Lifetime of the C++ object

Dawid Pilarski

dawid.pilarski@tomtom.com
dawid.pilarski@panicsoftware.com
blog.panicsoftware.com

Agenda

Theory

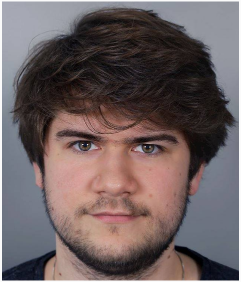
Object model intuition

Pointers and std::launder

Who am I?

Dawid Pilarski

- Senior Software Developer in TomTom
- Member of the ISO/JTC1/SC22/WG21
- Member of the PKN KT (programming languages)
- C++ blog writer



Questions.

Questions...

Question...

Do you think, that understanding objects and their lifetimes is basics?

What we talk about are basics.

6 Basics [basic]

6.7 Memory and objects [basic.memobj]

6.7.2 Object model [intro.object]

1 The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is created by a definition, by a *new-expression*, when implicitly changing the active member of a union, or when a temporary object is created ([conv.rval], [class.temporary]). An object occupies a region of storage in its period of construction ([class.cdtor]), throughout its lifetime, and in its period of destruction ([class.cdtor]). [*Note:* A function is not an object, regardless of whether or not it occupies storage in the way that objects do. — *end note*] The properties of an object are determined when the object is created. An object can have a name ([basic.pre]). An object has a storage duration ([basic.stc]) which influences its lifetime ([basic.life]). An object has a type ([basic.types]). Some objects are polymorphic ([class.virtual]); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the *expressions* ([expr.compound]) used to access them.

2 Objects can contain other objects, called *subobjects*. A subobject can be a *member subobject* ([class.mem]), a *base class subobject* ([class.derived]), or an array element. An object that is not a subobject of any other object is called a *complete object*. If an object is created in storage associated with a member subobject or array element *e* (which may or may not be within its lifetime), the created object is a subobject of *e*'s containing object if:

(2.1) — the lifetime of *e*'s containing object has begun and not ended, and

(2.2) — the storage for the new object exactly overlays the storage location associated with *e*, and

(2.3) — the new object is of the same type as *e* (ignoring cv-qualification).

3 If a complete object is created ([expr.new]) in storage associated with another object *e* of type "array of *N* `unsigned char`" or of type "array of *N* `std::byte`" ([cstddef.syn]), that array *provides storage* for the created object if:

(3.1) — the lifetime of *e* has begun and not ended, and

(3.2) — the storage for the new object fits entirely within *e*, and

Theory

Title decomposition

What's the lifetime of your object?

Title decomposition

What's the lifetime of your object?

- What is a lifetime?

Title decomposition

What's the lifetime of your object?

- What is a lifetime?
- What is an object?

Objects

The object

Objects are entities, that can be:

- created
- destroyed
- refered to
- accessed
- manipulated

The Object

Is created:

- by the definition

```
int a;
```

The Object

Is created:

- by the definition
- by the new expression

```
new int(5);
```

The Object

Is created:

- by the definition
- by the new expression
- when changing active member of a union

```
union U{int x; int y;};

U u;
u.y = 2; // active member y;
```

The Object

Is created:

- by the definition
- by the new expression
- when changing active member of a union
- by creation of the temporary

```
int{};
```

The object

Has:

- optional name

The object

Has:

- optional name
- lifetime

The object

Has:

- optional name
- lifetime
- storage and it's duration

The object

Has:

- optional name
- lifetime
- storage and it's duration
    - static

program duration

The object

Has:

- optional name
- lifetime
- storage and it's duration
    - static
    - thread

thread duration

The object

Has:

- optional name
- lifetime
- storage and it's duration
    - static
    - thread
    - automatic

enclosing scope duration

The object

Has:

- optional name
- lifetime
- storage and it's duration
  - static
  - thread
  - automatic
  - dynamic

controlled by user

The object

Has:

- optional name
- lifetime
- storage and it's duration
  - static
  - thread
  - automatic
  - dynamic
- type

The object

Has:

- optional name
- lifetime
- storage and it's duration
  - static
  - thread
  - automatic
  - dynamic
- type
- value

The reference

Is not an object (although reference has lifetime)

functions are not objects as well

The variable

Is created by a declaration of an object or the reference.

The variable

Is created by a declaration of an object or the reference.

```
int x;
```

Is a variable.

The variable

Is created by a declaration of an object or the reference.

```
int x;
```

Is a variable.

```
int& x = ...
```

Is a variable.

The variable

Is created by a declaration of an object or the reference.

```
int x;
```

Is a variable.

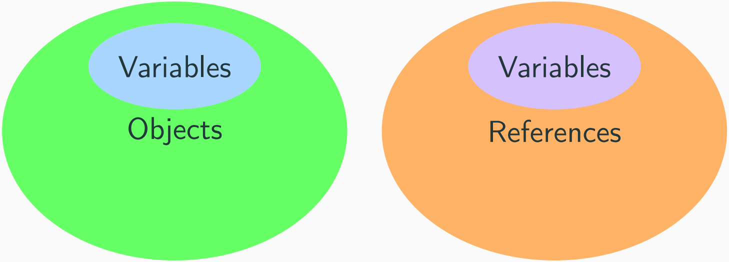
```
int& x = ...
```

Is a variable.

```
struct X{int y;}z;
```

Neither `X` nor `y` are variables.
`z` is a variable.

