Deprecate Delete of a Pointer to an Incomplete Type

Removing a trigger for UB

Document #:	P3144R0
Date:	2024-02-14
Project:	Programming Language C++
Audience:	SG17 EWG Incubator
	SG12 Undefined Behavior

Reply-to: Alisdair Meredith
<ameredith1@bloomberg.net>
Mungo Gill
<mgill83@bloomberg.net>
John Lakos
<jlakos@bloomberg.net>

Contents

1 Abstract

Deleting a pointer to an incomplete class type is undefined behavior, unless that class type has a trivial destructor and no class-specific deallocation function when completed. This proposal would deprecate such usage, with a view to making it ill-formed in a future standard.

2 Revision History

2024 February mailing (pre-Tokyo)

— Initial draft of this paper.

3 Introduction: Deleting Pointers to Incomplete Types

An incomplete type is a type for which there is a declaration but not a (complete) definition. For example, the class `C` below is of class type but nothing is known about its definition:

```
class C;            // OK, declaration of incomplete type.
class C *cp = 0;    // OK, pointer to incomplete type.
class C c;          // Error, cannot construct and incomplete type.
```

While it is perfectly fine to have a pointer (or even a reference) to an incomplete type, attempting to do anything with that type that requires knowing its size, layout, or member function definitions is typically ill-formed. A particular sore spot in the standard is what happens when someone tries to delete an object of incomplete type:

```
class C;          // OK, incomplete type
void delC(C *cp)
   // Delete the object pointed to by `cp`.
{
   delete cp;     // ??? What should happen here?
}
```

As the example above shows, function `delC` is being asked to know how to delete the object of type `C` without knowing its size, layout, or any other aspects of its definition. What happens in the standard today and what should happen instead?

The current C++ Standard allows for calling `delete` on a pointer to an incomplete type, and that call will produce undefined behavior unless the complete class type, when defined, has a trivial destructor *and* does not declare a class-specific `operator delete`. In that very special case, the defined behavior happens to be a simple no-op for the destructor, followed by a call to the global delete operator to reclaim the memory.

Otherwise, the simple act of failing to include a header can subtly turn a perfectly correct program into one that silently executes undefined behavior. Undefined behavior has all but unbounded potential for bad program behavior, including security leaks, data corruption, deadlocks, and so on.

Detecting the source of this problem — calling `delete` on a pointer to an incomplete class type — is easy for compilers, but detecting the small sliver of well-defined behavior before link time is not simple. The difficult task of providing effective user warnings for this common class of insidious errors is currently left to compiler implementers as a matter of quality of implementation(QoI).

4 Background

The original C++ Standard defined the behavior of calling delete on a pointer to an incomplete class type — i.e., when it just happened to be the same behavior as ignore the call. The compiler has no way of establishing the well-defined cases on a per-translation-unit basis, and why this behavior — that is coincidentally the same as if the type were complete — is singled out to be well defined is unclear:[1]

7.6.2.9 [expr.delete] Delete

5 "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."

We note that invoking `delete` on a pointer to a trivial type that does not overload the class-specific `operator delete` can be implemented as simply not invoking the destructor and instead calling just the global `operator delete` with the address of the trivial object, as the type of the object is not needed once we know the destructor is trivial.

On a related note, there is explicit treatment in the standard for allowing the reuse of storage for an object by constructing another object in its place.

6.7.3 [basic.life] Lifetime

1 "… The lifetime of an object *o* of type `T` ends when:

— if `T` is a non-class type, the object is destroyed, or
— if `T` is a class type, the destructor call starts, or
— the storage which the object occupies is released, or is reused by an object that is not nested within *o* (6.7.2 [intro.object])."

5 "A program may end the lifetime of an object of class type without invoking the destructor, by reusing or releasing the storage as described above.

[*Note 3:* A delete-expression (7.6.2.9 [expr.delete]) invokes the destructor prior to releasing the storage. —*end note*]

In this case, the destructor is not implicitly invoked.

[*Note 4:* The correct behavior of a program often depends on the destructor being invoked for each object of class type. —*end note*]"

So in the case of calling `delete` through a pointer to an incomplete class type, we would be ending the object lifetime by "releasing the storage" and without running the destructor. Note that leaking memory is, in general, well defined behavior, and the user determines whether retaining any resources held by such objects is a bug.

The current user experience for this sort of undefined behavior is decidedly suboptimal. Compilers like to warn users when their code strays within the bounds of undefined behavior, especially when such behavior is easy or reasonable to diagnose. As an experiment, we created two similar versions of a minimal program: (a) one that executes only well-defined behavior and (b) one that executes undefined behavior (the only difference being the triviality of the destructor of the `xyz::Widget` class):

[1]The paper authors believe that support for this exception essentially fell out of implementations' simply choosing to ignore the destructor entirely (which was explicitly permitted for C++98 trivial destructors) and call the global `operator delete` to reclaim memory. In any event, that sliver of defined behavior happens to match exactly the results of calling the correct `delete` operator if the incomplete type turns out to have a trivial destructor once completed.

