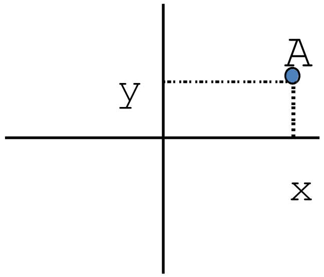
C++: Object Oriented Design

Using structures (Example)

1. Define a structure.
2. Provide a set of functions, which can manipulate the structure.

Example: 2D geometrical transformations



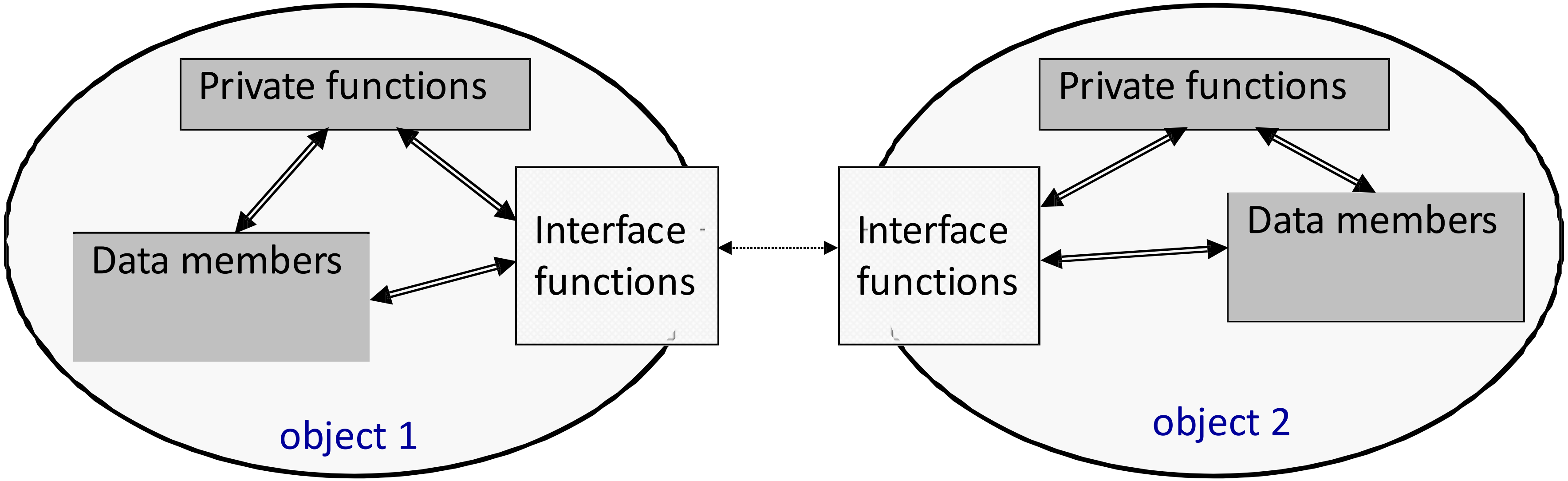
```
struct coordinate // type definition
{
  float x;
  float y;
};
// function prototypes
coordinate shiftHrz( coordinate point, float shift );
coordinate shiftVert( coordinate point, float shift );
coordinate rotate( coordinate point, float angle );
```

Using Structures (Example)

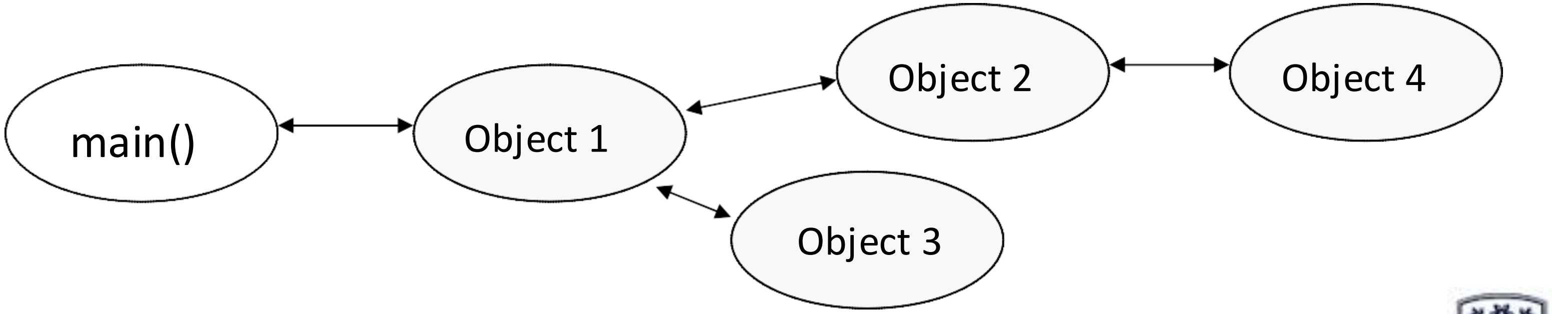
```
//---function definitions---
coordinate shiftHrz( coordinate point, float shift
)

{
  coordinates newPoint;
  newPoint.x = point.x + shift;
  newPoint.y = point.y;
  return newPoint;
}
coordinate rotate( coordinate point, float angle )

{
  coordinate newPoint;
  newPoint.x =  cos(angle)*point.x + sin(angle)*point.y;
  newPoint.y = -sin(angle)*point.x + cos(angle)*point.y;
  return newPoint;
}
```

Object-Oriented Methodology

- OOM is a methodology of developing software systems in which data structures are exclusively operated by a built-in set of functions and can be accessed <u>only</u> through interface functions.

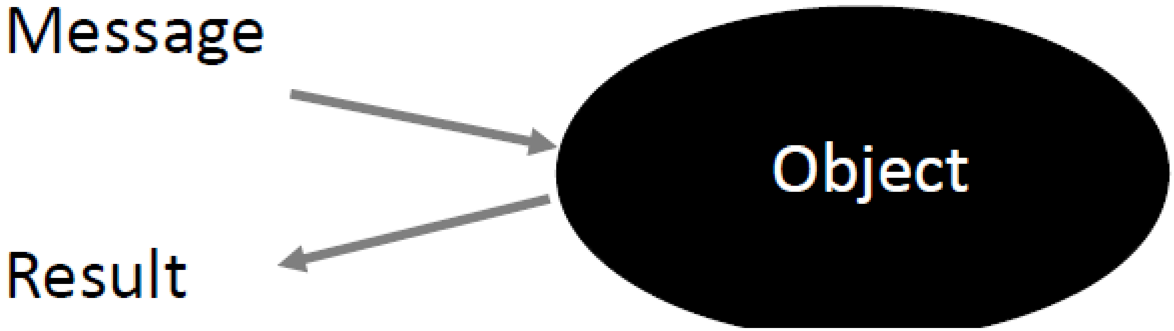


- A system is designed as a collection of interactive objects.



