CSCI851 Spring-2021
Advanced Programming **(S4b)**

Programming with Class II:
Constructors, Destructors, …

Outline

- Classifying member functions …
- Constructors.
- Destructors.
- Overloading.
- The rule(s) of (some number) …
- Copying and Moving.
- Copying.
- Default and delete.

Classifying Member Functions

- **Inspector functions (access functions)**:
  - Return information about an object's state, or display some or all of an object's attributes.
  - `getName()` and `displayValues()`.
    - A subcategory are **predicate (logic) functions**, such as `isDigit()`, `fail()` - to test various conditions.
- **Mutator functions (implementors)**:
  - Functions that change an object's attributes.
  - `setData()` and `processValue()`.

- **Auxiliary functions (facilitators)**:
  - Functions to perform actions or services.
  - `sortAscending()` or `findLowestValue()`

- **Object Management functions (manager functions):**
  - Constructors - Create objects.
  - Destructors - Destroy objects.

Object Management: Constructors and Destructors

- A **Constructor** function is called automatically each time an object is created.
  - Constructors are used to initialize the object in a specified way.
  - They are functions with the same name as the class, and no return type.
- There are two (basic) types:
  - **Default constructor:** No arguments.
  - **Non-default constructors:** At least one argument.

```
class energyBill {
    private:
        double totalAmount;
        double totalGST;
        int energyUsed;
        string dueDate;
        int refNumber;
        static double rate;
        static const int billerCode;
    public:
        energyBill();
        energyBill(double, double, int, string, int);
        energyBill(const energyBill &);
        static void changeRate(double);
        void showRate() const;
        void display() const;
};
```

- Compiler generated…

```
energyBill :: energyBill() {
    //data members are not initialized
}
```

- … or programmer specified.

```
energyBill :: energyBill() {
    /* Initialize data members */
    totalAmount = 0.0;
    totalGST = 0.0;
    energyUsed = 0;
    dueDate = "" ;
    refNumber = 0;
}
```

- Why need constructors?

```
int year = 2001;                                    // valid initialization
struct thing
{
    char * pn;
    int m;
};
thing amabob = {"wodget", -23};                     // valid initialization
Stock hot = {"Sukie's Autos, Inc.", 200, 50.25};    // NO! compile error
```

```
class Stock  // class declaration
{
private:
    char company[30];
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    void acquire(const char * co, int n, double pr);
    void buy(int num, double price);
    void sell(int num, double price);
    void update(double price);
    void show();
};    // note semicolon at the end
```

- **Default constructor:**
  - The compiler generated, or synthesized, default constructor will default initialize the data members.
  - Usually it's necessary to define your own default constructor since the compiler generates a default constructor if you define no constructors at all, even if the ones you define are all non-default.
  - We can tell a C++11 compliant compiler to generate the synthesized one for us as follows …

```
energyBill() = default;
```

  - It makes sense to use this if we have defined a non-default constructor.
  - This can be in the class definition, or not, the former being inline, the latter not by default.

Destructors

- Destructors are called for an object whenever the object goes out of scope.
- They have the name as the class but with a leading tilde (~).
  - Like ~X() for ADT X.
- One destructor per class.
  - Default set up by the compiler if none is specified.
- No parameters, no return type.

- Think of using the constructor to set up objects and the destructor to remove them nicely.
  - Stroustrup described this as:

    "A constructor establishes the environment for the members to run in; the destructor reverses its actions."
- Allow memory to be released, avoiding memory leaks.
  - Primarily using the `delete` operator on any `new`'d data members.
- They could be used to write some information about the object to a file or to standard out.

Lifetime and RAII

- Constructors and destructors are tied up with managing resources appropriately.
- This is an example of RAII: **R**esource **A**cquisition **I**s **I**nitialisation.
- https://en.cppreference.com/w/cpp/language/lifetime
- Less commonly, but more clearly, called Scope-Bound Resource Management (SBRM).
- More on this later.

Leaking and the order of destruction

```
#include<iostream>
using namespace std;

class House {
private:
   int* area;
public:
   House();
   ~House();
}
```

```
House::House()
{
    area = new int(300);
    cout<<"House up!"<<endl;
    cout<<this<<endl;
}

House::~House()
{
    cout<<"House down!"<<endl;
    cout<<this<<endl;
   // delete area;
}
```

```
int main()
{
   House aHouse[3];
   cout <<"Ending"<<endl;
}
```

```
House up!
ffbff720
House up!
ffbff724
House up!
ffbff728
Ending
House down!
ffbff728
House down!
ffbff724
House down!
ffbff720
```

The last created is the first destroyed!

What if we commented out the destructor?

...

Later → Rule of 5/3/0/6.

Can constructors be private?

- Yes.
- They are *usually* public but we may have a reason for making a constructor private.
  - We may create an object with certain values only under conditions controlled from within an object.
  - A private constructor can also be used in the context of inheritance.
  - We will see later that a non-public constructor is used to implement a design pattern called a singleton.

An aside: Can destructors be private?

- Well…
- Sure, with care. ☺
- The destructor is *usually* public but we can make them private.
- Why would we want to do this?
  - It can give us finality, something we will discuss in the context of inheritance.
  - It can be used to provide safe reference tracking.

- An object may be in use by more than one reference simultaneously.
  - If the destructor is public, a released reference can call it and destroy the object, despite there still being references to the object.
- A reference counting object tracks the number of references to itself.
  - It can only destroy itself when there are no references to it.
- We would have a private destructor and other functions which are careful enough to invoke the destructor only if the number of references becomes 0.

Don’t confuse the compiler.
And don’t get confused by the compilation results …

```
#include <iostream>
using namespace std;

class Example {
  private:
      ~Example(){}
};

int main()
{
  Example ex1;
}
```

Cannot be compiled

The compiler detects that the object `ex1` cannot be destructed.

The `private` is the problem…

```
#include <iostream>
using namespace std;

class Example {
  private:
      ~Example(){}
};

int main()
{
  Example *ex2;
}
```

Can be compiled

`ex2` is not an object, it’s a pointer.

An object has not been constructed yet and therefore there is no need to destruct it.

```
#include <iostream>
using namespace std;

class Example {
  private:
      ~Example(){}
};

int main()
{
  Example *ex2 = new Example;
  delete ex2;
}
```

Cannot be compiled

`delete` cannot get access to the private destructor. This will compile if you comment out the delete, but there will be a memory leak because we aren't tidying up correctly.

Overloading…

- When we first introduced constructors we gave sample code with a few constructors with different arguments…

```
energyBill();
energyBill(double, double, int, string, int);
energyBill(const energyBill &);
```

- This is an example of function overloading.
- Functions are overloaded when they have the same name but different argument lists.
- The parameters given to a function determine which one to use.

Function Overloading

- Function overloading isn't just for constructors…

```
int getMax( int x, int y );
char getMax( char first, char second );
double getMax( double red, double blue );
string getMax( string first, string second );

int mxNum = getMax(19, 4);       // calls int getMax(int, int );
char mxChar = getMax('A', 'V'); // calls char getMax(char, char);
```

- The definitions/logic of the functions are supposed to be different.
  - If they were identical, it may be easier to use function templates instead. Later …

A couple of warnings …

- The overloading is tied to the name and the argument lists have to differ.
- There will be problems if:
  - We provide default values for the function arguments so don't need them when we call …

```
void calculation(int num = 1);
void calculation(char ch = 'A');
```

  - We have two functions with the same name and argument list, but different return types.

```
int getMax( int x, int y );
float getMax( int x, int y );
```

The rule(s) of (some number) …

- C++ ADTs have some special member functions.

Special Member Function	For Class X;
Default constructor	`X();`
Destructor	`~X();`
Copy constructor	`X(const X&);`
Copy assignment	`X& operator=(const X&);`
Move constructor (C++11)	`X(X&&);`
Move assignment (C++11)	`X& operator=(X&&);`

5

3

- We will look at the details of these operations soon.

