Object Oriented Software Engineering (COMP2003)

Lecture 2: References and Polymorphism

Updated: 12$^{\text{th}}$ March, 2018

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

Reference Variables

- You should understand the basic idea of a "reference variable":
  - It *refers* (or *points*) to a memory location where the value can be found.
  - Essentially a "safe" pointer (for those familiar with C).

Java – Object variables (e.g. **`String`**s, **`List`**s, etc.) are references, while primitives (`int`s, `double`s, etc.) are not.

Python – *All* variables are reference variables.
- Integers and real numbers are actually objects (but in practice, this doesn't affect the way you use them).

C++ – Has C-style pointers, value types, and other special references.
- C-style pointers are the closest to Java/Python.
- We'll discuss the other ones later.
- There are good reasons to use each in different situations.

References and Polymorphism

- References make possible polymorphism – a key OO principle.
  - We can write generalised code that *doesn't need to know* exactly what datatype it's using.
  - Can improve re-use.
  - Can reduce coupling – so each part of our code doesn't assume too much about the other parts.
- Polymorphism *cannot* work with value/primitive types.
  - Without a reference, you're forced to know the exact datatype.
  - Because you have to directly provide the memory for it!
- Whoever allocates the memory knows the datatype.

Reference and Object Types

- There are two aspects to OO datatypes:
  - The object type – what you *actually* have, at runtime.
  - The reference type – what you "*think*" you have, at compile time.
- The reference type is what you declare in source code:

```
private Person p;                              // Java
public Person getPerson() {...}
public void setPerson(Person newP) {...}
```

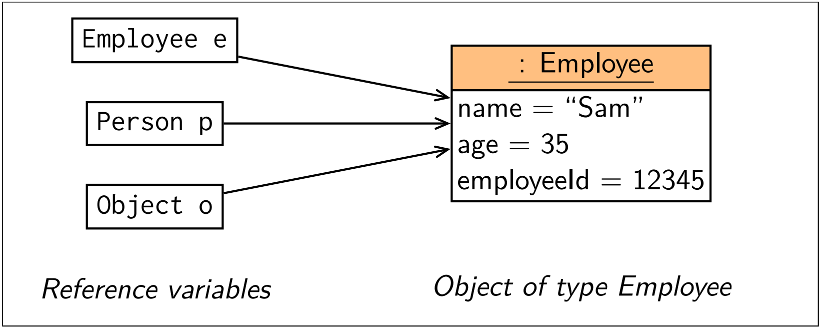
  - **Person** is the reference type in each case.
  - Applies to Java and C++, but not really to Python.
- The object type comes from the constructor call:

```
new Employee("Sam", 35, 12345);                // Java
```

- The reference type must be the same, or more "general", than the object type.
  - e.g. a **Person** field can refer to an **Employee** object.

Reference and Object Types

```
Employee e = new Employee("Sam", 35, 12345);
Person p = e;
Object o = e;
```



- The reference can be the same as the object type, OR
- The reference can be more “general”.

Upcasting

- An upcast is a conversion from a subclass reference to a superclass one; e.g.

```
Employee emp = new Employee(...);

Person p = emp; // Upcast from Employee to Person
```

  - The new object is definitely an **Employee**.
  - But **p** can still refer to it.
- We're not changing the object, just the reference type.
- We're choosing to "know less" about the object.
  - *From our point of view*, p just refers to a Person.

Upcasting Situations (1)

Upcasting happens in various situations; e.g.:

- Where method parameters are more general:

```
public void setPerson(Person p)
{
    // p may actually be an Employee, but within
    // this method we don't care.
    ...
}
```

```
Employee e = new Employee(...);
setPerson(e); // <-- Upcast Employee to Person
```

- e and p refer to the same object with different reference types.
- setPerson() doesn't know anything about Employee.
  - Reduced degree of coupling between method and caller.

Upcasting Situations (2)

- Where method return types are more specific:

```
public Employee createEmployee()
{
    return new Employee(...);
}
...
Person p = createEmployee();  // Upcast
```

  - The returned Employee reference is upcasted and stored in p.

