CS-1201 Object-Oriented Programming

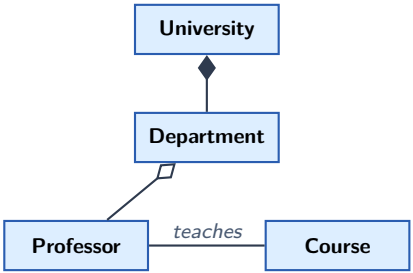
Relationships Between Classes

Arbish Akram

Department of Computer Science
Government College University

Objects Rarely Work Alone

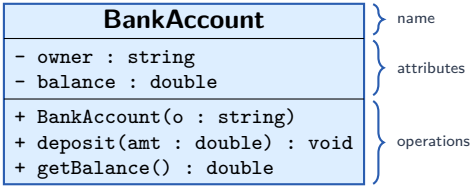
- So far, each lecture has focused on **one** class: its data, constructors, destructor and operators.
- Real programs contain many classes that **work together**. A `University` has `Departments`, a `Teacher` teaches a `Course`, a `Printer` prints a `Document`.
- The way two classes are connected decides how we write the C++ code: a parameter, a pointer member, or an object member.



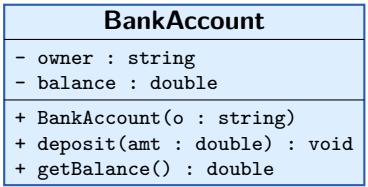
A **relationship** describes how objects of one class are connected to objects of another: who *knows* whom, who *owns* whom, and who *lives* how long. UML class diagrams give us a picture of these relationships before we write any code.

What Is a Class Diagram?

- The **Unified Modeling Language (UML)** is a standard graphical notation for designing object-oriented systems.
- A *class diagram* shows the classes of a system and the relationships between them.
- It is a **static** picture: it shows how the code is organised, not what happens while it runs.
- Each class is a box with three compartments:
  1. class name (bold, centred)
  2. attributes (data members)
  3. operations (member functions)
- The second and third compartments may be left out when only the relationships matter.



From UML to C++



Symbol	Visibility
+	public
-	private
#	protected

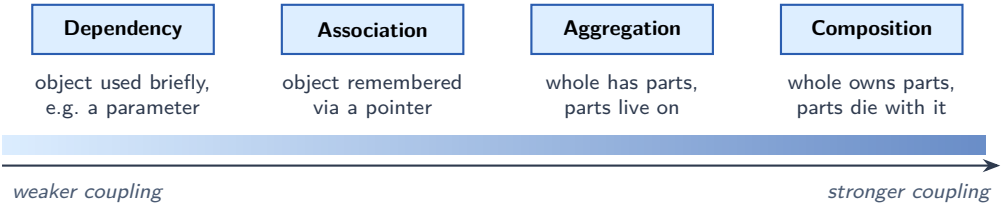
```cpp
class BankAccount {
private:
    string owner;           // - owner : string
    double balance{0.0};    // - balance : double
public:
    BankAccount(string o) : owner{o} { }
    void deposit(double amt) {
        balance += amt;
    }
    double getBalance() const {
        return balance;
    }
};
```

- UML writes `name : type`; C++ writes `type name`.
- UML also has ~ (package), which has no C++ equivalent.

The Relationship Notation at a Glance

Relationship	**UML line**	**Read as**	**Example**
Dependency		"uses-a" (temporarily)	`Printer` uses `Document`
Association		"knows-a"	`Teacher` — `Course`
Directed association		"knows-a" (one way)	`Driver` → `Car`
Aggregation		"has-a" (shared part)	`Department` has `Professors`
Composition		"consists-of" (owned)	`House` consists of `Rooms`
Generalization		"is-a"	`Dog` is an `Animal`
Realization		"implements"	`Circle` implements `Shape`

From Weak to Strong



- *Coupling* measures how much one class depends on another. The further right, the more a change in one class affects the other.
- Aggregation and composition are **special kinds of association**. When unsure, "association" is always a safe name.
- Prefer the **weakest** relationship that does the job.