Well-defined Behavior

```cpp
namespace xyz {
  struct Widget;  // forward decl.
  Widget* new_widget();
}  // close xyz

int main() {
  xyz::Widget *p = xyz::new_widget();
  delete p;
}

namespace xyz {

struct Widget {
  const char *d_name;
  int         d_data;

  // (implicit) trival destructor
  // This is the only difference
};

Widget* new_widget() {
  return new Widget();
}

}  // close namspace xyz
```

Undefined Behavior

```cpp
namespace xyz {
  struct Widget;   // forward decl.
  Widget* new_widget();
}  // close xyz

int main() {
  xyz::Widget *p = xyz::new_widget();
  delete p;
}

namespace xyz {

struct Widget {
  const char *d_name;
  int         d_data;

  ~Widget() {}  // nontrivial dtor.
  // This is the only difference
};

Widget* new_widget() {
  return new Widget();
}

}  // close namespace xyz
```

Notice that, in the example above, there is no problem with deleting the incomplete widget type in (a) because that type is trivially destructible (and doesn't overload `operator delete`). Just by adding a user-defined destructor for the `Widget type` in (b) — that has the exact same definition as the trivial destructor — we make the behavior of the program undefined.

We tested these two similar versions using the Godbolt Compiler Explorer to see which compilers warn on this potential undefined behavior, which warning flags they require, and how long those warnings have been available.

All compilers — even the oldest releases available at Godbolt — report a warning that the destructor might not run. Only Clang warns that the behavior is undefined, and only MSVC needs a command line switch, `/W2`, to enable the warning. Note that the default warning level for a Visual-Studio-created project is `/W3`, so the warning will be reported for the typical user experience.

Note, however, that *all* compilers give precisely the same warning for both the undefined case *and* the well-defined one. There is no easy way to clear the warning in the well-defined case other than to arrange to have the warning disabled for both.

Templates further complicate matters. Consider two similar programs where the only difference between the two is that in the reclaim function (a) *is* and (b) is *not* a template. Note that the `Widget` class is defined **after** the `reclaim` function (template) but before `main` in both cases.

Well-Defined Behavior

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

namespace xyz {
struct Widget;  // forward type decl.

void report();  // forward fun decl.
  // Print number of current widgets.

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

template <class T> // Note: function template
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; }
};
}  // close namespace xyz

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

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

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;
```

Undefined Behavior

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

namespace xyz {
struct Widget;  // forward type decl.

void report();  // forward fun decl.
  // Print number of current widgets.

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

// Only difference: Not a template!
void reclaim(Widget *p) {
  delete p;
}

struct Widget {
  static int  s_count; // # active

  const char *d_name;
  int         d_data;

  Widget()  { ++s_count; }
  ~Widget() { --s_count; }
};
}  // close namespace xyz

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

  reclaim(p);     // complete class
  xyz::report();  // Prints ??
}                 // (1 on gcc, 0 on Clang)

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;
```

In the examples above, example (a) exhibits well-defined behavior because the code for the `reclaim` function template is not generated until the point of instantiation, which occurs *after* the complete definition of the `Widget` struct is visible. On the other hand, example (b) invokes undefined behavior because the code for the (nontemplate) `reclaim` function is generated as soon as its definition is seen.

Now suppose we further modify the example above so that it is a hybrid of the two where the `reclaim` function is now a template function of two arguments, one of which is of the template type `T` and the other is not (does it invoke any undefined behavior, or is all of the behavior it exhibits well-defined?):

Original

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

namespace xyz {
struct Widget;  // forward type decl.

void report();  // forward fun decl.
  // Print number of current widgets.

Widget* new_widget();  // factory

~~template<class 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; }
};
}  // close namespace

~~int main() {~~                                 ___\
  ~~xyz::Widget* p = xyz::new_widget();~~ ~~~/
  ~~xyz::report(); // Prints 1.~~

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

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

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

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

Revisions

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

namespace xyz {

template<class T>
void reclaim(T *p, Widget *q) {
  delete p;
  delete q;
}

}  // close namespace

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

  xyz::Widget* q = xyz::new_widget();
  xyz::report();  // Prints 2.

  reclaim(p, q);
  xyz::report();  // Prints 0.
}
```

This example-paper author believes the Standard claims that this program is actually ill-formed, no diagnostic required (IFNDR), per 13.8.1 [temp.res.general] bullet (6.6), so the whole program is free of requirements regardless of whether the `reclaim` call occurs. However, all current implementations produce a program having the same behavior that invokes the correct `delete` operation for the complete type with both pointers. [2]

As a final twist in the tale of destructors, C++11 added the capability for a private destructor to be trivial, with the use of `= default`. That leads to the following example of a program that is well-defined *only* if the `Widget` type is incomplete.

Well-Defined Behavior

```cpp
// widget.h
namespace xyz {
   class Widget;    // forward declaration
   Widget* new_widget();
}  // close xyz


// program.cpp
#include "widget.h"
#include <new>

int main() {
   xyz::Widget *pWidget = xyz::new_widget();
   delete pWidget; // OK, trivial class
}                  // No access control checks


// widget.cpp
#include "widget.h"

namespace xyz {

// *private* trivial destructor for `Widget`
class Widget { ~Widget() = default; };

Widget* new_widget() {
   return new Widget();
}

}  // close xyz
```

Ill formed

```cpp
// widget.h
namespace xyz {
   class Widget { ~ Widget() = default;};
   Widget* new_widget();
}  // close xyz


// program.cpp
#include "widget.h"
#include <new>

int main() {
   xyz::Widget *pWidget = xyz::new_widget();
   ~~delete pWidget; // inaccessibledestructor~~
}


