# Class Inheritance

# OOP is great, BUT you have to do it right!

Strive for these Core Characteristics of Classes:

- Consistent Abstraction
  - Allows for a consistent visualization of the system
- Encapsulate Information and Hide Information
  - Hurts your brain less.
  - Easier to read (self-documenting)
  - Makes change easier (refactoring)
- Inherit (when it simplifies)
  - Capitalizes on re-use, less code, more abstraction.
- Identify and Isolate Areas Prone to Change
  - Design for change (if it is relatively easy)
- Loose Coupling Across Classes, Strong Cohesion Within

What components and concepts of OOP (or C++) help you design for change?

- Identify 3-4 concepts.
- For each, identify how these help you.

# Modularity and Code Reuse

Combining Classes
is probably your most powerful tool
for modularity and code reuse.

But will you use
INHERITANCE
or
COMPOSITION
??

# Coupling and Classes : Is it Composition or Inheritance ?

**“has-a” = Containment and Composition**

Employee “has-a” name = member of class.
Employee “has-a” UserAccount = UserAccount object is member.

**“is-a” = Inheritance**

Part-Time Employee “is-a” specialization of Employee,
PartTime inherits from Employee.

McConnel Examples

- Liskov Principle *“Subclasses must be usable through the base class interface.”*
  - Bank Accounts : Interest Bearing VS Interest Charging (p. 144)
- Overriding routines that do nothing.
  - ScratchlessTaillessMicelssMilklessCat (p. 146)

# Method and Member Access

public: public to everything.
private: private to everything!
protected: public to derived classes, private to everything else.

CAREFUL!

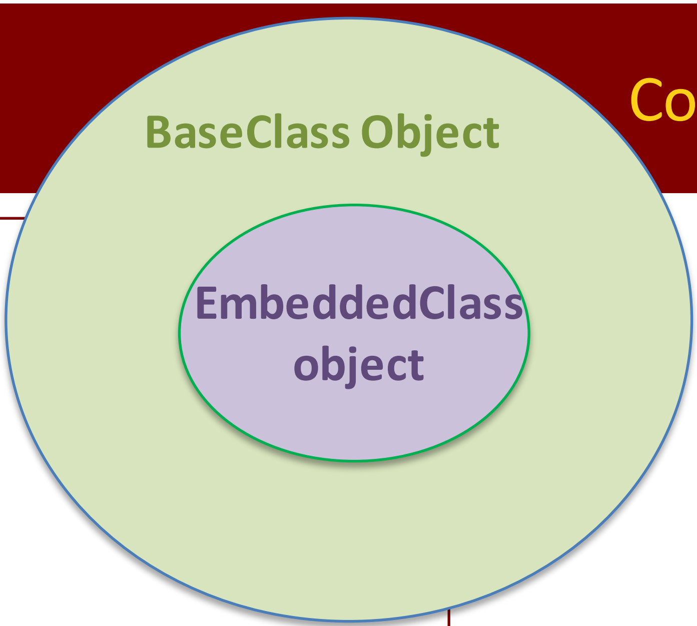
Subclasses cannot override
private elements of the base class.

A class that contains an object of another class
cannot override private elements of that object.

Using “virtual” keyword on a class method
does not guarantee that the subclass method
will override the base when upcasting.

# Composed and Derived Classes

```
class BaseClass {
public:
  virtual ...
.
private:
  EmbeddedClass  class_object_;
.
}
```

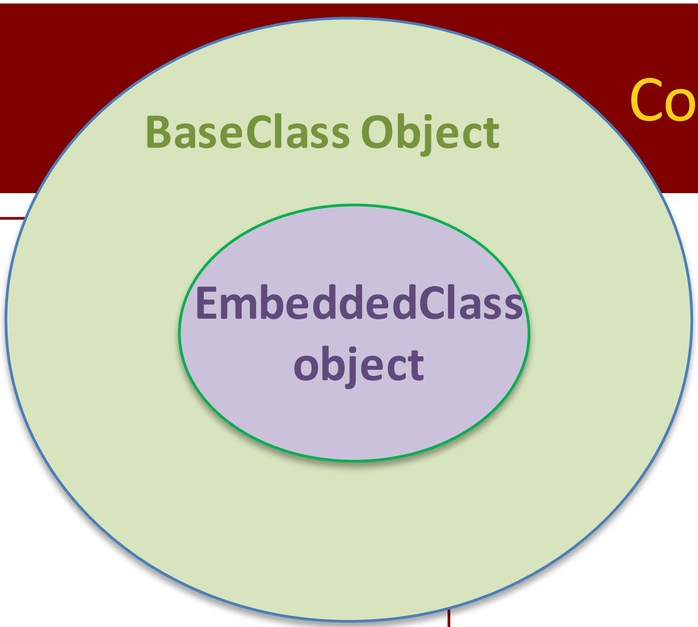


```
class EmbeddedClass{
.
.
.
}
```

To Understand Access, Visualize the Objects

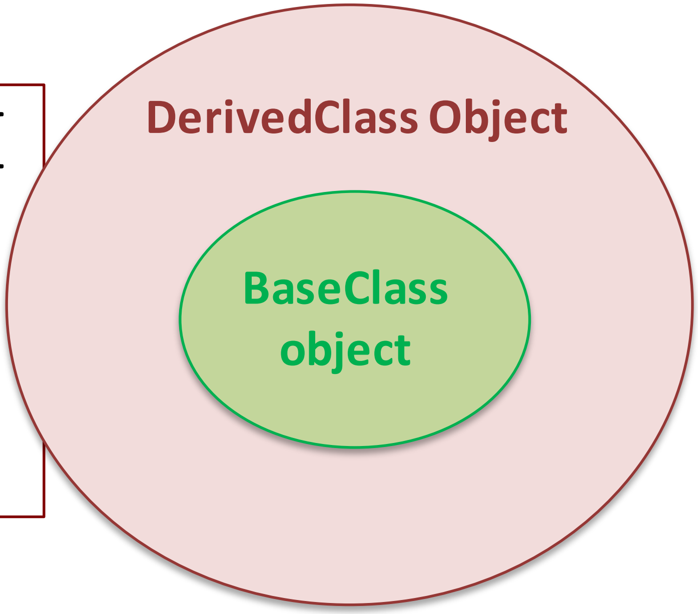
# Composed and Derived Classes

```
class BaseClass {
public:
  virtual …
.
private:
  EmbeddedClass  class_object_;
.
}
```



```
class EmbeddedClass{
.
.
.
}
```

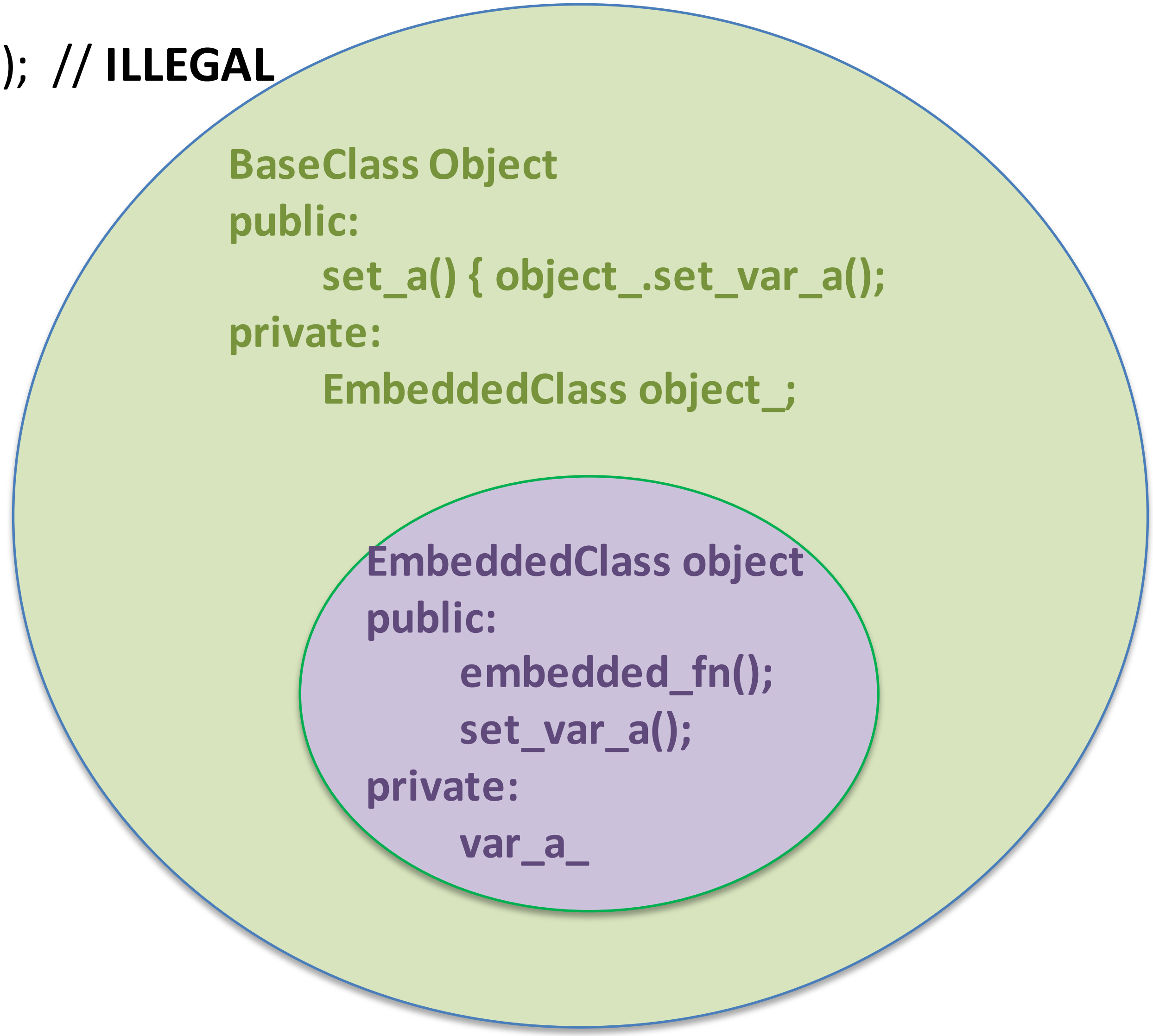
```
class DerivedClass  : public BaseClass {
.
.
.
}
```



To access any methods or members of Embedded, need to go through Base:

// instantiate
BaseClass base_obj_;

// **base_obj_.object_.**set_var_a(); // **ILLEGAL**
base_obj_.set_a()



