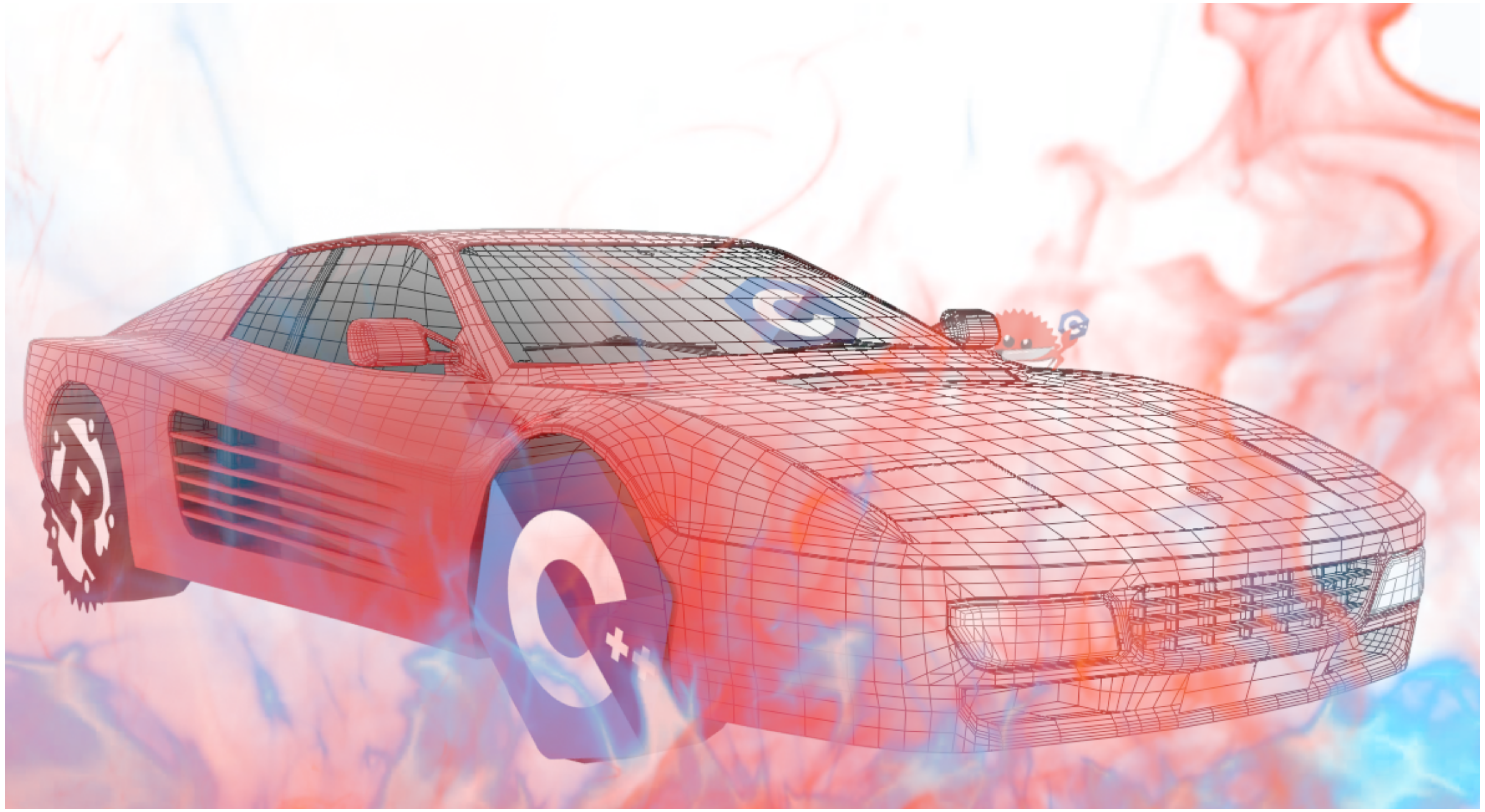
Effizientes Programmieren in C, C++ und Rust

Object-Oriented C++

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024



Exercise Two

Benchmark 1

Context: `tests/tokenize.cc:221`

- `private` / 24.05 13:15 / 8 days, 1 hour, 45 mins
- `default` / 19.05 19:33 / 4 days, 8 hours, 3 mins
- `loop` / 19.05 16:06 / 4 days, 4 hours, 36 mins
- `thread_local` / 17.05 13:58 / 2 days, 2 hours, 28 mins
- No more extensions
- Please fill survey