Summary: variable, reference, object



Lifetime

What is a lifetime?

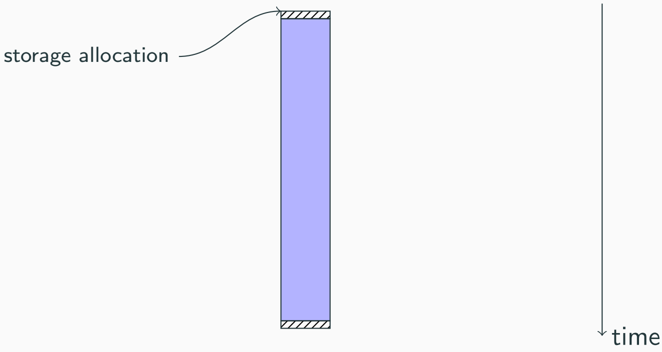
Lifetime is a property of an object.

During the lifetime of an object you can use it without additional restrictions.

What is a lifetime?

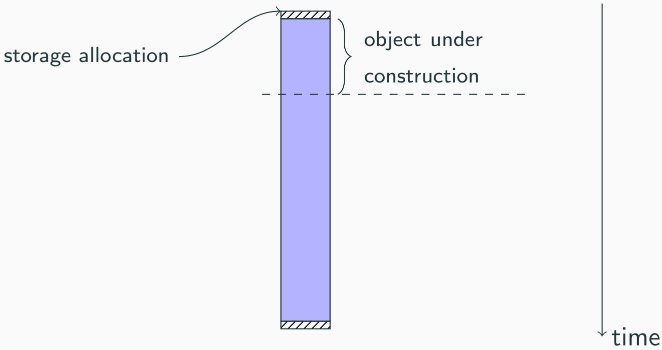
Lifetime is a property of an object.

During the lifetime of an object you can use it without additional restrictions.



What is a lifetime?

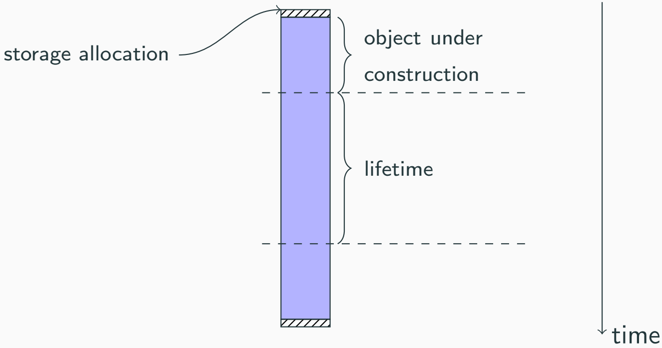
Lifetime is a property of an object.
During the lifetime of an object you can use it without additional restrictions.



What is a lifetime?

Lifetime is a property of an object.

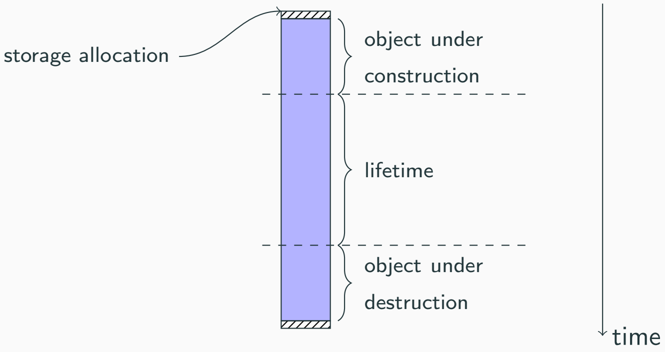
During the lifetime of an object you can use it without additional restrictions.



What is a lifetime?

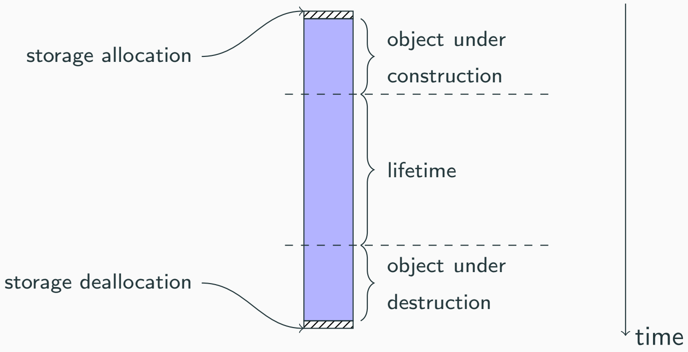
Lifetime is a property of an object.

During the lifetime of an object you can use it without additional restrictions.



What is a lifetime?

Lifetime is a property of an object.
During the lifetime of an object you can use it without additional restrictions.



When does the lifetime start?