Objects

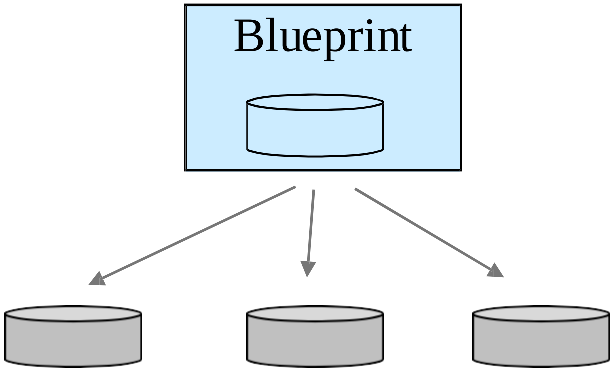
- OOM introduces the concept of objects.
- Like functions and variables in the function oriented methodology, objects are major software building blocks according to the Object Oriented Methodology.
- Rather than thinking about the functions that deal with variables we think about objects and what they can do.
- When you pass a message to an object, it responses by producing a result. You don't need to know an internal structure of an object. It is hidden.

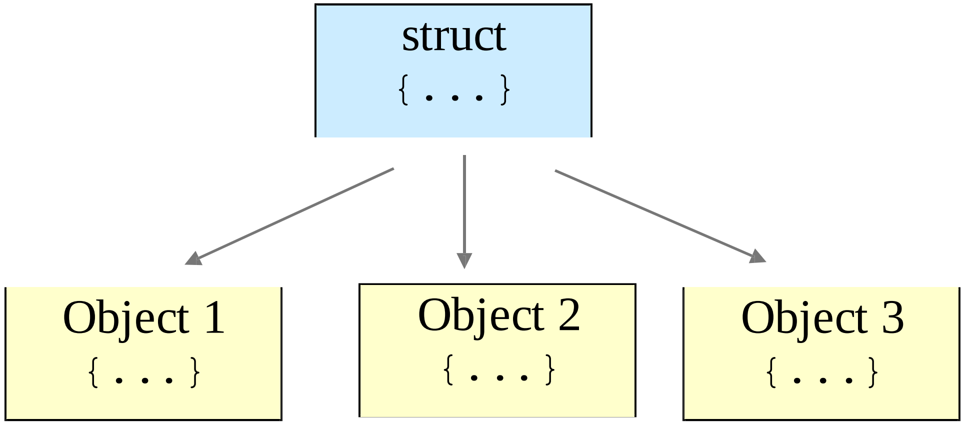


Objects and Structures

Parts of a mechanical system are produced based upon their descriptions – blueprints

Objects of a software system are generated based upon their descriptions – structures





- A `struct` is an abstraction - a user defined type.
- An `object` is an instance generated and placed in computer memory (many similar objects can be generated from one struct).

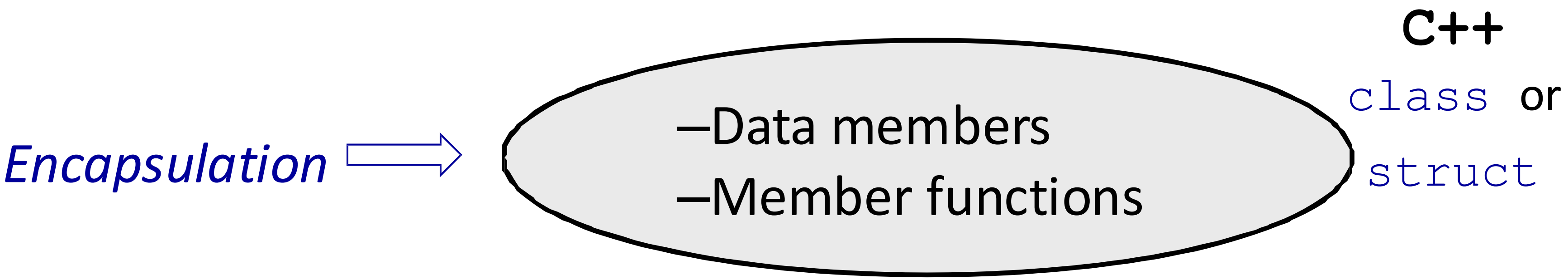
Classes and Structures

- C struct contains only data members.

  All functions needed to manipulate the struct can only be defined externally.

- C++ struct combines into a single unit:
  - data members
  - member functions to manipulate the struct.

- C++ provides another structured data type - class that also combines data members and member functions into a single unit.



Encapsulation

–Data members

–Member functions

C++

`class` or

`struct`

C++Structures and Classes

- What is the difference between `struct` and `class`?

```
struct time
{
   int minutes;
   int seconds;

   void setTime(int mn, int sc);
   void reset();
};
```

All data members and member functions of `struct` have **public** access by default

```
class time
{
   int minutes;
   int seconds;

   void setTime(int mn, int sc);
   void reset();
};
```

All data members and member functions of `class` have **private** access by default

What does this mean?

Structures and Classes

- C++ introduces the concept of data hiding to define access control of the data members and member functions:

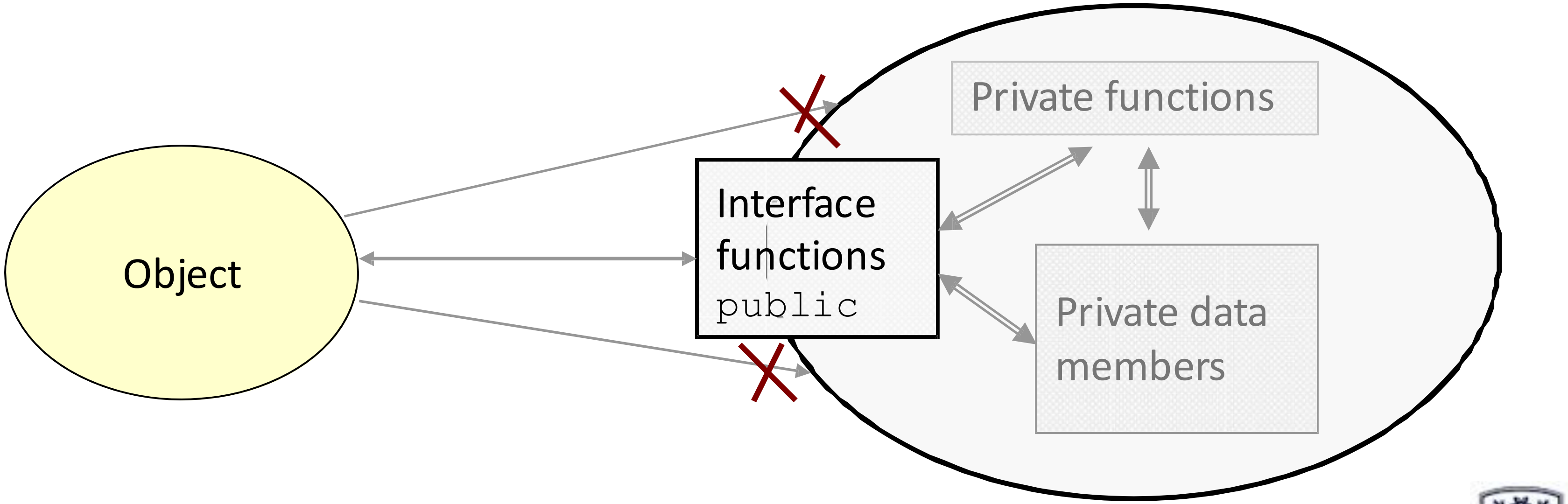
Definitions of **`struct`** and **`class`** have sections marked as:

