C++ now

An (In-)Complete Guide to C++ Object Lifetimes

Jonathan Müller

2024

What are objects and lifetime?

Objects are fundamental to C++

[intro.object]/1

> *The constructs in a C++ program create, destroy, refer to, access, and manipulate **objects**.*

Objects are fundamental to C++

[intro.object]/1

> *The constructs in a C++ program create, destroy, refer to, access, and manipulate* ***objects****.*

Key terms:

1. Storage

Objects are fundamental to C++

[intro.object]/1

> *The constructs in a C++ program create, destroy, refer to, access, and manipulate **objects**.*

Key terms:

1. Storage
2. Value

think-cell

Objects are fundamental to C++

[intro.object]/1

> *The constructs in a C++ program create, destroy, refer to, access, and manipulate* ***objects****.*

Key terms:

1. Storage
2. Value
3. Type

think-cell

1. Storage

Definition

[intro.memory]/1

*The fundamental storage unit in the C++ memory model is the **byte**. [...] The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique **address**.*

think-cell

1. Storage

Definition

[intro.memory]/1

The fundamental storage unit in the C++ memory model is the ***byte****. […] The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique* ***address****.*

49	7F	CF	2B	6C	C3	09	7E	E0	90	C6	0E	E3
40	E0	E9	90	3D	93	$\boxed{41}$	8A	94	4B	0B	36	0C
74	EA	28	17	F7	AD	92	A5	FB	9D	FF	D4	C1
6B	2C	C1	D1	F5	7A	66	68	B1	31	81	E0	CD

think-cell

What does `0x41` *mean*?

2. Value

What does `0x41` *mean*?

- 8-bit integer 65

2. Value

What does `0x41` *mean*?

- 8-bit integer 65
- character `'A'`

2. Value

What does `0x41` *mean*?

- 8-bit integer 65
- character `'A'`
- start of string `"A…"`

2. Value

What does `0x41` *mean*?

- 8-bit integer 65
- character `'A'`
- start of string `"A…"`
- part of a 32-bit integer

2. Value

What does `0x41` *mean*?

- 8-bit integer 65
- character `'A'`
- start of string `"A…"`
- part of a 32-bit integer
- …

2. Value

Definition

Elements of Programming

> *A **datum** is a finite sequence of 0s and 1s. [...] We refer to a datum together with its interpretation as a **value**.*

think-cell

3. Type

Definition

A **type** describes the interpretation of a datum.

3. Type

Definition

A **type** describes the interpretation of a datum.

`unsigned char` 1 byte of memory interpreted as an 8-bit unsigned integer.

3. Type

Definition

A **type** describes the interpretation of a datum.

`unsigned char` 1 byte of memory interpreted as an 8-bit unsigned integer.

`int` 4 bytes of memory interpreted as a 32-bit two's complement integer.

think-cell

3. Type

Definition

A **type** describes the interpretation of a datum.

`unsigned char` 1 byte of memory interpreted as an 8-bit unsigned integer.

`int` 4 bytes of memory interpreted as a 32-bit two's complement integer.

`std::string` 24 bytes of memory interpreted as pointer to a null-terminated sequence of `char`, size, and capacity.

think-cell

What are objects?

Definition

An **object** has a particular *type* and occupies a region of *storage* at a particular *address* where its *value* is stored.

```
int x = 42;         // object of type int, storing the value 42
float y = 3.14f;    // object of type float, storing the value 3.14
x = 11;             // change the value of object x
```

What aren't objects?

Functions aren't objects; only function pointers

```
void f() {} // not an object

void (*pf)() = f; // object whose value is the address of f
```

What aren't objects?

Functions aren't objects; only function pointers

```
void f() {} // not an object

void (*pf)() = f; // object whose value is the address of f
```

References aren't objects; they are aliases to objects

```
int x = 11;      // object
int& ref = x;    // alias for the object above
```

think-cell

What is lifetime?

Definition

[basic.life]/1

> *The **lifetime** of an object […] is a runtime property of the object […].*

What is lifetime?

Definition

[basic.life]/1

> *The **lifetime** of an object [...] is a runtime property of the object [...].*

[basic.life]/4

> *The properties ascribed to objects [...] throughout this document apply for a given object [...] only **during its lifetime**.*

think-cell

What is lifetime?

Definition

[basic.life]/1

*The **lifetime** of an object […] is a runtime property of the object […].*

[basic.life]/4

*The properties ascribed to objects […] throughout this document apply for a given object […] only **during its lifetime**.*

[basic.life]/Note 2

*In particular, before the lifetime of an object starts and after its lifetime ends there are **significant restrictions on the use** of the object.*

Object lifetime in a nutshell

1. Storage is allocated.

Object lifetime in a nutshell

1. Storage is allocated.
2. An object is “initialized”; the lifetime starts.

Object lifetime in a nutshell

1. Storage is allocated.
2. An object is “initialized”; the lifetime starts.
3. An object is used; its value changed or read.

think-cell

Object lifetime in a nutshell

1. Storage is allocated.
2. An object is “initialized”; the lifetime starts.
3. An object is used; its value changed or read.
4. An object is destroyed; the lifetime ends.

Object lifetime in a nutshell

1. Storage is allocated.
2. An object is “initialized”; the lifetime starts.
3. An object is used; its value changed or read.
4. An object is destroyed; the lifetime ends.
5. Storage is deallocated.

Note about terminology

Objects can be *created*: This does not necessarily start the lifetime yet.

Note about terminology

Objects can be *created*: This does not necessarily start the lifetime yet.

Objects can be *destroyed*: This ends the lifetime.

think-cell

Note about terminology

Lifetime is something the standard invented to describe semantics on the abstract machine.

It has nothing to do with the physical machine your code actually executes on.

Level 0: Variable declaration

think-cell

Level 0: Variable declaration

Object creation

[intro.object]/1

> *An object is created by a **definition** […].*

Level 0: Variable declaration

Object creation

[intro.object]/1

> *An object is created by a **definition** […].*

```
int main() {
    int x = 11; // create the object, allocate storage + start lifetime



}
```

think-cell

Level 0: Variable declaration

Object creation

[intro.object]/1

*An object is created by a **definition** […].*

```
int main() {
    int x = 11; // create the object, allocate storage + start lifetime
    std::print("x = {}\n", x); // use the object
    ++x;                        // use the object
    std::print("x = {}\n", x); // use the object
}
```

think-cell

Level 0: Variable declaration

Object creation

[intro.object]/1

*An object is created by a **definition** […].*

```
int main() {
    int x = 11; // create the object, allocate storage + start lifetime
    std::print("x = {}\n", x); // use the object
    ++x;                       // use the object
    std::print("x = {}\n", x); // use the object
} // destroy the object, end lifetime + deallocate storage
```

think-cell

Storage duration

Definition

[basic.stc.general]/1

> *The **storage duration** is the property of an object that defines the **minimum potential lifetime of the storage** containing the object. The storage duration is determined by the construct used to create the object.*

think-cell

Storage duration

Definition

[basic.stc.general]/1

*The **storage duration** is the property of an object that defines the **minimum potential lifetime of the storage** containing the object. The storage duration is determined by the construct used to create the object.*

[basic.stc.general]/2

***Static**, **thread**, and **automatic storage durations** are associated with objects introduced by declarations.*

think-cell

Automatic storage duration

Definition

[basic.stc.auto]/1

Variables that belong to a ***block or parameter scope*** *and are not explicitly declared* `static`, `thread_local`, *or* `extern` *have* ***automatic storage duration****. The storage for these entities* ***lasts until the block*** *in which they are created* ***exits****.*

think-cell

Automatic storage duration

Definition

[basic.stc.auto]/1

*Variables that belong to a **block or parameter scope** and are not explicitly declared `static`, `thread_local`, or `extern` have **automatic storage duration**. The storage for these entities **lasts until the block** in which they are created **exits**.*

```
int main() {
    int a; // allocation of a
    int b; // allocation of b
```

think-cell

Automatic storage duration

Definition

[basic.stc.auto]/1

*Variables that belong to a **block or parameter scope** and are not explicitly declared `static`, `thread_local`, or `extern` have **automatic storage duration**. The storage for these entities **lasts until the block** in which they are created **exits**.*

```
int main() {
    int a; // allocation of a
    int b; // allocation of b
    {
        int c; // allocation of c
```