Dependency: The “Uses-a” Relationship

```
class Document {
    string text;
public:
    Document(string t) : text{t} { }
    string getText() const { return text; }
};

class Printer {
public:
    // Document appears only as a parameter
    void print(const Document& doc) const {
        cout << "Printing: " << doc.getText()
             << endl;
    }
};

int main() {
    Printer printer;
    Document report{"Lab Report 3"};
    printer.print(report);   // USES report
    return 0;
}
```



- `Printer` has **no** `Document` data member. It meets a document only while `print` runs.
- If `Document` changes (e.g. `getText` is renamed), `Printer` must change too.

Output:

```
Printing: Lab Report 3
```

When Does a Dependency Arise?

A **dependency** exists when a change in one class (the *independent* one) may force a change in another (the *dependent* one). The dashed arrow points **to the class being used**.

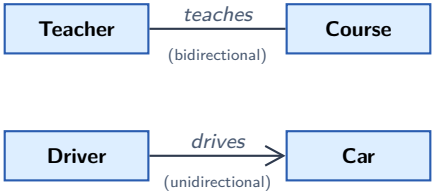
A class `A` depends on class `B` when `B` appears in `A` only:

- as a **parameter** type: `void print(const Document& d);`
- as a **return** type: `Receipt checkout();`
- as a **local variable**: `Logger log;` inside a member function.

The relationship is **temporary**: once the function returns, `A` no longer refers to the `B` object. becomes an association or stronger.

Association: The "Knows-a" Relationship

- An **association** is a lasting connection between objects of two classes. Drivers are related to cars, teachers to courses, patients to doctors.
- Typically, an object of one class **calls member functions** of an object of the other class, and **remembers** that object between calls.
- Both objects **exist independently**: neither creates nor destroys the other.
- In C++, an association is usually a **pointer** (or reference) data member.



Navigability: an open arrowhead shows which class knows the other. No arrowhead (or arrows on both ends) means each knows the other.

Unidirectional Association: Driver → Car

```cpp
class Car {
    string model;
public:
    Car(string m) : model{m} { }
    void start() const {
        cout << model << " started" << endl;
    }
};

class Driver {
    string name;
    Car* car{nullptr};        // Driver knows a Car
public:
    Driver(string n) : name{n} { }
    void assignCar(Car* c) { car = c; }
    void drive() const {
        if (car != nullptr) {
            cout << name << " is driving: ";
            car->start();
        }
    }
};
```

```cpp
int main() {
    Car civic{"Civic"};
    Driver ali{"Ali"};
    ali.assignCar(&civic);    // link
    ali.drive();
    return 0;
}
```

```
Ali is driving: Civic started
```

- Driver stores a **pointer**, not a Car: the car is created in main, not by the driver.
- Car has no idea a Driver exists: the association is one-way.
- nullptr means "no car yet", so drive() checks before using it.

Bidirectional Association: Teacher — Course

```cpp
class Course;                   // forward declaration
class Teacher {
    string name;
    Course* course{nullptr};  // OK: only a pointer
public:
    Teacher(string n) : name{n} { }
    string getName() const { return name; }
    void setCourse(Course* c) { course = c; }
    void showCourse() const;  // defined later
};
class Course {
    string title;
    Teacher* teacher{nullptr};
public:
    Course(string t) : title{t} { }
    string getTitle() const { return title; }
    void setTeacher(Teacher* t) { teacher = t; }
    void showTeacher() const {
        cout << title << " is taught by "
             << teacher->getName() << endl;
    }
};
```

```cpp
// Course is complete now
void Teacher::showCourse() const {
    cout << name << " teaches "
         << course->getTitle() << endl;
}
int main() {
    Teacher t{"Dr. Sana"};
    Course  c{"OOP"};
    t.setCourse(&c);    // link both
    c.setTeacher(&t);   // directions
    t.showCourse();
    c.showTeacher();
    return 0;
}
```

```
Dr. Sana teaches OOP
OOP is taught by Dr. Sana
```

Why the Forward Declaration?

- Each class mentions the other, so one of them must be used **before** it is defined.
- `class Course;` tells the compiler "`Course` is a class; details later". `Course` is now an *incomplete type*.
- With an incomplete type you may declare **pointers and references** to it, but you may not create an object of it or call its members.
- That is why `Teacher::showCourse()` is defined **after** `Course`.

```
class Course;           // incomplete type

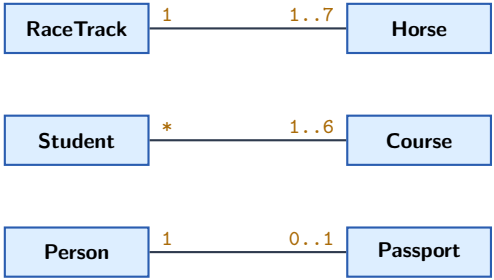
class Teacher {
    Course* c1;         // OK
    Course& c2;         // OK
    Course  c3;         // error: size unknown
    void f() {
        c1->getTitle();   // error: members
    }                     // not known yet
};
```

A bidirectional association means **two** links to keep consistent. Use one direction unless both are truly needed.

