# F6b: Smart Pointers

# Memory management

- Garbage collection – is permitted in C++ but is not common
- Smart pointers
- Other ways:

# Garbage Collection

- Se the objects as a forrest, all objects that is not **reachable from a root** can be collected.

- Roots:
  - All static variables
  - Everything in the Stack

- Needs knowledge about the type of all objects (takes extra memory for objects without virtual) The type of the variables in the stack is not obvious

# Other ways, an example

- Many system have a stage behavior, e.g. a scen in a game. When the control leaves the stage everything can be thrown away.
- Keep a set of all objects allocated during the stage.
- When leaving the stage delete every object in the set.
- If you want to delete earlier you can keep a deleted list to stop double deletion.

# Smart Pointers, Basic

- automatic delete at function exit
- Overloading “operator ->”

```
void Foo {
    Car* temp(new Car(…));
    …
    temp -> SetReg(“ABC001”);
    …
    cout << temp -> GetReg();
    …
    delete temp;
}
```

```
void Foo {
    DeletePtr<Car> temp(new Car(…));
    …
    temp -> SetReg(“ABC001”);
    …
    cout << temp -> GetReg();
    …

}  // temp deallokeras automatiskt
```

# DeletePtr CODE

# Problems

- Simple: when calling a function that wants a Car
  - Overload operator *

- When calling a function that wants a pointer to Car, declaration

```
void Bar(Car * p) {…}
```

  - Automatic conversion (operator T*)

```
Bar(temp);   //temp is converted to Car*
```

  - Explicit conversion

```
Bar(static_cast<Car *>(temp));
```

  - Access function: “get()” access the underlying pointer

```
Bar(temp.get());
```

# unique_ptr

- Assignment mean transfer of ownership:

```
unique_ptr<Car> p1(new Car());
unique_ptr<Car> p2(new Car());
p1 = p2
```

1. p1's car is deleted
2. p1 get ownership of p2's car
3. p2 don't hold any car anymore

# unique_ptr Code

# Shared pointer

- Reference counting to keep track of the references
- When the last shared pointer referencing the object is deleted, the object is deleted.
- Assignment => ++count
- Delete of smart pointer => -- count

# Shared, problems

- Needs a counter allocated on heap (other solutions possible but not used)
- Circular dependencies: the objects will never be deleted!
- weak pointer: The object a weak pointer reference can be deleted
- But the weak pointer will know if the object is live or dead