Automatic storage duration

Definition

[basic.stc.auto]/1

*Variables that belong to a **block or parameter scope** and are not explicitly declared `static`, `thread_local`, or `extern` have **automatic storage duration**. The storage for these entities **lasts until the block** in which they are created **exits**.*

```
int main() {
    int a; // allocation of a
    int b; // allocation of b
    {
        int c; // allocation of c
    } // deallocation of c
```

think-cell

Automatic storage duration

Definition

[basic.stc.auto]/1

*Variables that belong to a **block or parameter scope** and are not explicitly declared `static`, `thread_local`, or `extern` have **automatic storage duration**. The storage for these entities **lasts until the block** in which they are created **exits**.*

```cpp
int main() {
    int a; // allocation of a
    int b; // allocation of b
    {
        int c; // allocation of c
    } // deallocation of c
} // deallocation of a and b
```

think-cell

Static storage duration

Definition

[basic.stc.static]/1

> *All variables which do not have thread storage duration and belong to a **namespace scope** or are first declared with the **`static` or `extern`** keywords have **static storage duration**. The storage for these entities lasts for the **duration of the program**.*

Static storage duration

Definition

[basic.stc.static]/1

> *All variables which do not have thread storage duration and belong to a **namespace scope** or are first declared with the **`static` or `extern`** keywords have **static storage duration**. The storage for these entities lasts for the **duration of the program**.*

```
int global; // static storage
static int static_global; // static storage

void f() {
    static int function_local_static; // static storage
    extern int extern_global; // static storage
}
```

Thread storage duration

Definition

[basic.stc.thread]/1

*All variables declared with the **`thread_local`** keyword have **thread storage duration**. The storage for these entities lasts for the **duration of the thread** in which they are created. There is a distinct object or reference per thread, and use of the declared name refers to the entity associated with the current thread.*

think-cell

Thread storage duration

Definition

[basic.stc.thread]/1

*All variables declared with the **`thread_local`** keyword have **thread storage duration**. The storage for these entities lasts for the **duration of the thread** in which they are created. There is a distinct object or reference per thread, and use of the declared name refers to the entity associated with the current thread.*

```cpp
thread_local int thread_local_variable; // thread storage

int main() { // at this point, one copy of thread_local_variable exists
```

think-cell

Thread storage duration

Definition

[basic.stc.thread]/1

> *All variables declared with the **`thread_local`** keyword have **thread storage duration**. The storage for these entities lasts for the **duration of the thread** in which they are created. There is a distinct object or reference per thread, and use of the declared name refers to the entity associated with the current thread.*

```cpp
thread_local int thread_local_variable; // thread storage

int main() { // at this point, one copy of thread_local_variable exists
    std::thread thr(…); // allocate another copy
```

think-cell

Thread storage duration

Definition

[basic.stc.thread]/1

> *All variables declared with the **`thread_local`** keyword have **thread storage duration**. The storage for these entities lasts for the **duration of the thread** in which they are created. There is a distinct object or reference per thread, and use of the declared name refers to the entity associated with the current thread.*

```
thread_local int thread_local_variable; // thread storage

int main() { // at this point, one copy of thread_local_variable exists
    std::thread thr(…); // allocate another copy
} // deallocate the copy
```

think-cell

Storage duration vs. lifetime

In general, the storage duration is not the same as the lifetime of an object!

Storage duration vs. lifetime

In general, the storage duration is not the same as the lifetime of an object!

[basic.life]/1

The lifetime of an object of type T begins when:

- *storage with the proper alignment and size for type T is obtained, and*
- *its initialization (if any) is complete*

Storage duration vs. lifetime

Automatic storage duration: storage duration and lifetime match[1].

[1] Terms and conditions may apply.

Storage duration vs. lifetime

Automatic storage duration: storage duration and lifetime match[1].

Static and thread storage duration: it's complicated

- function-local static vs global scope
- `constinit` vs. dynamic initialization
- nifty counters, module dependency graph, `inline` variables

www.jonathanmueller.dev/talk/static-initialization-order-fiasco/

[1] Terms and conditions may apply.

Object lifetime and initial value

In general, an object can have its lifetime start without a known value!

Object lifetime and initial value

Definition

[basic.indet]/1-2

> ***When storage*** *for an object with automatic or dynamic storage duration* ***is obtained****, the object has an* ***indeterminate value****, and if* ***no initialization*** *is performed for the object, that object* ***retains an indeterminate value*** *until that value is replaced.ass]). If an indeterminate value is* ***produced by an evaluation****, the* ***behavior is undefined****.*

think-cell

Object lifetime and initial value

Definition

[basic.indet]/1-2

> ***When storage*** *for an object with automatic or dynamic storage duration* ***is obtained****, the object has an* ***indeterminate value****, and if* ***no initialization*** *is performed for the object, that object* ***retains an indeterminate value*** *until that value is replaced.ass]). If an indeterminate value is* ***produced by an evaluation****, the* ***behavior is undefined****.*

```
int main() {
    int x; // start the lifetime with indeterminate value



}
```

think-cell

Object lifetime and initial value

Definition

[basic.indet]/1-2

> ***When storage*** *for an object with automatic or dynamic storage duration* ***is obtained****, the object has an* ***indeterminate value****, and if* ***no initialization*** *is performed for the object, that object* ***retains an indeterminate value*** *until that value is replaced.ass]). If an indeterminate value is* ***produced by an evaluation****, the* ***behavior is undefined****.*

```
int main() {
    int x; // start the lifetime with indeterminate value
    std::print("x = {}\n", x); // UB


}
```

think-cell

Object lifetime and initial value

Definition

[basic.indet]/1-2

> ***When storage*** *for an object with automatic or dynamic storage duration* ***is obtained****, the object has an* ***indeterminate value****, and if* ***no initialization*** *is performed for the object, that object* ***retains an indeterminate value*** *until that value is replaced.ass]). If an indeterminate value is* ***produced by an evaluation****, the* ***behavior is undefined****.*

```
int main() {
    int x; // start the lifetime with indeterminate value
    std::print("x = {}\n", x); // UB
    x = 11;
    std::print("x = {}\n", x); // okay
}
```

think-cell

Erroneous behavior

In C++26: read of indeterminate value is **erroneous**, not undefined.

P2795

> *If the execution contains an operation specified as having erroneous behavior, the implementation is permitted to issue a diagnostic and is permitted to terminate the execution at an unspecified time after that operation.*

Level 1: `new` and `delete`

Level 1: `new` and `delete`

Object creation

[intro.object]/1

An object is created [...] by a ***`new-expression`****.*

Level 1: `new` and `delete`

Object creation

[intro.object]/1

*An object is created […] by a **`new-expression`**.*

[expr.delete]/1

The `delete-expression` operator destroys a most derived object or array created by a `new-expression`.

think-cell

Level 1: `new` and `delete`

Object creation

[intro.object]/1

An object is created […] by a ***`new-expression`****.*

[expr.delete]/1

The `delete-expression` operator destroys a most derived object or array created by a `new-expression`.

```
int main() {
    int* ptr = new int(11); // create the object and start the lifetime


}
```

Level 1: new and delete

Object creation

[intro.object]/1

*An object is created [...] by a **new-expression**.*

[expr.delete]/1

The `delete-expression` operator destroys a most derived object or array created by a `new-expression`.

```
int main() {
    int* ptr = new int(11); // create the object and start the lifetime
    std::print("*ptr = {}\n", *ptr); // use the object
    ++*ptr; // use the object
    std::print("*ptr = {}\n", *ptr); // use the object

}
```

