Relationships Between Objects

- In the real world, there are relationships between objects.
  Examples:
  - Students enroll in courses.
  - Classes have classrooms.
  - Professors have a list that contains the courses they offer.
  - The university consists of faculties, and faculties consist of departments.
  - The dean of the faculty is a professor.
  - A Ph.D. student is a kind of student.

- The objects can cooperate (interact with each other) to perform a specific task.
  Examples:
  - A professor can get the list of the students from the course object.
  - A student can get her grades from the related course objects.
  - A university can send an announcement to all faculties, and faculties can distribute this announcement to their departments.

Relationships Between Objects (cont'd)

- In object-oriented design (OOD), we try to **lower the representational gap** between real-world objects and the software components.
- This makes it easier to understand what the code is doing.
- To represent real-world relationships, we also create relationships between software objects.

Types of relationships in object-oriented design (OOD):

- There are two general types of relationships, i.e., **association** and **inheritance**.
  - **Association** is also called a "**has-a**" ("uses") relationship.
  - **Inheritance** is known as an "**is-a**" relationship.

  Examples:
  - A course **has a** classroom.
  - The dean of the faculty **is a** professor.
- In this section, we will cover association, aggregation, and composition.
  While association itself is a general "uses-a" relationship, its subtypes, **aggregation** and **composition** are forms of the has-a relationship.
- Inheritance ("is-a" relationship) will be covered in the coming sections.

Association ("**uses-a**" relationship):

Association means instances of class A **can use** services given by class B.

- Instances of A know instances of B.
  Programming: Class A has pointers (or references) to objects of class B.
- The relationship may be unidirectional or bidirectional (where the two objects are aware of each other).
  If the relationship is bidirectional, class B also has pointers (or references) to objects of class A.
- Instances of A and B can communicate with each other.
  Instances of class A can send messages to instances of another class B.
  Programming: Objects of class A can call methods of objects of class B.
- There may be one-to-one, one-to-many, or many-to-many associations between objects.
- The objects that are part of the association relationship can be created and destroyed independently.
  Each of these objects has its own life cycle.
  Programming: The constructor of a class does not have to call the constructor of the other class.
  The destructor of a class does not have to call the destructor of the other class.
- There is no "owner".

Association (cont'd):

Example:

Students register for courses.

Real World:

- A student can enroll in multiple courses.
- A course can have multiple students enrolled in it, and students can enroll in several courses (bidirectional).
- A student is associated with multiple courses. At the same time, one course is associated with multiple students (many-to-many).
- Students can get their grades from the course.
- Courses also can access some information about students, such as their IDs.
- Each of these objects has its own life cycle.
  The department can create new courses. In this case, new students are not created.
  When a course is removed from the department's plan, the students are not destroyed.
  Students can add or drop courses.

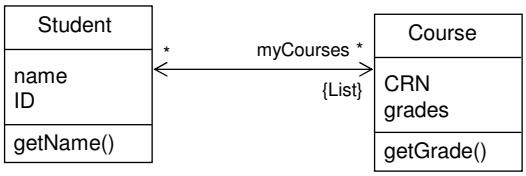
Example (cont'd):
Students register for courses.

Software:

- The `Student` class can have a collection (e.g., array, list) of `Course` objects.
- A `Course` class can also have a collection of the `Student` objects enrolled in that course (bidirectional).
- A `Student` object can call methods of `course` classes, for example, to get the grade.
- If there is a bidirectional relation, the `Course` class can also call the methods of the `Student` class.
- Each of these objects has its own life cycle.
  The `Student` class does not have to create or destroy `Course` objects.
  The `Course` class does not have to create or destroy `Student` objects.

Example: Association between students and courses:

UML Notation:
Software class diagram



Summary:

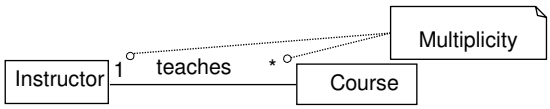
- An association is a weak "uses-a" relationship between two or more objects in which the objects have their own lifetimes, and **there is no owner**.

UML Class diagrams for association:

- UML class diagrams can also be used to present real-world conceptual classes.
- If the direction of messages is unspecified, both classes may send messages to each other.

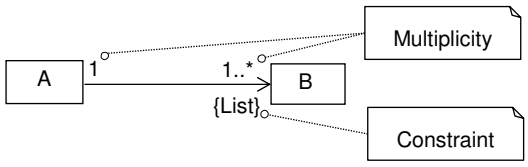
Multiplicity:

- Multiplicity defines how many objects of one class can be associated with one object of another class.
- In other words, it shows the number of objects from that class that can be linked at runtime with one instance of the class at the other end of the association line.



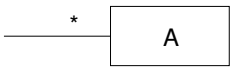
An instructor teaches zero or more courses (read from left to right).
An association may also be read in the reverse direction.
A course is taught exactly by one instructor (read from right to left).

Examples:
Multiplicity



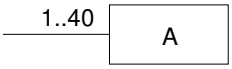
One object of class A is associated with one or more objects of class B at a time.

Class A includes **a list** that can contain one or more objects of class B.
We could have `{Vector}` instead of `{List}` as a constraint.

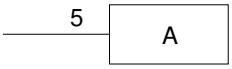


Zero or more, many

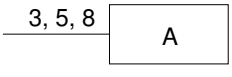
One or more



One to forty



Exactly five



Exactly 3, 5, or 8

Aggregation:

- Aggregation is a specialized form of association between two or more objects.
- It indicates a "Whole/Part" ("has-a") relationship.
- While each object has its own life cycle, there is also an ownership relationship between them.
- An object (or part) can belong to multiple objects (whole/owner) simultaneously.
- The whole (i.e., the owner) can exist without the part and vice versa.
- The relation is unidirectional. The whole owns the part(s), but the part does not own the whole.

