Advanced Programming Concepts

week 1 - memory safety

Dawid Zalewski, Robert de Groote

November 12, 2025

What and why we will do

Why memory safety?

What is memory safety?

Potential vulnerabilities that arise from:

- Reading of uninitialized objects
- Dangling pointers
- Buffer overflows
- Use after free
- Memory leaks
- Type safety violations (e.g., incorrect casting)

Solutions we already know

Introducing messages (be defensive)

```
typedef enum : uint64_t {  text = 1, binary } message_type;

const size_t max_data_size = 24;

typedef struct message {
  char data[max_data_size];
  message_type type;
} message;
```

Introducing messages (be defensive)

```
typedef enum : uint64_t {  text = 1, binary } message_type;

const size_t max_data_size = 24;

typedef struct message {
  char data[max_data_size];
  message_type type;
} message;

message create_message(size_t n, char const str[static n])
{
  assert(str != nullptr);
  auto msg = (message){ .data = { }, .type = text };
  auto size = std::min(n, max_data_size - 1);
  std::memcpy(msg.data, str, size);
  return msg;
}
```

Don’t use C

```
/// @brief Creates a message object with given data.
message create_message(size_t n, char const data[static n]);

// OK
auto msg = create_message(15, "Hello, safety!");

// BAD
message msg;

// VERY BAD
auto msg = (message){ .type = 666 };
memcpy(msg.data, "Will it fit into the buffer?", 29);
```

Encapsulate (and use a safer language)!

```
struct message {

  const static size_t max_data_size = 24;
  enum struct message_type : uint64_t { text = 1, binary };

  message(std::string_view str)
  : data_{  }
  , type_{ message_type::text }
  {
    auto size = std::min(str.size(), max_data_size);
    std::memcpy(data_.data(), str.data(), size - 1);
  }

private:
  std::array<std::byte, max_data_size> data_;
  message_type type_;
};
```

Encapsulate!

```
// OK
auto msg = message{ "Hello, safety!" };

// WON'T COMPILE
message msg;

// WON'T COMPILE
auto msg = message{ "Hello, safety!" };
memcpy(msg.data_, "Will it fit into the buffer?", 29);
```

Don’t use raw pointers (in public API’s)

```
struct message_broker {

  void enqueue(message const * msg){
    assert(msg != nullptr);
    queue_.push_back(msg);
  }

private:
  std::vector<message const *> queue_;
};
```

Don’t use raw pointers (in public API’s)

```
broker brk{};
auto msg =
    new message{ "Hello, safety!" };

brk.enqueue(msg);
```

```
struct message_broker {

  void enqueue(message const * msg);

private:
  std::vector<message const *> queue_;
};
```

Don’t use raw pointers (in public API’s)

```
broker brk{};
auto msg =
    new message{ "Hello, safety!" };

brk.enqueue(msg);

/* ~~~ */

delete msg;
```

```
struct message_broker {

  void enqueue(message const * msg);

private:
  std::vector<message const *> queue_;
};
```

Creator is responsible for deletion (no ownership transfer)

Don’t use raw pointers (in public API’s)

```
broker brk{};
auto msg =
    new message{ "Hello, safety!" };

brk.enqueue(msg);

/* ~~~ */
```

```
struct message_broker {

  void enqueue(message const * msg);

  ~message_broker() noexcept {
    for(auto m : queue_)
      delete m;
  }

private:
  std::vector<message const *> queue_;
};
```

message_broker takes ownership and is responsible for deletion

Raw pointers guidelines

- Refrain from using raw pointers in public APIs.
- Never use raw *owning* pointers in public APIs.
- Always document the role of raw pointers at interface boundaries.
- Use values and references whenever possible.

