

14VIRTUAL

Contents

1	CMakeLists.txt	1
2	examples.ml	1
3	src/animals1.hxx	3
4	src/animals1.cxx	5
5	src/animals2.hxx	8
6	src/animals2.cxx	9
7	src/animals3.hxx	11
8	src/animals3.cxx	13
9	src/Sprite_tree.hxx	16
10	src/Sprite_tree.cxx	18
11	test/mock_sprite_set.hxx	21
12	test/mock_sprite_set.cxx	21
13	test/sprite_tree_test.cxx	22

1 CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.3)
2 project(lec14 CXX)
3 include(.cs211/cmake/CMakeLists.txt)
4
5 add_program(animals1 src/animals1.cxx)
6 add_program(animals2 src/animals2.cxx)
7 add_program(animals3 src/animals3.cxx)
8
9 add_test_program(sprite_tree_test
10                 test/mock_sprite_set.cxx
```

```
11      test/sprite_tree_test.cxx
12      src/Sprite_tree.cxx)
13 target_link_libraries(sprite_tree_test ge211)
```

2 examples.ml

```
1  (* Some examples of parametric polymorphism in OCaml. *)
2
3  (* Reverses a list. *)
4  let reverse xs0 =
5    let rec loop acc xs =
6      match xs with
7      | []          -> acc
8      | x :: xs'   -> loop (x :: acc) xs'
9    in loop [] xs0
10 (* Inferred type scheme:
11    reverse : 'a list -> 'a list *)
12
13 (* Sums a list of ints. *)
14 let sum xs0 =
15   let rec loop acc = function
16     | []          -> acc
17     | x :: xs    -> loop (x + acc) xs
18   in loop 0 xs0
19 (* Inferred type scheme:
20    sum : int list -> int *)
21
22 (* Sums a list of floats. *)
23 let sum_float xs0 =
24   let rec loop acc = function
25     | []          -> acc
26     | x :: xs    -> loop (x +. acc) xs
27   in loop 0.0 xs0
28 (* Inferred type scheme:
29    sum_float : float list -> float *)
30
31 (* Reduces a list, left associatively. *)
32 let foldl (plus, zero, xs0) =
33   let rec loop acc = function
34     | []          -> acc
35     | x :: xs    -> loop (plus (x, acc)) xs
36   in loop zero xs0
37 (* Inferred type scheme:
```

3 src/animals1.hxx

```
38     foldl : (('a * 'b -> 'b) * 'b * 'a list) -> 'b *)
39
40 (* Reduces a list, left associatively. *)
41 let foldl_curried plus zero xs0 =
42     let rec loop acc = function
43     | []      -> acc
44     | x :: xs -> loop (plus x acc) xs
45     in loop zero xs0
46 (* Inferred type scheme:
47     foldl : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b *)
48
49 (* Sorts a list according to the given binary predicate. *)
50 let rec merge_sort (<) xs0 =
51
52     let split =
53         foldl_curried (fun x (xs1, xs2) -> (x :: xs2, xs1)) ([], []) in
54
55     let rec merge xs1 xs2 = match xs1, xs2 with
56     | x1 :: xs1', x2 :: xs2' ->
57         if x1 < x2
58         then x1 :: merge xs1' xs2
59         else x2 :: merge xs1 xs2'
60     | [], _ -> xs2
61     | _, [] -> xs1 in
62
63     let rec sort xs =
64         match xs with
65         | [] | [_] -> xs
66         | _ ->
67             let (xs1, xs2) = split xs in
68                 merge (sort xs1) (sort xs2) in
69
70     sort xs0
71 (* Inferred type scheme:
72     merge_sort : ('a -> 'a -> bool) -> 'a list -> 'a list *)
```

3 src/animals1.hxx