Table of Contents

Member Functions

Yes, You've Been Using Them

```
struct look_functions {

    void nothing() {}

    int inline_fkt() { // declare+define
        return 4;
    }

    int out_of_line(); // only declare

    int member_var = 4;

    int thisptr() {
        return this->member_var + this->out_of_line();
    }
};

int look_functions::out_of_line() { // only define
    return 5;
}
```

- In C++: `struct`s can have ~~methods~~ member functions
- They need to be *declared* in the struct, but can be *defined* both inside and outside of the class
- `this`-pointer: explicitly access variables

Member Functions

This-Pointer Shenanigans

- `this`-keyword also in Java
- in C++: *pointer* to the object
- Can be required (e.g., as a prefix)
  - when requiring pointer to self (e.g., to return)
  - shadowing (should mostly be avoided)
  - When that weird error message about `undeclared identifier` (clang) appears

The following example is for reference purposes only, and you need not understand it completely

```cpp
template<typename T>
struct foo{T a;};
template<typename T>
struct bar : public foo<T> {
    int wrong() { return a;}
    int right() { return this->a;}
};

int main() {
    return bar<int>().wrong() + bar<int>().right();
}
```

```
g++ -Wall -Wextra -pedantic -std=c++20 help.cc
help.cc: In member function ‘int bar<T>::wrong()’:
help.cc:5:26: error: ‘a’ was not declared in this scope
5 |     int wrong() { return a; }
  |                          ^
```

Member Functions

Performance 🚴 (non-`virtual` member functions)

```
struct foo {
    void whereami() { std::cout << this << std::endl; }
};
```

- Implementation: as you would expect, a method with another parameter

```
// non-conforming pseudocode
foo::whereami(foo *this) { std::cout << this << std::endl; }
```

- Literally as fast as function call
- Has all benefits of functions, e.g., inlining ↗

- "Proof" it's just a pointer (maybe undefined behavior (UB))

```
#include <iostream>

struct foo {
    void whereami() { std::cout << this << std::endl; }
};

int main() {
    static_cast<foo*>(nullptr)->whereami();
}
```

```
g++ -Wall -Wextra -pedantic whereami.cc && echo '=======' && ./a.out
whereami.cc: In function ‘int main()’:
whereami.cc:7:41: warning: ‘this’ pointer is null [-Wnonnull]
7 |     static_cast<foo*>(nullptr)->whereami();
  |     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~
whereami.cc:4:10: note: in a call to non-static member function ‘foo::whereami()’
4 |     void whereami() { std::cout << this << std::endl; }
  |          ^~~~~~~~
=======
0
```

- Second proof: `objdump --disassemble`

`const` Objects

A Story About Wrong Defaults

- Accidental/Intransparent modification dangerous
  - Invalidate invariants
  - Unexpected Behavior
  - Caller/Callee mismatch in expectations
- Solutions?
  - Ignore (often used, sadly) <
  - Comments <
  - Static checkers <
  - Tooling <
  - Runtime checks <
  - The type system

C++: type system support

```cpp
int const_ref(const int &c) {
    return c + 1;
}
int main() {
    int should_be_const = 5;
    // look, const ref for mutable object works
    // slideware, usually pass 1 int by copy

    const int actually_const = const_ref(should_be_const);

    should_be_const += 4;

    // error: assignment of read-only variable ‘actually_const’
    // actually_const += 3;
}
```

- Great: Compiler errors!
- Less great: wrong default
  - Should be `const` by default

`const` Objects

Making Members More `mutable`

```
struct foo {
    int x;
    void add1() {x += 1;}
};

int main() {const foo f; f.add1();}
```

Is this valid? No

```
struct foo {
    int x;
    int add1() {return x + 1;}
};
int main() {const foo f; f.add1();}
```

Is this better? (Sadly) No

```
struct foo {
    int x;
    int add1();
};
int main() {const foo f; f.add1();}
```

Is this valid? No

- (Not only) because of declaration only: we have to promise
- Add `const` at end of function

```
struct foo {
    int x;
    int add1() const;
};
int main() {const foo f; f.add1();}
```

- Now only linker error...

```
int foo::add1() const {return x + 1;} // fixed
```

- Still: compiler error on member modification.
- (Members marked `mutable` *can* still be modified. Use very infrequent)
- We can overload for const/non-const
  - Actually useful: e.g., `std::vector` returns either const or mutable ref
  - Overload your accessors

Working `class` Hero

Is Something to Be

```
struct a_struct {
    int x = 0;
};

class a_class {
    int x = 0;
};
```

Difference? Visibility

Visibility Levels in C++?

