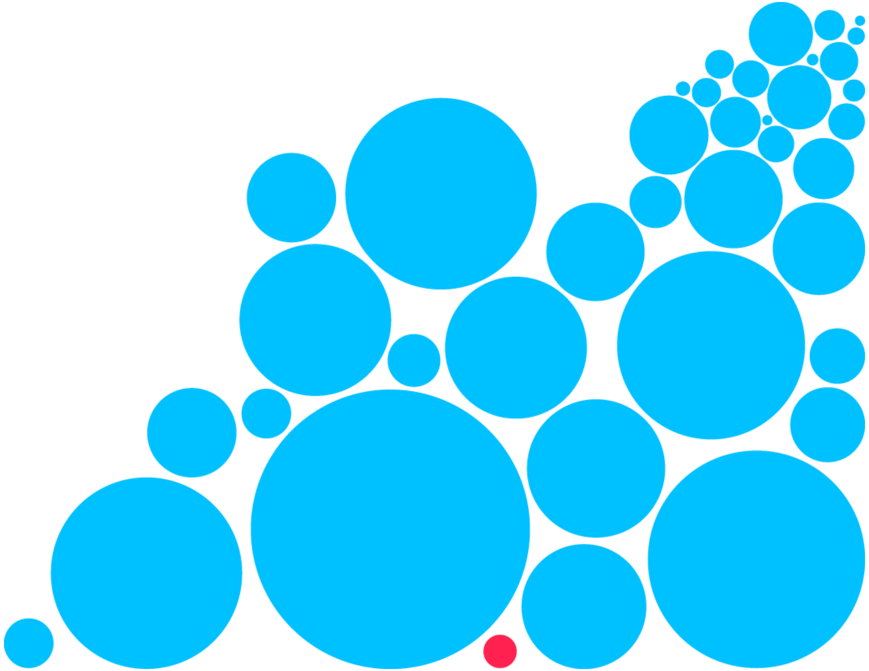
# Ownership

Lynden from the pub



# Agenda

- What it means
- Motivations
- Examples
- Managing ownership with types
- Other languages
- Gotchas
- Tips and Take-aways

# Agenda

- Please interrupt me
  - if something important is unclear or wrong
- Save language wars for when we have smart juice

# What I mean by ownership

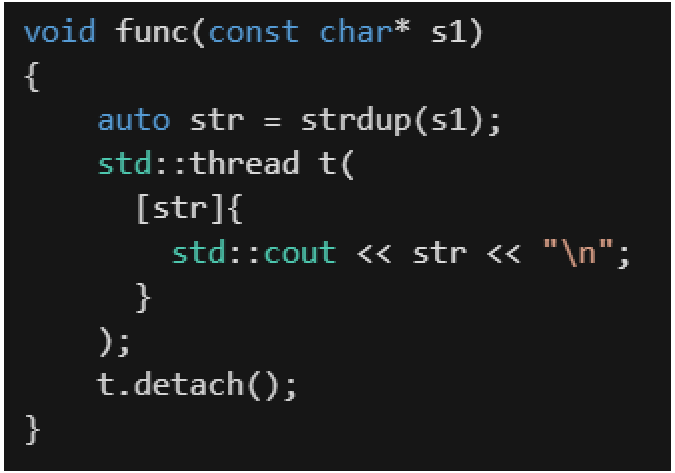
- Resource management
  - Allocation, registration, acquisition
  - Deallocation, reregistration, release-ification
- Applies to more than just memory
  - Locks etc
  - Mostly memory
- Ownership is not the same as aliasing
- Ownership matters even when not using smart pointers or C++

# Motivating errors

- Double delete / release
- Forgetting to delete / release
  - Exception safety
- Forgetting to allocate / acquire
- CVEs and security issues
  - Vast majority stem from memory management errors