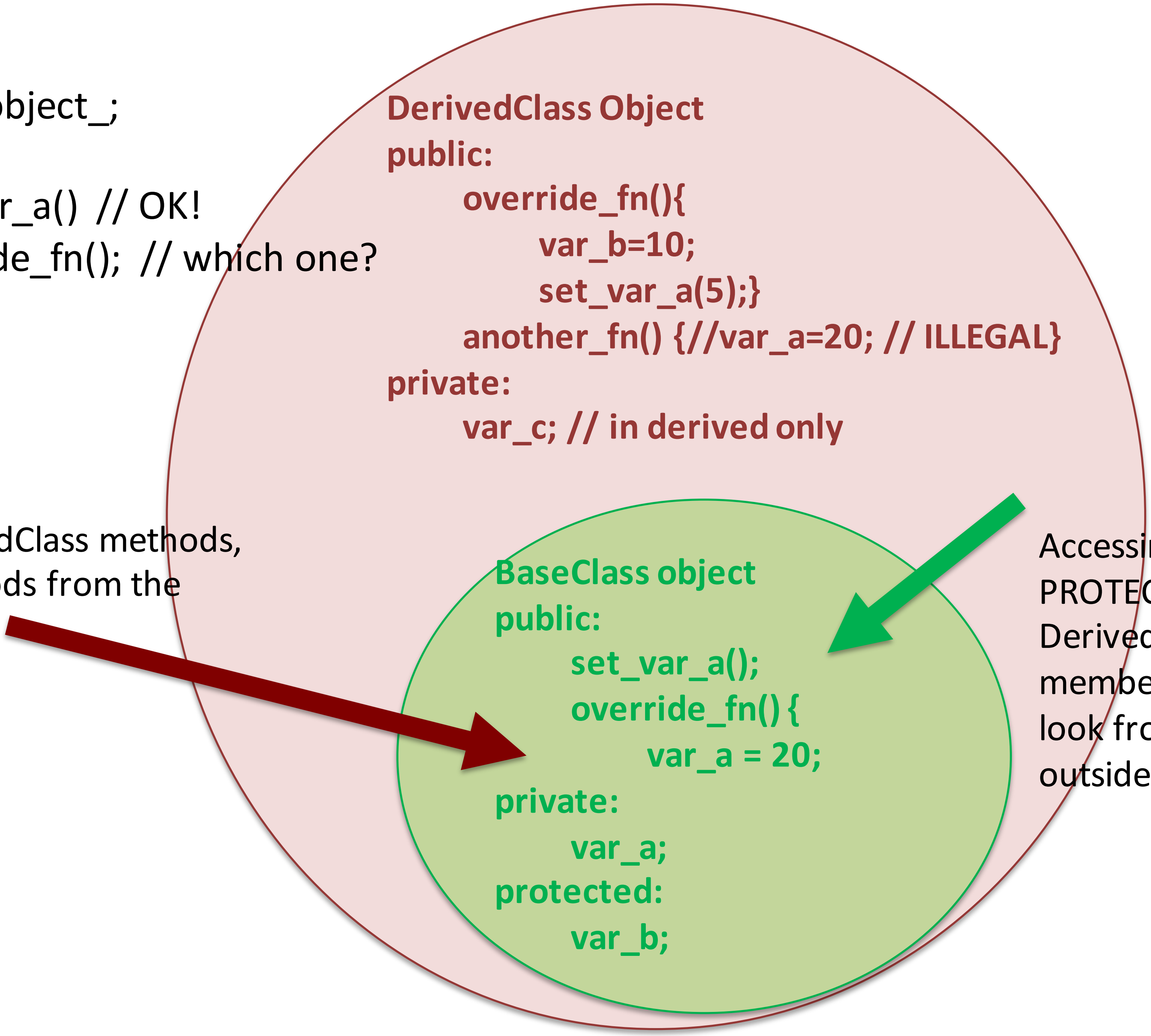
```
BaseClass Object
public:
        set_a() { object_.set_var_a();
private:
        EmbeddedClass object_;
```

```
EmbeddedClass object
public:
        embedded_fn();
        set_var_a();
private:
        var_a_
```

Members and methods might be in Derived or they might be in Base

```
// instantiate
DerivedClass object_;

object_.set_var_a() // OK!
object_.override_fn(); // which one?
```



**DerivedClass Object**

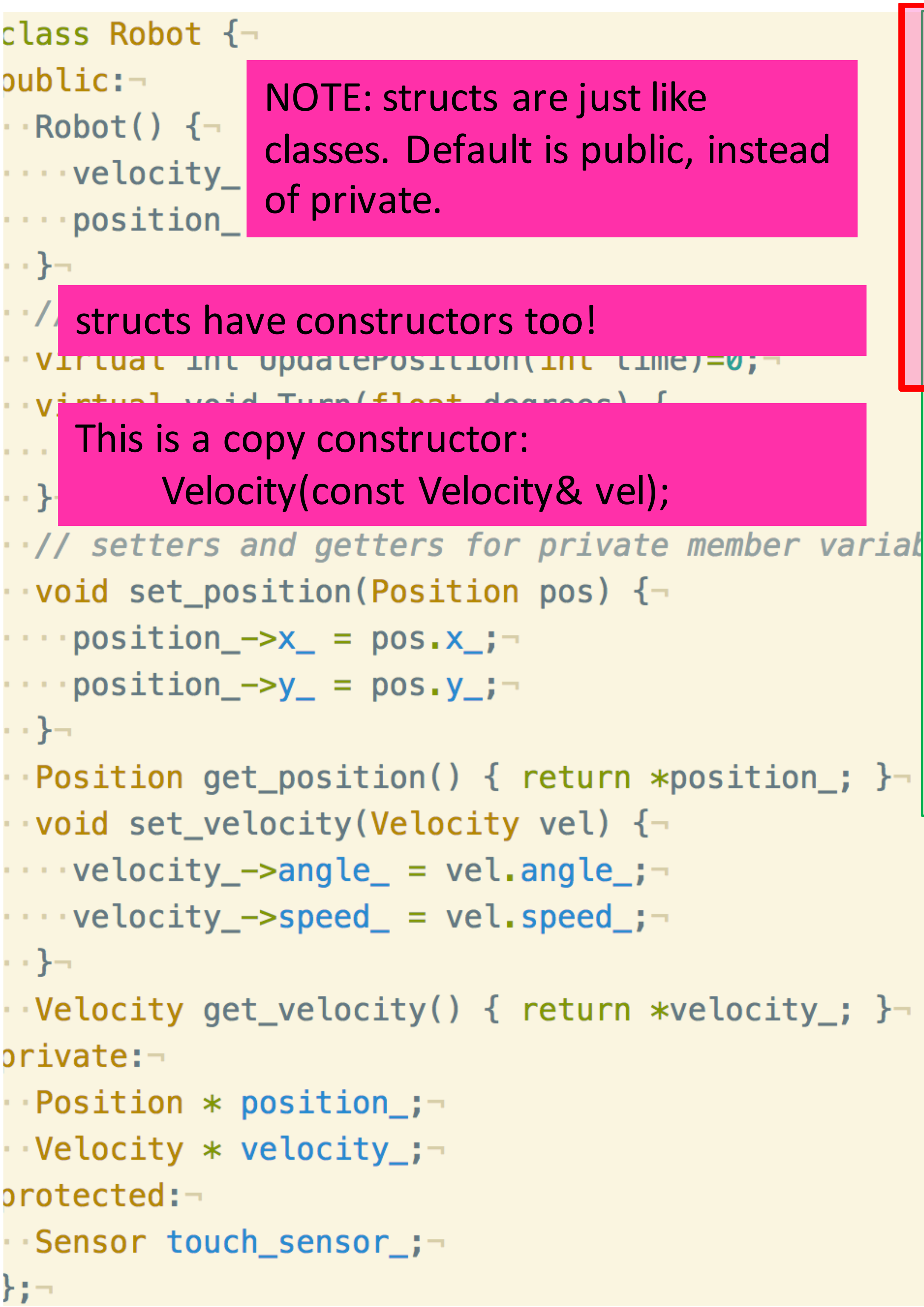
```
public:
        override_fn(){
                var_b=10;
                set_var_a(5);}
        another_fn() {//var_a=20; // ILLEGAL}
private:
        var_c; // in derived only
```

**BaseClass object**

```
public:
        set_var_a();
        override_fn() {
                var_a = 20;
private:
        var_a;
protected:
        var_b;
```

Calling
PUBLIC DerivedClass methods,
look for methods from the
outside in.

Accessing
PROTECTED
DerivedClass
members,
look from the
outside in.



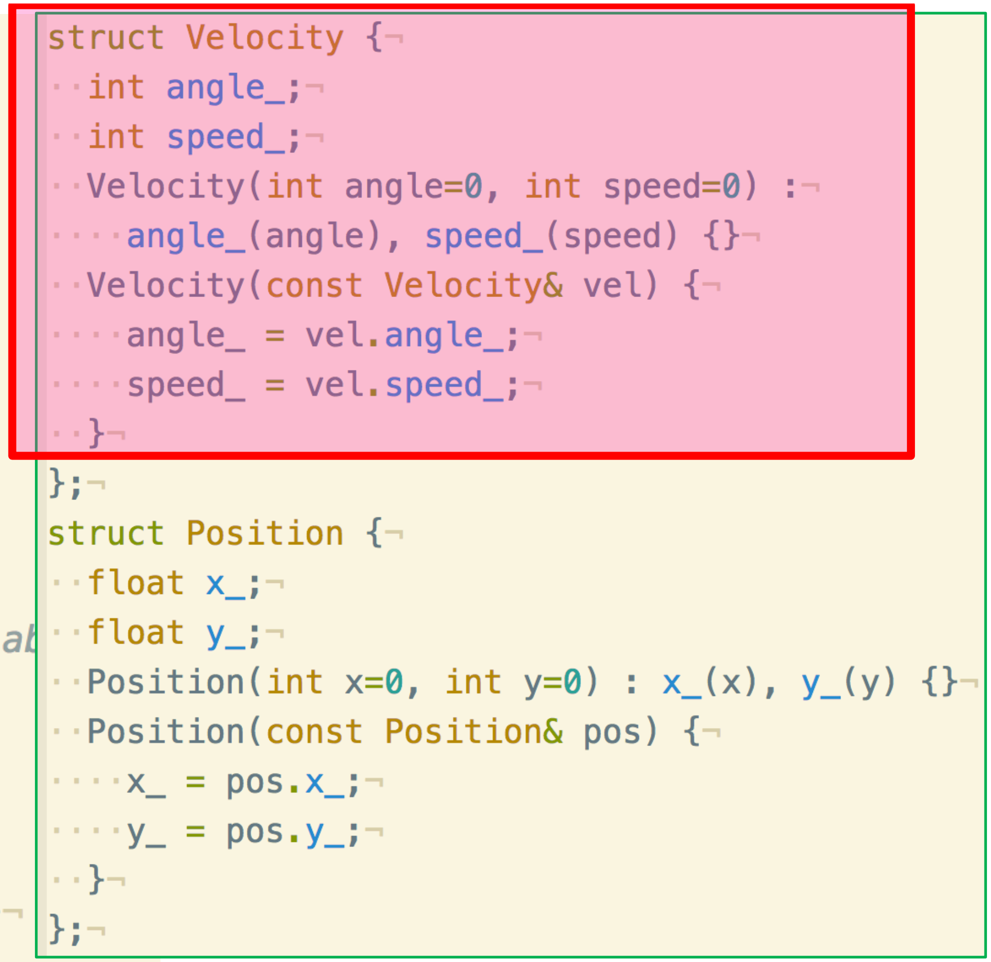
```
class Robot {
public:
  Robot() {
    velocity_
    position_
  }
  /
  virtual int UpdatePosition(int time)=0;
  virtual void Turn(float degrees) {

  }
  // setters and getters for private member variab
  void set_position(Position pos) {
    position_->x_ = pos.x_;
    position_->y_ = pos.y_;
  }
  Position get_position() { return *position_; }
  void set_velocity(Velocity vel) {
    velocity_->angle_ = vel.angle_;
    velocity_->speed_ = vel.speed_;
  }
  Velocity get_velocity() { return *velocity_; }
private:
  Position * position_;
  Velocity * velocity_;
protected:
  Sensor touch_sensor_;
};
```