// widget.cpp
#include "widget.h"

namespace xyz {

// `Widget` is defined in header


Widget* new_widget() {
   return new Widget();
}

}  // close xyz
```

As seen in the well-formed case, there are no access checks performed when the delete operator is called on a pointer to an incomplete type; however, the behavior is well-defined as long as the complete type has a trivial destructor, the language does not require an *accessible* trivial destructor.

In the ill-formed case, we complete the class type before invoking the `delete` operator and the access check for the destructor fails, rendering the program ill formed.

One final example demonstrates the second cause of undefined behavior when calling `delete` on a pointer to an incomplete class type, and that is for a class, even a class with a virtual destructor, providing a class-specific `operator delete`. In order to support the small sliver of well-defined behavior, the compiler must assume that

[2] Despite the seemingly artificially stringent restriction by the standard, the phased approach to template instantiation means that the code for calling the template function will not be generated until all definitions within the current TU have been seen.

after destroying an object it can reclaim its memory by calling the global `operator delete`. There is no known way for a compiler to do better than calling the global operator absent smart linker technology that can see the completed type and fix up the call; no-one is proposing such a feature, nor speculating that it would be implementable.

In this example, observe that the `Widget` class does have a trivial destructor and yet the `delete` call still produces undefined behavior due to the presence of the class-specific `operator delete`.

Defined Behavior

```cpp
// widget.h
namespace xyz {

struct Widget;    // forward declaration

Widget* new_widget();
   // Return an unowned pointer to a `Widget`
   // created with the `new` operator.

}  // close namespace


// program.cpp
#include "widget.h"
#include <new>

int main() {
   xyz::Widget *pWidget = xyz::new_widget();
   delete pWidget;  // OK, Widget is trivial
}


// widget.cpp
#include "widget.h"

namespace xyz {

struct Widget {
  const char *d_name;
  int         d_data;

  // does not have class-specific `delete`
};

Widget* new_widget() {
   return new Widget();
}

}  // close namespace
```

Undefined Behavior

```cpp
// widget.h
namespace xyz {

struct Widget;    // forward declaration

Widget* new_widget();
   // Return an unowned pointer to a `Widget`
   // created with the `new` operator.

}  // close namespace


// program.cpp
#include "widget.h"
#include <new>

int main() {
   xyz::Widget *pWidget = xyz::new_widget();
   delete pWidget;  ~~// UB, class-specific delete~~
}


// widget.cpp
#include "widget.h"

namespace xyz {

struct Widget {
   const char *d_name;
   int         d_data;

   void operator delete(void *) {}
};

Widget* new_widget() {
   return new Widget();
}

}  // close namespace
```

Hence, any attempt to completely resolve concerns about latent undefined behavior must address overloading the delete operator and not just handle the destructor. However, it should be evident that significantly more user classes have non-trivial destructors than provide a class-specific `operator delete`, so solving just one of the problems would be significant progress.

4.1 Improving Solutions With Erroneous Behavior

While we might like to make the whole construct ill formed in a future Standard, doing so for C++26 would present difficulties. Conversely, deprecating the behavior now opens the way to making such code ill formed in a future Standard, without resolving concerns regarding the potential for invoking undefined behavior.

More recently (c. Jan, 2023), Thomas Köppe has proposed [3] a new form of behavior, *erroneous behavior (EB)*, that in many cases can replace cases of *undefined behavior (UB)* with minimally specified behavior that is not itself intended to be reliable, but is not unbounded like UB. In so doing, we are often able to plug a potentially dangerous security vulnerability without sacrificing our ability to subsequently detect and report latent correctness defects in new or legacy code.

The idea behind specifying behavior as erroneous is to preserve all the discouragement by implementations and tools that find and report undefined behavior but to define behavior sufficiently to remove the lack of *any* requirements and thus the unbounded risk (e.g., time travel) associated with undefined behavior.

Erroneous behavior goes beyond specifying and deprecating the behavior by permitting — or even encouraging — implementations to detect such behavior at run time and then perform some well-defined operation (e.g., initialize a variable to a known, very wrong value or even aborting the program), making clear to users that this behavior is neither intended nor supported.

Thus, not calling the destructor would be considered erroneous rather than undefined behavior, but not calling the class-specific `operator delete` would still be UB. However, this paper's authors do not see a suitable defined behavior that could be declared as erroneous when a class-specific `operator delete` is involved on an incomplete type.

We will exhibit two new, refined solutions employing erroneous behavior below.

[3] [P2795] Erroneous behaviour for uninitialized reads

4.2 Doing the right thing

All things being equal, a nearly ideal solution would be to simply do the right thing and call the appropriate destructor even in the presence of an incomplete type. The only drawback being that programs that invoked undefined behavior and seemed to work anyway, might now exhibit new, and possibly even undesirable behavior.

We will consider this general idea as one of our alternate solutions below. There variety of ways to stash a deleter that does the right thing deterministically. While we have some ideas of how to make that work, it is unlikely that we'd be able to do so without violating ABI compatibility or incurring a non negligible (and, for some, unacceptable) runtime overhead on each call to delete, thus running afoul of the zero-overhead principle. Moreover, some errors may not be diagnosable until link time, which is usually expressed as IFNDR (note that we can see clear benefits to mandating a link-time diagnostic).

To demonstrate the feasibility of this approach, we briefly describe two possible implementations directions one might consider to achieve the desired behavior.

1. **`new` expressions stash a deleter** — Have every `new` expression stash the corresponding `delete` behavior as a function pointer preceding the allocated object so that `delete` can always access the necessary metadata to run the correct cleanup code. Note that this is the behavior of the type-erased deleter in `std::shared_ptr`, for example, and similar to the extra bookkeeping the compiler performances behind the scenes when using *array* `new`.

2. **`delete` on incomplete type invokes a *magic* function** — When a class `T` is defined, the compiler can define a *magic* function, `__delete_incomplete(T *ptr)`. As part of the class definition, the compiler can see the class destructor and any class-specific `operator new` overloads. Since the magic function has a known name, the translation unit calling `delete` would know the name of the magic function that will be found at link time. The function is guaranteed to be available when linking, lest the earlier call to `new` would not be possible. Conversely, if the `delete` is called on an invalid pointer (i.e., a pointer to an object that was not allocated by a call to `new`), then the behavior is already undefined before trying to find the *magic* function.

Note that both suggestions above have problems when the destructor or class-specific `new` operator is not publicly accessible. Adopting this solution may require the introduction of a requirement that either the `new` expression has access, perhaps via friendship, or that the compiler assumes access that cannot easily be checked at the call site for `delete`.

Also note that we have no easy way to propagate the friendship check, which at best could produce a runtime failure if such a check could be enabled at all. The earlier example of a program that is well-formed *only* if delete is invoked on a pointer to an incomplete type might become ill formed should we decide to following this direction.

One additional, orthogonal, design decision remains regarding whether to call the stashed function pointer on every call to `delete` or only for `delete` with a pointer to an incomplete class type allowing *static dispatch* in the cases where the type is complete, which would match the case for well defined behavior today. A requirement to always call the stashed deleter is effectively a dynamic dispatch, adding indirection and an additional function call to every `delete` expression; this change, however, would also give the correct behavior when deleting a pointer to a base class, even when the destructor is not declared as `virtual`.

Finally note that this approach would introduce hard-to-diagnose behavior changes in code that previously relied on some `delete` calls (incorrectly) not invoking the destructor; note that such code has undefined behavior today but might be working exactly as a user intends on their specific platform.

5 Problem Statement

Provide a better way for the C++ standard to describe the behavior resulting from the deletion of an *incomplete type* — one whose definition is not known at the point that the code used to delete the object is generated.

5.1 Illustrative Example

This example demonstrates that whether a program has well-defined behavior or undefined behavior cannot be resolved at translation time but depends upon information typically available to only the linker; in this case, the question is whether the completed class type has a trivial destructor.

Well-defined Behavior