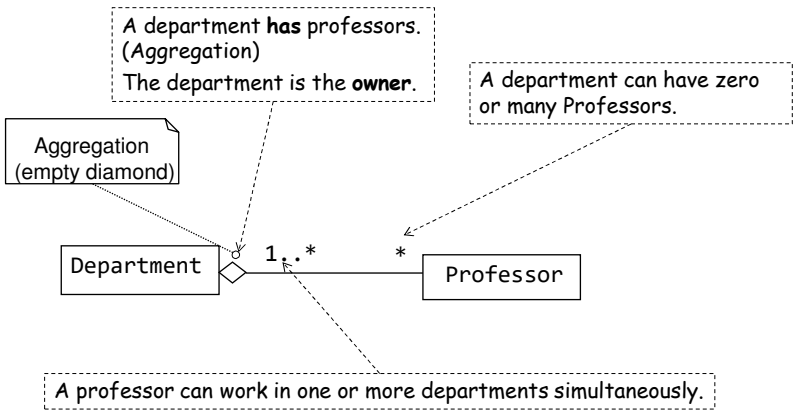
Example:

- A department of the faculty **has** professors.
  - A professor may belong to more than one department at some universities.
  - Parts (professors) can still exist even if the whole (the department) does not exist.
  - If all professors retire or resign, the department can still exist and wait for new professors.
  - A department may own a professor, but the professor does not own the department.

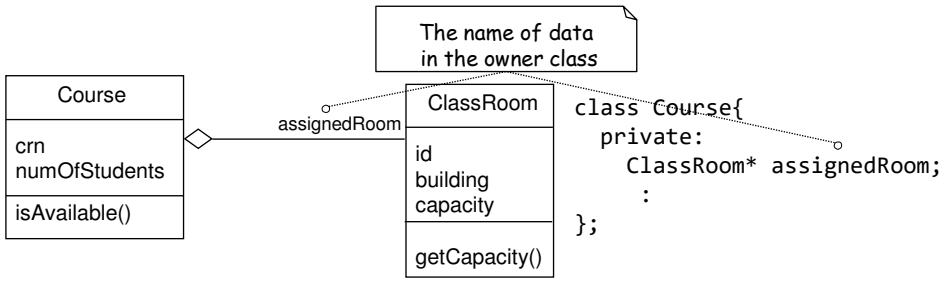
Example (cont'd):

A department **has** professors.

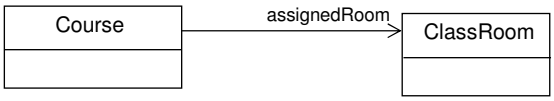
In UML diagrams, we use an empty diamond to present the aggregation relationship.



Example: An active course **has a** classroom.



- If the owner is clearly identified, there is no need to indicate it on the diagrams.
- We can also use an arrow to represent aggregation as we did for association.
- Remember: Aggregation is a special type of association where there is an owner.



- In C++, the implementations of association and aggregation relationships are quite similar.

Example: An active course **has a** classroom (cont'd)

```
class ClassRoom {                    // Declaration/definition of the Classroom
public:
    :
   unsigned int getCapacity() const { return m_capacity; }
private:
  std::string m_building;
  std::string m_id;
  unsigned int m_capacity{};         // capacity initialized to zero
};


class Course {
public:
  // Initialize crn, number of students, and the classroom
  Course(const std::string&, unsigned int, const ClassRoom*);
  bool isAvailable() const;          // Are there available seats?
private:
  :
  const ClassRoom* m_classRoom;      // The course has a classroom
};
```

Constructor gets the address of the assigned classroom.

Course has a pointer to ClassRoom objects.

Example: An active course **has a** classroom (cont'd)

```
// Constructor to initialize crn, number of students, and the classroom
Course::Course(const std::string& in_crn, unsigned int in_numOfStudents,
                                                    const ClassRoom* in_classRoom )
        : m_crn{ in_crn }, m_numOfStudents{ in_numOfStudents }, m_classRoom{ in_classRoom }
{}
```

A Course object **does not create** or **delete** ClassRoom objects. Each object has its own life cycle.

The pointer in the Course object points to the ClassRoom object.

```
bool Course::isAvailable() const {
  return m_classRoom->getCapacity() > m_numOfStudents;
}
```

The Course object calls the method of the ClassRoom.

Example e06_1.cpp

```
int main(){
  ClassRoom classRoom1{ "BBF", "Z-16", 100 };        // Classroom is created
  Course BLG252E{ "23135", 110, &classRoom1 };       // Course is created
  if (BLG252E.isAvailable()){
    room_id = BLG252E.getClassRoom()->getId();       // Chain of function calls
   ...
```

Returns the pointer to the ClassRoom object.

getId() of the ClassRoom is called.