`private`
`public`
`protected`

} Access modifiers

Objects and Classes

- Private members in a class or **struct** can be accessed only by its own member functions. Private members are hidden from external objects.
- Public members in a **class** or **struct** can be accessed by other objects.
- The data stored in the **private section** are referenced as the **internal state** of the object.
- **Public members** constitute the **interface** components.



Encapsulation of Class Members

- C++ **`class`** provides a mechanism for data hiding that is called **encapsulation** (the same term that is used for combining data and functions)

```
class Example
{
   private:
      int data1;
      float data2;
      char data3;
      float func4(char p1);
   public:
      void func1(void);
      double func2(double pr1, int pr2);
      float func3(void);
};
```

Private data members **`data1`**, **`data2`**, and **`data3`** are only accessible through the public function members **`func1()`**, **`func2()`**, **`func3()`**

`func(4)` is a private function that also can access private data members, but can't be called from outside

Objects and Classes

name of a class *name of an object*

```
Example myExample;
```

Declares an object `myExample` of type `example`

- Accessing public class members (with . operator)

name of the object *public member function*

```
myExample.func1();                        //  legal
myExample.func2(67.8, 45 );               //  legal
float dataEx = myExample.func3();         // legal
int anInt = myExample.data1;              // illegal
float flt = myExample.func4('A');         // illegal
```

Objects and Classes

- An object can be assigned to another object of the same class

```
Example myExample1;
Example myExample2;
myExample2 = myExample1;
```

Data members of `myExample1` will be copied into the corresponding members of `myExample2`.

- Objects can be passed (by value or by reference) as function parameters and returned as function values

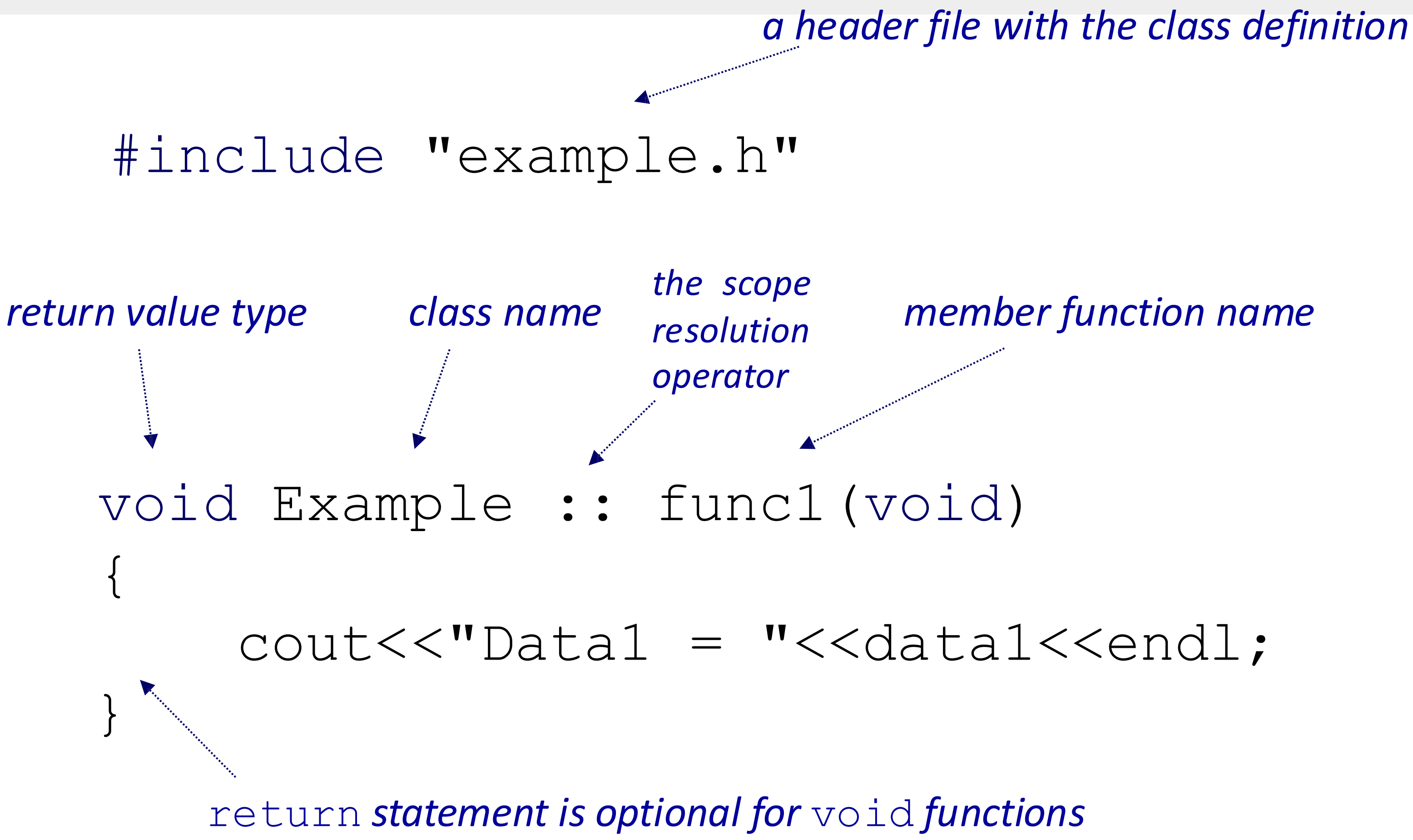
```
Example processData( Example ex1 );
Example& selectObject( Example array[], int size );
```

Objects and Classes

- Definition of class member functions:
  - Function prototypes are included in the class definition.
  - Usually, class definitions are placed in a separate header file, **`example.h`**
  - Definitions of the member functions are usually placed in a separate .cpp file, **`example.cpp`**
  - Class member function have scope within the class

