

Tutorial 5: Object-Oriented Programming

The Art of Computer Programming

14 October 2026

Objective. This tutorial puts into practice the fifth lecture: grouping data in a `struct` in C and writing the functions that work on it, then turning the same data into a class, in C++ (constructor, destructor, visibility, copies) and in Python; giving a class of your own operators and special methods; and a small hierarchy of classes, with a specialized method and functions using its objects polymorphically. Do Parts A and B (structures, classes) in the first half of the session and Parts C and D (operators, inheritance and polymorphism) in the second half; Part E is for those who finish early. Exercises are roughly ordered by importance within each part.

Reminders

- Create a folder `tutorial05`, one file per exercise. Compile C programs with `$gcc -Wall -Wextra -std=c23 predict.c -o predict` and C++ programs with `$g++ -Wall -Wextra -std=c++17 shapes.cpp -o shapes`. Do not call a Python file `fractions.py`: it would hide the standard module of that name, used in Q7.
- *The memory checker* (tutorial 4): `-g -fsanitize=address,undefined` on the `gcc` or `g++` command line, or Compiler Explorer (<https://godbolt.org/>) if your compiler lacks it. It is useful in Q2, Q3, and Q4.
- To stop the program on an error, write `fprintf(stderr, "...\\n");` and then `exit(1);` in C (`<stdlib.h>`), `std::cerr << "...\\n";` and then `std::exit(1);` in C++ (`<cstdlib>`), or `raise ValueError(...)` in Python, as in the lecture.
- The greatest common divisor: your function of tutorial 2 in C, `std::gcd` in C++ (`<numeric>`), `math.gcd` in Python. In C++, π is `std::acos(-1.0)` (`<cmath>`).

Part A: Structures in C

Q1. Predict before running. Write down what the following program prints, line by line, then compile and run it to check (%g prints a `double` without useless trailing zeros, e.g., 1 rather than 1.000000).

```
#include <stdio.h>

typedef struct { double x, y; } Point;
struct Rec { char a; double v; char b; };

void move1(Point p) { p.x += 1; }
void move2(Point *p) { p->x += 1; }
Point moved(Point p) { p.y += 1; return p; }

int main(void) {
    Point a = {1, 2};
    Point b = a;
    b.x = 10;
}
```

```
printf("%g %g\\n", a.x, b.x);
move1(a);
printf("%g\\n", a.x);
move2(&a);
printf("%g\\n", a.x);
a = moved(a);
printf("%g %g\\n", a.x, a.y);
Point t[2] = {{0, 0}, {3, 4}};
Point *p = &t[1];
printf("%g %g\\n", p->x, (*p).y + t[0].x);
printf("%zu %zu\\n", sizeof(Point),
        sizeof(struct Rec));
// printf("%d\\n", a == b);
}
```

- Explain each line. Which of the three functions modify `a`, and why? Compare with an array passed to a function (tutorial 4, Q1).
- Explain the size of `struct Rec` (on a 64-bit machine). Reorder its three fields to make it smaller, and check.
- Restore the commented-out line and read the compiler's error. How would you compare two points?

Q2. Fractions in C. A fraction is a pair of integers, kept in lowest terms with a positive denominator: this *invariant* makes the representation of each fraction unique.

```
typedef struct {
    long long num, den; // invariant: den > 0 and gcd(num, den) = 1
} Fraction;
```

- (a) Write the following functions. `frac_make` returns the fraction num/den in lowest terms with a positive denominator (stop the program if $den = 0$); the others go through `frac_make` to build their result.

```
Fraction frac_make(long long num, long long den);
Fraction frac_add(Fraction a, Fraction b);
Fraction frac_mul(Fraction a, Fraction b);
bool frac_equal(Fraction a, Fraction b);
void frac_print(Fraction a); // prints -3/2, or 3 when den is 1
```

Test them: `frac_make(6, -4)` is $-3/2$, $1/2 + 1/3 = 5/6$, $1/2$ and $2/4$ are equal. Why can `frac_equal` simply compare the fields?

- (b) The functions of the lecture take a pointer to the structure (`const DynArray *a`), these take the fraction by value. Why is it reasonable here?
- (c) Compute and print the harmonic numbers $H_n = 1 + 1/2 + \dots + 1/n$ for n from 1 to 50 (check $H_{10} = 7381/2520$), together with their value as a `double`. From which n on is the result wrong, and why? What does the checker say?