- Where a return type is more general than the return value:

```
public Person createPerson()
{
    return new Employee(...);  // Upcast
}
```

  - The return expression produces an Employee reference. This is upcasted and then returned as a Person reference.

Downcasting

- *Downcasting* means converting a general reference back to a more specific one.

```
Employee emp = (Employee)person;
```

- Only possible if you know (somehow) what the more-specific type should be.
  - e.g. you've checked it using Java's `instanceof` operator.
  - The above code will throw an exception if the `person` variable does not actually refer to an `Employee` object.
- Downcasting is usually a terrible idea! (With a few rare exceptions.)
  - It can only be done to a previously upcast reference.
  - It just undoes the de-coupling advantages of upcasting.
  - Rather than upcast-then-downcast, it's simpler to just stick with the same reference type (and avoid casting altogether).

Downcasting in Java

- Java's **equals()** method takes an `Object` parameter.
  - Rare case...
  - We *must* downcast it to get the right reference type.
- But we must use `instanceof` to verify the type beforehand.

```
public boolean equals(Object obj)
{
    boolean eq = false;
    if(obj instanceof Person)
    {
        Person p = (Person)obj; // Downcast Object to
                                //           Person
        eq = name.equals(p.name) && age == p.age;
    }
    return eq;
}
```

Downcasting in C++

- C++ has "dynamic_cast".
- This performs a downcast, or returns NULL if it's not allowed.
- Not many legitimate reasons to use it though.

```
Person* p = new Employee(...);
...

Employee* e = dynamic_cast<Employee*>(p);
if(e != NULL)
{
    // Access Employee-specific methods in 'e'.
}
```

References in Python

- Python, as a dynamically-typed language, is simpler.
- There are references, but no such thing as a “reference type”.
  - Any reference variable can refer to *any* object.
  - No such thing as upcasting or downcasting.
- *But*, at a design level, you should still be clear about what type(s) each reference should refer to.

Upcasting Containers

- List<Person> can present some upcasting confusion.
  - The type is a bit more complicated than before.

You *can* do this:

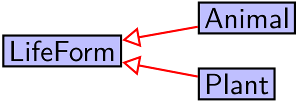
```
LinkedList<Person> linked = new LinkedList<>();
List<Person> list = linked; // LinkedList -> List
Object o = linked;          // LinkedList -> Object
```

But you *cannot* do this:

```
List<Employee> empList = ...;
List<Person> pList = empList; // Malfunction:
                              // does not compute.
```

You cannot upcast a container-of-Employee to a container-of-Person.

Upcasting Containers (2)

- What gives? Well, say that 

- A *container*-of-`LifeForm` has a two-part contract:

  Out – It must retrieve and return only `LifeForm` references.

  In – It must accept and store any `LifeForm` references.
- A container-of-`Animal` would violate the "in" part.
  - You cannot add a `Plant` to it.
- In code:

```
List<LifeForm> lifeForms = ...;
List<Animal> animals = ...;
lifeForms.add(new Plant()); // Required by contact.
animals.add(new Plant());   // Invalid.
```

- So, a container-of-`Animal` *cannot* be treated as a container-of-`LifeForm`.

Upcasting Containers: What If...?

- Just to dash your hopes, it doesn't work *either* way around:

```
List<LifeForm> lifeForms = ...;
List<Animal> animals = ...;

lifeForms = animals; // Violates the "in" contract.
animals = lifeForms; // Violates the "out" contract:
                     // lifeForms.get(0) might return a Plant.
```

- Java does have a more subtle way of dealing with this, though:

```
List<? extends LifeForm> returnsLifeForms = ...;
List<? super Animal> acceptsAnimals = ...;
```

  - Special super-types that omit either the "in" or "out" part of the contract.
  - We'll get back to this in the generics lecture.