Multiplicity

Multiplicity states **how many** objects of each class take part in an association. It is written near each end of the line.

Symbol	Meaning
`1`	exactly one
`0..1`	zero or one
`*` or `0..*`	zero or more
`1..*`	one or more
`2..4`	two, three or four
`7,11`	seven or eleven



- One track runs 1 to 7 horses.
- A student takes 1 to 6 courses; a course has any number of students.
- A person holds at most one passport.

Read the number at the **far** end: "one `RaceTrack` is associated with `1..7 Horses`".

Association Names and Role Names



- An *association name* (**Executes**) describes the relationship. The small filled triangle gives the reading direction: "one `ATM` executes zero or one `Withdrawal`".
- A *role name* (*currentTransaction*) describes the part an object plays in this association. It often becomes the name of the pointer data member: `Withdrawal* currentTransaction;`
- A class can play different roles in different associations: a `Person` may be a *professor* to students and a *colleague* to other professors.
- Names are optional. Leave them out when the meaning is obvious.

Implementing Multiplicity in C++

```
class Horse {
    string name;
public:
    Horse(string n) : name{n} { }
    string getName() const { return name; }
};
class RaceTrack {
    static const int MAX = 7;  // 1..7 horses
    Horse* horses[MAX];        // links, not objects
    int count{0};
public:
    bool addHorse(Horse* h) {
        if (count == MAX) return false;  // full
        horses[count++] = h;
        return true;
    }
    void showLineUp() const {
        for (int i = 0; i < count; ++i)
            cout << i + 1 << ". "
                 << horses[i]->getName() << endl;
    }
};
```

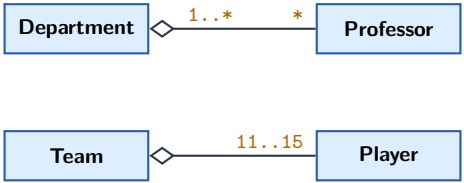


Multiplicity	Typical member
1	`B* b;` (always set)
0..1	`B* b{nullptr};`
0..n	`B* arr[n]; int count;`
*	a growable list of `B*`

The upper limit is enforced **by our code** (`count == MAX`), not by the compiler.

Aggregation: The "Has-a" Relationship

- **Aggregation** is a special association that models a **whole–part** relationship: a library *has* books, a team *has* players.
- The whole is "organisationally superior" to its parts, but it does **not own** them:
  - parts are created **outside** the whole,
  - parts **survive** when the whole is destroyed,
  - a part may be **shared** by several wholes.
- Example: a computer has a monitor, but the same monitor can be plugged into another computer, and it still exists when the computer is thrown away.



The **hollow diamond** sits at the **whole**. A professor may belong to more than one department.

Aggregation in C++: Department and Professor

```cpp
class Professor {
    string name;
public:
    Professor(string n) : name{n} { }
    ~Professor() { cout << "Professor " << name
                        << " destroyed" << endl; }
    string getName() const { return name; }
};
class Department {
    string title;
    Professor* staff[5];     // parts are NOT owned
    int count{0};
public:
    Department(string t) : title{t} { }
    ~Department() {          // note: no delete
        cout << "Department " << title
             << " closed" << endl;
    }
    void hire(Professor* p) {
        if (count < 5) staff[count++] = p;
    }
    void show() const;       // prints all names
};
```

```cpp
int main() {
    Professor p1{"Sara"}, p2{"Usman"};
    {
        Department cs{"CS"};
        cs.hire(&p1);
        cs.hire(&p2);
        cs.show();
    }   // cs destroyed here
    cout << p1.getName()
         << " still exists" << endl;
    return 0;
}
```

Output:

```
CS: Sara Usman
Department CS closed
Sara still exists
Professor Usman destroyed
Professor Sara destroyed
```

Reading the Aggregation Output

1. `p1` and `p2` are created in `main`, **before** the department exists.
2. The department stores only their **addresses**.
3. At the closing brace, `cs` is destroyed. Its destructor does **not** `delete` the professors.
4. The professors are still alive and are destroyed at the end of `main`, in reverse order of creation.

