redhat.

declared as defaulted, defined as deleted

Stephan Bergmann

LibreOffice Conference, September 2018

C++ Quiz Time

defaulted & deleted

- What functions can be defined as deleted?

defaulted & deleted

- What functions can be defined as deleted?
  - Any functions

```
void f() = delete;
```

defaulted & deleted

- What functions can be defined as deleted?
  - Any functions

```
void f() = delete;
```

- What functions can be declared as defaulted?

defaulted & deleted

- What functions can be defined as deleted?
  - Any functions

```
void f() = delete;
```

- What functions can be declared as defaulted?
  - Only some special member functions

```
struct S {
    S() = default;
    ~S() = default;
    S(S const &) = default;
    S(S &&) = default;
    S & operator =(S const &) = default;
    S & operator =(S &&) = default;
};
```

defaulted & deleted

- Can a function be both defaulted and deleted?

defaulted & deleted

- Can a function be both defaulted and deleted?
  - Yes, (implicitly) declared as defaulted, and implicitly defined as deleted

```
struct S {
    std::unique_ptr<...> m;
    S(S const &) = default;
};
```

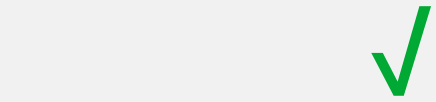
defaulted & deleted

- Can a function be only defined as defaulted (not declared as defaulted)?

defaulted & deleted

- Can a function be only defined as defaulted (not declared as defaulted)?
  - Yes

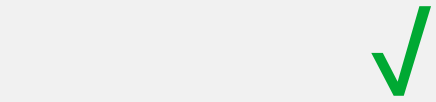
```
// .hxx:
struct S { S(); };

// .cxx:
S::S() = default;
```



defaulted & deleted

- Can a function be only defined as defaulted (not declared as defaulted)?
  - Yes

```
// .hxx:
struct S { S(); };

// .cxx:
S::S() = default;
```

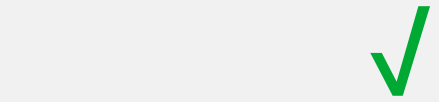


- And does that also work for deleted functions?

defaulted & deleted

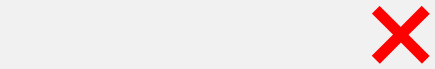
- Can a function be only defined as defaulted (not declared as defaulted)?
  - Yes

```
// .hxx:
struct S { S(); };

// .cxx:
S::S() = default;
```



- And does that also work for deleted functions?
  - No

```
// .hxx:
struct S { S(); }

// .cxx:
S::S() = delete;
```



Why do you ask?

- Because GCC 9 has -Werror=deprecated-copy
  - Implicitly defaulted copy functions are deprecated when a class has any user-declared copy function or destructor

```
struct S { ~S(); };
S a, b;
a = b;  // will eventually stop compiling
```

  - Lots of boilerplate added:

```
struct S {
    virtual ~S() {}
    S() = default;
    S(S const &) = default;
    S(S &&) = default;
    S & operator =(S const &) = default;
    S & operator =(S &&) = default;
};
```

inline

- What is an inline function?

inline

- What is an inline function?
  - One that can be defined in multiple TUs
    - Which is especially useful if it contains static local variables:

```
inline int counter() { static int n = 0; return n++; }
```

inline

- What is an inline function?
  - One that can be defined in multiple TUs
    - Which is especially useful if it contains static local variables:

```
inline int counter() { static int n = 0; return n++; }
```

- What is an inline variable?

inline

- What is an inline function?
  - One that can be defined in multiple TUs
    - Which is especially useful if it contains static local variables:

```
inline int counter() { static int n = 0; return n++; }
```

- What is an inline variable?
  - One that can be defined in multiple TUs

inline

- What is an inline function?
  - One that can be defined in multiple TUs
    - Which is especially useful if it contains static local variables:

```
inline int counter() { static int n = 0; return n++; }
```

- What is an inline variable?
  - One that can be defined in multiple TUs
    - Which is especially useful for constexpr static data members:

```
struct S { static constexpr OUStringLiteral magic("x"); }
// no extra: OUStringLiteral S::magic;
```

inline

- What is an inline namespace?

inline

- What is an inline namespace?
  - Something completely different

Why do you ask?