😱
```
message * msg = new message{ "Hello APC!" };
```

😁
```
message msg{ "Hello APC!" };
```

Use value semantics & references

```
struct message_broker {

  void enqueue(message const& msg){
    // assert(msg != nullptr); check not needed
    queue_.push_back(msg);
  }

private:
  std::vector<message> queue_;
};
```

Use value semantics & references

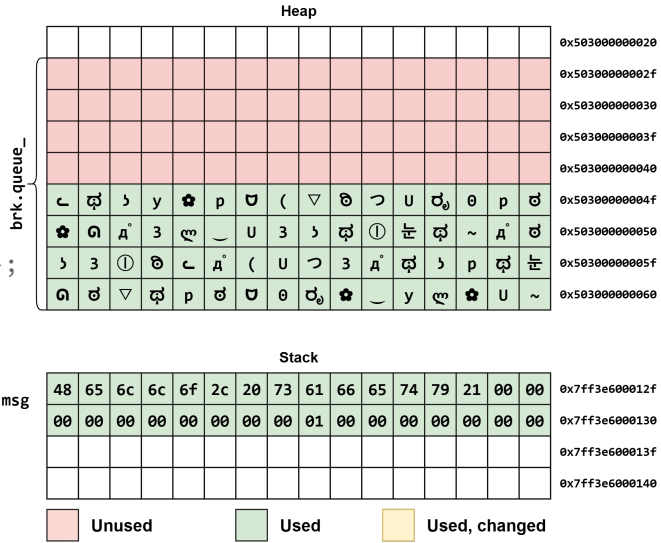
```
broker brk{};
auto msg =
     message{ "Hello, safety!" };

brk.enqueue(msg);

// No need to delete msg
// RAII takes care of it
```

```
struct message_broker {

  void enqueue(message const& msg);

  ~message_broker() noexcept {
    // No code needed here
    // std::vector will clean up itself
  }

private:
  std::vector<message> queue_;
};
```

Working with values and references eliminates ownership issues

Be aware of the costs of values

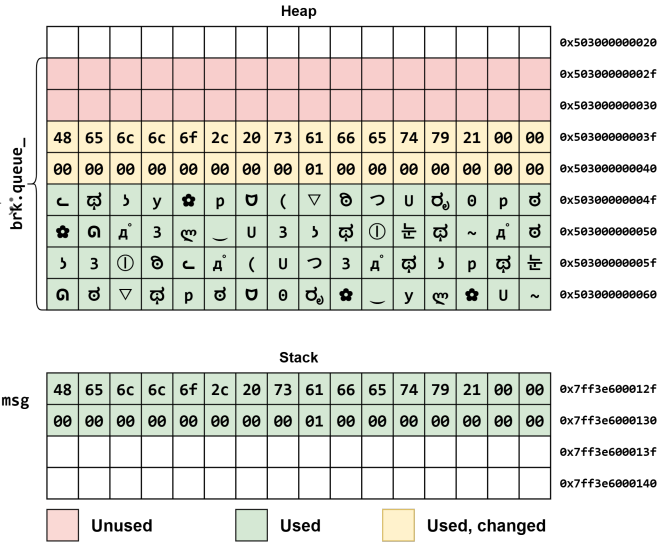
```
message_broker brk{};

message msg{ "Hello, safety!" };
```



Be aware of the costs of values

```
message_broker brk{};

message msg{ "Hello, safety!" };

brk.enqueue( msg );
```

The whole msg object
is copied into the brk's vector.



Move semantics to the rescue

Fixing the issues

What we have:

- `message` objects are small (24 bytes data + 8 bytes type) 😁
- copying a `message` == full binary copy 😕
- `message` cannot store longer data 😩

What we'd want:

- `message` objects that are cheap to put into the `message_broker`
- `message` objects that can store longer (arbitrary sized) data

Fixing the issues: dynamic memory

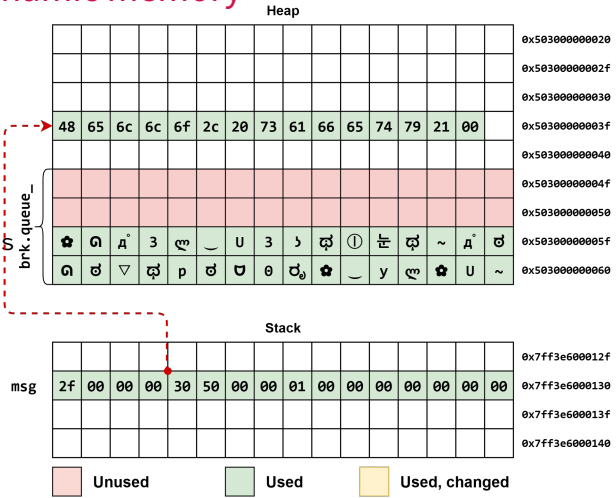
```
struct message {

  message(std::string_view str)
    : data_{ new std::byte[str.size() ] {} }
    , size_{ str.size() }
  {
    std::memcpy(data_, str.data(), size_ );
  }

  ~message() {  delete[] data_; }

private:
  std::byte* data_;
  std::size_t size_;
};
```

Fixing the issues: dynamic memory

```
struct message {

  message(message const& other)
    : data_{ new std::byte[other.size_ ] }
    , size_{ other.size_ }
  {
    std::memcpy(data_, other.data_, other.size_);
  }

  message& operator=(message other) {
    std::swap(data_, other.data_);
    std::swap(size_, other.size_);
    return *this;
  }
 private:
  std::byte* data_;
  std::size_t size_;
};
```

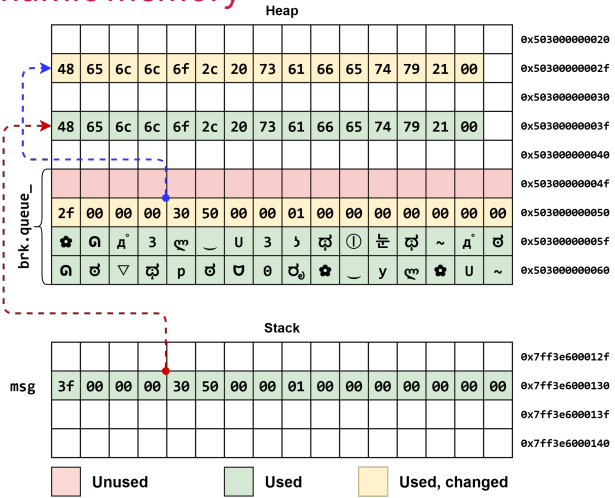
Fixing the issues (somewhat): dynamic memory

```
message_broker brk{};

message msg{ "Hello, safety!" };
```



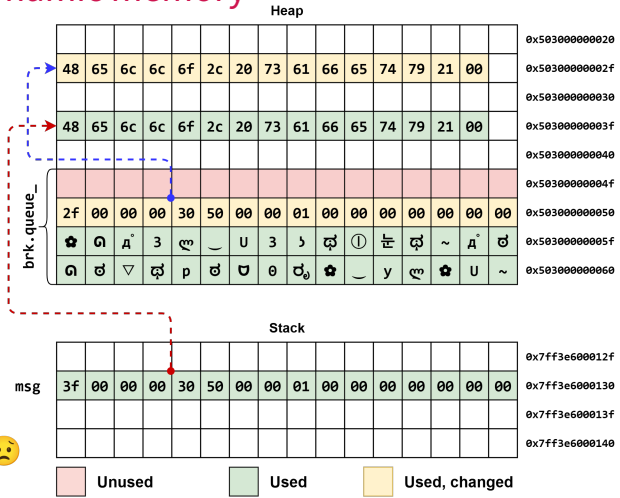
Fixing the issues (somewhat): dynamic memory

```
message_broker brk{};

message msg{ "Hello, safety!" };


brk.enqueue( msg );
```



Fixing the issues (somewhat): dynamic memory

```
message_broker brk{};

message msg{ "Hello, safety!" };


brk.enqueue( msg );
```

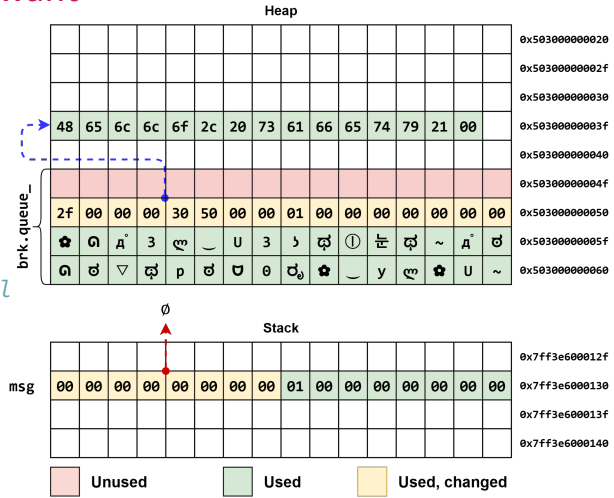


- Adaptable sizing of message::data 😁
- Still copying the whole message object 😟
- Copying incurs heap allocation 😱

Fixing the issues: what we really want

```
message_broker brk{};

message msg{ "Hello, safety!" };

// msg not needed after this call
brk.enqueue( msg );
```

- Dynamic sizing of message::data 😁
- Heap data is reused (hijacked) 🤨



Fixing the issues: tagging for hijacking

```
struct hijack_tag{};

struct message {

  message(message& other, hijack_tag)
    : data_{ other.data_ }
    , size_{ other.size_ }
  {
    other.data_ = nullptr;
    other.size_ = 0;
  }
};

message org{ "Hello, safety!" };

message copied{ org }; // "normal"copy constructor

message hijacked{ org, hijack_tag{} }; // hijack constructor
```

This technique is called ”tag dispatching”

Fixing the issues: tagging for hijacking

```
struct message_broker {

  void enqueue(message const& msg){
    queue_.push_back(msg);
  }

  void enqueue(message& msg, hijack_tag){
    queue_.push_back( message{ msg, hijack_tag{} } );
  }

private:
  std::vector<message> queue_;
};
```

This works, but how? 🤔

Fixing the issues: rvalue references & move semantics

- Tagging is ugly
- Tagging doesn't work for assignment:

```
message first{ "Hello, safety!" };
message second{ "Safety, hello!" };
second = first; // no way to tag here
```

- How does `std::vector` know that it can *hijack* a message put into it?

Fixing the issues: rvalue references & move semantics

- A: `std::vector::push_back` has two overloads:

```
void push_back(const T& value); // copy version
void push_back(T&& value);      // hijack version
```

- && stands for *rvalue reference*
- *rvalue reference* == object can be hijacked (moved from)

Fixing the issues: rvalue references & move semantics

```
struct message {
  message(message const& other); // copy constructor

  message(message&& other)       // move constructor
    : data_{ other.data_ }
    , size_{ other.size_ }
  {
    other.data_ = nullptr;
    other.size_ = 0;
  }
};

message org{ "Hello, safety!" };

message copied{ org }; // "normal" copy constructor

message hijacked{ org }; // also the copy constructor
```

Fixing the issues: rvalue references & move semantics

```
struct message {
  message(message const& other); // copy constructor

  message(message&& other)       // move constructor
    : data_{ other.data_ }
    , size_{ other.size_ }
  {
    other.data_ = nullptr;
  }
};

message org{ "Hello, safety!" };

message copied{ org }; // "normal" copy constructor

message hijacked{ std::move(org) }; // move constructor
```

std::move creates an rvalue reference.

Fixing the issues: rvalue references & move semantics

```
struct message {
  message& operator=(message const& other); // copy assignment

  message& operator=(message&& other)       // move assignment
  {
    delete data_;
    data_ = other.data_;
    other.data_ = nullptr;
    size_ = other.size_;
    other.size_ = 0;
    return *this;
  }
};

message first{ "Hello, safety!" };

message second{ "Safety, hello!" };

second = std::move(first); // move assignment
```

std::move creates an rvalue reference.

Fixing the issues: rvalue references & move semantics

```
struct message {

  message& operator=(message const& other); // copy assignment

  message& operator=(message&& other)       // move assignment
  {
    delete std::exchange(data_, nullptr);
    size_ = std::exchange(other.size_, 0);
    return *this;
  }
};

message first{ "Hello, safety!" };

message second{ "Safety, hello!" };

second = std::move(first); // move assignment
```

std::move creates an rvalue reference.

Rvalue references

- Rvalue reference is created with `std::move`

```
template <typename T>
T&& move(T& obj) {
  return static_cast<T&&>(obj);
}
```

- `std::move` moves nothing. It just casts to T&&
- Rvalue reference indicates that the object can be (moved from)
- Being an *rvalue reference* is a **transient property** :

```
message msg{ "Hello, safety!" };

message&& rref = std::move(msg); // rref is an rvalue reference
message hijacked{ rref }; // rref is not an rvalue reference anymore
message hijacked{ std::move(rref) }; // and now it is again!
```

Rvalue references and push_back

To utilize move semantics in `message_broker::enqueue`, add an overload

```
broker brk{};
auto msg =
     message{ "Hello, safety!" };

brk.enqueue( std::move(msg) );
```

```
struct message_broker {

  void enqueue(message&& msg) {
    queue_.push_back( std::move(msg) );
  }

private:
  std::vector<message> queue_;
};
```

Remembering that rvalue-referenceness is not a permanent property!

Unique ownership and pointers

Pointers, pointers, pointers…

This is a *code smell*:

```
broker brk{};

auto msg =
    new message{ "Hello, safety!" };

// ownership transfer:
brk.enqueue(msg);
```

```
struct message_broker {

  void enqueue(message const * msg);

  ~message_broker() noexcept;

private:
  std::vector<message const *> queue_;
};
```

But sometimes there is no choice...

Unique pointer: motivation

```
struct message_broker {

  void enqueue(std::unique_ptr<message const> msg);

  // destructor is not needed
private:
  std::vector< std::unique_ptr<message const> > queue_;
};

/* ~~~ */
broker brk{};

auto msg =
  std::unique_ptr<message const>{ new message{ "Hello, safety!" } };

// ownership transfer:
brk.enqueue( std::move(msg) );
```

But we can do this instead!

Unique pointers: what are they?

- `std::unique_ptr<T>` is a *smart* pointer that points to an object of type T.
- `std::unique_ptr<T>` owns the object it points to.
- `std::unique_ptr<T>` is the only owner of the object it points to.
- `std::unique_ptr<T>` deletes the owned object when it goes out of scope.
- `std::unique_ptr<T>` **cannot be copied, only moved**.

Unique pointer: creating

- Either explicitly with new, e.g.:

```
std::unique_ptr<message> ptr_msg{ new message{ "Hello, safety!" } };
```

- Or preferably with `std::make_unique`, e.g.:

```
auto ptr_msg = std::make_unique<message>( "Hello, safety!" );
```

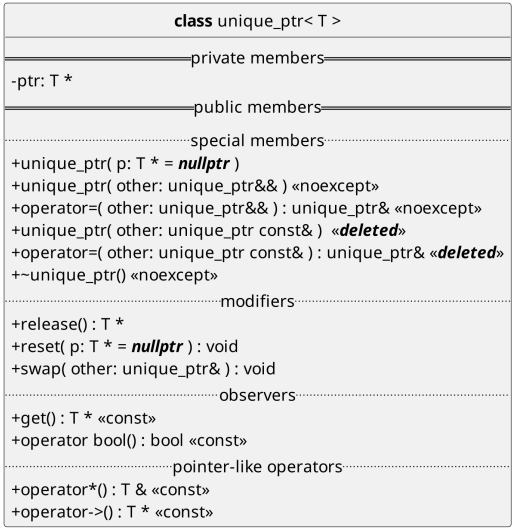
Unique pointer: is a normal pointer

With a `std::unique_ptr<T>`, you can:

- check if it owns an object
- use `->` to access its members
- use `*` to dereference it

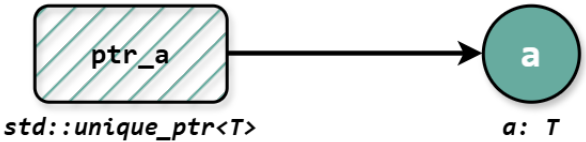
```
std::unique_ptr<T> ptr_a{ new T{ /* ... */ } };

if (ptr_a) {

  ptr_a->do_something();

  std::cout << *ptr_a << std::endl;

}
```

Unique pointer: the UML



Unique pointer: RAII for resources

`std::unique_ptr<T>` owns an object it points to

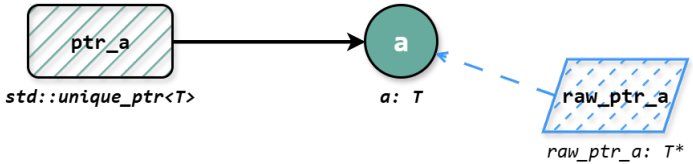


When the `std::unique_ptr<T>` goes out of scope, it deletes the owned object

Unique pointer: obtaining raw pointer

To get the raw pointer to an object std::unique_ptr owns:

```
T* raw_ptr_a = ptr_a.get();
```



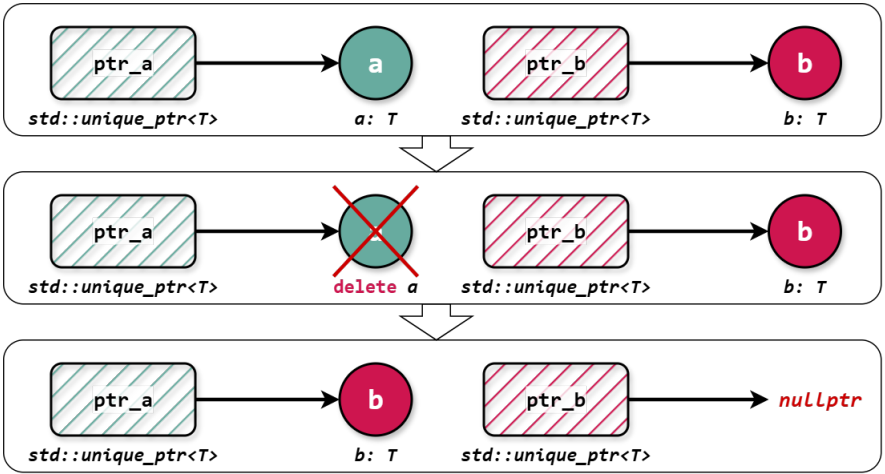
Unique pointer: ownership transfer

The ownership can be transferred from one std::unique_ptr to another:

```
std::unique_ptr<T> ptr_a{ new T{ /* ... */ } };
std::unique_ptr<T> ptr_b{ new T{ /* ... */ } };
```


```
ptr_a = std::move( ptr_b );
// or:
ptr_b.reset( ptr_a.release() );
```

Unique pointer: ownership transfer (cont.)



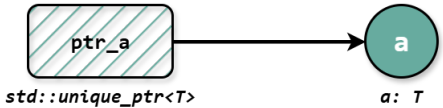
Unique pointer: swapping ownership

Two unique pointers can swap ownership of their objects:

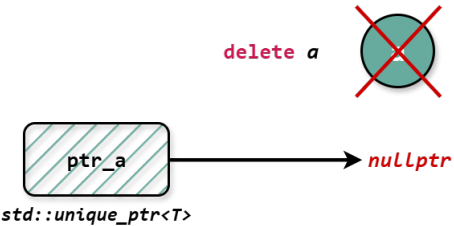

```
ptr_a.swap( ptr_b );
```

Unique pointer: deleting owned object

A unique pointer can delete the owned object explicitly:

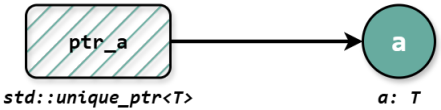



```
ptr_a.reset();
```



Unique pointer: releasing ownership

A unique pointer can release ownership of the owned object:




```
T* raw_ptr_a = ptr_a.release();
```

Unique pointer: creating

To create a `std::unique_ptr<T>`:

- Use the constructor and explicit `new`:

```
std::unique_ptr<T> ptr_text{ new T{ ... } };
```

- Or use the `std::make_unique` factory function (preferred):

```
auto ptr_text = std::make_unique<T>( ... );
```

Unique pointer: copying

std::unique_ptr cannot be copied!

```
std::unique_ptr<T> ptr_a{ new T{ /* ... */ } };

std::unique_ptr<T> ptr_b = ptr_a; // COMPILATION ERROR
```

Moving a std::unique_ptr is fine.

Unique pointer: example

```cpp
auto ptr_text = std::make_unique<std::string>("Hello, unique_ptr!");

if (ptr_text) {
  std::cout << *ptr_text << std::endl;

  std::cout << "Length: " << ptr_text->length() << std::endl;
}
```

Unique pointer and functions

A `std::unique_ptr<T>` can be passed to functions by:

- reference (no ownership transfer):

```
void process(const std::unique_ptr<T>& ptr) {
  // Use ptr without taking ownership
}

std::unique_ptr<T> ptr = std::make_unique<T>(...);
process(ptr);
```

- rvalue reference (ownership transfer):

Unique pointer and functions (cont.)

A `std::unique_ptr<T>` can be passed to functions by:

- reference (no ownership transfer):
- rvalue reference (ownership transfer):

```
void process(std::unique_ptr<T>&& ptr) {
  // Take ownership of ptr
}

std::unique_ptr<T> ptr = std::make_unique<T>(...);
process( std::move(ptr) );
```

Unique pointer can manage any resource

A pointer that automatically closes `FILE*` when it goes out of scope:

```
// Custom deleter for FILE*
struct FileDeleter {
  void operator()(FILE* file) const {
    if (file) {
      fclose(file);
    }
  }
};

std::unique_ptr<FILE, FileDeleter> filePtr(fopen("example.txt", "r"));
```

Always prefer to use a unique pointer

When you use a pointer to own a resource, always prefer `std::unique_ptr<T>` over a raw pointer `T*`

```
struct message {

  message(std::string_view str)
    : data_{ std::make_unique<std::byte[]>(str.size()) }
    , size_{ str.size() }
  {
    std::memcpy(data_.get(), str.data(), size_);
  }

private:
  std::unique_ptr<std::byte[]> data_;
  std::size_t size_;
};
```

Follow the rule of zero/five

In your classes either:

- Don’t define any of the special member functions (rule of zero)
- Or define all of them (rule of five):
  - Destructor
  - Copy constructor
  - Copy assignment operator
  - Move constructor
  - Move assignment operator

Follow the rule of zero/five

Rule of zero (this `message` cannot be copied):

```
struct message {

  message(std::string_view str);

private:
  std::unique_ptr<std::byte[]> data_;
  message_type type_;
};
```

Follow the rule of zero/five

Rule of five (this `message` can be copied and moved):

```
struct message {

  message(std::string_view str);

  message(const message& other);
  message& operator=(const message& other);
  message(message&& other) noexcept;
  message& operator=(message&& other) noexcept;
  ~message() = default;

private:
  std::unique_ptr<std::byte[]> data_;
  message_type type_;
};
```

Follow the rule of zero/five

```
message::message(const message& other)
  : data_{ std::make_unique<std::byte[]>(other.size()) }
  , size_{ other.size_ }
{
  std::memcpy(data_.get(), other.data_.get(), size_);
}

message& message::operator=(message other) {
  std::swap(data_, other.data_);
  std::swap(size_, other.size_);
  return *this;
}
```

Follow the rule of zero/five

```
message::message(message&& other) noexcept
  : data_{ std::exchange(other.data_, nullptr) }
  , size_{ std::exchange(other.size_, 0)  }
{
}

message& message::operator=(message&& other) noexcept {
  data_ = std::exchange(other.data_, nullptr);
  size_ = std::exchange(other.size_, 0);
  return *this;
}
```

Summary

- Use value semantics and references whenever possible
- Never use owning raw pointers
- Use `std::unique_ptr` for ownership of dynamic resources
- Follow the rule of zero/five for resource management