14VIRTUAL

Contents

1 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.3)
project(lec14 CXX)
include(.cs211/cmake/CMakeLists.txt)

add_program(animals1 src/animals1.cxx)
add_program(animals2 src/animals2.cxx)
add_program(animals3 src/animals3.cxx)

add_test_program(sprite_tree_test
        test/mock_sprite_set.cxx
```

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

2 examples.ml

```
(* Some examples of parametric polymorphism in OCaml. *)

(* Reverses a list. *)
let reverse xs0 =
  let rec loop acc xs =
    match xs with
    | []        -> acc
    | x :: xs'  -> loop (x :: acc) xs'
  in loop [] xs0
(* Inferred type scheme:
    reverse : 'a list -> 'a list *)

(* Sums a list of ints. *)
let sum xs0 =
  let rec loop acc = function
  | []      -> acc
  | x :: xs -> loop (x + acc) xs
  in loop 0 xs0
(* Inferred type scheme:
    sum : int list -> int *)

(* Sums a list of floats. *)
let sum_float xs0 =
  let rec loop acc = function
  | []      -> acc
  | x :: xs -> loop (x +. acc) xs
  in loop 0.0 xs0
(* Inferred type scheme:
    sum_float : float list -> float *)

(* Reduces a list, left associatively. *)
let foldl (plus, zero, xs0) =
  let rec loop acc = function
  | []      -> acc
  | x :: xs -> loop (plus (x, acc)) xs
  in loop zero xs0
(* Inferred type scheme:
```

```
    foldl : (('a * 'b -> 'b) * 'b * 'a list) -> 'b *)

(* Reduces a list, left associatively. *)
let foldl_curried plus zero xs0 =
  let rec loop acc = function
  | []       -> acc
  | x :: xs -> loop (plus x acc) xs
  in loop zero xs0
(* Inferred type scheme:
    foldl : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b *)

(* Sorts a list according to the given binary predicate. *)
let rec merge_sort (<) xs0 =

  let split =
    foldl_curried (fun x (xs1, xs2) -> (x :: xs2, xs1)) ([], []) in

  let rec merge xs1 xs2 = match xs1, xs2 with
    | x1 :: xs1', x2 :: xs2' ->
      if x1 < x2
        then x1 :: merge xs1' xs2
        else x2 :: merge xs1 xs2'
    | [], _ -> xs2
    | _, [] -> xs1 in

  let rec sort xs =
    match xs with
    | [] | [_] -> xs
    | _        ->
      let (xs1, xs2) = split xs in
        merge (sort xs1) (sort xs2) in

  sort xs0
(* Inferred type scheme:
    merge_sort : ('a -> 'a -> bool) -> 'a list -> 'a list *)
```

3 src/animals1.hxx

```
/////
///// START HERE
/////

#include <string>
```

3 src/animals1.hxx

```cpp

///
/// Inheritance
///

/*
 * C++ provides a mechanism for defining families of classes that are
 * similar in some ways and different in others. For example, suppose
 * we want classes to represent different kinds of pet animal, which
 * have operations like sleeping and playing. We can combine their
 * common behavior into a base class and then derive the specific
 * animal classes from that.
 *
 * Let's call the base class Animal. We'll initially define it as
 * follows:
 */

class Animal
{
public:
    Animal(const std::string& name, unsigned int weight);

    void eat(unsigned int amount);
    void play();

    const std::string& get_name() const { return name_; }
    unsigned int get_weight() const { return weight_; }

private:
    std::string name_;
    unsigned int weight_;
};

/*
 * That is, every animal has a name and a weight, and has operations to
 * eat and to play. The implementations of the operations are in
 * animals1.cpp.
 *
 * We then *derive* other classes from the `Animal` class. Deriving
 * makes the new class like the old class, but then lets us add to or
 * change it. Here's `Cat`:
 */

class Cat : public Animal
{
```

3 src/animals1.hxx

```
public:
    Cat(const std::string& name);
    void speak();
};