Composition:

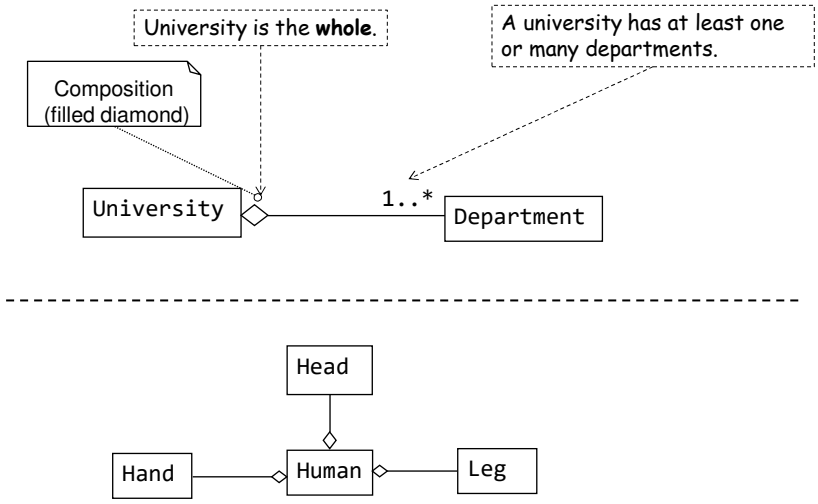
- The Composition is also a specialized form of association and a specialized form of aggregation. Composition is a **strong** kind of "has-a" relationship.
- It is also called a "part-of" or "belongs-to" relationship.
- There is an owner.
- The objects' lifecycles are tied.
  - The part object (e.g., room) cannot exist without the owner/whole (e.g., house).
  - The whole and part objects are created together.
  - Constructors in C++ will ensure the creation of the parts when the owner is created.
  - When the owner object is deleted, the part objects are also deleted.
- The relation is unidirectional.

Examples:

- A university is composed of departments, or departments are parts of a university.
- A rectangle is composed of four points.
- Rooms belong to a house.

Composition (cont'd):

In UML diagrams, a filled diamond represents composition.



Order of creation and destruction of objects in a composition relationship

- The parts (member objects) are constructed before their enclosing class objects (wholes/owners).
- When the whole (owner) object goes out of scope, the destructors are called in the reverse order of creation, i.e., the whole object is destroyed before the member objects (parts).

Default Constructors and Destructors in a composition relationship:

- If the `Whole` and `Part` classes have default constructors and destructors, they are automatically invoked in the proper order.
  - Constructors of the part objects run before the constructor of the whole object.
  - The part objects are constructed in the order in which they are declared in the class definition.
  - At the end of the scope, the destructor of the whole object runs before the destructors of the parts.

Example: `Whole` has two `Parts`

Example e06_2a.cpp

```
Part Constructor
Part Constructor
Whole Constructor
------------------
Whole Destructor
Part Destructor
Part Destructor
```

- Two part objects are created before the whole object.
- At the end of the program, the whole object is destroyed before the parts.

- Remember: If the class creator does not provide constructors, the compiler supplies a default default constructor.
- Members can also be initialized in the class definition.

```
int m_data{ 5 };
```

Example e06_2b.cpp

Order of creation and destruction of objects in a composition relationship (cont'd)

Constructors with parameters:

- If the `Part` class contains constructors that take parameters (instead of a default constructor), the `Whole` class **must initialize the `Part` object(s)** using one of the following two techniques:

  A. Initializing part objects in the class definition of the `Whole`.

  Example: `Part` class has a constructor that receives two parameters

```
class Whole{
  :
private:
  Part m_part1 {1, 2}, m_part2 {3, 4};
};
```

Example e06_3a.cpp

The initial values are determined by the **creator of the `Whole` class**.

OR,

  B. The `Whole` class **must have a constructor** that calls one of the `Part` class's constructors in its **member initializer list** (not in the body).

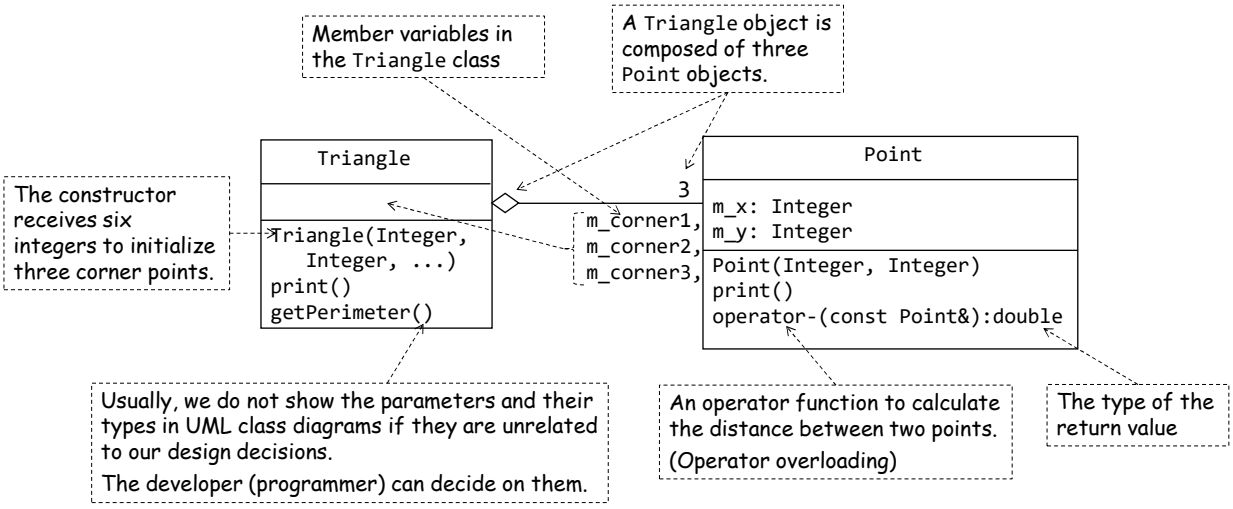
```
Whole::Whole(int in1, int in2, int in3, int in4) : m_part1{in1, in2}, m_part2{in3, in4}
{}
```

The initial values are determined by the **user of the `Whole` class**.

Example e06_3b.cpp

- The program does not compile if the `Whole` does not initialize the `Part` objects.