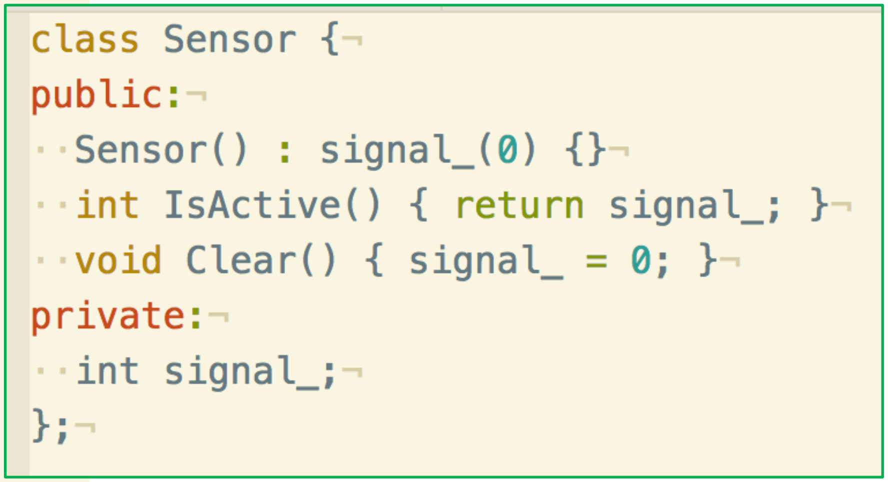
NOTE: structs are just like classes. Default is public, instead of private.

structs have constructors too!

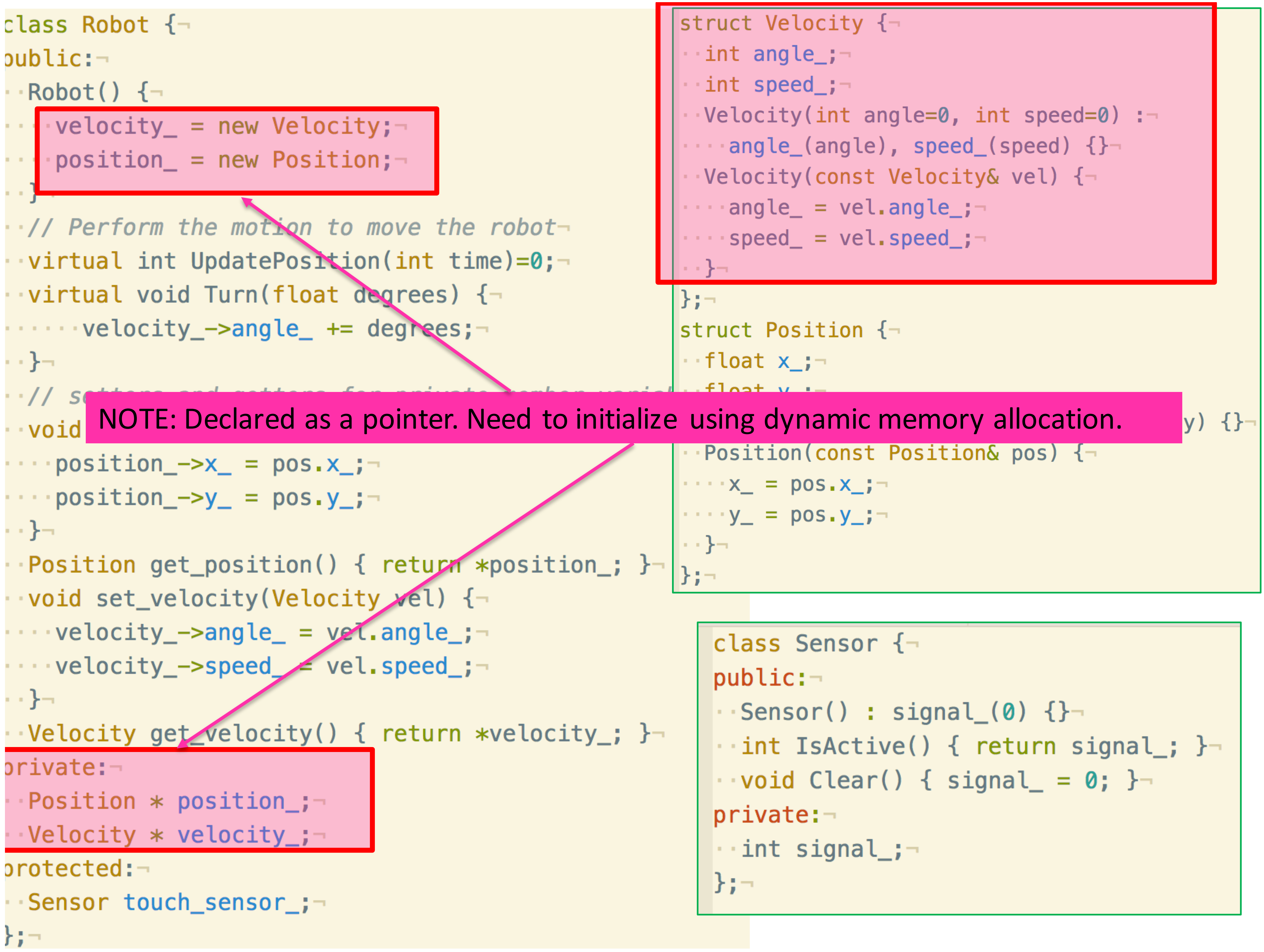
This is a copy constructor:
Velocity(const Velocity& vel);



```
struct Velocity {
  int angle_;
  int speed_;
  Velocity(int angle=0, int speed=0) :
    angle_(angle), speed_(speed) {}
  Velocity(const Velocity& vel) {
    angle_ = vel.angle_;
    speed_ = vel.speed_;
  }
};
struct Position {
  float x_;
  float y_;
  Position(int x=0, int y=0) : x_(x), y_(y) {}
  Position(const Position& pos) {
    x_ = pos.x_;
    y_ = pos.y_;
  }
};
```



```
class Sensor {
public:
  Sensor() : signal_(0) {}
  int IsActive() { return signal_; }
  void Clear() { signal_ = 0; }
private:
  int signal_;
};
```

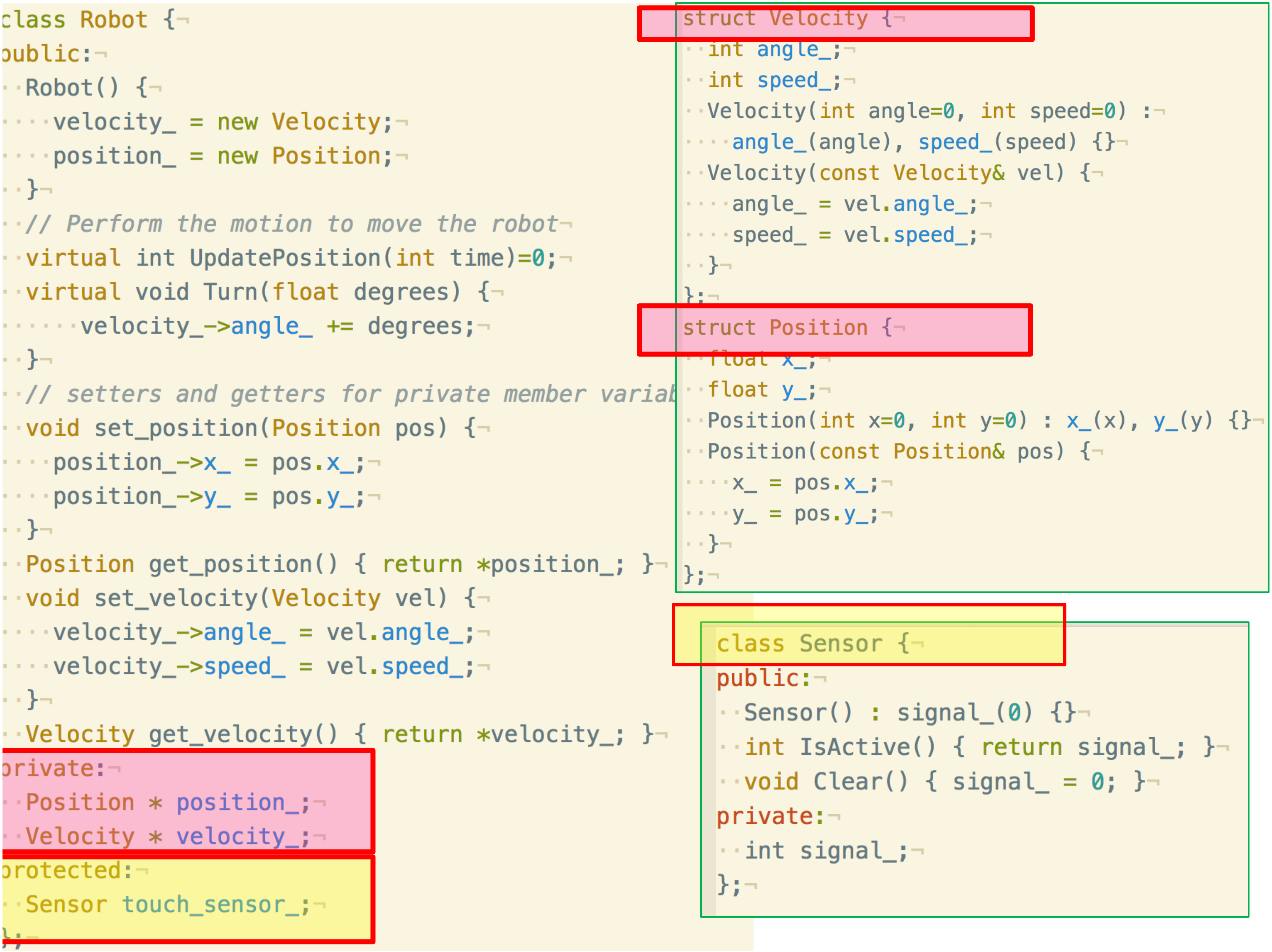


```
class Robot {
public:
  Robot() {
    velocity_ = new Velocity;
    position_ = new Position;
  }
  // Perform the motion to move the robot
  virtual int UpdatePosition(int time)=0;
  virtual void Turn(float degrees) {
      velocity_->angle_ += degrees;
  }
  // setters and getters for private member variabl
  void
    position_->x_ = pos.x_;
    position_->y_ = pos.y_;
  }
  Position get_position() { return *position_; }
  void set_velocity(Velocity vel) {
    velocity_->angle_ = vel.angle_;
    velocity_->speed_ = vel.speed_;
  }
  Velocity get_velocity() { return *velocity_; }
private:
  Position * position_;
  Velocity * velocity_;
protected:
  Sensor touch_sensor_;
};
```

```
struct Velocity {
  int angle_;
  int speed_;
  Velocity(int angle=0, int speed=0) :
    angle_(angle), speed_(speed) {}
  Velocity(const Velocity& vel) {
    angle_ = vel.angle_;
    speed_ = vel.speed_;
  }
};
struct Position {
  float x_;
  float y_;
                                        y) {}
  Position(const Position& pos) {
    x_ = pos.x_;
    y_ = pos.y_;
  }
};
```

```
class Sensor {
public:
  Sensor() : signal_(0) {}
  int IsActive() { return signal_; }
  void Clear() { signal_ = 0; }
private:
  int signal_;
};
```

