Inheritance

Object-Oriented Programming

Object-oriented programming is based on three fundamental concepts

- Data abstraction
  - Implemented by classes in C++
  - Covered previously
- Inheritance
  - Implemented by class derivation in C++
  - Derived Classes inherit the members of its base class(es)
  - Covered in this lecture
- Dynamic Binding (Polymorphism)
  - Implemented by virtual functions in C++
  - Programs need not care about the specific types of objects in an inheritance hierarchy
  - Covered in this lecture

Derived Classes (1)

Any class type may be derived from one or more *base classes*

- Possible for both `class` and `struct`
- Base classes may in turn be derived from their own base classes
- Classes form an *inheritance hierarchy*

High-level Syntax

```
class class-name : base-specifier-list {
    member-specification
};
```

```
struct class-name : base-specifier-list {
    member-specification
};
```

Derived Classes (2)

The *`base-specifier-list`* contains a comma-separated list of one or more *`base-specifier`*s with the following syntax

```
access-specifier virtual-specifier base-class-name
```

Explanation

- *`access-specifier`* controls the *inheritance mode* (more details soon)
- *`access-specifier`* is optional; if present it can be one of the keywords `private`, `protected` or `public`
- *`base-class-name`* is mandatory, it specifies the name of the class from which to derive
- *`virtual-specifier`* is optional; if present it must be the keyword `virtual` (only used for multiple inheritance)

Derived Classes (3)

Examples

```
class Base {
    int a;
};

class Derived0 : Base {
    int b;
};

class Derived1 : private Base {
    int c;
};

class Derived2 : public virtual Base, private Derived1 {
    int d;
};
```

Constructors and Initialization (1)

Constructors of derived classes account for the inheritance

1. The direct non-virtual base classes are initialized in left-to-right order as they appear in the `base-specifier-list`
2. The non-static data members are initialized in the order of declaration in the class definition
3. The body of the constructor is executed

The initialization order is independent of any order in the member initializer list

Base classes are default-initialized unless specified otherwise

- Another constructor can explicitly be invoked using the delegating constructor syntax

Constructors and Initialization (2)

Consider the class definitions

foo.hpp

```
struct Base {
    int a;

    Base();
    explicit Base(int a);
};

struct Derived : Base  {
    int b;

    Derived();
    Derived(int a, int b);
};
```

foo.cpp

```
#include "foo.hpp"
#include <iostream>

using namespace std;

Base::Base()
    : a(42) {
    cout << "Base::Base()" << endl;
}

Base::Base(int a)
    : a(a) {
    cout << "Base::Base(int)" << endl;
}

Derived::Derived() {
    : b(42) {
    cout << "Derived::Derived()" << endl;
}

Derived::Derived(int a, int b)
    : Base(a), b(b) {
    cout << "Derived::Derived(int, int)" << endl;
}
```

Constructors and Initialization (3)

Using the above class definitions, consider the following program

main.cpp

```
#include "foo.hpp"

int main() {
    Derived derived0;
    Derived derived1(123, 456);
}
```

Then the output of this program would be

```
$ ./foo
Base::Base()
Derived::Derived()
Base::Base(int)
Derived::Derived(int, int)
```

Destructors (1)

Similarly to constructors, destructors of derived classes account for the inheritance

1. The body of the destructor is executed
2. The destructors of all non-static members are called in reverse order of declaration
3. The destructors of all direct non-virtual base classes are called in reverse order of construction

The order in which the base class destructors are called is deterministic

- It depends on the order of construction, which in turn only depends on the order of base classes in the *base-specifier-list*

Destructors (2)

Consider the class definitions

foo.hpp

```cpp
struct Base0 {
    ~Base0();
};

struct Base1 {
    ~Base1();
};

struct Derived : Base0, Base1 {
    ~Derived();
};
```

foo.cpp

```cpp
#include "foo.hpp"
#include <iostream>

using namespace std;

Base0::~Base0() {
    cout << "Base0::~Base0()" << endl;
}

Base1::~Base1() {
    cout << "Base1::~Base1()" << endl;
}

Derived::~Derived() {
    cout << "Derived::~Derived()" << endl;
}
```

Destructors (3)

Using the above class definitions, consider the program

main.cpp

```
#include "foo.hpp"

int main() {
    Derived derived;
}
```

Then the output of this program would be

```
$ ./foo
Derived::~Derived()
Base1::~Base1()
Base0::~Base0()
```

Unqualified Name Lookup (1)

It is allowed (although discouraged) to use a name multiple times in an inheritance hierarchy

- Affects unqualified name lookups (lookups without the use of the scope resolution operator `::`)
- A deterministic algorithm decides which alternative matches an unqualified name lookup
- Rule of thumb: Declarations in the derived classes "hide" declarations in the base classes

Multiple inheritance can lead to additional problems even without reusing a name

- In a diamond-shaped inheritance hierarchy, members of the root class appear twice in the most derived class
- Can be solved with *virtual* inheritance
- Should still be avoided whenever possible

Unqualified Name Lookup (2)

Single inheritance example

```
struct A {
    void a();
};

struct B : A {
    void a();
    void b() {
        a();  // calls B::a()
    }
};

struct C : B {
    void c() {
        a();  // calls B::a()
    }
};
```

Unqualified Name Lookup (3)

Diamond inheritance example

```
struct X {
    void x();
};

struct B1 : X { };
struct B2 : X { };

struct D : B1, B2 {
    void d() {
        x();  // ERROR: x is present in B1 and B2
    }
};
```

Qualified Name Lookup

Qualified name lookup can be used to explicitly resolve ambiguities

- Similar to qualified namespace lookups, a class name can appear to the left of the scope resolution operator `::`

```
struct A {
    void a();
};

struct B : A {
    void a();
};

int main() {
    B b;
    b.a();     // calls B::a()
    b.A::a();  // calls A::a()
}
```

Object Representation

The object representation of derived class objects accounts for inheritance

- The base class object is stored as a *subobject* in the derived class object
- Thus, derived classes may still be trivially constructible, copyable, or destructible

foo.cpp

```cpp
struct A {
    int a = 42;
    int b = 123;
};

struct B : A {
    int c = 456;
};

int main() {
    B b;
}
```

foo.o

```asm
main:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $42, -12(%rbp)
    movl    $123, -8(%rbp)
    movl    $456, -4(%rbp)
    movl    $0, %eax
    popq    %rbp
    ret
```

Polymorphic Inheritance

By default, inheritance in C++ is non-polymorphic

- Member definitions in a derived class can hide definitions in the base class
- For example, it matters if we call a function through a pointer to a base object or a pointer to a derived object

```
#include <iostream>

struct Base {
    void foo() { std::cout << "Base::foo()" << std::endl; }
};

struct Derived : Base {
    void foo() { std::cout << "Derived::foo()" << std::endl; }
};

int main() {
    Derived d;
    Base& b = d;

    d.foo();  // prints Derived::foo()
    b.foo();  // prints Base::foo()
}
```

The `virtual` Function Specifier (1)

Used to mark a non-static member function as *virtual*

- Enables dynamic dispatch for this function
- Allows the function to be overriden in derived classes
- A class with at least one virtual function is *polymorphic*

The overridden behavior of the function is preserved even when no compile-time type information is available

- A call to an overridden virtual function through a pointer or reference to a base object will invoke the behavior defined in the derived class
- This behavior is suppressed when qualified name lookup is used for the function call

The virtual Function Specifier (2)

Example

```
#include <iostream>

struct Base {
    virtual void foo() { std::cout << "Base::foo()" << std::endl; }
};
struct Derived : Base {
    void foo() { std::cout << "Derived::foo()" << std::endl; }
};
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()
}
```

Conditions for Overriding Functions (1)

A function overrides a virtual base class function if

- The function name is the same
- The parameter type list (but not the return type) is the same
- The cv-qualifiers of the function are the same
- The ref-qualifiers of the function are the same

If these conditions are met, the function overrides the virtual base class function

- The derived function is also virtual and can be overridden by further-derived classes
- The base class function does not need to be visible
- The return type must be the same or *covariant*

If these conditions are not met, the function may hide the virtual base class function

Conditions for Overriding Functions (2)

Example

```
struct Base {
    private:
    virtual void bar();

    public:
    virtual void foo();
};

struct Derived : 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();
}
```

The Final Overrider (1)

Every virtual function has a *final overrider*

- The final overrider is executed when a virtual function call is made
- A virtual member function is the final overrider unless a derived class declares a function that overrides it

A derived class can also inherit a function that overrides a virtual base class function through multiple inheritance

- There must only be one final overrider at all times
- Multiple inheritance should be avoided anyway

The Final Overrider (2)

Example

```
struct A {
    virtual void foo();
    virtual void bar();
    virtual void baz();
};
struct B : A {
    void foo();
    void bar();
};
struct C : B {
    void foo();
};
int main() {
    C c;
    A& cr = c;

    cr.foo();  // invokes C::foo()
    cr.bar();  // invokes B::bar()
    cr.baz();  // invokes A::baz()
}
```

The Final Overrider (3)

The final overrider depends on the actual type of an object

```
struct A {
    virtual void foo();
    virtual void bar();
    virtual void baz();
};
struct B : A {
    void foo();
    void bar();
};
struct C : B {
    void foo();
};
int main() {
    B b;
    A& br = b;

    br.foo();  // invokes B::foo()
    br.bar();  // invokes B::bar()
    br.baz();  // invokes A::baz()
}
```

Covariant Return Types (1)

The overriding and base class functions can have *covariant* return types

- Both types must be single-level pointers or references to classes
- The referenced/pointed-to class in the base class function must be a direct or indirect base class of the referenced/pointed-to class in the derived class function
- The return type in the derived class function must be at most as cv-qualified as the return type in the base class function
- Most of the time, the referenced/pointed-to class in the derived class function is the derived class itself

Covariant Return Types (2)

Example

```
struct Base {
    virtual Base* foo();
    virtual Base* bar();
};

struct Derived : Base {
    Derived* foo();  // Overrides Base::foo()
    int bar();       // ERROR: Overrides Base::bar() but has
                     // non-covariant return type
};
```

Construction and Destruction

Virtual functions have to be used carefully during construction and destruction

- During construction and destruction, a class behaves as if no more-derived classes exist
- I.e., virtual function calls during construction and destruction call the final overrider in the constructor's or destructor's class

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

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

int main() {
    Derived d;  // On construction, Base::foo() is called
}
```

Virtual Destructors

Derived objects can be deleted through a pointer to the base class

- Undefined behavior unless the destructor in the base class is virtual
- The destructor in a base class should either be protected and non-virtual or public and virtual

```
#include <memory>

struct Base {
    virtual ~Base() { };
};

struct Derived : Base { };

int main() {
    Base* b = new Derived();
    delete b;  // OK
}
```

The override Specifier

The `override` specifier should be used to prevent bugs

- The `override` specifier can appear directly after the declarator in a member function declaration or inline member function definition
- Ensures that the member function is virtual and overrides a base class method
- Useful to avoid bugs where a function in a derived class actually hides a base class function instead of overriding it

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

struct Derived : Base {
    void foo(float i) override;  // ERROR
    void bar() const override;   // ERROR
};
```

The `final` Specifier (1)

The `final` specifier can be used to prevent overriding a function

- The `final` specifier can appear directly after the declarator in a member function declaration or inline member function definition

```
struct Base {
    virtual void foo() final;
};

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

The final Specifier (2)

The final specifier can be used to prevent inheritance from a class

- The final specifier can appear in a class definition, immediately after the class name

```
struct Base final {
    virtual void foo();
};

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

Abstract Classes (1)

C++ allows abstract classes which cannot be instantiated, but used as a base class

- Any class which declares or inherits at least one *pure virtual* function is an abstract class
- A pure virtual member function declaration contains the sequence `= 0` after the declarator and `override`/`final` specifiers
- Pointers and references to an abstract class can be declared

A definition can still be provided for a pure virtual function

- Derived classes can call this function using qualified name lookup
- The pure specifier `= 0` cannot appear in a member function definition (i.e. the definition can not be provided inline)

Making a virtual function call to a pure virtual function in the constructor or destructor of an abstract class is **undefined behavior**

Abstract Classes (2)

Example

```
struct Base {
    virtual void foo() = 0;
};

struct Derived : Base {
    void foo() override;
};

int main() {
    Base b;          // ERROR
    Derived d;
    Base& dr = d;
    dr.foo();        // calls Derived::foo()
}
```

Abstract Classes (3)

A definition may be provided for a pure virtual function

```
struct Base {
    virtual void foo() = 0;
};

void Base::foo() { /* do something */ }

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

Abstract Classes (4)

The destructor may also be marked as pure virtual

- Useful when a class needs to be abstract, but has no suitable functions that could be declared pure virtual
- In this case a definition *must* be provided

```
struct Base {
    virtual ~Base() = 0;
};

Base::~Base() { }

int main() {
    Base b;  // ERROR
}
```

Abstract Classes (5)

Abstract classes cannot be instantiated

- Programs have to refer to abstract classes through pointers or references
- Smart pointers (owning), references (non-owning), or raw pointers (if `nullptr` is possible)

```
#include <memory>

struct Base {
    virtual ~Base();
    virtual void foo() = 0;
};

struct Derived : Base { void foo() override; };

void bar(const Base& b) { b.foo(); }

int main() {
    std::unique_ptr<Base> b = std::make_unique<Derived>();
    b->foo();  // calls Derived::foo()

    bar(*b);   // calls Derived::foo() within bar
}  // destroys b, undefined behavior unless ~Base() is virtual
```

dynamic_cast (1)

Converts pointers and references to classes in an inheritance hierarchy

- Syntax: `dynamic_cast < new_type > ( expression )`
- *new_type* may be a pointer or reference to a class type
- *expression* must be an lvalue expression of reference type if *new_type* is a reference type, and an rvalue expression of pointer type otherwise

Most common use case: Safe downcasts in an inheritance hierarchy

- Involves a runtime check whether *new_type* is a base of the actual polymorphic type of *expression*
- If the check fails, returns `nullptr` for pointer types, and throws an exception for reference types
- Requires runtime type information which incurs some overhead

For other use cases: See the reference documentation

dynamic_cast (2)

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();
    }
}
```

dynamic_cast (3)

dynamic_cast has a non-trivial performance overhead

- Notable impact if many casts have to be performed
- Alternative: Use a type enum in conjunction with 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) { }
};
```

dynamic_cast (4)

Example (continued)

```
void bar(const Base* basePtr) {
    switch (basePtr->type) {
    case Base::Type::Base:
        /* do something with Base */
        break;
    case Base::Type::Derived:
        const Derived* derivedPtr
            = static_cast<const Derived*>(basePtr);

        /* do something with Derived */

        break;
    }
}
```

Vtables (1)

Polymorphism does not come for free

- Dynamic dispatch has to be implemented somehow
- The C++ standard does not prescribe a specific implementation
- Compilers typically use *vtables* to resolve virtual function calls

Vtables setup and use

- One vtable is constructed per class with virtual functions
- The vtable contains the addresses of the virtual functions of that class
- Objects of classes with virtual functions contain an additional pointer to the base of the vtable
- When a virtual function is invoked, the pointer to the vtable is followed and the function that should be executed is resolved

Vtables (2)

Example

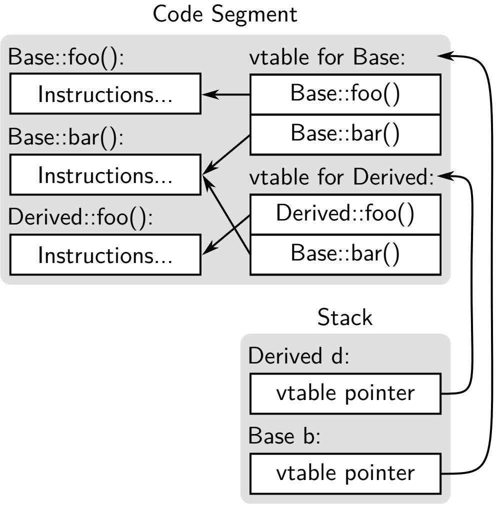
```
struct Base {
    virtual void foo();
    virtual void bar();
};

struct Derived : Base {
    void foo() override;
};

int main() {
    Base b;
    Derived d;

    Base& br = b;
    Base& dr = d;

    br.foo();
    dr.foo();
}
```



Performance Implications

Virtual function calls incur an additional indirection

- The pointer to the vtable is followed
- The pointer to the actual function is followed
- Each step may incur a cache miss
- Can be very notable when invoking a virtual function millions of times

Polymorphic objects have larger size

- Each object of a polymorphic class needs to store a pointer to the vtable
- In our example, both `Base` and `Derived` occupy 8 bytes of memory despite having no data members

Inheritance Modes

Recall the definition of a *base-specifier*

access-specifier virtual-specifier base-class-name

The *access-specifier* specifies the *inheritance mode*

- The inheritance mode controls the access mode of base class members in the derived class
- If no *access-specifier* is given, derived classes defined with `struct` have `public` inheritance mode by default
- If no *access-specifier* is given, derived classes defined with `class` have `private` inheritance mode by default

Public Inheritance (1)

Semantics

- Public base class members are usable as public members of the derived class
- Protected base class members are usable as protected members of the derived class

Models the subtyping (IS-A) relationship of object-oriented programming

- Pointers and references to a derived object should be usable wherever a pointer to the a base object is expected
- A derived class must maintain the class invariants of its base classes
- A derived class must not strengthen the preconditions of any member function it overrides
- A derived class must not weaken the postconditions of any member function it overrides

Public Inheritance (2)

Example

