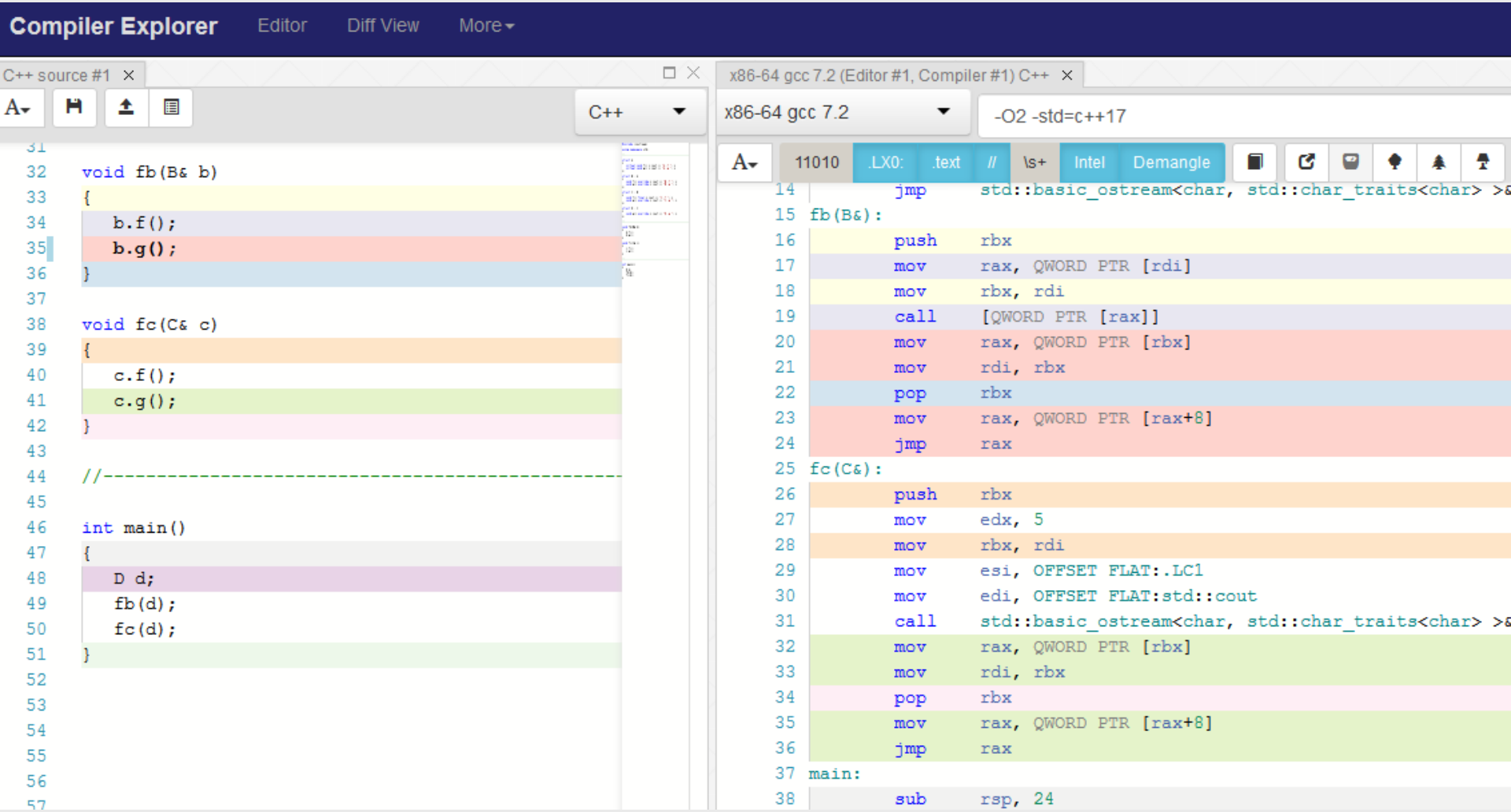
```
namespace Fly
{
   struct Control
   {
      virtual void execWmSetFocus(bool);
      virtual void execWmSetFocusChild(Control&, bool);
      virtual void execWmKillFocus(Control*);
      virtual void execWmKillFocusChild(Control*);
      ...
   };
   struct View : Control
   {
      void execWmSetFocus(bool) final;
      void execWmSetFocusChild(Control&, bool) final;
      void execWmKillFocus(Control*) final;
      void execWmKillFocusChild(Control*) final;
      virtual void getFocus(Control&) {}
      virtual void looseFocus(Control&) {}
      ...
   };
}
```