```
1  ////
2  //// START HERE
3  ////
4
5  #include <string>
```

```

6
7 ///
8 /// Inheritance
9 ///
10
11 /*
12  * C++ provides a mechanism for defining families of classes that are
13  * similar in some ways and different in others. For example, suppose
14  * we want classes to represent different kinds of pet animal, which
15  * have operations like sleeping and playing. We can combine their
16  * common behavior into a base class and then derive the specific
17  * animal classes from that.
18  *
19  * Let's call the base class Animal. We'll initially define it as
20  * follows:
21  */
22
23 class Animal
24 {
25 public:
26     Animal(const std::string& name, unsigned int weight);
27
28     void eat(unsigned int amount);
29     void play();
30
31     const std::string& get_name() const { return name_; }
32     unsigned int get_weight() const { return weight_; }
33
34 private:
35     std::string name_;
36     unsigned int weight_;
37 };
38
39 /*
40  * That is, every animal has a name and a weight, and has operations to
41  * eat and to play. The implementations of the operations are in
42  * animals1.cpp.
43  *
44  * We then derive other classes from the `Animal` class. Deriving
45  * makes the new class like the old class, but then lets us add to or
46  * change it. Here's `Cat`:
47  */
48
49 class Cat : public Animal
50 {

```

```

51 public:
52     Cat(const std::string& name);
53     void speak();
54 };
55
56 /*
57  * The notation `: public Animal` means that `Cat` starts out as a copy
58  * of `Animal`, with all of `Animal`'s public members public in `Cat` as
59  * well. We then define a constructor, which takes a name and passes it
60  * on to the base class's constructor, along with a fixed weight of 10.
61  *
62  * `Cat` also has a new member function, `speak`, that `Animal` does not
63  * have.
64  *
65  * We also define `Dog` by deriving from `Animal`, but unlike `Cat`,
66  * `Dog`'s playing behavior is a bit more complicated. In particular,
67  * each `Dog` has some number of bones, and each time we play the dog
68  * gets a new bone. And then when the dog speaks, it says "woof"
69  * repeatedly, once for each of its bones. Here's the definition of the
70  * `Dog` class:
71  */
72
73 class Dog : public Animal
74 {
75 public:
76     Dog(const std::string& name);
77     void speak();
78     void play();
79
80 private:
81     unsigned int nbones_;
82 };
83
84 /*
85  * As you can see, `Dog` adds a member variable `nbones_` to `Animal`,
86  * adds a member function `speak`, and replaces the base class's
87  * `Animal::play` function with a `Dog`-specific `play` function. The
88  * `Dog::play` function delegates to `Animal::play` to do the playing
89  * and also increments `nbones_`.
90  *
91  * Now we can write a program involving cats and dogs. See the bottom of
92  * animals1.cpp for main and an introduction to inheritance polymorphism.
93  */

```

4 src/animals1.cxx

```

1  //////
2  ////// SEE animals1.h first
3  //////
4
5  // The implementations are here, but the implementations aren't very
6  // interesting.
7
8  #include "animals1.hxx"
9  #include <iostream>
10
11 Animal::Animal(const std::string& name, unsigned int weight)
12     : name_(name), weight_(weight)
13 { }
14
15 void Animal::eat(unsigned int amount)
16 {
17     weight_ += amount;
18 }
19
20 void Animal::play()
21 {
22     std::cout << get_name() << " plays.\n";
23 }
24
25 // The `Cat` constructor passes its the name to the `Animal` constructor
26 // along with a starting cat weight.
27 Cat::Cat(const std::string& name) : Animal(name, 10)
28 { }
29
30 void Cat::speak()
31 {
32     std::cout << get_name() << " says meow!\n";
33 }
34
35 // The `Dog` constructor has to initialize both the base class `Animal` and
36 // its own member `nbones_`.
37 Dog::Dog(const std::string& name)
38     : Animal{name, 50}
39     , nbones_{1}
40 { }
41
42 void Dog::speak()

```

