Concepts of C++ Programming

Lecture 12: Inheritance

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

Object-Oriented Programming

Concepts of object-oriented programming:

- Data abstraction/encapsulation
  - ⇝ Classes in C++
- Inheritance
  - ⇝ Class derivation in C++
  - Derived classes inherit the members of the base class
- Dynamic Binding (Polymorphism)
  - ⇝ Virtual functions in C++
  - Derived classes can override methods of base classes
  - By default, C++ inheritance is non-polymorphic

Derived Classes

- Class may be derived from one or more base classes
  - ⇝ Inheritance hierarchy
- Syntax: `class` *`class-name`* `:` *`base-specifier-list`*
- Base specifiers: `public`/`protected`/`private`; `virtual` (optional)

```
struct Base {
    int a;
};
struct DerivedA : public Base {
    int b;
};
struct DerivedB : private Base, public DerivedA {
    int c;
};
```

Constructors

- Constructors of derived classes also construct base classes
  1. Direct base classes are initialized in left-to-right order
  2. Non-static data members are initialized in declaration order
  3. Constructor body is executed
- Base classes default-initialized unless specified otherwise
  - Delegating constructor syntax: `Derived() : Base(arg1, arg2) {}`

Constructors: Example

```
struct Base {
    Base() { std::println("Base()"); }
    Base(int) { std::println("Base(int)"); }
};
struct Derived : public Base {
    Derived() { std::println("Derived()"); }
    Derived (int a, int) : Base(a) {
        std::println("Derived(int,␣int)"); }
};
int main() {
    Derived a;
    Derived b{12, 34};
}
```

Output:

```
Base()
Derived()
Base(int)
Derived(int, int)
```

Copy Constructors

△ Quiz: What is the output of the program?

```
#include <print>
struct A {
    A() { std::println("A"); }
    A(const A&) { std::println("a"); }
};
struct B : A {
    B() { std::println("B"); }
    B(const B&) { std::println("b"); }
};
int main() {
    B b1, b2(b1);
}
```

A. (compile error) B. (unspecified) C. `ABAB` D. `ABAb` E. `ABab`

Destructors

- Destructors are executed in the opposite order as constructors

◯ Quiz: What is the output of the program?

```
#include <print>
struct A { ~A() { std::print("A"); } };
struct B { ~B() { std::print("B"); } };
struct D : A, B { ~D() { std::print("D"); } };
int main() { D d; }
```

A. (unspecified) B. `ABD` C. `BAD` D. `DAB` E. `DBA`

Constructors: Multiple Inheritance

★ Quiz: What is the output of the program?

```
#include <print>
struct A { A() { std::print("A"); } };
struct B : A { B() { std::print("B"); } };
struct C : A { C() { std::print("C"); } };
struct D : B, C { D() { std::print("D"); } };
int main() { D d; }
```

A. (compile error) B. (unspecified) C. `ABCD` D. `ABACD` E. `BACD`

Unqualified Name Lookup[147]

- Names can be defined multiple times in inheritance hierarchy
- Unqualified (no ::) lookup algorithm decides which name to choose
- Approximation: declarations in derived classes hide names from base classes

```
struct A { void a(); };
struct B : public A { void a(); void b() { a(); /* B::a() */ } };
struct C : public B {
  void c1() { a(); /* B::a() */ }
  void c2() { A::a(); /* A::a() */ } // qualified lookup
};
```

[147] https://en.cppreference.com/w/cpp/language/unqualified_lookup

Unqualified Name Lookup: Diamond Inheritance

```
struct A { void a(); };
struct B1 : public A { };
struct B2 : public A { };

struct C : public B1, public B2 {
  void c1() { a(); /* ERROR: ambiguous, a() present in B1 and B2 */ }
  void c2() { B1::a(); /* OK */ }
};
```

Object Representation

- Base classes are stored as subobjects of the derived class

```
#include <cstddef>
struct A {
    int& a1;
    char a2;
};
struct B {
    short b;
};
struct C : public A, public B {
    int c;
};
static_assert(offsetof(C, a1) == 0);
static_assert(offsetof(C, a2) == 8);
static_assert(offsetof(C, b) == 10);
static_assert(offsetof(C, c) == 12);
```

Inheritance Modes: `public`

- `public` inheritance: public base members become public derived members
  protected base members become protected derived members
- Default when derived class declared as `struct`