The lifetime of an object starts, when:

- storage with the proper alignment and size for type T is obtained

When does the lifetime start?

The lifetime of an object starts, when:

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

*In case of default construction of trivial type, there is no initialization performed

When does the lifetime start?

The lifetime of an object starts, when:

- storage with the proper alignment and size for type T is obtained
- its initialization (if any)* is complete
- if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member

When does the lifetime end?

The lifetime of an object ends:

class types when it’s destructor is called,

When does the lifetime end?

The lifetime of an object ends:

class types when it's destructor is called,

non-class types when we expect it to end its lifetime,

When does the lifetime end?

The lifetime of an object ends:

class types when it's destructor is called,

non-class types when we expect it to end its lifetime,

- when object exits the scope,

When does the lifetime end?

The lifetime of an object ends:

class types when it's destructor is called,

non-class types when we expect it to end its lifetime,

- when object exits the scope,
- delete expression,

When does the lifetime end?

The lifetime of an object ends:

class types when it's destructor is called,

non-class types when we expect it to end its lifetime,

- when object exits the scope,
- delete expression,
- when temporary ends its lifetime etc.

When does the lifetime end?

The lifetime of an object ends:

class types when it's destructor is called,

non-class types when we expect it to end its lifetime,

- when object exits the scope,
- delete expression,
- when temporary ends its lifetime etc.

any type when storage occupied by an object is reused or released.

Limitations on objects - Storage allocated cdtor not yet called

During this phase you must treat every pointer/reference to such object as if it was raw storage.

During construction you cannot:

- pass pointer as `delete` argument,

Limitations on objects - Storage allocated cdtor not yet called

During this phase you must treat every pointer/reference to such object as if it was raw storage.

During construction you cannot:

- pass pointer as `delete` argument,
- access any non-static members (unless type has trivial ctor),

Limitations on objects - Storage allocated cdtor not yet called

During this phase you must treat every pointer/reference to such object as if it was raw storage.

During construction you cannot:

- pass pointer as `delete` argument,
- access any non-static members (unless type has trivial ctor),
- `static_cast` to types other than:

Limitations on objects - Storage allocated cdtor not yet called

During this phase you must treat every pointer/reference to such object as if it was raw storage.

During construction you cannot:

- pass pointer as `delete` argument,
- access any non-static members (unless type has trivial ctor),
- `static_cast` to types other than:
  - `void*`

Limitations on objects - Storage allocated cdtor not yet called

During this phase you must treat every pointer/reference to such object as if it was raw storage.

During construction you cannot:

- pass pointer as `delete` argument,
- access any non-static members (unless type has trivial ctor),
- `static_cast` to types other than:
  - `void*`
  - `char*`

Limitations on objects - Storage allocated cdtor not yet called

During this phase you must treat every pointer/reference to such object as if it was raw storage.

During construction you cannot:

- pass pointer as `delete` argument,
- access any non-static members (unless type has trivial ctor),
- `static_cast` to types other than:
  - `void*`
  - `char*`
  - `unsigned char*`

Limitations on objects - Storage allocated cdtor not yet called

During this phase you must treat every pointer/reference to such object as if it was raw storage.

During construction you cannot:

- pass pointer as `delete` argument,
- access any non-static members (unless type has trivial ctor),
- `static_cast` to types other than:
  - `void*`
  - `char*`
  - `unsigned char*`
  - `std::byte*`

Limitations on objects - Storage allocated cdtor not yet called

During this phase you must treat every pointer/reference to such object as if it was raw storage.

During construction you cannot:

- pass pointer as `delete` argument,
- access any non-static members (unless type has trivial ctor),
- `static_cast` to types other than:
    - `void*`
    - `char*`
    - `unsigned char*`
    - `std::byte*`
- `dynamic_cast` it.

Limitations on objects - object under (con|de)struction

You can access non-static members of a class, but only via this pointer.

Example...

Limitations on objects - object under (con|de)struction

```
struct C {
  int c;
  C() : c(0) { no_opt(); }
};

const C cobj;
void no_opt() {
  int i = cobj.c * 100; //unspecified value
  std::cout << i << std::endl;
}
```

Limitations on objects - object under (con|de)struction

```
struct C {
  int c;
  C() : c(0) { no_opt(); }
};

const C cobj;
void no_opt() {
  int i = cobj.c * 100; //unspecified value
  std::cout << i << std::endl;
}
```

Limitations on objects - object under (con|de)struction

```
struct C {
  int c;
  C() : c(0) { no_opt(); }
};

const C cobj;
void no_opt() {
  int i = cobj.c * 100; //unspecified value
  std::cout << i << std::endl;
}
```

Limitations on objects - object under (con|de)struction

```
struct C {
  int c;
  C() : c(0) { no_opt(); }
};

const C cobj;
void no_opt() {
  int i = cobj.c * 100; //unspecified value
  std::cout << i << std::endl;
}
```

Limitations on objects - object under (con|de)struction

```
struct C {
  int c;
  C() : c(0) { no_opt(); }
};

const C cobj;
void no_opt() {
  int i = cobj.c * 100; //unspecified value
  std::cout << i << std::endl;
}
```

Limitations on objects - object under (con|de)struction

During object creation and destruction, base object, whose constructor is running is considered most derived object.