    Use the scope resolution operator :: (double colon) to reference class members

Definition of Member Functions



a header file with the class definition

```
#include "example.h"
```

return value type

class name

the scope resolution operator

member function name

```
void Example :: func1(void)
{
     cout<<"Data1 = "<<data1<<endl;
}
```

`return` ***statement is optional for*** `void` ***functions***

Objects and Classes

- *Example:* We re-write the `struct coordinate` as a class

```
/* -- Header file coordinate.h for the class coordinate --*/
#ifndef COORDINATE_H
#define COORDINATE_H

// class definition
class Coordinate{
   public:
          void shiftHrz(float shift);
          void shiftVert(float shift);
          void rotate(float angle);
          void setCoord(float a, float b);
   private:
          float x;
          float y;
          static const float PI;
};
#endif // COORDINATE_H
```

If you want a constant to have a scope limited to the class, define it together with data members

Objects and Classes

```
/* --- coordinate.cpp --- */

const float Coordinate :: PI = 3.141; // class scope constant

//--member function definitions---

void Coordinate :: shiftHrz(float shift)
{
    x += shift;
}

void Coordinate :: shiftVert(float shift)
{
    y += shift;
}

void Coordinate :: setCoord( float a, float b )
{
    x = a;
    y = b;
}
```

Objects and Classes

```
#include <iostream>
#include "coordinate.h"
using namespace std;

int main(void)
{
  //-- declare objects pointA and pointB---
  Coordinate pointA, pointB;
  float xC, yC;

  cout << "Enter the coordinates of point A (x,y): ";
  cin >> xC >> yC;

  // initialize the pointA
  pointA.setCoord( xC, yC );

  /*--shift A horizontally left by 3.56--*/
  pointA.shiftHrz( -3.56 );
```

Objects and Classes

```
    pointB = pointA; // assign pointA to pointB

    pointB.rotate( 0.32 ); //rotate clockwise 0.32  radians


  //-- declare an object dynamically---
    Coordinate* pointC = new Coordinate;

    pointC->setCoord( xC, yC ); // initialize the pointC

    pointC->shiftVert( 8.5 );

    delete pointC;

    return 0;
  }
```

Object Constructors

- What may happen if we don’t call the method

  ```
  pointA.setCoord( 0, 0 );
  ```

  before calling the method

  ```
  pointA.shiftHrz(-3.56);
  ```

- The initial state of `pointA` is unknown. It can be any random value.

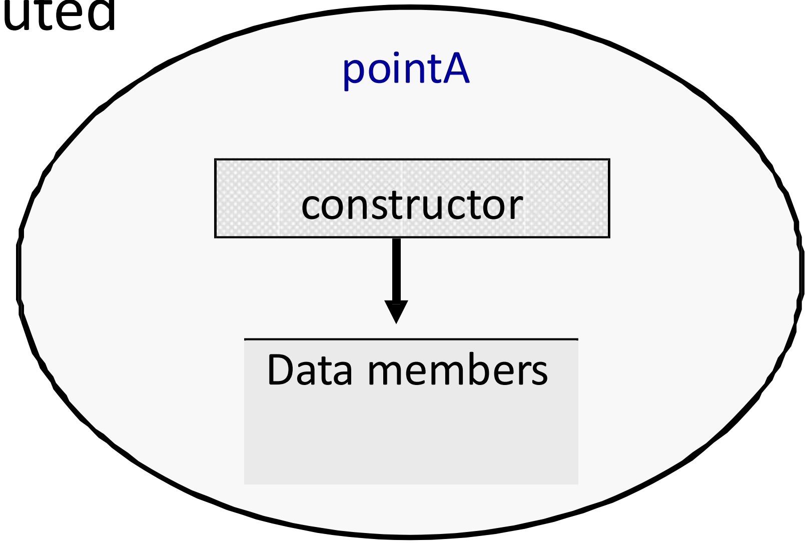
- To guarantee that data members are always initialized, C++ introduces a concept of object constructors.

Constructors

- A constructor is automatically executed when an object is declared.

```
Coordinate pointA;
```

Leads to an automatic call of
pointA.Coordinate();



- Constructors have the following properties:
  - The name of the constructor is the same as the name of the class.
  - A constructor does not return anything
    - Not even **`void`**
  - A class can have more than one constructor; all must have the same name.
  - If a class has more than one constructor, they must have different sets of parameters.

Constructors

- The default constructor is a constructor without parameters. It is executed when you declare the object as:

```
className objectName;
```

  - *Example:*

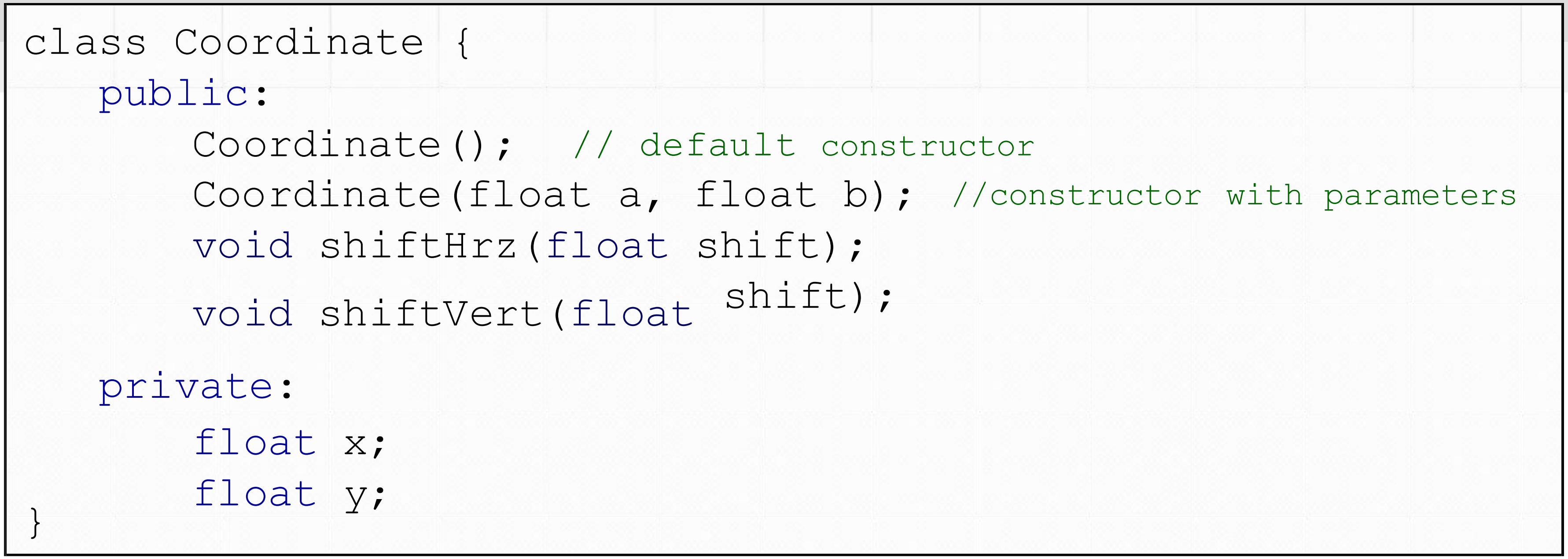
```
Coordinate pointA;
```

- An implementation of the default constructor for the class `Coordinate` can be:

