82. Abstract classes

Objectives

- Understand the purpose of a pure virtual function
- Write pure virtual function
- Understand the purpose of abstract classes
- Write and use abstract classes

Abstract base classes

Abstract classes are abstract in the sense that they do not create objects. The actual objects are created in a nonabstract subclass of the abstract class. Members variables are declared and methods defined in the subclass.

They are used for specifying an **interface**. An interface is the collection of signatures of all methods of the class. The **signature** of a function or method is made up of the name and parameter types. The interface of a class tells you what an object of the class is capable of doing.

Allows software designers to design an abstract interface without exposing implementation details.

Here are some examples

Abstract base	Subclasses
Shape	Circle, Square, …
HardDrive	WesternDigital, Seagate, …

You create an abstract class by defining a class with **at least one** pure virtual function.

A **pure virtual function** is a method in a class that is declared but not defined (i.e., has prototype but no body). To tell C++ that a virtual method is pure, just attach **"= 0"** to the right of the method.

```
#include <iostream>

class P
{
public:
    virtual void f() = 0;
private:
    int x;
};

int main()
{
  P p;
  return 0;
}
```

A virtual method is pure virtual if it is virtual **and has zero initializer**

Abstract bases usually do not have member variables

ERROR: Can't instantiate objects from abstract class

A class is abstract if there is **at least one** pure virtual function. Suppose C inherits P, and P is abstract. If C does not define (i.e. give bodies to) all the pure virtual functions of P, then C is still abstract.

Pure virtual functions are virtual.

If objects can be instantiated from a class, the class is said to be **concrete** (i.e, not pure virtual functions).

```
#include <iostream>

class Shape2D
{
public:
    virtual void draw() const = 0;
    virtual void rotate(int angle) = 0;
};

class Circle: public Shape2D
{
public:
    Circle(int x0, int y0, int r0)
    : x(x0),y(y0),r(r0)
    {}
    void draw() const;
    void rotate(int angle);
private:
    int x, y, r;
};

void Circle::draw() const
{
    std::cout << "draw circle\n";
}

void Circle::rotate(int angle)
{
    std::cout << "rotate circle " << angle
              << " deg\n";
}

int main()
{
    Shape2D * s = new Circle(0, 0, 10);
    s->draw();
    s->rotate(10);
    return 0;
}
```

Note that although you cannot create an object using an abstract class, i.e., the following is an error:

```
class P
{
public:
     P(int x)
     : x_(x)
     {}
     virtual void f() = 0;
};

int main()
{
  P p(42); // WRONG!!! ERROR!!! BAD!!!
  return 0;
}
```

Nonetheless a subclass of P can call the constructor of P:

```
class P
{
public:
     P(int x)
     : x_(x)
     {}
      virtual void f() = 0;
};

class C: public P
{
public:
     C(int x)
     : P(x) // OK!!!
     {}
      void f()
      {}
};

int main()
{
     C c(42);
     return 0;
}
```

Design

VERY IMPORTANT!!! LISTEN!!! Notice the way I wrote the program

The abstract class defines what objects of the subclasses **can** do. But in `main()`, I only call methods **in the base class**.

I let **polymorphism** decide which **specific method** to invoke during runtime.

In `main()`, I try my best to work with only the methods declared in the abstract base.

Suppose I'm writing a game. There are many `Shape2D` pointers to concrete objects created. I keep the pointers in a list. I can use a `for`-loop to run through the pointers to get each to draw the object its pointing to. If you want to add more new type of objects to the game, just create a new concrete subclass from the abstract base.

My `main()` code **does not change**. That's the point.

So first design your program by deciding **abstractly** what all objects in the concrete classes should "do" (their methods). Design the abstract base (interface) based on the above information. This becomes a contract for subclasses. Write your main code using only the abstract interface. Create concrete subclasses one at a time and test as you go along. Once the main code is done, you can "plug" in new subclasses without too many changes to the `main()` code.

Example. Consider a game. Here's a possible piece of code.

```
while (1)
{
    for (int i = 0; i < obj_arr_size; i++)
    {
        obj = obj_arr[i];
        obj->clear();
        if (obj->hit(missile_arr, missile_arr_size))
            obj->explode();
        else
            obj->move();
        obj->draw();
    }
}
```

Notice that if I need to add another kind of `obj` to the game I just need to make sure that the new class supports

- clear
- hit
- explode

- move
- draw

That's one of the powers of OO.