- Because e.g. “loplugin:constfields in xmloff”:

```
 private:
-    OUString m_aColorPropName;
+    static constexpr OUStringLiteral g_aColorPropName = "FillColor";
     Property m_aColorProp;
 };

+#if !HAVE_CPP_INLINE_VARIABLES
+constexpr OUStringLiteral lcl_ColorPropertySetInfo::g_aColorPropName;
+#endif
```

copy/move

- Does this compile (in C++17)?

```
struct S {
    S();
    S(S &) = delete;
};
S f() { return S(); }
S s = f();
```

copy/move

- Does this compile (in C++17)?

```
struct S {
    S();
    S(S &) = delete;
};
S f() { return S(); }
S s = f();
```

copy/move

- Does this compile (in C++17)?

```
struct S {
    S();
    S(S &) = delete;
};
S f() { return S(); }
S s = f();
```

- And does this compile?

```
struct S {
    S();
    S(S &) = delete;
};
S f() { S s; return s; }
S s = f();
```

copy/move

- Does this compile (in C++17)?

```
struct S {
    S();
    S(S &) = delete;
};
S f() { return S(); }
S s = f();
```

- And does this compile?

```
struct S {
    S();
    S(S &) = delete;
};
S f() { S s; return s; }
S s = f();
```

copy/move

- Is the std::move good or bad?

```
struct S { ... };
S f() {
    S s;
    return std::move(s);
}
```

copy/move

- Is the std::move good or bad?

```
struct S { ... };
S f() {
    S s;
    return std::move(s);
}
```

  - -Wpessimizing-move
    - Because “return s;” is eligible for optional copy elision

copy/move

- Is the std::move good or bad?

```
struct S { ... };
S f() {
    S s;
    return std::move(s);
}
```

  - -Wpessimizing-move
    - Because “return s;” is eligible for optional copy elision
    - And if not done, “return s;” is still special and selects the move ctor

copy/move

- Is the std::move good or bad?

```
struct S { ... };
struct T { T(S &&); }
T f() {
    S s;
    return std::move(s);
}
```

copy/move

- Is the std::move good or bad?

```
struct S { ... };
struct T { T(S &&); }
T f() {
    S s;
    return std::move(s);
}
```

copy/move

- Is the std::move good or bad?

```
struct S { ... };
struct T { T(S &&); }
T f() {
    S s;
    return std::move(s);
}
```

- Is the std::move good or bad?

```
struct S { ... };
struct T { explicit T(S &&); }
T f() {
    S s;
    return T(std::move(s));
}
```

copy/move

- Is the std::move good or bad?

```
struct S { ... };
struct T { T(S &&); }
T f() {
    S s;
    return std::move(s);
}
```

- Is the std::move good or bad?

```
struct S { ... };
struct T { explicit T(S &&); }
T f() {
    S s;
    return T(std::move(s));
}
```

copy/move

- Is the std::move good or bad?

```
struct S1 { ... };
struct S2: S1 { ... };
S1 f() {
    S2 s;
    return std::move(s);
}
```

copy/move

- Is the std::move good or bad?

```
struct S1 { ... };
struct S2: S1 { ... };
S1 f() {
    S2 s;
    return std::move(s);
}
```

  - Depends on whether moving just S1 sub-object is OK
    - Clang gives -Wreturn-std-move warning anyway (suggesting to add std::move)

Why do you ask?

- Because, in the second std::move example

```
std::unique_ptr<AnimationEntry> AnimationEntryList::clone()
{
    std::unique_ptr<AnimationEntryList> pNew( ... );
    for(const auto &i : maEntries)
        pNew->append(*i);
#if HAVE_CXX_CWG1579_FIX
    return pNew;
#else
    return std::move(pNew);
#endif
}
```

Why do you ask?

- Because, in the fourth std::move example

```
vcl::Font EditEngine::CreateFontFromItemSet( ... )
{
    SvxFont aFont;
    CreateFont( aFont, rItemSet, true, nScriptType );
#if HAVE_GCC_BUG_87150
    return aFont;
#else
// <https://gcc.gnu.org/bugzilla/show_bug.cgi?id=87150#c15>:
#if defined __GNUC__ && __GNUC__ == 9 && !defined __clang__
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wredundant-move"
#endif
    return std::move(aFont);
#if defined __GNUC__ && __GNUC__ == 9 && !defined __clang__
#pragma GCC diagnostic pop
#endif
#endif
}
```

redhat.

“Give way to your worst impulse”

—Eno/Schmidt, Oblique Strategies