CS 112

Introduction to Data Structures

WEEK 03

Classes and Operator Overloading

Eric Araújo

Calvin University · Fall 2026

This Week

1. A class is a new type
2. Interface vs. implementation
3. `public` , `private` , `const`
4. Constructors and destructors
5. Test-driven development
6. Operator overloading **KEY**
7. Designing a class together

A Class Is a New Type

Week 0: the language gave you `int`, `double`, `char`. Now you make your own.

Built in

```
int    x = 7;
double d = 3.14;
```

Yours

```
Pair p(7, 11);
Enemy goblin("green", 3);
```

A class bundles **data** with the **operations** that make sense on it, and hides how the data is actually stored.

Interface and Implementation

Pair.h: what exists

```
class Pair {
 public:
  Pair();
  Pair(Item f, Item s);
  Item getFirst() const;
  Item getSecond() const;
 private:
  Item myFirst;
  Item mySecond;
};
```

Pair.cpp: how it works

```
#include "Pair.h"

Pair::Pair() {
  myFirst  = 0;
  mySecond = 0;
}

Item Pair::getFirst() const {
  return myFirst;
}
```

`Pair::` says "this function belongs to Pair". The header is what other files `#include`: the same split you met in week 0.

`public` and `private`

- **public**: anyone may use it. This is your promise to the outside world.
- **private**: only the class's own methods may touch it. Default for data.

```
Pair p(7, 11);
p.getFirst();     // ✓ public
p.myFirst;        // □ compile error: private
```

> Data stays private so you can *change how it's stored* later without breaking every program that uses your class. That freedom is the whole point.

TALK TO YOUR NEIGHBOR · TTYN

Is `other.x = x;` legal inside this class?

```
class XYZ {
 public:
  void copyXTo(XYZ other) {
    other.x = x;      // legal?
  }
 private:
  int x;
};
```

A. Yes: a method may touch private data of any object of its own class

B. No: `x` is private, so only `this` object's `x` is reachable

C. No: you cannot pass an object of the class to its own method

D. Only if `copyXTo` is declared a friend

`const` Methods

The third promise a class makes, right alongside `public` and `private`.

```
Item getFirst() const;   // promises not to change it
void setFirst(Item v);   // may change it
```

- The `const` goes **after** the parameter list
- Inside a `const` method, assigning to a member is a compile error
- Only `const` methods may be called on a `const` object

> Mark every method that only reads. It documents intent, and it's what lets your class be passed by const reference, the cheap, safe way from week 1. You'll see `const` on every getter from here on.

Constructors

```
class Pair {
 public:
  Pair();                          // default
  Pair(Item f, Item s);            // explicit values
  ...
};

Pair a;                            // calls Pair()
Pair b(7, 11);                     // calls Pair(Item, Item)
```

- Same name as the class, **no return type**
- Runs automatically when the object is created
- Its job: leave the object in a valid state: no uninitialized members

Destructors

```
class Pair {
 public:
  ~Pair();       // no args, no return
};
```

It runs automatically when the object dies:

- a local object reaches the end of its scope
- a `delete` is called on it
- the object it lives inside is destroyed

For `Pair` you don't need to write one. Next week, when objects start owning memory from `new`, you will, and forgetting it leaks.

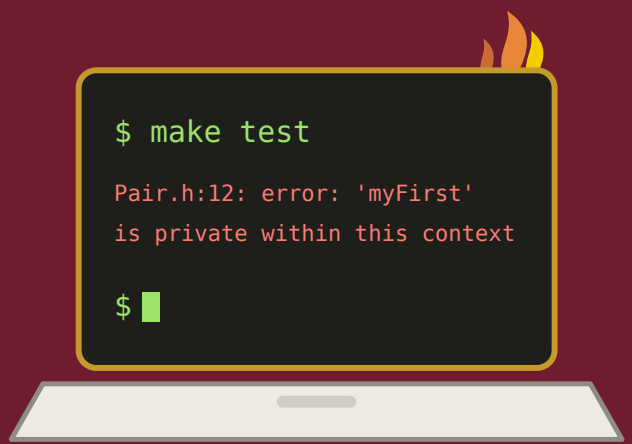
Write the Tests First

```
TEST_CASE("Pair stores what we give it") {
    Pair p(7, 11);
    REQUIRE(p.getFirst()  == 7);
    REQUIRE(p.getSecond() == 11);
}

TEST_CASE("default Pair starts at zero") {
    Pair p;
    REQUIRE(p.getFirst() == 0);
}
```

Writing the test first forces the question *what should this class do?* before *how do I build it?* The tests become the specification, and they stay useful when you change the implementation.

Demo



Building a class from its tests.

(private means private, even to the person who typed it)

Why Overload an Operator?

Without

```
p.print(cout);
cout << endl;
```

With

```
cout << p << endl;
```

Your type gets to behave like the built-in ones. Same idea as Python's `__str__` and `__eq__`: C++ just spells it with operators.

Overloading `<<`

```
void operator<<(ostream &out, const Pair &p) {
    out << "(" << p.getFirst() << ", "
        << p.getSecond() << ")";
}

cout << p;                  // ✓ works
```

Not a method, a free function. The left operand is the `ostream`, not your object.

Overloading `<<`

```
ostream& operator<<(ostream &out, const Pair &p) {
    out << "(" << p.getFirst() << ", "
        << p.getSecond() << ")";
    return out;                    // hand the stream back
}

cout << p << endl;          // ✓ now this works
```

`cout << p << endl` is really `(cout << p) << endl`. If the first call returns `void`, there's no stream left for `endl`. Returning `ostream&` is what makes chaining work.

TALK TO YOUR NEIGHBOR · TTYN

Why must `operator<<` return `ostream&` rather than `ostream`?

A. To avoid copying the stream, streams cannot be copied

B. So that `endl` can be used

C. Because `cout` is global

D. It makes no difference

Design One Together: `Enemy`

What data?

- colour
- speed
- hit points

What operations?

- take damage
- is it still alive?
- print it

With your neighbour: which members are `private`? Which methods are `const`? What is the *first test* you would write?

Week 03 Recap

- A class is a **new type**: data plus the operations that belong with it
- `.h` declares the interface, `.cpp` holds the implementation
- `private` data buys you the freedom to change how it's stored
- Constructors run on creation, destructors on death: both automatically
- Write the tests first; they are the specification
- Return `ostream&` from `operator<<` so output can chain

Next: a04 →

This deck stands on earlier CS112 materials by
Joel Adams and **Victor Norman** · adapted and extended by **Eric Araújo**