Interface Inheritance

- Subclasses inherit both the public interface and implementation of their superclasses.
  - (Although the superclass's implementation is private, subclass objects still acquire it.)
- Of the two, interface inheritance is actually more important.
  - The "Go" language *only* supports this.
- This is the key to *polymorphism* ("many shapes").
  - Say your code needs to work with other objects.
  - You specify what *interface* you need to communicate with.
  - Many different classes can implement that interface.
  - Your code doesn't care which class it actually gets.
  - Very flexible.
  - Very low coupling between classes.

Pure Interface Inheritance – Java

- To define a Java interface:

```
public interface Shape
{
    int getArea(); // Implicitly public abstract
    int getPerimeter();
}
```

- Then:

```
public class Square implements Shape
{
    ...
    @Override public int getArea() { ... }
    @Override public int getPerimeter() { ... }
}
```

- Alternatively, **Shape** could be a normal abstract class, with only abstract methods.
  - But that "uses up" the `extends` relationship – not as flexible.

Pure Interface Inheritance – C++

- Use an ordinary class with only “pure virtual” methods:

```
class Shape
{
    public:
        virtual int getArea() = 0;
        virtual int getPerimeter() = 0;
}; // Nothing to implement, no .cpp file needed
```

- Then:

```
class Square : public Shape
{
    ...
    public:
        int getArea() override;
        int getPerimeter() override;
}; // We need square.hpp and square.cpp as usual
```

- **Square** could implement many different interfaces this way.

“Duck Typing” – Python

(“If it looks like a duck and quacks like a duck. . . ”)

- In Python, you don’t need an interface at all.

```
class Square:
    ...
    def getArea(self):
        return self.size * self.size
```

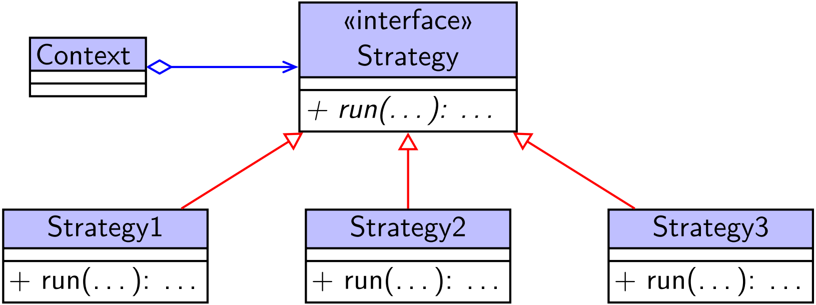
```
class Triangle: # Totally independent of Square
    ...
    def getArea(self):
        return self.base * self.height * 0.5
```

- No inheritance is required for polymorphism to work here.

```
def printArea(thing): # Takes anything
    print("The area is " + thing.getArea())
```

The Strategy Pattern

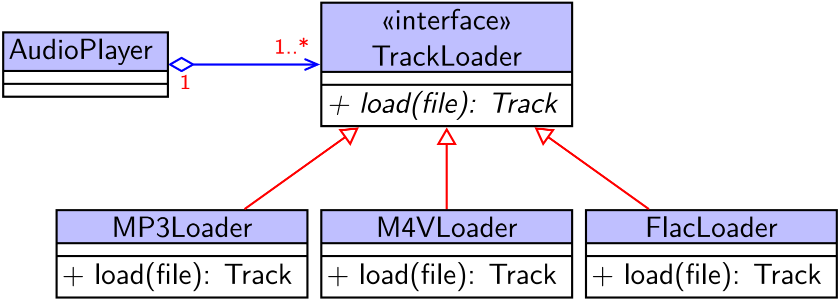
- The Strategy pattern demonstrates polymorphism.
- Used when you have several algorithms that all address the same/similar goals.
- Each algorithm gets its own class, with one method.
- The classes implement a common interface.



(The “Context” is whatever class needs to use a strategy object.)