- Typically used to model subtyping/is-a relationship
  - Pointers/references of derived should be usable when base class is expected
  - Derived class should maintain invariants of base class
  - Derived class should not strengthen preconditions of overridden members
  - Derived class should not weaken postconditions of overridden members

Inheritance Modes: `private`

- `private` inheritance: public/protected base members become private derived members
- Default when derived class declared as `class`
- Derived class can be used as base class only in derived class

- Sometimes useful
  - Mixins (e.g., special storage management methods)

Inheritance Modes: `protected`

- `protected` inheritance: public/protected base members become protected derived members
- Derived class can be used as base class in all further derived classes

- Rarely useful
- “Controlled polymorphism”: inheritance should be shared with subclasses

(Non-)Polymorphic Inheritance

◯ Quiz: What is problematic about this code?

```
#include <vector>
struct Base { int a; };
struct Derived : Base { int b; Derived(int a, int b) : Base{a}, b(b) {} };
void foo(std::vector<Base>& v) {
  v.push_back(Derived(1, 2));
}
```

A. Compile error: cannot convert `Derived` to `Base`.

B. The vector only stores `Base`; the value for `b` is discarded.

C. The vector stores `Derived`, but it consumes two entries.

D. Nothing, the vector now contains a `Derived` as last element.

(Non-)Polymorphic Inheritance

△ Quiz: What is the exit code of this program?

```
struct A { int compute() { return 5; } };
struct B : public A {
    int compute() { return A::compute() + 10; }
};
int callCompute(A& a) { return a.compute(); }
int main() { B b; return callCompute(b); }
```

A. Compile error: `A::compute` – attempt to call as static member
B. Compile error: cannot pass B as `A&`
C. Program always exits with code 5
D. Program always exits with code 15

`virtual` Function Specifier[149]

- `virtual` enables dynamic dispatch for a function
  - ⇒ Allows function to be overridden in derived classes
  - A class with at least one virtual function is *polymorphic*
  - Overriding function can be annotated with `override` (see later)

- Calling a virtual function through pointer/reference of base class invokes behavior defined in derived class
- Suppressed when using qualified name lookup for function call

[149] https://en.cppreference.com/w/cpp/language/virtual

virtual: Example

```cpp
#include <print>
struct Base {
  virtual void foo() { std::println("Base::foo()"); }
};
struct Derived : Base {
  void foo() override { std::println("Derived::foo()"); }
};
int main() {
  Base b;
  Derived d;
  Base& br = b;
  Base& dr = d;
  d.foo(); // prints Derived::foo()
  dr.foo(); // prints Derived::foo()
  d.Base::foo(); // prints Base::foo()
  dr.Base::foo(); // prints Base::foo()
  br.foo(); // prints Base::foo()
}
```

Overriding Functions

A function overrides a virtual base class function if:

- Same name, cv-qualifiers, ref-qualifiers, and
- Same parameter type list (but not the return type)

If conditions met:

- Function is also virtual and can be overridden in derived classes
- Return type must be same or *covariant*
  - E.g., `virtual Base* m();` can be overriden by `Derived* m();`

Otherwise: function might **hide base class function**

Overriding Functions: Example

```
struct Base {
  virtual void bar();
  virtual void foo();
};
struct Derived : public Base {
  void bar(); // Overrides Base::bar()
  void foo(int baz); // Hides Base::foo()
};
int main() {
  Derived d;
  Base& b = d;
  d.foo(); // ERROR: lookup finds only Derived::foo(int)
  b.foo(); // invokes Base::foo();
}
```

override Specifier[150]

- override: specify that function actually overrides a virtual function
- Useful to avoid bugs where function in derived class hides base class function

```
struct Base {
  virtual void foo(int i);
  virtual void bar();
};

struct Derived : public Base {
  void foo(float i) override; // ERROR: no override, different parameter types
  void bar() const override; // ERROR: no override, different cv-qualifier
};
```

[150] https://en.cppreference.com/w/cpp/language/override

Final Overrider

- Final overrider: function that gets executed on virtual call
- Typically the overrider in the most derived class
- Can be more complex with multiple inheritance

Exception:

- During construction/destruction: behaves as if no more-derived classes exist
  - While constructing the base class, the derived class doesn't yet exist

⇝ Care must be taken when using virtual functions in these cases

final Specifier

- final functions: cannot be overridden
- final classes: cannot be inherited from

```
struct Base { virtual void foo() final; };
struct Derived : Base {
  void foo() override; // ERROR
}
struct Base final { virtual void foo(); };
struct Derived : Base { // ERROR
  void foo() override;
}
```