The whole must not delete what it does not own.

```
~Department() {
    for (int i = 0; i < count; ++i)
        delete staff[i];    // WRONG
}
```

`p1` and `p2` were not created with `new`. Deleting them is undefined behaviour, and a professor shared with another department would be destroyed twice.

The opposite danger: if a professor is destroyed while a department still holds its address, that pointer becomes *dangling*. The parts must outlive every whole that refers to them.

A Common Mistake: Storing a Copy

Looks like aggregation, is not

```
class Car {
    Engine engine;          // an OBJECT
public:
    Car(const Engine& e) : engine{e} { }
};
Engine v8;
Car car{v8};    // car.engine is a COPY of v8
```

- The copy constructor makes a **new** engine inside the car.
- Changing v8 later does not affect the car, and the car's engine dies with the car.
- That is **composition** with a copied initial value.

Real aggregation

```
class Car {
    Engine* engine;         // a LINK
public:
    Car(Engine* e) : engine{e} { }
};
Engine v8;
Car car{&v8};   // car refers to v8 itself
```

- The car refers to the **same** engine object as v8.
- The engine can be moved to another car and outlives this one.

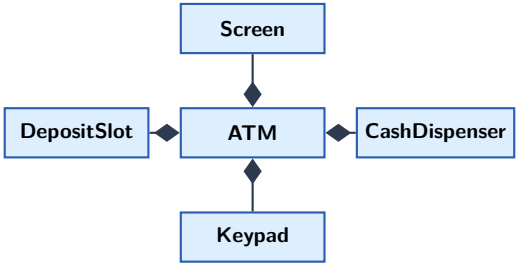
In C++, the data member's type tells the story: an **object** member means the whole owns it; a **pointer or reference** member means it is only linked to it.

Composition: The "Consists-of" Relationship

Composition is a stronger form of aggregation. In UML it has three properties:

1. Only **one** class is the whole: the diamond is on one end only.
2. The parts **live only as long as the whole**. The whole creates its parts and destroys them.
3. A part belongs to **only one whole at a time**.

A car consists of doors and an engine; a room consists of a floor, a ceiling and walls. If any property fails, use aggregation instead.



The **filled diamond** sits at the **whole**. An ATM consists of one screen, one keypad, one deposit slot and one cash dispenser.

Composition in C++: Member Objects

```cpp
class Room {
    string name;
public:
    Room(string n) : name{n} { }
    void describe() const {
        cout << "  - " << name << endl;
    }
};
class House {
    string address;
    Room livingRoom;      // House owns its Rooms
    Room bedroom;
public:
    House(string a, string r1, string r2)
        : address{a}, livingRoom{r1},
          bedroom{r2} { }
    void describe() const {
        cout << "House at " << address << endl;
        livingRoom.describe();
        bedroom.describe();
    }
};
```




```cpp
House home{"12 Mall Road",
           "Living Room",
           "Master Bedroom"};
home.describe();
```

```
House at 12 Mall Road
  - Living Room
  - Master Bedroom
```

- The rooms are **inside** the house object; nobody else can hold them. They are built and destroyed with it.
- `livingRoom{r1}` in the *member initializer list* calls `Room(string)`.

Constructor and Destructor Order

```
class Date {
    int day, month, year;
public:
    Date(int d, int m, int y)
        : day{d}, month{m}, year{y} {
        cout << "Date constructor: ";
        print();  cout << endl;
    }
    ~Date() {
        cout << "Date destructor: ";
        print();  cout << endl;
    }
    void print() const {
        cout << day << '/' << month
             << '/' << year;
    }
};
```

```
class Employee {
    string name;
    Date birthDate;       // member objects
    Date hireDate;
public:
    Employee(string n, int bd, int bm, int by,
                       int hd, int hm, int hy)
        : name{n}, birthDate{bd, bm, by},
          hireDate{hd, hm, hy} {
        cout << "Employee constructor: "
             << name << endl;
    }
    ~Employee() {
        cout << "Employee destructor: "
             << name << endl;
    }
};
```

`Employee` is the *host* object; `birthDate` and `hireDate` are its member objects.