# Examples

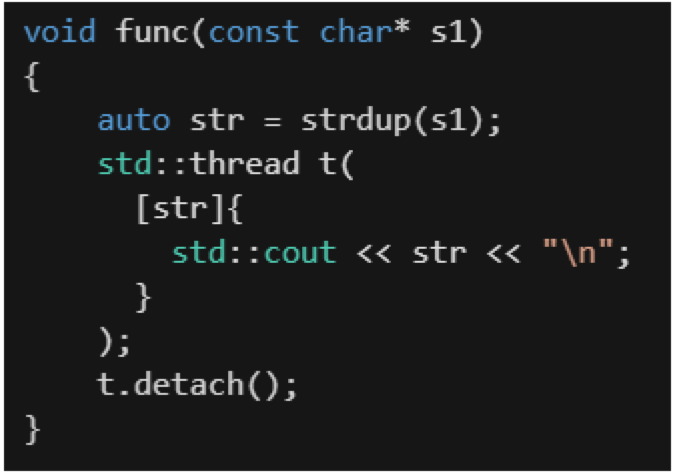
## Find the bug:



```
void func(const char* s1)
{
    auto str = strdup(s1);
    std::thread t(
      [str]{
        std::cout << str << "\n";
      }
    );
    t.detach();
}
```

# Examples

## Find the bug:

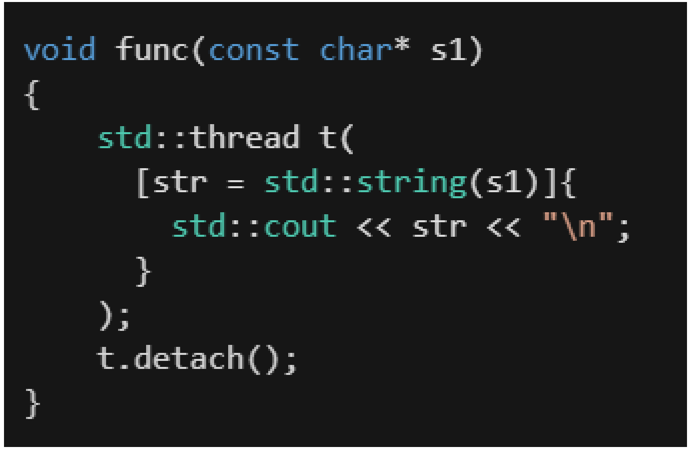


```
void func(const char* s1)
{
    auto str = strdup(s1);
    std::thread t(
      [str]{
        std::cout << str << "\n";
      }
    );
    t.detach();
}
```

Never frees str

# Examples

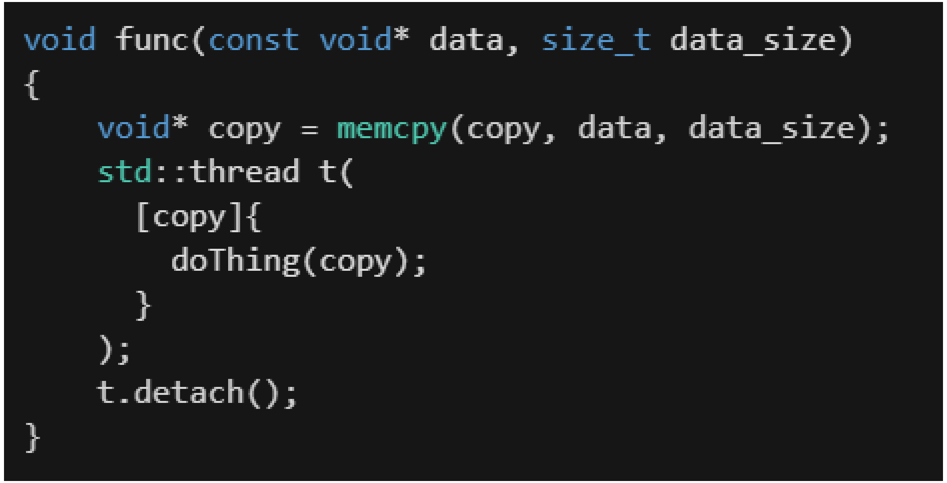
## A solution:



```
void func(const char* s1)
{
    std::thread t(
      [str = std::string(s1)]{
        std::cout << str << "\n";
      }
    );
    t.detach();
}
```