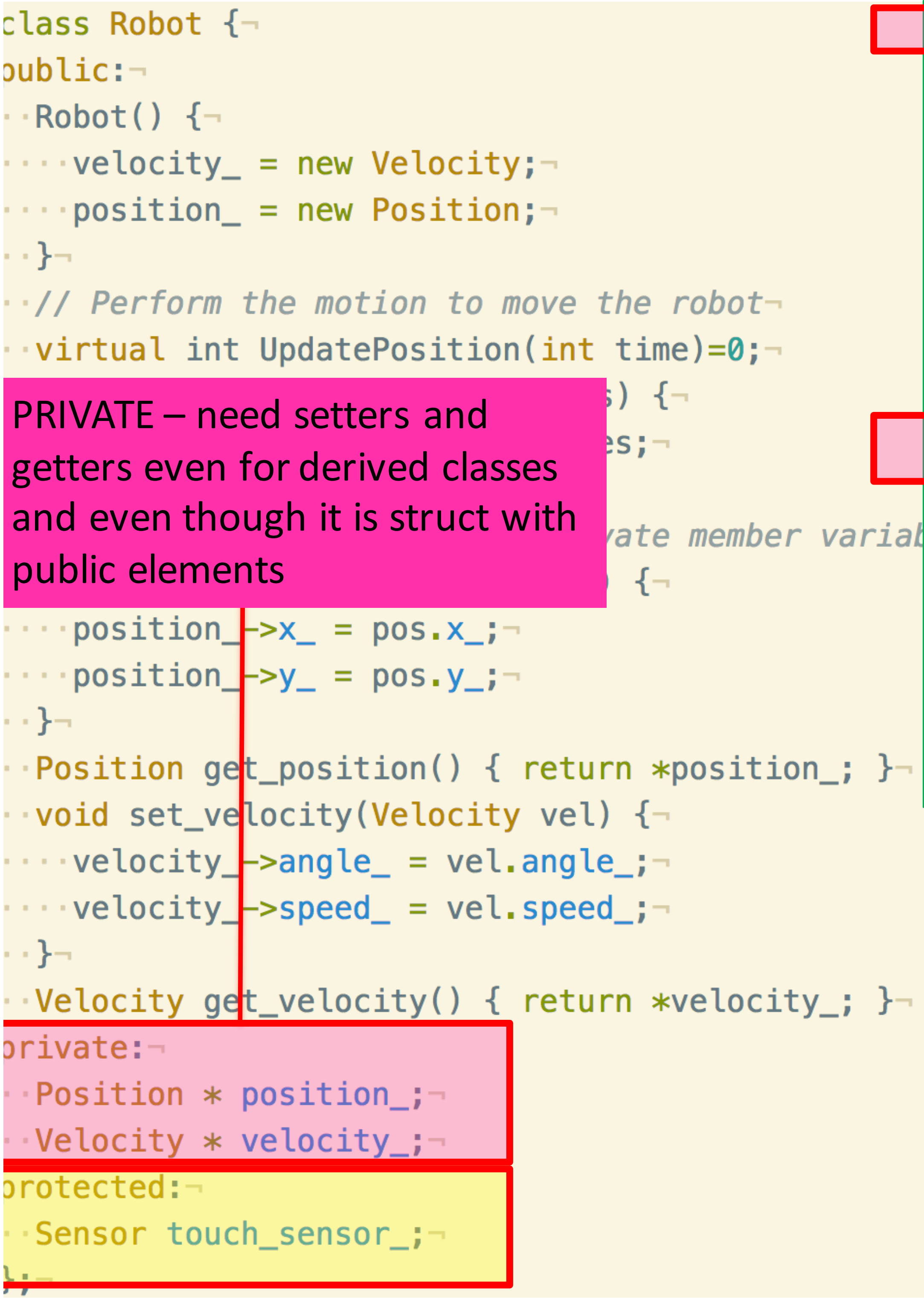
NOTE: Declared as a pointer. Need to initialize using dynamic memory allocation.



```
class Robot {
public:
  Robot() {
    velocity_ = new Velocity;
    position_ = new Position;
  }
  // Perform the motion to move the robot
  virtual int UpdatePosition(int time)=0;
  virtual void Turn(float degrees) {
      velocity_->angle_ += degrees;
  }
  // setters and getters for private member variab
  void set_position(Position pos) {
    position_->x_ = pos.x_;
    position_->y_ = pos.y_;
  }
  Position get_position() { return *position_; }
  void set_velocity(Velocity vel) {
    velocity_->angle_ = vel.angle_;
    velocity_->speed_ = vel.speed_;
  }
  Velocity get_velocity() { return *velocity_; }
private:
  Position * position_;
  Velocity * velocity_;
protected:
  Sensor touch_sensor_;
};
```

```
struct Velocity {
  int angle_;
  int speed_;
  Velocity(int angle=0, int speed=0) :
    angle_(angle), speed_(speed) {}
  Velocity(const Velocity& vel) {
    angle_ = vel.angle_;
    speed_ = vel.speed_;
  }
};
struct Position {
  float x_;
  float y_;
  Position(int x=0, int y=0) : x_(x), y_(y) {}
  Position(const Position& pos) {
    x_ = pos.x_;
    y_ = pos.y_;
  }
};
```

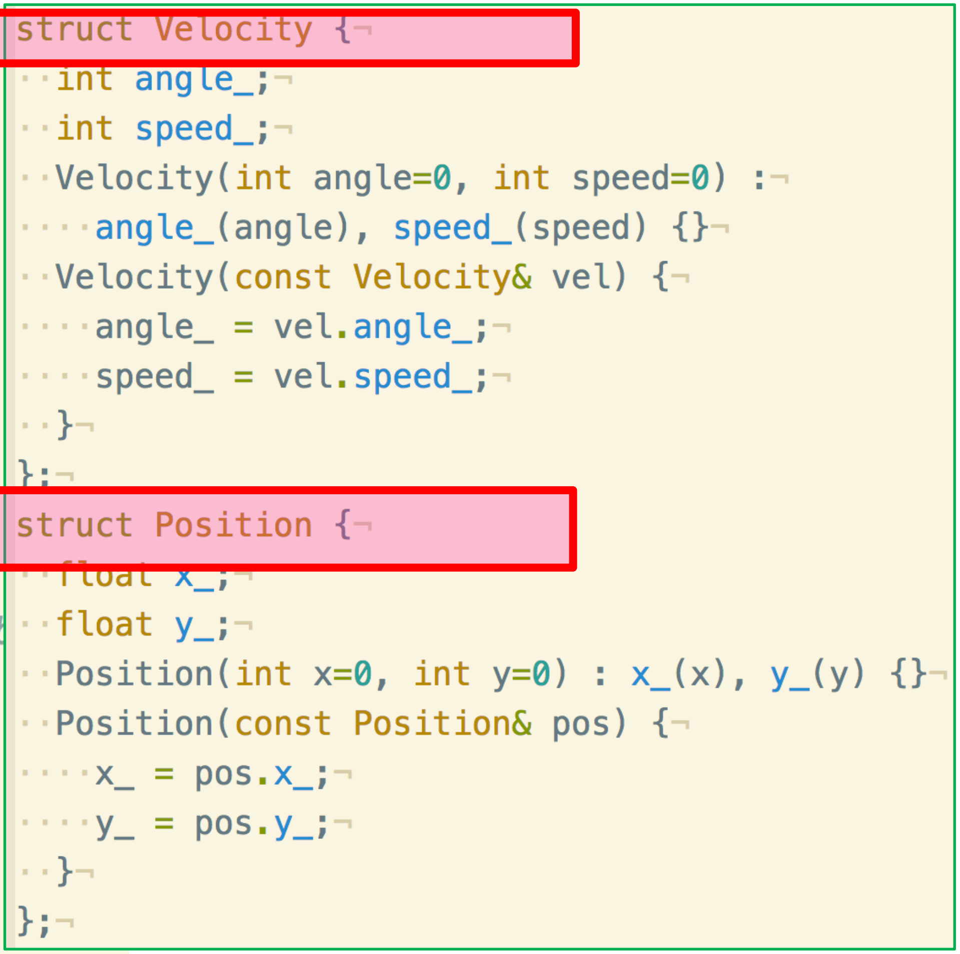
```
class Sensor {
public:
  Sensor() : signal_(0) {}
  int IsActive() { return signal_; }
  void Clear() { signal_ = 0; }
private:
  int signal_;
};
```



