

Bug Avoidance

override

- Tell the compiler, that you think you override a virtual function of a base class with the new keyword:
override

```

/* Struct to hold information about cars.
 *
 * To have multiple cars of a certain different type, you can inherit from this struct.
 */
struct Car {
    /** Returns a generic estimate of the weight of a car (1 ton), if not overridden.*/
    virtual int getWeightInKg();

    /** Returns a generic estimate of the speed of a car (180), if not overridden.*/
    virtual int getMaxSpeedInKmh() const;

    /** Returns a generic estimate of the number of wheels of a car (4), if not overridden.*/
    virtual int getNWheels();

    /** Returns a generic estimate of the number of seats in a car (5), if not overridden.*/
    int getNSeats();
};

/** Class to use with the bigger and slower Monster Truck special car.*/
struct MonsterTruck: public Car {
    /** Monster Trucks are much heavier on average the average cars (3 tons).*/
    int getWeightInKg();

    /** Monster Trucks are much slower than average cars (80).*/
    int getMaxSpeedInKmh();

    /** Monster Trucks have double wheels (--> 8) due to their big weight.*/
    virtual int getNWheels();

    /** Monster Trucks have one big front bench with 3 seats.*/
    int getNSeats();
};

```

Have a glimpse at these two classes...

```
int main()
{
    using namespace std;

    MonsterTruck monsterTruck;
    Car& car = monsterTruck;
    cout << "Weight: " << car.getWeightInKg() << endl;
    cout << "Speed: " << car.getMaxSpeedInKmh() << endl;
    cout << "Number of Wheels: " << car.getNWheels() << endl;
    cout << "Number of Seats: " << car.getNSeats() << endl;

    cout << "Once more the Speed: " << monsterTruck.getMaxSpeedInKmh() << endl;
    cout << "Once more the number of Seats: " << monsterTruck.getNSeats() << endl;
}
```

**We assume you write some code
like this...
Already written for you!**

Warm Up

- Take file INIOVERRIDE.cc and compile it with
`g++ -std=c++11 -Wall INIOVERRIDE.cc -o INIOVERRIDE`
- Execute the executable!

```

int main()
{
    using namespace std;

    MonsterTruck monsterTruck;
    Car& car = monsterTruck;
    cout << "Weight:          " << car.getWeightInKg() << endl;
    cout << "Speed:            " << car.getMaxSpeedInKmh() << endl;
    cout << "Number of Wheels: " << car.getNWheels() << endl;
    cout << "Number of Seats:  " << car.getNSeats() << endl;

    cout << "Once more the Speed: " << monsterTruck.getMaxSpeedInKmh() << endl;
    cout << "Once more the number of Seats: " << monsterTruck.getNSeats() << endl;
}

```

```

[1642] [heck@ekplx76:/home/heck/GridKaSchool/meinKram]$
g++ -std=c++11 INI0verride.cc -o INI0verride -Wall

```

```

[1642] [heck@ekplx76:/home/heck/GridKaSchool/meinKram]$
./INI0verride
Weight:          3000
Speed:          180
Number of Wheels: 8
Number of Seats: 5
Once more the Speed: 80
Once more the number of Seats: 3

```

Programmers **should** expect the same result in the calls to the reference and to the object itself!

```

/* Struct to hold information about cars.
 *
 * To have multiple cars of a certain different type, you can inherit from this struct.
 */
struct Car {
    /** Returns a generic estimate of the weight of a car (1 ton), if not overridden.*/
    virtual int getWeightInKg();

    /** Returns a generic estimate of the speed of a car (180), if not overridden.*/
    virtual int getMaxSpeedInKmh() const;

    /** Returns a generic estimate of the number of wheels of a car (4), if not overridden.*/
    virtual int getNWheels();

    /** Returns a generic estimate of the number of seats in a car (5), if not overridden.*/
    int getNSeats();
};

```

```

/** Class to use with the bigger and slower Monster Truck special car.*/
struct MonsterTruck: public Car {
    /** Monster Trucks are much heavier on average than average cars (3 tons).*/
    int getWeightInKg();

    /** Monster Trucks are much slower than average cars (80).*/
    int getMaxSpeedInKmh();

    /** Monster Trucks have double wheels (--> 8) due to their big weight.*/
    virtual int getNWheels();

    /** Monster Trucks have one big front bench with 3 seats.*/
    int getNSeats();
};

```

A const function is not the same as its non-const counterpart. Therefore this is not overriding anything.