Running the Employee Program

```
int main() {
    Employee e{"Bob", 24, 7, 1949,
                      12, 3, 1988};
    cout << "--- end of main ---" << endl;
    return 0;
}
```

Objects are built **inside-out** (parts first) and destroyed **outside-in** (whole first): exactly the reverse order.

Output:

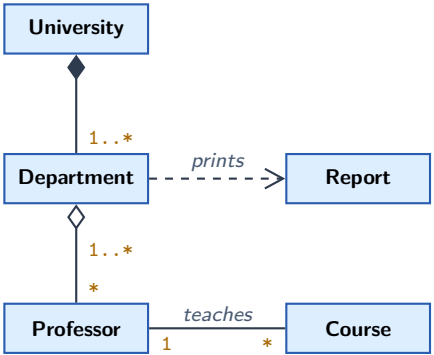
```
Date constructor: 24/7/1949    1st member
Date constructor: 12/3/1988    2nd member
Employee constructor: Bob      host body
-- end of main --
Employee destructor: Bob       host body
Date destructor: 12/3/1988     2nd member
Date destructor: 24/7/1949     1st member
```

Aggregation vs. Composition

	Aggregation ◇—	**Composition** ◆—
Phrase	"has-a"	"consists-of" / "part-of"
Ownership	whole does **not** own parts	whole **owns** parts
Part's lifetime	independent of the whole	same as the whole
Sharing	part may belong to several wholes	part belongs to one whole
Who creates part?	someone else, before or after	the whole
Typical C++	`Part* p;` (no `delete`)	`Part p;` or `new`/`delete` in the whole
Example	`Department` / `Professor`	`House` / `Room`

The test: destroy the whole. If the parts must go too, it is composition. If they carry on, it is aggregation.

Putting It Together: A University



- **Composition:** if the university closes, its departments disappear. They depend on it for their existence.
- **Aggregation:** if a department closes, its professors do not disappear. They can join another department.
- **Association:** a professor teaches courses, but neither owns the other.
- **Dependency:** a department uses a `Report` object only while printing it.

Mapping Relationships to C++

Relationship	**How `A` refers to `B`**	**Who manages `B`'s lifetime?**
Dependency	parameter, return type or local variable	the caller
Association	data member `B* b;` or `B& b;`	someone outside `A`
Aggregation	data member `B* b;` (often many)	someone outside `A`
Composition	data member `B b;`	`A` (automatically)
	or `B* b;` with `new` / `delete` in `A`	`A` (explicitly)
Generalization	`class A : public B { ... };`	`B` is part of every `A`

Association and aggregation produce the **same code**. The difference is in meaning: aggregation adds the idea that `A` is a whole and `B` is one of its parts.

Check Your Understanding

Name the relationship and draw its UML line.

1. A `Calculator` has a member function `double add(const Fraction& a, const Fraction& b)`.
2. A `Library` holds pointers to `Books` that are created elsewhere and can be lent to other libraries.
3. A `Human` has a `Heart` data member of type `Heart`.
4. A `Patient` stores a pointer to their `Doctor`, and the `Doctor` stores pointers to all their `Patients`.
5. A `SavingsAccount` is a `BankAccount` with an interest rate.
6. An `Order` creates its `OrderLines` with `new` and deletes them in its destructor.

Check Your Understanding: Answers

	Relationship	UML	Reason
1	Dependency		`Fraction` is only a parameter type
2	Aggregation		whole–part, but books live on and can be shared
3	Composition		object member: built and destroyed with the human
4	Association		both directions, neither owns the other; 1 to *
5	Generalization		"is a": `class SavingsAccount : public BankAccount`
6	Composition		the order creates and destroys its lines

Summary

- UML class diagrams show classes as three-compartment boxes and relationships as lines; `+`, `−`, `#` mark visibility.
- **Dependency** (dashed arrow): `A` uses `B` briefly, as a parameter, return type or local variable.
- **Association** (line, optional arrow): `A` remembers `B` through a pointer. Multiplicity, association names and role names add detail; two-way links need a forward declaration.
- **Aggregation** (hollow diamond): a whole–part association where parts are created elsewhere, may be shared, and outlive the whole. Never `delete` parts you do not own.
- **Composition** (filled diamond): the whole owns its parts. Usually an object member, constructed before the whole and destroyed after it, initialized in the member initializer list.