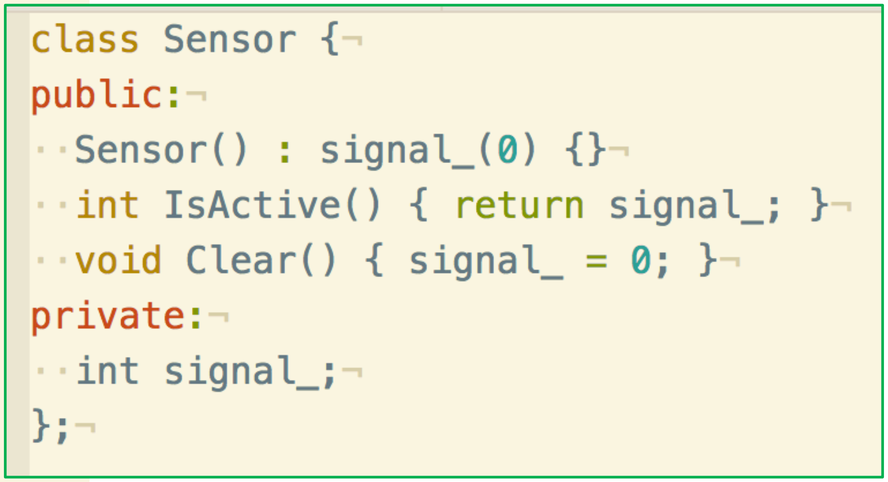
```
class Robot {
public:
  Robot() {
    velocity_ = new Velocity;
    position_ = new Position;
  }
  // Perform the motion to move the robot
  virtual int UpdatePosition(int time)=0;
                                       s) {
                                       es;

                                       vate member variab
                                       ) {
    position_->x_ = pos.x_;
    position_->y_ = pos.y_;
  }
  Position get_position() { return *position_; }
  void set_velocity(Velocity vel) {
    velocity_->angle_ = vel.angle_;
    velocity_->speed_ = vel.speed_;
  }
  Velocity get_velocity() { return *velocity_; }
private:
  Position * position_;
  Velocity * velocity_;
protected:
  Sensor touch_sensor_;
};
```

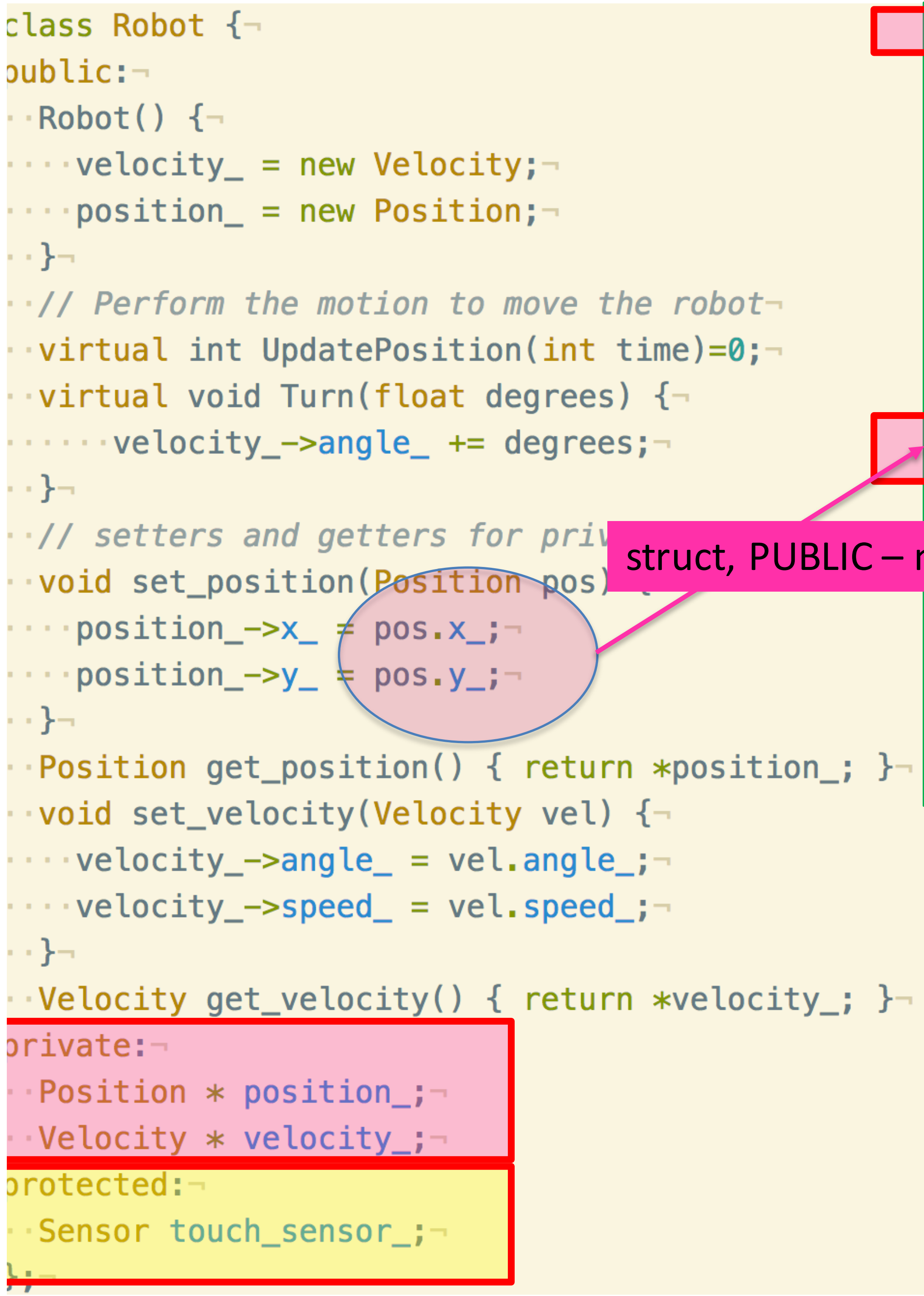
PRIVATE – need setters and getters even for derived classes and even though it is struct with public elements



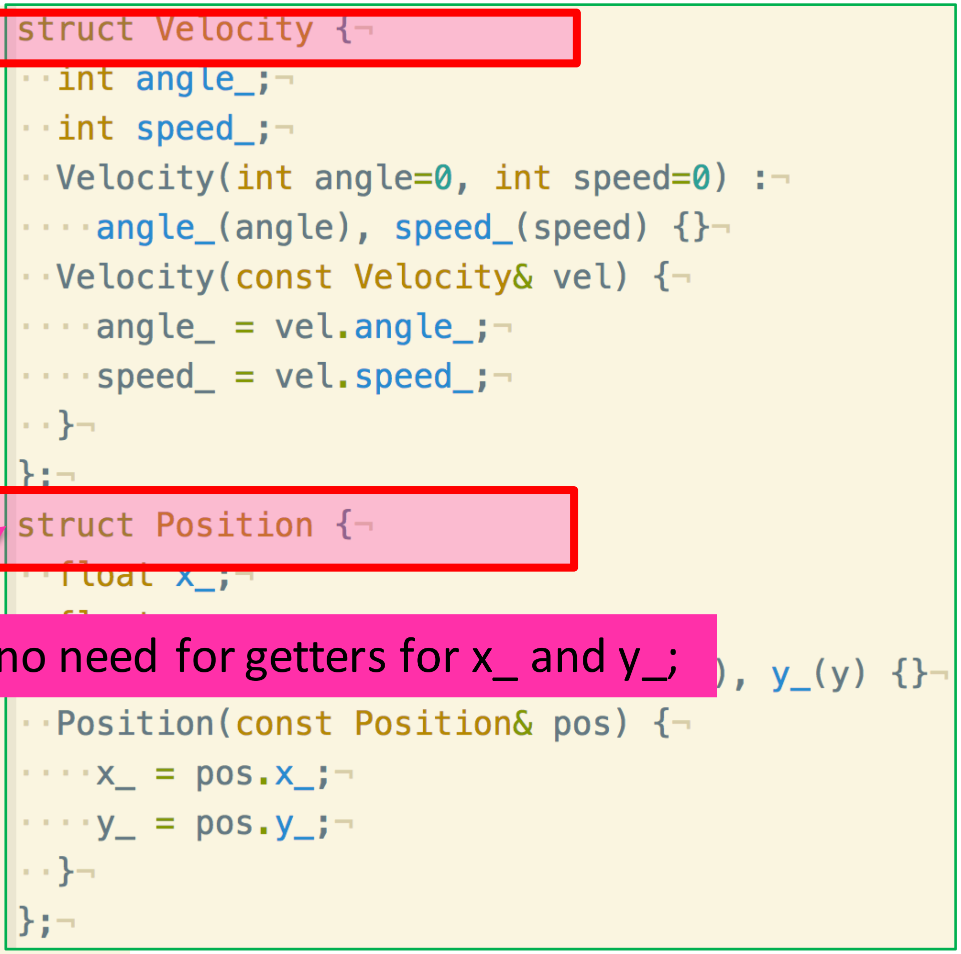
```
struct Velocity {
  int angle_;
  int speed_;
  Velocity(int angle=0, int speed=0) :
    angle_(angle), speed_(speed) {}
  Velocity(const Velocity& vel) {
    angle_ = vel.angle_;
    speed_ = vel.speed_;
  }
};
struct Position {
  float x_;
  float y_;
  Position(int x=0, int y=0) : x_(x), y_(y) {}
  Position(const Position& pos) {
    x_ = pos.x_;
    y_ = pos.y_;
  }
};
```



```
class Sensor {
public:
  Sensor() : signal_(0) {}
  int IsActive() { return signal_; }
  void Clear() { signal_ = 0; }
private:
  int signal_;
};
```

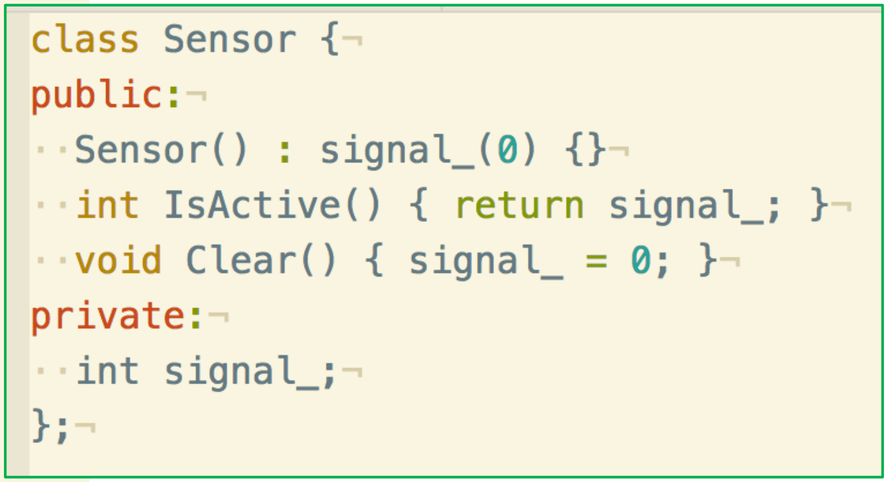


```
class Robot {
public:
  Robot() {
    velocity_ = new Velocity;
    position_ = new Position;
  }
  // Perform the motion to move the robot
  virtual int UpdatePosition(int time)=0;
  virtual void Turn(float degrees) {
      velocity_->angle_ += degrees;
  }
  // setters and getters for priv
  void set_position(Position pos)
    position_->x_ = pos.x_;
    position_->y_ = pos.y_;
  }
  Position get_position() { return *position_; }
  void set_velocity(Velocity vel) {
    velocity_->angle_ = vel.angle_;
    velocity_->speed_ = vel.speed_;
  }
  Velocity get_velocity() { return *velocity_; }
private:
  Position * position_;
  Velocity * velocity_;
protected:
  Sensor touch_sensor_;
};
```