Rule of three/five/zero/six …

- There are rules, fairly firm guidelines, regarding what we do with the default special member functions.
  - http://en.cppreference.com/w/cpp/language/rule_of_three
  - https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming)
- The numbers are three, five, and zero, because the default constructor is often treated differently but not always …
  - … so the rule of six is mentioned sometimes too: https://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf

The rule of three …

- The rule of three says if you define one of destructor, assignment operator, or copy operator, you should define all three.
  - Why?
  - If you need to define one it's unlikely everything will be consistent with the others by default.
- See http://en.cppreference.com/w/cpp/language/rule_of_three

The rule of five (or seven)

- The rule of five is the C++11 extension of the rule of three to take into account the need to typically define the move constructor and move assignment too.
  - These are sometimes referred to as copy control and, if we don't provide them, the compiler does.
  - Why seven? Include a non-default constructor and the default constructor.
  - If you didn't need moving for whatever reason the rule of three applies.
  - The rule of zero is for the case where you don't need copying or moving.

Copying and moving

- What are they?
- They are to do with what happens when we initialise an object from another of the same type … and a bit more…
- We will leave Move operations until later!

Special Member Function	**For Class X;**
Copy constructor	`X(const X&);`
Copy assignment	`X& operator=(const X&);`
Move constructor (C++11)	`X(X&&);`
Move assignment (C++11)	`X& operator=(X&&);`

- In C++ copying occurs in two forms:

- **Implicit** : The compiler makes a copy of the contents of an object into a new memory location of the same type.

- **Explicit** : The programmer specifies that a particular object's values are to be transferred to the memory of another object, this is copy assignment.

Copy constructor: `X(X const&);`

```
class Point {
   int x;
   int y;
};
```

- This is defined whether we specify other constructors or not, if it's needed by the compiler.
- It's invoked by the compiler if we …
  - Declare an object and use it to initialize another object of the same class.

```
Point pointA(pointB);
Point pointA = pointB;
```

  - Pass an object by value to a function.

```
float getDistance(point pointA);
```

  - Return objects by value from functions.

```
point getPosition();
```

At least roughly . . . some compilers behave a bit differently.

Synthesised copying / copy assignment…

```
class Point {
   int x;
   int y;
};
```

- The default/synthesised copy constructor makes a member-by-member copy of the non-static data members of the objects.
- The copying implied by …

```
Point pointA(pointB);
```

- … will be

```
pointA.x(pointB.x);
pointA.y(pointB.y);
```

So?

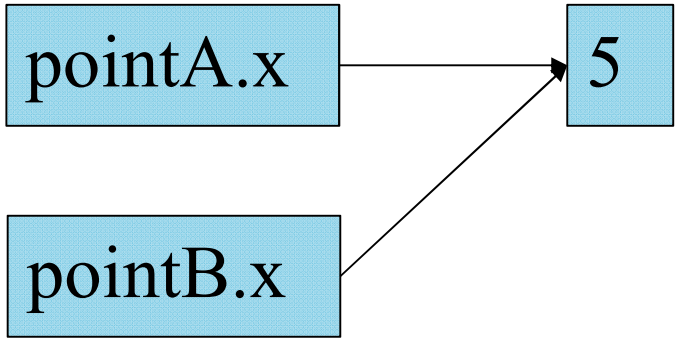
```
class Point {
   int x;
   int y;
};
```

- Well this might be okay, and it is for `Point`, but we run into problems if we have links to content outside the object.
- Effectively an instance of `Point` is stored as two contiguous integers, and we are just copying a chunk of memory to somewhere else.
- But if we have pointers or dynamic memory, this type of copying may not work correctly, it may be too shallow.
  - You may have come across the idea of shallow and deep copying in Java in CSIT121.

- What happens with this more complex class, with a pointer in it?
- It may as well have been a struct.

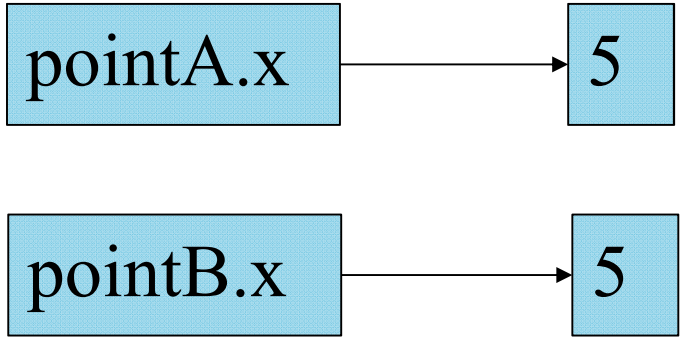
```
#include <iostream>
using namespace std;

class Pointier {
public:
        int *x;
        Pointier(){x = new int(5);}
        void display(){cout << *x << endl;}
};

int main()
{
        Pointier PointA;
        Pointier PointB(PointA);
        PointA.display();
        PointB.display();
        *(PointB.x)=3;
        PointA.display();
        PointB.display();
}
```

Shallow → Deep copying …



- In deep copying we make sure we create new resources to cover the resources outside of the original object.

```
Pointier(const Pointier& point) {
      x=new int(*(point.x));
}
```



- If we have an array we want to make sure we copy all the elements of the array, and assign resources for them, rather than just copying a pointer to a location.
- If we don't carry out this type of deep copy we can be left with a dangling pointer.
  - A dangling pointer is one that refers to memory that no longer holds an object.
  - It's pretty much the same as not having a pointer initialised.

Copy assignment operator:
`X& operator=(const X&);`

- Similar but this is an overloaded operator now, and the copying implied by …

```
point pointA,pointB;…;
pointA = pointB;
```

- … will be

```
pointA.x = pointB.x;
pointA.y = pointB.y;
```

- We have the same problem as before and the solution is the same, define the appropriate copy assignment operator.

- Note the difference between using the copy constructor in

```
Point pointB;
Point pointA = pointB;
```

- … and the copy assignment in …

```
point pointA,pointB;…;
pointA = pointB;
```

`X& operator=(const X&);`

- Assignment operators typically return a reference to their left-hand operand, as this one does.
- This is consistent with the built in types.
  - Note that the default operators are being used in the default/synthesised copy assignment.
- We will get to general overloaded operators soon.
- The textbook notes it's important to treat self-assignment carefully.
  - Make sure that you don't accidentally remove all copies.

The last words here =default and =delete

- Way back on slide 8 we mentioned that we can tell a C++11 compliant compiler to generate the synthesized default constructor for us as follows …

```
energyBill() = default;
```

- A syntactically similar C++11 concept is that of deleted functions…
- … which we can make use of if copying and copy-assigning don't make sense.

- Deleted functions are declared but otherwise cannot be used, but they block a synthesised version being generated by the compiler.
- In the example below, from the textbook, we explicitly state that we want to use the default constructor and destructor, and that we don't want to allowing copy or copy assignments to be made.

```
Struct NoCopy {
      NoCopy() = default;
      NoCopy(const NoCopy&) = delete;
      NoCopy &operator=(const NoCopy&) = delete;
      ~NoCopy() = default;
      // other members
};
```

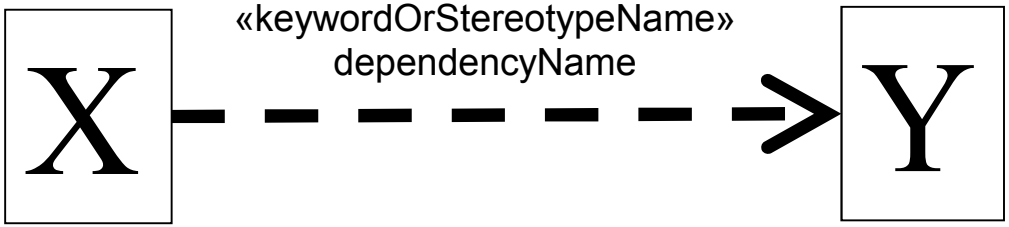
Class/object relations

- Class symbols by themselves provides quite limited information about the system.
- A system will often have multiple classes related in various ways, and a model of the system should reflect such relationship.
- UML defines the following relationship types:
  - Dependency.
  - Association.
  - Aggregation.
  - Composition.
  - Generalisation.