```
// widget.h
namespace xyz {

struct Widget;    // forward declaration

Widget* new_widget();
   // Return an unowned pointer to a `Widget`
   // created with the `new` operator.

}  // close namespace


// program.cpp
#include "widget.h"
#include <new>

int main() {
   xyz::Widget *pWidget = xyz::new_widget();
   delete pWidget;  // OK, Widget is trivial.
}


// widget.cpp
#include "widget.h"
#include <new>

namespace xyz {

struct Widget {
  const char *d_name;
  int         d_data;

  // rule of zero trivial class
};

Widget* new_widget() {
   return new Widget();
}

}  // close namespace
```

Undefined Behavior

```
// widget.h
namespace xyz {

struct Widget;    // forward declaration

Widget* new_widget();
   // Return an unowned pointer to a `Widget`
   // created with the `new` operator.

}  // close namespace


// program.cpp
#include "widget.h"
#include <new>

int main() {
   xyz::Widget *pWidget = xyz::new_widget();
   delete pWidget;  ~~// UB, Widget not trivial.~~
}


// widget.cpp
#include "widget.h"
#include <new>

namespace xyz {

struct xyz::Widget {
   const char *d_name;
   int         d_data;

   ~Widget() {}   // nontrivial destructor
};

Widget* new_widget() {
   return new Widget();
}

}  // close namespace
```

5.2 Business Justification

Undefined behavior is unpredictable and frequently a source of hard-to-detect bugs that often lead to security vulnerabilities.

5.3 Measure of Success

More behavior for programs becomes well defined without sacrificing a means of ensuring other forms of correctness, thereby benignly reducing this strain of hard-to-diagnose buggy behavior.

6 Probative Questions

— Is it desirable to provide some definition for the undefined behavior?
— Would such a definition compromise our ability to ensure correctness, now or in the future.
— Is it desirable to continue supporting the well-defined behavior?
— What might be the impact of making some of the current undefined behavior ill formed?

7 Solutions

In this section describe each of the solution candidates in its own separate subsection comprising its motivating principles and any concerns we might have should that solution be adopted. Note that we avoid repeating motivating principles unless they are especially relevant to a later solution as all solutions will ultimately be scored against all principles in the compliance table below (see [please ref compliance table]).

7.1 Proposed Solution: `delete` for Incomplete Class Types Is Deprecated

Absent a strong motivation to support the special case of trivial destructors (and no deallocation function), making such code ill formed might seem best. However, much code in use today likely does indeed use this construct, and making such code ill formed for C++26 could be a big barrier to adoption. Therefore, we propose deprecating such use today with a goal of making such code ill formed in a future Standard, once we have a better understanding of how much of this proposed soon-to-be deprecated code serves a valuable purpose.

We note that there are (at least) two distinct reasons for preferring not to make deleting incomplete types ill formed for C++26: 1. There is a small sliver of use of this construct that is well defined and were we to make all such use ill-formed we would break perfectly correct well-formed, well-defined code. 2. Just because a function contains a code path that, if called, might invoke UB doesn't mean that it necessarily does (or even that the function is always or even ever called). Hence, perfectly correct programs having this construct in some piece of dead code would abruptly stop working.

- Motivating Principles
  - Solution must be implementable
    - A solution that has no known implementation strategy is not viable.
  - Solution must be specifiable
    - Some aspects of implementation (e.g., what happens when undefined behavior occurs, or required diagnostics before forced termination) are outside the scope of the standard.
  - Detect and report as much UB as possible at compile time
    - The more potential UB that's detected at compile time, the fewer opportunities there are for security vulnerabilities and correctness bugs.
  - Minimize requiring implementations to break ABI compatibility
    - An aspect of stability that has long been held dear in WG21, perhaps to a fault, is that the new versions of the standard must minimize the need for implementations to break binary compatibility with their existing client base.
  - Minimize breaking existing code at compile time across one release
    - Ideally we will always give developers at least one Standard release to address working code that is being deprecated, so they have time to upgrade their code incrementally rather than having to change all the code in lockstep before upgrading at all.
  - Avoid making well-formed code into ill-formed code
    - Backward compatibility of correct programs is essential to the stability and practical use of the language for production use.
  - Avoid making undefined-behavior ill-formed across one release
    - Just because there is the potential for undefined behavior to occur does not mean that it will in all (or even any) circumstances.
  - Minimizes silent changes to essential runtime behavior
    - Maintaining explicitly stated defined behavior is essential for the stable, safe, and reliable use of the C++ language.
  - Minimizes silent changes to nonessential runtime behavior
    - Changing any behavior of conforming code without diagnostics can result in production problems that are very difficult to diagnose.
  - Minimizes silent changes to undefined runtime behavior
    - In theory, any change to undefined behavior that results from defining it is fully Liskov substitutable. From a practical perspective, however, forcing a particular behavior that differs from what implementations do now might adversely affect the (e.g., revenue-producing) production