Example...

Limitations on objects - object under (con|de)struction

```
struct Base{
  Base();
  virtual void foo(){}
};
struct Derived : Base{};

Base::Base(){
    std::cout << typeid(*this).name() << std::endl;
    std::cout << dynamic_cast<Derived*>(this) << std::endl;
}

Derived d;
```

Limitations on objects - object under (con|de)struction

```cpp
struct Base{
  Base();
  virtual void foo(){}
};
struct Derived : Base{};

Base::Base(){
    std::cout << typeid(*this).name() << std::endl;
    std::cout << dynamic_cast<Derived*>(this) << std::endl;
}

Derived d;
```

Limitations on objects - object under (con|de)struction

```cpp
struct Base{
  Base();
  virtual void foo(){}
};
struct Derived : Base{};

Base::Base(){
    std::cout << typeid(*this).name() << std::endl;
    std::cout << dynamic_cast<Derived*>(this) << std::endl;
}

Derived d;
```

Limitations on objects - object under (con|de)struction

```
struct Base{
  Base();
  virtual void foo(){}
};
struct Derived : Base{};

Base::Base(){
    std::cout << typeid(*this).name() << std::endl;
    std::cout << dynamic_cast<Derived*>(this) << std::endl;
}

Derived d;
```

Output:

4Base

Limitations on objects - object under (con|de)struction

```
struct Base{
  Base();
  virtual void foo(){}
};
struct Derived : Base{};

Base::Base(){
    std::cout << typeid(*this).name() << std::endl;
    std::cout << dynamic_cast<Derived*>(this) << std::endl;
}

Derived d;
```

Output:

0

Limitations on objects - object under (con|de)struction

Watch out for vptr - its not thread safe.

Example...

Limitations on objects - object under (con|de)struction

```
struct runnable{
void execute(){
  run();
  lock_guard lck(m_);
  done = true;
  cv_.notify_one();
}
~runnable(){
  unique_lock<mutex> lck(m_);
  cv.wait(lck,
    [this]{return done_;});
}
private:
  virtual void run()=0;
  //...
};
```

Limitations on objects - object under (con|de)struction

```
struct runnable{
void execute(){
  run();
  lock_guard lck(m_);
  done = true;
  cv_.notify_one();
}
~runnable(){
  unique_lock<mutex> lck(m_);
  cv.wait(lck,
    [this]{return done_;});
}
private:
  virtual void run()=0;
  //...
};
```

Limitations on objects - object under (con|de)struction

```
struct runnable{
void execute(){
  run();
  lock_guard lck(m_);
  done = true;
  cv_.notify_one();
}
~runnable(){
  unique_lock<mutex> lck(m_);
  cv.wait(lck,
    [this]{return done_;});
}
private:
  virtual void run()=0;
  //...
};
```

Limitations on objects - object under (con|de)struction

```
struct runnable{
void execute(){
  run();
  lock_guard lck(m_);
  done = true;
  cv_.notify_one();
}
~runnable(){
  unique_lock<mutex> lck(m_);
  cv.wait(lck,
    [this]{return done_;});
}
private:
  virtual void run()=0;
  //...
};
```

Limitations on objects - object under (con|de)struction

```
struct runnable{
void execute(){
  run();
  lock_guard lck(m_);
  done = true;
  cv_.notify_one();
}
~runnable(){
  unique_lock<mutex> lck(m_);
  cv.wait(lck,
    [this]{return done_;});
}
private:
  virtual void run()=0;
  //...
};
```

```
runnable* d = new Derived;
thread th1([d]{d->execute()});
thread th2([d]{delete d;});
```

Limitations on objects - object under (con|de)struction

```
struct runnable{
void execute(){
  run();
  lock_guard lck(m_);
  done = true;
  cv_.notify_one();
}
~runnable(){
  unique_lock<mutex> lck(m_);
  cv.wait(lck,
    [this]{return done_;});
}
private:
  virtual void run()=0;
  //...
};
```

vptr read

```
runnable* d = new Derived;
thread th1([d]{d->execute()});
thread th2([d]{delete d;});
```

Limitations on objects - object under (con|de)struction

```
struct runnable{
void execute(){
  run();
  lock_guard lck(m_);
  done = true;
  cv_.notify_one();
}
~runnable(){
  unique_lock<mutex> lck(m_);
  cv.wait(lck,
    [this]{return done_;});
}
private:
  virtual void run()=0;
  //...
};
```

vptr read

```
runnable* d = new Derived;
thread th1([d]{d->execute()});
thread th2([d]{delete d;});
```

vptr write

Limitations on objects - object under (con|de)struction

```
struct runnable{
void execute(){
  run();
  lock_guard lck(m_);
  done = true;
  cv_.notify_one();
}
~runnable(){
  unique_lock<mutex> lck(m_);
  cv.wait(lck,
    [this]{return done_;});
}
private:
  virtual void run()=0;
  //...
};
```

vptr read

```
runnable* d = new Derived;
thread th1([d]{d->execute()});
thread th2([d]{delete d;});
```

vptr write

data race!