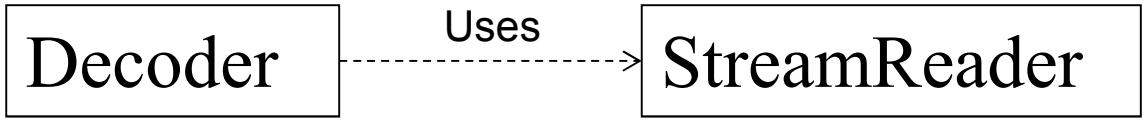
- The nature of the relationship can significantly impact how changes we make to one class, or to objects of that class, can impact on other classes or objects.
- In a certain sense those relations are in order of binding strength.

Dependency.
Association.
Aggregation.
Composition.
Generalisation.

Dependency



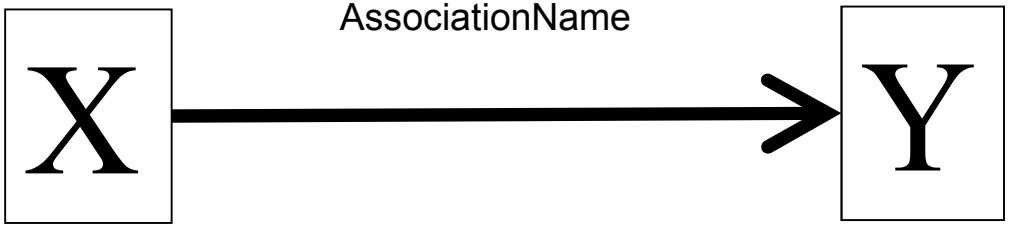
- Very general, one directional relationship indicating that one class uses another class, or depends on it in someway.
  - X uses Y but Y isn't influenced by X.



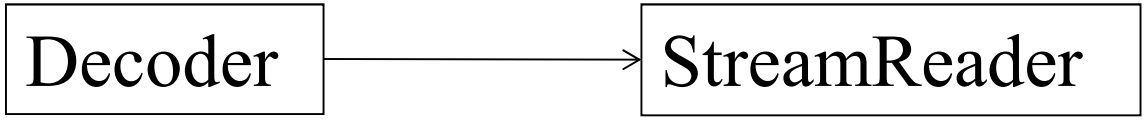

```
class Decoder {
 private:
   .  .  .
 public:
   void readStream( StreamRd *stR )
   { stR->getStream(); … }
   ...
};
```

```
class StreamRd{
   private:
        .  .  .
   public:
      void getStream();
   ...
};
```

Association



- One class retains a relationship with another class.
  - Often a collection of object links.
  - The illustration is one directional but these are often bi-directional and then a line without arrows is used.
- Association can be described as a “has a” type of relationship.



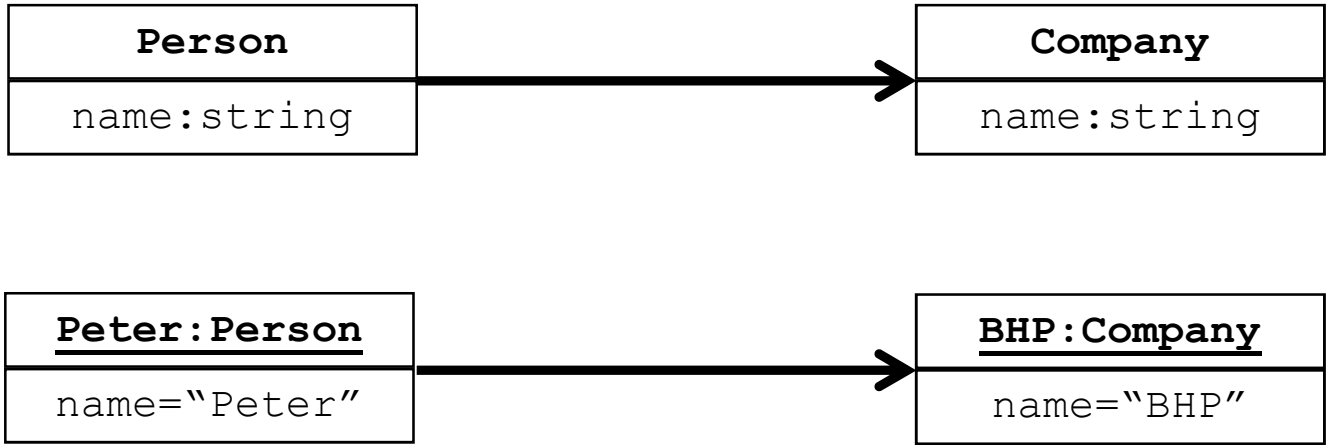

```
class Decoder {
 private:
   StreamRd *stR;
 public:
   Decoder();
   Decoder( StreamRd *stream )
   { stR = stream; … }
   ...
};
```

```
class StreamRd{
   .   .   .
};
```

- In the implementation we have an object of one type being in another class
  - as opposed to the object being based into a function of the class as seen in the dependency.
- Association, aggregation, and composition all typically have this property
  - the distinction between them is based around the type of connection.

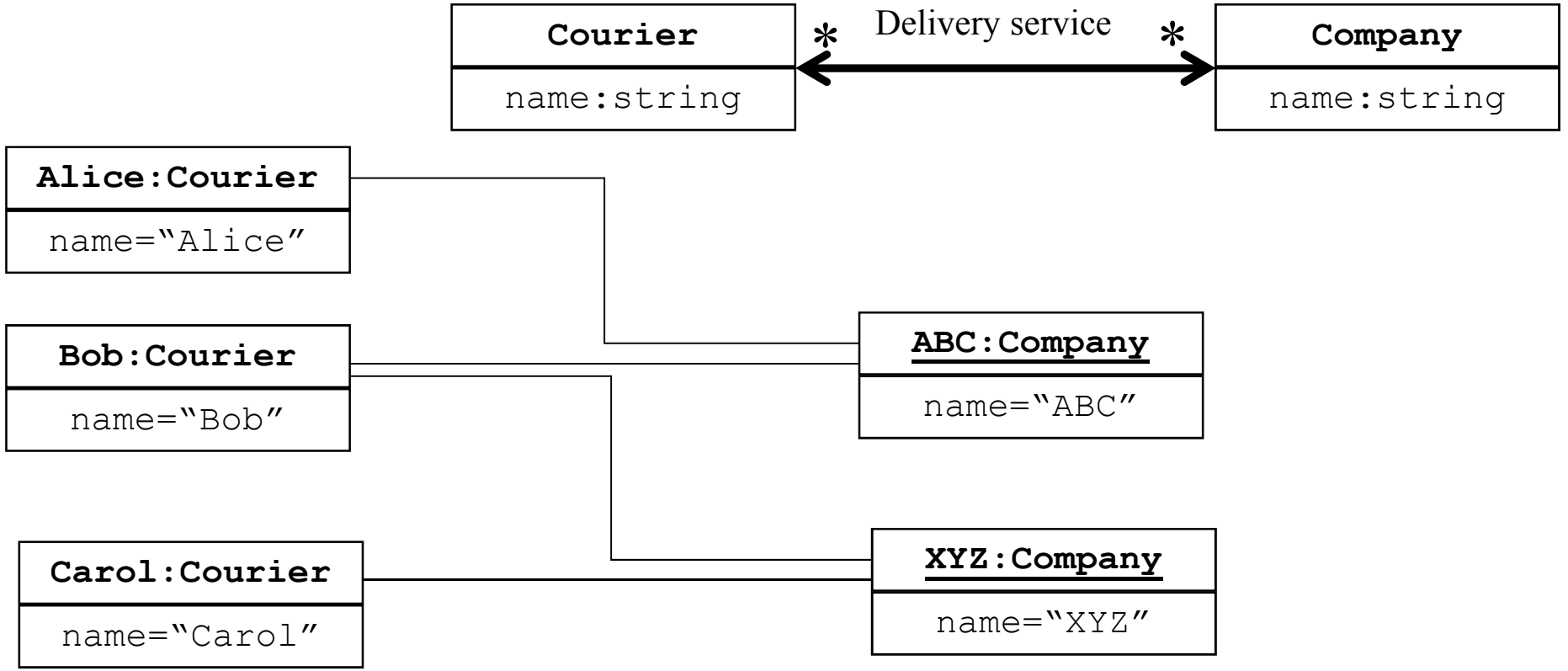
Object links and Class associations

- An association in a class diagram is represented by links between objects in an object diagram, showing that the objects are related/communicate in each other.
  - You can link objects together providing there is a relationship between their classes.
- The links reflect only a static view of the interaction between objects.