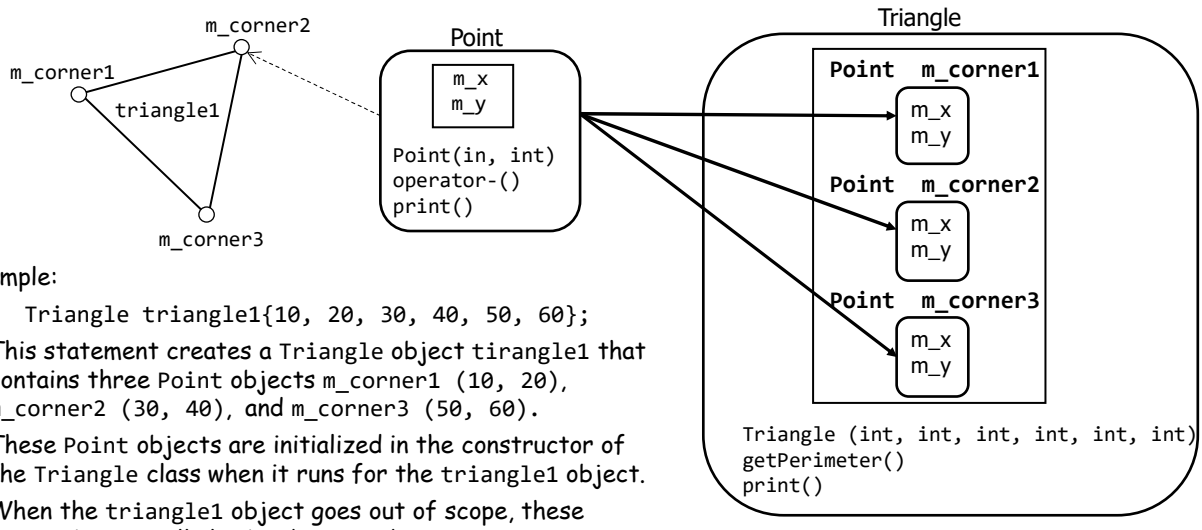
Example: A triangle is **composed of** three `Point` objects



These classes can also have additional constructors with different parameters, e.g., copy constructors.

Example: A triangle is composed of three `Point` objects (cont'd)

A `Triangle` object in memory:



Example:

```
Triangle triangle1{10, 20, 30, 40, 50, 60};
```

- This statement creates a `Triangle` object `tirangle1` that contains three `Point` objects `m_corner1 (10, 20)`, `m_corner2 (30, 40)`, and `m_corner3 (50, 60)`.
- These `Point` objects are initialized in the constructor of the `Triangle` class when it runs for the `triangle1` object.
- When the `triangle1` object goes out of scope, these `Point` objects will also be destroyed.

Example: A triangle is composed of three `Point` objects (cont'd).

```
class Point {
public:
  Point(int, int);                  // Constructor to initialize x and y coordinates
    :
private:
  int m_x{ MIN_x }, m_y{ MIN_y };   // x and y coordinates
};
```

Since the `Point` class has a constructor that receives two parameters, the constructor of the `Triangle` class must supply these arguments.

```
class Triangle {
public:
                                    // Constructor with the coordinates of three corners
  Triangle(int, int, int, int, int, int);
    :
private:
                                    // Corners of the triangle are three Point objects
   Point m_corner1, m_corner2, m_corner3;  // Composition
};
```

- When a `Triangle` object is created, these variables (`m_corner1`, `m_corner2`, and `m_corner3`) will also be created.
- When a `Triangle` object goes out of scope, these automatic variables will be destroyed.

Example: A triangle is composed of three `Point` objects (cont'd)

- The creator of **the `Triangle` class calls the constructors of the `Point` class** to initialize `Point` objects.
- The constructor of the `Triangle` class must call one of these constructors in the **member initializer list** (not in the body).

```
// Constructor of Triangle with the coordinates of three corners
Triangle::Triangle(int corner1_x, int corner1_y, int corner2_x,
                   int corner2_y, int corner3_x, int corner3_y)
                      : m_corner1{ corner1_x, corner1_y }, m_corner2{ corner2_x, corner2_y },
                        m_corner3{ corner3_x, corner3_y }
{} // The body can be empty
```

The constructor of the `Point` is called three times.

- This constructor takes the x and y coordinates of three corner points (six integers) and calls the constructor of the `Point` class three times, once for each corner point.

Example e06_4a.cpp

```
int main() {
 Triangle triangle1{10, 20, 30, 40, 50, 60}; // The points are created before the triangle
  :
 return 0;                                   // The triangle is destroyed before the points
}
```

When `triangle1` goes out of scope, the member objects (`m_corner1`, `m_corner2`, and `m_corner3`) and the `triangle1` object are destroyed automatically.

Example: A triangle is composed of three `Point` objects and the `Point` class contains a copy constructor

- In this example, we assume that the `Point` class contains a **copy constructor**.

```
Point(const Point&);
```

- The `Triangle` class can have another constructor that receives references to three existing `Point` objects.

```
// Constructor receives references to three points
Triangle::Triangle(const Point& in_corner1, const Point& in_corner2, const Point& in_corner3)
              : m_corner1{ in_corner1 }, m_corner2{ in_corner2 }, m_corner3{ in_corner3 }
{}
```

The copy constructor of the `Point` is called.

- This constructor calls the copy constructor of the `Point` class three times, once for each corner point.
- The member points of the triangle are initialized as copies of the input points.

Example e06_4b.cpp

```
int main() {
  Point point1{0, 20};                              // 1. Point object
  Point point2{10, 20};                             // 2. Point object
  Point point3{30, 40};                             // 3. Point object
  Triangle triangle2{ point1, point2, point3 };     // Existing Point objects are sent
     :
  return 0;
}
```

Members are initialized as copies of the existing input points.