# Examples

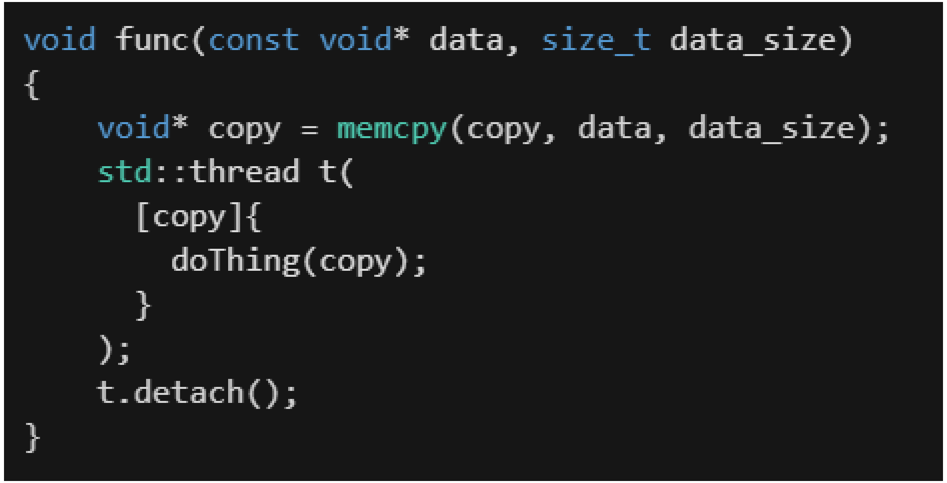
## Find the bug:



```
void func(const void* data, size_t data_size)
{
    void* copy = memcpy(copy, data, data_size);
    std::thread t(
      [copy]{
        doThing(copy);
      }
    );
    t.detach();
}
```

# Examples

## Find the bug:

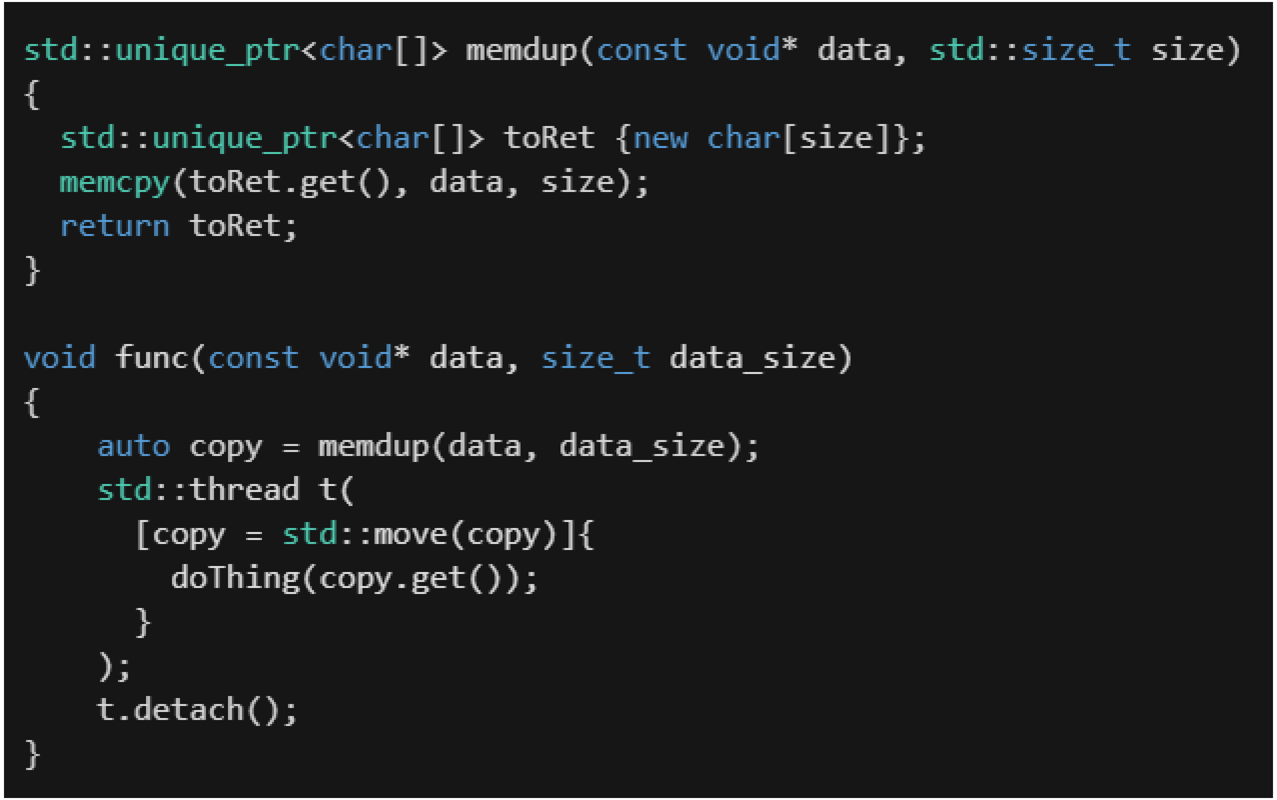


```
void func(const void* data, size_t data_size)
{
    void* copy = memcpy(copy, data, data_size);
    std::thread t(
      [copy]{
        doThing(copy);
      }
    );
    t.detach();
}
```