Limitations on objects - object under (con|de)struction

When casting this pointer to a base subobject, if any of the subobjects involved in the casting path didn't have their constructor started, the behavior is undefined.

Example...

Limitations on objects - object under (con|de)struction

```
struct A{
  A(A*){}
};
struct B : A{
  B() : A(this){}
};
```

pointer to object, which ctor didn't start

Limitations on objects - object under (con|de)struction

```cpp
struct C;
struct B{
  B(C*);
};
struct C{
  C(B*);
};
struct D{
  D() :
    c(&b), // UB, b's vtor didn't start yet.
    b(&c){}
  C c; // constructed first
  B b;
};
```

Limitations on objects - object under (con|de)struction

```
struct C;
struct B{
  B(C*);
};
struct C{
  C(B*);
};
struct D{
  D() :
    c(&b), // UB, b's vtor didn't start yet.
    b(&c){}
  C c; // constructed first
  B b;
};
```

Limitations on objects - object under (con|de)struction

```
struct C;
struct B{
  B(C*);
};
struct C{
  C(B*);
};
struct D{
  D() :
    c(&b), // UB, b's vtor didn't start yet.
    b(&c){}
  C c; // constructed first
  B b;
};
```

Limitations on objects - object under (con|de)struction

```
struct C;
struct B{
  B(C*);
};
struct C{
  C(B*);
};
struct D{
  D() :
    c(&b), // UB, b's vtor didn't start yet.
    b(&c){}
  C c; // constructed first
  B b;
};
```

Limitations on objects - object under (con|de)struction

```
struct C;
struct B{
  B(C*);
};
struct C{
  C(B*);
};
struct D{
  D() :
    c(&b), // UB, b's vtor didn't start yet.
    b(&c){}
  C c; // constructed first
  B b;
};
```

Lifetime of an object

No previously mentioned limitations exists when object is in its lifetime.

Object model intuition

The usual mindset towards object

- We have got a memory

...
24
42
...

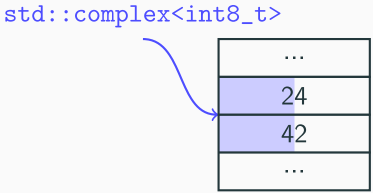
The usual mindset towards object

- We have got a memory
- Objects are the way to represent value in that memory

...
24
42
...

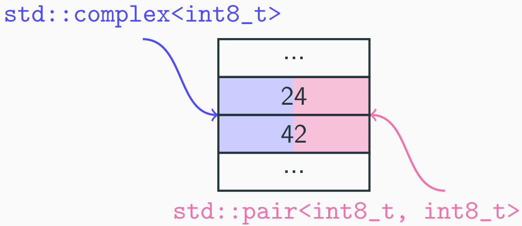
The usual mindset towards object

- We have got a memory
- Objects are the way to represent value in that memory



The usual mindset towards object

- We have got a memory
- Objects are the way to represent value in that memory



Examples of invalid C++ - union type punning

```
struct rgba{
  uint8_t red;
  uint8_t green;
  uint8_t blue;
  uint8_t alpha;
};

union color{
  rgba color;
  uint32_t as_int;
};

color c = {255, 120, 0, 50};
display(c.as_int);
```

Examples of invalid C++ - union type punning

```
struct rgba{
  uint8_t red;
  uint8_t green;
  uint8_t blue;
  uint8_t alpha;
};

union color{
  rgba color;
  uint32_t as_int;
};

color c = {255, 120, 0, 50};
display(c.as_int);
```

Examples of invalid C++ - union type punning

```
struct rgba{
  uint8_t red;
  uint8_t green;
  uint8_t blue;
  uint8_t alpha;
};

union color{
  rgba color;
  uint32_t as_int;
};

color c = {255, 120, 0, 50};
display(c.as_int);
```

Examples of invalid C++ - reinterpret_cast

```
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) unsigned char buff[sizeof(T)];
  read_stream(s, buff);

  auto* element = reinterpret_cast<T*>(buff);
  return *element;
}
```

Examples of invalid C++ - reinterpret_cast

```
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) unsigned char buff[sizeof(T)];
  read_stream(s, buff);

  auto* element = reinterpret_cast<T*>(buff);
  return *element;
}
```

Examples of invalid C++ - reinterpret_cast

```
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) unsigned char buff[sizeof(T)];
  read_stream(s, buff);

  auto* element = reinterpret_cast<T*>(buff);
  return *element;
}
```

Examples of invalid C++ - reinterpret_cast

```
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) unsigned char buff[sizeof(T)];
  read_stream(s, buff);

  auto* element = reinterpret_cast<T*>(buff);
  return *element;
}
```

Examples of invalid C++ - reinterpret_cast

```
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) unsigned char buff[sizeof(T)];
  read_stream(s, buff);

  auto* element = reinterpret_cast<T*>(buff);
  return *element;
}
```

reinterpret_cast attempt 2

```cpp
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) char buff[sizeof(T)];
  read_stream(s, buff);

  T* element = new(buff) T;
  return *element;
}
```

But... why?