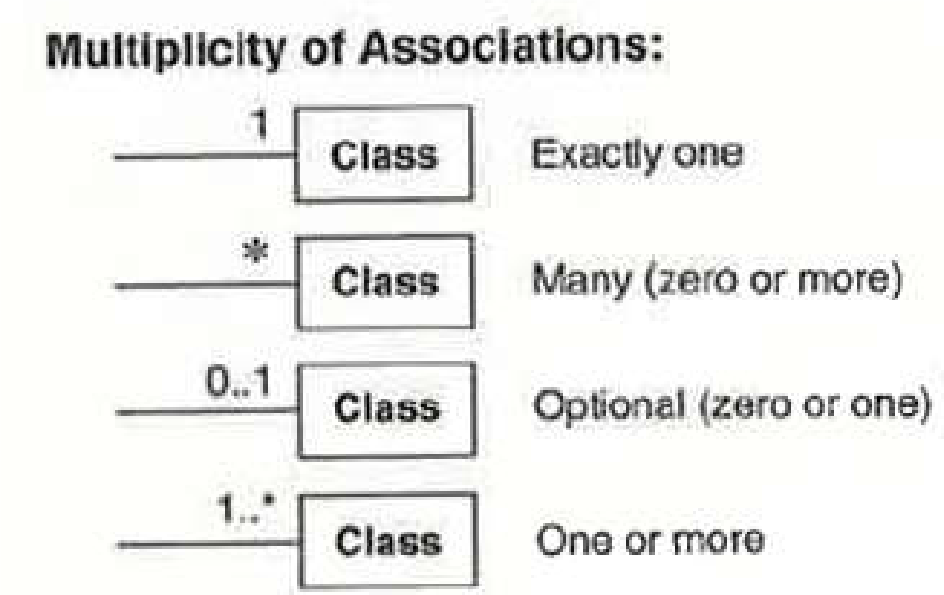


Linking multiple objects

- Sometimes we want to link multiple objects from one class to an object of another.
  - We illustrate this in a UML diagram using the multiplicity, represented by a *.



- From Blaha & Rumbaugh:



- For example:

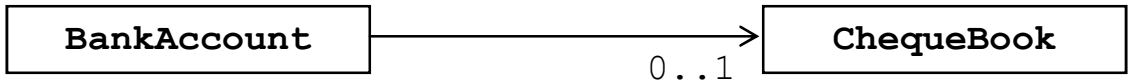
- Actually some countries have more than one capital.
- Countries have many cities, and we can use this to illustrate a range of values allowed for a multiplicity, in general “a..b”.

Some Implementation Examples

- Optional one-way Association:

  Example: A Cheque Book may be issued for use with a Bank Account, but some account holders may not be interested.

  The `ChequeBook` "doesn't exist" independent of the `BankAccount`.

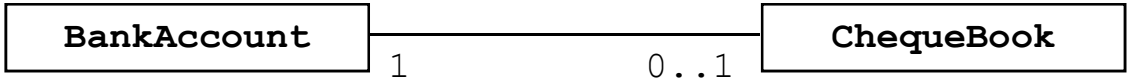



```
class BankAccount {
   private:
        ChequeBook *chBook;
   public:
      BankAccount(CheckBook *cP);
   ...
};
```

```
class ChequeBook {


   ...
};
```

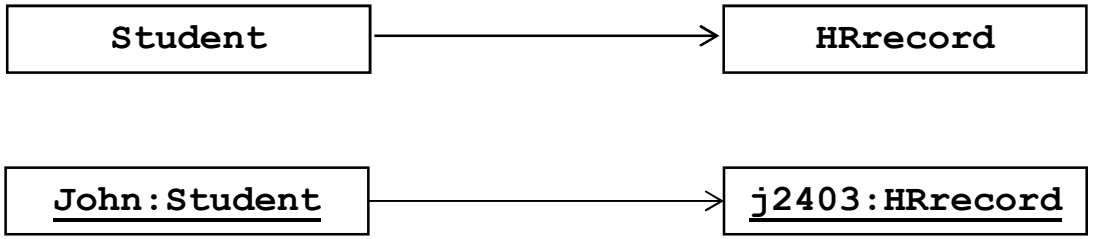
- You need to keep track of whether a `ChequeBook` exists or not, if it doesn't the `chBook` would effectively be NULL.
- We can "make one", effectively initialise `chBook`, only if the previously one has been deleted.

- Bi-directional Association (one-to-optional):
  - We could have a Cheque Book for the Bank Account, as before, but…
  - … the Cheque Book has to be linked to a Bank Account.




```
class BankAccount {
   private:
      ChequeBook *chBook;
   . . .
};

class ChequeBook {
   private:
      BankAccount *account;
   . . .
};
```

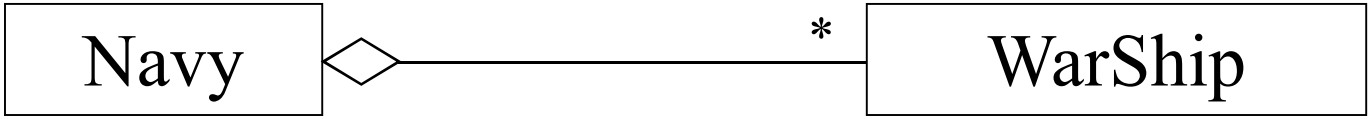
One way link between objects




```
class HRrecord{
  . . .
};

class Student{
  HRrecord *record;
  . . .
};

. . .
HRrecord *j2403 = new HRrecord();
Student *John = new Student();

John->addRecord( j2403 );
```

Aggregation, or shared aggregation

- Aggregation is a stronger Association that reflects "contains" or "owns" type of relationship.
  - The need for ownership means asymmetry.
  - Parts can be in several composites and the destruction of the container/composite doesn't destroy the part.



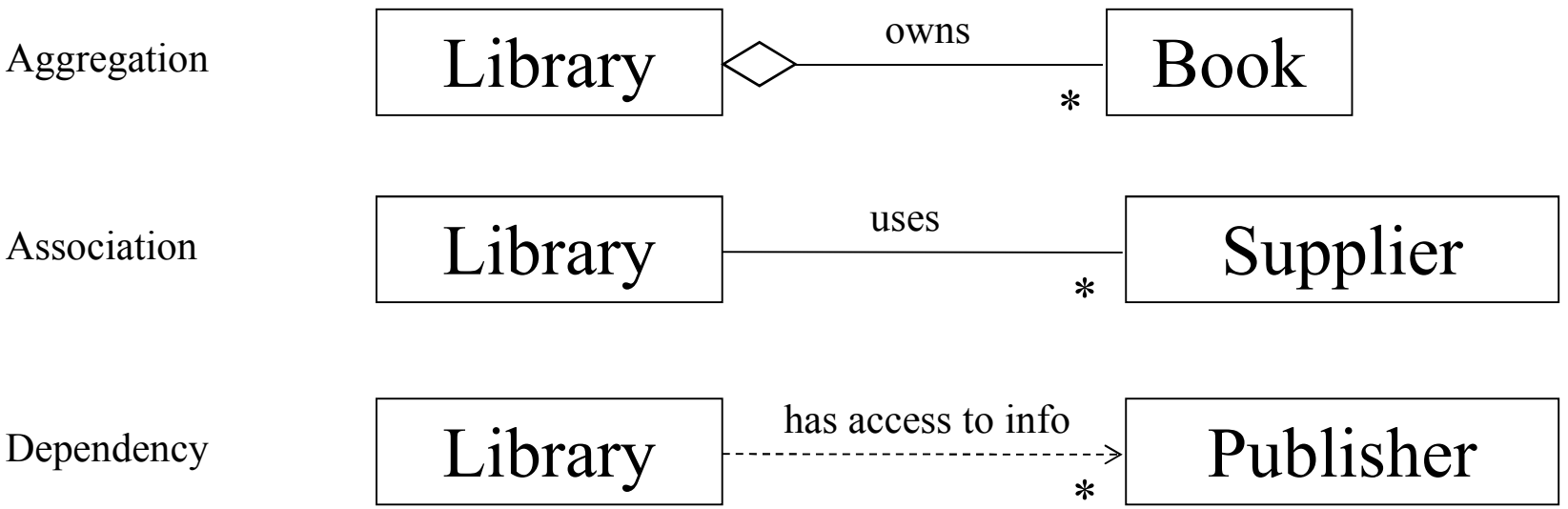

```
class Navy {
 private:
   int currentlyInService;
   WarShip *ships; // array of ships
 public:
   void addnewShip( WarShip *nShip );
   void decommissionShip( WarShip *nShip );
   ...
};
```

```
class WarShip{
   .   .   .
};
```

The implementation of aggregation is similar to association, it's a special case.