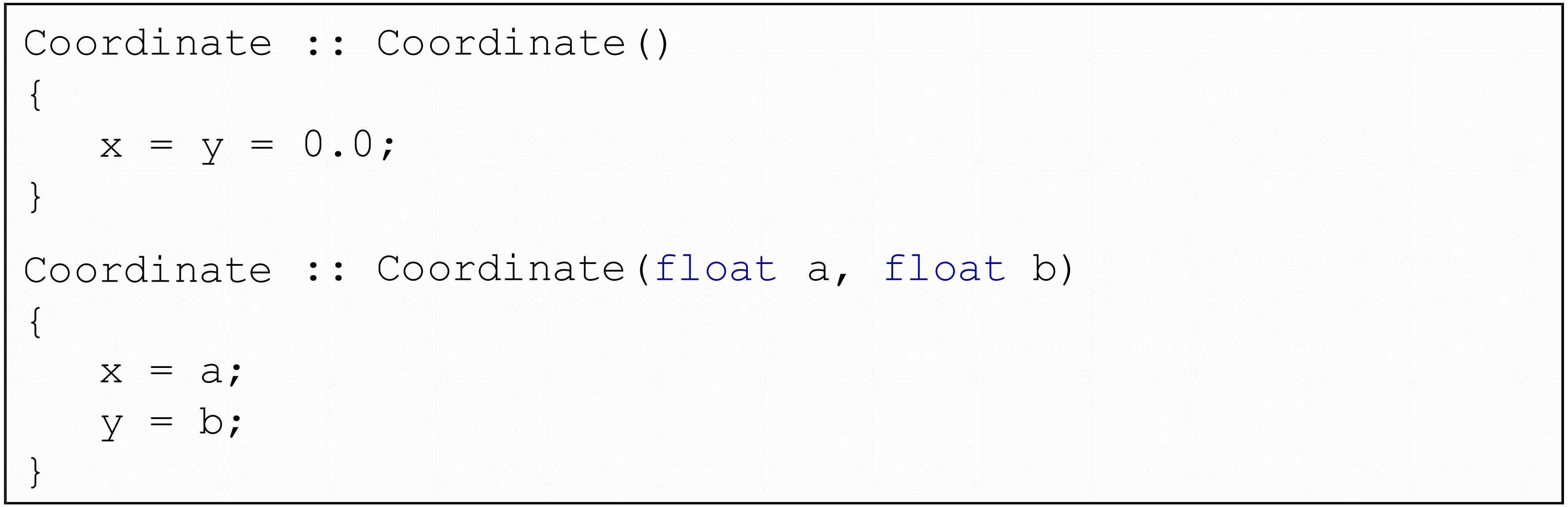
```
Coordinate :: Coordinate()
{
   x = 0.0;
   y = 0.0;
}
```

the same as the class name

Constructors



```
class Coordinate {
   public:
       Coordinate();   // default constructor
       Coordinate(float a, float b); //constructor with parameters
       void shiftHrz(float shift);
       void shiftVert(float shift);

   private:
       float x;
       float y;
}
```



```
Coordinate :: Coordinate()
{
   x = y = 0.0;
}

Coordinate :: Coordinate(float a, float b)
{
   x = a;
   y = b;
}
```

Constructors

- Constructor can have default parameters
  - If all constructor parameters have default values it becomes the default constructor.
  - Only one default constructor can be defined in a class.

```
class coordinate{
  public:
     ~~coordinate()~~; // must be removed
     coordinate( float x = 0.0, float y = 0.0 );
      void shiftHrz(float shift);
      void shiftVert(float shift);
      void rotate(float angle);
      void setCoord(float a, float b);
  private:
      float x;
      float y;
};
```

Constructors

Example: A constructor with default parameters serves as a default constructor

```
class Coordinate {
   public:
       Coordinate( float a=0.0, float b=0.0 ); //constructor
       void shiftHrz(float shift);
       void shiftVert(float shift);

   private:
       float x;
       float y;
}
```

```
Coordinate :: Coordinate( float a, float b )
{
   x = a;
   y = b;
}
```

Constructors

- If there are no user-defined constructors for a class A, the compiler implicitly declares a default constructor A::A()   that does nothing

```
class aClass
{
  public:
      float getYValue();
      float getXvalue();
  private:
            float x;
            float y;
};


.   .   .
aClass obj1;   //invokes obj1.aClass();
               //implicitly declared by compiler
               //the that does nothing
```

Constructors

- If there is a user-defined constructor for a class `A`, the compiler does not implicitly declare a default constructor.

Simple function definitions can be placed inside class definitions

```
class aClass{
  public:
     aClass( float f1, float f2) { x=f1; y=f2; }
     float getYValue();
     float getXvalue();
  private:
          float x;
          float y;
};

.   .   .
aClass ob2( 3.5, 7.8 );  // OK

aClass ob1;
```

Compilation error

`aClass()` default constructor must be defined, or not called

Constructors

There are two ways how constructors can initialise data members

1. Using the assignment operator in the function body

```
class EnergyBill {
    private:
       float totalAmount;
       int energyUsed;
    public:
        EnergyBill();
        EnergyBill(float tA, int eU);
         void showTotalAmount();
         void displayBill();
};
EnergyBill :: EnergyBill(float tA, int eU)
{
   totalAmount = tA;
   energyUsed = eU;
}
```