Why all the attempts are wrong?

Compiler doesn't think in terms of objects and memory

Why all the attempts are wrong?

Compiler doesn't think in terms of objects and memory

Compiler thinks in terms of objects and their types.

Test with two different class types

```cpp
struct S{
  int a;
};
```

```cpp
struct T {
  int a;
};
```

```cpp
int test(S& val1, T& val2){
  val1.a = 10;
  val2.a = 2;

  return val1.a+val2.a;
}
```

Test with two different class types

```
struct S{
  int a;
};

struct T {
  int a;
};

int test(S& val1, T& val2){
  val1.a = 10;
  val2.a = 2;

  return val1.a+val2.a;
}
```

Test with two different class types

```
struct S{
  int a;
};

struct T {
  int a;
};

int test(S& val1, T& val2){
  val1.a = 10;
  val2.a = 2;

  return val1.a+val2.a;
}
```

Test with two different class types

```
struct S{
  int a;
};

struct T {
  int a;
};

int test(S& val1, T& val2){
  val1.a = 10;
  val2.a = 2;

  return val1.a+val2.a;
}
```

Q: What is the return value?

Test with two same structures

```cpp
struct S{
  int a;
};

int test(S& val1, S& val2){
  val1.a = 10;
  val2.a = 2;

  return val1.a+val2.a;
}
```

Test with two same structures

```
struct S{
  int a;
};

int test(S& val1, S& val2){
  val1.a = 10;
  val2.a = 2;

  return val1.a+val2.a;
}
```

Test with two same structures

```cpp
struct S{
  int a;
};

int test(S& val1, S& val2){
  val1.a = 10;
  val2.a = 2;

  return val1.a+val2.a;
}
```

Test with two same structures

```
struct S{
  int a;
};

int test(S& val1, S& val2){
  val1.a = 10;
  val2.a = 2;

  return val1.a+val2.a;
}
```

Q: What is the return value now?

Assumptions, that compiler does

Code:

```
int test(S& val1, T& val2){
  val1.a = 10;
  val2.a = 2;

  return val1.a+val2.a;
}
```

Assembly:

```
test(S&, T&):
  mov r2, #10
  mov r3, #2
  str r2, [r0]
  str r3, [r1]
  mov r0, #12
  bx lr
```

Assumptions, that compiler does

Code:

```
int test(S& val1, S& val2){
  val1.a = 10;
  val2.a = 2;

  return val1.a+val2.a;
}
```

Assembly:

```
test(S&, S&):
  mov r3, #2
  mov r2, #10
  str r2, [r0]
  str r3, [r1]
  ldr r0, [r0]
  add r0, r0, r3
  bx lr
```

Conclusion

We cannot allow 2 complete objects of different types (not subobjects) live:

- in the same space.
- at the same time.

Explanations: type punning in union

```
struct rgba{
  uint8_t red;
  uint8_t green;
  uint8_t blue;
  uint8_t alpha;
};

union color{
  rgba color;
  uint32_t as_int;
};

color c = {255, 120, 0, 50};
display(c.as_int);
```

errors:

- Accessing inactive member of union,

Explanations: type punning in union

```
struct rgba{
  uint8_t red;
  uint8_t green;
  uint8_t blue;
  uint8_t alpha;
};

union color{
  rgba color;
  uint32_t as_int;
};

color c = {255, 120, 0, 50};
display(c.as_int);
```

errors:

- Accessing inactive member of union,
- Reading not existing object.

How to do it right?

```
struct color{
  uint8_t red(){
    return as_int>>24;
  }

  void red(uint8_t value){
    uint32_t r32 = static_cast<uint32_t>(red()) << (24);
    as_int ^= r32;
    as_int |= (static_cast<uint32_t>(value) << (24));
  }

  uint32_t as_int;
};
```

How to do it right?

```
struct color{
  uint8_t red(){
    return as_int>>24;
  }

  void red(uint8_t value){
    uint32_t r32 = static_cast<uint32_t>(red()) << (24);
    as_int ^= r32;
    as_int |= (static_cast<uint32_t>(value) << (24));
  }

  uint32_t as_int;
};
```

How to do it right?

```
struct color{
  uint8_t red(){
    return as_int>>24;
  }

  void red(uint8_t value){
    uint32_t r32 = static_cast<uint32_t>(red()) << (24);
    as_int ^= r32;
    as_int |= (static_cast<uint32_t>(value) << (24));
  }

  uint32_t as_int;
};
```

Explanations: reading from a stream

```
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) unsigned char buff[sizeof(T)];
  read_stream(s, buff);

  auto* element = reinterpret_cast<T*>(buff);
  return *element;
}
```

errors:

- reading object of type T, that was not created

How to do it right?

```
struct T{
  // ...
};
```

trivially copyable

```
T process_element(Stream& s){
  unsigned char buff[sizeof(T)];
  read_stream(s, buff);

  T element;
  std::memcpy(&element, buff, sizeof(T));
  return element;
}
```