    programs of their clients. Implementers would nonetheless be forced to make that change to remain compliant.
  - Satisfies the Zero overhead principle
    - The addition of a language feature will ideally have zero runtime performance or per-object space impact on code that is not written in terms of that feature.
  - Minimizes additional runtime overhead for pre-existing code
    - The runtime performance and per-object space requirement of pre-existing code shall not (by default) be pessimized when a new C++ Standard is adopted. New Standards can be adopted without penalizing existing users is important.
  - Does not leave whether code compiles to QoI
    - Allowing QoI to determine what does and doesn't compile means that code that compiles and runs on one compiler might not compile on another, limiting the portability of the language.
  - Does not preclude the possibility of better solutions
    - Backward compatibility for a better, more practically viable solution, should one come along, is not precluded. We want to avoid adopting a less-than-ideal solution if it would preempt future improvements.
- Concerns
  - This solution does not prevent errors.
    - Deprecation warnings are left to implementers as a matter of QoI.
  - Well-defined behavior is deprecated along with UB.
    - we have no way of knowing the difference.

7.2 Alternative Solution: `delete` Expressions Must *Do The Right Thing.*

The most obvious resolution of undefined behavior is to just *do the right thing.* That is, we simply remove the current permission to produce UB and require compilers to produce the correct `delete` behavior as if the class type were complete. For some ideas on how this abstract requirement might be accomplished, see Doing the Right Thing.

- Motivating Principles
  - Makes C++ harder to use incorrectly
    - Minimizes misuse by removing sharp edges, such as needless undefined behavior, so we can improve the user experience (ideally without loss of functionality, expressibility, or performance).
  - Reasonably defines gratuitous undefined behavior
    - The undefined behavior has a clear and intuitive defined behavior that is not likely to wind up having a better interpretation or purpose.
  - Makes C++ harder to use incorrectly
    - C++ has rightly earned a reputation of being a sharp tool. We aim to reduce misuse by removing sharp edges, such as needless undefined behavior, so we can improve the user experience without loss of functionality or expressibility.
  - Makes C++ easier to use correctly
    - C++ has rightly earned a reputation of being a sharp tool. We should make it easier for practitioners, especially students, to make the best choices.
- Concerns
  - This solution is not known to be implementable in an efficient or effective manner.

7.3 Alternative Solution: Define Behavior to Not Call the Destructor

Another option is to define the behavior of invoking `delete` on a pointer to an incomplete class type to simply not call the destructor, consistent with the discussion in the background material.

Note, the (proposed) deprecated behavior still opens up the potential for two template instantiations of the same function to violate the ODR if one instantiation sees the complete class type, while the other sees an incomplete class type. Making the deprecated use case ill formed would remove this subtle violation in future standards as well.

- Motivating Principles
  - Solution must be implementable
  - Solution must be specifiable
  - Avoid making well-formed code into ill-formed code
  - Minimizes silent changes to essential runtime behavior
  - Minimizes silent changes to nonessential runtime behavior
  - Does not leave whether code compiles to QoI
  - Makes C++ easier to use correctly
  - Makes C++ harder to use incorrectly
- Concerns
  - Leaking is arguably bad behavior.
    - Silently leaking of objects, where their memory is reclaimed but no other resources are managed by such objects. Deterministic calling of destructors separates C++ from garbage-collected languages managing only memory, so silently failing to deliver on a key feature of C++ language design would arguably be an active step backward.
  - This solution removes permission for implementations to do better.
    - That is, by codifying this suboptimal behavior, we make it manifestly backward incompatible to change that defined behavior in the future, should some better alternative present itself.

7.4 Alternative Solution: `delete` for Incomplete Class Types Is Ill formed

While the long term goal is to make such code ill formed, doing so is likely to break too much code if we make that change directly in C++26. Much of that ill-formed code would already be undefined behavior (so technically already broken), yet well-defined cases having trivial destruction can be relied upon today. Moreover, invoked undefined behavior leaking in dead code isn't actually a defect, it's just suboptimal for maintenance purposes. Finally, code breaking noisily when it previously silently behaved as expected (or as could be tolerated) could be a deterrent to timely adoption of a new standard. C.f. Hyrum's Law:

> With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.

- Motivating Principles
  - Detect and report as much UB as possible at compile time
  - Minimize requiring implementations to break ABI compatibility
  - Minimizes additional runtime overhead for pre-existing code
  - Does not leave whether code compiles to QoI.
  - Does not preclude the possibility of better solutions
  - Minimize requiring implementations to break ABI compatibility
  - Makes C++ harder to use incorrectly
- Concerns
  - This solution breaks well-defined code.