Constructors

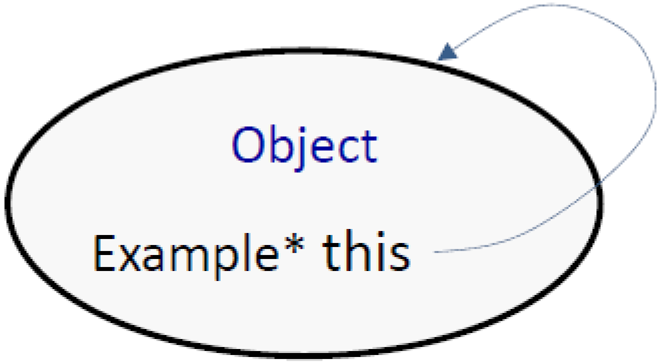
2. Using an initialization list

```
class EnergyBill {
  private:
    float total;
     int  enrUsed;
  public:
     EnergyBill();
     EnergyBill(float tA, int eU) : total(tA), enrUsed(eU) { }
     void showTotalAmount();
     void displayBill();
};
```

this Pointer

- When an object of a class is declared, a memory block is allocated to store this object.
- Address of the allocated memory block is attached to each object. This pointer has a reserved name this.

```
Class Example {
   private:
      int data;
   public:
      Example();
      Example( int data );
      void displayData();
};
```



```
Example :: Example( int data )
{
   this->data  =  data;
}
```

class member *function parameter*

Destructors

- When an object is no longer required, or it goes out of scope, (a function executes `return`, etc.) it is destroyed.
- A destructor is automatically executed to “clean up” after an object.
- A class only needs to have one destructor.
- A destructor has the same name as the class preceded by a ~
- It does not take any parameters and does not return any value.

Automatically calls a constructor

```
Coordinate* pointA = new Coordinate;

.   .   .   .
```

Automatically calls a destructor pointA-> ~Coordinate();

```
delete pointA;
```

Destructor

- *Constructors:*

```
Coordinate();                          // default constructor
Coordinate( float a, float b);         // constructor with
                                       // two parameters
```

- *Destructor:*

```
~Coordinate();
```

```
class Coordinate {
  public:
      Coordinate( float a=0.0, float b=0.0 );
      ~Coordinate();
      void shiftHrz(float shift) { x += shift; }
      void shiftVert(float shift) { y += shift; }
  private:
      float x;
      float y;
};
```

Destructor

- If the destructor is not defined, the compiler calls a default destructor – this may not be appropriate in some cases.
- When you have **dynamically allocated data members** in your class, you **have to** provide the destructor to clean up memory.

```
class Example{
  public:
       Example();
     ~Example();
      void setData( int s, char* p);
  private:
       int size;
       char *ptr;
 };

 Example :: Example()    // default constructor

 { size = 80;  ptr = new char[size]; }

 Example :: ~Example()   // destructor
 { delete [] ptr; }
```

to delete a memory block allocated by the constructor

Quiz

This code can’t be compiled. Why?

```
class Coordinate
{
   private:
       float x;
       float y;
       ~Coordinate();
   public:
       Coordinate(float a=0.0, float b=0.0);
       void shiftHrz(float shift) { x += shift; }
        void shiftVert(float shift) { y += shift; }
};

int main()
{
    Coordinate pointA;

    return 0;
}
```

Should automatically calls the destructor

`pointA.~Coordinate();`

but it is private

Copy Constructor

```
Coordinate pointA;
Coordinate pointB = pointA;
```

- When an object is declared and initialized using another object at the time of declaration, a copy constructor is called automatically.
- If a copy constructor is not defined in a class, a default copy constructor is generated by the compiler.

  Example: the data members of pointA are copied member-wise into pointB by a default copy constructor generated by the compiler:

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

- Such basic member-wise copy may not be sufficient in some cases.

Copy Constructor

- A copy constructor needs to be provided instead of the default one **if dynamic memory allocation is used** to store data.

- Copy constructor has the following form:

```
className(const className& source);

cString( const cString& source );
```

- The copy constructor must **explicitly copy content** of dynamically and statically allocated data members.

Copy Constructor

- The copy constructor is invoked by the compiler when you:

  1. declare an object and initialize it to another object of the same class

     Example: `Coordinate pointB = pointA;`

  2. pass an object by value to a function

     Example: `float getDistance( Coordinate point );`

  3. return an object by value from functions

     Example: `Coordinate getPosition();`

- If there is no copy constructor defined in your class, the compiler uses the default one and this copy constructor may not always work correctly.

Quiz

- Which function call will invoke a copy constructor?

```
class Example {
. . . .
};

// function prototypes (non-member functions)
void processData( Example ex1 );
void displayData( Example& ex1 );
. . .

Example obj1;
// function calls
processData( obj1 );
displayData( obj1 );
```

```
processData( obj1 );
```

const Functions

```
class Example {
  private:
      int data;
  public:
      Example( int number=0 );
      void setValue(int val);
      int getValue() const;
      void display() const;
};
```

These functions can modify data members of the class

These functions cannot modify data members

```
int Example::getValue() const {
     return data;
}
void Example::display() const {
     cout << data ;
}
```

Only const member functions can be used with const objects

Quiz

Find a bug in this code

```
class Data{
   private:
        int data;
   public:
        Data( int number=0 )
        void setValue( int data) {this->data = data;}
        int getValue() const { return data; }
};

void process( const Data& dataObj, int newValue )
{
    int c = dataObj.getValue();

    dataObj.setValue(newValue);
}
```

setValue() is not a const function and therefore it cannot be used with an object passed as const reference

Relationship Between Classes

Dr Obada Al Khatib

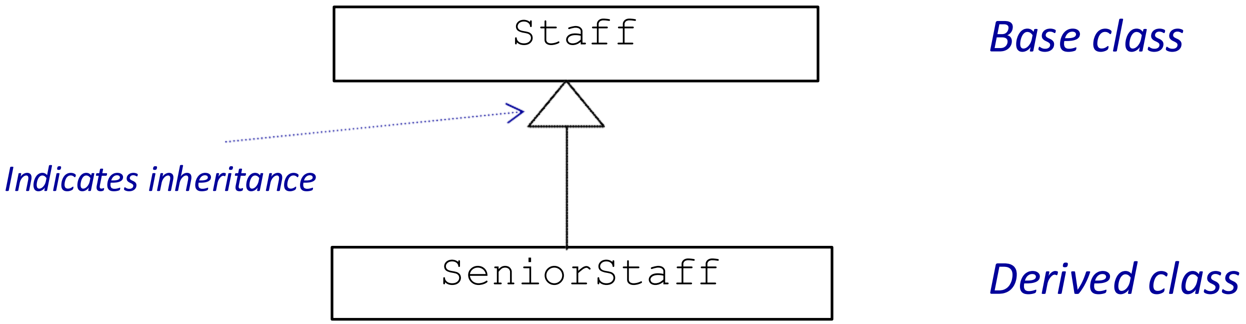
obadaalkhatib@uowdubai.ac.ae

UNIVERSITY
OF WOLLONGONG
IN DUBAI

Inheritance

- If you need to write a program using a new class named `SeniorStaff`, it may be easier if `SeniorStaff` could reuse properties of already defined class `Staff`.
  - `SeniorStaff` will not need to redefine members which are already defined as `Staff` members.
  - `SeniorStaff` may also require some additional data members and functions (e.g., `bonus` or `getBonus()`).
  - The `SeniorStaff` class might require a different display format than the `Staff` class, so `display()` function may need to be substituted with a new version.