```

43 {
44     std::cout << get_name() << " says";
45     for (size_t i = 0; i < nbones_; ++i)
46         std::cout << " woof";
47     std::cout << "!\n";
48 }
49
50 // `Dog`'s `play` behavior is different from the base `Animal`'s.
51 void Dog::play()
52 {
53     Animal::play();
54     ++nbones_;
55 }
56
57 // Forward declaration, in case we want to call `play_twice` from `main`.
58 void play_twice(Animal&);
59
60 int main()
61 {
62     // My mom has four cats and two dogs.
63     Dog willie("Willie");
64     Cat vinny("Vinny");
65     Cat francie("Francie");
66
67     willie.play();
68     vinny.play();
69     francie.eat(2);
70 }
71
72 /*
73  * An additional feature of inheritance is a form of polymorphism.
74  * Polymorphism is when the same variable (or more abstractly, entity)
75  * can come in different forms. In this case, the polymorphism is that
76  * a reference or pointer whose type says it's of the base class can
77  * actually refer to an object of any derived class. Here we have a
78  * function that takes an `Animal` reference:
79  */
80
81 void play_twice(Animal& an)
82 {
83     an.play();
84     an.play();
85 }
86
87 /*

```

5 src/animals2.hxx

```
88  * We can use the reference according to the public interface of
89  * `Animal`, but at run time, the actual object the reference points to
90  * is allowed to be any derived class of `Animal`:
91  *
92  *   play_twice(willie);
93  *   play_twice(vinny);
94  *
95  * There are two problems with this, however:
96  *
97  * Because we made `speak` different for the two derived classes and
98  * didn't include it in the base class, we can't call it via a base
99  * class (`Animal`) reference.
100 *
101 * When we call a function on the base class reference, it uses the
102 * base class version of the function. That is, when we say `an.play()`,
103 * it uses the `Animal::play` member function rather than `Dog::play`,
104 * even when `an` is a `Dog`. In particular, the `nbones_` member
105 * variable won't be incremented when we call `play_twice` on the `Dog`
106 * `willie`, even though `Dog::play` increments `nbones_`.
107 *
108 * For the solution to these problems, see animals2.h.
109 */
```

5 src/animals2.hxx

```
1  #include <string>
2
3  ///
4  /// Polymorphism
5  ///
6
7  /*
8   * The solution to the problem at the end of animals2.cpp is a *virtual*
9   * function. Declaring a function virtual in the base class means
10  * that we expect the derived classes to *override* it, and we want to
11  * get the overridden derived class behavior, even with working with a
12  * base class reference. To do this, we change the declaration of
13  * Animal::play to be virtual:
14  */
15  class Animal
16  {
17  public:
18      Animal(const std::string& name, unsigned int weight);
```



```

19
20 void eat(unsigned int amount);
21 virtual void play(); // <<< CHANGE IS HERE
22
23 const std::string& get_name() const { return name_; }
24 unsigned int get_weight() const { return weight_; }
25
26 // When using inheritance, the base class must have a virtual
27 // destructor, even if it doesn't have to do anything.
28 virtual ~Animal() { }
29
30 private:
31     std::string name_;
32     unsigned int weight_;
33 };
34
35 /*
36  * The `Cat` class remains the same as before because we want `Cat` to have
37  * the default `play` behavior inherited from `Animal`.
38  */
39 class Cat : public Animal
40 {
41 public:
42     Cat(const std::string& name);
43     void speak();
44 };
45
46 /*
47  * Then in `Dog`, we redefine `play` and indicate that we are overriding
48  * the base definition of `play`:
49  */
50 class Dog : public Animal
51 {
52 public:
53     Dog(const std::string& name);
54     void speak();
55     void play() override; // <<< OTHER CHANGE IS HERE
56
57 private:
58     unsigned int nbones_;
59 };
60
61 /*
62  * Now when we use a `Dog` object via an `Animal&` reference, it uses
63  * `Dog::play` rather than `Animal::play`.

```

6 src/animals2.cxx