Level 1: `new` and `delete`

Object creation

[intro.object]/1

*An object is created [...] by a **`new-expression`**.*

[expr.delete]/1

The `delete-expression` operator destroys a most derived object or array created by a `new-expression`.

```
int main() {
    int* ptr = new int(11); // create the object and start the lifetime
    std::print("*ptr = {}\n", *ptr); // use the object
    ++*ptr; // use the object
    std::print("*ptr = {}\n", *ptr); // use the object
    delete ptr; // destroy the object and end the lifetime
}
```

Level 1: new and delete

Also possible to create one with indeterminate value:

```
int main() {
    int* ptr = new int; // start the lifetime with indeterminate value



}
```

Level 1: new and delete

Also possible to create one with indeterminate value:

```
int main() {
    int* ptr = new int; // start the lifetime with indeterminate value
    std::print("*ptr = {}\n", *ptr); // UB


}
```

Level 1: new and delete

Also possible to create one with indeterminate value:

```
int main() {
    int* ptr = new int; // start the lifetime with indeterminate value
    std::print("*ptr = {}\n", *ptr); // UB
    *ptr = 11;
    std::print("*ptr = {}\n", *ptr); // okay
}
```

Level 2: Temporary objects

think-cell

Level 2: Temporary objects

Object creation

[intro.object]/1

> *An object is created […] when a* ***temporary object*** *is created.*

Level 2: Temporary objects

Object creation

[intro.object]/1

*An object is created […] when a **temporary object** is created.*

```
void f(const int& ref);

int main() {
    f(42); // creation of temporary object
}
```

Temporary materialization conversion

Definition

[conv.rval]/1

*A **prvalue** of type T can be **converted to an xvalue** of type T. This conversion **initializes a temporary object** of type T from the prvalue by evaluating the prvalue with the temporary object as its result object, and produces an xvalue denoting the temporary object.*

think-cell

Temporary materialization conversion

Definition

[conv.rval]/1

*A **prvalue** of type T can be **converted to an xvalue** of type T. This conversion **initializes a temporary object** of type T from the prvalue by evaluating the prvalue with the temporary object as its result object, and produces an xvalue denoting the temporary object.*

Whenever a prvalue is used in a context where an xvalue is expected, a temporary object is created:

- binding a reference to a prvalue

think-cell

Temporary materialization conversion

Definition

[conv.rval]/1

> *A **prvalue** of type T can be **converted to an xvalue** of type T. This conversion **initializes a temporary object** of type T from the prvalue by evaluating the prvalue with the temporary object as its result object, and produces an xvalue denoting the temporary object.*

Whenever a prvalue is used in a context where an xvalue is expected, a temporary object is created:

- binding a reference to a prvalue
- member-access on a prvalue

Temporary materialization conversion

Definition

[conv.rval]/1

*A **prvalue** of type T can be **converted to an xvalue** of type T. This conversion **initializes a temporary object** of type T from the prvalue by evaluating the prvalue with the temporary object as its result object, and produces an xvalue denoting the temporary object.*

Whenever a prvalue is used in a context where an xvalue is expected, a temporary object is created:

- binding a reference to a prvalue
- member-access on a prvalue
- using an array prvalue

think-cell

Temporary materialization conversion

Definition

[conv.rval]/1

> *A **prvalue** of type T can be **converted to an xvalue** of type T. This conversion **initializes a temporary object** of type T from the prvalue by evaluating the prvalue with the temporary object as its result object, and produces an xvalue denoting the temporary object.*

Whenever a prvalue is used in a context where an xvalue is expected, a temporary object is created:

- binding a reference to a prvalue
- member-access on a prvalue
- using an array prvalue
- discarding the result of a function call that returns a prvalue

think-cell

Lifetime of a temporary

When a temporary is created, its lifetime starts.

think-cell

Lifetime of a temporary

When a temporary is created, its lifetime starts.

Object destruction

[class.temporary]/4

*Temporary objects are **destroyed as the last step in evaluating the full-expression** that (lexically) contains the point where they were created.*

```
void f(auto&& ... args);

int g();

int main() {
    f(11, g()); // two temporary objects are created
    //        ^ temporaries destroyed here
}
```

Temporary lifetime extension

1. When a reference is bound to a temporary, the lifetime of the temporary is extended to the lifetime of the reference.

```
int main() {
    const int& ref = 42; // temporary created here
    std::print("ref = {}\n", ref);
} // temporary destroyed here when ref is destroyed
```

think-cell

Temporary lifetime extension

Careful: It has to be a reference that directly binds to a temporary.

```
std::vector<std::string> get_strings();

int main() {
    const auto& strings = get_strings();    // extended
    const auto& string  = get_strings()[0]; // not extended
}
```

think-cell

Temporary lifetime extension

Careful: It has to be a reference that directly binds to a temporary.

```
std::vector<std::string> get_strings();

int main() {
    const auto& strings = get_strings();    // extended
    const auto& string  = get_strings()[0]; // not extended
}
```

```
template <typename T>
T& std::vector<T>::operator[](std::size_t idx);
```

Temporary lifetime extension

2 All temporaries created within the range expression of `for` are destroyed after the loop.

```
std::vector<std::string> get_strings();

int main() {
    for (auto&& str : get_strings()) {
        std::print("{}\n", str);
    } // temporary destroyed here




}
```

think-cell

Temporary lifetime extension

2 All temporaries created within the range expression of `for` are destroyed after the loop.

```
std::vector<std::string> get_strings();

int main() {
    for (auto&& str : get_strings()) {
        std::print("{}\n", str);
    } // temporary destroyed here

    for (auto&& c : get_strings()[0]) {
        std::print("{}\n", c);
    } // also okay, temporary destroyed here
}
```

think-cell

Level 3: Placement new

Level 3: Placement new

Object creation

[intro.object]/1

An object is created […] by a ***`new-expression`****.*

Level 3: Placement new

Object creation

[intro.object]/1

An object is created […] by a ***`new-expression`****.*

Placement new: explicit constructor call.

```
void* memory = …;
int* ptr = ::new(memory) int(11); // create an object
```

Manually create an object

Placement new

```
::new(static_cast<void*>(ptr)) T(…);
```

Placement new can be overloaded!

Manually create an object

Placement new

```
::new(static_cast<void*>(ptr)) T(…);
```

Placement new can be overloaded!

```
namespace std {
    template <typename T, typename ... Args>
    constexpr T* construct_at(T* ptr, Args&&... args) {
        return ::new(static_cast<void*>(ptr)) T(std::forward<Args>(args)...);
    }
}
```

think-cell

Manually destroy an object

Explicit destructor call

```
x.~T();
```

- syntax does not allow builtin types
- syntax does not allow namespace qualifiers

Manually destroy an object

Explicit destructor call

```
x.~T();
```

- syntax does not allow builtin types
- syntax does not allow namespace qualifiers

```
namespace std {
    template <typename T>
    void destroy_at(T* ptr) {
        ptr->~T();
    }
}
```

think-cell

How to allocate memory without creating objects?

How to allocate memory without creating objects?

1. `std::malloc`

```
void* memory = std::malloc(sizeof(int)); // allocate storage
int* ptr = ::new(memory) int(11);        // start lifetime
```

think-cell

How to allocate memory without creating objects?

1 `std::malloc`

```
void* memory = std::malloc(sizeof(int)); // allocate storage
int* ptr = ::new(memory) int(11);        // start lifetime
std::print("*ptr = {}\n", *ptr);         // use the object
```

think-cell

How to allocate memory without creating objects?

1. `std::malloc`

```
void* memory = std::malloc(sizeof(int)); // allocate storage
int* ptr = ::new(memory) int(11);        // start lifetime
std::print("*ptr = {}\n", *ptr);         // use the object
std::destroy_at(ptr);                    // end lifetime
```

think-cell

How to allocate memory without creating objects?