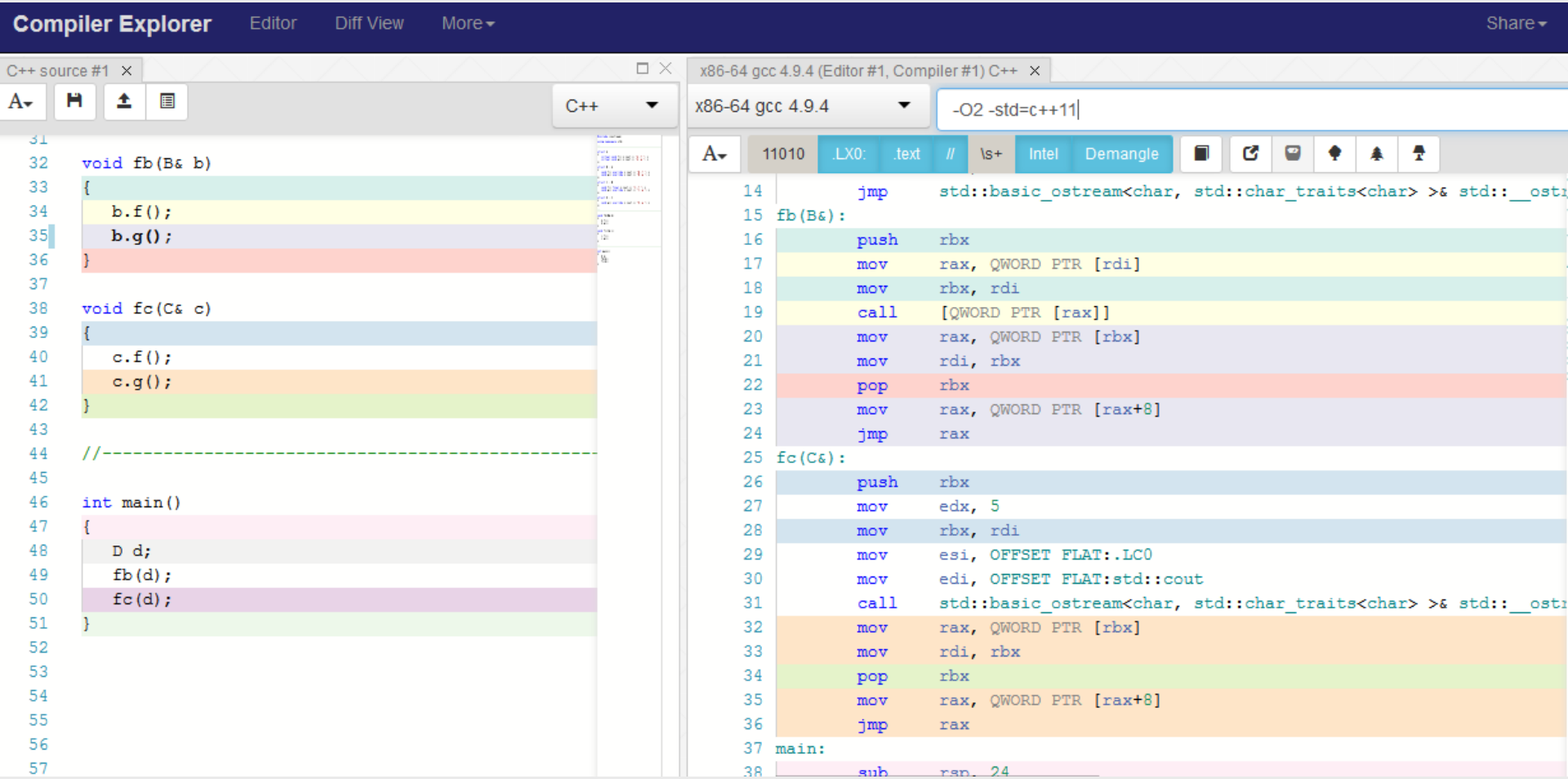
- **3. Performance**
  - Ein virtueller Funktions-Aufruf ist langsamer als ein normaler Aufruf
  - Ein virtueller Funktions-Aufruf kann nicht *„geinlinet“* werden
  - Aber wenn eine Klasse *„final“* ist bzw. eine Funktion *„final“* ist
  - Dann kann der Funktions-Aufruf devirtualisiert werden
  - Der Compiler kann also wieder für optimale Performance compilieren

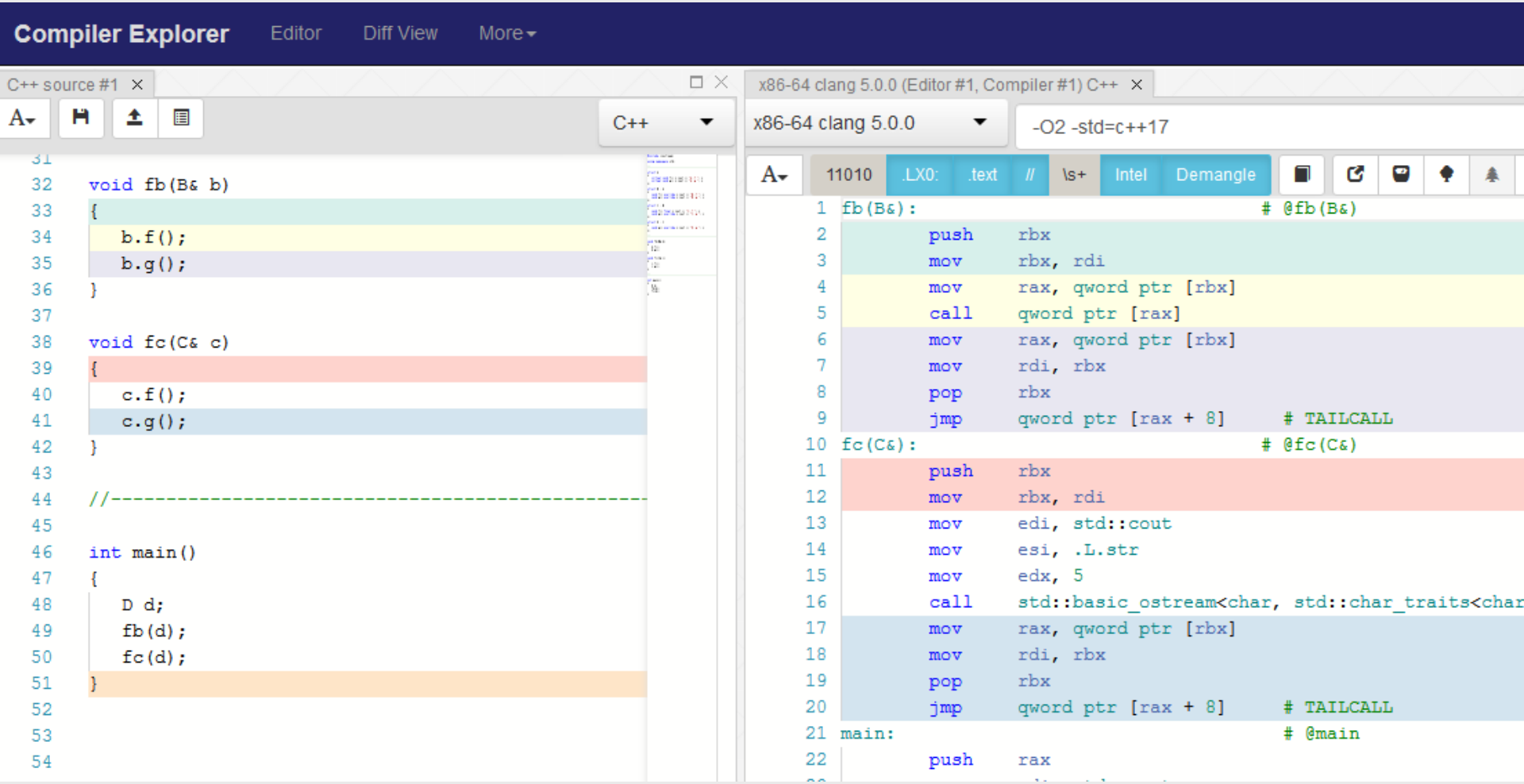
- **3. Performance**
  - Ein virtueller Funktions-Aufruf ist langsamer als ein normaler Aufruf
  - Ein virtueller Funktions-Aufruf kann nicht *„geinlinet“* werden
  - Aber wenn eine Klasse *„final“* ist bzw. eine Funktion *„final“* ist
  - Dann kann der Funktions-Aufruf devirtualisiert werden
  - Der Compiler kann also wieder für optimale Performance compilieren
  - Aber macht er das auch?
  - Fragen wir doch mal den Compiler-Explorer!
    - Test-Beispiel auf der folgenden Folie
    - GCC 7.2 -O2 –std=c++17
    - GCC 4.9.4 -O2 –std=c++11
    - Clang 5.0.0 -O2 –std=c++17
    - Clang 3.9.0 -O2 –std=c++11
    - Microsoft Visual Studio 2017 /O2
  - Was meint ihr – was kommt da wohl raus?