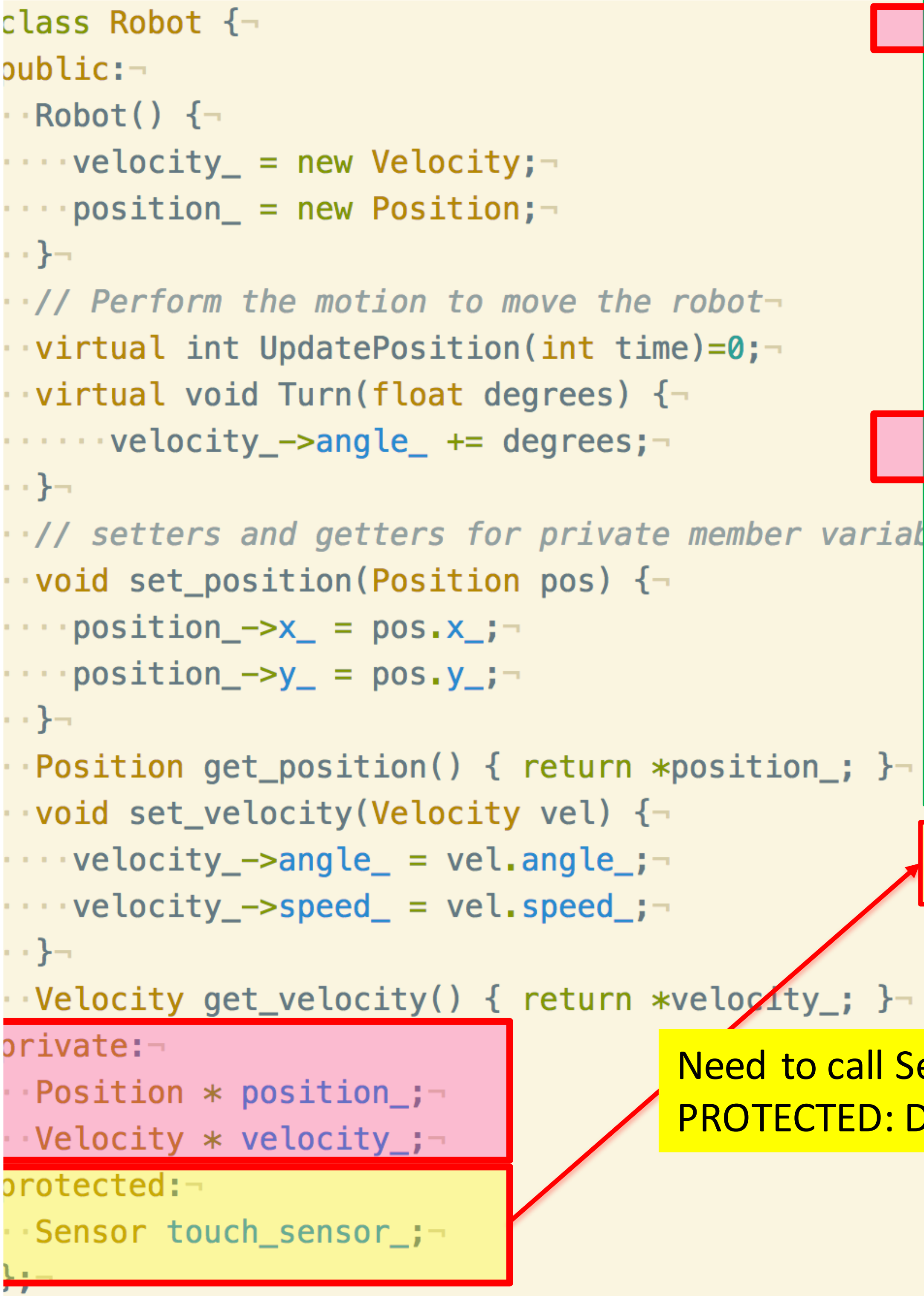


```
struct Velocity {
  int angle_;
  int speed_;
  Velocity(int angle=0, int speed=0) :
    angle_(angle), speed_(speed) {}
  Velocity(const Velocity& vel) {
    angle_ = vel.angle_;
    speed_ = vel.speed_;
  }
};
struct Position {
  float x_;
  float
                                       ), y_(y) {}
  Position(const Position& pos) {
    x_ = pos.x_;
    y_ = pos.y_;
  }
};
```

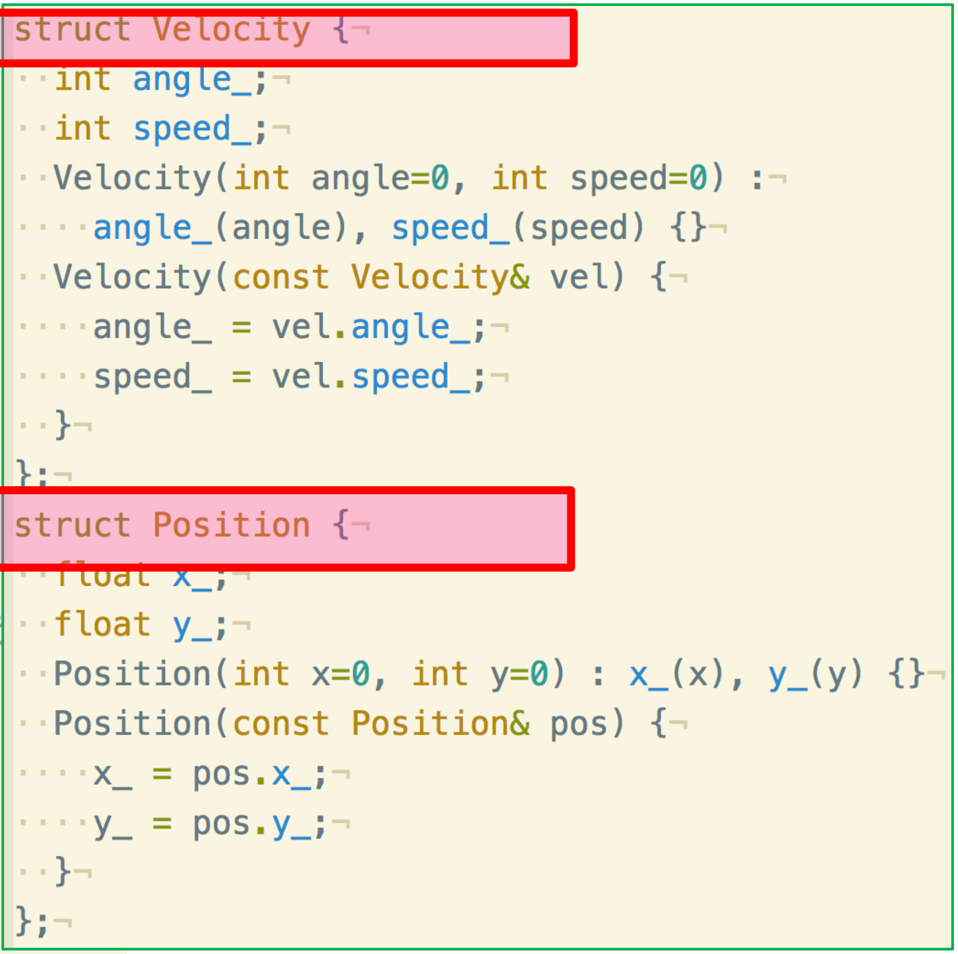
struct, PUBLIC – no need for getters for x_ and y_;



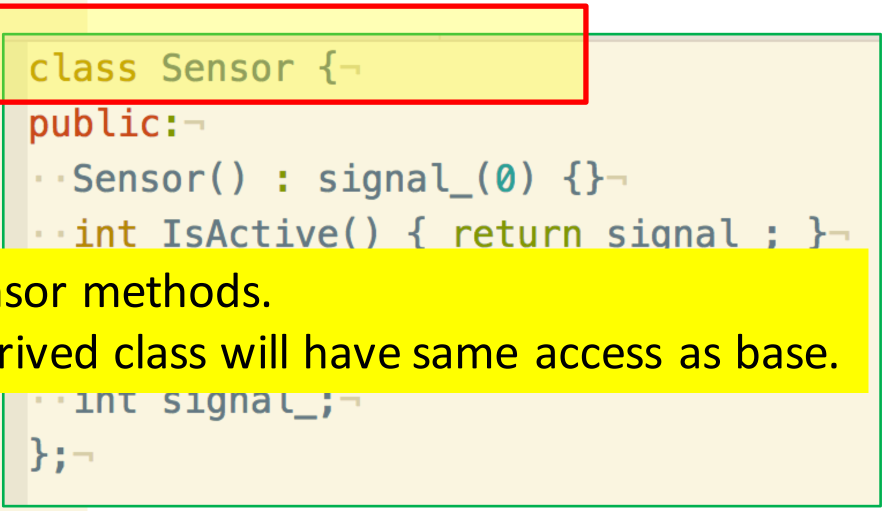
```
class Sensor {
public:
  Sensor() : signal_(0) {}
  int IsActive() { return signal_; }
  void Clear() { signal_ = 0; }
private:
  int signal_;
};
```



```
class Robot {
public:
  Robot() {
    velocity_ = new Velocity;
    position_ = new Position;
  }
  // Perform the motion to move the robot
  virtual int UpdatePosition(int time)=0;
  virtual void Turn(float degrees) {
      velocity_->angle_ += degrees;
  }
  // setters and getters for private member variab
  void set_position(Position pos) {
    position_->x_ = pos.x_;
    position_->y_ = pos.y_;
  }
  Position get_position() { return *position_; }
  void set_velocity(Velocity vel) {
    velocity_->angle_ = vel.angle_;
    velocity_->speed_ = vel.speed_;
  }
  Velocity get_velocity() { return *velocity_; }
private:
  Position * position_;
  Velocity * velocity_;
protected:
  Sensor touch_sensor_;
};
```



```
struct Velocity {
  int angle_;
  int speed_;
  Velocity(int angle=0, int speed=0) :
    angle_(angle), speed_(speed) {}
  Velocity(const Velocity& vel) {
    angle_ = vel.angle_;
    speed_ = vel.speed_;
  }
};
struct Position {
  float x_;
  float y_;
  Position(int x=0, int y=0) : x_(x), y_(y) {}
  Position(const Position& pos) {
    x_ = pos.x_;
    y_ = pos.y_;
  }
};
```