Inheritance



- `SeniorStaff` class is **derived** from `Staff` class. It inherits properties and behaviours of `Staff` class and can use them.
- `SeniorStaff` "is a" `Staff` member too.
- `Staff` is called a **parent class,** or **base class.**
- `SeniorStaff` is called a **child class** or **derived class.**

Example: using inheritance

Indicates that `Staff` is a base class for `SeniorStaff`

```
class Staff
{
  private:
    int idNum;
    string firstName;
    string lastName;
  public:
    Staff();
    ~Staff();
    void display();
    int getId();
};
```

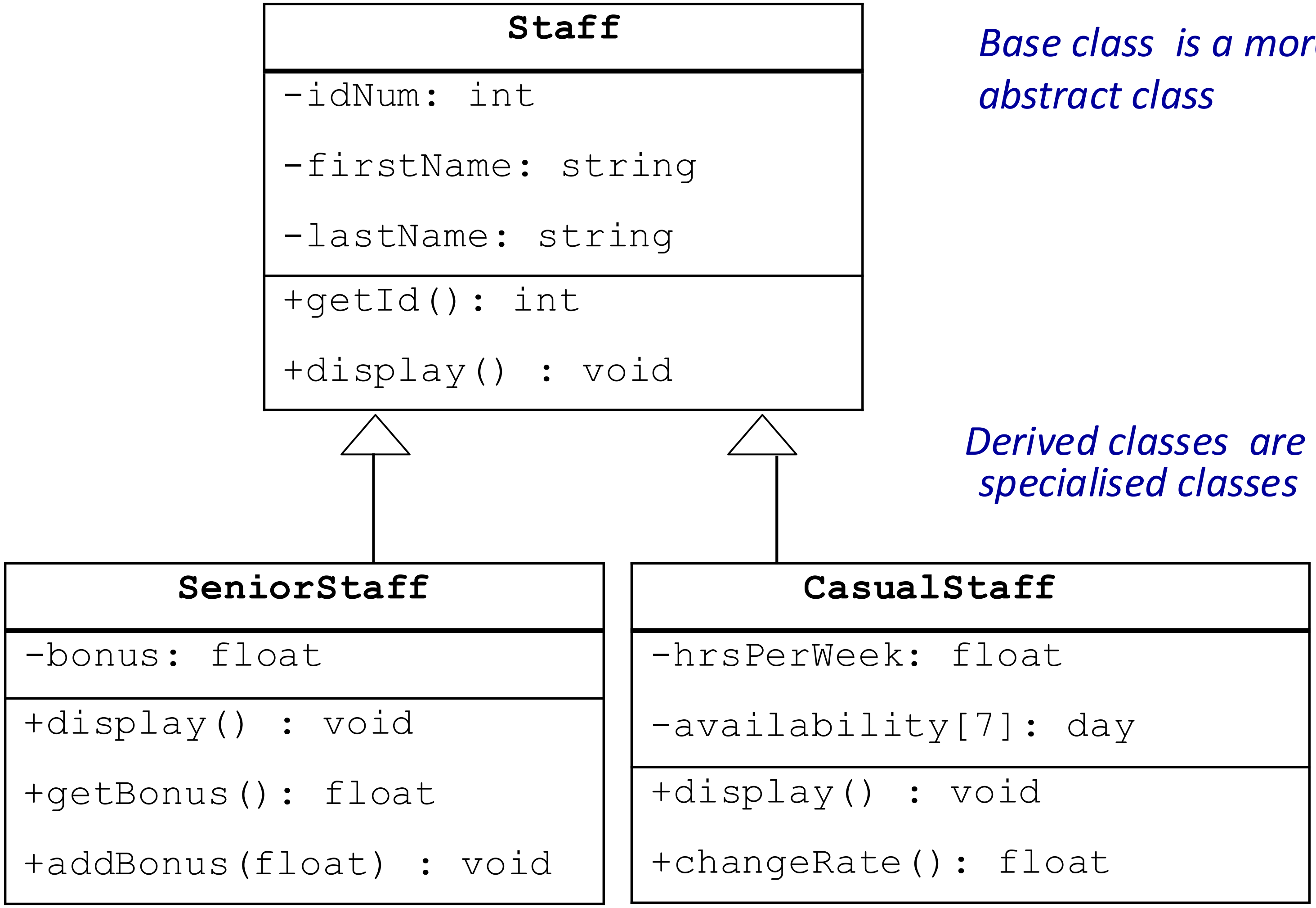
```
class SeniorStaff  : public Staff
{
  private:
    float bonus;
  public:
    SeniorStaff();
    ~SeniorStaff();
    void display();
    float getBonus();
    void addBonus(float);
};
```

Inheritance

- Inheritance is a form of software reusability.
- New classes are created from existing classes by:
  - Absorbing their properties and behaviours.
  - Upgrading (overriding) some of behaviours with new capabilities specific to derived classes.
  - Adding new properties and behaviours.
- Inheritance has an important feature:
  - An object of a derived class type may also be treated as an object of the base type.

`SeniorStaff` member **is a** `Staff` member too.

Inheritance: UML class diagram



Base class is a more abstract class

Derived classes are more specialised classes

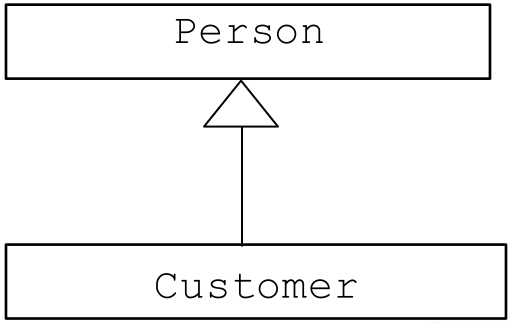
Example: Base Class

```
class Person {
   private:
        int    idNum;
        string firstName;
        string lastName;
   public:
        void setData(int id, string f, string l);
        void printData();
};
void Person::setData(int id, string first, string last) {
    idNum = id;
    lastName = last;
    firstName = first;
}
void Person::printData() {
   cout << "ID: " << idNum << ", Name: " << firstName << " "
   << lastName << endl;
}
```