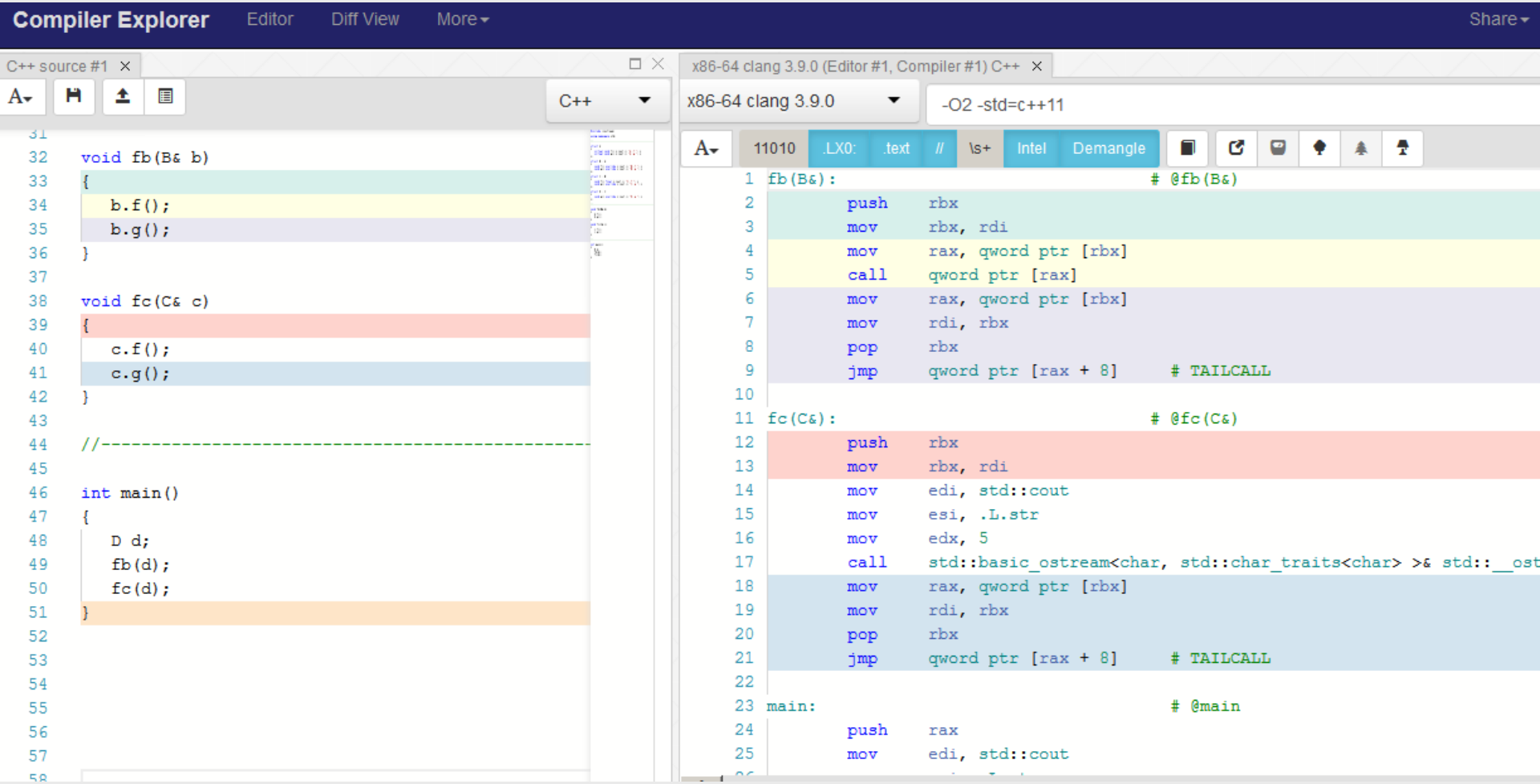
```
struct A
{
   virtual void f() { cout << "A::f"; }
   virtual void g() { cout << "A::g"; }
};

struct B : A
{
   void f() override { cout << "B::f"; }
   void g() override { cout << "B::g"; }
};

struct C : B
{
   void f() final    { cout << "C::f"; }
   void g() override { cout << "C::g"; }
};

struct D : C
{
   void g() override { cout << "D::g"; }
};
```