1 `std::malloc`

```
void* memory = std::malloc(sizeof(int)); // allocate storage
int* ptr = ::new(memory) int(11);        // start lifetime
std::print("*ptr = {}\n", *ptr);         // use the object
std::destroy_at(ptr);                    // end lifetime
std::free(memory);                       // deallocate storage
```

How to allocate memory without creating objects?

2 `::operator new`

```
void* memory = ::operator new(sizeof(int)); // allocate storage
```

think-cell

How to allocate memory without creating objects?

2 `::operator new`

```
void* memory = ::operator new(sizeof(int)); // allocate storage
int* ptr = ::new(memory) int(11);            // start lifetime
```

How to allocate memory without creating objects?

2 ::operator new

```
void* memory = ::operator new(sizeof(int)); // allocate storage
int* ptr = ::new(memory) int(11);           // start lifetime
std::print("*ptr = {}\n", *ptr);            // use the object
```

How to allocate memory without creating objects?

2 `::operator new`

```
void* memory = ::operator new(sizeof(int)); // allocate storage
int* ptr = ::new(memory) int(11);           // start lifetime
std::print("*ptr = {}\n", *ptr);            // use the object
std::destroy_at(ptr);                       // end lifetime
```

How to allocate memory without creating objects?

2 `::operator new`

```
void* memory = ::operator new(sizeof(int)); // allocate storage
int* ptr = ::new(memory) int(11);           // start lifetime
std::print("*ptr = {}\n", *ptr);            // use the object
std::destroy_at(ptr);                       // end lifetime
::operator delete(memory);                  // deallocate storage
```

think-cell

How to allocate memory without creating objects?

3. Array of `unsigned char` or `std::byte`

```
int main() {
  alignas(int) unsigned char buffer[sizeof(int)];        // allocate storage
```

think-cell

How to allocate memory without creating objects?

3. Array of `unsigned char` or `std::byte`

```
int main() {
  alignas(int) unsigned char buffer[sizeof(int)];       // allocate storage
  int* ptr = ::new(static_cast<void*>(buffer)) int(11); // start lifetime
```

How to allocate memory without creating objects?

3. Array of `unsigned char` or `std::byte`

```
int main() {
  alignas(int) unsigned char buffer[sizeof(int)];       // allocate storage
  int* ptr = ::new(static_cast<void*>(buffer)) int(11); // start lifetime
  std::print("*ptr = {}\n", *ptr);                      // use the object
```

think-cell

How to allocate memory without creating objects?

3. Array of `unsigned char` or `std::byte`

```
int main() {
  alignas(int) unsigned char buffer[sizeof(int)];         // allocate storage
  int* ptr = ::new(static_cast<void*>(buffer)) int(11); // start lifetime
  std::print("*ptr = {}\n", *ptr);                        // use the object
  std::destroy_at(ptr);                                   // end lifetime
```

think-cell

How to allocate memory without creating objects?

3. Array of `unsigned char` or `std::byte`

```
int main() {
  alignas(int) unsigned char buffer[sizeof(int)];        // allocate storage
  int* ptr = ::new(static_cast<void*>(buffer)) int(11); // start lifetime
  std::print("*ptr = {}\n", *ptr);                      // use the object
  std::destroy_at(ptr);                                 // end lifetime
}                                                       // deallocate storage
```

How to allocate memory without creating objects?

4 Re-use memory of an existing object

```
int main() {
    int x = 11;                                    // create an object
```

How to allocate memory without creating objects?

4. Re-use memory of an existing object

```
int main() {
    int x = 11;                                   // create an object
    std::destroy_at(&x);                          // end lifetime
```

How to allocate memory without creating objects?

4 Re-use memory of an existing object

```
int main() {
    int x = 11;                                          // create an object
    std::destroy_at(&x);                                 // end lifetime
    int* ptr = ::new(static_cast<void*>(&x)) int(42); // start lifetime
```

How to allocate memory without creating objects?

4 Re-use memory of an existing object

```
int main() {
    int x = 11;                                     // create an object
    std::destroy_at(&x);                            // end lifetime
    int* ptr = ::new(static_cast<void*>(&x)) int(42); // start lifetime
    std::print("*ptr = {}\n", *ptr);                // use the object
```

How to allocate memory without creating objects?

4 Re-use memory of an existing object

```
int main() {
    int x = 11;                                      // create an object
    std::destroy_at(&x);                             // end lifetime
    int* ptr = ::new(static_cast<void*>(&x)) int(42); // start lifetime
    std::print("*ptr = {}\n", *ptr);                 // use the object
}                                                    // end lifetime
```

think-cell

Be careful when re-using memory

const is const

[basic.life]/10

> ***Creating a new object*** *within the storage that a* **`const`***, complete object with static, thread, or automatic storage duration occupies, or within the storage that such a* `const` *object used to occupy before its lifetime ended, results in* ***undefined behavior****.*

```cpp
int main() {
    const int x = 11;
    std::destroy_at(&x);                          // end lifetime
    ::new(static_cast<void*>(&x)) int(42); // UB
}
```

think-cell

Be careful when re-using memory

const is const (mostly)

[basic.life]/10

> ***Creating a new object*** *within the storage that a* **`const`**, *complete object with static, thread, or automatic storage duration occupies, or within the storage that such a* `const` *object used to occupy before its lifetime ended, results in* ***undefined behavior***.

```cpp
int main() {
    const int* ptr = new const int(11);
    std::destroy_at(ptr);                    // end lifetime
    ::new(static_cast<void*>(ptr)) int(42); // okay
}
```

think-cell

Be careful when re-using memory

The destructor still runs

[basic.life]/9

> *If a program **ends the lifetime** of an object of type T with static, thread, or automatic storage duration and if T has a **non-trivial destructor**, and **another object** of the original type **does not occupy** that same storage location when the **implicit destructor call** takes place, the **behavior of the program is undefined**.*

```
int main() {
    std::string str = "non-trivial destructor";
    std::destroy_at(&str); // end lifetime
} // end lifetime again...
```

think-cell

Be careful when re-using memory

```
int main() {
    int x = 11;
    std::destroy_at(&x);
    int* ptr = ::new(static_cast<void*>(&x)) int(42);
    std::print("*ptr = {}\n", *ptr); // okay

}
```

think-cell

Be careful when re-using memory

```
int main() {
    int x = 11;
    std::destroy_at(&x);
    int* ptr = ::new(static_cast<void*>(&x)) int(42);
    std::print("*ptr = {}\n", *ptr); // okay
    std::print("x = {}\n", x);       // also okay?
}
```

think-cell

Transparent replacement of objects

Definition

[basic.life]/8

> *If, **after the lifetime of an object has ended** and before the storage which the object occupied is reused or released, a **new object is created** at the storage location which the original object occupied, a **pointer that pointed to the original object**, a **reference that referred to the original object**, or the **name of the original object** will **automatically refer to the new object** and, once the lifetime of the new object has started, can be used to manipulate the new object, **if the original object is transparently replaceable by the new object**.*

think-cell

Transparent replacement of objects

Definition

a is transparently replacable by b if:

- a and b use the same storage, and
- a and b have the same type (ignoring top-level cv-qualifiers)

Transparent replacement of objects

Definition

a is transparently replacable by b if:

- a and b use the same storage, and
- a and b have the same type (ignoring top-level cv-qualifiers)

However, you cannot transparently replace:

- `const` objects
- base classes
- `[[no_unique_address]]` members

think-cell

Transparent replacement of objects

Definition

`a` is transparently replacable by `b` if:

- `a` and `b` use the same storage, and
- `a` and `b` have the same type (ignoring top-level `cv`-qualifiers)

However, you cannot transparently replace:

- `const` objects
- base classes
- `[[no_unique_address]]` members

When replacing subobjects (member variables or array elements), the rules apply recursively to the parent object.