- `public`: (Default in `struct`)
  - Everyone can see contents
- `protected`:
  - Only derived classes can see contents
- `private`: (Default in `class`)
  - Only class members and `friend`s can see contents

Aside: `friends`

- Often: strongly linked classes
  - Iterators
  - Factories
  - Accessors
- In Java: same package, or subclass
- In C++: explicit access to members
- Either functions or classes

```
class foo {
    int x; // private
public:
    int y;
    static foo make_foo() {return foo{};}
    friend std::ostream& operator<<(
              std::ostream&str, const foo &f) {
        return str << f.x << "," << f.y;
    }
}
```

`class` Conflicts

Declaring War

```
struct foo;
class foo {
    int x;
};
```

- Is this legal?
- Let's ask the compiler

```
clang++ -Wall -Wextra -pedantic help3.cc
help3.cc:2:1: warning: 'foo' defined as a class here but previously declared as a struct;
this is valid, but may result in linker errors under the Microsoft C++ ABI [-Wmismatched-tags]
    2 | class foo {
      | ^
help3.cc:1:1: note: did you mean class here?
    1 | struct foo;
      | ^~~~~~
      | class
help3.cc:3:6: warning: private field 'x' is not used [-Wunused-private-field]
    3 |         int x;
      |             ^
2 warnings generated.
```

Inheritance Kingdom

Selecting Your Heir Efficiently Since 1983

```cpp
struct Base {
    int based;
};

struct Derived : public Base {
    int derived;
};
```

- `Derived` *is a* `Base`
- Minimal example, what do we actually use inheritance for?
  - Defining/Modelling interfaces & object relations
  - Specialization of general objects
  - Dynamic Dispatch (selecting functionality at runtime)
  - Code re-use
  - Post-compile extensibility

Quick reminder: idiomatic object-oriented (OOP) design

- A deer which can eat() *is a* mammal which can eat() *is an* animal which can eat()
- Peter who can talk() and research() *is a* lecturer who can teach() and Peter *is a* researcher who can research()
- An amphibious craft which can move() *is a* ship which can move() and *is a* car which can move(), and both a ship *is a* vehicle which can move() and a car *is a* vehicle which can move().

Single Inheritance

Simple Start

A deer which can eat() *is a* mammal which can eat() *is an* animal which can eat()

```
#include <iostream>
auto p = [](auto el) { std::cout << el; };

struct animal {
    virtual void name() const { p("Animal"); }
    void eat() const { this->name(); p(" eating\n"); }
};
struct mammal : public animal {
    void name() const override { p("Mammal"); }
};
struct deer : public mammal {
    void name() const override { p("Deer"); }
    void eat() const { p("Deer consuming\n"); }
};

int main() {
    animal a;       a.eat();  a.name();  p("\n");
    mammal m;       m.eat();  m.name();  p("\n");
    deer d;         d.eat();  d.name();  p("\n");
    animal &ar = a; ar.eat(); ar.name(); p("\n");
    animal &mr = m; mr.eat(); mr.name(); p("\n");
    animal &dr = d; dr.eat(); dr.name(); p("\n");
}
```

- Difference `virtual` and non-virtual functions
  - `virtual`: can be *overridden* (use `override` even though not strictly necessary)
  - `virtual`: Mark methods as `virtual` to enable this behavior, *once*
  - `virtual`: Do not mix virtual functions and overloading!
  - non-virtual: can be *shadowed*, but not dynamically selected

For future reference: `eat()` is non-virtual here!

Single Inheritance

Simple Implementation

- `sizeof(animal)`? 8 (one pointer)
- Let's do some undefined behavior!

```
    p(sizeof(ar)); p("\n");
    p(*reinterpret_cast<void***>(&ar));p("\n");
    p(**reinterpret_cast<void***>(&ar));p("\n");
    p(sizeof(mr)); p("\n");
    p(*reinterpret_cast<void***>(&mr));p("\n");
    p(**reinterpret_cast<void***>(&mr));p("\n");
    p(sizeof(dr)); p("\n");
    p(*reinterpret_cast<void***>(&dr));p("\n");
    p(**reinterpret_cast<void***>(&dr));p("\n");
    reinterpret_cast<void(*)(void*)>(
        **reinterpret_cast<void***>(&dr))(&dr); p("\n");

8
0x402088
0x4013fe
8
0x402070
0x40144e
8
0x402058
0x40146c
Deer
```

What's happening here?!