Q3. Polynomials in C. A polynomial of degree d has $d + 1$ coefficients, stored in an array on the heap, like the dynamic array of the lecture:

```
typedef struct {
    int degree; // the coefficients are coef[0], ..., coef[degree]
    double *coef; // coef[i]: coefficient of x^i, on the heap
} Polynomial;
void poly_init(Polynomial *p, int degree); // all coefficients 0
void poly_free(Polynomial *p);
double poly_eval(const Polynomial *p, double x);
void poly_print(const Polynomial *p); // 2 x^3 + 0 x^2 + -3 x^1 + 1 x^0
Polynomial poly_derivative(const Polynomial *p);
```

- (a) Write these functions. `poly_eval` uses Horner's method, $c_0 + x(c_1 + x(c_2 + \dots + x c_d))$: a single loop from c_d down to c_0 , with d multiplications. Build $2x^3 - 3x + 1$ and print its values at 0, 1, and 2 (1, 0, and 11).
- (b) `poly_derivative` returns a new polynomial: who owns it? Print the derivative of the example, then free both polynomials, and check with the memory checker (if you have it) that nothing leaks.

Part B: Classes

Q4. Polynomials in C++. Turn the structure of Q3 into a class whose properties are private:

```
class Polynomial {
    int deg;
    double *coef; // new double[deg + 1], owned by the object
public:
    Polynomial(int degree); // all coefficients 0
    ~Polynomial(); // delete[] coef, and print a message
    int degree() const;
    double get(int i) const;
    void set(int i, double c); // stop the program unless 0 <= i <= deg
    double eval(double x) const;
    void print() const;
};
```

- (a) Write the methods inside the class, the constructor with an initializer list, and the destructor printing destroying a polynomial of degree 3. Rewrite the `main` of Q3 with it. Where is the object stored, where are its coefficients? When is the message printed? What became of `poly_init` and `poly_free`?

- (b) In main, try `p.deg = 10;`; read the compiler's error. Write a function `show`, whose parameter `p` is a `const Polynomial &`, printing `p.eval(2)`; then remove the `const` after the parameters of `eval`: what happens, and why?
- (c) Write `Polynomial q = p;`, then `q.set(0, 5);`, and print both. Explain the output and the end of the program (with and without the checker). Add the copy constructor of the lecture, adapted to this class, and check that the copy is now deep. (The rule of three also asks for a copy assignment, `operator=`, which you do not have to write here.)
- (d) Replace `double *coef` by a `std::vector<double>` (`degree()` then becomes `coef.size() - 1`), and delete the destructor and the copy constructor. Check that copies are still deep: why do they need nothing written (rule of three)?

Q5. Fractions in C++. Turn the structure of Q2 into a class:

```
class Fraction {
    long long num_, den_;    // invariant: den_ > 0 and gcd(num_, den_) = 1
public:
    Fraction(long long num = 0, long long den = 1);
    long long num() const { return num_; }
    long long den() const { return den_; }
    double value() const;
};
```

- (a) Write the constructor, normalizing the fraction with `std::gcd` as `frac_make` did, in its body. Thanks to the default arguments, `Fraction f;` is 0 and `Fraction g(3);` is 3: check, and check that `Fraction h(6, -4);` is $-3/2$.
- (b) Can the invariant of Q2 still be broken from outside the class? Why does the class have getters but no setters?

Q6. Fractions in Python. In `fraction.py`, write a class `Fraction` whose constructor, with parameters `self`, `num`, and `den=1`, normalizes as in Q5, storing `self._num` and `self._den`, a method `value`, and two read-only accessors `numerator` and `denominator` (`@property`, no setter). What happens if you write `f._num = 4`? And `f.numerator = 4`? Compare with C++.

Part C: Operators and special methods

Q7. Special methods. Add to the Python `Fraction` of Q6 the special methods `__str__` ($-3/2$, or 3 when the denominator is 1), `__eq__`, `__lt__`, `__add__`, and `__mul__`, the arithmetic ones returning a new fraction.

- (a) Before writing `__eq__`, what does `Fraction(1, 2) == Fraction(2, 4)` give? (Without `__eq__`, `==` tells whether both sides are the same object, as `is` does.) And after?
- (b) Check that the first line below prints $5/6$. Then try the second one: why does it fail? (`sum` starts from 0.)

```
print(Fraction(1, 2) + Fraction(1, 3))
print(sum(Fraction(1, n) for n in range(1, 11)))
```

Make `__add__` and `__mul__` also accept an integer as their parameter `other`, which they then replace with `Fraction(other)`: test it with `isinstance(other, int)`. Then add `__radd__` and `__rmul__`, so that `f + 1`, `1 + f`, `2 * f`, and `sum` all work.