/*
 * The notation `: public Animal` means that `Cat` starts out as a copy
 * of `Animal`, with all of `Animal`'s public members public in `Cat` as
 * well. We then define a constructor, which takes a name and passes it
 * on to the base class's constructor, along with a fixed weight of 10.
 *
 * `Cat` also has a new member function, `speak`, that `Animal` does not
 * have.
 *
 * We also define `Dog` by deriving from `Animal`, but unlike `Cat`,
 * `Dog`'s playing behavior is a bit more complicated. In particular,
 * each `Dog` has some number of bones, and each time we play the dog
 * gets a new bone. And then when the dog speaks, it says "woof"
 * repeatedly, once for each of its bones. Here's the definition of the
 * `Dog` class:
 */

class Dog : public Animal
{
public:
    Dog(const std::string& name);
    void speak();
    void play();

private:
    unsigned int nbones_;
};

/*
 * As you can see, `Dog` adds a member variable `nbones_` to `Animal`,
 * adds a member function `speak`, and replaces the base class's
 * `Animal::play` function with a `Dog`-specific `play` function. The
 * `Dog::play` function delegates to `Animal::play` to do the playing
 * and also increments `nbones_`.
 *
 * Now we can write a program involving cats and dogs. See the bottom of
 * animals1.cpp for main and an introduction to inheritance polymorphism.
 */
```

4 src/animals1.cxx

```
/////
///// SEE animals1.h first
/////

// The implementations are here, but the implementations aren't very
// interesting.

#include "animals1.hxx"
#include <iostream>

Animal::Animal(const std::string& name, unsigned int weight)
        : name_(name), weight_(weight)
{ }

void Animal::eat(unsigned int amount)
{
    weight_ += amount;
}

void Animal::play()
{
    std::cout << get_name() << " plays.\n";
}

// The `Cat` constructor passes its the name to the `Animal` constructor
// along with a starting cat weight.
Cat::Cat(const std::string& name) : Animal(name, 10)
{ }

void Cat::speak()
{
    std::cout << get_name() << " says meow!\n";
}

// The `Dog` constructor has to initialize both the base class `Animal` and
// its own member `nbones_`.
Dog::Dog(const std::string& name)
        : Animal{name, 50}
        , nbones_{1}
{ }

void Dog::speak()
```

```cpp
{
    std::cout << get_name() << " says";
    for (size_t i = 0; i < nbones_; ++i)
        std::cout << " woof";
    std::cout << "!\n";
}

// `Dog`'s `play` behavior is different from the base `Animal`'s.
void Dog::play()
{
    Animal::play();
    ++nbones_;
}

// Forward declaration, in case we want to call `play_twice` from `main`.
void play_twice(Animal&);

int main()
{
    // My mom has four cats and two dogs.
    Dog willie("Willie");
    Cat vinny("Vinny");
    Cat francie("Francie");

    willie.play();
    vinny.play();
    francie.eat(2);
}

/*
 * An additional feature of inheritance is a form of polymorphism.
 * Polymorphism is when the same variable (or more abstractly, entity)
 * can come in different forms. In this case, the polymorphism is that
 * a reference or pointer whose type says it's of the base class can
 * actually refer to an object of any derived class. Here we have a
 * function that takes an `Animal` reference:
 */

void play_twice(Animal& an)
{
    an.play();
    an.play();
}

/*
```

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

5 src/animals2.hxx

```
#include <string>

///
/// Polymorphism
///

/*
 * The solution to the problem at the end of animals2.cpp is a *virtual*
 * function. Declaring a function `virtual` in the base class means
 * that we expect the derived classes to *override* it, and we want to
 * get the overridden derived class behavior, even with working with a
 * base class reference. To do this, we change the declaration of
 * `Animal::play` to be virtual:
 */
class Animal
{
public:
    Animal(const std::string& name, unsigned int weight);
```

```
    void eat(unsigned int amount);
    virtual void play(); // <<< CHANGE IS HERE

    const std::string& get_name() const { return name_; }
    unsigned int get_weight() const { return weight_; }

    // When using inheritance, the base class *must* have a virtual
    // destructor, even if it doesn't have to do anything.
    virtual ~Animal() { }

private:
    std::string name_;
    unsigned int weight_;
};

/*
 * The `Cat` class remains the same as before because we want `Cat` to have
 * the default `play` behavior inherited from `Animal`.
 */
class Cat : public Animal
{
public:
    Cat(const std::string& name);
    void speak();
};

/*
 * Then in `Dog`, we redefine `play` and indicate that we are overriding
 * the base definition of `play`:
 */
class Dog : public Animal
{
public:
    Dog(const std::string& name);
    void speak();
    void play() override; // <<< OTHER CHANGE IS HERE

private:
    unsigned int nbones_;
};

/*
 * Now when we use a `Dog` object via an `Animal&` reference, it uses
 * `Dog::play` rather than `Animal::play`.
```

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

6 src/animals2.cxx

```
// The implementations are here, but the implementations aren't very
// interesting, since they're the same as in animals1.cpp. You probably
// want animals2.h or if you've been there already, animals3.h.