The Strategy Pattern: Examples

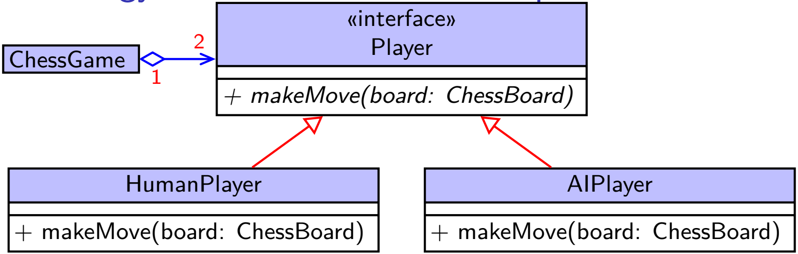
- There are many different uses for this:
  - Different file format saving/loading algorithms, for word processors, other office-type software, and many other things.
  - Different artificial intelligence engines in games.
  - Different sorting algorithms.



The Strategy Pattern: Discussion

- Strategy classes may not have any fields.
  - And, consequently, no accessors or mutators, and only a trivial constructor.
  - This isn't *necessarily* true, but often is.
- Each strategy class/object embodies an algorithm.
- Something else "owns" these objects.
  - The owning object invokes the strategy whenever required.
  - Something else again may construct strategy objects.
  - So, the choice of algorithm can be *decoupled* from the choice of timing.
- Strategy objects might also appear in a map container.
  - The key might be a piece of information known when the algorithm is about to be used.
  - e.g. a filename extension, to choose between file loading algorithms.

The Strategy Pattern: Another Example




```
public class ChessGame {
    private Player p1, p2; // Human or AI? We don't care.
    ...
    public void game() {
        do {
            p1.makeMove(chessBoard);
            p2.makeMove(chessBoard);
        } while(...);
    } // ChessGame doesn't need to know whether players are
}     // both human, both AI or a combination.
```

Yet Another Example: java.util.Comparator (1)

- Here's Java's Comparator interface:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object o);
}
```

- Subclasses of Comparator represent different ways (strategies) to compare things. e.g.: Person objects by name:

```
public class ComparePeopleByName
                        implements Comparator<Person> {
    @Override public int compare(Person p1,
                                  Person p2) {...}
    @Override public boolean equals(Object o) {...}
}
```

- Other subclasses could compare by age, by height, etc.

Yet Another Example: `java.util.Comparator` (2)

`Comparator` is used in a few different contexts:

- The `Arrays` and `Collections` classes contain utility methods for sorting and searching.
  - Some of these methods take `Comparator` objects to determine the sort or search order.
- `TreeSet` and `TreeMap` can (optionally) use a `Comparator` to determine how to keep their elements in sorted order.
- Anything else that needs to know the "ordering" of objects.

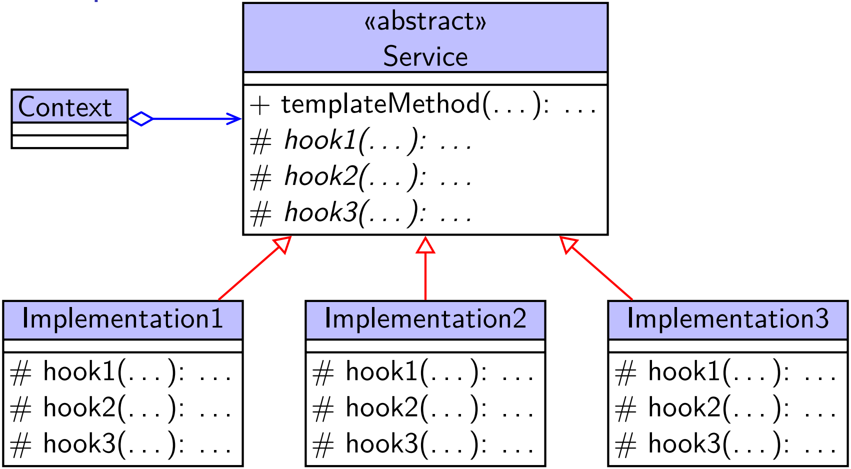
Implementation Inheritance

- This is the "ordinary" inheritance that you know and love.
  - `extends` in Java.
- A way of reusing code common to several classes.
- *Also* creates a common interface for those classes.
  - "Impure" interface inheritance.
- The superclass (or the "base" class) contains the common functionality.
- The subclasses (or "derived" classes) add their own distinctiveness.

The Template Method Pattern

- Demonstrates implementation inheritance.
- Used if you need several algorithms that are all identical, except for one or more specific steps.
  - You could also use Strategy for this, but Template Method is another approach.
- You have an abstract class, with the following:
  - A *non-abstract* method containing an algorithm (the "template method").
  - One or more `protected abstract` methods, called directly by the template method.
- The `protected abstract` methods are "hooks" into the algorithm.
  - By implementing these methods, you can hook-up your own code at pre-defined points in the algorithm.
  - This is done differently by each subclass.

The Template Method Pattern: UML

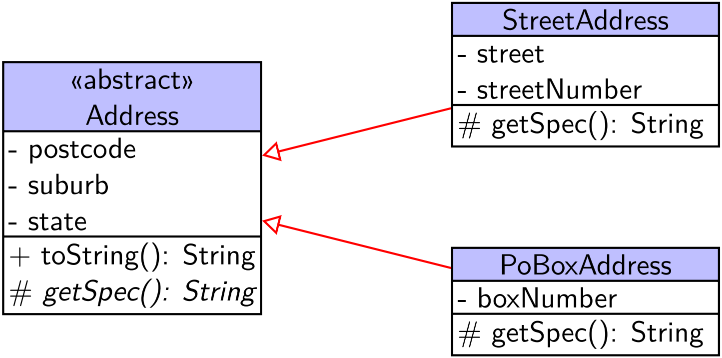


- Something else in the system calls `templateMethod()`.
- This in turn calls `hook1()`, `hook2()`, `hook3()`, etc.

Why `protected abstract`?

- `abstract` because the superclass doesn't know how to do it.
- `protected` because it's implementation, not part of the superclass's contract to the rest of the system.
- *But* these are not critical to the way the pattern works.
  - Patterns have variations.
- You can leave off "`abstract`".
  - You may have a "default" implementation that subclasses *can* override but don't have to.
  - But you would still expect *some* overriding!
- You can make use "`public`" instead of "`protected`".
  - The method may be useful by itself, not just as part of the algorithm.

Template Method: Example A (1)



- Different kinds of addresses contain different information.
- `toString()` is the template method.
- It calls `getSpec()`, which returns type-specific info.

Template Method: Example A (2)

