An in-line defaulted destructor should keep the copy- and move-operations

Document Number:	**P2917 R1**
Date:	2023-06-15
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Reply-to:	Andreas Fertig ⟨ isocpp@andreasfertig.com ⟩
Audience:	EWG

Contents

1 Introduction

Defining a class with a defaulted `virtual` destructor requires users to also default the move- and copy-operations. Otherwise, the class is not moveable as of today, and the behavior for the copy-operations is deprecated and on the list of being removed [P2863R0] §6.8, which was also asked on the reflector [DepCopyOps]. These additional defaulted special members are boilerplate code. This aims to reduce the lines of code required for a class with a defaulted destructor.

2 Motivation

Since C++11, the language has move semantics and with `=default` a way to ask for the compiler-provided implementation of a special member function.

C++11 changed the rules for the implicit declaration of copy functions [depr.impldec]. The behavior that the copy operations are implicitly provided by the compiler if that class has a user-declared destructor is deprecated since C++11.

The author thinks that the interaction between the copy and move operations is correct. However, the destructor is special. The destructor is the only special member function always present independently of the other special member functions. We only lose the destructor if the class contains a member without a destructor.

This paper proposes to decouple the destructor from the other special member functions. A class with an explicitly defaulted destructor on its first declaration (whether `virtual` or not) would then still be copy- and move-able.

Currently

```
class S {
public:
  virtual ~S() = default;
  S(const S&) = default;
  S& operator=(const S&) =default;
  S(S&&) = default;
  S& operator=(S&&) =default;

  virtual void Open();
  virtual void Close();
};
```

With proposal

```
class  S {
public:
  virtual ~S() =default;
  virtual void Open();
  virtual void Close();
};
```

```
class Apple {
public:
  ~Apple() =default;
  Apple(const Apple&) =default;
  Apple& operator=(const Apple&)
                                =default;
  Apple(Apple&&) = default;
  Apple& operator=(Apple&&) =default;
};
```

```
class Apple {
public:
  ~Apple() =default;
};
```

Unchanged

```
class Apple {
public:
  ~Apple();
  Apple(const Apple&) =default;
  Apple& operator=(const Apple&)
                                =default;
  Apple(Apple&&) =default;
  Apple& operator=(Apple&&) =default;
};

Apple::~Apple() =default;
```

```
class Apple {
public:
  ~Apple();
  Apple(const Apple&) =default;
  Apple& operator=(const Apple&)
                                   =default;
  Apple(Apple&&) =default;
  Apple& operator=(Apple&&) =default;
};

Apple::~Apple() =default;
```

3 The design

3.1 Why only for in-class defaulted destructors?

Allowing this behavior only for in-class defaulted destructors seems to be the most consistent with the compiler's behavior for an unmodified class. Allowing also out-of-line destructors could make the implementation hard.

3.2 Historical context

The behavior as currently specified seems to have its root in [N3201] with the wording [N3203]. While the paper mainly targets move and copy the destructor is addressed there as well, as it is part of at least a move. Under the section *Generated moves wouldn't be legal* the paper says the following:

> Note that the presence of a user-defined destructor inhibits the implicit generation of moves so moves are unlikely to be implicitly generated for resource handles.
>
> This concern about destructors wouldn't even arise if a swap was used to replace the state.
>
> For copy assignment, a destroy followed by a move construction would do the trick.

This intend seems to be unchanged by this paper, as a user-declared destructor doesn't allow users to implement a resource handling class.
[N3201] looks inconsistent. In the *Suggestion* section [N3201] says:

> ...
> a. If any move, copy, or destructor is explicitly specified (declared, defined, =default, or =delete) by the user, no copy is generated by default.
>
> b. If any move, copy, or destructor is explicitly specified (declared, defined, =default, or =delete) by the user or if the class is not default constructible, no move is generated by default.

At this point the paper talks about `=default` while later in *Generated moves wouldn't be legal* the paper names *user-defined destructor* as rational.

3.3 Breaking change(s)

This proposal would change existing code and can cause breaking changes, silent and unsilent once.