#include "animals2.hxx"
#include <iostream>

Animal::Animal(const std::string& name, unsigned int weight)
        : name_(name), weight_(weight)
{ }

void Animal::eat(unsigned int amount)
{
    weight_ += amount;
}

void Animal::play()
{
    std::cout << get_name() << " plays.\n";
}

Cat::Cat(const std::string& name) : Animal(name, 10)
{ }

void Cat::speak()
{
    std::cout << get_name() << " says meow!\n";
}

Dog::Dog(const std::string& name)
        : Animal{name, 50}
        , nbones_{1}
{ }
```

```cpp
void Dog::speak()
{
    std::cout << get_name() << " says";
    for (size_t i = 0; i < nbones_; ++i)
        std::cout << " woof";
    std::cout << "!\n";
}

void Dog::play()
{
    Animal::play();
    ++nbones_;
}

// This function uses inheritance polymorphism, since it takes an `Animal`
// reference that might refer to a derived class of `Animal` such as `Dog` or
// `Cat`. Further, because `Animal::play` is virtual, it will use the derived
// class's version of `play` if the derived class overrides it.
void play_twice(Animal& an)
{
    an.play();
    an.play();
}

/*
 * We can't do the same for `speak` because `Animal` doesn't define a `speak`
 * function, so there's no guarantee that a particular `Animal&` will be from
 * a class that defines `speak`.
 *
 *     void speak_twice(Animal& an)
 *     {
 *         an.speak();
 *         an.speak();
 *     }
 */

int main()
{
    Dog willie("Willie");
    Cat vinny("Vinny");
    Cat francie("Francie");

    play_twice(willie);
    play_twice(vinny);
```

```
    francie.eat(2);
    francie.speak();
    willie.speak();
}
```

7 src/animals3.hxx

```
#include <string>

///
/// Pure virtual functions
///

/*
 * We did not define a `speak` function in the `Animal` class, since
 * that function is different for the derived classes, and so there was
 * no sharing to be had. But this prevents us from calling `speak` on an
 * `Animal&`, even if the object referred to is actually a `Cat` or
 * `Dog`. It has to be this way because there's no guarantee of every
 * derived class of `Animal` will define `speak`. We can fix this by
 * declaring in `Animal` that every derived class of `Animal` must
 * define `speak`. We do this by declaring `speak` to be *pure virtual*.
 */

class Animal
{
public:
    Animal(const std::string& name, unsigned int weight);

    void eat(unsigned int amount);
    virtual void play();
    virtual void speak() = 0; // << NEW LINE HERE

    /*
     * The `= 0` means that `Animal` will not define `speak`, but that
     * its derived classes must. In particular, `Animal` is considered an
     * *abstract class* because its definition is incomplete, and this
     * prevents `Animal` objects from being instantiated. However, we
     * define (*override*) `speak` in `Cat` and `Dog`, and this makes
     * those classes concrete and instantiable. See below.
     */

    const std::string& get_name() const { return name_; }
```

```cpp
    unsigned int get_weight() const { return weight_; }

    // When using inheritance, the base class *must* have a virtual
    // destructor, even if it doesn't have to do anything.
    virtual ~Animal() { }

private:
    std::string name_;
    unsigned int weight_;
};

/*
 * Class `Cat` inherits from abstract class `Animal`. In order for `Cat` to
 * be a concrete class (in order to create instances of `Cat`), it has to
 * override `Animal`'s pure virtual function `speak`:
 */
class Cat : public Animal
{
public:
    Cat(const std::string& name);
    void speak() override; // << ADDED override HERE
};

/*
 * Class `Dog` also has to override `speak` in order to be concrete.
 */
class Dog : public Animal
{
public:
    Dog(const std::string& name);
    void speak() override; // << ADDED override HERE
    void play() override;

private:
    unsigned int nbones_;
};