Function in base class is not declared virtual, therefore no overriding, but shadowing. In a good compiler this throws a warning.

Try out override!

- Write override in the derived classes at all points, where you think it makes sense.

Syntax is as follows:

```
/** Monster Trucks are much heavier on average the average cars (3 tons).*/  
int getWeightInKg() override;
```

Then try compilation again!

This is what you should get

```
g++ -std=c++11 -Wall Override.cc -o Override
```

```
Override.cc:27:7: error: "int MonsterTruck::getMaxSpeedInKmh()" marked  
override, but does not override
```

```
Override.cc:33:7: error: "int MonsterTruck::getNSeats()" marked override, but  
does not override
```

Should you expect warnings?

```
/** Struct to hold information about cars.
 * To have multiple cars of a certain different type, you can inherit from this struct.
 */
struct Car {
    /** Returns a generic estimate of the weight of a car (1 ton), if not overridden.*/
    virtual int getWeightInKg();

    /** Returns a generic estimate of the speed of a car (180), if not overridden.*/
    virtual int getMaxSpeedInKmh() const;

    /** Returns a generic estimate of the number of wheels of a car (4), if not overridden.*/
    virtual int getNWheels();

    /** Returns a generic estimate of the number of seats in a car (5), if not overridden.*/
    int getNSeats();
};

/** Class to use with the bigger and slower Monster Truck special car.*/
struct MonsterTruck: public Car {
    /** Monster Trucks are much heavier on average than average cars (3 tons).*/
    int getWeightInKg();

    /** Monster Trucks are much slower than average cars (80).*/
    int getMaxSpeedInKmh();

    /** Monster Trucks have double wheels (--> 8) due to their big weight.*/
    virtual int getNWheels();

    /** Monster Trucks have one big front bench with 3 seats.*/
    int getNSeats();
};
```

Should(?) warn, that there is already a function you inherit of. Don't use virtual at the derived function.

Warns about shadowing another function.

- With all warnings on, a good compiler...

Summary for **override**

- Use it consequently just as you use **const** consequently to make the compiler help you to detect bugs.
- Don't use **virtual** for overriding functions. It should indicate to be at the top of the polymorphism order. **override** now indicates the fact, that a function is lower in the polymorphism order.

final

- Tell the compiler to not allow further overriding of a function with the keyword **final**

```
#include <iostream>

struct Vehicle {
    enum class VehicleType{
        HooverVehicle, // = 0
        GroundVehicle // = 1
    };

    virtual VehicleType getVehicleType() const = 0;
};


struct Car: public Vehicle {
    VehicleType getVehicleType() const override {
        return VehicleType::GroundVehicle;
    };
};

struct MonsterTruck: public Car {
    VehicleType getVehicleType() const override {
        return VehicleType::HooverVehicle;
    };
};

int main()
{
    using namespace std;

    MonsterTruck monsterTruck;
    Car& car = monsterTruck;
    cout << "Vehicle Type: " << static_cast<int>(car.getVehicleType()) << endl;
}
```

This is a reference to a car! Conceptionally you expect a car to be a ground vehicle, not a hoovering vehicle!



```
#include <iostream>

struct Vehicle {
    enum class VehicleType{
        HooverVehicle, // = 0
        GroundVehicle // = 1
    };

    virtual VehicleType getVehicleType() const = 0;
};

struct Car: public Vehicle {
    //Declared final, because it will be very confusing to have references of type car, that
    // are not declared a GroundVehicle;
    VehicleType getVehicleType() const override final {
        return VehicleType::GroundVehicle;
    };
};

struct MonsterTruck: public Car {
    VehicleType getVehicleType() const override {
        return VehicleType::HooverVehicle;
    };
};

int main()
{
    using namespace std;

    MonsterTruck monsterTruck;
    Car& car = monsterTruck;
    cout << "Vehicle Type: " << static_cast<int>(car.getVehicleType()) << endl;
}
```

This is the syntax you need.

Modify INIFinal.cc accordingly and compile!