think-cell

Transparent replacement of objects

```
int main() {
  int x = 11;
  std::destroy_at(&x);
  ::new(static_cast<void*>(&x)) int(42); // transparent replacement
  std::print("x = {}\n", x);             // okay
}
```

think-cell

Transparent replacement of objects

```
foo& foo::operator=(const foo& other) {
  std::destroy_at(this);
  ::new(static_cast<void*>(this)) foo(other); // transparent replacement
  return *this;                               // okay
}
```

Transparent replacement of objects

```
struct foo {
  int x;

  void f() {
    std::destroy_at(&x);
    ::new(static_cast<void*>(&x)) int(42); // transparent replacement
  }
};
```

Non-transparent replacement of objects

```cpp
const int* ptr = new const int(11);
std::destroy_at(ptr);
int* new_ptr = ::new(static_cast<void*>(ptr)) const int(42); // non-transparent
std::print("*new_ptr = {}\n", *new_ptr);                     // okay
std::print("*ptr = {}\n", *ptr);                             // UB
```

std::launder to the rescue

```
namespace std {
    template <typename T>
    T* launder(T* ptr) noexcept; // magic identity function
}
```

std::launder to the rescue

```
namespace std {
    template <typename T>
    T* launder(T* ptr) noexcept; // magic identity function
}
```

```
const int* ptr = new const int(11);
std::destroy_at(ptr);
auto new_ptr = ::new(static_cast<void*>(ptr)) int(42); // non-transparent
std::print("*new_ptr = {}\n", *new_ptr);                // okay
std::print("*ptr = {}\n", *std::launder(ptr));          // okay
```

std::launder even helps with references

```
const int* ptr = new const int(11);
const int& ref = *ptr;
std::destroy_at(ptr);
::new(static_cast<void*>(ptr)) int(42);         // non-transparent
std::print("ref = {}\n", ref);                  // UB
std::print("ref = {}\n", *std::launder(&ref)); // okay
```

std::launder does not prevent UB

```cpp
static_assert(
  sizeof(float) == sizeof(int) && alignof(float) == alignof(int)
);

int main() {
  float* f_ptr = new float(3.14f);
  std::destroy_at(f_ptr);
  int* i_ptr = ::new(static_cast<void*>(f_ptr)) int(42); // non-transparent
  std::print("*i_ptr = {}\n", *i_ptr);                   // okay
  std::print("*f_ptr = {}\n", *f_ptr);                   // UB
  std::print("*f_ptr = {}\n", *std::launder(f_ptr));     // still UB
}
```

When do I need to use `std::launder`?

When you want to re-use the storage of

- `const` *heap* objects,
- base classes, or
- `[[no_unique_address]]` members.

When do I need to use `std::launder`?

When you want to re-use the storage of

- `const` *heap* objects,
- base classes, or
- `[[no_unique_address]]` members.

Never?

think-cell

Level 4: Implicit object creation

Level 4: Implicit object creation

Object creation

[intro.object]/1

*An object is created […] by an **operation that implicitly creates** objects.*

Level 4: Implicit object creation

Object creation

[intro.object]/1

An object is created […] by an ***operation that implicitly creates*** *objects.*

```
int* ptr = static_cast<int*>(std::malloc(sizeof(int)));
*ptr = 11; // should not be UB
```

Implicit object creation

Definition

[intro.object]/11

> *For each operation that is specified as implicitly creating objects, that operation* ***implicitly creates and starts the lifetime*** *of zero or more objects of* ***implicit-lifetime types*** *in its specified region of storage* ***if doing so*** *would result in the program having* ***defined behavior****. If no such set of objects would give the program defined behavior, the behavior of the program is undefined. If multiple such sets of objects would give the program defined behavior, it is unspecified which such set of objects is created.*

think-cell

Implicit object creation

Definition

[intro.object]/11

> *For each operation that is specified as implicitly creating objects, that operation **implicitly creates and starts the lifetime** of zero or more objects of **implicit-lifetime types** in its specified region of storage **if doing so** would result in the program having **defined behavior**. If no such set of objects would give the program defined behavior, the behavior of the program is undefined. If multiple such sets of objects would give the program defined behavior, it is unspecified which such set of objects is created.*

If it helps you, the compiler creates objects for you.

think-cell

Implicit lifetime type

Definition

[basic.types.general]/9

> ***Scalar types****, implicit-lifetime class types, array types, and cv-qualified versions of these types are collectively called implicit-lifetime types.*

[class.prop]/9

> *A class `S` is an implicit-lifetime class if*
>
> - *it is an* ***aggregate*** *whose destructor is not user-provided or*
> - *it has at least one* ***trivial eligible constructor*** *and a* ***trivial, non-deleted destructor****.*

Implicit lifetime type

Definition

[basic.types.general]/9

__Scalar types__, implicit-lifetime class types, array types, and cv-qualified versions of these types are collectively called implicit-lifetime types.

[class.prop]/9

A class `S` is an implicit-lifetime class if

- *it is an __aggregate__ whose destructor is not user-provided or*
- *it has at least one __trivial eligible constructor__ and a __trivial, non-deleted destructor__.*

Construction and destruction do nothing.

think-cell

Which operations implicitly create objects?

1. `std::malloc` and variants, `::operator new`, `std::allocator::allocate`, and other allocation functions.

```
int* ptr = static_cast<int*>(std::malloc(sizeof(int))); // create an int
*ptr = 11;
```

think-cell

Which operations implicitly create objects?

2 Anything that starts the lifetime of an `unsigned char`/`std::byte` array.

```
alignas(int) unsigned char buffer[sizeof(int)]; // create an int
int* ptr = reinterpret_cast<int*>(buffer);
*ptr = 11;
```

Which operations implicitly create objects?

2. Anything that starts the lifetime of an `unsigned char`/`std::byte` array.

```
alignas(int) unsigned char buffer[sizeof(int)]; // create an int
int* ptr = std::launder(reinterpret_cast<int*>(buffer));
*ptr = 11;
```

P3006 makes `std::launder` unnecessary here.

think-cell

Which operations implicitly create objects?

3. `std::memcpy` and `std::memmove`

```
alignas(int) char buffer[sizeof(int)];        // creates nothing
std::memcpy(buffer, &some_int, sizeof(int)); // create an int
int* ptr = std::launder(reinterpret_cast<int*>(buffer));
std::print("*ptr = {}\n", ptr);              // okay
```

think-cell

Which operations implicitly create objects?

4. Implementation-defined set of operations like `mmap` or `VirtualAlloc`.

```
int* ptr = static_cast<int*>(mmap(…)); // create an int
std::print("*ptr = {}\n", *ptr);       // okay
```

Implicit object creation uses time travel

```
static_assert(
  sizeof(int) == sizeof(float) && alignof(int) == alignof(float)
);

alignas(int) unsigned char buffer[sizeof(int)]; // create int or float
if (rand() % 2)
    *std::launder(reinterpret_cast<int*>(buffer)) = 11;
else
    *std::launder(reinterpret_cast<float*>(buffer)) = 3.14f;
```

think-cell

Implicit object creation does not prevent UB

```cpp
static_assert(
  sizeof(int) == sizeof(float) && alignof(int) == alignof(float)
);

int i = 11;
float f = *std::launder(reinterpret_cast<float*>(&i)); // still UB
```

think-cell

Explicit implicit object creation

```cpp
struct data {
    std::uint8_t op;
    std::uint32_t a, b, c;
};
```

```cpp
void process(unsigned char* buffer, std::size_t size) {
    data* ptr = std::launder(reinterpret_cast<data*>(buffer));
    std::print("*ptr = {}\n", *ptr); // might be UB
}
```

Explicit implicit object creation

```
struct data {
    std::uint8_t op;
    std::uint32_t a, b, c;
};
```

```
void process(unsigned char* buffer, std::size_t size) {
    data* ptr = ::new(static_cast<void*>(buffer)) data;
    std::print("*ptr = {}\n", *ptr); // okay, but could be wrong
}
```

Explicit implicit object creation

```
struct data {
    std::uint8_t op;
    std::uint32_t a, b, c;
};
```

```
void process(unsigned char* buffer, std::size_t size) {
    data* ptr = std::start_lifetime_as<data>(buffer);
    std::print("*ptr = {}\n", *ptr); // okay
}
```

Also `std::start_lifetime_as_array<data>(ptr, count)`.