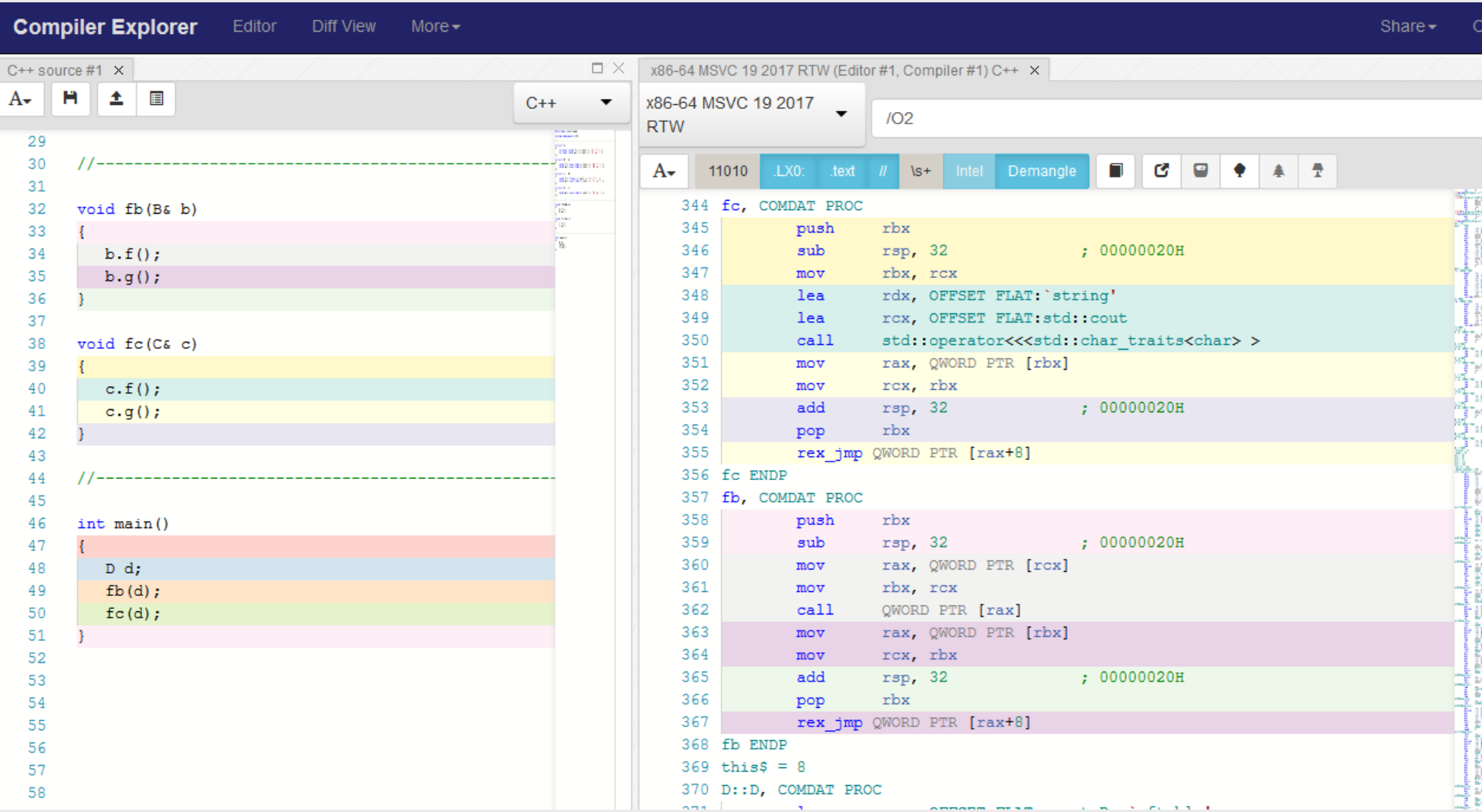
```
void fb(B& b)
{
   b.f();
   b.g();
}

void fc(C& c)
{
   c.f();
   c.g();
}

int main()
{
   D d;
   fb(d);
   fc(d);
   cout << endl;
}
```


Compiler Explorer
Editor
Diff View
More
C++ source #1
C++
x86-64 gcc 7.2 (Editor #1, Compiler #1) C++
x86-64 gcc 7.2
-O2 -std=c++17
11010
.LX0:
.text
//
\s+
Intel
Demangle
32 void fb(B& b)
33 {
34 b.f();
35 b.g();
36 }
38 void fc(C& c)
39 {
40 c.f();
41 c.g();
42 }
44 //----------
46 int main()
47 {
48 D d;
49 fb(d);
50 fc(d);
51 }
14 jmp std::basic_ostream<char, std::char_traits<char> >&
15 fb(B&):
16 push rbx
17 mov rax, QWORD PTR [rdi]
18 mov rbx, rdi
19 call [QWORD PTR [rax]]
20 mov rax, QWORD PTR [rbx]
21 mov rdi, rbx
22 pop rbx
23 mov rax, QWORD PTR [rax+8]
24 jmp rax
25 fc(C&):
26 push rbx
27 mov edx, 5
28 mov rbx, rdi
29 mov esi, OFFSET FLAT:.LC1
30 mov edi, OFFSET FLAT:std::cout
31 call std::basic_ostream<char, std::char_traits<char> >&
32 mov rax, QWORD PTR [rbx]
33 mov rdi, rbx
34 pop rbx
35 mov rax, QWORD PTR [rax+8]
36 jmp rax
37 main:
38 sub rsp, 24


