Delete if incomplete?

Addressing a needless undefined behavior

P3320 Slides for EWG telecon

Alisdair Meredith
amerediith1@bloomberg.net

May 15, 2024

Bloomberg Engineering

TechAtBloomberg.com

Overview

- State the problem
- Provide examples
- Explore design directions
- Propose Solution

What is the Problem?

Gratuitous Undefined Behavior!

- The C++ standard states that it is undefined behavior to call `delete` on a pointer to an incomplete class type, *unless it satisfies some very specific properties* when the type is completed in the whole program
- These properties are impossible to diagnose in a single translation unit

What is the Problem?

Gratuitous Undefined Behavior!

- The C++ standard states that it is undefined behavior to call `delete` on a pointer to an incomplete class type, *unless it satisfies some very specific properties* when the type is completed in the whole program
- These properties are impossible to diagnose in a single translation unit

7.6.2.9 [expr.delete] Delete
"If the object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or a deallocation function, the behavior is undefined."

Preferred Solution

A path towards a complete solution

- Do not immediately break valid C++23 code
- Deprecate even the valid C++23 cases for consistent compile-time diagnostics
  - Intend to make ill-formed in a future standard
  - Ill-formed future will also remove the remaining UB
- Use *Erroneous Behavior* to address destructor issues
  - All usage is erroneous, including valid C++23 cases
- Retain UB if complete class overloads `operator delete`
  - Resolved when future standard makes the call ill-formed

Example 1a
Well defined

```
namespace xyz {
  struct Widget;                  // forward declaration
  Widget* new_widget();           // factory function
} // close xyz

int main() {
  xyz::Widget *p = xyz::new_widget();
  delete p;                       // delete of incomplete class type
}

namespace xyz {

struct Widget {
  const char *d_name;
  int         d_data;

  ~Widget() = default;           // trivial destructor
};

Widget* new_widget() {
  return new Widget();           // needs the complete type or diagnosed error
}

} // close xyz
```

Example 1b

Undefined behavior

```
namespace xyz {
  struct Widget;              // forward declaration
  Widget* new_widget();       // factory function
} // close xyz

int main() {
  xyz::Widget *p = xyz::new_widget();
  delete p;                   // delete of incomplete class type
}

namespace xyz {

struct Widget {
  const char *d_name;
  int         d_data;

  ~Widget() {}                // non-trivial destructor
};

Widget* new_widget() {
  return new Widget();        // needs the complete type or diagnosed error
}

} // close xyz
```

Example 2a

Well defined

```
namespace xyz {
  struct Widget;               // forward declaration
  Widget* new_widget();        // factory function
} // close xyz

int main() {
  xyz::Widget *p = xyz::new_widget();
  delete p;                    // delete of incomplete class type
}

namespace xyz {

struct Widget {
  const char *d_name;
  int         d_data;

  ~Widget() = default;                 // trivial destructor


};

Widget* new_widget() {
  return new Widget();                 // needs the complete type or diagnosed error
}

} // close xyz
```

Example 2b

Undefined behavior

```
namespace xyz {
  struct Widget;              // forward declaration
  Widget* new_widget();       // factory function
} // close xyz

int main() {
  xyz::Widget *p = xyz::new_widget();
  delete p;                   // delete of incomplete class type
}

namespace xyz {

struct Widget {
  const char *d_name;
  int         d_data;

  ~Widget() = default;                // trivial destructor

  void operator delete(void *) {}     // class-specific deleter
};

Widget* new_widget() {
  return new Widget();                // needs the complete type or diagnosed error
}

} // close xyz
```

Observations

Part 1

- Does not apply to incomplete types other than class types. e.g., enumerations or arrays of unknown bound
- The well-defined cases *match the behavior* of not calling a destructor, and immediately calling global `operator delete`
  - As it's impossible to diagnose well-defined case from UB, the expectation is that UB will do the same
  - UB of not calling the destructor has a different impact of calling the wrong deleter
  - However it is *not* UB to end the lifetime of an object without running its destructor

Example 3a

Well defined: wording has not been touched since 1998

```
namespace xyz {
  class Widget;                                // forward declaration
  Widget* new_widget();                        // factory function
} // close xyz

int main() {
  xyz::Widget *p = xyz::new_widget();
  delete p;                   // delete of incomplete class type
}

namespace xyz {

class Widget {
  ~Widget() = default;        // trivial destructor
};

Widget* new_widget() {
  return new Widget();        // needs the complete type or diagnosed error
}

} // close xyz
```