All objects (`triangle2`, `m_corner1`, `m_corner2`, `m_corner3`, `point1`, `point2`, and `point3`) are destroyed automatically.

Example: A triangle is composed of three `Point` objects (cont'd)

- Since the `Point` class does not contain a default constructor in these examples, the author of the `Triangle` class **cannot** create and initialize corner points as follows:

```
// Constructor that calls the default constructor of the Point
Triangle::Triangle() : m_corner1{}, m_corner2{}, m_corner3{}
{}
```

The Point class must contain a default constructor.

or

```
// Constructor that calls the default constructor of the Point
Triangle::Triangle()
{}                          //Error! If the Point does not contain a default constructor
```

- Remember: The class creator sets the rules, and the class user must follow them.
- In our examples, the `Triangle` class is the user of the `Point` class.

Collaboration between objects:

- Objects of the whole (owner) can use their members' `public` methods (services) to fulfill their tasks.

Example: Objects of the `Triangle` use `public` methods of their member points.

```
// Calculates and returns the perimeter of the triangle
double Triangle::getPerimeter()const {     // using operator overloading (operator-)
   return (m_corner2 - m_corner1) + (m_corner3 - m_corner2) + (m_corner1 - m_corner3);
```

or without operator overloading

```
   return  m_corner1.getDistance(m_corner2) +
           m_corner2.getDistance(m_corner3) +
           m_corner3.getDistance(m_corner1);     // without operator overloading
}

// Prints the corners of a triangle
void Triangle::print()const {
   std::println("Corners of the triangle:");
   m_corner1.print();  m_corner2.print();  m_corner3.print();
}
```


Dynamic member objects (Pointers as members) in composition

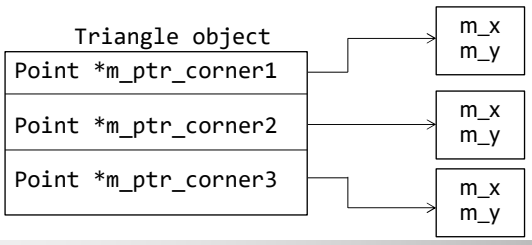
- Instead of automatic objects, data members of a class may also be pointers to objects of other classes (parts).

Example: The `Triangle` class contains pointers to `Point` objects.

```
class Triangle {
   :
private:

 Point *m_ptr_corner1, *m_ptr_corner2, *m_ptr_corner3;       // Pointers to the corners
};
```

Now, only the pointers (addresses) of `Point` objects are contained in the objects of the `Triangle`.



Dynamic member objects (Pointers as members) in composition (cont'd)

- If the relationship is composition, the whole must create (allocate memory) and initialize part objects in the constructor.

Example: The `Triangle` class contains pointers to `Point` objects.

```
// Constructor with the coordinates of three corners
Triangle::Triangle(int corner1_x, int corner1_y,int corner2_x,int  corner2_y,
                   int corner3_x, int corner3_y)
                   :  m_ptr_corner1{ new Point{corner1_x, corner1_y} },
                      m_ptr_corner2{ new Point{corner2_x, corner2_y} },
                      m_ptr_corner3{ new Point{corner3_x, corner3_y} }
{}
```

Memory is allocated for newly created objects.

The constructor of the `Point` is called.

- If the relationship is composition and memory is allocated in the constructor, then these memory locations must be released (in most cases) in the destructor.

```
// Destructor
Triangle::~Triangle() {
   delete m_ptr_corner1;
   delete m_ptr_corner2;
   delete m_ptr_corner3;
}
```

The destructor of the `Point` is called.

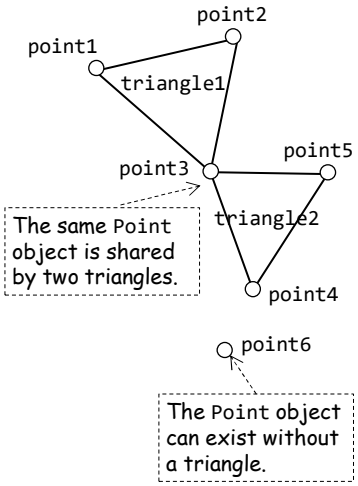
Example e06_4c.cpp

Deciding between aggregation and composition

- We can determine the relationships between objects in the system based on the requirements.

Example: A triangle is an aggregation of three `Point` objects

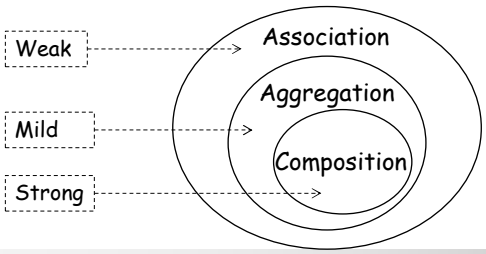
- Requirements: `Point` objects can belong to multiple triangles. Points can exist without triangles.
- Based on the requirements, the relationship between `Point` and `Triangle` can shift from composition to aggregation.
- If the relationship is aggregation, the owner will not create or destroy member objects.
- The `Triangle` will contain pointers to corner points.
- The point objects will be created outside the Triangle.
- The constructor of the `Triangle` will get the addresses of its corner points.
- The corner points are not created or destroyed by the `Triangle`.
- This relationship is similar to the relationship between `Course` and `ClassRoom`.
- Remember: Example e06_1.cpp



Summary: Association, Aggregation, Composition

Property	Association	Aggregation	Composition
Relationship type	Otherwise unrelated	Whole/part	Whole/part
Relationship verb	Uses-a	Has-a	Part-of
Members can belong to multiple classes	Yes	Yes	No
Members' existence managed by owner	No	No	Yes
Directionality	Unidirectional or bidirectional	Unidirectional	Unidirectional

- The key aspect of programming is determining when and how objects are created and destroyed.
- One of these relationships can be used depending on the requirements.