First Finding

- In contrast to override the application of final can't follow simple rules. It is a *conceptional* decision to decide, that subclasses shouldn't behave differently.

As well classes can be marked final

The only function can be used more efficiently!

g++ -std=c++11 -Wall -O3 -ftree-vectorizer-verbose=6 INIFinal2.cc

```
#include <vector>
#include <iostream>
```

```
template<class ANumber>
struct AbsCalculator {
    virtual ANumber addNumbers(ANumber firstNumber, ANumber secondNumber){
        return (firstNumber + secondNumber);
    }
};
```

```
struct IntegerCalculatorBase {
    virtual int addIntegerNumbers(int firstNumber, int secondNumber) = 0;
};
```

```
struct FinalIntegerCalculator final public IntegerCalculatorBase {
    int addIntegerNumbers(int firstNumber, int secondNumber) override {
        return (firstNumber + secondNumber);
    }
};
```

```
struct NonFinalIntegerCalculator : public IntegerCalculatorBase {
    int addIntegerNumbers(int firstNumber, int secondNumber) override {
        return (firstNumber + secondNumber);
    }
};
```

```
int main (){
int addends1[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int addends2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum[10];
```

```
AbsCalculator<int> templateCalculator;
for (int ii = 0; ii < 10; ii++){
    sum[ii] = templateCalculator.addNumbers(addends1[ii], addends2[ii]);
}
std::cout << sum[0] << std::endl;
```

```
NonFinalIntegerCalculator* nonFinalPointer = new NonFinalIntegerCalculator();
for (int ii = 0; ii < 10; ii++){
    sum[ii] = nonFinalPointer->addIntegerNumbers(addends1[ii], addends2[ii]);
}
std::cout << sum[0] << std::endl;
```

```
FinalIntegerCalculator* finalPointer = new FinalIntegerCalculator();
for (int ii = 0; ii < 10; ii++){
    sum[ii] = finalPointer->addIntegerNumbers(addends1[ii], addends2[ii]);
}
std::cout << sum[0] << std::endl;
```


... and a less obvious example for efficiency

```
#include <iostream>
struct DriftChamberHitBase{
};

struct DriftChamberHitHelium{
};

struct DriftChamberStereoHit : public DriftChamberHitHelium{
    DriftChamberStereoHit(float stereoAngle) : m_stereoAngle(stereoAngle){}

    float getStereoAngle() {
        return m_stereoAngle;
    }

private:
    float m_stereoAngle;
};

int main(){
    DriftChamberHitHelium hits[8];
    for (auto hit : hits) {
        hit = DriftChamberStereoHit(0.003); //STUPID, but compiler won't warn.
    }
}
```

g++ -std=c++11 -Wall INIFinal3.cc -o INIFinal3

Usually, to avoid such things from happening, people are using arrays (or vectors) of pointers. However, this can cause the actual objects to be scattered around in the memory and make the caching in the CPU less efficient.

If you make `DriftChamberHitHelium` final, you don't have to worry about mistakes, when using an array of objects.

Second Finding

- **final** can improve efficiency, both by facilitating vectorization and by helping with better data structures.

Smart Pointers

- Although already available via boost, smart pointers have increased in prominence.

- What is a smart pointer?

- an abstract data type that simulates a pointer while providing additional features, such as automatic memory management.
- These additional features are intended to **reduce bugs** caused by the misuse of pointers.

- What does it do?

- It prevents **most** situations of memory leaks by making the resource de-allocation automatic.
- The resource controlled by a smart pointer is automatically destroyed when the last (or only) owner of the resource is destroyed.
- It also eliminates dangling pointers by postponing destruction until the resource is no longer in use.

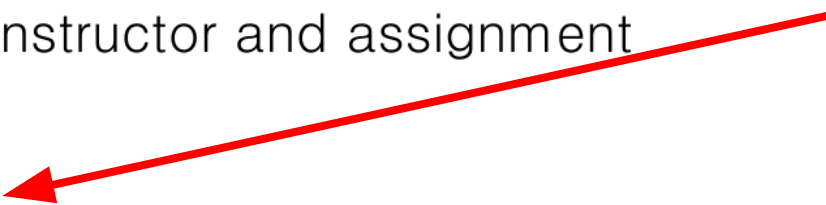
- C++11 provides:

- (1) `unique_ptr` : memory is released when it goes out of the scope.
- (2) `shared_ptr` : can have multiple `shared_ptr` pointing a same object.
- (3) `weak_ptr` : work with `shared_ptr` but have no ownership.

(1) unique_ptr

- This pointer type has its copy constructor and assignment operator explicitly deleted.
- This cannot be copied.
- This can be moved using `std::move`, which allows one `unique_ptr` object to transfer ownership to another.

A bit more on this in a moment.



```
std::unique_ptr<int> p1(new int(5));  
std::unique_ptr<int> p2 = p1; //Compile error.  
std::unique_ptr<int> p3 = std::move(p1); //Transfers ownership.  
  
p3.reset(); //Deletes the memory.  
p1.reset(); //Does nothing.
```

(2) shared_ptr

- Each copy of the same shared_ptr owns the same pointer.
- This represents reference-counted ownership of a pointer.
 - Counter is in(de)cremented if # of shared_ptr in(de)creases.
- The pointer will only be freed if all instances of the shared_ptr in the program are destroyed.

```
std::shared_ptr<int> p1(new int(5));  
std::shared_ptr<int> p2 = p1; //Both now own the memory.  
  
p1.reset(); //Memory still exists, due to p2.  
p2.reset(); //Deletes the memory, since no one else owns the memory.
```

- Reference counting → circular references are potentially a problem.

```
class loop{  
public:  
    loop(){  
    ~loop(){  
    std::shared_ptr<loop> ptr;  
};
```

```
int main(){  
    std::shared_ptr<loop> p1(new loop);  
    std::shared_ptr<loop> p2(new loop);  
    p1->ptr = p2;  
    p2->ptr = p1;  
    return 0;  
} // instance is not released
```

(3) weak_ptr

- Have no ownership.
- To break up cycles, weak_ptr can be used to access the stored object.
- The stored object will be deleted if the only references to the object are weak_ptr references.

```
std::shared_ptr<int> p1(new int(5));
std::weak_ptr<int> wp1 = p1; //p1 owns the memory.

{
    std::shared_ptr<int> p2 = wp1.lock(); //Now p1 and p2 own the memory.
    if(p2) //Always check to see if the memory still exists
    {
        //Do something with p2
    }
} //p2 is destroyed. Memory is owned by p1.

p1.reset(); //Memory is deleted.

std::shared_ptr<int> p3 = wp1.lock(); //Memory is gone, so we get an empty shared_ptr.
if(p3)
{
    //Will not execute this.
}
```

lvalues and rvalues

- What are lvalues and rvalues? Originally:
 - An lvalue is something that can occur on the left hand side of an assignment.
 - An rvalue is something that can only occur on the right hand side.

```
x = 5; // x is an lvalue
```

```
y = 7; // y is an lvalue
```

```
3 = x; // Wrong, 3 is not an lvalue
```

```
x * y = 3; // Wrong, x * y is not an lvalue
```

- A better definition might be that an lvalue is something that can be assigned to, or something that has a memory location that can be referred to, or perhaps something that has a name.
- An rvalue is then something that is not an lvalue.

Copies and moves

- There are many places in c++ where we may perform copy operations which use a lot of resources, e.g.:

```
vector<int> addSeven (const vector<int>& v) {  
    vector<int> new_values;  
    for (auto itr = v.begin(), end_itr = v.end(); itr != end_itr; ++itr) {  
        new_values.push_back(*itr + 7);  
    }  
    return new_values;  
}
```

```
int main() {  
    vector<int> v;  
    for (int i = 0; i < 100; ++i) {  
        v.push_back(i);  
    }  
    v = addSeven(v);  
    ...  
}
```

- A temporary vector is created by the return, and then the assignment at the end of main.

(Note: In many cases compiler's return value optimisation or specialisations within standard library classes ameliorate the need for extra copies for us).

- Another example is the swap function, which performs extra copies:
- ```
template<class T> swap(T& a, T& b) {
 T tmp(a); // now we have two copies of a
 a = b; // now we have two copies of b
 b = tmp; // now we have two copies of tmp (aka a)
}
```
- This can use a lot more resources than need to be used, (particularly if the copy operation is particularly expensive in terms of resources).
- We may not actually want to use a copy if we just want to move something.
  - I.e we make a copy of something that is going to disappear anyway:
  - We would like to move the temporary to where it will be used,
  - But we cannot provide an address/reference for an rvalue.

# rvalue references

- C++11 introduces a new type of reference: the rvalue reference.

- This is identified by &&:

```
T&& rref;
```

- The familiar c++ reference can now be referred to as the lvalue reference:

```
T& lref;
```

- The introduction of rvalue references allows for the introduction of moves to c++. This is most commonly seen in:

- Move constructor,
- Move assignment operator.

- These are comparable to the copy constructor, and copy assignment operator.

- This is shown for the move constructor (on the next slide), the move assignment operator is obtained via a similar analogy.

# Move constructor

- Consider a simple class `myClass`, which contains an array (`_values`) and a size (`_size`):
- The copy constructor takes an lvalue reference to another instance of my class, and copies all of the contents into a new instance.

```
// copy constructor
myClass (const myClass& other)
: _values(new int[other._size])
, _size(other._size) {

 for (int i(0); i < _size; ++i) {
 _values[i] = other._values[i];
 }
}
```

# Move constructor

- Consider a simple class `myClass`, which contains an array (`_values`) and a size (`_size`):
- The copy constructor takes an lvalue reference to another instance of my class, and copies all of the contents into a new instance.

```
// copy constructor
myClass (const myClass& other)
: _values(new int[other._size])
, _size(other._size) {

 for (int i(0); i < _size; ++i) {
 _values[i] = other._values[i];
 }
}
```

```
// move constructor
myClass (myClass&& other)
: _values(other._values)
, _size(other._size) {

 other._values = NULL;
 other._size = 0;
}
```

- The move constructor takes a (non-const) rvalue reference to another instance of the class.
- It takes the contents of the other instance.
- It then sets the contents of other to NULL.
  - This is necessary so that they are not destroyed when the destructors for (temporary) instances are called.

# More on moving, and `std::move`

- If the class you wish to write a move constructor for contains an object then it is not enough to do:

```
_someObject(other._someObject)
```

- It is necessary to use

```
_someObject(std::move(other._someObject))
```

in case that the object in question itself calls a move function.

- `std::move` is a function that comes from `<utility>`.
- It takes something that is an lvalue, and allows it to be used as an rvalue, and moved.
- Note: `x = std::move(y);`
- Here the content of `y` is moved to `x`, but `y` is still available but its contents may be invalid. This is not an issue if we are copying temporary objects, as is commonly the case.

# Back to the swap example

- Swap can now be written:
- ```
template<class T>
void swap(T& a, T& b) {
    T tmp = std::move(a); // could invalidate a
    a = std::move(b);     // could invalidate b
    b = std::move(tmp);   // could invalidate tmp
}
```
- There are now no copies, some objects could be invalidated, but these will be thrown away anyway.

Other uses of moves

- There are cases where a set of arguments might be passed to function, which are then passed to another function:
 - For example, arguments are passed to a factory method, which then passes them to a constructor.
- It is possible that the arguments are not passed correctly.
- This can now be avoided using `std::forward`, which ensures that the arguments are exactly the same.

```
▪ template <class T, class A1>
  std::shared_ptr<T>
  factory(A1&& a1) {
    return std::shared_ptr<T>(new T(std::forward<A1>(a1)));
  }
```


Summary

- C++11 introduces rvalue references, denoted by &&.
- These allow for the introduction of moves which can ameliorate the need for lots of unnecessary copy actions.
- Commonly these are seen move constructors and move assignment operators:

```
myClass(const myClass&);           // copy constructor
myClass(myClass&&);                 // move constructor
myClass& operator=(const myClass&); // copy assignment
myClass& operator=(myClass&&);     // move assignment
```
- The `std::move` function can be used to move objects. It can also be considered as changing an lvalue to an rvalue.
- The addition of move allows for other options such as perfect forwarding.

Resources

- Some websites with information on the issues discussed here:
- <http://www.stroustrup.com/C++11FAQ.html#rval>
- <http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html>
- <http://en.cppreference.com/w/cpp/utility/move>
- <http://en.cppreference.com/w/cpp/utility/forward>
- http://en.cppreference.com/w/cpp/language/move_constructor
- <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html>