How to do it right?

```
struct T{
  // ...
};
```

trivially copyable

```
T process_element(Stream& s){
  unsigned char buff[sizeof(T)];
  read_stream(s, buff);

  T element;
  std::memcpy(&element, buff, sizeof(T));
  return element;
}
```

Or with C++20

```
struct T{
  // ...
};

T process_element(Stream& s){
  unsigned char buff[sizeof(T)];
  read_stream(s, buff);

  return std::bit_cast<T>(buff);
}
```

Explanations: reading from a stream - placement new

```
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) char buff[sizeof(T)];
  read_stream(s, buff);

  T* element = new(buff) T;
  return *element;
}
```

- placement new reuses the storage (ends lifetime of buff),

Explanations: reading from a stream - placement new

```
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) char buff[sizeof(T)];
  read_stream(s, buff);

  T* element = new(buff) T;
  return *element;
}
```

- placement new reuses the storage (ends lifetime of buff),
- value is a property of an object (doesn't exist outside of it's lifetime)

Explanations: reading from a stream - placement new

```
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) char buff[sizeof(T)];
  read_stream(s, buff);

  T* element = new(buff) T;
  return *element;
}
```

- placement new reuses the storage (ends lifetime of buff),
- value is a property of an object (doesn't exist outside of it's lifetime)
- T is created uninitialized

Explanations: reading from a stream - placement new

```
struct T{
  // ...
};

T process_element(Stream& s){
  alignas(T) char buff[sizeof(T)];
  read_stream(s, buff);

  T* element = new(buff) T;
  return *element;
}
```

- placement new reuses the storage (ends lifetime of buff),
- value is a property of an object (doesn't exist outside of it's lifetime)
- T is created uninitialized
- reading T is reading indeterminate value, which is UB

Pointers and std::launder

Tricky placement new

```
struct T{
  //...
};

alignas(T) unsigned char
buff[sizeof(T)];
new(buff) T;
T* ptr =
   reinterpret_cast<T*>(buff);
//use ptr
```

- sometimes you know, there is an object of given type under given address.

Tricky placement new

```
struct T{
  //...
};

alignas(T) unsigned char
buff[sizeof(T)];
new(buff) T;
T* ptr =
   reinterpret_cast<T*>(buff);
//use ptr
```

- sometimes you know, there is an object of given type under given address.
- but you cannot just `reinterpret_cast` the pointer to get to the object.

What’s the solution to the problem?

Until C++14 No solution. Go on with UB.

What's the solution to the problem?

Until C++14 No solution. Go on with UB.

Since C++17 `std::launder`.

What is std::launder?

Reading cppreference:

```
template <class T>
constexpr T* launder(T* p) noexcept;
```

What is std::launder?

Reading cppreference:

```
template <class T>
constexpr T* launder(T* p) noexcept;
```

Obtains a pointer to the object located at the address represented by p.

What is std::launder?

Reading cppreference:

```
template <class T>
constexpr T* launder(T* p) noexcept;
```

Obtains a pointer to the object located at the address represented by p.

Usage of std::launder

```cpp
struct T{
  //...
};

alignas(T) unsigned char
buff[sizeof(T)];
new(buff) T;
T* ptr = std::launder((T*)buff);
//use ptr
```

Usage of std::launder

```cpp
struct T{
  //...
};

alignas(T) unsigned char
buff[sizeof(T)];
new(buff) T;
T* ptr = std::launder((T*)buff);
//use ptr
```

Usage of std::launder

```
struct T{
  //...
};

alignas(T) unsigned char
buff[sizeof(T)];
new(buff) T;
T* ptr = std::launder((T*)buff);
//use ptr
```

Usage of std::launder

```
struct T{
  //...
};

alignas(T) unsigned char
buff[sizeof(T)];
new(buff) T;
T* ptr = std::launder((T*)buff);
//use ptr
```

valid

Other use-cases of the std::launder

```
struct T{
  //...
};
```

not assignable

```
T a;
T b;

// wild desire to assign ends with:
T* newa = new(&a) T(b);

use(*newa);
use(a);
```

Other use-cases of the std::launder

```
struct T{          not assignable
  //...
};

T a;
T b;

// wild desire to assign ends with:
T* newa = new(&a) T(b);

use(*newa);
use(a);
```

Other use-cases of the std::launder

```
struct T{
  //...
};

T a;
T b;

// wild desire to assign ends with:
T* newa = new(&a) T(b);

use(*newa);
use(a);
```

not assignable

Other use-cases of the std::launder

```
struct T{
  //...
};

T a;
T b;

// wild desire to assign ends with:
T* newa = new(&a) T(b);

use(*newa);
use(a);
```

not assignable

Are those correct?

Rules for placement new for such cases

You can do placement new on old object with same type and keep using old object.