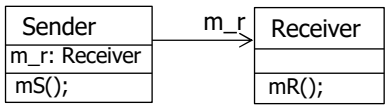
- Example 1: A triangle can be **composed** of three points.
  In that case, the points will be created and destroyed by the owner (triangle). Examples: e06_4a.cpp and e06_4b.cpp
- Example 2: A triangle can be an **aggregation** of three points.
  In that case, the points will be created and destroyed outside the owner (triangle).

Visibility Between Objects

- Visibility means that one object can "see" or have a reference to another object.
- To send a message to another object, the sender must have a reference or pointer to the receiver object.
- How can the Sender call the Receiver's mR() method?

  The sender must be able to "see" the receiver.

```
ref.mR();     // if ref is an object name
ref->mR();    // if ref is a pointer to an object
```

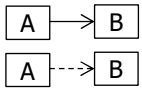


- When designing a system as a set of interacting objects, it is essential to ensure that the necessary visibility is achieved between these objects to facilitate message interaction.

Types of visibility:

There are four ways that visibility can be established from object A to object B:

- **Attribute visibility:** B is an attribute of A.
- **Parameter visibility:** B is a parameter of a method of A.
- **Local visibility:** B is a (non-parameter) local object within a method of A.
- **Global visibility:** B is in the global space of A.



Types of visibility:

Example: Attribute visibility

- In the example e06_1.cpp, the `Course` class has a pointer to its classroom.

```
class Course{
  private:
    ClassRoom* assignedRoom;  // The course has a classroom
     :
};
```

- In the main function, we create the object of the `ClassRoom` and send its reference to the constructor of the `Course` object to establish the attribute visibility from the `Course` object to the `ClassRoom` object.
- Now, the `Course` object can "see" the `ClassRoom` object.

```
ClassRoom classRoom1{ "BBF", "Z-16", 100 }          // ClassRoom object is created
Course BLG252E{ "23135", 110, &classRoom1 };        // Visibility
```

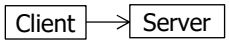
Example:

- In examples e06_4a.cpp, e06_4b.cpp, and e06_4c.cpp, corner points of the `Triangle` are created and initialized in the constructor of the `Triangle` class.
- There is attribute visibility from the `Triangle` to the corner objects.

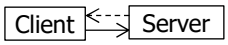
Example: Parameter visibility

Sending `this` as an argument to establish visibility:

- In an object-oriented program, a class (`Client`) may get services from another class (`Server`) by calling its methods.



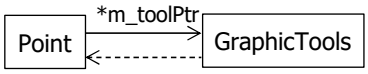
- The `Server` class may also need to access the members of the `Client` class to provide these services.
- If this is the case, the `Client` object can send its address (`this`) to the `Server` object to enable the `Server` to (see) access the `public` members of the `Client` object. Now, we have a bidirectional association (visibility).



Example:

- We have a class called `GraphicTools` that contains tools `Point` objects can use.
- The method `distanceFromOrigin` of `GraphicTools` calculates the distance of a `Point` object from the origin (`0,0`).
- We assume that the `Point` class does not have the ability to calculate distances.
- The `Point` class may contain a pointer to the object of `GraphicTools` (visibility from `Point` to `GraphicTools`).
- The `distanceFromOrigin` method of `GraphicTools` can get the reference to a `Point` object for which the distance is calculated (visibility from `GraphicTools` to `Point`).
- Now, both of the objects can see each other.

Example: Parameter visibility (cont'd)




```
class Point {
public:
  // Constructor receives the address of the GraphicTools object for visibility
  Point(int, int, GraphicTools*);
private:
  GraphicTools * m_toolPtr;     // Visibility to GrpahicTools
};
:
double Point::distanceFromOrigin() const {
  return m_toolPtr->distanceFromOrigin(*this); // sending this for visibility
}
```

- The methods of `Point` can access the methods of `GraphicTools`.
- Since the `distanceFromOrigin()` method of `Point` sends the `this` pointer, the method of `GraphicTools` can also access methods of the `Point` class (bidirectional association).

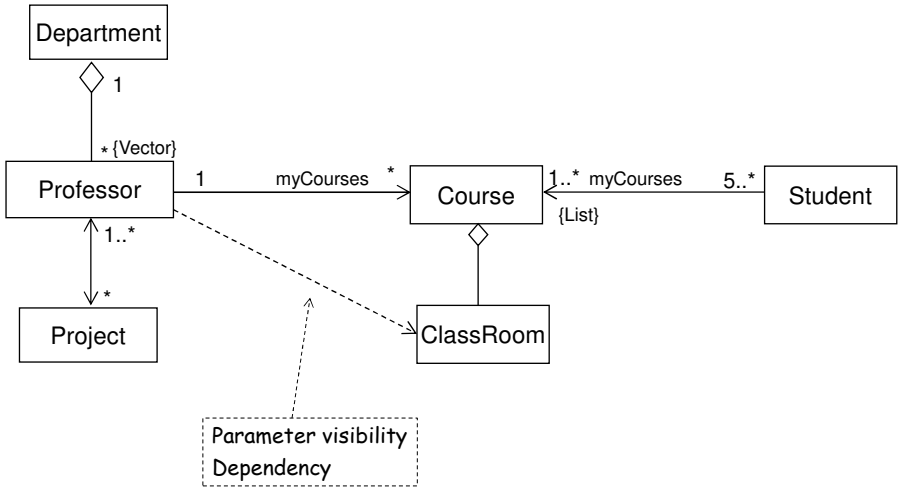
Parameter visibility

```
double GraphicTools::distanceFromOrigin(const Point& in_point) const {
  double local_x = in_point.getX();        // can call methods of Point
  double local_y = in_point.getY();
  return sqrt(local_x * local_x + local_y * local_y);
}
```

Example e06_5.cpp

Example: Partial class diagram of an exemplary software system for a school.