Implementing `std::start_lifetime_as`

```
template <typename T>
T* start_lifetime_as(void* ptr) {
    std::memmove(ptr, ptr, sizeof(T));
    return std::launder(static_cast<T*>(ptr));
}
```

Standard library implementation is a no-op that also works for `const`.

Implicit destruction of objects

Definition

[basic.life]/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`.*

Implicit destruction of objects

Definition

[basic.life]/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`.*

```
int main() {
  int x = 11;
  ::new(static_cast<void*>(&x)) int(42); // end + start lifetime
  std::print("x = {}\n", x);
}
```

think-cell

Implicit destruction of objects

Definition

[basic.life]/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`.*

```
int main() {
  alignas(int) unsigned char buffer[sizeof(int)]; // start lifetime
  int* ptr = ::new(static_cast<void*>(buffer)) int(11); // end + start lifetime
  std::print("*ptr = {}\n", *ptr);
}
```

think-cell

Memory leaks are not undefined behavior

```cpp
int main() {
  std::string str = "long string so we don't have SSO";
  ::new(static_cast<void*>(&str)) std::string("a different long string");
  std::print("str = {}\n", str);
}
```

think-cell

Level 5: Provenance

Pointers aren't just addresses

```
void do_sth(int* ptr);
```

```
int foo() {
    int x, y;
    y = 11;

    do_sth(&x);
    return y; // optimize to return 11
}
```

Pointers aren't just addresses

```
void do_sth(int* ptr);
```

```
int foo() {
    int x, y;
    y = 11;

    do_sth(&x);
    return y; // optimize to return 11
}
```

```
void do_sth(int* ptr) {
    *(ptr + 1) = 42;
}
```

Pointers aren't just addresses

```
void do_sth(int* ptr);
```

```
int foo() {
    int x, y;
    y = 11;
    if (&x + 1 == &y)
        do_sth(&x);
    return y; // optimize to return 11!?
}
```

```
void do_sth(int* ptr) {
    *(ptr + 1) = 42;
}
```

think-cell

Just because two pointers are equal doesn't mean they point to the same object!

Pointer provenance

Definition

A pointer `T*` is logically a pair `(address, provenance)`:

- The `address` is the only thing that is physically observable.
- The `provenance` identifies to the object or allocation the pointer was derived from.

Pointer provenance

Definition

A pointer `T*` is logically a pair `(address, provenance)`:

- The `address` is the only thing that is physically observable.
- The `provenance` identifies to the object or allocation the pointer was derived from.

A pointer dereference is only valid if:

- The address is in the range of allowed addresses for the provenance.
- The current provenance of that address is the same as the provenance of the pointer.

think-cell

Pointer provenance

Definition

A pointer `T*` is logically a pair `(address, provenance)`:

- The `address` is the only thing that is physically observable.
- The `provenance` identifies to the object or allocation the pointer was derived from.

A pointer dereference is only valid if:

- The address is in the range of allowed addresses for the provenance.
- The current provenance of that address is the same as the provenance of the pointer.

The pointer provenance cannot be changed using pointer arithmetic!

think-cell

Pointer provenance

Address not in range:

```
int foo() {
    int x, y;
    y = 11;
    if (&x + 1 == &y)
        do_sth(&x);
    return y;
}

void do_sth(int* ptr) {
    *(ptr + 1) = 42; // UB, address not in range
}
```

think-cell

Pointer provenance

Different provenance:

```
const int* ptr = new const int(11);                    // provenance A
std::destroy_at(ptr);
int* new_ptr = ::new(static_cast<void*>(ptr)) int(42); // provenance B
```

```
std::print("*new_ptr = {}\n", *new_ptr);        // okay
```

think-cell

Pointer provenance

Different provenance:

```
const int* ptr = new const int(11);                      // provenance A
std::destroy_at(ptr);
int* new_ptr = ::new(static_cast<void*>(ptr)) int(42); // provenance B
```

```
std::print("*new_ptr = {}\n", *new_ptr);        // okay
std::print("*ptr = {}\n", *ptr);                // UB
```

think-cell

Pointer provenance

Different provenance:

```cpp
const int* ptr = new const int(11);                        // provenance A
std::destroy_at(ptr);
int* new_ptr = ::new(static_cast<void*>(ptr)) int(42); // provenance B
```

```cpp
std::print("*new_ptr = {}\n", *new_ptr);        // okay
std::print("*ptr = {}\n", *ptr);                // UB
std::print("*ptr = {}\n", *std::launder(ptr)); // okay
```

think-cell

References have provenance too

```
const int* ptr = new const int(11);
const int& ref = *ptr;
std::destroy_at(ptr);
::new(static_cast<void*>(ptr)) int(42);

std::print("ref = {}\n", ref);                  // UB
std::print("ref = {}\n", *std::launder(&ref)); // okay
```

think-cell

Provenance changes

Provenance

- Each object has a unique provenance.
- All objects in an array have the same provenance.
- Re-using the memory of an object changes the provenance unless the object is transparently replaced.

think-cell

Provenance changes

Provenance

- Each object has a unique provenance.
- All objects in an array have the same provenance.
- Re-using the memory of an object changes the provenance unless the object is transparently replaced.

Use `std::launder` to update the provenance of an object.

think-cell

Level 6: Type punning

think-cell

Strict aliasing

```
static_assert(
  sizeof(int) == sizeof(float) && alignof(int) == alignof(float)
);
```

```
int i = 11;
int* i_ptr = &i;                              // okay
float* f_ptr = reinterpret_cast<float*>(i_ptr); // okay?
std::print("*f_ptr = {}\n", *f_ptr);          // UB
```

think-cell

Strict aliasing violations

Colloquial: You can't `reinterpret_cast` between unrelated types.

Strict aliasing violations

Colloquial: You can't `reinterpret_cast` between unrelated types.

Strict aliasing rule

[basic.lval]/11

> *If a program attempts to **access** the stored value of **an object through a glvalue** whose **type is not similar to** one of the following types the behavior is undefined:*
>
> - *the dynamic **type of the object**, [...]*

You can't **access** an object through a pointer of an unrelated type.

think-cell

General rule

The type of a pointer or reference is only relevant when accessing the referred object.

No strict aliasing violation

```cpp
int i = 11;
float* f_ptr = ::new(static_cast<void*>(&i)) float(3.14);
std::print("*f_ptr = {}\n", *f_ptr); // okay
std::print("i = {}\n", i);           // UB
```

No strict aliasing violation

```cpp
int i = 11;
float* f_ptr = std::start_lifetime_as<float>(&i);
std::print("*f_ptr = {}\n", *f_ptr); // okay
std::print("i = {}\n", i);           // UB
```

Be careful about getting the pointer

```
int i = 11;
float* f_ptr = reinterpret_cast<float*>(&i);
::new(static_cast<void*>(&i)) float(3.14);
std::print("*f_ptr = {}\n", *f_ptr); // UB
```

Be careful about getting the pointer

```
int i = 11;
::new(static_cast<void*>(&i)) float(3.14);
float* f_ptr = reinterpret_cast<float*>(&i);
std::print("*f_ptr = {}\n", *f_ptr); // UB
```

Be careful about getting the pointer

```
int i = 11;
float* f_ptr = ::new(static_cast<void*>(&i)) float(3.14);
std::print("*f_ptr = {}\n", *f_ptr); // okay
```

Be careful about getting the pointer

```
int i = 11;
float* f_ptr = reinterpret_cast<float*>(&i);
::new(static_cast<void*>(&i)) float(3.14);
std::print("*f_ptr = {}\n", *std::launder(f_ptr)); // okay
```

think-cell

This currently also applies to `unsigned char`

```
alignas(int) unsigned char buffer[sizeof(int)];
int* ptr = reinterpret_cast<int*>(buffer);
*ptr = 11;
```

This currently also applies to `unsigned char`

```
alignas(int) unsigned char buffer[sizeof(int)];
int* ptr = reinterpret_cast<int*>(buffer);
*ptr = 11;
```

I consider this a bug in the standard, P3006 fixes it.

When do I need to use `std::launder`?

When you want to re-use the storage of

- `const` *heap* objects,
- base classes,
- `[[no_unique_address]]` members.

Or when re-using memory as storage for a different type.

When do I need to use `std::launder`?

When you want to re-use the storage of

- `const` *heap* objects,
- base classes,
- `[[no_unique_address]]` members.

Or when re-using memory as storage for a different type.

Never[2].

[2] Terms and conditions may apply

think-cell

Strict aliasing exceptions

Strict aliasing rule

[basic.lval]/11

If a program attempts to access the stored value of an object through a glvalue whose type is not similar to one of the following types the behavior is undefined:

- *the dynamic type of the object,*
- *a type that is the `signed` or `unsigned` type corresponding to the dynamic type of the object, or*
- *a `char`, `unsigned char`, or `std::byte` type.*

```
int i = -1;
std::print("{}\n", *reinterpret_cast<unsigned*>(&i));  // okay
std::print("{}\n", *reinterpret_cast<std::byte*>(&i)); // okay
```

think-cell

Object representation

Idea: Allow access to the *object representation*, the sequence of bytes the object represents in memory.

```
int object = 11;
std::byte* ptr = reinterpret_cast<std::byte*>(&object);
for (auto i = 0z; i != sizeof(object); ++i) {
    std::print("{:02x} ", static_cast<int>(*ptr++));
}
```

think-cell

Object representation

Idea: Allow access to the *object representation*, the sequence of bytes the object represents in memory.

```
int object = 11;
std::byte* ptr = reinterpret_cast<std::byte*>(&object);
for (auto i = 0z; i != sizeof(object); ++i) {
    std::print("{:02x} ", static_cast<int>(*ptr++));
}
```

In practice: Currently UB due to a bug in the standard, P1839 fixes it.

think-cell

Type punning

Goal: Interpret bytes stored in object of type T1 as an object of type T2.

Type punning with `std::start_lifetime_as` works

```cpp
int i = 11;
std::print("float = {}\n", *reinterpret_cast<float*>(&i));       // UB
std::print("float = {}\n", *std::start_lifetime_as<float>(&i)); // okay
```

Type punning with std::start_lifetime_as works

```
int i = 11;
std::print("float = {}\n", *reinterpret_cast<float*>(&i));       // UB
std::print("float = {}\n", *std::start_lifetime_as<float>(&i)); // okay
```

You can't use the old name/pointer/reference after the cast!

think-cell

Type punning with std::start_lifetime_as works

```
int i = 11;
std::print("float = {}\n", *reinterpret_cast<float*>(&i));       // UB
std::print("float = {}\n", *std::start_lifetime_as<float>(&i)); // okay
```

You can’t use the old name/pointer/reference after the cast!

```
int i = 11;
std::print("float = {}\n", *std::start_lifetime_as<float>(&i)); // okay
std::start_lifetime_as<int>(&i);
std::print("int = {}\n", i); // UB!
std::print("int = {}\n", *std::launder(&i)); // okay
```

think-cell

Type punning via std::memcpy works

```
int i = 11;
float f;
std::memcpy(&f, &i, sizeof(f));
std::print("f = {}\n", f); // okay
std::print("i = {}\n", i); // also okay
```

Type punning via `std::bit_cast` works

```
int i = 11;
float f = std::bit_cast<float>(i);
std::print("f = {}\n", f); // okay
std::print("i = {}\n", i); // also okay
```

think-cell

Type punning for parent struct works

```cpp
struct A {
    int member;
};

A a{.member = 11};
int* i_ptr = reinterpret_cast<int*>(&a);
std::print("*i_ptr = {}\n", *i_ptr); // okay
std::print("member = {}\n",
    reinterpret_cast<A*>(i_ptr)->member  // also okay
);
```

think-cell

Type punning for parent struct works

Definition

[basic.compound]/4

> *If two objects are **pointer-interconvertible**, then they have the **same address**, and it is possible to obtain a pointer to one from a pointer to the other via a `reinterpret_cast`.*

Type punning for parent struct works

Definition

[basic.compound]/4

> *If two objects are **pointer-interconvertible**, then they have the **same address**, and it is possible to obtain a pointer to one from a pointer to the other via a `reinterpret_cast`.*

[basic.compound]/4

> *Two objects `a` and `b` are pointer-interconvertible if:*
>
> - *they are the same object, or*
> - *one is a **union object** and the other is a **non-static data member** of that object, or*
> - *one is a **standard-layout class object** and the other is the **first non-static data member** of that object or any base class subobject of that object, or*
> - *there exists an object `c` such that `a` and `c` are pointer-interconvertible, and `c` and `b` are pointer-interconvertible.*

think-cell

Type punning via `union` does not work

Object creation

[intro.object]/1

> *An object is created […] when implicitly* ***changing the active member of a union****.*

Type punning via union does not work

Object creation

[intro.object]/1

*An object is created […] when implicitly **changing the active member of a union**.*

```
union U {
    int i;
    float f;
};
U u{.i = 11};
u.f = 3.14f;
std::print("u.f = {}\n", u.f); // okay
std::print("u.i = {}\n", u.i); // UB
```

Aside: Common initial sequence

Definition

[class.mem.general]/26

> *In a standard-layout union with an active member of struct type* `T1`*, it is permitted to* ***read a non-static data member*** `m` ***of another union member*** *of struct type* `T2` ***provided*** `m` ***is part of the common initial sequence*** *of* `T1` *and* `T2`*; the behavior is as if the corresponding member of* `T1` *were nominated.*

think-cell

Aside: Common initial sequence

```
union U {
    struct A {
        int prefix;
        int i;
    } a;
    struct B {
        int prefix;
        float f;
    } b;
};
U u{.a = {.prefix = 0, .i = 11}};
std::print("prefix = {}\n", u.a.prefix); // okay
std::print("prefix = {}\n", u.b.prefix); // okay
```

think-cell

Level 7: Invalid and zombie pointers

Access outside of lifetime

[basic.life]/6

> ***Before the lifetime*** *of an object has started but after the storage which the object will occupy has been allocated or,* ***after the lifetime*** *of an object has ended and before the storage which the object occupied is reused or released,* ***any pointer*** *that represents the address of the storage location where the object will be or was located* ***may be used but only in limited ways****.*

[basic.life]/7

> *Similarly,* ***before the lifetime of an object*** *has started but after the storage which the object will occupy has been allocated or,* ***after the lifetime*** *of an object has ended and before the storage which the object occupied is reused or released,* ***any glvalue*** *that refers to the original object* ***may be used but only in limited ways****.*

think-cell

Access outside of lifetime

```cpp
int x = 11;
std::destroy_at(&x);

int* ptr1 = &x;     // okay
int& ref1 = x;      // okay
int* ptr2 = &ref1; // okay
int& ref2 = *ptr2; // okay

assert(ptr1 == ptr2); // okay
```

think-cell

Access outside of lifetime

```cpp
int x = 11;
std::destroy_at(&x);

int y = x;     // UB
int* ptr = &x;
int z = *ptr; // UB
```

Access outside of lifetime

Undefined behavior if such a pointer or reference is used:

- to access the value of the object,

Access outside of lifetime

Undefined behavior if such a pointer or reference is used:

- to access the value of the object,
- to call a non-static member function on the object,

think-cell

Access outside of lifetime

Undefined behavior if such a pointer or reference is used:

- to access the value of the object,
- to call a non-static member function on the object,
- to delete the object,

think-cell

Access outside of lifetime

Undefined behavior if such a pointer or reference is used:

- to access the value of the object,
- to call a non-static member function on the object,
- to delete the object,
- for anything to do with `virtual` base classes or `dynamic_cast`.

Access outside of lifetime

[basic.life]/6

> *Before the lifetime of an object has started but **after the storage which the object will occupy has been allocated** or, after the lifetime of an object has ended and **before the storage which the object occupied is reused or released**, any pointer that represents the address of the storage location where the object will be or was located may be used but only in limited ways.*

[basic.life]/7

> *Similarly, before the lifetime of an object has started but **after the storage which the object will occupy has been allocated** or, after the lifetime of an object has ended and **before the storage which the object occupied is reused or released**, any glvalue that refers to the original object may be used but only in limited ways.*

think-cell

Invalid pointers

Pointer lifetime-end zap

[basic.stc.general]/4

> *When the **end of the duration of a region of storage** is reached, the values of **all pointers** representing the address of any part of that region of storage **become invalid pointer values**. **Indirection** through an invalid pointer value and passing an invalid pointer value to a **deallocation** function **have undefined behavior**. Any other **use of an invalid pointer value has implementation-defined behavior**.*

[basic.stc.general]/Footnote 24

> *Some implementations might define that copying an invalid pointer value causes a system-generated runtime fault.*

think-cell

Invalid pointers

```cpp
int main() {
    int x = 11;
    int* ptr = &x;
    std::destroy_at(&x);
    std::print("*ptr = {}\n", *ptr);    // UB
    std::print("ptr == nullptr? {}\n",
        ptr == nullptr                  // okay
    );
}
```

Invalid pointers

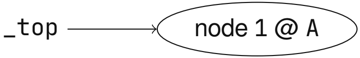
```
int main() {

    int* ptr = new int;
    delete ptr;
    std::print("*ptr = {}\n", *ptr);    // UB
    std::print("ptr == nullptr? {}\n",
        ptr == nullptr                  // implementation-defined
    );
}
```

LIFO push

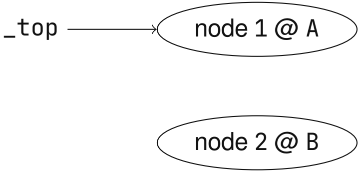
Taken from P2414R2.

```
struct list {
    std::atomic<node*> _top;
    void push(node* new_node) {
        while (true) {
            auto old_top = _top.load();
            new_node->set_next(old_top);
            if (_top.compare_exchange_weak(old_top, new_node)) return;
        }
    }
    node* pop_all() {
        return _top.exchange(nullptr);
    }
};
```

LIFO push can lead to invalid pointer



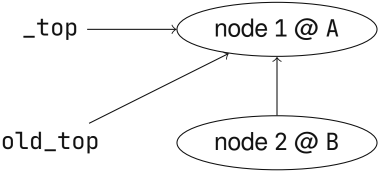
LIFO push can lead to invalid pointer



Thread T1

- Allocate node 2.

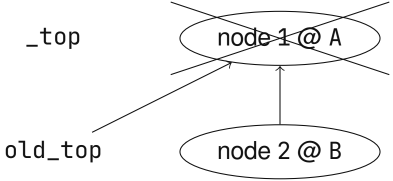
LIFO push can lead to invalid pointer



Thread T1

- Allocate node 2.
- Execute `auto old_top = _top.load()` and `new_node->set_next(old_top)`.

LIFO push can lead to invalid pointer



Thread T1

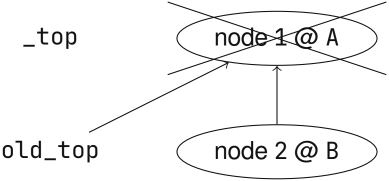
- Allocate node 2.
- Execute `auto old_top = _top.load()` and `new_node->set_next(old_top)`.

Thread T2

- Execute `pop_all()`.
- Delete node 1.

think-cell

LIFO push can lead to invalid pointer



Thread T1

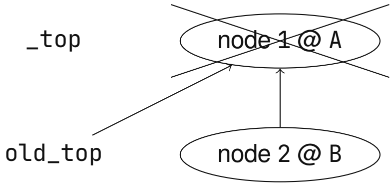
- Allocate node 2.
- Execute `auto old_top = _top.load()` and `new_node->set_next(old_top)`.
- Execute `compare_exchange_weak`; implementation-defined!

Thread T2

- Execute `pop_all()`.
- Delete node 1.

think-cell

LIFO push can lead to zombie pointer



Thread T1

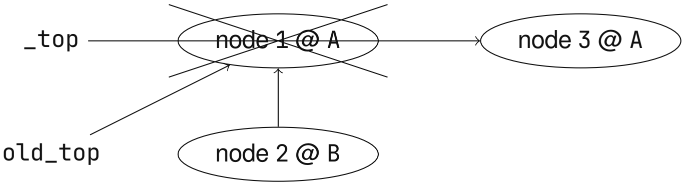
- Allocate node 2.
- Execute `auto old_top = _top.load()` and `new_node->set_next(old_top)`.

Thread T2

- Execute `pop_all()`.
- Delete node 1.

think-cell

LIFO push can lead to zombie pointer



Thread T1

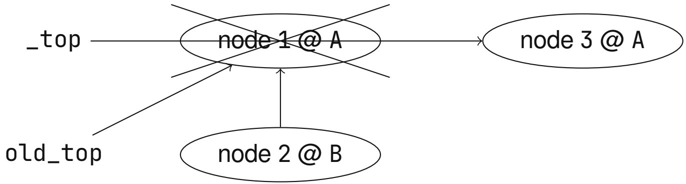
- Allocate node 2.
- Execute `auto old_top = _top.load()` and `new_node->set_next(old_top)`.

Thread T2

- Execute `pop_all()`.
- Delete node 1.
- Allocate node 3 which re-uses location of 1.

think-cell

LIFO push can lead to zombie pointer



Thread T1

- Allocate node 2.
- Execute `auto old_top = _top.load()` and `new_node->set_next(old_top)`.
- Execute `compare_exchange_weak`; implementation-defined!

Thread T2

- Execute `pop_all()`.
- Delete node 1.
- Allocate node 3 which re-uses location of 1.

think-cell

LIFO push can lead to zombie pointer

What is the result of the comparison?

- If the comparison returns `true` because the addresses are the same, node 2's next pointer has the wrong provenance for access.
- If the comparison returns `false` because the provenance is different, it is not implementable in hardware.

Solution

This is a bug in the C++ standard.

This is a bug in the C++ standard.

Proposed solution:

- Make comparison of invalid pointer values meaningful.
- Introduce some `std::launder`-like mechanism to fix the provenance of all pointers reachable from a root.

think-cell

Conclusion

Personal guidelines

- Don't rely on implicit object creation:

Personal guidelines

- Don’t rely on implicit object creation:
  - Use placement new to explicitly create a new object.

Personal guidelines

- Don't rely on implicit object creation:
  - Use placement new to explicitly create a new object.
  - Use `std::start_lifetime_as` to re-interpret raw bytes as an object.

Personal guidelines

- Don't rely on implicit object creation:
  - Use placement new to explicitly create a new object.
  - Use `std::start_lifetime_as` to re-interpret raw bytes as an object.
- Whenever possible, use the pointer from placement new and `std::start_lifetime_as` directly.

Personal guidelines

- Don't rely on implicit object creation:
  - Use placement new to explicitly create a new object.
  - Use `std::start_lifetime_as` to re-interpret raw bytes as an object.
- Whenever possible, use the pointer from placement new and `std::start_lifetime_as` directly.
- Use `union { char empty; T t; }` instead of `alignas(T) unsigned char buffer[sizeof(T)]`.

think-cell

Conclusion

We're hiring: think-cell.com/cppnow

Developer blog: think-cell.com/en/career/devblog/overview

jonathanmueller.dev/talk/lifetime

@foonathan@fosstodon.org
youtube.com/@foonathan

think-cell