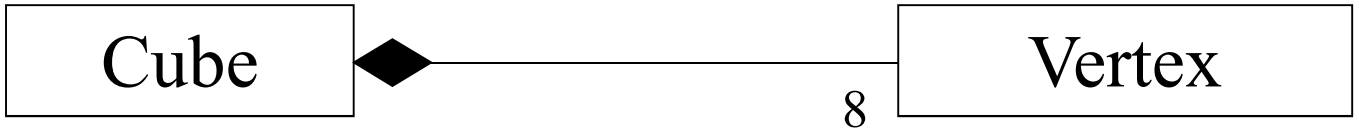
Aggregation vs Association

- As Aggregation is a special case of Association the difference between them is subtle.
  - The major difference is that Aggregation presumes that there is a container that ‘owns’ component objects.
- Aggregation usually presumes multiple relationship.



Composition or composite aggregation

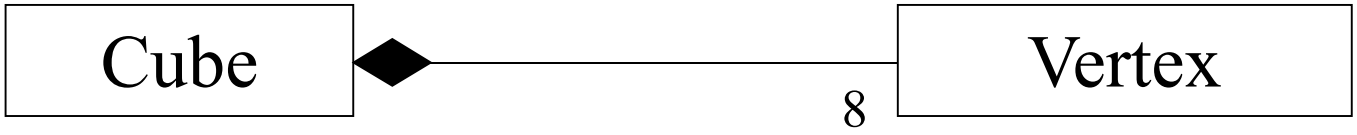
- Composition is the strongest version of Association that reflects a “contains” or “owns” relationship.
  - It extends aggregation by having components as part of the container, with their lifespan typically synchronized with the container.
  - Parts can be included in one instance at a time.




```
class Cube {
 private:
   Vertex points[8];
 public:
   Cube(float coordinates[]);
   ...
};
```

```
class Vertex{
   .   .   .
};
```

- Composition can also be implemented using dynamic memory allocation.




```
class Cube {
 private:
   Vertex *points;
 public:
   Cube() {
      points = new Vertex [8];
   }
   ~Cube() {
      delete [] points;
   }
  ...
};
```

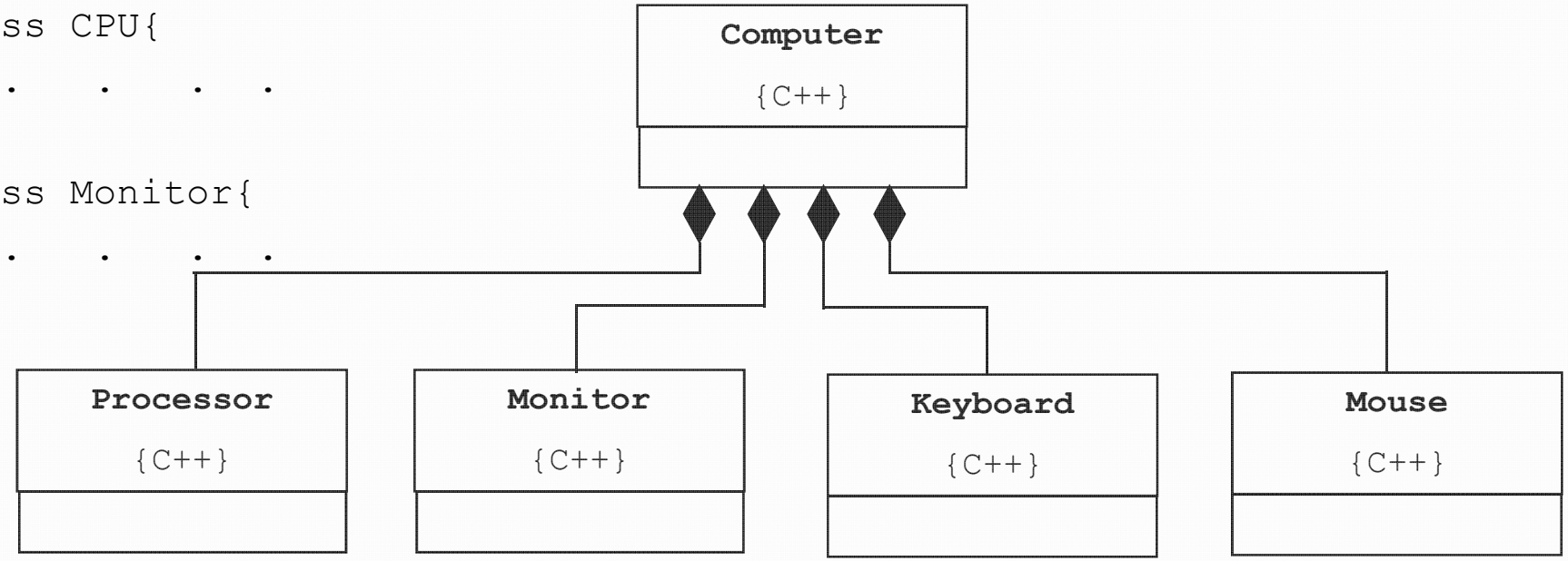
```
class Vertex{
   .   .   .
};
```

Vertex objects are created dynamically by the constructor of class `Cube`.

The objects are deleted by the destructor of class `Cube`.

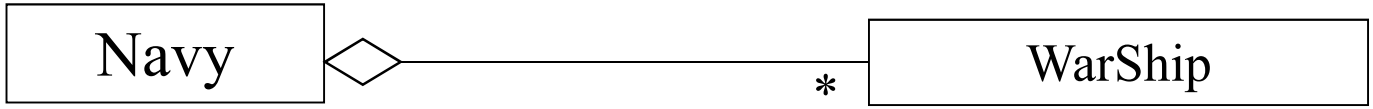
Composition

```
class Computer {
  private:
        Processor cPu;
        Monitor cDisplay
        Keyboard cKb;
        Mouse cMouse;
   .   .   .   .
  public:
        Computer(Processor pr, Monitor mt, Keyboard kb, Mouse ms);
};

class CPU{
   .   .   .   .
};
class Monitor{
   .   .   .   .
};
```