```
64 *
65 * This is polymorphic because the `Animal&` does not necessarily refer
66 * to an `Animal` object, but different kinds of derived classes with
67 * potentially different behaviors.
68 *
69 * Continue in animals3.h.
70 */
```

6 src/animals2.cxx

```
1 // The implementations are here, but the implementations aren't very
2 // interesting, since they're the same as in animals1.cpp. You probably
3 // want animals2.h or if you've been there already, animals3.h.
4
5 #include "animals2.hxx"
6 #include <iostream>
7
8 Animal::Animal(const std::string& name, unsigned int weight)
9     : name_(name), weight_(weight)
10 { }
11
12 void Animal::eat(unsigned int amount)
13 {
14     weight_ += amount;
15 }
16
17 void Animal::play()
18 {
19     std::cout << get_name() << " plays.\n";
20 }
21
22 Cat::Cat(const std::string& name) : Animal(name, 10)
23 { }
24
25 void Cat::speak()
26 {
27     std::cout << get_name() << " says meow!\n";
28 }
29
30 Dog::Dog(const std::string& name)
31     : Animal{name, 50}
32     , nbones_{1}
33 { }
```

```

34
35 void Dog::speak()
36 {
37     std::cout << get_name() << " says";
38     for (size_t i = 0; i < nbones_; ++i)
39         std::cout << " woof";
40     std::cout << "\n";
41 }
42
43 void Dog::play()
44 {
45     Animal::play();
46     ++nbones_;
47 }
48
49 // This function uses inheritance polymorphism, since it takes an `Animal`
50 // reference that might refer to a derived class of `Animal` such as `Dog` or
51 // `Cat`. Further, because `Animal::play` is virtual, it will use the derived
52 // class's version of `play` if the derived class overrides it.
53 void play_twice(Animal& an)
54 {
55     an.play();
56     an.play();
57 }
58
59 /*
60 * We can't do the same for `speak` because `Animal` doesn't define a `speak`
61 * function, so there's no guarantee that a particular `Animal&` will be from
62 * a class that defines `speak`.
63 *
64 *     void speak_twice(Animal& an)
65 *     {
66 *         an.speak();
67 *         an.speak();
68 *     }
69 */
70
71 int main()
72 {
73     Dog willie("Willie");
74     Cat vinny("Vinny");
75     Cat francie("Francie");
76
77     play_twice(willie);
78     play_twice(vinny);

```

```

79     francie.eat(2);
80     francie.speak();
81     willie.speak();
82 }

```

7 src/animals3.hxx

```

1  #include <string>
2
3  ///
4  /// Pure virtual functions
5  ///
6
7  /*
8   * We did not define a `speak` function in the `Animal` class, since
9   * that function is different for the derived classes, and so there was
10  * no sharing to be had. But this prevents us from calling `speak` on an
11  * `Animal&`, even if the object referred to is actually a `Cat` or
12  * `Dog`. It has to be this way because there's no guarantee of every
13  * derived class of `Animal` will define `speak`. We can fix this by
14  * declaring in `Animal` that every derived class of `Animal` must
15  * define `speak`. We do this by declaring `speak` to be pure virtual.
16  */
17
18  class Animal
19  {
20  public:
21     Animal(const std::string& name, unsigned int weight);
22
23     void eat(unsigned int amount);
24     virtual void play();
25     virtual void speak() = 0; // << NEW LINE HERE
26
27     /*
28      * The `= 0` means that `Animal` will not define `speak`, but that
29      * its derived classes must. In particular, `Animal` is considered an
30      * abstract class because its definition is incomplete, and this
31      * prevents `Animal` objects from being instantiated. However, we
32      * define (override) `speak` in `Cat` and `Dog`, and this makes
33      * those classes concrete and instantiable. See below.
34      */
35
36     const std::string& get_name() const { return name_; }

```