/*
 * The function definitions in animals3.cpp remain unchanged. However, see
 * the end of animals3.cpp for how to store `Animal`s in a collection.
 */
```

8 src/animals3.cxx

```cpp
#include "animals3.hxx"

#include <iostream>
#include <memory>
#include <vector>

Animal::Animal(const std::string& name, unsigned int weight)
        : name_(name), weight_(weight)
{ }

void Animal::eat(unsigned int amount)
{
    weight_ += amount;
}

void Animal::play()
{
    std::cout << get_name() << " plays.\n";
}

Cat::Cat(const std::string& name) : Animal(name, 10)
{ }

void Cat::speak()
{
    std::cout << get_name() << " says meow!\n";
}

Dog::Dog(const std::string& name)
        : Animal{name, 50}
        , nbones_{1}
{ }

void Dog::speak()
{
    std::cout << get_name() << " says";
    for (size_t i = 0; i < nbones_; ++i)
        std::cout << " woof";
    std::cout << "!\n";
}

void Dog::play()
```

```
{
    Animal::play();
    ++nbones_;
}

void play_twice(Animal& an)
{
    an.play();
    an.play();
}

/*
 * In C++, inheritance polymorphism works only through references and
 * pointers, because base classes and derived classes may have different
 * sizes. That is, you cannot have an `Animal` variable that actual contains
 * a `Cat` or `Dog` objects. (Actually, because `Animal` is abstract now, you
 * cannot have `Animal` variable at all.) However, you can have an `Animal&`
 * that actually refers to a `Dog` or a `Cat`, and similarly for pointer types.
 *
 * Collection types like vectors usually depend on storing elements directly,
 * but that's a problem for inheritance because a std::vector<Animal> can't
 * store `Cat`s and `Dog`s. But a std::shared_ptr<Animal> *can* point to a
 * `Cat` or a `Dog` on the free store. So we can make a vector of shared
 * pointers to `Animal`s, where each pointer actually points to a concrete
 * derived class of `Animal`.
 */

using animal_ptr = std::shared_ptr<Animal>;
using Animal_vec = std::vector<animal_ptr>;

Animal_vec get_animals()
{
    Animal_vec animals;
    animals.push_back(std::make_shared<Dog>("Willie"));
    animals.push_back(std::make_shared<Cat>("Vinny"));
    animals.push_back(std::make_shared<Cat>("Francie"));
    return animals;
}

/*
 * We can `push_back` a `shared_ptr<Cat>` into a vector of
 * `shared_ptr<Animal>` because `shared_ptr` is specially designed to work
 * with inheritance. If we have a vector of shared pointers to `Animal`s, we
 * can use the pointers, even though we don’t know what kind of `Animal` each
 * pointer points to.
```

```
 */

void example1()
{
    Animal_vec animals = get_animals();
    for (animal_ptr an : animals) an->speak();
    for (animal_ptr an : animals) play_twice(*an);
    for (animal_ptr an : animals) an->speak();
}

/*
 * It's possible to convert in the other direction, from a shared pointer to
 * a base class into a shared pointer to a derived class, but the conversion
 * is partial, because it can only happen if the pointer actually is pointing
 * to the desired derived class. Below, we show how to attempt to convert a
 * `shared_ptr<Animal>` into a `shared_ptr<Cat>` using
 * `std::dynamic_pointer_cast`. The cast returns `nullptr` if the shared
 * pointer doesn't actually point to a `Cat`.
 */

void feed_only_cats(const Animal_vec& animals, unsigned int amount)
{
    std::cout << "Feeding " << amount << " to each cat.\n";
    for (animal_ptr an : animals) {
        std::shared_ptr<Cat> cat = std::dynamic_pointer_cast<Cat>(an);
        if (cat != nullptr) cat->eat(amount);
    }
}

void census(const Animal_vec& animals)
{
    for (animal_ptr an : animals)
        std::cout << an->get_name() << " weighs " << an->get_weight() << ".\n";
}

void example2()
{
    Animal_vec animals = get_animals();

    census(animals);
    feed_only_cats(animals, 1);
    census(animals);
}

int main()
```

```
{
    example1();
    example2();
}
```

9 src/Sprite_tree.hxx

```
#pragma once

#include <ge211.hxx>
#include <memory>
#include <map>

namespace widget
{

namespace detail
{

struct ISprite_set
{
    virtual void add_sprite(ge211::Sprite const&,
                            ge211::Position,
                            int,
                            ge211::Transform) = 0;
};

class Sprite_set_adapter : ISprite_set
{
    ge211::Sprite_set& base_;

public:
    explicit Sprite_set_adapter(ge211::Sprite_set& base)
            : base_(base)
    { }

private:
    void add_sprite(ge211::Sprite const&,
                    ge211::Position,
                    int z,
                    ge211::Transform) override;
};

```

```
struct Sprite_tree_link
{
    virtual int do_draw(ISprite_set&, int z) = 0;
    virtual ~Sprite_tree_link() = default;
};

class Sprite_tree : private Sprite_tree_link
{
public:
    Sprite_tree&
    add_branch(Sprite_tree&& branch,
               int z = 0) &;

    Sprite_tree&
    add_leaf(ge211::Sprite const&,
             ge211::Position,
             int z = 0,
             ge211::Transform const& = ge211::Transform()) &;

    Sprite_tree&&
    add_branch(Sprite_tree&& branch,
               int z = 0) &&;

    Sprite_tree&&
    add_leaf(ge211::Sprite const&,
             ge211::Position,
             int z = 0,
             ge211::Transform const& = ge211::Transform()) &&;

    int draw(ISprite_set&, int);

protected:
    int do_draw(ISprite_set&, int) override;

private:
    std::multimap<int, std::unique_ptr<Sprite_tree_link>>
            children_;
};

} // end namespace detail

} // end namespace widget
```

10 src/Sprite_tree.cxx

```
#include "Sprite_tree.hxx"
#include <limits>

using namespace ge211;

namespace widget
{

namespace detail
{

struct Sprite_tree_leaf : Sprite_tree_link
{
    Sprite_tree_leaf(Sprite const& sprite,
                     Position pos,
                     Transform const& transform);

    int do_draw(ISprite_set&, int) override;

    Sprite const&   sprite_;
    Position        pos_;
    Transform       transform_;
};

void Sprite_set_adapter::add_sprite(ge211::Sprite const& sprite,
                                    ge211::Position pos,
                                    int z,
                                    ge211::Transform transform)
{
    base_.add_sprite(sprite, pos, z, transform);
}

Sprite_tree&
Sprite_tree::add_branch(Sprite_tree&& branch, int z) &
{
    std::unique_ptr<Sprite_tree_link>
            new_node(new Sprite_tree(std::move(branch)));
    children_.insert({z, std::move(new_node)});
    return *this;
}

Sprite_tree&
```

```
Sprite_tree::add_leaf(Sprite const& sprite,
                      Position pos,
                      int z,
                      Transform const& transform) &
{
    std::unique_ptr<Sprite_tree_link>
            new_node(new Sprite_tree_leaf(sprite, pos, transform));
    children_.emplace(z, std::move(new_node));
    return *this;
}

Sprite_tree&& Sprite_tree::add_branch(Sprite_tree&& branch, int z)&&
{
    ((Sprite_tree*)(this))->add_branch(std::move(branch), z);
    return std::move(*this);
}

Sprite_tree&& Sprite_tree::add_leaf(ge211::Sprite const& sprite,
                                    ge211::Position xy, int z,
                                    ge211::Transform const& transform) &&
{
    ((Sprite_tree*)(this))->add_leaf(sprite, xy, z, transform);
    return std::move(*this);
}

int Sprite_tree::do_draw(ISprite_set& set, int z)
{
    if (children_.empty())
        return z;

    int max_z    = z;
    int previous = children_.begin()->first;

    for (auto const& child : children_) {
        if (child.first != previous) {
            z          = max_z + 1;
            previous   = child.first;
        }

        max_z = std::max(max_z, child.second->do_draw(set, z));
    }

    return max_z;
}

```

```
Sprite_tree_leaf::Sprite_tree_leaf(
        Sprite const& sprite,
        Position pos,
        Transform const& transform)
        : sprite_(sprite)
          , pos_(pos)
          , transform_(transform)
{ }

int Sprite_tree_leaf::do_draw(ISprite_set& set, int z)
{
    set.add_sprite(sprite_, pos_, z, transform_);
    return z;
}

int Sprite_tree::draw(ISprite_set& set, int z)
{
    return do_draw(set, z);
}

} // end namespace detail

} // end namespace widget
```

11 test/mock_sprite_set.hxx

```
#include "Sprite_tree.hxx"

#include <vector>
#include <initializer_list>
#include <iostream>

struct Pos_sprite
{
    ge211::Sprite const& sprite;
    ge211::Position      xy;
    int                  z;
};

struct Mock_sprite_set : widget::detail::ISprite_set
{
    Mock_sprite_set() = default;
    Mock_sprite_set(std::initializer_list<Pos_sprite>);
```

```

    std::vector<Pos_sprite> sprites;

    void add_sprite(ge211::Sprite const&,
                    ge211::Position, int, ge211::Transform) override;
};

bool operator==(Pos_sprite const&, Pos_sprite const&);
bool operator==(Mock_sprite_set const&, Mock_sprite_set const&);

std::ostream& operator<<(std::ostream&, Pos_sprite const&);
std::ostream& operator<<(std::ostream&, Mock_sprite_set const&);
```

12 test/mock_sprite_set.cxx

```
#include "mock_sprite_set.hxx"

bool operator==(Pos_sprite const& a, Pos_sprite const& b)
{
    return &a.sprite == &b.sprite &&
           a.xy == b.xy &&
           a.z == b.z;
}

std::ostream& operator<<(std::ostream& os, Pos_sprite const& ps)
{
    return os
            << "Pos_sprite{" << &ps.sprite << " @ ("
            << ps.xy.x << ", " << ps.xy.y << ", " << ps.z << ")}";
}

Mock_sprite_set::Mock_sprite_set(std::initializer_list<Pos_sprite> elts)
{
    std::move(elts.begin(), elts.end(), std::back_inserter(sprites));
}

void Mock_sprite_set::add_sprite(ge211::Sprite const& sprite,
                                 ge211::Position xy, int z, ge211::Transform)
{
    sprites.push_back(Pos_sprite{sprite, xy, z});
}

bool operator==(Mock_sprite_set const& a, Mock_sprite_set const& b)
```

```
{
    return a.sprites == b.sprites;
}

std::ostream& operator<<(std::ostream& os, Mock_sprite_set const& mss)
{
    os << "{\n";

    for (auto const& ps : mss.sprites)
        os << "   " << ps << "\n";

    return os << "}";
}
```

13 test/sprite_tree_test.cxx

```
#include "Sprite_tree.hxx"
#include "mock_sprite_set.hxx"

#include <catch.hxx>

using namespace ge211;
using namespace widget::detail;

TEST_CASE("Sprite_tree::draw")
{
    Circle_sprite sprite1(5);
    Rectangle_sprite sprite2({10, 15});
    Sprite_tree tree;

    tree.add_leaf(sprite1, {0, 0}, 0);
    tree.add_leaf(sprite2, {10, 0}, 0);

    Sprite_tree child1;
    child1.add_leaf(sprite1, {0, 5}, -6);
    child1.add_leaf(sprite2, {10, 5}, -4);
    tree.add_branch(std::move(child1), 1);

    tree.add_leaf(sprite1, {0, 15}, 0);

    Mock_sprite_set set;
    CHECK( tree.draw(set, 5) == 7 );
    CHECK( set == Mock_sprite_set{
```

```
        {sprite1, { 0,  0}, 5},
        {sprite2, {10,  0}, 5},
        {sprite1, { 0, 15}, 5},
        {sprite1, { 0,  5}, 6},
        {sprite2, {10,  5}, 7},
    } );
}

TEST_CASE("fluent")
{
    Circle_sprite sprite0(5);

    auto tree = Sprite_tree()
            .add_leaf(sprite0, {0, 0}, 8)
            .add_leaf(sprite0, {1, 0}, 7)
            .add_branch(Sprite_tree()
                .add_leaf(sprite0, {2, 0}, 6)
                .add_leaf(sprite0, {3, 0}, 5),
                4)
            .add_leaf(sprite0, {4, 0}, 3);

    Mock_sprite_set set;
    CHECK( tree.draw(set, 0) == 4 );
    CHECK( set == Mock_sprite_set{
            {sprite0, {4, 0}, 0},
            {sprite0, {3, 0}, 1},
            {sprite0, {2, 0}, 2},
            {sprite0, {1, 0}, 3},
            {sprite0, {0, 0}, 4},
    } );
}
```