    - Right now, classes having trivial destructors and no overloaded `operator delete` work just fine; some user may feel disenfranchised.

7.5 Alternative Solution: All Destructors Are Virtual

The only incomplete types that can be deleted are class types, as no other incomplete type can make it to a `delete` expression before being completed. (For example, an `enum` type is incomplete only within its definition for these purposes since the underlying type is determined by any forward declaration).

Since only class types are impacted, we might decide to require all destructors to become virtual, regardless of whether the keyword is used, just as `noexcept` is deduced for destructors when no exception specification is explicitly supplied. As there is only one spelling of the destructor, the linker will know exactly how to resolve the dynamic dispatch.

- Motivating Principles

    - Solution must be specifiable
    - Avoid making well-formed code into ill-formed code
    - Avoid making undefined-behavior ill-formed across one release.
    - Minimizes silent changes to essential runtime behavior
    - Does not leave whether code compiles to QoI
    - Makes C++ easier to use correctly
    - Makes C++ harder to use incorrectly
- Concerns
    - Forcing dynamic dispatch on all `delete` operators to find the virtual destructor will likely incur runtime overhead.
    - This solution breaks ABI throughout the language.
    - The observable behavior of programs that compile today will be changed.
    - This solution cannot solve the problem for unions without even more significant changes to the language.
    - The problem of overloading class-specific `operator delete` remains unsolved.

7.6 Alternative Solution: `delete` for Incomplete Class Types Aborts

Both the well-defined behavior (for types that turn out to be trivial with no overloaded `delete` operator) and undefined behavior could be replaced with a well-defined call to `std::terminate` when invoking `delete` at runtime on a pointer to an object having incomplete type.[4][5]

- Motivating Principles
    - Minimize requiring implementations to break ABI compatibility
    - Minimize breaking existing code at compile time across one release
    - Avoid making well-formed code into ill-formed code
    - Avoid making undefined-behavior ill-formed across one release
    - Does not leave whether code compiles to QoI
- Concerns
    - Whether this solution is implementable without breaking ABI or at no cost is unclear.
    - This solution might cause currently working (including some currently correct) code to terminate rather than behave as desired.

7.7 Refined Solution: Erroneous Behavior Does Not Call Destructor

This solution specifies that the currently undefined behavior for deleting an incomplete type that does not overload `operator delete` will simply not call the destructor; that (defined) behavior will be *erroneous*. Note that deleting a pointer to an object of incomplete type that overloads `operator delete` is still undefined behavior.

- Motivating Principles

    - Minimize requiring implementations to break ABI compatibility
    - Minimize breaking existing code at compile time across one release
    - Avoid making well-formed code into ill-formed code
    - Avoid making undefined-behavior ill-formed across one release
    - Minimizes silent changes to essential runtime behavior
    - Minimizes silent changes to nonessential runtime behavior

[4]Note that `std::terminate` is intended for failures in the C++ exception handling facility, and calling `std::abort` might be more appropriate. Alternatively, once contracts are added to the language [P2900R3], violating this runtime requirement could be treated as a precondition violation invoking the violation handler.

[5]As an implementation-defined QoI alternative we might, consider allowing implementations to emit a static trace before calling `std::abort`. A stack trace is not much help to most users but is quite helpful to developers trying to track down where the problematic `delete` occurred. C++23 provides a stack trace library that might be used to help display a track trace, although that library is not required for a freestanding implementation. We note that whether this solution is implementable without ABI break or at no runtime cost is unclear. Please also note that stack traces are a layer below what the Standard is able to specific (hence we have not considered it as a separate solution).

    - Minimizes additional runtime overhead for pre-existing code
    - Does not leave whether code compiles to QoI
    - Does not preclude the possibility of better solutions
    - Makes C++ harder to use incorrectly
    - Reasonably defines gratuitous undefined behavior
    - Makes C++ harder to use incorrectly
    - Makes C++ easier to use correctly

- Concerns

    - All diagnostics are at runtime rather than compile time.
        - There is no way for the compiler to know when a call to delete is erroneous.
    - Does not address UB with operator delete.
        - It is sill UB to call delete on an incomplete type that overloads `operator delete`.

7.8 Refined Solution: Deprecate, Erroneous Behavior Does Not Call Destructor

This solution, like the previous one, specifies that the currently undefined behavior for deleting an incomplete type that does not overload `operator delete` will not call the destructor and will be *erroneous*. Additionally, we will deprecate calling `delete` on a pointer to an object of any incomplete type.

- Detect and report as much UB as possible at compile time

- Concerns

    - All diagnostics are at runtime rather than compile time.
    - Does not address UB with operator delete.

8 Curated, Refined, Characterized, and Ranked Principles

From here on out we follow the principled-design process described by paper [P3004R0].