```

37     unsigned int get_weight() const { return weight_; }
38
39     // When using inheritance, the base class *must* have a virtual
40     // destructor, even if it doesn't have to do anything.
41     virtual ~Animal() { }
42
43 private:
44     std::string name_;
45     unsigned int weight_;
46 };
47
48 /*
49  * Class `Cat` inherits from abstract class `Animal`. In order for `Cat` to
50  * be a concrete class (in order to create instances of `Cat`), it has to
51  * override `Animal`'s pure virtual function `speak`:
52  */
53 class Cat : public Animal
54 {
55 public:
56     Cat(const std::string& name);
57     void speak() override; // << ADDED override HERE
58 };
59
60 /*
61  * Class `Dog` also has to override `speak` in order to be concrete.
62  */
63 class Dog : public Animal
64 {
65 public:
66     Dog(const std::string& name);
67     void speak() override; // << ADDED override HERE
68     void play() override;
69
70 private:
71     unsigned int nbones_;
72 };
73
74 /*
75  * The function definitions in animals3.cpp remain unchanged. However, see
76  * the end of animals3.cpp for how to store `Animal`s in a collection.
77  */

```

8 src/animals3.cxx

```
1  #include "animals3.hxx"
2
3  #include <iostream>
4  #include <memory>
5  #include <vector>
6
7  Animal::Animal(const std::string& name, unsigned int weight)
8      : name_(name), weight_(weight)
9  { }
10
11 void Animal::eat(unsigned int amount)
12 {
13     weight_ += amount;
14 }
15
16 void Animal::play()
17 {
18     std::cout << get_name() << " plays.\n";
19 }
20
21 Cat::Cat(const std::string& name) : Animal(name, 10)
22 { }
23
24 void Cat::speak()
25 {
26     std::cout << get_name() << " says meow!\n";
27 }
28
29 Dog::Dog(const std::string& name)
30     : Animal{name, 50}
31     , nbones_{1}
32 { }
33
34 void Dog::speak()
35 {
36     std::cout << get_name() << " says";
37     for (size_t i = 0; i < nbones_; ++i)
38         std::cout << " woof";
39     std::cout << "!\n";
40 }
41
42 void Dog::play()
```

```

43 {
44     Animal::play();
45     ++nbones_;
46 }
47
48 void play_twice(Animal& an)
49 {
50     an.play();
51     an.play();
52 }
53
54 /*
55  * In C++, inheritance polymorphism works only through references and
56  * pointers, because base classes and derived classes may have different
57  * sizes. That is, you cannot have an `Animal` variable that actual contains
58  * a `Cat` or `Dog` objects. (Actually, because `Animal` is abstract now, you
59  * cannot have `Animal` variable at all.) However, you can have an `Animal&`
60  * that actually refers to a `Dog` or a `Cat`, and similarly for pointer types.
61  *
62  * Collection types like vectors usually depend on storing elements directly,
63  * but that's a problem for inheritance because a std::vector<Animal> can't
64  * store `Cat`s and `Dog`s. But a std::shared_ptr<Animal> can point to a
65  * `Cat` or a `Dog` on the free store. So we can make a vector of shared
66  * pointers to `Animal`s, where each pointer actually points to a concrete
67  * derived class of `Animal`.
68  */
69
70 using animal_ptr = std::shared_ptr<Animal>;
71 using Animal_vec = std::vector<animal_ptr>;
72
73 Animal_vec get_animals()
74 {
75     Animal_vec animals;
76     animals.push_back(std::make_shared<Dog>("Willie"));
77     animals.push_back(std::make_shared<Cat>("Vinny"));
78     animals.push_back(std::make_shared<Cat>("Francie"));
79     return animals;
80 }
81
82 /*
83  * We can `push_back` a `shared_ptr<Cat>` into a vector of
84  * `shared_ptr<Animal>` because `shared_ptr` is specially designed to work
85  * with inheritance. If we have a vector of shared pointers to `Animal`s, we
86  * can use the pointers, even though we don't know what kind of `Animal` each
87  * pointer points to.

```