The diagram presents relations and visibilities between classes.



Smart pointers:

- Industrial software systems generally comprise many collaborating objects linked together using pointers and references.
- All these objects must be **created**, linked together (visibility), and **destroyed** at the end.
- It is challenging to destroy members properly, especially if an object is aggregated by multiple owners in an aggregate association.
- The Standard Library of C++ includes **smart pointers**, which ensure all objects are deleted in a timely manner.
- A **smart pointer** is a wrapper class template that owns a raw pointer and overloads necessary operators, such as * and -> .
- Smart pointers are used like raw (standard) pointers.
- Unlike raw pointers, they can destroy objects automatically when necessary.

***C++* Standard Library smart pointers:**

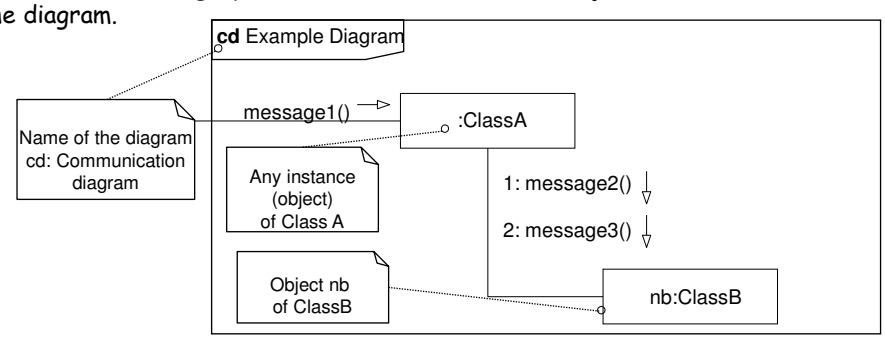
- **`std::unique_ptr<type>`**: It ensures the object is deleted if it is not referenced anymore.
- **`std::shared_ptr<type>`**: It is used when an object has (is shared by) multiple owners. It is a reference-counted smart pointer.
  The raw pointer is not deleted until all `shared_ptr` owners have gone out of scope or given up ownership.
  We will cover smart pointers in detail in Chapter 10.

UML Interaction Diagrams

- Interaction diagrams illustrate how **objects** interact via messages in **runtime**.
- There are two common types: **communication** and **sequence** interaction diagrams.
- Both can express similar interactions.
- Sequence diagrams are more notationally rich, but communication diagrams also have their use, especially for wall sketching.

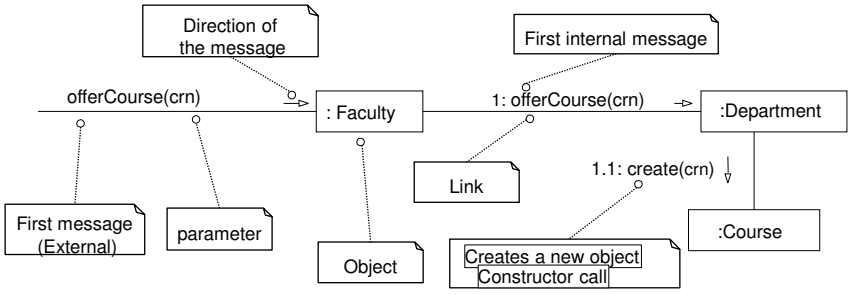
Communication diagrams:

- They illustrate object interactions in a graph or network format, where objects can be placed anywhere on the diagram.

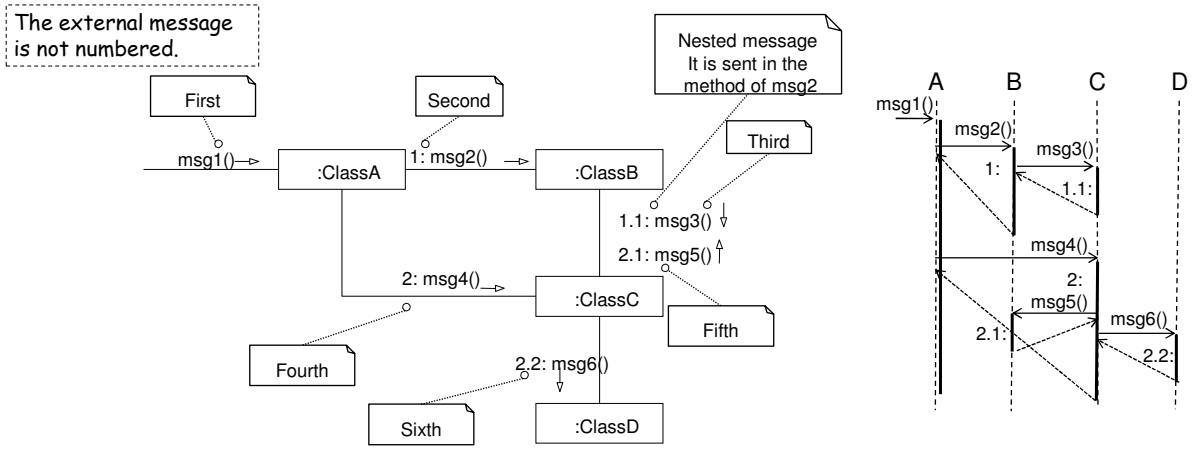


Example:

A communication diagram that presents a message flow about offering a new course at the beginning of a semester:

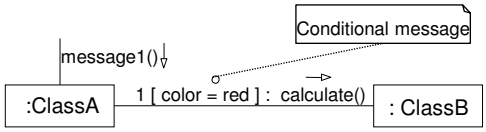


Sequence numbers of messages:

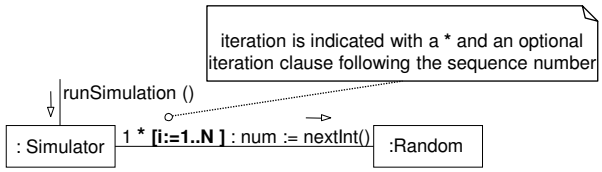


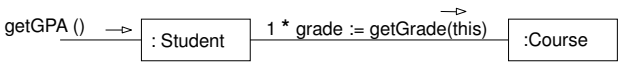
Conditional Messages:

The message is sent only if the clause evaluates to *true*.



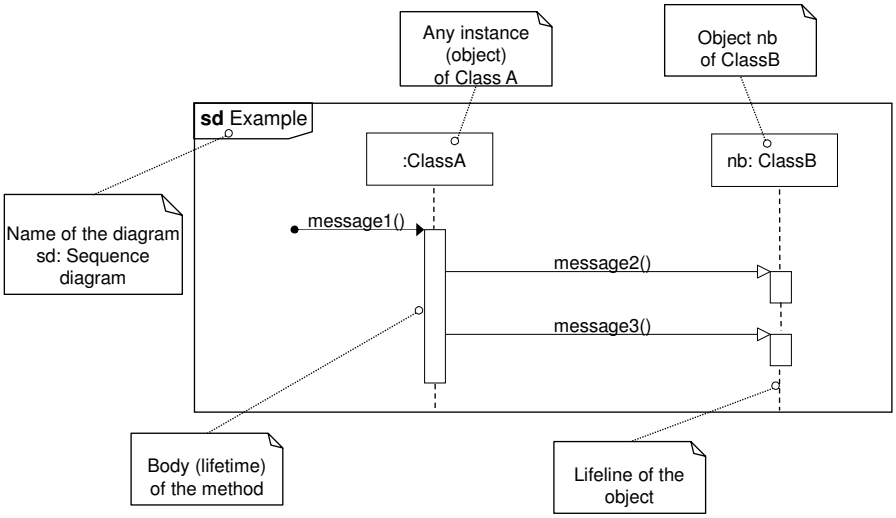
Iteration or Looping:



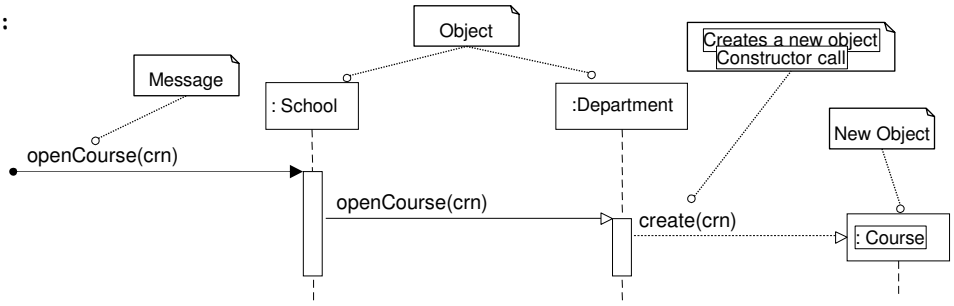


Sequence diagrams:

- Sequence diagrams also illustrate the interactions between objects.
- They clearly show the sequence or time ordering of messages.



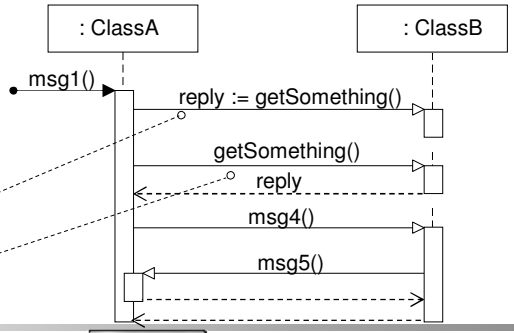
Example:



Illustrating Reply or Returns:

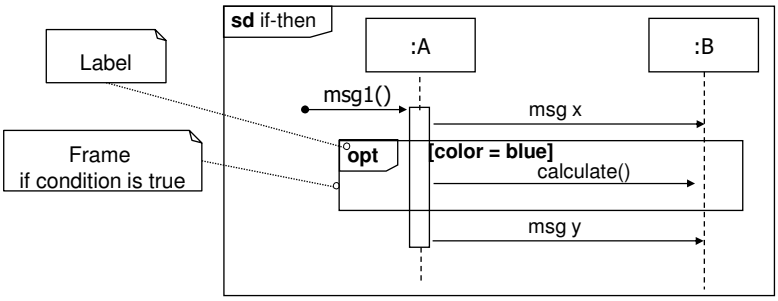
- A sequence diagram may optionally show the return from a message as a dashed open-arrowed line at the end of an activation box.
- There are two ways to show the return result from a message:

  1. Using the message syntax:
  `returnVar := message(parameter)`
  2. Using a reply (return) message line.



Conditional Messages:

- To support conditional and looping constructs, UML uses frames.
- Frames are regions or fragments of the diagrams; they have an operator or label (such as loop or opt) and a guard (conditional clause).
- To illustrate conditional messages, an **opt** frame is placed around one or more messages.



Looping: