Let's get schizophrenic

...by doing our laundry

Matthis Kruse
@Natriumchlorid_

Saarland University

```
struct Paul
{
virtual int do_life();
};

struct Sarah : Paul
{
int do_life() override;
};
```

```
#include <new>

struct Paul
{
virtual int do_life();
};

struct Sarah : Paul
{
int do_life() override
{ new(this) Paul; return 2; }
};

int Paul::do_life()
{ new(this) Sarah; return 1; }
```

```
int main(int argc, char** argv)
{
Sarah sarah;

const int n = sarah.do_life();
const int m = sarah.do_life();

return n + m; // is ???
}
```

```
int main(int argc, char** argv)
{
Sarah sarah;

const int n = sarah.do_life();
const int m = sarah.do_life();

return n + m; // is ???
}
```

Undefined Behavior

6.6.3.8 basic.life

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 storage for the new object exactly overlays the storage location which the original object occupied, and
- the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
- the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
- neither the original object nor the new object is a potentially-overlapping subobject.

6.6.3.8 basic.life

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 storage for the new object exactly overlays the storage location which the original object occupied, and
- the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
- the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
- neither the original object nor the new object is a potentially-overlapping subobject.

6.6.3.8 basic.life

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 storage for the new object exactly overlays the storage location which the original object occupied, and
- the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
- the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
- neither the original object nor the new object is a potentially-overlapping subobject.

6.6.3.8 basic.life

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 storage for the new object exactly overlays the storage location which the original object occupied, and
- the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
- the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
- neither the original object nor the new object is a potentially-overlapping subobject.

6.6.3.8 basic.life

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 storage for the new object exactly overlays the storage location which the original object occupied, and
- the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
- the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
- neither the original object nor the new object is a potentially-overlapping subobject.

std::launder to the rescue!

On std::launder()

```
template <class T>
constexpr T* launder(T* p) noexcept; // (since C++17)
```

On std::launder()

```
template <class T>
constexpr T* launder(T* p) noexcept; // (since C++17)

template <class T> // (since C++20)
[[nodiscard]] constexpr T* launder(T* p) noexcept;
```

On std::launder()

```
template <class T>
constexpr T* launder(T* p) noexcept; // (since C++17)

template <class T> // (since C++20)
[[nodiscard]] constexpr T* launder(T* p) noexcept;
```

- the pointer p represents the address A of a byte in memory
- an object X is located at the address A
- X is within its lifetime
- the type of X is the same as T, ignoring cv-qualifiers at every level
- every byte that would be reachable through the result is reachable through p

```cpp
int main(int argc, char** argv)
{
Sarah sarah;

const int n = sarah.do_life();
const int m = std::launder(&sarah)->do_life();

return n + m; // is ???
}
```

```cpp
int main(int argc, char** argv)
{
Sarah sarah;

const int n = sarah.do_life();
const int m = std::launder(&sarah)->do_life();

return n + m; // is 3
}
```

The why

```cpp
#include <new>

struct storage { const int num; };

int main()
{
storage* data = new storage { 31 };
const int old = data->num;

new(data) storage { 12 };

return old + data->num; // is ???
}
```

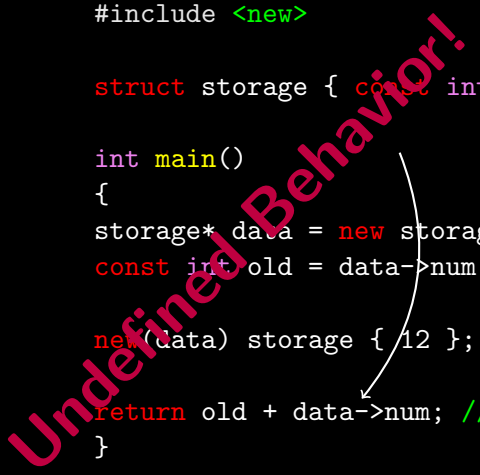
```cpp
#include <new>

struct storage { const int num; };

int main()
{
storage* data = new storage { 31 };
const int old = data->num;

new(data) storage { 12 };

return old + data->num; // is ???
}
```



6.6.3.8 basic.life

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 storage for the new object exactly overlays the storage location which the original object occupied, and
- the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
- the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
- neither the original object nor the new object is a potentially-overlapping subobject.

6.6.3.8 basic.life

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 storage for the new object exactly overlays the storage location which the original object occupied, and
- the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
- the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
- neither the original object nor the new object is a potentially-overlapping subobject.

6.6.3.8 basic.life

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 storage for the new object exactly overlays the storage location which the original object occupied, and
- the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
- the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
- neither the original object nor the new object is a potentially-overlapping subobject.

```cpp
#include <new>

struct storage { const int num; };

int main()
{
storage* data = new storage { 31 };
const int old = data->num;

new(data) storage { 12 };

return old + std::launder(data)->num; // is ???
}
```

```cpp
#include <new>

struct storage { const int num; };

int main()
{
storage* data = new storage { 31 };
const int old = data->num;

new(data) storage { 12 };

return old + std::launder(data)->num; // is 43
}
```

```cpp
#include <new>

struct storage { const int num; };

int main()
{
storage* data = new storage { 31 };
const int old = data->num;

data = new(data) storage { 12 };

return old + data->num; // also 43
}
```

```
struct U { const int i; };

std::vector<U> B;
B.push_back({ 42 });
B.clear();
B.push_back({ 48988 });

const int v = B[0].i;
```

```cpp
struct U { const int i; };

std::vector<U> B;
B.push_back({ 42 });
B.clear();
B.push_back({ 48988 });

const int v = B[0].i; // <- UB
```

```
struct U { const int i; };

std::vector<U> B;
B.push_back({ 42 });
B.clear();
B.push_back({ 48988 });

const int v = B[0].i; // <- UB
```

Idea: Use memory laundering!

```
T& operator[](std::size_t i)
{
return std::launder(_elems)[i];
}
```

```
T& operator[](std::size_t i)
{
return std::launder(_elems)[i];
}
```

...but be careful, there is undefined behavior lurking in there...

```
T& operator[](std::size_t i)
{
return std::launder(_elems)[i];
}
```

...but be careful, there is undefined behavior lurking in there...

...because `_elems` might not be a raw pointer type.

```
T& operator[](std::size_t i)
{
return std::launder(_elems)[i];
}
```

...but be careful, there is undefined behavior lurking in there...

...because `_elems` might not be a raw pointer type.

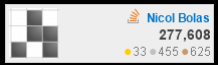
```
std::allocator_traits<A>::pointer
```

must only fullfill the `std::pointer_traits` requirements.

The how

```cpp
/// Pointer optimization barrier [ptr.launder]
template<typename _Tp>
[[nodiscard]] constexpr _Tp* launder(_Tp* __p) noexcept
{ return __builtin_launder(__p); }
```

"Money laundering is used to prevent people from tracing where you got your money from. Memory laundering is used to prevent the compiler from tracing where you got your object from, thus forcing it to avoid any optimizations that may no longer apply."

- 

Thank you for your attention!