- (c) Compute H_{100} exactly. Why does Python not have the problem of Q2(c)? Compare with the standard module: `import fractions`, then `fractions.Fraction(1, 3)`.
- (d) Sort the list of fractions $3/4$, $-1/2$, $2/3$, $5/8$, $1/3$ with `sorted`, and take its `max`: which special method do they use?

Q8. Operator overloading. Add to the C++ `Fraction` of Q5 the operators `+`, `*`, `==`, and `<` as `const` methods, and a free function `operator<<` so that `std::cout << f` prints a fraction as in Q7.

- (a) Try `1 + f`: why does it not compile, with `operator+` a method? Then write a free function `Fraction operator+(long long n, const Fraction &f)`, as the lecture does for `3 * m`, and check. Which Python special method plays this role?

- (b) Store the five fractions of Q7(d) in a vector `v` of `Fractions`, sort it with the function `std::sort` of `<algorithm>`, called as `std::sort(v.begin(), v.end())`, and print it. Which operator does `std::sort` use?

Part D: Inheritance and polymorphism

Q9. A hierarchy of shapes, in Python. In `shapes.py`, write an abstract class `Shape` (ABC) with two abstract methods `area` and `perimeter`, a method `name` returning `"shape"`, and a method `__str__` returning, e.g., circle of area 3.14 and perimeter 6.28 (use `self.name()`, `self.area()`, and `self.perimeter()`). Then write its subclasses `Circle(radius)` and `Rectangle(width, height)`, and `Square(side)`, a subclass of `Rectangle` whose constructor calls that of `Rectangle` with `super()`, and which specializes `name` only.

- Print a circle, a rectangle, and a square. Which methods does `Square` inherit, which does it specialize? `__str__` is written once, in `Shape`: how does it print the right name and area for each class?
- Write two functions `total_area(shapes)` and `largest(shapes)` (the shape of largest area), taking a list of shapes, with no test on their class.
- Try `Shape()`. Then comment out the method `perimeter` of `Circle` and create a circle: when is the error detected? Put it back.
- Write a class `Disk`, *not* derived from `Shape`, with only a constructor and an `area`. Does `total_area` accept a list containing a disk? And a function summing the perimeters? What is this called?
- Is a square an instance of `Rectangle`, a rectangle an instance of `Square`? Predict, then check with `isinstance`; compare with `type(Square(2))` `is` `Rectangle`.

Q10. The same hierarchy in C++. Write `Shape` with two pure virtual methods `area` and `perimeter`, a virtual `std::string name() const` returning `"shape"`, and a virtual destructor printing `~Shape`; then `Circle`, `Rectangle` (with `protected` properties), and `Square : public Rectangle`, specializing `name` with `override`. Give `Rectangle` and `Square` destructors printing their names too.

- Write a function `void describe(const Shape &s)` printing the name, area, and perimeter of a shape, and a function `totalArea` taking a vector of pointers to shapes by `const` reference. In `main`, fill a `std::vector<Shape*>` with a circle, a rectangle, and a square created with `new`, describe them, print their total area, and `delete` them. In which order are the destructors called? Why does the vector hold pointers?
- Try `Shape s;`. Remove `virtual` in front of `name` in `Shape`: what does the compiler say? Remove the `overrides` too: what does `describe` print now? Put everything back.
- Write `void describeRectangle(Rectangle r)`, printing `r.name()`, and call it on a `Square`. What is printed, and why? Fix it by changing one character.
- Remove `virtual` in front of the destructor of `Shape`: read the warning, and see which destructors run on `delete`.

Part E: If you have time

Q11. Is a square a rectangle? Give the `Rectangle` of Q9 two accessors `width` and `height`, each with a setter refusing a value that is not positive, and use them in the constructor.

- Build `s = Square(2)`, set `s.width = 3`, and print `s`. What went wrong?
- Suppose `Square` specializes both setters so that they change both sides. A function `stretch(r)`, whose documentation promises to double the area of the rectangle `r`, runs `r.width = 2 * r.width`: what does it do to a square? Conclude: is a square whose sides can change a rectangle, in the sense of the lecture ("Inheritance or composition?")? What if neither class had setters?

Check

By the end of the session you should be comfortable with: a `struct` in C and the functions on it; a C++ class with private properties, a constructor, `const` methods, and, when it owns memory, a destructor and a copy constructor; a Python class with a few special methods; shallow and deep copies; a subclass specializing a method, and a function working on a whole hierarchy. If one of them is still unclear, ask us, or finish the corresponding exercises at home.