Compiler Explorer
Editor
Diff View
More
Share
C++ source #1
C++
x86-64 gcc 4.9.4 (Editor #1, Compiler #1) C++
x86-64 gcc 4.9.4
-O2 -std=c++11
11010
.LX0:
.text
//
\s+
Intel
Demangle
void fb(B& b)
{
b.f();
b.g();
}
void fc(C& c)
{
c.f();
c.g();
}
//----------
int main()
{
D d;
fb(d);
fc(d);
}
jmp std::basic_ostream<char, std::char_traits<char> >& std::__osti
fb(B&):
push rbx
mov rax, QWORD PTR [rdi]
mov rbx, rdi
call [QWORD PTR [rax]]
mov rax, QWORD PTR [rbx]
mov rdi, rbx
pop rbx
mov rax, QWORD PTR [rax+8]
jmp rax
fc(C&):
push rbx
mov edx, 5
mov rbx, rdi
mov esi, OFFSET FLAT:.LC0
mov edi, OFFSET FLAT:std::cout
call std::basic_ostream<char, std::char_traits<char> >& std::__osti
mov rax, QWORD PTR [rbx]
mov rdi, rbx
pop rbx
mov rax, QWORD PTR [rax+8]
jmp rax
main:
sub rsp, 24


Compiler Explorer
Editor
Diff View
More
C++ source #1
C++
x86-64 clang 5.0.0 (Editor #1, Compiler #1) C++
x86-64 clang 5.0.0
-O2 -std=c++17
11010
.LX0:
.text
//
\s+
Intel
Demangle
void fb(B& b)
{
b.f();
b.g();
}
void fc(C& c)
{
c.f();
c.g();
}
//----------
int main()
{
D d;
fb(d);
fc(d);
}
fb(B&): # @fb(B&)
push rbx
mov rbx, rdi
mov rax, qword ptr [rbx]
call qword ptr [rax]
mov rax, qword ptr [rbx]
mov rdi, rbx
pop rbx
jmp qword ptr [rax + 8] # TAILCALL
fc(C&): # @fc(C&)
push rbx
mov rbx, rdi
mov edi, std::cout
mov esi, .L.str
mov edx, 5
call std::basic_ostream<char, std::char_traits<char
mov rax, qword ptr [rbx]
mov rdi, rbx
pop rbx
jmp qword ptr [rax + 8] # TAILCALL
main: # @main
push rax


Compiler Explorer
Editor
Diff View
More
Share
C++ source #1
C++
x86-64 clang 3.9.0 (Editor #1, Compiler #1) C++
x86-64 clang 3.9.0
-O2 -std=c++11
11010
.LX0:
.text
//
\s+
Intel
Demangle
void fb(B& b)
{
b.f();
b.g();
}
void fc(C& c)
{
c.f();
c.g();
}
//----------
int main()
{
D d;
fb(d);
fc(d);
}
fb(B&): # @fb(B&)
push rbx
mov rbx, rdi
mov rax, qword ptr [rbx]
call qword ptr [rax]
mov rax, qword ptr [rbx]
mov rdi, rbx
pop rbx
jmp qword ptr [rax + 8] # TAILCALL
fc(C&): # @fc(C&)
push rbx
mov rbx, rdi
mov edi, std::cout
mov esi, .L.str
mov edx, 5
call std::basic_ostream<char, std::char_traits<char> >& std::__ost
mov rax, qword ptr [rbx]
mov rdi, rbx
pop rbx
jmp qword ptr [rax + 8] # TAILCALL
main: # @main
push rax
mov edi, std::cout


Compiler Explorer
Editor
Diff View
More
Share
C++ source #1
C++
x86-64 MSVC 19 2017 RTW (Editor #1, Compiler #1) C++
x86-64 MSVC 19 2017 RTW
/O2
11010
.LX0:
.text
//
\s+
Intel
Demangle
29
30 //--
31
32 void fb(B& b)
33 {
34 b.f();
35 b.g();
36 }
37
38 void fc(C& c)
39 {
40 c.f();
41 c.g();
42 }
43
44 //--
45
46 int main()
47 {
48 D d;
49 fb(d);
50 fc(d);
51 }
52
53
54
55
56
57
58
344 fc, COMDAT PROC
345 push rbx
346 sub rsp, 32 ; 00000020H
347 mov rbx, rcx
348 lea rdx, OFFSET FLAT:`string'
349 lea rcx, OFFSET FLAT:std::cout
350 call std::operator<<<std::char_traits<char> >
351 mov rax, QWORD PTR [rbx]
352 mov rcx, rbx
353 add rsp, 32 ; 00000020H
354 pop rbx
355 rex_jmp QWORD PTR [rax+8]
356 fc ENDP
357 fb, COMDAT PROC
358 push rbx
359 sub rsp, 32 ; 00000020H
360 mov rax, QWORD PTR [rcx]
361 mov rbx, rcx
362 call QWORD PTR [rax]
363 mov rax, QWORD PTR [rbx]
364 mov rcx, rbx
365 add rsp, 32 ; 00000020H
366 pop rbx
367 rex_jmp QWORD PTR [rax+8]
368 fb ENDP
369 this$ = 8
370 D::D, COMDAT PROC

- **Okay, das war mehr als erwartet, oder?**
- **Gibt es etwa noch mehr?**

- **Okay, das war mehr als erwartet, oder?**
- **Gibt es etwa noch mehr?**
- **Ja!**

