CSI 32 **Virtual Functions**
in-class work

1. Start by grabbing the file virtualFunctions.h from our website or Blackboard.
Here is the definition of `class A` from it:

```
class A
{
public:
   A(std::string n, int x) : name{ n }, age{ x }
   {
      if (x < 0) { age = 0; } // no negative ages
   }

   void play() const
   {
      std::cout << "Name: " << name
         << ", age: " << age << std::endl;
   }

private:
        std::string name;
        int age;
};
```

The class A has two data attributes: `age` and `name` (that can be anything, but you will see later that we are considering trees/bushes/etc.)
It also has a public member function play that will display the name and the age, and a constructor with two required arguments.

Stage 1:

Let's define a class B, that will inherit (use public inheritance) from class A (so A is a *base class* and B is a *derived class*).
`class B` will add one more detail, a `type`: `"bush"`, `"tree"`, `"grass"`, or `""`(no type).
Also, modify member function play() to display all three pieces of information for objects of class B.
Go ahead, try to do the work, then look at the suggested code on the next page.

Here is a possible definition of class B (if you decide to use it, do not copy and paste - retype):

```
class B: public A {

public:

     B(std::string n, int x, std::string y) : A(n, x)
     {

          if (y == "bush" or y == "tree" or y == "")
          {
               type = y;
          }
          else { type = "grass";   }
     }

     void play() const
     {
          std::cout << "Name: " << name
            << ", type: " << type
            << ", age: " << age << std::endl;
     }


protected:
     std::string type;

};
```

By the way, I had to go back and modify class A: change **private** to **protected**, so that B has access to data attributes name and age. I also decided to keep type in the protected block (not **private**), as I'm planning to define another class that inherits from B and I want it to have access to type attribute.

```
class A
{
public:
   ...

private protected:
     std::string name;
     int age;
};
```

This completes Stage 1.

Stage 2:

Let's add a class C, that will inherit from B, and will add one more detail: height (in feet, cannot be negative).
As before, modify member function play() to display all four pieces of information for objects of class C.

Go ahead, try it, then look at the suggested code on the next page.

```
class C: public B
{
public:

     C(std::string n, int x, std::string y, double h): B(n, x, y)
     {

          // no negative heights
          if (h < 0) { height = 0.0; }
          else { height = h; }
     }

     void play() const
    {
          std::cout << "Name: " << name
           << ", type: " << type
           << ", age: " << age
           << ", height: " << height << std::endl;
     }


private:
     double height; // in feet

};
```

I did end up with putting `height` into `private` block as I don't intend to define classes that inherit from C.

This completes Stage 2. Now let's move on to some testing.

Testing, first round:

Create testing.cpp file (you can use any other name), include all the necessary libraries and type the following definition of the main function:

```
int main()
{
     A a1("Birch", 1);
     B b2("Oak", 2, "tree");
     C c3("Lilac", 4, "bush", 3.1);

     // carefully look at what each play function will display
     a1.play();
     b2.play();
     c3.play();

     A* object; // object is a pointer to type A objects,
     // it will however work with any inherited from type A objects,
     // although not in a way we might have expected it.

     object = &c3; // C3 is an instance of class C
     object->play(); // which play() function is called?

     object = &b2; // b2 is an object of type B
     object->play(); //  which play() function is called?

     object = &a1; // a1 is an object of type A
     object->play(); // which play() function is called?

}
```

Testing, first round follow up:

Did you notice that every time **play()** function was called through the pointer `object`, the **play()** function of class `A` was working?

Most likely this was not what you were expecting.
Indeed, `c3.play()` displayed all four pieces of information: the name, the age, the type, and the height, the same should happen when `object = &c3;` and `object->play();` are called.

Virtual functions are the ones that will help us out:
Go back to class A, add "virtual" keyword right before the return type of function play().
You can do the same in classes B and C (although it isn't mandatory)

```
class A
{
public:
   A(std::string n, int x) : name{ n }, age{ x }
   {
      if (x < 0) { age = 0; } // no negative ages
   }

   virtual void play() const
   {
      std::cout << "Name: " << name
         << ", age: " << age << std::endl;
   }

private:
        std::string name;
        int age;
};
```

Testing, second round:

Re-run the test code you have in `main()` function.

Then check explanations on the next page.

Explanations:

```
int main()
{
     // … skipped

     a1.play(); // static-binding, resolved at compile time
     b2.play(); // static-binding, resolved at compile time
     c3.play(); // static-binding, resolved at compile time

     A* object; // object is a pointer to type A objects,
     // it will however work with any inherited from type A objects,
     // although not in a way we might have expected it.
     object = &c3;
     object->play(); // dynamic binding (during run-time),
    // that's the result of using the virtual keyword.

     object = &b2;
     object->play(); // dynamic binding (during run-time),
    // that's the result of using the virtual keyword.

     object = &a1;
     object->play(); // dynamic binding (during run-time),
    // that's the result of using the virtual keyword.

     // the program chooses the correct derived class method play()
     // based on the object type, not the pointer/reference type

}
```