# Comp151

# Overriding vs. Overloading

# Overriding and Virtual Functions

- When a derived class defines a method with the same name as a base class method, it overrides the base class method. e.g.

  `Student::print()` overrides `Person::print()`

- This is necessary if the behaviour of the base class method is not good enough for derived classes. The derived classes should all respond to the same message (`print()`!), but their response varies depending on the object.
- The designer of the base class (`Person`) must realize that this will be necessary, and declare `print()` to be a virtual function.
- Overriding is not possible if the method is not virtual.

# Virtual Functions vs. Non-Virtual Functions

- The designer of the base class must distinguish carefully between two kinds of methods:
  - If the method works exactly the same for all derived classes, it should not be a virtual method.
  - If the precise behaviour of the method depends on the object, it should be a virtual function.
- However, derived classes have to be careful in implementing this method because of the substitution principle. The "effect" (meaning) of calling the derived class method must be the "same" as for the base class method.

# Virtual Functions vs. Non-Virtual Functions . . .

- For example, `print()` should not be a method that does something completely different.
- Overriding is for specializing a behaviour, not changing the *semantics*.
- `fly()` must do what it promises, therefore we could not implement `Penguin` as a derived class of `Bird`.
- The compiler can only check that overriding is done *syntactically* correct, not whether the semantics of the method are preserved.

# Overriding vs. Overloading



Overloading: allows us to use functions or methods with the same name, but different arguments.

- The decision on which function to use (overload resolution) is done by the compiler when the program is compiled.
- There is NO dynamic binding.

Overriding: allows a derived class to provide a different implementation for a method declared in the base class.

- Overriding is only possible with inheritance and dynamic binding — without inheritance there is no overriding.
- The decision which method to use is done at the moment that the method is called.
- It only applies to member methods, not free functions.

# Example 1



```
int main()
{
    Person *p = new Person("Bill Clinton", "White House", law);
    delete p;

    Student *s = new Student("Simpson", "Springfield", computer_sci);
    s->enroll_course("comp151");
    delete s;
}
```

This works fine.

# Example 2



```
int main()
{
    Student *s = new Student("Simpson", "Springfield", computer_sci);
    s->enroll_course("comp151");

    Person *p = s;
    delete p;
}
```

- `delete p` calls the `Person` destructor. The `Student` destructor is not executed.
- The `Student` object itself is removed from the heap, but the resources it owns are not deleted.
- Therefore there is a memory leak in this code. The course array for Simpson is not destructed.

## Virtual Destructor



Again, the solution is to switch on dynamic binding, in this case for the destructor:

```
class Person
{
  public:
    Person(const string n, const string a, Department d);
    virtual ~Person() { }
    ...
};
```

- Now, `delete p` correctly calls the `Student` destructor if `p` points to a `Student` object.
- When a class does not have a virtual destructor, this is a strong hint that the class is not designed to be used as a base class.

# Non Virtual Example

```
//Non_Virt_Dest.cpp
# include <iostream.h>

class B {
   private:  int x;
   public:
      B(){cout << "creating B\n";}
     ~B(){ cout << "destroying B\n";}
};

class D: public B {
   private:  int  y;
   public:
      D(){cout << "creating D\n";}
     ~D(){cout << "destroying D\n";}
};

main() { B *b = new D; delete b; }
```

What is the output?

# Virtual Example



```
//Virt_Dest.cpp
# include <iostream.h>

class B {
   private: int x;
   public:
      B(){cout << "creating B\n";}
      virtual ~B(){ cout << "destroying B\n";}
};

class D: public B {
   private:  int  y;
   public:
      D(){cout << "creating D\n";}
      virtual ~D(){cout << "destroying D\n";}
};

main()
{ B *b = new D; delete b; }
```

What is the output?

# Calling Virtual Functions in Constructors: Example

```
class B {
  public:
    B() { this->f(); }
    virtual void f() { cout << "B::f()" << endl; }
};

class D : public B {
  public:
    D() { }
    virtual void f() { cout << "D::f()" << endl; }
};

int main() {
    B *p = new D;
    cout << "Object created" << endl;
    p->f();
}
```

# Calling Virtual Functions in Constructors: Output

(Note: *this* points to the current object.
We will see a formal definition later)

The output is:

```
B::f()
Object created
D::f()
```

- Do not rely on the virtual function mechanism during the execution of a constructor.
- This is not a bug, but necessary — how can the derived object provide services if it has not been constructed yet?