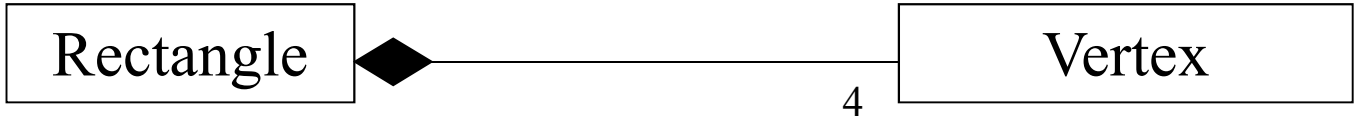


Composition vs Aggregation

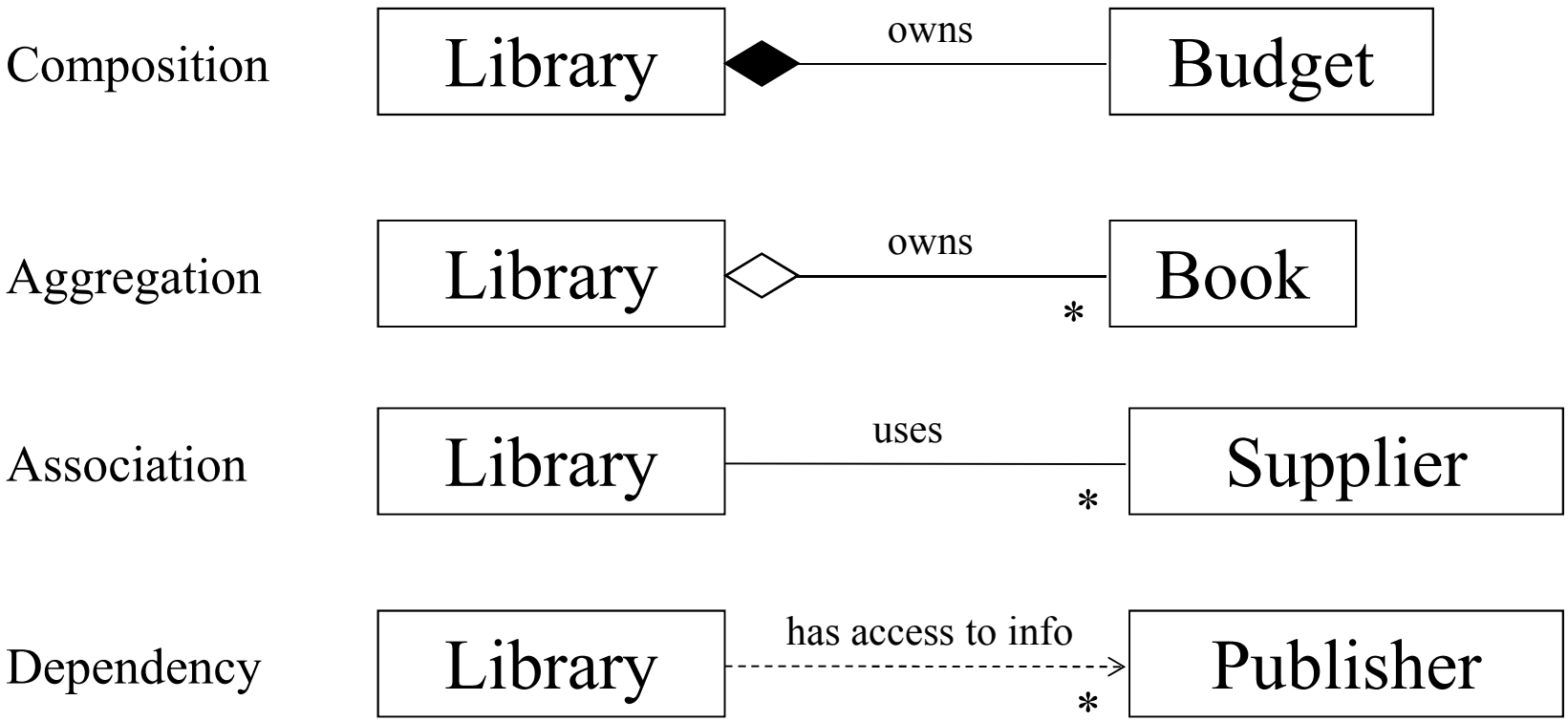
- **Aggregation:**
  - Deletion of component objects does not destroy the container.
    - If some ships are removed or added, it is still a navy.
  - Deletion of the container does not destroy the component objects.
    - The navy can be disbanded and all ships passed to coast guards.



- **Composition:**
  - Component objects must exist during the life cycle of the container as the container needs all of them for its operation.
  - Container deletion may destroy the component objects.



The relations to date …



A question …

- What is the most appropriate relationship between the following classes?

Table

Tabletop

Legs

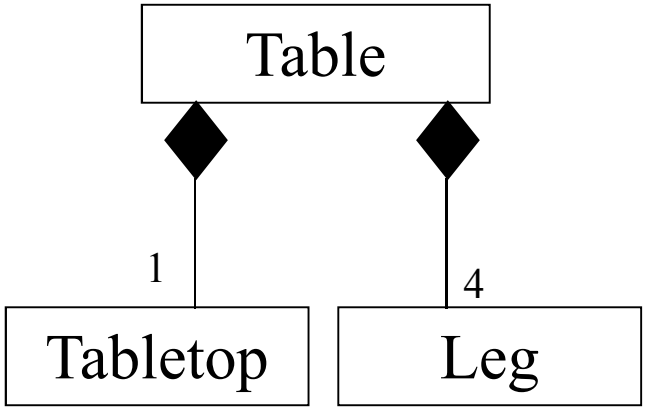
A question …

- What is the most appropriate relationship between the following classes?

Table

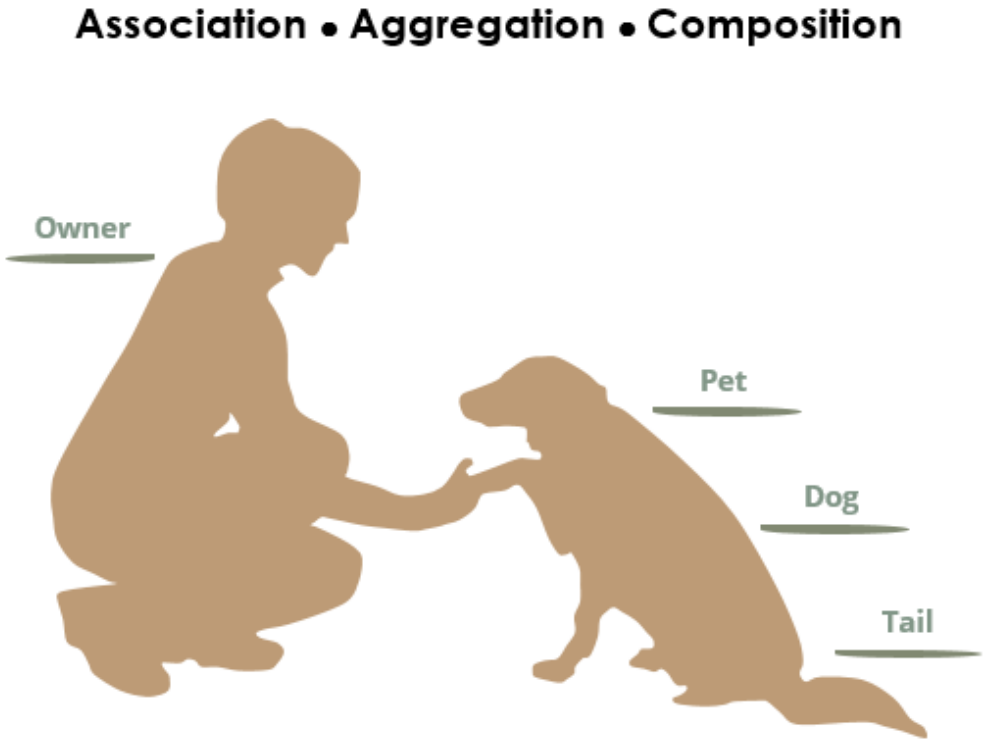
Tabletop

Legs




```
class Table {
 private:
   Tabletop tableTop;
   Leg tableLegs[4];
 public:
   Table(Tabletop tT, Leg tL[]);
   ...
};
```

Another example …



We see the following relationships:

- owners feed pets, pets please owners (**association**)
- a tail is a part of both dogs and cats (**aggregation** / **composition**)
- a cat is a kind of pet (**inheritance** / **generalization**)

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

Constructor initialiser lists

- This could have been mentioned earlier but it becomes more useful when we have aggregation, composition and inheritance.
- You may recognise this code from earlier…
- … we can shortcut a constructor.

```
class energyBill {
    private:
        double totalAmount;
        double totalGST;
        int    energyUsed;
        string dueDate;
        int refNumber;
        static double rate;
        static const int billerCode;
    public:
        energyBill();
        energyBill(double, double, int, string, int);
        energyBill(const energyBill &);
        static void changeRate(double);
        void showRate() const;
        void display() const;
};
```

```
energyBill(double totAm, double totGST,
     int EnUse, string due, int ref) :
    totalAmount(totAm), totalGST(totGST),
    energyUsed(EnUse), dueDate(due),
    refNumber(ref){};
```

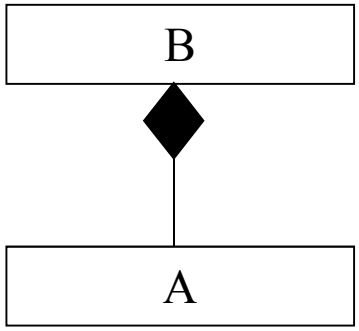
- Here we call the constructors for the individual data members and initialise them with the value from the constructor.
- Why would we this?