1. Collect the motivating principles presented in the solutions above.
2. Strip put any duplicates
3. Refined them in place, as needed.
4. Split, merge as needed.
5. Append (to the end) any additional relevant principles that come to mind.
6. Provide mnemonic principle ID for each principle for easy reference.
7. Characterize each principle in terms of
   — Objectivity
   — Importance
   — Pair-wise comparison
8. Rank the Principles in decreasing order of weight.

In successive subsequence sections we will * Score the proposed solutions against these ordered principles. * Analyze the results. * Review the results, look for new solutions, and repeat until satisfied. * Provide recommendations.

The table below comprises the refined, characterized, and ranked principles.

Table 7: Rank, Importance, Objectivity, and ID for each Principle Statement.

Rank	i	o	Principle ID	Principle Statement
1	@	@	CanImplement	Solution must be implementable
2	@	@	CanSpecify	Solution must be specifiable
12	5	5	MaxUBDetect	Maximize detecting and reporting UB at compile time
7	9	@	MinABIBreak	Minimize requiring implementations to break ABI compatibility
4	9	@	MinBrkNonDeprec	Minimize breaking non-deprecated code at compile time
5	9	@	MinAnyWFtoIF	Minimize making any well-formed code into ill-formed code
10	5	@	MinUBtoIFinSR	Minimize making undefined-behavior ill-formed in one release
3	@	@	MinSilentChgEB	Minimize silent changes to essential runtime behavior
11	5	@	MinSilentChgNE	Minimize silent changes to nonessential runtime behavior
17	1	@	MinSilentChgUB	Minimize silent changes to undefined runtime behavior
8	9	5	MaxZeroOverhead	Maximize satisfaction of the Zero overhead principle
6	9	@	MinRuntimeOH	Minimize additional runtime overhead for pre-existing code
9	9	@	MinQoICompiles	Minimize ability to compile on QoI
13	5	5	MinPreclusion	Minimize precluding a better solutions
14	5	5	MaxReplaceUB	Maximize replacement of gratuitous undefined behavior
15	5	5	MinMisuse	Minimize accidental misuse of C++
16	5	-	MaxEaseOfUse	Maximize ease of use of C++

9 The Compliance Table

Now we will score the solutions against the ranked principles. We introduce the status-quo solution as the current baseline that viable solutions must beat. Recall that the scale is @ means `true` (100%), 9 means 90%, ..., % means 50%, 1 means 10%, and - means `false` (0%).

A. Status Quo: No Change to the Standard
B. Proposed Solution: `delete` for incomplete class types is Deprecated
C. Alternative Solution: `delete` Expressions Must *Do The Right Thing*
D. Alternative Solution: Define behavior not to call the destructor
E. Alternative Solution: `delete` for incomplete class types is ill formed
F. Alternative Solution: All destructors are virtual
G. Alternative Solution: `delete` for incomplete class types aborts
H. New Solution: Erroneous Behavior replaces current undefined behavior
I. New Solution: undefined -> erroneous behavior + deprecation approach

Table 8: Compliance Table

Rank	i	o	Principle ID	A	B	C	D	E	F	G	H	I
1	@	@	CanImplement	@	@	@	@	@	9	@	@	@
2	@	@	CanSpecify	@	@	@	@	@	9	@	@	@
3	@	@	MinSilentChgEB	@	@	9	@	@	-	5	@	@
4	9	@	MinBrkNonDeprec	@	@	9	@	-	@	9	@	@
5	9	@	MinAnyWFtoIF	@	@	9	@	-	@	@	@	@
6	9	@	MinRuntimeOH	@	@	5	@	@	1	@	@	@
7	9	@	MinABIBreak	@	@	3	@	@	@	@	@	@
8	9	5	MaxZeroOverhead	@	@	5	@	@	1	@	@	@
9	9	3	MinQoICompiles	@	@	@	@	@	@	@	@	@
10	5	@	MinUBtoIFinSR	@	@	@	@	-	@	@	@	@
11	5	@	MinSilentChgNE	@	@	@	@	@	@	@	@	@
12	5	5	MaxUBDetect	-	9	@	5	@	7	1	5	9
13	5	5	MinPreclusion	@	@	1	-	@	-	5	@	@
14	5	5	MaxReplaceUB	-	-	@	7	@	7	@	7	7
15	5	5	MinMisuse	-	5	@	5	@	@	5	7	9
16	5	-	MaxEaseOfUse	7	5	@	@	5	@	7	5	7
17	1	@	MinSilentChgUB	@	@	-	9	@	3	-	9	9

10 Analysis of the Compliance Table

At this point, we now look at successive select rows, starting with Rank 1, of the compliance table that cause one or more of candidate solutions to be decimated (reduced to a lowercase letter) or eliminated from consideration (replaced by an underscore).

Ranks 1–3: A B C D E F G H I —

The first two rows have full marks across the board, so the third row is the first to differentiate. Solution C is expected to incur a minor change of behavior by calling trivial destructors through a callback; however, there is not expected to be any observable change of behavior, so we allow it to continue.

On the other hand, solution F has silent visible changes in almost all programs, as all destructors become virtual; we reject solution F outright at this point.

Solution G clearly changes the sliver of well-defined behavior by turning correct destruction into an `abort`, but there is some argument that that would be a necessary-but-not-minimal silent change in order to resolve the many broken programs with undefined behavior, so we score it a 5 and demote it to lower case.

Rank 4: A B C D E _ g H I — MinBrkNonDeprec(9,@)

There is a risk that solution C might break the extreme corner case introduced by C++11 where private destructors can be trivial, creating program that are well defined *only* if pointing to an incomplete type; again, we let solution C continue.

Solution E clearly breaks well-formed code that is not yet deprecated, and does so deliberately; it is rejected at this point.

Solution G changes well-defined behavior to a runtime failure, but this is not a compile-time breakage so we let it pass with a 9.

Rank 5: A B C D _ _ g H I — MinAnyWFtoIF(9,@)

Row 5 is a stronger form of row 4 that does not allow for breaking even deprecated code. All solutions still in contention satisfy this principle.

Rank 6: A B C D _ _ g H I — MinRuntimeOH(9,@)

Row C clearly anticipates some kind of runtime overhead to perform some kind of virtual dispatch on every delete, storing an extra function pointer with every `new`; however, the overhead is essential to produce correct program behavior, so we score a 5 and demote to lower case.