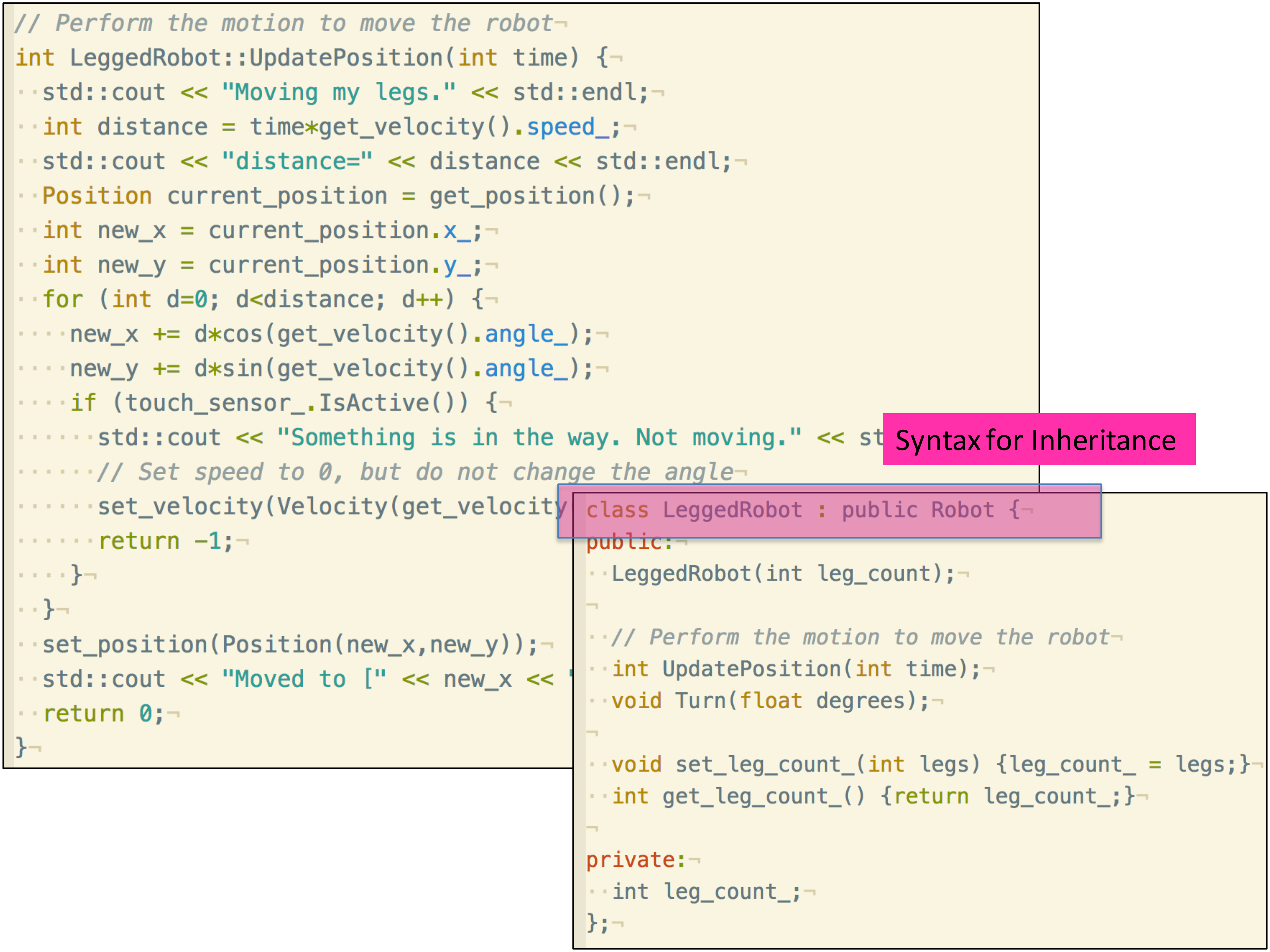


```
class Sensor {
public:
  Sensor() : signal_(0) {}
  int IsActive() { return signal_; }

  int signal_;
};
```

Need to call Sensor methods.

PROTECTED: Derived class will have same access as base.



```
// Perform the motion to move the robot
int LeggedRobot::UpdatePosition(int time) {
  std::cout << "Moving my legs." << std::endl;
  int distance = time*get_velocity().speed_;
  std::cout << "distance=" << distance << std::endl;
  Position current_position = get_position();
  int new_x = current_position.x_;
  int new_y = current_position.y_;
  for (int d=0; d<distance; d++) {
    new_x += d*cos(get_velocity().angle_);
    new_y += d*sin(get_velocity().angle_);
    if (touch_sensor_.IsActive()) {
      std::cout << "Something is in the way. Not moving." << st
      // Set speed to 0, but do not change the angle
      set_velocity(Velocity(get_velocity
      return -1;
    }
  }
  set_position(Position(new_x,new_y));
  std::cout << "Moved to [" << new_x <<
  return 0;
}
```

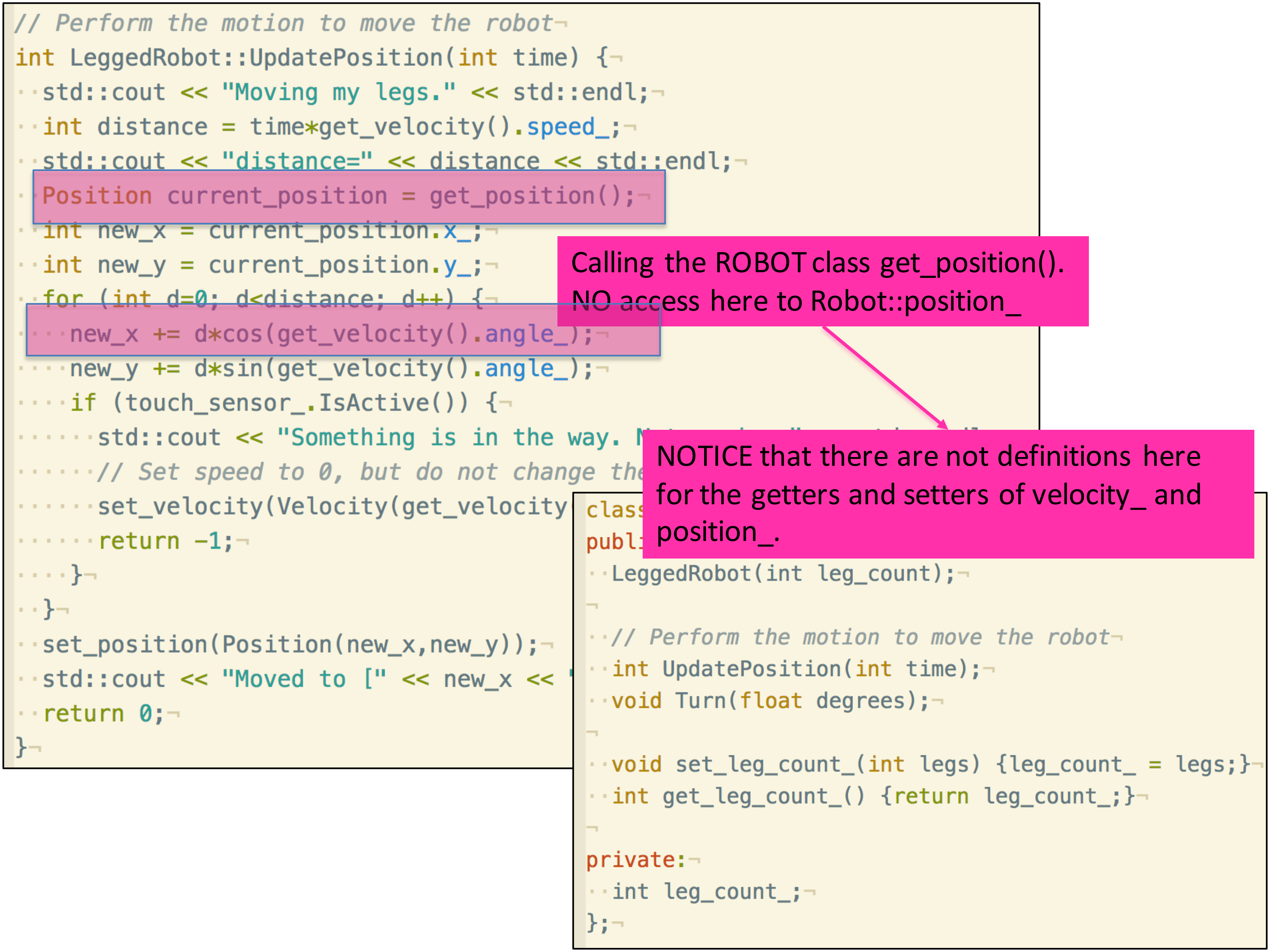
Syntax for Inheritance

```
class LeggedRobot : public Robot {
public:
  LeggedRobot(int leg_count);

  // Perform the motion to move the robot
  int UpdatePosition(int time);
  void Turn(float degrees);

  void set_leg_count_(int legs) {leg_count_ = legs;}
  int get_leg_count_() {return leg_count_;}

private:
  int leg_count_;
};
```



```
// Perform the motion to move the robot
int LeggedRobot::UpdatePosition(int time) {
  std::cout << "Moving my legs." << std::endl;
  int distance = time*get_velocity().speed_;
  std::cout << "distance=" << distance << std::endl;
  Position current_position = get_position();
  int new_x = current_position.x_;
  int new_y = current_position.y_;
  for (int d=0; d<distance; d++) {
    new_x += d*cos(get_velocity().angle_);
    new_y += d*sin(get_velocity().angle_);
    if (touch_sensor_.IsActive()) {
      std::cout << "Something is in the way. N
      // Set speed to 0, but do not change the
      set_velocity(Velocity(get_velocity
      return -1;
    }
  }
  set_position(Position(new_x,new_y));
  std::cout << "Moved to [" << new_x <<
  return 0;
}
```