3.3.1 Now moveable

For example, the class `A` in Listing 3.3.1 currently is not moveable, but would be under this proposal.

```
struct A {
  std::unique_ptr<int> pi;
  virtual ~A() = default;
};
```

3.3.2 Moves now instead of copying

With this proposal, Listing 3.3.2 would now actually move and no longer copy. However, once the deprecation in [N3203] is enforced, Listing 3.3.2 no longer compiles.

```
struct B {
    std::string s;
    virtual ~B() = default;
};

B b1;
B b2;
b2 = std::move(b1); // copies, not moves
```

3.3.3 Disabling moveability with the destructor no longer works

One pattern some people use is to only default the destructor to disable the move operations (see Listing 3.3.3). Under the rules of this proposal, this would no longer be the case and, as such, be a silent breaking change.

```
struct A {
  // The intention here is to surpress move operations
  ~A() = default;
};
```

3.3.4 Slicing

One issue mentioned on the reflector by Nico Josuttis is slicing. In Listing 3.3.4, the copy assignment operator of `A` should probably be `protected` to avoid slicing of `B`.

```
struct A {
  int x{};

  virtual ~A() = default;

  // other virtual member functions
};

struct B : A
{
  int y{};
};
```

```
13
14
15 B b{};
16 A a = b;
```

As long as the deprecation of [N3203] isn't enforced, the code in Listing 3.3.4 can run into a slicing issue. This proposal would extend this slicing risk even if [N3203] is enforced.

4 Implementation

This proposal was not implemented yet.

5 Proposed wording

This wording is based on the working draft [N4950] and was not yet reviewed by a Core expert.

Change **[class.copy.ctor]** 11.4.5.3:

6 If the class definition does not explicitly declare a copy constructor, a non-explicit one is declared *implicitly*. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted (dcl.fct.def). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor other than an explicitly defaulted destructor on its first declaration (depr.impldec).

8 If the definition of a class X does not explicitly declare a move constructor, a non-explicit one will be implicitly declared as defaulted if and only if

- X does not have a user-declared copy constructor,
- X does not have a user-declared copy assignment operator,
- X does not have a user-declared move assignment operator, and
- X does not have a user-declared destructor other than an explicitly defaulted destructor on its first declaration.

Change **[class.copy.assign]** 11.4.6:

2 If the class definition does not explicitly declare a copy assignment operator, one is declared *implicitly*. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy assignment operator is defined as deleted; otherwise, it is defined as defaulted (dcl.fct.def). The latter case is deprecated if the class has a user-declared copy constructor or a user-declared destructor (depr.impldec) other than an explicitly defaulted destructor on its first declaration.

4 If the definition of a class X does not explicitly declare a move assignment operator, one will be implicitly declared as defaulted if and only if

- X does not have a user-declared copy constructor,
- X does not have a user-declared move constructor,
- X does not have a user-declared copy assignment operator, and
- X does not have a user-declared destructor other than an explicitly defaulted destructor on its first declaration.

6 Acknowledgements

Thanks to Nevin Liber for, among other feedback, the examples Listing 3.3.1 Listing 3.3.2.
Thanks for Ville Voutilainen, Nico Josuttis, and Zhihao Yuan for feedback on the first revision of the paper helping to correct some mistakes.

7 Revision History

Version	Date	Changes
0	2023-06-14	Initial draft
1	2023-06-15	• Added reference to the paper that introduced todays behavior. • Added code examples of that that changes/breaks. • Corrected typos. • Adjusted Tony tables.

Bibliography

[P2863R0] Alisdair Meredith: *"Review Annex D for C++26"*, P2863R0, 2023-05-15.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2863r0.html

[N3201] Bjarne Stroustrup: *"Moving right along"*, N3201, 2010-10-23.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3201.pdf

[N3203] Jens Maurer: *"Tightening the conditions for generating implicit moves"*, N3203, 2010-11-11.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3203.htm

[N4950] Thomas Köppe: *"Working Draft, Standard for Programming Language C++"*, N4950, 2023-05-10.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4950.pdf

[DepCopyOps] Deprecation of defaulted copy constructor and copy assignment
http://lists.isocpp.org/core/2023/02/13934.php