- Which equals to:

```
energyBill(double totAm, double totGST,
     int EnUse, string due, int ref){
    totalAmount = totAm;
     totalGST = totGST);
    energyUsed = EnUse;
     dueDate = due;
    refNumber =ref;
}
```

Constructors of the composite class

```
class A {
        int valA;
   public:
        A( int number = 0) { valA = number;}
};

class B {
        int valB;
        A objA;
   public:
        B( int numB, int numA ) { valB = numB;  objA = A(numA); }
};


int main()
{
    B objB( 3, 7 );
}
```



1. **A default constructor `A()` is called to create an object.**

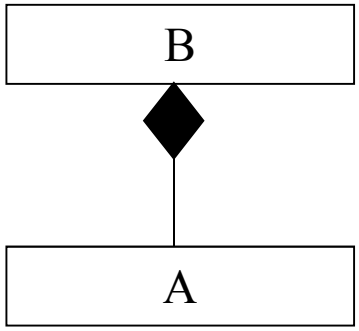
2. **`A(int)` constructor with a parameter is called to initialise data members using the assignment operator.**

3. **`B(int,int)` constructor with parameters is called.**

- We can put output statements in the constructors to see this!

Initialisation List

```
class A {
        int valA;
   public:
        A( int number = 0) { valA = number;}
};

class B {
        int valB;
        A objA;
   public:
        B( int numB, int numA )   :   objA(numA),   valB(numB)  {}
};

int main()
{
    B objB( 3, 7 );
}
```

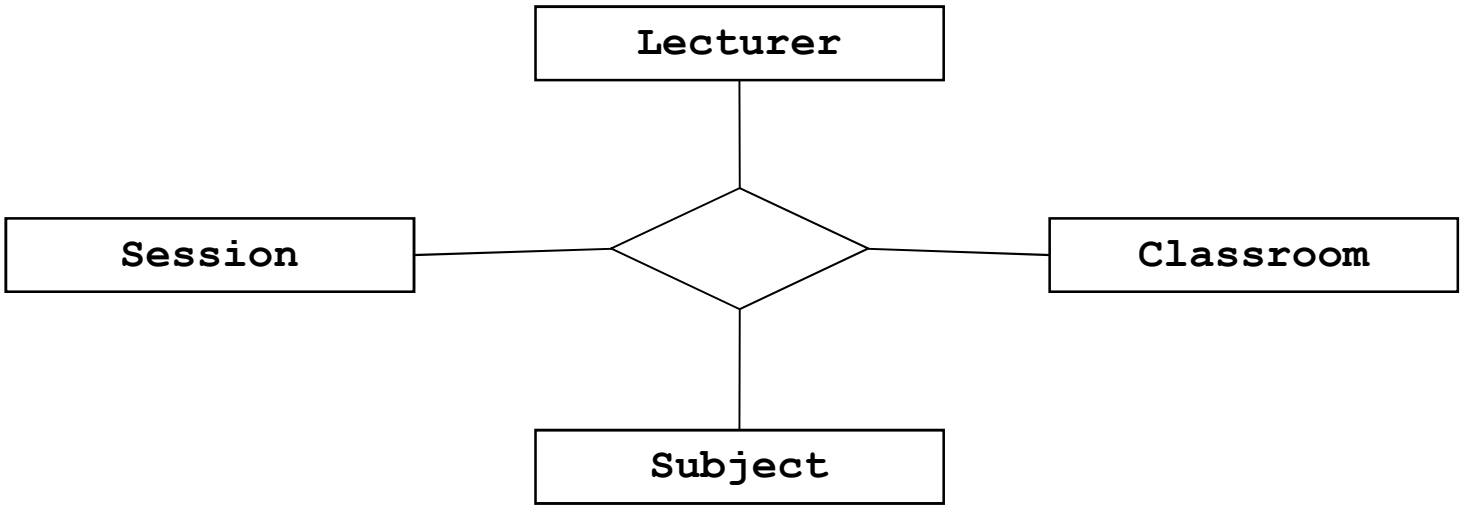


Pass a parameter to the constructor of the class `A`.

The constructor of `A` is called only once.

N-ary Associations

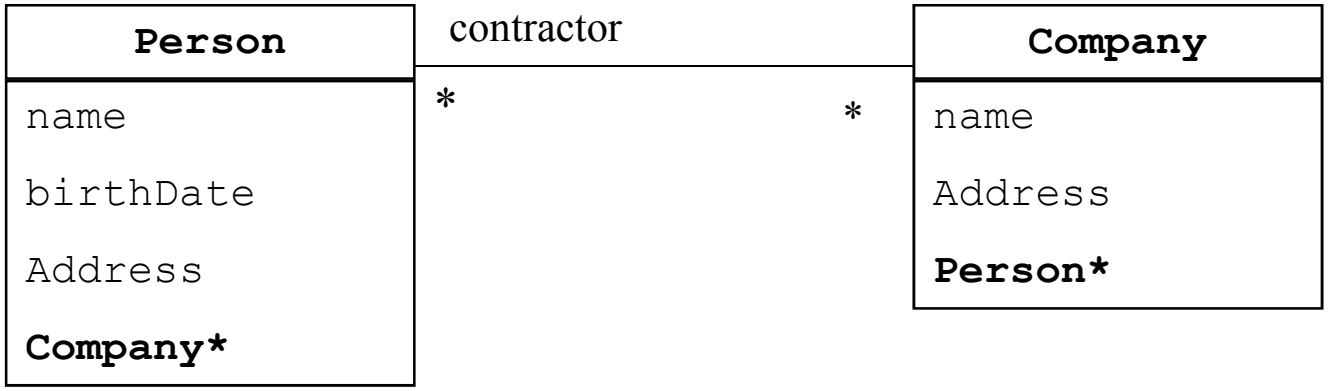
- Associations can be between more than two classes, as in this example…



- Most programming languages cannot appropriately express N-ary associations, so we promote the relationship into an association class.

Association classes

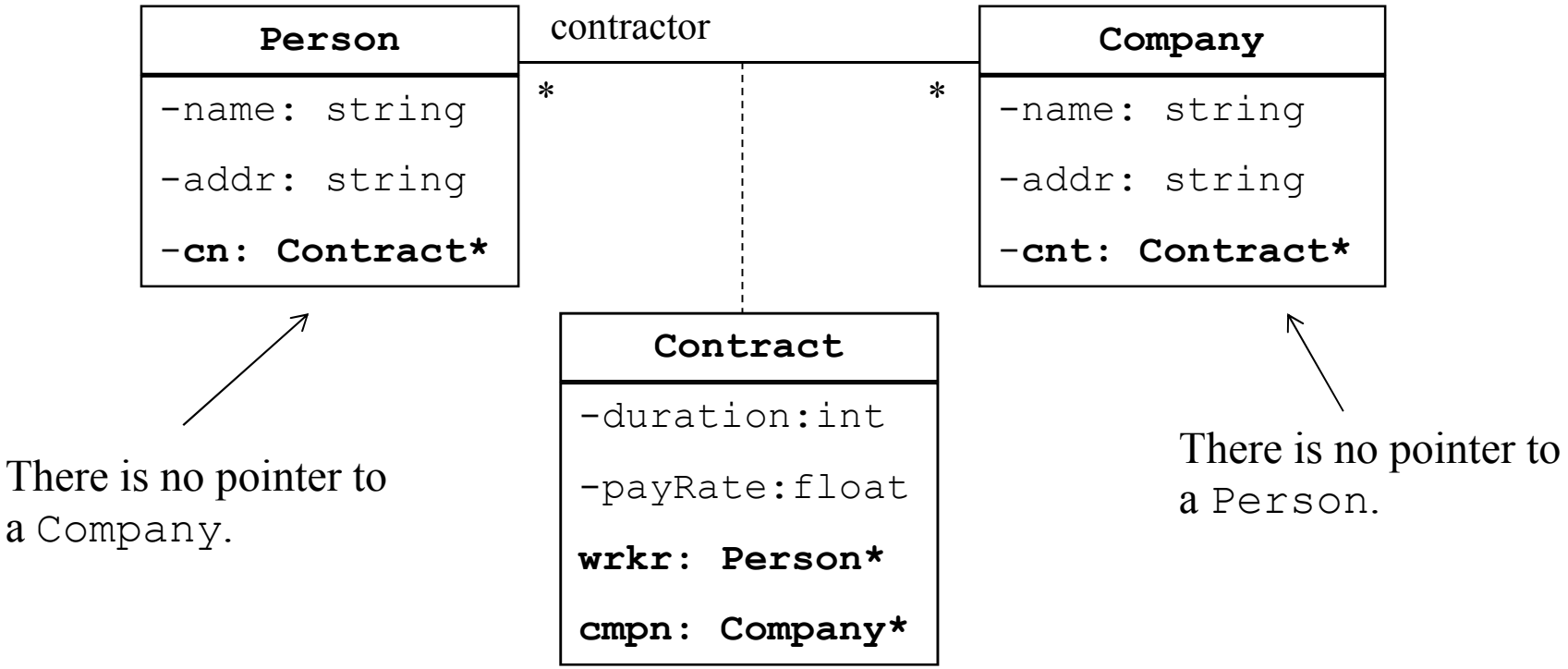
- Simple association can be implemented through pointers to objects as class data members.