Calling the ROBOT class get_position().
NO access here to Robot::position_

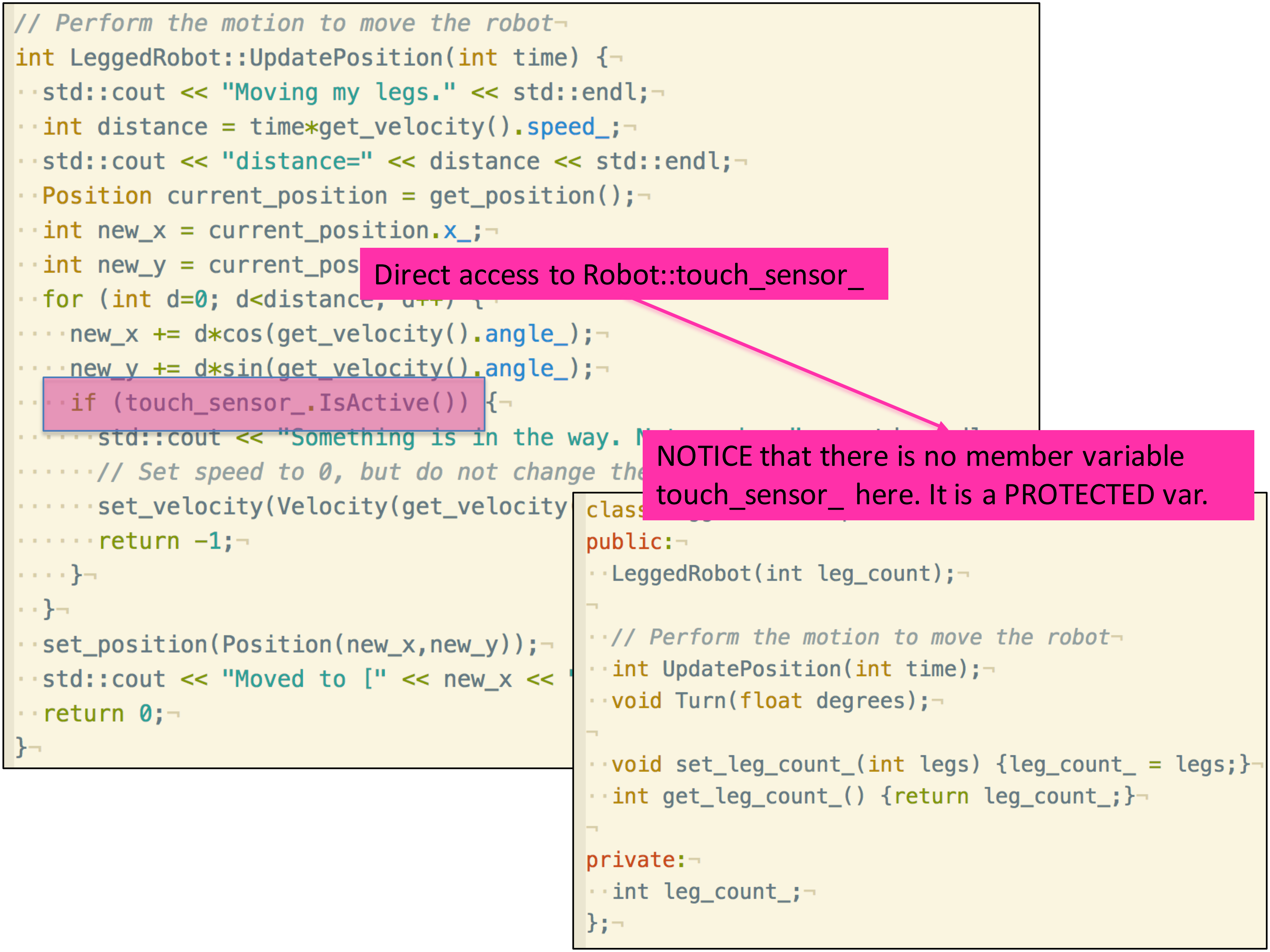
NOTICE that there are not definitions here for the getters and setters of velocity_ and position_.

```
class
publi
  LeggedRobot(int leg_count);

  // Perform the motion to move the robot
  int UpdatePosition(int time);
  void Turn(float degrees);

  void set_leg_count_(int legs) {leg_count_ = legs;}
  int get_leg_count_() {return leg_count_;}

private:
  int leg_count_;
};
```



```
// Perform the motion to move the robot
int LeggedRobot::UpdatePosition(int time) {
  std::cout << "Moving my legs." << std::endl;
  int distance = time*get_velocity().speed_;
  std::cout << "distance=" << distance << std::endl;
  Position current_position = get_position();
  int new_x = current_position.x_;
  int new_y = current_pos
  for (int d=0; d<distance; d++) {
    new_x += d*cos(get_velocity().angle_);
    new_y += d*sin(get_velocity().angle_);
    if (touch_sensor_.IsActive()) {
      std::cout << "Something is in the way. N
      // Set speed to 0, but do not change the
      set_velocity(Velocity(get_velocity
      return -1;
    }
  }
  set_position(Position(new_x,new_y));
  std::cout << "Moved to [" << new_x <<
  return 0;
}
```

Direct access to Robot::touch_sensor_

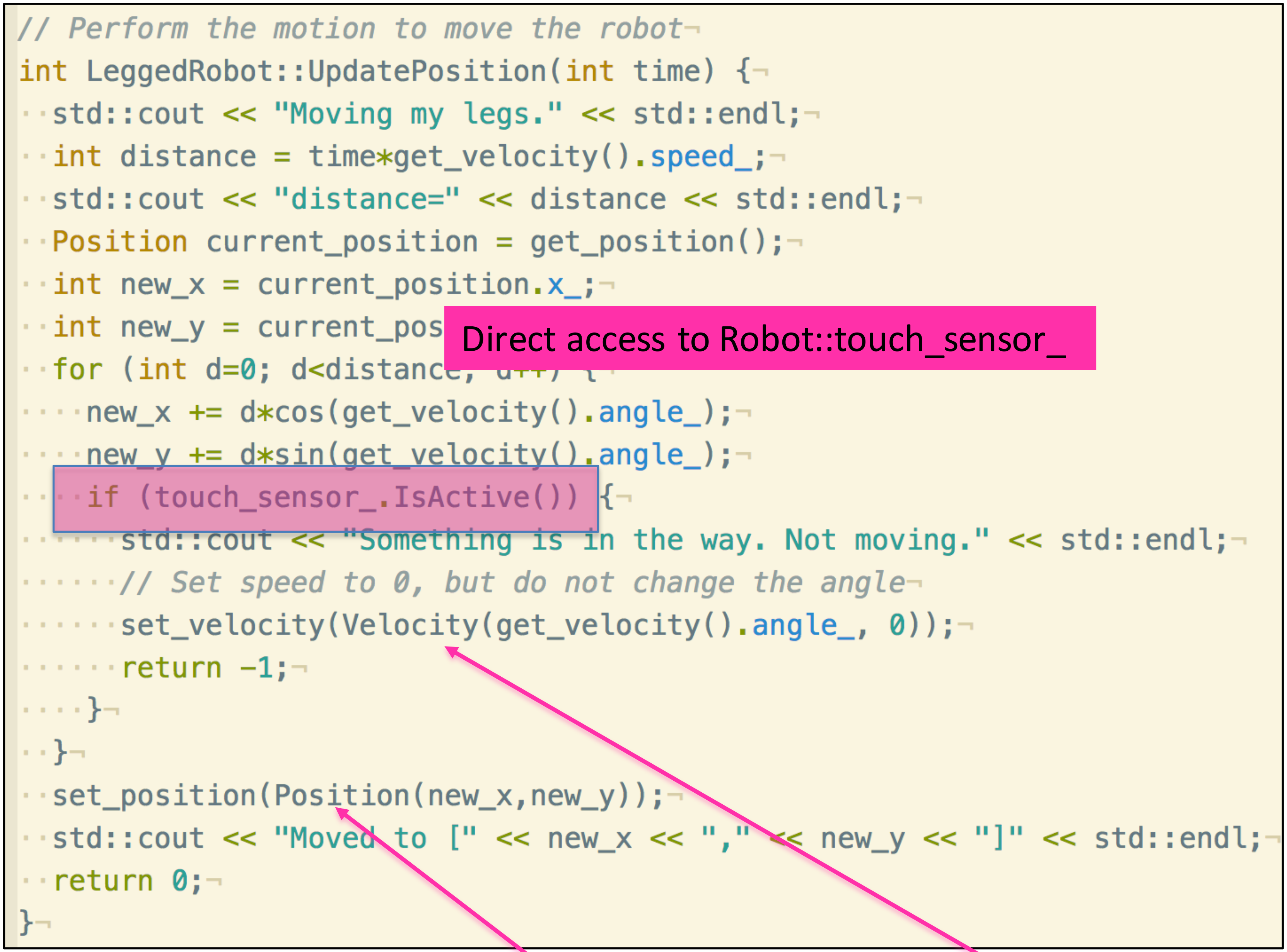
```
class
public:
  LeggedRobot(int leg_count);

  // Perform the motion to move the robot
  int UpdatePosition(int time);
  void Turn(float degrees);

  void set_leg_count_(int legs) {leg_count_ = legs;}
  int get_leg_count_() {return leg_count_;}

private:
  int leg_count_;
};
```

NOTICE that there is no member variable touch_sensor_ here. It is a PROTECTED var.



```cpp
// Perform the motion to move the robot
int LeggedRobot::UpdatePosition(int time) {
  std::cout << "Moving my legs." << std::endl;
  int distance = time*get_velocity().speed_;
  std::cout << "distance=" << distance << std::endl;
  Position current_position = get_position();
  int new_x = current_position.x_;
  int new_y = current_pos
  for (int d=0; d<distance; d++) {
    new_x += d*cos(get_velocity().angle_);
    new_y += d*sin(get_velocity().angle_);
    if (touch_sensor_.IsActive()) {
      std::cout << "Something is in the way. Not moving." << std::endl;
      // Set speed to 0, but do not change the angle
      set_velocity(Velocity(get_velocity().angle_, 0));
      return -1;
    }
  }
  set_position(Position(new_x,new_y));
  std::cout << "Moved to [" << new_x << "," << new_y << "]" << std::endl;
  return 0;
}
```

Direct access to Robot::touch_sensor_

NOTICE that we are setting position_ and velocity_ by creating new objects. Maybe we want to override the setters to set elements of the struct?

# Composition and Inheritance

- HAS-A : composition (embed in another class)
- IS-A : inheritance

- Composition and Inheritance does not get you around "private"
- Subclasses contain a base class object. If the base class element is "protected," then nothing hidden.
- Access goes from the outside in.
- Subclasses can redefine variables (then they both exist!) and override functions.