Inheritance

C++ terminology: a superclass is a base class, a subclass is a derived class. Inheritance syntax:

```
class Vehicle { ... };
class Car : public Vehicle { ... };
```

The type rules are the same as in Java: a pointer (or reference) of a base class type may point (refer) to objects of its class and all derived classes. Differences between inheritance in Java and in C++:

- C++ by default uses static binding of method calls.
- C++ has `public`, `private`, and `protected` inheritance.
- C++ has multiple inheritance.

Dynamic Binding, Virtual Functions

In Java, the type of the *object* determines which implementation of a function that is called. This is called dynamic binding. To achieve this in C++ you must specify the function as `virtual`.

```
class A {
public:
    virtual void f() const { ... }
};

class B : public A {
public:
    virtual void f() const override { ... }
};

A* pa = new A; pa->f(); // calls A::f
pa = new B;    pa->f(); // calls B::f
```

Repeated `virtual`, `override`

- It is not necessary to repeat `virtual` on the overriding functions in derived classes.
- `override` C++11 checks that there really is a virtual function to override. This catches many errors (`f` and `g` in `B` do not override `f` and `g` in `A`, instead new functions are defined):

```
class A {
public:
    virtual void f() const;
    virtual void g(int x);
};

class B : public A {
public:
    virtual void f(); // not const
    virtual void g(double x); // different parameter type
};
```

Virtual Functions and References

Virtual functions also work as expected when called through a reference. Example (same classes as on slide 176):

```
void check(const A& r) { // r may refer to an A object
    r.f();               // or a B object
}

void g() {
    A a;
    B b;
    check(a); // will call A::f
    check(b); // will call B::f
}
```

Don't forget the & on the parameter — if you do, the parameter is called by value and the object r in check will always be an A object ("slicing").

Initializing Objects of Derived Classes

The constructor in a derived class must explicitly call the constructor in the base class (done with `super(...)` in Java). The call must be written in the constructor initialization list.

```
class Shape {
public:
    Shape(int ix, int iy) : x(ix), y(iy) {}
    ...
};

class Square : public Shape {
public:
    Square(int ix, int iy, int is) : Shape(ix, iy), s(is) {}
    ...
};
```

Pure Virtual (Abstract) Functions

An operation may be specified as abstract, "pure virtual" in C++, by adding `= 0` after the function signature:

```
class Shape {
public:
    virtual void draw(const Window& w) const = 0; // abstract
    ...
};

class Square : public Shape {
public:
    virtual void draw(const Window& w) const override { ... }
    ...
};
```

You cannot create an object of a class that contains pure virtual functions.

Interfaces

Java has interfaces, classes with only abstract operations and no attributes. C++ has no separate concept for interfaces; instead you use classes with only pure virtual functions and public inheritance (multiple inheritance, if a class implements more than one interface).

```
class Drawable { // interface
public:
    virtual ~Drawable() = default;
    virtual void draw(const Window& w) const = 0;
};

class Square : public Drawable { // "implements Drawable"
public:
    virtual void draw(const Window& w) const override { ... }
    ...
};
```

Type Rules, Runtime Type Information (RTTI)

A pointer (or reference) of a base class type may point (refer) to objects of its class and all derived classes.

```
Shape* psh = new Shape(1, 2);
Square* psq = new Square(2, 3, 5);

psh = psq; // always correct, all squares are shapes (upcast)
psq = psh; // not allowed, needs an explicit cast (downcast)
```

In Java, every object carries information about its type (used in method dispatch and in tests with `instanceof`). This type information is called runtime type identification, RTTI.

In C++, only objects of classes with at least one virtual function have RTTI. This is because it entails some memory overhead (one extra pointer in each object).

Downcasting Pointers and References

Downcasting is performed with `dynamic_cast`:

```
dynamic_cast<newtype*>(expression)
```

If the cast succeeds, the value is a pointer to the object. If it fails, i.e., if the expression doesn't point to an object of type `newtype`, the value is null.

```
Shape* psh = list.firstShape();
if (Square* psq = dynamic_cast<Square*>(psh)) {
    cout << "Side length: " << psq.getSide() << endl;
}
```

`dynamic_cast` can also be used with references. Since there are no "null references" the cast throws `std::bad_cast` if it fails, instead of returning a null value.

Copying Base Classes and Derived Classes

The rules for upcasting and downcasting concern pointers and references.
There are similar rules for copying objects:

```
Shape sh(1, 2);
Square sq(2, 3, 5);

sh = sq; // allowed, but Square members are not copied (slicing)
sq = sh; // not allowed, would leave the Square members undefined
```

Note that slicing can occur if a function argument is passed by value:

```
void f(const Shape s) { // probably meant Shape&
    ...                 // here s is a Shape object,
    s.draw();           // Shape::draw is always called
}
```

Public and Private Inheritance

This is "normal" inheritance:

```
class A { ... };
class B : public A { ... };
```

The relationship B "is-an" A holds. Anywhere that an A object is expected, a B object can be supplied. In C++, there also is private (and protected) inheritance:

```
class A { ... };
class B : private A { ... };
```

B inherits everything from A but the public interface of A becomes private in B. A B object cannot be used in place of an A object. The relationship is that B "is-implemented-using" A.

Private Inheritance, Example

We wish to implement a class describing a stack of integers and decide to use the standard `vector` class to hold the numbers (in practice, we would have used the standard `stack` class).

We have several alternatives. This one isn't a serious candidate:

```
class Stack : public vector<int> { // public inheritance
public:
    void push(int x) { push_back(x); }
    int pop() { int x = back(); pop_back(); return x; }
};
```

Not good: we get the functionality we need, but also much more. The class also inherits, among others, the `insert` function which can insert an element at any location in the vector.

Private Inheritance Example, cont'd

```
class Stack { // composition, like in Java
public:
    void push(int x) { v.push_back(x); }
    int pop() { int x = v.back(); v.pop_back(); return x; }
private:
    vector<int> v;
};
```

Good: only push and pop are visible. But note that the operations must delegate the work to the vector object, which means some overhead.

```
class Stack : private vector<int> { // private inheritance
public:
    void push(int x) { push_back(x); }
    int pop() { int x = back(); pop_back(); return x; }
};
```

Good: only push and pop are visible, no overhead.

Exposing Individual Members

We wish to add a `bool empty()` member to the `Stack` class. It can be done like this:

```
class Stack : private vector<int> {
public:
    bool empty() const { return vector<int>::empty(); }
    ...
};
```

But it's better to expose the member `empty()` from `vector<int>`:

```
class Stack : private vector<int> {
public:
    using vector<int>::empty;
    ...
};
```

Copy Control

If a base class defines a copy constructor and an assignment operator, they must be called from the derived classes:

```
class A {
public:
    A(const A& a) { ... }
    A& operator=(const A& rhs) { ... }
};

class B : public A {
public:
    B(const B& b) : A(b) { ... }
    B& operator=(const B& rhs) {
        if (&rhs == this) {
            return *this;
        }
        A::operator=(rhs);
        ...
    }
};
```

Destructing Derived Objects

Suppose that classes `A` and `B` manage separate resources. Then, each class needs a destructor:

```
class A {
public:
    ~A();
    ...
};

class B : public A {
public:
    ~B();
};
```

When a `B` object is destroyed and the destructor `~B()` is called, the destructor for the base class, `~A()`, is automatically called. This continues up to the root of the inheritance tree.

Virtual Destructors

There is a problem with destructors in derived objects. Consider the following statements:

```
A* pa = new B;
...
delete pa;
```

The wrong destructor, `~A()`, will be called when `pa` is deleted, since also destructors use static binding by default. To correct this error, you must make the destructor `virtual`:

```
virtual ~A();
```

Most base classes should have virtual destructors.

Multiple Inheritance

The relationship "is-a" can hold between a class and more than one other class. A panda is a bear and an endangered species, a teaching assistant is a student and a teacher, an amphibious vehicle is a land vehicle and a water vehicle, ...

In C++, a class may be derived from several base classes:

```
class TeachingAssistant : public Student, public Teacher {
    ...
};
```

Think carefully before you use multiple inheritance. There are problems with data duplication that must be considered. Multiple inheritance of interfaces (classes with only pure virtual functions) doesn't give any problems.

Conversions and Member Access

Since public inheritance is used, the normal rules for conversion to a base class apply:

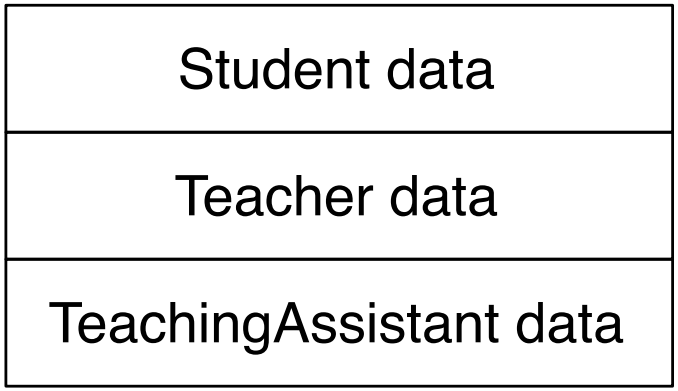
```
void f(const Student& s) { ... }
void g(const Teacher& t) { ... }

TeachingAssistant ta1;
f(ta1); // ok, ta1 is a student
g(ta1); // ok, ta1 is a teacher
```

A derived class sees the union of the members of the base classes. Name conflicts must be explicitly resolved.

```
cout << ta1.getProgram() << endl; // from Student
cout << ta1.getSalary() << endl;  // from Teacher
ta1.Student::print(); // if both Student and Teacher
ta1.Teacher::print(); // have a print() member
```

Memory Layout

A `TeachingAssistant` object looks like this in memory:



There is an obvious problem: what about duplicated data? If, for example, both students and teachers have names, a teaching assistant will have two names. This can be avoided using *virtual* inheritance.

Replicated Base Classes

We start by factoring out the common data (name, address, ...) and place it in a separate class `Person`:

```
class Person { ... };
class Student : public Person { ... };
class Teacher : public Person { ... };
class TeachingAssistant : public Student, public Teacher { ... };
```

However, this doesn't solve the problem. A `TeachingAssistant` object will contain all `Student` data, which contains `Person` data, and `Teacher` data, which also contains `Person` data.

Virtual Inheritance

The duplication of data can be avoided by specifying the inheritance of the common base class as `virtual`:

```
class Person { ... };
class Student : public virtual Person { ... };
class Teacher : public virtual Person { ... };
class TeachingAssistant : public Student, public Teacher { ... };
```

When virtual inheritance is used, data from a common base class is only stored once in an object.

We have only scratched on the surface of multiple inheritance. There is much more to consider: initialization order, destruction order, virtual functions, conversions to virtual base classes, ...