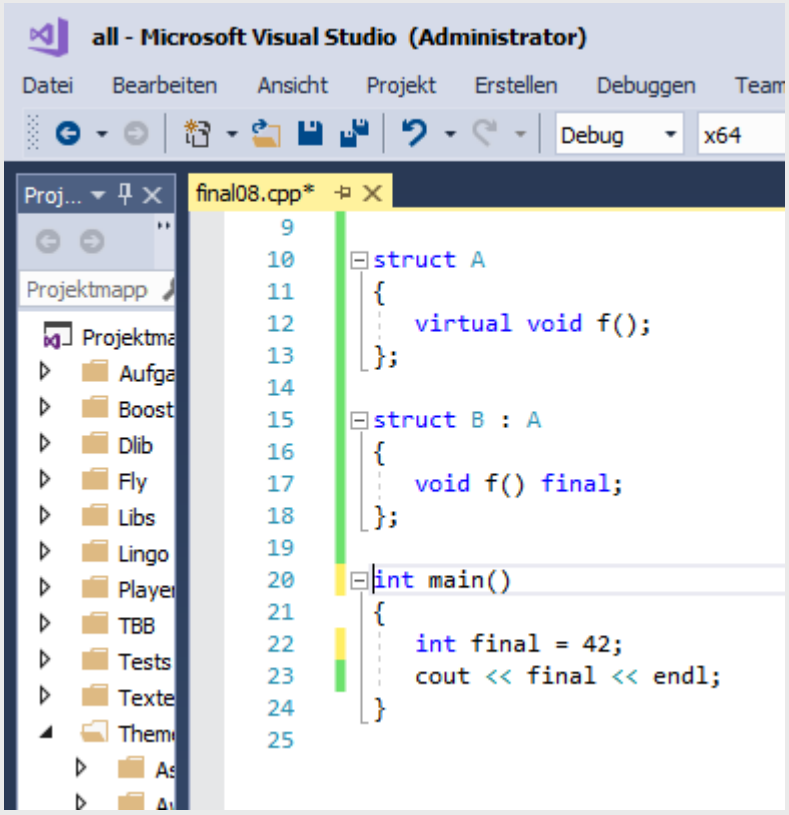
- **Okay, das war mehr als erwartet, oder?**
- **Gibt es etwa noch mehr?**
- **Ja!**

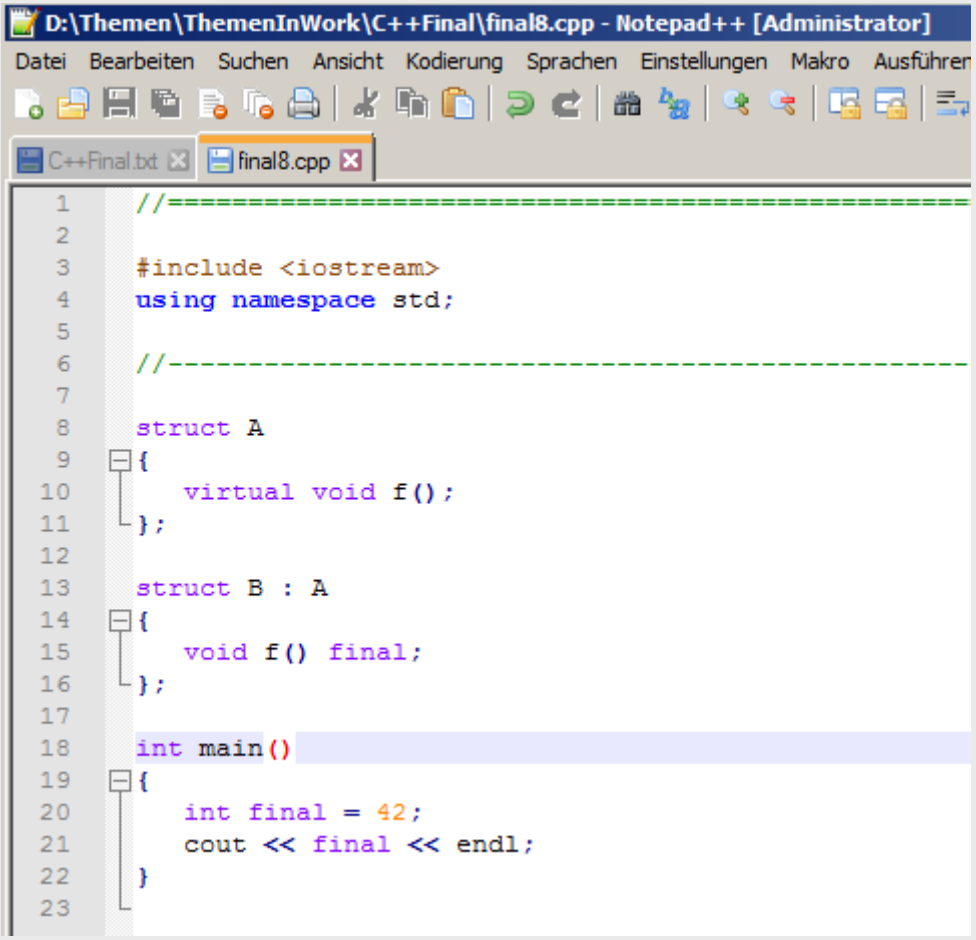
1. **„final“ ist kein Schlüsselwort der Sprache**

- **„final“ ist kein Schlüsselwort der Sprache**
  - Es ist ein fast normaler Bezeichner
  - Hat aber in bestimmten Kontexten eine besondere Bedeutung
    - C++11 Standard
      § 2.11, Item 2 und Tabelle 3
      ISO/IEC 14882, Third edition, 2011-09-01
- **=> „final" kann z.B. als Variablen-Namen eingesetzt werden**
  - Hintergrund ist das C++11 keinen alten C++ Code brechen wollte
  - Hinweis
    - Gilt genauso für „override“

```
int main()
{
   int final = 42;                    // Funktioniert
   cout << final << endl;
}
```