Never allocates copy

# Examples

## A solution:

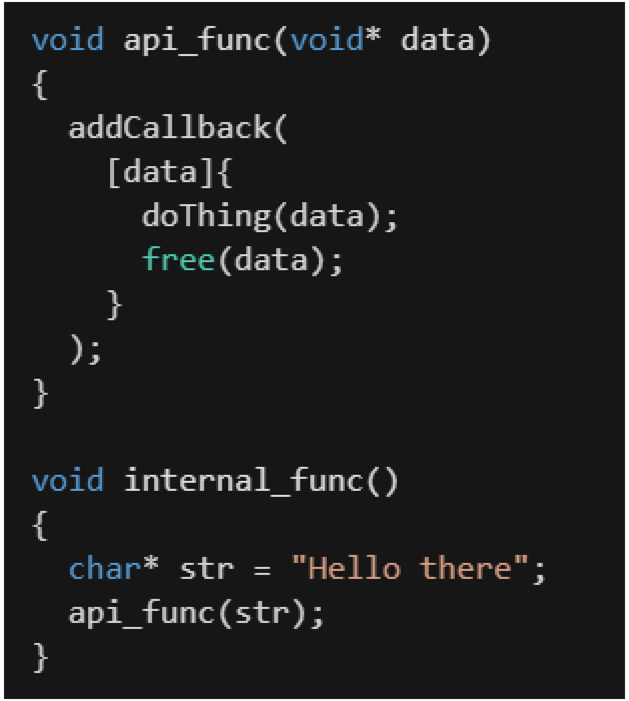


```cpp
std::unique_ptr<char[]> memdup(const void* data, std::size_t size)
{
  std::unique_ptr<char[]> toRet {new char[size]};
  memcpy(toRet.get(), data, size);
  return toRet;
}

void func(const void* data, size_t data_size)
{
    auto copy = memdup(data, data_size);
    std::thread t(
      [copy = std::move(copy)]{
        doThing(copy.get());
      }
    );
    t.detach();
}
```

Or use a static analyser

# Examples

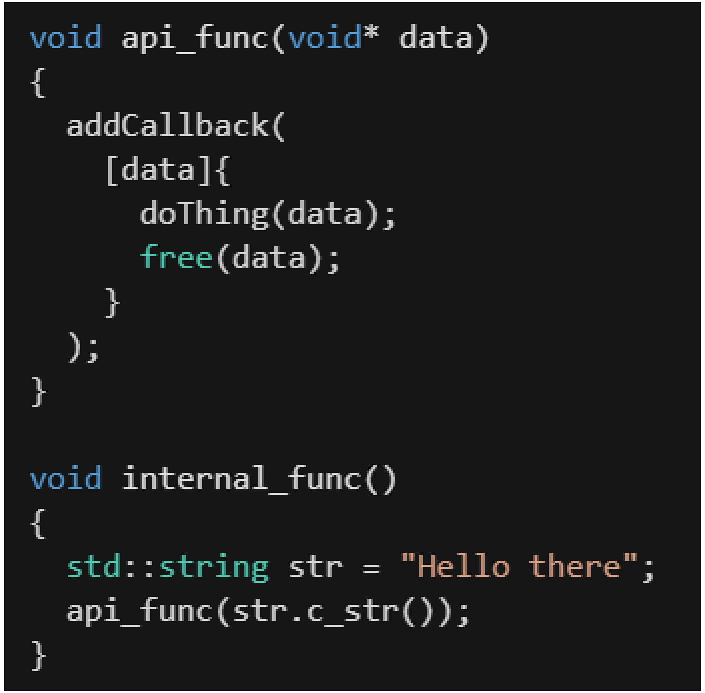
Find the bug:



```
void api_func(void* data)
{
  addCallback(
    [data]{
      doThing(data);
      free(data);
    }
  );
}

void internal_func()
{
  char* str = "Hello there";
  api_func(str);
}
```

# Examples

Find the bug:

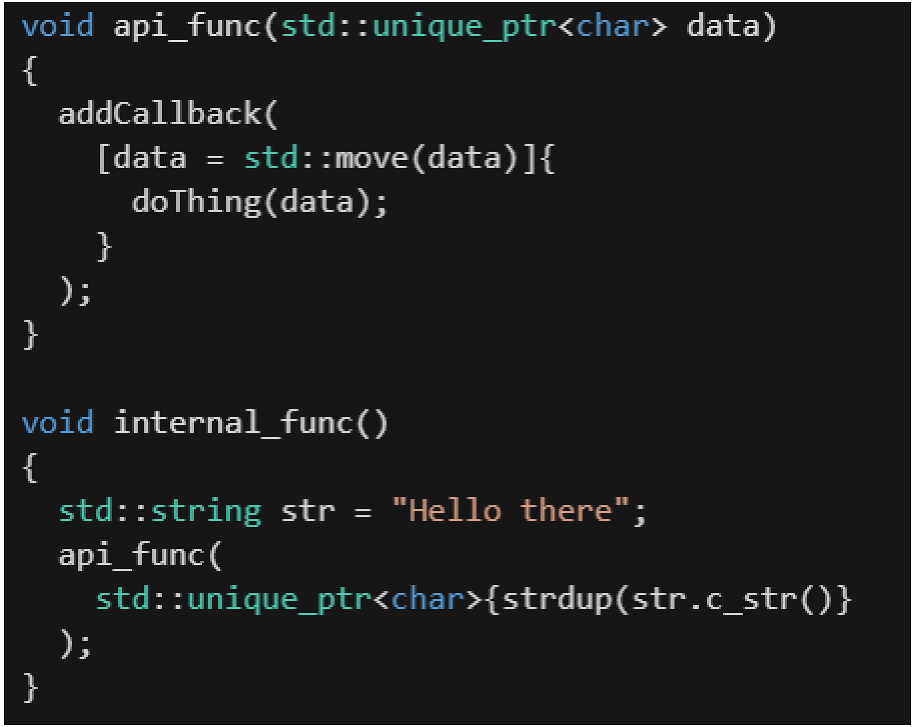


```
void api_func(void* data)
{
  addCallback(
    [data]{
      doThing(data);
      free(data);
    }
  );
}

void internal_func()
{
  std::string str = "Hello there";
  api_func(str.c_str());
}
```

str’s buffer is freed twice

# Examples

A solution:



```
void api_func(std::unique_ptr<char> data)
{
  addCallback(
    [data = std::move(data)]{
      doThing(data);
    }
  );
}

void internal_func()
{
  std::string str = "Hello there";
  api_func(
    std::unique_ptr<char>{strdup(str.c_str()}
  );
}
```

# Examples

- typedef or using declaration that adds "owning" to the pointer type names.
- Better yet, using a smart pointer that handles the resource management for you and use APIs that support them

# Gotchas 1

- memory leaks and double deletes still possible - you still need to think about ownership

- Smart pointers also don't remove the need to do null checks

# Gotchas 2

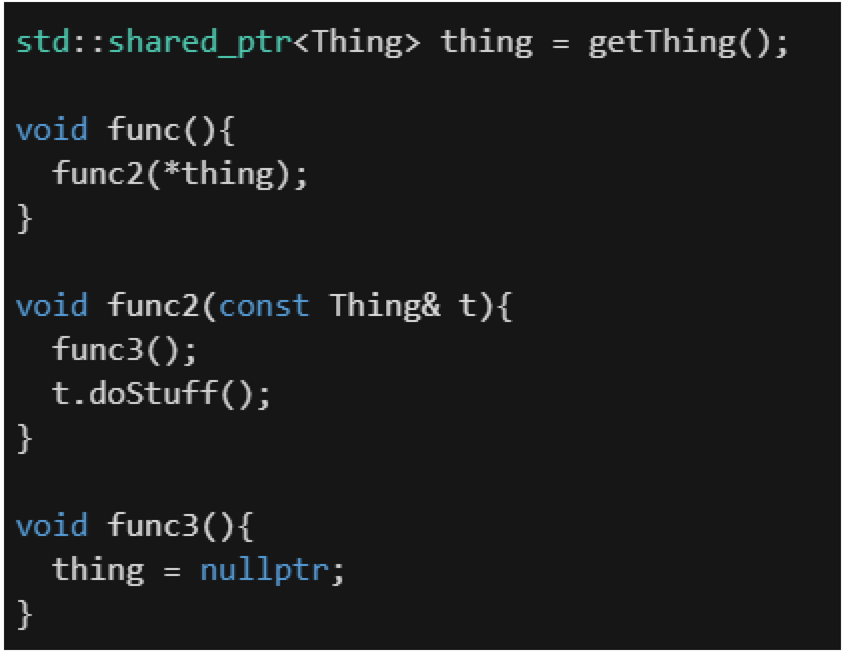
- Separate chains of std::shared_ptr to the same object **will** cause issues
  - use copy constructor or copy assignment operator
- typedefs and using declarations are weak;