All other remaining solutions satisfy this principle.

Rank 7: A B c D _ _ g H I — MinABIBreak(9,@)

It is not expected that solution C can be implemented without breaking ABI, so it is eliminated by this principle.

All other remaining solutions satisfy this principle.

Ranks 8–11: A B _ D _ _ g H I —

All remaining solutions satisfy these principle.

Rank 12: A B _ D _ _ g H I — MaxUBDetect(5,5)

Solution A makes no effort to report undefined behavior so is rejected at this point. Solution G tries to define away all undefined behavior by turning into a runtime failure, which is a poor substitute for finding likely errors at compile time, so it too is rejected by this principle.

Solutions D and H silently resolve the most common cause of undefined behavior by defined `delete` to not call the destructor in these cases. That entirely eliminates the category of UB so that it does not need to be reported, but also defines a program to silently leak objects (not memory) and their associated resources that may accumulate over runtime; we score them a 5 and they are demoted to lower case letters.

Solutions B and I score a 9 for deprecating the causes of undefined behavior, but do not get full marks as deprecation warnings are a matter for QoI, unlike making the construct ill-formed.

Rank 13: _ B _ d _ _ _ h I — MinPreclusion(5,5)

Solution D, which is already demoted, goes a long way to providing a complete solution that cannot be replaced without breaking the new valid programs; it is rejected by this principle.

All other remaining solutions satisfy this principle.

Rank 14: _ B _ d _ _ _ h I — MaxReplaceUB(5,5)

Solution B does nothing to actually protect current users from undefined behavior so is eliminated by this principle.

Solutions H and I define the most significant cause of UB and make it erroneous allowing them to continue through the remaining principles.

Rank 15–17: _ B _ d _ _ _ h I

For ease of comparison, we reprint the subset of the compliance table specific to the remaining principles and solutions.

Table 9: Reduced Compliance Table

Rank	**i**	**o**	**Principle ID**	**h**	**I**
15	5	5	MinMisuse	7	9
16	5	-	MaxEaseOfUse	5	7
17	1	@	MinSilentChgUB	9	9

It is evident that solution I is superior to the already demoted solution H, so solution I proceeds as our preferred solution.

11 Recommendations

The preferred solution is to deprecate all attempts to call `delete` on a pointer to an incomplete type and define the behavior in such cases to be erroneous, ending the lifetime of the target object without running its destructor.

This resolution preserves the existing well-defined behavior, now deprecated. Further, open-ended undefined behavior is now restricted to only the cases of a class type that provides a class-specific `delete` operator. The use of erroneous behavior in the case that objects are leaked provides a hook for implementations to give better diagnostics, at both compile time and run time. This approach can be augmented with deprecation, which would make way for properly making this use of delete ill formed in some future standard.

12 Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N4971], the latest draft at the time of writing. It also assumes that a revision of [P2795] has landed that defines the term *erroneous behavior*.

6.7.3 [basic.life] Lifetime

5 A program may end the lifetime of an object of class type without invoking the destructor, by reusing or releasing the storage as described above.

[*Note 3:* A *delete-expression* (7.6.2.9 [expr.delete]) invokes the destructor prior to releasing the storage, unless the class type is incomplete at the point of deletion. —*end note*]

In this case, the destructor is not implicitly invoked.

[*Note 4:* The correct behavior of a program often depends on the destructor being invoked for each object of class type. —*end note*]”

7.6.2.9 [expr.delete] Delete

5 If the object being deleted has incomplete class type at the point of deletion ~~and~~ the destructor is not called; this behavior is deprecated and erroneous. ([depr.expr.delete]). If the complete class has ~~a non-trivial destructor or~~ a deallocation function, the behavior is undefined.

[*Note X:* The correct behavior of a program often depends on the destructor being invoked for each object of class type. —*end note*]

6 If the value of the operand of the *delete-expression* is not a null pointer value and the selected deallocation function (see below) is not a destroying operator delete and the type of the operand is not pointer to an incomplete class type, the *delete-expression* will invoke the destructor (if any) for the object or the elements of the array being deleted. In the case of an array, the elements will be destroyed in order of decreasing address (that is, in reverse order of the completion of their constructor; see 11.9.3 [class.base.init]).

D.X Delete an object of incomplete class type [depr.expr.delete]

1 It is deprecated to call the `delete` operator (7.6.2.9 [expr.delete]) with an operand whose type is pointer to an incomplete class type.

13 Acknowledgements

Thanks to Michael Park for the pandoc-based framework used to transform this document's source from Markdown.

Thanks to Bill Chapman, Mike Giroux, and Oleg Subbotin of the BDE team who provided much of the feedback that helped evolve this proposal.

14 References

[N4971] Thomas Köppe. 2023-12-18. Working Draft, Programming Languages — C++.
https://wg21.link/n4971

[P2795] Thomas Köppe. Erroneous behaviour for uninitialized reads.
https://wg21.link/p2795

[P2900R3] Joshua Berne, Timur Doumler, Andrzej Krzemieński. 2023-12-17. Contracts for C++.
https://wg21.link/p2900r3

[P3004R0] John Lakos, Harold Bott, Bill Chapman, Mungo Gill, Mike Giroux, Alisdair Meredith, Oleg Subbotin. Principled Design for WG21.
https://wg21.link/p3004r0