- **Das ist übrigens ein interessanter Test für ihren Editor und sein Syntax-Highligthing**
- **Kann er damit umgehen?**
- **Wie schlägt sich da ihr Editor?**


all - Microsoft Visual Studio (Administrator)
Datei Bearbeiten Ansicht Projekt Erstellen Debuggen
Debug
x64
final08.cpp*
Projektmapp
9
10 struct A
11 {
12 virtual void f();
13 };
14
15 struct B : A
16 {
17 void f() final;
18 };
19
20 int main()
21 {
22 int final = 42;
23 cout << final << endl;
24 }
25


D:\Themen\ThemenInWork\C++Final\final8.cpp - Notepad++ [Administrator]
Datei Bearbeiten Suchen Ansicht Kodierung Sprachen Einstellungen Makro
C++Final.txt
final8.cpp
1 //=====
2
3 #include <iostream>
4 using namespace std;
5
6 //-----
7
8 struct A
9 {
10 virtual void f();
11 };
12
13 struct B : A
14 {
15 void f() final;
16 };
17
18 int main()
19 {
20 int final = 42;
21 cout << final << endl;
22 }
23

- **Eine Konsequenz daraus ist übrigens:**
  - Der Präprozessor beachtet die Sprach-Kontexte nicht
  - Daher könnte ein Präprozessor-Makro „final“ zu Problemen führen
- **=>**
  - C++11 verbietet daher explizit den Bezeichner „final" als Präprozessor-Makro
    - C++11 Standard
      C.2.7, Clause 17, Item 17.6.5.3
      Zusätzliche Beschränkungen von Makro-Namen
      ISO/IEC 14882, Third edition, 2011-09-01
  - Hinweis
    - Gilt natürlich wieder genauso für „override“

- **Okay, das war mehr als erwartet, oder?**
- **Gibt es etwa noch mehr?**
- **Ja!**

1. **„final“ ist kein Schlüsselwort der Sprache**
2. **Unbenamte Klassen und „final“**

- **Ein unbenamte Klasse kann nicht final sein**
  - C++14 Standard
    9, Item 1, Note
    ISO/IEC 14882, Fourth edition, 2014-12-15
- **Mein Problem – wie mache ich eine „unnamed class“ final?**

- **Ein unbenamte Klasse kann nicht final sein**
  - C++14 Standard
    9, Item 1, Note
    ISO/IEC 14882, Fourth edition, 2014-12-15
- **Mein Problem – wie mache ich eine „unnamed class“ final?**
  - Ist doch ganz einfach!

```
struct final
{
   void f() {}
} x;
```

- **Ein unbenamte Klasse kann nicht final sein**
  - C++14 Standard
    9, Item 1, Note
    ISO/IEC 14882, Fourth edition, 2014-12-15
- **Mein Problem – wie mache ich eine „unnamed class“ final?**
  - Ist doch ganz einfach!
  - Nein, es ist nicht so einfach – das ist eine Klasse mit dem Namen „final“

```
struct final
{
   void f() {}
} x;

int main()
{
   final y;           // Alles okay
}
```

- **Ein unbenamte Klasse kann nicht final sein**
  - C++14 Standard
    9, Item 1, Note
    ISO/IEC 14882, Fourth edition, 2014-12-15
- **Stimmt schon**
- **Sie kann nicht final sein**
- **Nicht: sie darf nicht final sein**

- **Okay, das war mehr als erwartet, oder?**
- **Gibt es etwa noch mehr?**
- **Ja!**

1. **„final“ ist kein Schlüsselwort der Sprache**
2. **Unbenamte Klassen und „final“**
3. **Destruktoren**

- **Dürfen Destruktoren final sein?**
- **Ja, sie dürfen**

- **Dürfen Destruktoren final sein?**
- **Ja, sie sie dürfen**
- **Damit ist aber letztlich die Klasse „final“**
  - Sollte man das doch machen – die Klasse mit "final" deklarieren

```
struct A
{
   virtual ~A() = default;
};

struct B : A
{
   ~B() final = default;
};

struct C : B          // Compiler-Error
{
};
```

- **Okay, das war mehr als erwartet, oder?**
- **Gibt es etwa noch mehr?**
- **Ja!**

1. **„final“ ist kein Schlüsselwort der Sprache**
2. **Unbenamte Klassen und „final“**
3. **Destruktoren**
4. **Type-Traits**

- **Kann man abfragen, ob eine Klasse „final“ ist?**
- **Ja**
  - Header <type_traits>
  - template <class T> struct is_final;
    - T muss ein kompletter Typ sein
- **Elementare Datentypen und so gelten übrigens nicht als final**
  - Auch wenn man sich von ihnen nicht ableiten kann

```
#include <type_traits>
using namespace std;

struct A {};

struct B final {};

static_assert(!is_final<int>::value, "int ist NICHT final");
static_assert(!is_final<A>::value, "A ist NICHT final");
static_assert(is_final<B>::value, "B ist final");
```

- **Okay, das war mehr als erwartet, oder?**
- **Gibt es etwa noch mehr?**
- **Ja!**

1. **„final“ ist kein Schlüsselwort der Sprache**
2. **Unbenamte Klassen und „final“**
3. **Destruktoren**
4. **Type-Traits**
5. **Quantitäten**

- **Implementation quantities**
  - C++14 Standard
    Annex B (informative)
    ISO/IEC 14882, Fourth edition, 2014-12-15
- **Ein Compiler muss mindestens 16.384 final overriding virtual Funktionen in einer Klasse unterstützten**

- **Okay, das war mehr als erwartet, oder?**
- **Gibt es etwa noch mehr?**
- **Ja!**

1. **„final“ ist kein Schlüsselwort der Sprache**
2. **Unbenamte Klassen und „final“**
3. **Destruktoren**
4. **Type-Traits**
5. **Quantitäten**
6. **Und wie war das Leben in Vor-C++ 11 Zeiten?**