```

88  */
89
90  void example1()
91  {
92      Animal_vec animals = get_animals();
93      for (animal_ptr an : animals) an->speak();
94      for (animal_ptr an : animals) play_twice(*an);
95      for (animal_ptr an : animals) an->speak();
96  }
97
98  /*
99   * It's possible to convert in the other direction, from a shared pointer to
100  * a base class into a shared pointer to a derived class, but the conversion
101  * is partial, because it can only happen if the pointer actually is pointing
102  * to the desired derived class. Below, we show how to attempt to convert a
103  * `shared_ptr<Animal>` into a `shared_ptr<Cat>` using
104  * `std::dynamic_pointer_cast`. The cast returns `nullptr` if the shared
105  * pointer doesn't actually point to a `Cat`.
106  */
107
108  void feed_only_cats(const Animal_vec& animals, unsigned int amount)
109  {
110      std::cout << "Feeding " << amount << " to each cat.\n";
111      for (animal_ptr an : animals) {
112          std::shared_ptr<Cat> cat = std::dynamic_pointer_cast<Cat>(an);
113          if (cat != nullptr) cat->eat(amount);
114      }
115  }
116
117  void census(const Animal_vec& animals)
118  {
119      for (animal_ptr an : animals)
120          std::cout << an->get_name() << " weighs " << an->get_weight() << ".\n";
121  }
122
123  void example2()
124  {
125      Animal_vec animals = get_animals();
126
127      census(animals);
128      feed_only_cats(animals, 1);
129      census(animals);
130  }
131
132  int main()

```



```

133 {
134     example1();
135     example2();
136 }

```

9 src/Sprite_tree.hxx

```

1  #pragma once
2
3  #include <ge211.hxx>
4  #include <memory>
5  #include <map>
6
7  namespace widget
8  {
9
10 namespace detail
11 {
12
13 struct ISprite_set
14 {
15     virtual void add_sprite(ge211::Sprite const&,
16                             ge211::Position,
17                             int,
18                             ge211::Transform) = 0;
19 };
20
21 class Sprite_set_adapter : ISprite_set
22 {
23     ge211::Sprite_set& base_;
24
25 public:
26     explicit Sprite_set_adapter(ge211::Sprite_set& base)
27         : base_(base)
28     { }
29
30 private:
31     void add_sprite(ge211::Sprite const&,
32                    ge211::Position,
33                    int z,
34                    ge211::Transform) override;
35 };
36

```

```

37 struct Sprite_tree_link
38 {
39     virtual int do_draw(ISprite_set&, int z) = 0;
40     virtual ~Sprite_tree_link() = default;
41 };
42
43 class Sprite_tree : private Sprite_tree_link
44 {
45 public:
46     Sprite_tree&
47     add_branch(Sprite_tree&& branch,
48               int z = 0) &;
49
50     Sprite_tree&
51     add_leaf(ge211::Sprite const&,
52             ge211::Position,
53             int z = 0,
54             ge211::Transform const& = ge211::Transform()) &;
55
56     Sprite_tree&&
57     add_branch(Sprite_tree&& branch,
58               int z = 0) &&;
59
60     Sprite_tree&&
61     add_leaf(ge211::Sprite const&,
62             ge211::Position,
63             int z = 0,
64             ge211::Transform const& = ge211::Transform()) &&;
65
66     int draw(ISprite_set&, int);
67
68 protected:
69     int do_draw(ISprite_set&, int) override;
70
71 private:
72     std::multimap<int, std::unique_ptr<Sprite_tree_link>>
73         children_;
74 };
75
76 } // end namespace detail
77
78 } // end namespace widget

```

10 src/Sprite_tree.cxx