- We're doing manual object orientation
- Start of object: *V-Table Pointer*
- V-Table: contains (another) pointer to all virtual functions
- Our contrived example: exactly one virtual function, i.e., the first v-table entry is our `name()` method
- So, one indirect fetch and one indirect jump: virtual methods
- **Actually solid performance!**🚴
- Future reference: One additional dereference to undo `operator&()`.

A *More Readable* Example

```
#include <iostream>
struct mystruct;
using functionptr = void(*)(mystruct* self, int param);

struct mystruct {
    functionptr *v_ptr;
    int a;
    int b;
};
void add_print_a(mystruct *self, int param) {
    std::cout << self->a << " " << param << " " << self->a + param << std::endl;
}
void sub_print_b(mystruct *self, int param) {
    std::cout << self->b << " " << param << " " << self->b - param << std::endl;
}
constexpr auto ADD_PRINT_A = 0;
constexpr auto SUB_PRINT_B = 1;
functionptr v_table [] = {add_print_a, sub_print_b};

mystruct make_mystruct(int a, int b) {
    return mystruct {v_table, a, b};
}
int main() {
    auto s = make_mystruct(1, 2);
    s.v_ptr[ADD_PRINT_A](&s, 5);
    s.v_ptr[SUB_PRINT_B](&s, 7);
}
```

- Explicit `v_table`
- Explicit `v_ptr`
- Explicit dynamic dispatch

Guaranteed 100% syntactic sugar free.

Pure Virtual

Cool Kids Don't Say Abstract.

- You know Java `abstract` methods
- Usage: Declare interface, but do not / cannot implement yet.
- Defined in subclasses
- Obviously: also in C++
- Called: the *pure virtual member function*

```cpp
#include <memory>
#include <iostream>
#include <string>

struct nameable {
    virtual std::string get_name() const = 0;
};

struct peter : public nameable {
    std::string get_name() const override {
        return "Peter"; }
};

int main() {
    // error: cannot declare variable ‘name’
    // to be of abstract type ‘nameable’
    // nameable name;
    std::unique_ptr<nameable> base_ptr =
        std::make_unique<peter>();
    std::cout << base_ptr->get_name() << std::endl;
}
```

Pure Virtual Shenanigans

Concrete Abstract Out-Of-Class Member Functions

- Default-implement pure virtual methods
- Usable in subclasses
- *Note: use qualified name (prefix class-name and ::) to devirtualize (works always)*
- See example on other side
- Must be implemented outside of class

```cpp
#include <iostream>
#include <string>

struct nameable {
    virtual std::string get_name() const = 0;
};

std::string nameable::get_name() const {
    return "unimplemented";
}

struct lazy_lecturer : public nameable {
    std::string get_name() const override {
        return this->nameable::get_name();
    }
};

int main() {
    lazy_lecturer lecturer;
    std::cout << lecturer.get_name() << std::endl;
}
```

public, protected, and private Inheritance

Ich sehe was, was du nicht siehst, und das ist `segmentation fault: core dumped`

- Simple Idea: *visibility* of inherited members same as inheritance type
  - `public`: just copy visibility (default in `struct`)
  - `protected`: Make everything protected
  - `private`: Make everything private (default in `class`)
- Export using `using`
- private/protected inheritance: reuse code
- public inheritance: *is a* inheritance (and template metaprogramming)

```
struct x {
    int get_prot() { return prot; }
protected:
    int prot;
};

struct z : protected x {
    using x::prot;
};

int main() {
    x x;
    // error: ‘int x::prot’ is protected
        // within this context
    //(void)x.prot;
    z z;
    (void)z.prot;
}
```

Multiple Inheritance

Biology Requires Two Biological Parents

Peter who can talk() and research() *is a* lecturer who can teach() and Peter *is a* researcher who can research()