```
public abstract class Address
{
    private int postcode;
    private String suburb;
    private String state;
    ...
    // Gets a type-specific part of the address.
    protected abstract String getSpec();

    public String toString() // Template method
    {
        return getSpec() + ", " +
            suburb + " " + postcode + " " + state;
    }
}
```

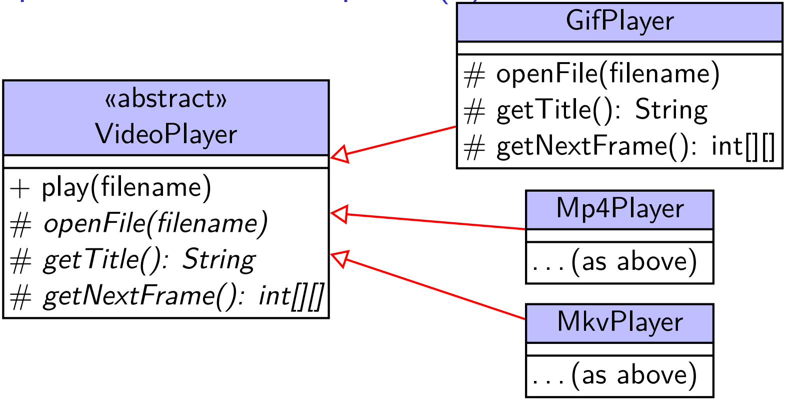
Template Method: Example A (3)

```
public class StreetAddress extends Address
{
    private String street
    private int streetNumber;
    ...
    @Override
    protected String getSpec()
    {
        return streetNumber + " " + street;
    }
}
```

Template Method: Example A (4)

```
public class PoBoxAddress extends Address
{
    private int boxNumber;
    ...
    @Override
    protected String getSpec()
    {
        return "PO Box " + boxNumber;
    }
}
```

Template Method: Example B (1)



- `VideoPlayer.play()` knows the overall video-playing algorithm.
- But it must call subclass methods to open a video file, extract the video title, and then decode each frame.
- These differ depending on the format.

Template Method: Example B (2)

```
public abstract class VideoPlayer
{
    protected abstract void openFile(String filename);
    protected abstract String getTitle();
    protected abstract int[][] getNextFrame();
    public void play(String filename) // <-- template method
    {
        openFile(filename);
        UI.displayText(getTitle());

        int[][] frame = getNextFrame();
        while(frame != null)
        {
            UI.displayImage(frame);
            frame = getNextFrame();
        }
    }   // Calls to subclass methods are highlighted.
}
```

Another Example: `javax.swing.JComponent`

- Java's GUI framework is made up of "components", all inheriting from `javax.swing.JComponent`.
  - All windows, buttons, sliders, etc. are components.
- `JComponent` defines these methods (among many others):

```
public void paint(Graphics g) {...}
protected void paintBorder(Graphics g) {...}
protected void paintChildren(Graphics g) {...}
protected void paintComponent(Graphics g) {...}
```

- The `paint()` method is called to display the component.
  - But while `JComponent` knows *when* to paint itself, it doesn't entirely know *how*.
  - `paint()` just calls the other three methods.
  - Each subclass of `JComponent` (e.g. `JButton`) overrides those methods.
  - (They are non-abstract, so there is a default implementation.)

C++: Values, Pointers and References

- C++ has some special complications, so we need to discuss it further.
- There are four ways to use objects in C++:

Value types – Where a variable directly holds an entire object.

C-style pointers – Where a variable holds a memory address of an object.
  - Most similar to Java/Python references.
  - As in C, you must "dereference" pointers (with "*" or "->").

C++ references – A limited kind of pointer.

Smart pointers – A special class that contains and manages a pointer.

Value Types

- C++ lets you have objects without pointers/references at all:

```
void myFunction()
{
    Person p("Sam", 42); // Declaration/construction
    p.displayName();
}                        // Destruction (automatic)
```

- Here, **p** is an object allocated on the stack.
- Operators act on the entire object; e.g.:
  - `Person p2 = p;` – creates a copy of the object.
  - `if(p == p2) {...}` – if the whole objects are equal, then...
- The constructor is called at declaration.
  - If you write "`Person p;`" you are still creating an object!
  - Compile error if there's no zero-argument constructor.
- The destructor is called automatically when the function ends.

"Resource Acquisition Is Initialisation" (RAII)

- RAII is a key C++ design philosophy, and uses value types:
  - You hold a "resource" (object) for a certain period of time.
  - It is constructed when declared (*never* left uninitialised).
  - It is destructed when it goes "out of scope".
- C++ has no garbage collector.
  - So we avoid pointers unless we really need them. They can potentially lead to memory leaks.
- Instead, RAII can be quite elegant; e.g. for file IO:

```
std::string myFunction()
{
    std::string s;
    std::ifstream myFile("xyz.txt"); // Open file
    std::getline(myFile, s);         // Read line
    return s;
}                                    // Close file
```

(**myFile** is declared/constructed on the stack.)

Value Types in Expressions

- You can construct a value type “on the fly”:

```
Person("Sam", 42).displayName();
```

```
if(p == Person("Sam", 42)) {...}
```

- Syntactically, the constructor behaves like a function that returns a Person value.

Value Types as Fields

- Let's also look at value types as class fields:

```
class RockBand
{
    private:
        std::string name;
        Person drummer;
        Person guitarist;
        Person singer;

    public:
        ...
}
```

- This leads to two questions:
  1. How do we initialise them?
  2. When do they get destructed?

Value Types as Fields: Construction

- C++'s constructors have an "initialisation list":

```
RockBand::RockBand(std::string newName) :
    name(newName),           // Member initialisers
    drummer("Barry", 19),
    guitarist("Sally", 23),
    singer("Amanda", 20)
{
}
```

- Why not just write "`singer = Person("Amanda", 20);`" in the main block? RAII is why.
- Being value types, these objects *cannot* be uninitialised, even momentarily while the constructor is running.
- We must initialise them not just *in* the constructor, but *before* the constructor's body.

Value Types as Fields: Destruction

- Fields go "out of scope" when their containing object is destructed.

```
void myFunction()
{
    RockBand myBand("Smeg");
    myBand.getSinger().displayName();
}   // name, drummer, guitarist and singer are all
    // destructed here, along with myBand.
```

- At the end of `myFunction`, `myBand` is destructed and so are all its value-type fields.
- If `Person` defines a destructor, it will be called here (once for each `Person` field).

C++: “References”

- C++ has a *special kind* of reference.
- Used to pass parameters, return-by-reference and handle exceptions.

```
void myFunction(TheClass& obj) {
    obj.method(); // obj is an alias of myObject
}
...
myFunction(myObject);  // Passing by reference
```

- Quite limited:
  - C++ references are not really variables, just aliases.
  - They cannot be used to construct an object.
  - They cannot be modified (you can’t change what it refers to).
  - They cannot be `null`.

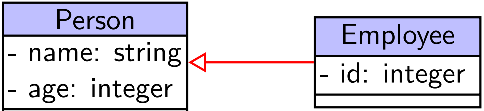
C++: Pointers

- C++ also has C-style pointers.

```
TheClass* obj;               // Declaration
obj = new TheClass(...);     // Construction
...
delete obj;                  // Destruction
```

- `new` and `delete` are similar to C's `malloc()` and `free()`, *except*:
  - After allocation, `new` invokes a constructor.
  - Before de-allocation, `delete` invokes a destructor.
- Don't forget `delete`, or you'll get a memory leak.
  - C++ has no real garbage collection (unlike Java and Python).

The Problem with Raw Objects – Illustrated



(8 bytes)

(12 bytes, including name and age)

```
void doStuff(Person& p) { ... }    // C++ reference
```

- **p** can be **Person** *or* **Employee** (as in Java and Python).
- **doStuff()** allocates memory for a reference only.
- **doStuff()** doesn't need to know how big the object is.

```
void doStuff(Person p) { ... }    // C++ raw object
```

- **p** must be **Person** exactly, not **Employee**.
- **doStuff()** allocates p's memory itself – 8 bytes only!

C++: What to Use? (1)

- Parameters – Use C++ references or pointers:

```
void myFunction(TheClass& obj) {...}
```

```
void myFunction(TheClass* obj) {...}
```

  - No advantage to value-typed objects here.
  - They're inefficient, due to copying, and will break inheritance.
- Local/Temporary Objects – Use value-typed (stack-based) objects when creating and destroying objects inside a single function/method:

```
void myFunction() {
    TheClass obj("Hello", 42);
    ...
}
```

  - Automatic object destruction – avoids memory leaks.
  - You know the exact object type, so no need to cater for subclasses.

C++: What to Use? (2)

- For return values, pointers are your best all-round solution:

```
TheClass* myFunction()
{
    TheClass* obj = new TheClass();
    return obj;
}
...
TheClass* obj = myFunction();
```

  - Works as you would expect in Java or Python.
  - *But* you must use “new” when creating a new object to return. Else, it will be wiped off the stack, giving an invalid pointer!
- You can return C++ references, but only if:
  - The caller is accessing an *existing* object; and
  - The caller doesn’t intend to “gain ownership” of the object.
- You can return value types, but only if:
  - You don’t mind that the whole object will be copied.
  - You’re not using inheritance.

C++: What to Use? (3)

- Mostly use pointers for class fields:

```
class TheOtherClass {
    private:
        TheClass* obj;
    ...
};
```

- C++ references are basically illegal here.
- You can use value types too:

```
private:
    TheClass obj;
```

  - But `TheClass` *should not* have any subclasses! Value types are not subject to inheritance.
  - `obj` is part of the same memory block as its containing object.
  - `obj` will be automatically destructed – avoids memory leaks.