Destructors and Inheritance

△ Quiz: What is the output of this program?

```cpp
#include <memory>
#include <print>
struct A { ~A() { std::print("A"); } };
struct B : public A { ~B() { std::print("B"); } };
int main() {
  B b;
  std::unique_ptr<A> a = std::make_unique<B>();
}
```

A. Compile error: cannot assign `unique_ptr<B>` to `unique_ptr<A>`

B. `ABA`

C. `BAA`

D. `ABAB`

E. `BABA`

Destructors and Inheritance

- Derived objects can be deleted through pointer to base class
- **Undefined behavior** unless destructor is virtual

⇒ Destructor in base class should be public and virtual;
or: should be protected and non-virtual;
or: you know what you are doing

```
#include <memory>
#include <print>
struct A { virtual ~A() {} };
struct B : public A { };
int main() {
  A* a = new B();
  delete a; // OK
}
```

Abstract Classes[151]

- Class which cannot be instantiated, but used as a base class
- Any class with a *pure virtual* function is abstract
- Pure virtual function: virtual declaration ending with `= 0;`
- Pure virtual function can still be defined out-of-line

```
struct Base {
  virtual void foo() = 0; // pure virtual
};
struct Derived : Base {
  void foo() override;
};
int main() {
  Base b; // ERROR: Base is abstract
  Derived d; // OK
  Base& dr = d; // OK: pointers/references/smart pointers/etc. to abstract class
  dr.foo(); // calls Derived::foo()
}
```

[151] https://en.cppreference.com/w/cpp/language/abstract_class

Pure Virtual Destructor

- Destructor can be marked as pure virtual
- Useful when class shall be abstract, but no suitable functions exists
- Out-of-line definition *must* be provided

```
struct Base {
  virtual ~Base() = 0;
};
Base::~Base() {}
int main() {
  Base b; // ERROR: Base is abstract
}
```

Calling Pure Virtual Functions

△ Quiz: What is the problem with this code?

```
struct A {
  virtual ~A() { cleanup(); }
  virtual void cleanup() = 0;
};
struct B : A {
  void cleanup() override {}
};
int main() { B b; }
```

A. Compile error: cannot call pure virtual method in base class

B. Undefined behavior: calling pure virtual function in constructor/destructor

C. Semantic problem: `B::cleanup` doesn't get called, instead nothing happens

D. No problem: `B::cleanup()` gets called

Virtual Base Classes

- `virtual` base class: contained only once in the derived class, even if it occurs multiple times in the inheritance DAG
- Changes rules for unqualified name lookup
- Advice: try to avoid multiple inheritance

```
struct A {int a;};
struct B1 : virtual A {};
struct B2 : virtual A {};
struct C : B1, B2 {};
int getA(C& c) { return c.a; /* OK, only one a in C */ }
```

dynamic_cast[152]

- Convert pointers/references to classes in inheritance hierarchy
- Syntax: `dynamic_cast<`*new-type*`>(`*expression*`)`
  - *new-type* can be pointer or reference to class type

- Most common use case: checked/safe downcast
- Runtime check whether *new-type* is actually a base of the type of expression
- Failure: `nullptr` (pointers)/exception (references)
- Requires runtime type information (enabled by default)

- Other use cases: see reference

[152] https://en.cppreference.com/w/cpp/language/dynamic_cast

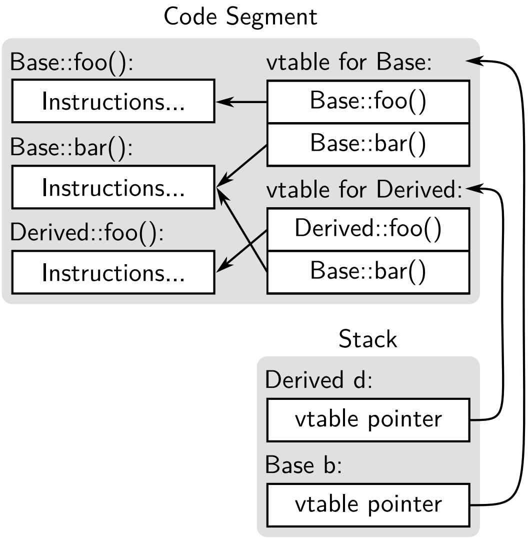
dynamic_cast: Example

```
struct A {
  virtual ~A() = default;
};
struct B : A {
  void foo() const;
};
struct C : A {
  void bar() const;
};
void baz(const A* aptr) {
  if (const B* bptr = dynamic_cast<const B*>(aptr)) {
    bptr->foo();
  } else if (const C* cptr = dynamic_cast<const C*>(aptr)) {
    cptr->bar();
  }
}
```