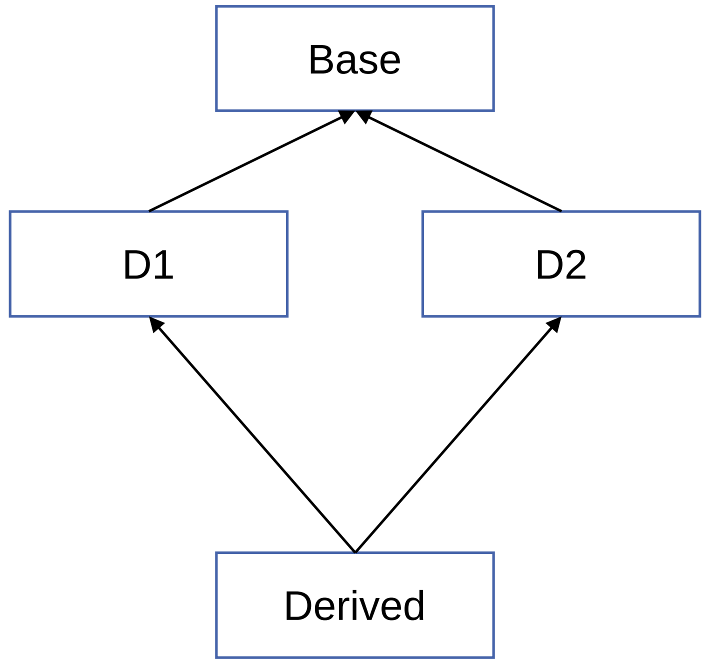
```
#include <iostream>
#include <string>

struct researcher {
    std::string orcid;
    virtual void research() const {
        std::cout << "researching()" << std::endl;
    }
};

struct lecturer {
    std::string lecture_name;
    virtual void teach() const {
        std::cout << "teaching()" << std::endl;
    }
};

struct peter : public researcher, lecturer {
    void research() const override {
        std::cout << "reseaching GPUs" << std::endl;
    }
    void teach() const override {
        std::cout << "teaching C++" << std::endl;
    }
};
```

```
int main() {
    lecturer l; l.teach();
    researcher r; r.research();
    peter p; p.teach(); p.research();
    lecturer *lp = &p; lp->teach();
    researcher *rp = &p; rp->research();
    std::cout << lp << " " << rp << std::endl;
}
```

- Multiple inheritance: Works as you would expect
- Several subobjects
- Several v-table pointers
- Pointer cast may change pointer (select correct v-table)
- Performance 🚴: similar to single inheritance

Yo Dawg We Heard You Like Pointers, So We Added Pointers to Your Pointers to Point to Your Pointers

- Diamond problem: D1, D2 derive Base, Derived derives D1, D2
  - Suddenly: two Base subobjects
  - Inconsistency is preprogrammed
- Solution: unite the base objects
  - *Just add a pointer*
  - In D1, D2: point to common Base
  - Therefore: *virtual inheritance* at D1, D2, not really at Derived




```
#include <iostream>
struct Base {
    int x;
};
struct D1_v : virtual Base {};
struct D2_v : virtual Base {};
struct Derived_v : D1_v, D2_v {};
struct D1_nv : Base {};
struct D2_nv : Base {};
struct Derived_nv : D1_nv , D2_nv {};

int main() {
    Derived_v v {};
    Derived_nv nv {};
    nv.D1_nv::x = 1; nv.D2_nv::x = 2;
    v.D1_v::x = 1; v.D2_v::x = 2;
    std::cout << nv.D1_nv::x << " " << nv.D2_nv::x <<
        " " << v.D1_v::x << " " << v.D2_v::x << std::endl;
    std::cout << sizeof(v) << " " << sizeof(nv) << std::endl;
}
```

Result:

```
1 2 2 2
24 8
```

Performance: use it if you really need it

Class Invariants

To Create or to Destroy

- An object has a lifetime
  - Starts: when created (duh)
  - Stops: when destroyed
  - Creation: When first defined
  - Destroyed: At end of enclosing scope
- An object (may) have *invariants*
- Invariants hold from end of creation $\Rightarrow$ begin of destruction
- Invariant creation and destruction needs to be programmable
- $\Rightarrow$ *constructors and destructors*

CONSTRUCTOR

- two tasks:
  - Call constructors of subobjects!
  - Establish invariants of own object
- Syntax for both in C++
- Can be overloaded