Example 3b

Ill formed, diagnostic required

```
namespace xyz {
  class Widget { ~Widget() = default; };  // class definition
  Widget* new_widget();                   // factory function
} // close xyz

int main() {
  xyz::Widget *p = xyz::new_widget();
  delete p;                    // delete of complete type with private destructor
}

namespace xyz {




Widget* new_widget() {
  return new Widget();         // needs the complete type or diagnosed error
}

} // close xyz
```

Example 3a revisited
Well defined

```
namespace xyz {
  struct Widget;              // forward declaration
  Widget* new_widget();       // factory function
} // close xyz

int main() {
  xyz::Widget *p = xyz::new_widget();
  delete p;                   // delete of incomplete class type
}

namespace xyz {

struct Widget {
  ~Widget() = default;        // trivial destructor
};

Widget* new_widget() {
  return new Widget();        // needs the complete type or diagnosed error
}

} // close xyz
```

Example 3c

Broken!

```
namespace xyz {
  struct Widget;              // forward declaration
  Widget* new_widget();       // factory function
} // close xyz

int main() {
  xyz::Widget *p = xyz::new_widget();
  delete p;                   // delete of incomplete class type
}

namespace xyz {

struct Widget {
  ~Widget() = delete;         // deleted trivial destructor must be called!
};

Widget* new_widget() {
  return new Widget();        // needs the complete type or diagnosed error
}

} // close xyz
```

Observations

Part 2

- Wording has not been touched since 1998
  - C++11 introduces deleted and defaulted destructors
  - Current wording demands we call the trivial destructors
- Classes can now have private defaulted destructors that are trivial
  - Calling inaccessible (trivial) destructor violates access control
- Deleted destructors are trivial
  - Not clear what it means to call a deleted destructor
  - Open a core issue?

Example 4: Templates introduce a grey zone

Must define `Widget` before first *call* to the template, rather than its definition

```
#include <iostream>
#include <new>

namespace xyz {
  struct Widget;      // forward type decl.
  void report();      // forward function decl.

  auto new_widget() -> Widget*;    // factory

  template <typename T>
  void reclaim(T *p) {
    delete p;
  }

  struct Widget {
    static int s_count;          // # active
    const char *d_name;
    int         d_data;
    Widget()  { ++s_count; }
    ~Widget() { --s_count; }    // non-trivial
  };
} // close xyz
```

```
int main() {
  xyz::Widget* p = xyz::new_widget();
  xyz::report();    // Prints 1

  reclaim(p);         // Sees complete class
  xyz::report();    // Prints 0
}


// Implementation details


void xyz::report() {
  using namespace std;
  cout << Widget::s_count << '\n';
}

auto xyz::new_widget() -> Widget* {
  return new Widget();
}

int xyz::Widget::s_count = 0;
```

Explore Design Directions

- Make ill-formed
  - Deprecate first
  - Breaks valid C++23 code
- Define behavior
  - Do The Right Thing
  - Leak and reclaim memory
  - Unspecified if destructor is called; behavior is erroneous

Do The Right Thing: Implementation A

Store a pointer to deleter with every new expression

- Similar to how `delete[]` works
- Similar to how `shared_ptr` works
    - Handles `delete` through base class with non-virtual destructor
- Type must be complete before call to call `new`
    - Well defined even if the class overloads `operator delete`
    - Valid deleter guaranteed to be stored for `delete` to call
        - UB to call `delete` on a pointer that was not a result of `new`
- Breaks ABI
- Adds access check for destructor when invoking `new`

Do The Right Thing: Implementation B

Delete looks for an implementation defined trampoline function

- Defers error detection to the linker
  - Was the class ever completed?
- Must perform both destructor and memory reclaim to get the correct overload of `operator delete`
- Trampoline emitted in TU with class definition
- Can be safely defined in multiple TUs as identical inline definition
- May selectively ignore access check if type is incomplete, as trampoline is effectively a class member or friend?