- **Und wie war das Leben in Vor-C++ 11 Zeiten?**
- **Konnte man sich helfen?**
- **Konnte man „final“ simulieren?**

- **Und wie war das Leben in Vor-C++ 11 Zeiten?**
- **Konnte man sich helfen?**
- **Konnte man „final“ simulieren?**

- **Ja, für Klassen ging das:**
  - Virtuelle Basis-Klasse nutzen
  - Siehe Code auf der nächsten Folie

- **Virtuelle Basisklassen müssen zuerst erzeugt werden**
- **B kann „Final()“ nicht aufrufen**
  - Denn „Final()“ ist protected
  - Und A leitet sich virtual private von Final ab

```
class Final
{
protected:
   Final() = default;
   virtual ~Final() = default;
};

struct A : virtual private Final
{
};

struct B : A                                    // Compiler-Error
{
};
```

- Okay, das war mehr als erwartet, oder?
- Gibt es etwa noch mehr?
- Ja!

1. „final“ ist kein Schlüsselwort der Sprache
2. Unbenamte Klassen und „final“
3. Destruktoren
4. Type-Traits
5. Quantitäten
6. Und wie war das Leben in Vor-C++ 11 Zeiten?
7. Leere Objekte – ein Blick in die Zukunft

- **Empty Base Class Optimization (EBO)**

```
struct Base
{
};

struct Derived : Base
{
   int i;
};

// Die Groesse jedes Typs ist mindestens 1
static_assert(sizeof(Base) >= 1, "");

// EBO erlaubt die Entfernung einer leeren Basis-Klasse
static_assert(sizeof(Derived) == sizeof(int));
```

- **Empty Base Class Optimization (EBO)**
  - http://en.cppreference.com/w/cpp/language/ebo
- **Was hat das mit "final" zu tun?**

```
struct Base
{
};

struct Derived : Base
{
   int i;
};

// Die Groesse jedes Typs ist mindestens 1
static_assert(sizeof(Base) >= 1, "");

// EBO erlaubt die Entfernung einer leeren Basis-Klasse
static_assert(sizeof(Derived) == sizeof(int));
```

- **Empty Base Class Optimization (EBO)**
- **Was hat das mit “final” zu tun?**
- **EBO hat Grenzen bzw. erzeugt einige Probleme**
  - U.a. funktioniert EBO nicht bei „final“ Klassen
  - Denn von denen kann man sich nicht ableiten
  - Also muss man aus der Basisklasse einen Member machen
  - Aber Member haben keine Empty Class Optimization

```
struct Base final
{
};

struct D
{
   Base b;          // final => nur noch als Member moeglich
   int i;
};

static_assert(sizeof(D) > sizeof(int));   // D ist groesser als ein int
```

- **Attribute haben keine Empty Class Optimization**
- **Darum gibt es ein Proposal für “empty member optimization”**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0840r0.html
- **Noch Zukunft, aber vielleicht C++20**

```
struct Base final
{
};

struct D
{
   [[no_unique_address]] Base b;              // Vielleicht C++20
   int i;
};

static_assert(sizeof(D) == sizeof(int));      // Und wieder gleich gross
```

- Okay, das war mehr als erwartet, oder?
- Gibt es etwa noch mehr?
- Ja!

1. „final“ ist kein Schlüsselwort der Sprache
2. Unbenamte Klassen und „final“
3. Destruktoren
4. Type-Traits
5. Quantitäten
6. Und wie war das Leben in Vor-C++ 11 Zeiten?
7. Leere Objekte – ein Blick in die Zukunft

- Nein! Nicht mehr – das war es!

Links

- http://en.cppreference.com/w/cpp/language/final
- http://en.cppreference.com/w/cpp/language/override

- https://arne-mertz.de/2017/04/final-classes/
- https://akrzemi1.wordpress.com/2012/09/30/why-make-your-classes-final/
- http://www.modernescpp.com/index.php/override-and-final
- https://arne-mertz.de/2015/12/modern-c-features-override-and-final/
- http://arne-mertz.de/2015/12/modern-c-features-override-and-final/
- http://foonathan.github.io/blog/2016/05/27/final.html
- https://blog.feabhas.com/2015/09/bitesize-modern-c-override-and-final/

- http://www.bfilipek.com/2017/04/finalact.html
- http://741mhz.com/final-override/
- http://bannalia.blogspot.de/2014/05/fast-polymorphic-collections-with.html
- https://codeyarns.com/2016/11/21/override-and-final-in-c/
- http://www.bogotobogo.com/cplusplus/C11/C11_override_final.php
- https://www.nosid.org/cxx11-virtual-function-specifiers.html
- http://marcofoco.com/final-override-again/
- http://katecpp.github.io/override-and-final/
- https://www.heise.de/developer/artikel/C-Core-Guidelines-Klassenhierarchien-3852049.html

- https://stackoverflow.com/questions/47556287/may-a-destructor-be-final
- https://stackoverflow.com/questions/29412412/does-final-imply-override
- https://stackoverflow.com/questions/8824587/what-is-the-purpose-of-the-final-keyword-in-c11-for-functions
- https://stackoverflow.com/questions/11704406/whats-the-point-of-a-final-virtual-function
- https://stackoverflow.com/questions/36895012/does-c-final-imply-final-in-all-aspects
- https://stackoverflow.com/questions/44153281/does-it-make-sense-to-add-final-keyword-to-the-virtual-function-in-a-class-that
- https://stackoverflow.com/questions/44717803/c-final-or-sealed

- https://stackoverflow.com/questions/16808184/use-cases-for-final-classes
- https://stackoverflow.com/questions/41637559/why-are-c11-override-and-final-not-attributes
- https://stackoverflow.com/questions/37414995/is-final-used-for-optimization-in-c

final ;-)