- But if the association is implemented through a formal contract, where should the contact attributes be placed?
  - In the class `Person`?
  - In the class `Company`?
- How many contracts can a company have?
- Technically possible, but likely to be confusing and inefficient.

Association classes

- Complex associations can be modelled through classes.
- The notation for an association class is a class attached to an association by a dashed line.



```
#include <iostream>
#include <string>
using namespace std;

class Person;
class Contract;
class Company
{
     string name;
     string addresss;
     Contract *contr; //association via Contract class
  public:
     Company(string Name="") : name(Name), contr(NULL) {}
     string getName() const { return(name); }
     void setContract(Contract *cn) { contr = cn;}
     Contract *getContract() const { return( contr); }
};
```

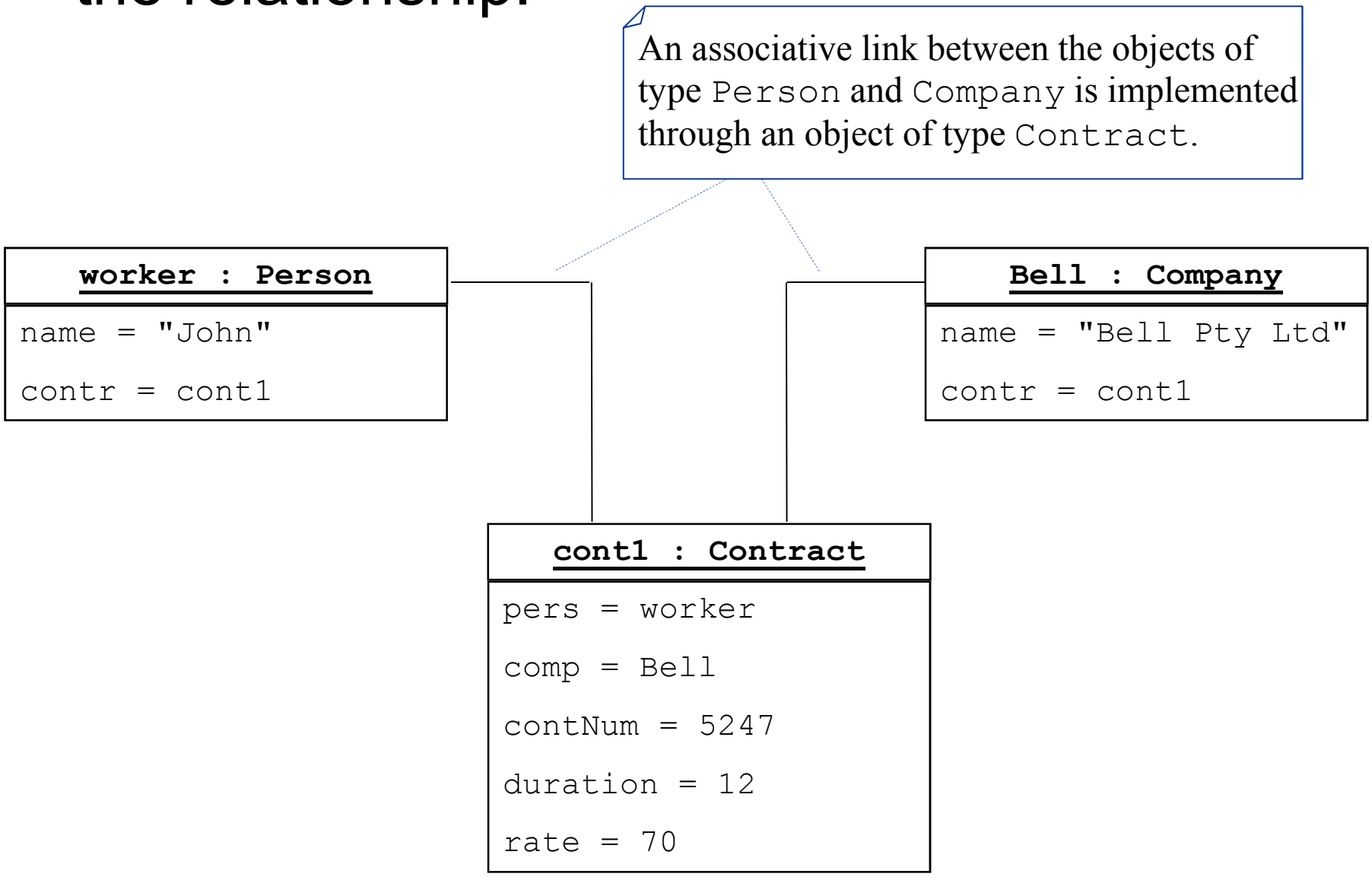
```
class Person
{
    string name;
    Contract *contr; //association via Contract class
  public:
    Person(string Name="") : name(Name), contr(NULL) {}
    string getName() const  { return(name);}
    void setContract(Contract *cn) { contr = cn;}
    Contract *getContract() const { return( contr); }
};
```

```
class Contract
{
    Person *pers;  // association link to the class Person
    Company *comp; // association link to the class Company
    int contNum;
    int duration;
    static float rate;
  public:
    Contract( Person* worker, Company* empl, int cN, int dr )
    : pers(worker), comp(empl), contNum(cN), duration(dr)
    {
       worker->setContract( this ); // set a link Person->Contract
       comp->setContract( this );   // set a link Company->Contract
    }

    string getPersonName() const { return(pers->getName());}
    string getCompName() const { return(comp->getName()); }
    float getRate() const { return(rate); }
    int getDuration() const { return(duration); }
    int getContractNumber() const { return(contNum); }
};
float Contract::rate = 70.00;
```

```
int main()
{
  Person *worker = new Person( "John" );
  Company *Bell = new Company( "Bell Pty Ltd" );

  Contract *cont1 = new Contract(worker,Bell,5247,12 );

  cout <<worker->getName()<<" has a contract number "
  << worker->getContract()->getContractNumber()
  << " with " <<worker->getContract()->getCompName()
  << endl;

  cout<<"Duration: "
  << worker->getContract()->getDuration()
  << " months" << endl;
  cout << "Rate: $" << worker->getContract()->getRate()
  << "/hr " <<endl;

  return 0;
}
```

```
John has a contract number 5247 with Bell Pty Ltd
 Duration: 12 months
 Rate: 70 $/hr
```

- Here goes an object diagram representing the relationship.



The `sizeof` operator again…

- We have seen this for built in types already.
- With combining objects together in new ways it's useful to see the impact of the size of the new composite entities.
- This will be explored in the lab.