# Gotchas 3

- If you have a smart pointer that's aliased or otherwise accessible through means outside of function parameters, e.g. a global object or something that can be looked up, then make sure you "pin" an owning copy of the smart pointer before passing a pointer or reference to the object down a call chain
    - A function you call might reset the smart pointer, which might otherwise destroy the object

# Gotchas 3

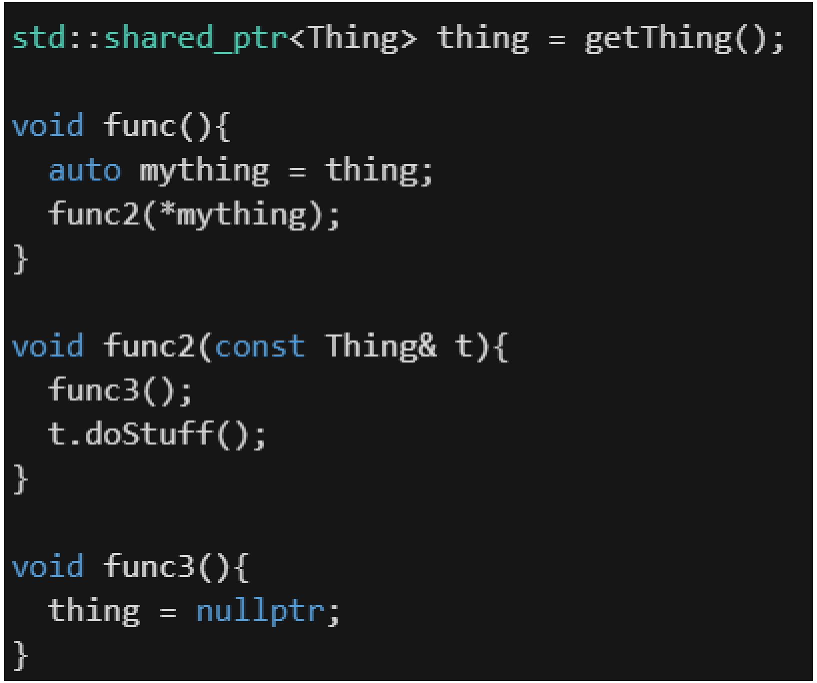
Example:



```
std::shared_ptr<Thing> thing = getThing();

void func(){
  func2(*thing);
}

void func2(const Thing& t){
  func3();
  t.doStuff();
}

void func3(){
  thing = nullptr;
}
```

# Gotchas 3

Good:



```
std::shared_ptr<Thing> thing = getThing();

void func(){
  auto mything = thing;
  func2(*mything);
}

void func2(const Thing& t){
  func3();
  t.doStuff();
}

void func3(){
  thing = nullptr;
}
```

# Tips and take-aways

- Dos
  - Do use the type system to enforce or at least inform ownership
  - Do pass by const& by default
  - Do prefer automatic storage over smart pointers
  - Do make sure you understand the ownership semantics of C and legacy C++ APIs that you interface with and try to represent that in your code

# Tips and take-aways

- Don'ts
  - Don't use shared ownership unless you really need to (it's pretty rare)
    - Sorry Marx :(
  - Don't take parameters by smart pointer unless you need to manipulate ownership
  - Don't put things on the free store / heap unless you have to (it's less common than most C++ developers think)
  - Don’t use `new` and `delete` – even if you’re using the heap

# Find out more

- the article I wrote as the basis for this talk
  - "I didn't have time to write a short letter, so I wrote a long one instead."
  - It has much more detailed explanations
  - Also has guidelines for choosing the right type in various situations
- C++ core guidelines (it’s on github)

# Thank

- Please ask me anything at any time
- Please give me ideas for similar talks
- ༼ つ ◕_◕ ༽つ Please gib feedbacks ༼ つ ◕_◕ ༽つ
- Do the significant eat and drink