Implementation of Virtual Functions

- Vtable: table of function pointers to final overrider for every class
- Vtable pointer stored at beginning of every object
- Function invocation: load vtable, load fn pointer, do indirect call

```
struct Base {
  virtual void foo();
  virtual void bar();
};
struct Derived : Base {
  void foo() override;
};
int main() { Base b; Derived d; }
```



Implementation of dynamic_cast

- Vtable contains pointer to data structure that describes type
- Type checks tend to be rather expensive ⇒ noticable performance impact

- Alternative: type enum and static_cast

```
struct Base {
  enum class Type { Base, Derived, };
  Type type;

  Base() : type(Type::Base) {}
  Base(Type type) : type(type) {}

  virtual ~Base();
};
struct Derived : Base {
  Derived() : Base(Type::Derived) {}
};
```

```
void foo(Base* b) {
  switch (b->type) {
  case Base::Type:
    // use Base
    break;
  case Base::Derived: {
    auto* d = static_cast<Derived*>(b);
    // use Derived
    break;
  }
  }
}
```

Polymorphism: Recommendations

- If performance doesn't matter: whatever
- Generally avoid `dynamic_cast`, use type enum and static cast
  - Runtime type information (RTTI) is big, cast is much more expensive
- Avoid virtual function calls where performance matters
  - Indirection is very expensive, can be very noticeable when invoked frequently
  - When important it is often possible and recommendable to avoid these

Compile-Time Polymorphism

- How to avoid virtual function calls? Templates
- Curiously Recurring Template Pattern (CRTP):[153]
  base class takes derived class as template parameter

```
template <class Derived> struct Base {
  Derived* derived() { return static_cast<Derived*>(this); }
  int foo() { return derived()->bar(); }
  int bar() { return 12; }

protected:
  Base() = default; // prohibit creation of Base objects
};
struct MyImpl : public Base<MyImpl> {
  int bar() { return 42; }
};
int main() { return MyImpl().foo(); } // returns 42
```

[153] https://en.cppreference.com/w/cpp/language/crtp

Deducing `this`

- C++23 introduces explicitly object member functions
- Type of `this` specified explicitly; `this` unusable in function body

```
struct Base {
  int foo(this auto&& self) { return self.bar(); }
  int bar() { return 12; }

protected:
  Base() = default; // prohibit creation of Base objects
};
struct MyImpl : public Base {
  int bar() { return 42; }
};
int main() { return MyImpl().foo(); } // returns 42
```

CRTP Compared to Virtual Functions

+ No runtime overhead of virtual function calls
+ Base class can call into functions of derived class

– Definitions (generally) need to go into header files
– Less flexibility
– Cannot have container of polymorphic objects
  - I.e., no `std::vector<std::unique_ptr<Base>>`

Mixins

- Mixin: compose functionality from multiple classes

```
template <class D> struct Greeter {
  D* derived() { return static_cast<D*>(this); }
  void greet() {
    std::println("Hello,␣I’m␣{}!", derived()->name());
  }
};
struct Person : Greeter<Person> {
  std::string_view n;
  std::string_view name() const { return n; }
};
int main() {
  Person p{.n = "foo"};
  p.greet();
}
```

Inheritance – Summary

- C++ supports very flexible inheritance of classes
- Classes can have zero, one, or more base classes
- Base classes can be inherited `public`/`protected`/`private`
- By default, inheritance is non-polymorphic
- `virtual` functions enable overriding and dynamic polymorphism
- Polymorphism needs care for implementing constructors/destructors
- `dynamic_cast` allows dynamic type checking and casting
- Templates allow implementing static polymorphism at compile-time
- Dynamic polymorphism often has considerable runtime overhead

Inheritance – Questions

- In which order are constructors/destructors of base classes executed?
- How does inheritance change the object representation of classes?
- What is an advantage of non-polymorphic inheritance?
- What is a disadvantage of non-polymorphic inheritance?
- Why is using the `override` specifier highly recommendable?
- How are virtual functions conceptually implemented?
- Why should destructors of base classes often be virtual?
- How to use CRTP for compile-time polymorphism?