DESTRUCTORS

- Only one destructor, cannot be overloaded
- Only destroy invariants of own object
- Subobject-destructors called implicitly at end
- Make destructor virtual if you intend to inherit

Constructors

Use Initializer Lists!

```
struct bad_vector {
    int size;
    int *x;
    bad_vector(int num) : size(num), x(nullptr) {
        if (size > 0)
            x = new int[num];
    }
    // constructors calling constructors is fine
    bad_vector() : bad_vector(0) {}
    ~bad_vector() {
        if (size > 0) delete []x;
    }
};
```

- constructor: `class name`, no return type
- destructor: ~`class name`

- Call subconstructors explicitly!
  - Initializer list: the part between `:` and `{`
  - Otherwise: default constructor + assignment
- Initialized *in order of declaration*, not in order of list

  `warning: 'foo::y' will be initialized after [-Wreorder]`
- Also used for base class constructors

()-Constructors vs {}-Constructors

Wrong Defaults 2.0

- In C++11: Added `{}` constructors
- Avoids dangerous casts
  (`unsigned i {-5};` does not compile)
- Allows explicit enumeration of object
- Those two conflict
  - `std::vector<int> v(5);`
    5 integers, each zeroed
  - `std::vector<int> v{5};`
    1 integer with value 5

- Rule of thumb: Use `{}` whenever possible
- Be wary of the difference

RAII

Let's Start the Acronymization

RAII?

- Resource Aquisition Is Initialization
- Good Idea: Never have incomplete invariants
- Bad (non-RAII) idea:

```
foo f;

if(!f.setup()) return ERROR;
// use f
f.destroy();
```

  - Means: after construction, nothing is to be done
  - Also means: no explicit cleanup, all done in destructor

RAII is a pattern, not really a language feature

- Use whenever resources are needed
  - Memory (`std::unique_ptr`)
  - Mutexe (`std::lock_guard`)
  - Database handles
  - Network connections
  - ...
- Foreshadowing: Can Constructors Fail?

Placement new

In-Memory construction

- Issue: sometimes, we can't decide where to put our object
  - Shared Memory
  - Manual mmap
  - Device memory
  - The special allocator our game engine uses
  - ...
- Solution: call constructor anywhere
- `new (pos) TYPENAME(params...);`
- *Seldomly used, but extremely handy sometimes*
- Performance: opens up a lot of optimizations 🚴

- Issue: the memory survives, no implicit destructor
- Solution: call destructor explicitly

```
foo *mem = (foo*)malloc(sizeof(int));
new(mem) foo(5);
mem->~foo();
```

Copy/Move Semantics

Add More to Do Less

- C++ has lots of objects
- C++ has value semantics: except if explicitly stated, everything is a value
- Usually: return by value
- Also: destruction at end of function
- Solution: just copy to other object
- Is that fast? 🚴 No!
- Some optimizations *(return value opt)* help
- Copying so important, there are special functions

```
struct copy {
    int x;
    copy() = default;
    copy(const copy&other) = default;
    copy& operator=(const copy&other) = default;
};
```

- Still slow
- Often: No copy needed, just transfer resource
- C++11: "Move"
- Sadly: reference constructor already blocked
- Simple solution: add another reference!
- `copy(copy&& move)`
- This reference type is usually not autogenerated
- But can be enforced: `std::move`
- Do not use on return!
- *Aside: you see that default?*

Copy/Move Semantics

A Rule of Nothing

- 5 special functions:
  - Copy constructor `copy(const copy& other)`
  - Copy assignment `copy &operator=(const copy& other)`
  - Move constructor `move(const move& other)`
  - Move assignment `move &operator=(const move& other)`
  - Destructor `~cls()`
- They come in pairs (constructor & assignment), which should *never be implemented alone*
- If you have resources, you need a destructor

- Rule of 5: Implement them all (good)
- Rule of 3: Implement only copy (correct semantic, slower, C++98, ok)
- Rule of 3: Implement only move (disables copy, often good (`std::unique_ptr`))
- Rule of 0: Don't implement anything, because someone else already did the work (great!)
- *Aside: explicit* `= delete;` *possible, too (sometimes useful)*

What we did today

- Today:
  - Member functions
  - `const`
  - `class`/`public`/`protected`/`private`
  - Inheritance
  - Class invariants
  - Copy and move semantics
- Next Week:
  - Constructors Can Fail (Gracefully)