```
class A {
    protected:
    int a;

    public:
    int b;
};

class B : public A {
    public:
    void foo() {
        return a + 42;  // OK: a is usable as protected member of B
    }
};

int main() {
    B b;
    b.b = 42;  // OK: b is usable as public member of B
    b.a = 42;  // ERROR: a is not visible
}
```

Private Inheritance (1)

Semantics

- Public base class members are usable as private members of the derived class
- Protected base class members are usable as private members of the derived class

Some specialized use cases

- Policy-based design using templates (more details later)
- Mixins
- Model composition if some requirements are met
  - The base object needs to be constructed or destructed before or after some object in the derived object
  - The derived class needs access to protected members of the base class
  - The derived class needs to override virtual methods in the base class

Private Inheritance (2)

Example

```
class A {
    protected:
    A(int);  // Constructor is protected for some reason
};

class C : private A {
    public:
    C() : A(42) { }

    const A& getA() {  // Act as if we have a member of type A
        return *this;
    }
};
```

Protected Inheritance (1)

Semantics

- Public base class members are usable as protected members of the derived class
- Protected base class members are usable as protected members of the derived class
- Within the derived class and all further-derived classes, pointers and references to a derived object may be used where a pointer or reference to the base object is expected

Models “controlled polymorphism”

- Mainly used for the same purposes as private inheritance, where inheritance should be shared with subclasses
- Rarely seen in practice

Protected Inheritance (2)

Example

```
class A {
    protected:
    int a;

    public:
    int b;
};

class B : protected A {
    public:
    void foo() {
        return a + 42;  // OK: a is usable as protected member of B
    }
};

int main() {
    B b;
    b.b = 42;  // ERROR: b is not visible
    b.a = 42;  // ERROR: a is not visible
}
```

Multiple Inheritance

C++ supports multiple inheritance

- Rarely required
- Easy to produce convoluted code
- Leads to implementation issues (e.g. diamond-inheritance)

There are C++ language features to address such issues

- You will likely never need multiple inheritance during this lecture
- For details: Check the reference documentation
- Multiple inheritance should be avoided whenever possible

Exceptions in C++

C++ supports exceptions with similar semantics as other languages

- Exceptions transfer control and information up the call stack
- Can be thrown by `throw`-expressions, `dynamic_cast`, `new`-expressions and some standard library functions

While transferring control up the call stack, C++ performs *stack unwinding*

- Properly cleans up all objects with automatic storage duration
- Ensures correct behavior e.g. of RAII classes

Exceptions do not have to be handled

- Can be handled in `try`-`catch` blocks
- Unhandled exceptions lead to termination of the program though
- Errors during exception handling also lead to termination of the program

Throwing Exceptions

Objects of any complete type may be thrown as exception objects

- Usually exception objects should derive directly or indirectly from `std::exception`, and contain information about the error condition
- Syntax: `throw` *`expression`*
- Copy-initializes the exception object from *`expression`* and throws it

```
#include <exception>

void foo(unsigned i) {
    if (i == 42)
        throw 42;

    throw std::exception();
}
```

Handling Exceptions

Exceptions are handled in try-catch blocks

- Exceptions that occur while executing the try-block can be handled in the catch-blocks
- The parameter type of the catch-block determines which type of exception causes the block to be entered

```
#include <exception>

void bar() {
    try {
        foo(42);
    } catch (int i) {
        /* handle exception */
    } catch (const std::exception& e) {
        /* handle exception */
    }
}
```

Usage Guidelines

Exceptions should only be used in rare cases

- Main legitimate use case: Failure to (re)establish a class invariant (e.g. failure to acquire a resource in an RAII constructor)
- Functions should **not** throw exceptions when preconditions are not met – use assertions instead
- Exceptions should **not** be used for control flow

Some functions must not throw exceptions

- Destructors
- Move constructors and assignment operators
- See reference documentation for details

Generally, exceptions should be avoided where possible