```

1  #include "Sprite_tree.hxx"
2  #include <limits>
3
4  using namespace ge211;
5
6  namespace widget
7  {
8
9  namespace detail
10 {
11
12 struct Sprite_tree_leaf : Sprite_tree_link
13 {
14     Sprite_tree_leaf(Sprite const& sprite,
15                     Position pos,
16                     Transform const& transform);
17
18     int do_draw(ISprite_set&, int) override;
19
20     Sprite const&   sprite_;
21     Position        pos_;
22     Transform       transform_;
23 };
24
25 void Sprite_set_adapter::add_sprite(ge211::Sprite const& sprite,
26                                     ge211::Position pos,
27                                     int z,
28                                     ge211::Transform transform)
29 {
30     base_.add_sprite(sprite, pos, z, transform);
31 }
32
33 Sprite_tree&
34 Sprite_tree::add_branch(Sprite_tree&& branch, int z) &
35 {
36     std::unique_ptr<Sprite_tree_link>
37         new_node(new Sprite_tree(std::move(branch)));
38     children_.insert({z, std::move(new_node)});
39     return *this;
40 }
41
42 Sprite_tree&

```

```

43 Sprite_tree::add_leaf(Sprite const& sprite,
44                       Position pos,
45                       int z,
46                       Transform const& transform) &
47 {
48     std::unique_ptr<Sprite_tree_link>
49         new_node(new Sprite_tree_leaf(sprite, pos, transform));
50     children_.emplace(z, std::move(new_node));
51     return *this;
52 }
53
54 Sprite_tree&& Sprite_tree::add_branch(Sprite_tree&& branch, int z)&&
55 {
56     ((Sprite_tree*)(this))->add_branch(std::move(branch), z);
57     return std::move(*this);
58 }
59
60 Sprite_tree&& Sprite_tree::add_leaf(ge211::Sprite const& sprite,
61                                   ge211::Position xy, int z,
62                                   ge211::Transform const& transform) &&
63 {
64     ((Sprite_tree*)(this))->add_leaf(sprite, xy, z, transform);
65     return std::move(*this);
66 }
67
68 int Sprite_tree::do_draw(ISprite_set& set, int z)
69 {
70     if (children_.empty())
71         return z;
72
73     int max_z = z;
74     int previous = children_.begin()->first;
75
76     for (auto const& child : children_) {
77         if (child.first != previous) {
78             z = max_z + 1;
79             previous = child.first;
80         }
81
82         max_z = std::max(max_z, child.second->do_draw(set, z));
83     }
84
85     return max_z;
86 }
87

```

11 test/mock_sprite_set.hxx

```
88 Sprite_tree_leaf::Sprite_tree_leaf(  
89     Sprite const& sprite,  
90     Position pos,  
91     Transform const& transform)  
92     : sprite_(sprite)  
93       , pos_(pos)  
94       , transform_(transform)  
95 { }  
96  
97 int Sprite_tree_leaf::do_draw(ISprite_set& set, int z)  
98 {  
99     set.add_sprite(sprite_, pos_, z, transform_);  
100    return z;  
101 }  
102  
103 int Sprite_tree::draw(ISprite_set& set, int z)  
104 {  
105     return do_draw(set, z);  
106 }  
107  
108 } // end namespace detail  
109  
110 } // end namespace widget
```

11 test/mock_sprite_set.hxx

```
1 #include "Sprite_tree.hxx"  
2  
3 #include <vector>  
4 #include <initializer_list>  
5 #include <iostream>  
6  
7 struct Pos_sprite  
8 {  
9     ge211::Sprite const& sprite;  
10    ge211::Position      xy;  
11    int                  z;  
12 };  
13  
14 struct Mock_sprite_set : widget::detail::ISprite_set  
15 {  
16     Mock_sprite_set() = default;  
17     Mock_sprite_set(std::initializer_list<Pos_sprite>);
```

12 test/mock_sprite_set.cxx

```
18
19     std::vector<Pos_sprite> sprites;
20
21     void add_sprite(ge211::Sprite const&,
22                    ge211::Position, int, ge211::Transform) override;
23 };
24
25 bool operator==(Pos_sprite const&, Pos_sprite const&);
26 bool operator==(Mock_sprite_set const&, Mock_sprite_set const&);
27
28 std::ostream& operator<<(std::ostream&, Pos_sprite const&);
29 std::ostream& operator<<(std::ostream&, Mock_sprite_set const&);
```

12 test/mock_sprite_set.cxx