Rules for placement new for such cases

You can do placement new on old object with same type and keep using old object. With some exceptions (until C++20). For example:

- We cannot use original object when it has:

Rules for placement new for such cases

You can do placement new on old object with same type and keep using old object.
With some exceptions (until C++20). For example:

- We cannot use original object when it has:
  - references

Rules for placement new for such cases

You can do placement new on old object with same type and keep using old object. With some exceptions (until C++20). For example:

- We cannot use original object when it has:
  - references
  - const members

Example: const member or reference

```
struct T{
  const int a;
  int& b;
};

T a{/**/};
T b{/**/};

T* newa = new(&a) T(b);

use(*newa); // correct
use(a); // correct since C++20
use(*std::launder(&a)); // correct, possible since C++17
```

Example: const member or reference

```
struct T{
  const int a;
  int& b;
};

T a{/**/};
T b{/**/};

T* newa = new(&a) T(b);

use(*newa); // correct
use(a); // correct since C++20
use(*std::launder(&a)); // correct, possible since C++17
```

Example: const member or reference

```
struct T{
  const int a;
  int& b;
};

T a{/**/};
T b{/**/};

T* newa = new(&a) T(b);

use(*newa); // correct
use(a); // correct since C++20
use(*std::launder(&a)); // correct, possible since C++17
```

Example: const member or reference

```
struct T{
  const int a;
  int& b;
};

T a{/**/};
T b{/**/};

T* newa = new(&a) T(b);

use(*newa); // correct
use(a); // correct since C++20
use(*std::launder(&a)); // correct, possible since C++17
```

Example: standard layout types

```
struct T{
  int a;
  int b;
};

T a{/**/};
T b{/**/};

T* newa = new(&a) T(b);

use(*newa); // correct
use(a); // correct
use(*std::launder(&a)); // correct, possible since C++17
```

Example: standard layout types

```
struct T{
  int a;
  int b;
};

T a{/**/};
T b{/**/};

T* newa = new(&a) T(b);

use(*newa); // correct
use(a); // correct
use(*std::launder(&a)); // correct, possible since C++17
```

Implicit object creation

Implicit object creation

Until C++20 there is one case of implicit object creation:

for defaulted, trivial assignment operators of union members.

Implicit object creation

Until C++20 there is one case of implicit object creation:

for defaulted, trivial assignment operators of union members.

What does that mean?

Implicit object creation for unions

```
struct T{
  int a;
  int b;
};

union U{
  T t;
  char b;
};

U u;
u.t = T{42, 24};
u.b = 'a';
```

Implicit object creation for unions

```
struct T{
  int a;
  int b;
};

union U{
  T t;
  char b;
};

U u;
u.t = T{42, 24};
u.b = 'a';
```

Implicit object creation for unions

```
struct T{
  int a;
  int b;
};

union U{
  T t;
  char b;
};

U u;
u.t = T{42, 24};
u.b = 'a';
```

correct

Implicit object creation for unions

```
struct T{
  int a;
  int b;
  T& operator=(const T&){/**/}
};

union U{
  T t;
  char b;
};

U u;
u.t = T{42, 24}; //UB
u.b = 'a';
```

Implicit object creation for C++20

Implicit object creation in the C++20 will be extended for:

- malloc-like functions

```
struct T{/**/};
T* ptr = (T*)
        malloc(sizeof(struct T));
```

Implicit object creation for C++20

Implicit object creation in the C++20 will be extended for:

- malloc-like functions
- operator new

```
struct T{/**/};
T* ptr = (T*)
        operator new(sizeof(T));
```

Implicit object creation for C++20

Implicit object creation in the C++20 will be extended for:

- malloc-like functions
- operator new
- std::allocator<T>::allocate

```
struct T{/**/};
std::allocator<T> allocator;
T* ptr = allocator.allocate(1);
```

Implicit object creation for C++20

Implicit object creation in the C++20 will be extended for:

- malloc-like functions
- operator new
- std::allocator<T>::allocate
- memcpy, memmove

```
alignas(T) unsigned char
buff2[sizeof(T)];
std::memcpy(buff2, buff1,
            sizeof(buff2));
T* ptr = (T*)buff;
```

Implicit object creation for C++20

Implicit object creation in the C++20 will be extended for:

- malloc-like functions
- operator new
- std::allocator<T>::allocate
- memcpy, memmove
- creation of arrays of:
  - char
  - unsigned char
  - std::byte

```
alignas(float) unsigned char
buff[sizeof(float)];
float* ptr = (float*)buff;
```

Summary

Summary

- Do think about objects and its type not memory,

Summary

- Do think about objects and its type not memory,
- Don't do the type-punning in C++,

Summary

- Do think about objects and its type not memory,
- Don't do the type-punning in C++,
- Be careful when using objects outside of their lifetimes.

Thank you!

I would like to say thank you to:

- All the people on CppLang [standardese] in helping me to understand standard wording,
- Richard Smith on making the implicit object creation proposal.

Further readings

blog.panicsoftware.com

Further readings

blog.panicsoftware.com

github.com/dawidpilarski/LifetimePresentation

Further readings

blog.panicsoftware.com

github.com/dawidpilarski/LifetimePresentation

Slack: nav-cpp-cop

Thank you!

Questions?

blog.panicsoftware.com
github.com/dawidpilarski/LifetimePresentation
Slack: nav-cpp-cop