Exercise. You know that in C/C++, an array is homogeneous, i.e., an array can only hold values of one type. An `int` array can only hold integer values. A `double` array can only hold double values. We have a `Rational` class. A `Rational` array can only hold `Rational` values. Create an array that contains 3 values of different types so that you can do the following:

```
std::vector< ??? > v;
// put something into v[0], v[1], v[2]
for (int i = 0; i < 3; ++i)
{
    std::cout << [... something to do with v[i] ...]
              << ' ';
}
// Assume v[0] models 42, v[1] models 3.14, v[2]
// models 1/2.
std::cout << '\n';
```

so that you can this output:

```
42 3.14 1/2
```

Exercise. Continuing the above, write a function that increments the values in the values in the above `std::vector` by 1.

```
std::vector< ??? > v;
// put something into v[0], v[1], v[2]

for (int i = 0; i < 3; ++i)
{
    std::cout << [... something to do with v[i] ...]
              << ' ';
}
std::cout << '\n';
increment(v);
for (int i = 0; i < 3; ++i)
{
    std::cout << [... something to do with v[i] ...]
              << ' ';
}
std::cout << '\n';
```

so that you get this output:

```
42 3.14 1/2
43 4.14 3/1
```

The above exercise is very important. In your programs that you have been writing (since CISS240), you work with integers and doubles more or less separately, mixing them only when necessary. In fact integers are

collected together into the `int` type. Doubles are collected together into the `double` type. This exercise says if you know enough C/C++, you can treat then on equal footing and they can be mixed together. When do you need to know this? Well … when you write more complex software such as a compiler, your integers and doubles appear together in the same string as C/C++ program. And you compiler (g++ or some other compiler) actually has to handle these values in a "uniform way". In other words your compiler has lots of algorithms that does not differentiate between `int` and `double` values. (Of course some do.)

Note that you can create such a `std::vector< T >` of different types `T`. But `std::vector< int >`, `std::vector< double >`, `std::vector< Rational >` are all separate things. And sure, you can write a function template to print values of `std::vector< T >`. But the function to print `std::vector< int >` that is instantiated is different from the function to print `std::vector< double >`.

So the way in which template classes/functions/structs handle multiple types is **very different** from the way in which abstract base classes and polymorphism handle problems involving multiple types.

Exercise. Continuing the above, let you increment function allow you to specify the increment amount of `int` or `double` or `Rational` type:

```
std::vector< ??? > v;
// put something into v[0], v[1], v[2]
for (int i = 0; i < 3; ++i)
{
    std::cout << [... something to do with v[i] ...]
              << ' ';
}
std::cout << '\n';
increment(v, 1);
increment(v, 0.1);
increment(v, Rational(1, 3));
```

(What should a `double` be after you add 1/3 to it? I'm not talking about the value. I'm talking about the type.) The above description is a skeleton and might require quite a few changes here and there.

Exercise. You can given three classes:

```
Scissors
{};
Paper
{};
Stone
{};
```

When you print a `Scissors` object, you see `scissors`. When you print a `Paper` object, you see `paper`. When you print a `Stone` scissors object, you see `stone`. Create a `std::vector` of 3 values so that when you print them, you see

```
paper scissors stone
```

In a loop, let the first two values compete. paper scissors beats paper, etc. Remove the loser from the vector. If the two values are the same, that's a draw, and you keep both. Move on to the next two and let them compete. For the above you get

```
paper scissors stone
scissors stone
stone
```

If you have the following initialization

```
paper paper scissors stone paper stone stone
```

you will get the following when they compete:

```
paper paper scissors stone paper stone stone
paper paper scissors stone paper stone stone
paper scissors stone paper stone stone
paper stone paper stone stone
paper paper stone stone
paper paper stone
paper paper
```

You may add methods to the classes. The above description is a skeleton and might require quite a few changes here and there.

Exercise. Write enough classes to achieve the following.

```
SineFunction s;         // s models sin(x)
CosineFunction c;       // c models cos(x)
MonomialFunction x(1);  // x models x
MonomialFunction x2(2); // x2 models x^2
ConstFunction c(1);     // c models constant function 1

std::cout << s << ' ' << c << ' ' << x ' ' << x2
          << ' ' << c '\n';
// prints sin(x) cos(x) x x^2 1

SumFunction f(x, s);    // f models x + sin(x)
std::cout << f << '\n'; // prints x + sin(x)
ProdFunction g(x, s);   // g models x * sin(x)
std::cout << g << '\n'; // prints 1 * sin(x) +  x * cos(x)

std::cout << Derivative(x) << '\n';  // prints 1
std::cout << Derivative(x2) << '\n'; // prints 2 * x
std::cout << Derivative(s) << '\n';  // prints cos(x)
std::cout << Derivative(f) << '\n';  // prints 1 + cos(x)
```

The above is a skeleton and might require quite a few changes here and there. (Can you do `SumFunction f(x, SumFunction(s, c))`?)