```
1  #include "mock_sprite_set.hxx"
2
3  bool operator==(Pos_sprite const& a, Pos_sprite const& b)
4  {
5      return &a.sprite == &b.sprite &&
6             a.xy == b.xy &&
7             a.z == b.z;
8  }
9
10 std::ostream& operator<<(std::ostream& os, Pos_sprite const& ps)
11 {
12     return os
13         << "Pos_sprite{" << &ps.sprite << " @ ("
14         << ps.xy.x << ", " << ps.xy.y << ", " << ps.z << ")}";
15 }
16
17 Mock_sprite_set::Mock_sprite_set(std::initializer_list<Pos_sprite> elts)
18 {
19     std::move(elts.begin(), elts.end(), std::back_inserter(sprites));
20 }
21
22 void Mock_sprite_set::add_sprite(ge211::Sprite const& sprite,
23                                 ge211::Position xy, int z, ge211::Transform)
24 {
25     sprites.push_back(Pos_sprite{sprite, xy, z});
26 }
27
28 bool operator==(Mock_sprite_set const& a, Mock_sprite_set const& b)
```

```

29 {
30     return a.sprites == b.sprites;
31 }
32
33 std::ostream& operator<<(std::ostream& os, Mock_sprite_set const& mss)
34 {
35     os << "{\n";
36
37     for (auto const& ps : mss.sprites)
38         os << " " << ps << "\n";
39
40     return os << "}";
41 }

```

13 test/sprite_tree_test.cxx

```

1  #include "Sprite_tree.hxx"
2  #include "mock_sprite_set.hxx"
3
4  #include <catch.hxx>
5
6  using namespace ge211;
7  using namespace widget::detail;
8
9  TEST_CASE("Sprite_tree::draw")
10 {
11     Circle_sprite sprite1(5);
12     Rectangle_sprite sprite2({10, 15});
13     Sprite_tree tree;
14
15     tree.add_leaf(sprite1, {0, 0}, 0);
16     tree.add_leaf(sprite2, {10, 0}, 0);
17
18     Sprite_tree child1;
19     child1.add_leaf(sprite1, {0, 5}, -6);
20     child1.add_leaf(sprite2, {10, 5}, -4);
21     tree.add_branch(std::move(child1), 1);
22
23     tree.add_leaf(sprite1, {0, 15}, 0);
24
25     Mock_sprite_set set;
26     CHECK( tree.draw(set, 5) == 7 );
27     CHECK( set == Mock_sprite_set{

```

```

28     {sprite1, { 0, 0}, 5},
29     {sprite2, {10, 0}, 5},
30     {sprite1, { 0, 15}, 5},
31     {sprite1, { 0, 5}, 6},
32     {sprite2, {10, 5}, 7},
33 } );
34 }
35
36 TEST_CASE("fluent")
37 {
38     Circle_sprite sprite0(5);
39
40     auto tree = Sprite_tree()
41         .add_leaf(sprite0, {0, 0}, 8)
42         .add_leaf(sprite0, {1, 0}, 7)
43         .add_branch(Sprite_tree()
44             .add_leaf(sprite0, {2, 0}, 6)
45             .add_leaf(sprite0, {3, 0}, 5),
46             4)
47         .add_leaf(sprite0, {4, 0}, 3);
48
49     Mock_sprite_set set;
50     CHECK( tree.draw(set, 0) == 4 );
51     CHECK( set == Mock_sprite_set{
52         {sprite0, {4, 0}, 0},
53         {sprite0, {3, 0}, 1},
54         {sprite0, {2, 0}, 2},
55         {sprite0, {1, 0}, 3},
56         {sprite0, {0, 0}, 4},
57     } );
58 }

```