Leak and Reclaim

- In other contexts, it is well defined to end an object's lifetime without running its destructor, c.f., ending lifetime by re-using or releasing storage
- Memory is reclaimed only for types that use the global `operator new` and global `operator delete` for memory management
  - Common belief that this is the overwhelming majority of cases
  - UB remains for classes overloading `operator delete`
- Consistent with many implementations today
- Undiagnosed object leaks are still not a great solution

Erroneous Behavior

Unspecified whether destructor is called

- Erroneous behavior is the runtime analog of deprecation
- Behavior is minimally specified in order to remove undefined behavior
  - Erroneous is specifically unreliable, as implementations are encouraged to provide instrumentation and reporting at runtime
  - Reporting may include program termination
- Does erroneous cover the existing well-defined sliver?
  - Easier to instrument and diagnose if it does
  - May break currently valid programs

Observations

- We cannot solve the class-specific delete without breaking either API or ABI
- We can define the destructor behavior without breaking either API or ABI
  - UB regarding destructor is the overwhelmingly common case
- Preferred long term direction may dictate a different transitional solution
  - We should accept that transitional may also be final if we remain committed to no breakage in a future standard

Possible Directions

- Long term:
  - Remove all potential for UB
    - Option A: ill-formed — API break
    - Option B: do it right — ABI break

- Transitional
  - Address only the destructor concerns
    - UB to `delete` if complete type overloads `operator delete`
  - Option A: deprecate all usage; specify as Erroneous Behaviour when called; unspecified whether destructor is called
  - Option B: defer destructor to link time; IFNDR if type is never completed

Comparing solutions across examples

Color Key	Perfect clean-up	Inconsistent specification	Unbounded bad behavior

	Ex 1a trivial	Ex 1b Non-trivial	Ex 3 priv non-triv	Ex 3 deleted	Ex 3 private trivial	Ex 2 overload op	Ex 5 template
C++23	Cleans up	UB	UB	UB[1]	Break access control	UB	IFNDR
Do not destroy	Cleans up	Leak object	Leak object	Leak object	Cleans up	UB	IFNDR
Erroneous behavior	Cleans up	Deprecated	Deprecated	Deprecated	Deprecated[2]	UB	IFNDR
Ill-formed	API break	API break	API break	API break	API break	API break	API break
Call destructor	Cleans up	Cleans up	Break access control	IFNDR	Break access control	UB	Cleans up
Get it "right" Break ABI	Cleans up	ABI break	Break access control	IFNDR	Break access control	ABI break	ABI break

Footnote 1: C++23 specification suggests UB as long as we assume that deleted destructors are never trivial

Footnote 2: The erroneous behavior cleans up correctly, as it is specified to not call the (inaccessible trivial) destructor

Comparing solutions across examples

Color Key	Perfect clean-up	Inconsistent specification	Unbounded bad behavior

	Ex 1a trivial	Ex 1b Non-trivial	Ex 3 priv non-triv	Ex 3 deleted	Ex 3 private trivial	Ex 2 overload op	Ex 5 template
C++23	Cleans up	UB	UB	UB[1]	Break access control	UB	IFNDR
Do not destroy	Cleans up	Leak object	Leak object	Leak object	Cleans up	UB	IFNDR
Erroneous behavior	Cleans up	Deprecated	Deprecated	Deprecated	Deprecated[2]	UB	IFNDR
Ill-formed	API break	API break	API break	API break	API break	API break	API break
Call destructor	Cleans up	Cleans up	Break access control	IFNDR	Break access control	UB	Cleans up
Get it "right" Break ABI	Cleans up	ABI break	Break access control	IFNDR	Break access control	ABI break	ABI break

Footnote 1: C++23 specification suggests UB as long as we assume that deleted destructors are never trivial

Footnote 2: The erroneous behavior cleans up correctly, as it is specified to not call the (inaccessible trivial) destructor

Preferred Solution

We know how to migrate an *API* break, but not an *ABI* break

- Do not immediately break valid C++23 code
- Deprecate even the valid C++23 cases for consistent compile-time diagnostics
    - Intend to make ill-formed in a future standard
    - Ill-formed future will also remove the remaining UB
- Use *Erroneous Behavior* to address destructor issues
    - All usage is erroneous, including valid C++23 cases
- Retain UB if complete class overloads `operator delete`
    - Resolved when future standard makes the call ill-formed