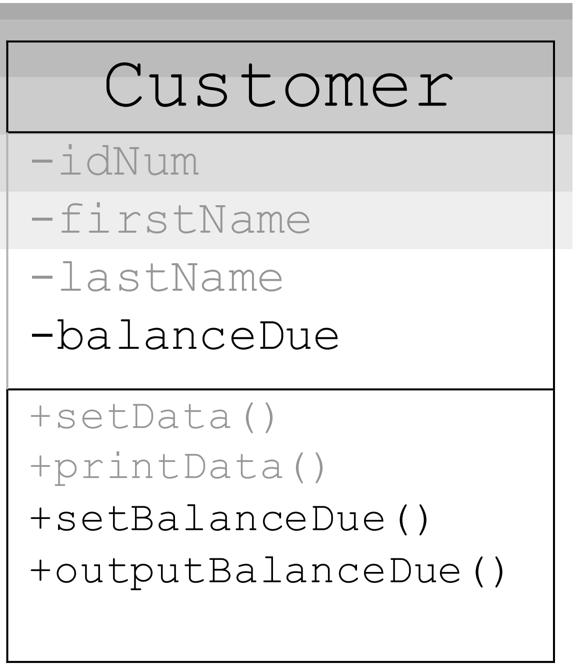
Derived Class

- You can say every derived class “is a” parent class too:
  - *For example*, every `Customer` “is a” `Person`
- The `Customer` class shown below contains all the members of `Person` because it inherits them.

```
class Customer : public Person
{
   …
};
```



```
class Customer : public Person {
   private:
        double balanceDue;
   public:
        void setBalanceDue(double);
        void outputBalanceDue();
};
void Customer::setBalanceDue(double bal)
{
   balanceDue = bal;
}
void Customer::outputBalanceDue() {
   cout << "Balance due $" << balanceDue << endl;
}
```



```
Customer
-idNum
-firstName
-lastName
-balanceDue
+setData()
+printData()
+setBalanceDue()
+outputBalanceDue()
```

- The `Customer` class has members inherited from the `Person` class and some additional members which are specific to `Customer`.

Using Both Inherited and Own Members

```
int main() {
   Customer cust1;


  cust1.setData (537, “John”, “Hanley”);
  cust1.printData();


  cust1.setBalanceDue(123.45);
  cust1.outputBalanceDue();
}
```

Methods defined in the base class

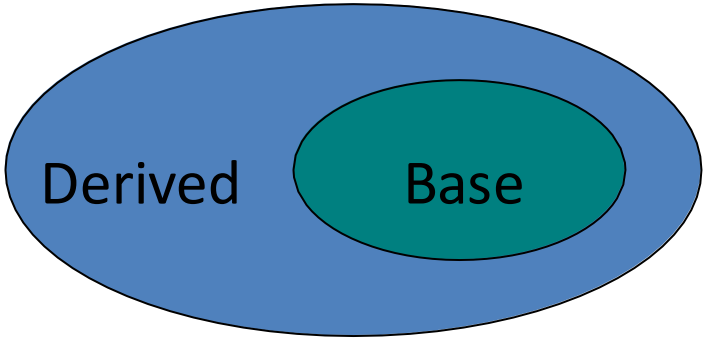
Methods defined in the derived class

Instantiation of a Derived Class

```
Customer cust1;
```

- When an object of a derived class is declared it results in:
  1. calling a default constructor of the base class
  2. calling a default constructor of the derived class

- This can be interpreted as a derived class object that contains a base class object.



Inheritance Restrictions: Private Members

```
void Customer :: outputBalanceDue() {
    cout << "ID :" << idNum << ", balance due $" << balanceDue << endl;
}
```

This data member will cause a compilation error

- The private members of the base class cannot be directly accessed by the member functions of the derived classes.

Solution 1

```
class Person { // base class
  private:
       int     idNum;
       string firstName;
       string lastName;
  public:
       void setFields(int id, string f, string l);
       void outputData();
       int getID() { return idNum; }
};
.   .   .   .
```

Member functions of a derived class can access private data members of the base class through the public functions of the base class

```
void Customer :: outputBalanceDue() {
   cout << "ID :" << getID() << ", balance due $" <<
   balanceDue << endl;
}
```

Solution 2: Protected Members

```
class Person { // base class
   protected:
         int        idNum;
         string    firstName;
         string    lastName;
   public:
         void setFields(int id, string f, string l);
         void outputData();
};
```

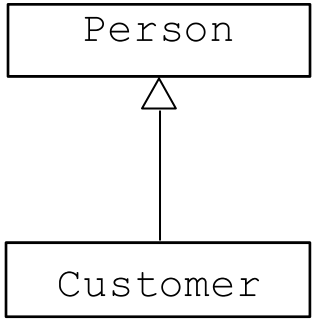
- `protected` members of the base class can be accessed directly by member functions of derived classes

```
void Customer :: outputBalanceDue() {
    cout << "ID :" << idNum << ", balance due $" << balanceDue << endl;
}
```

As it's a `protected` *member of the base class, it can be accessed directly form a derived class*

Class Access Specifiers

- When you define a derived class you can specify it's relationship to the base class
- There are three options



Example: `Customer` is a `Person`

```
1. class Customer : public Person
2. class Customer : protected Person
3. class Customer : private Person
```

- These three class access specifiers **don't affect** how the **derived** class accesses members of the base class (access is based purely on the base class access modifiers).
- The `private` access specifier changes **all base class members** to `private` in relation to **external functions** (including main) which use objects of the derived class. The specifier doesn't affect access to its own members from external functions.

Class access specifiers

- If a derived class uses `public` specifier for inheritance:
  - `public` base class members remain `public` in its derived class.
  - `protected` base class members remain `protected` in its derived classes.
  - `private` base class members remain inaccessible directly from its derived class.
- If a derived class uses `protected` for inheritance:
  - `public` base class members become `protected` in its derived Class.
  - `protected` base class members remain `protected` in its derived class.
  - `private` base class members remain inaccessible directly from its derived classes.
- If a derived class uses `private` for inheritance, all base class members become `private` in its derived class.

private Access Specifier

```
class Base {
  private:
    int num;
  public:
    void setNum(int n);
    int getNum();
};

void Base::setNum(int n){
    num = n;
}

int Base::getNum() {
    return num;
}
```

```
class Derived : private Base {
    private:
        int data;
    public:
        void setDat(int n, int d);
        int getValue();
        int getNumber();
};

void Derived::setDat(int n,int d)
{
    setNum(n); // can be accessed
    data = d;
}

int Derived::getValue() {
    return value;
}
int Derived::getNumber() {
    int nmbr=getNum();// can be accessed
    return number;
 }
```

private Access Specifier

```
int main()
{
    Base b; // declare an object of Base class
    b.setNum(4);             // OK as it’s public in Base
    int nm = b.getNum(); // OK as it’s public in Base

    Derived d1; // declare an object of Derived class
    d1.setDat(10, 20); // OK as it’s public in Derived
    cout << “value=" << d1.getValue() << endl;
    cout << "num=" << d1.getNum() << endl;
}
```

Technically this is possible, however you need to consider if private access specifier is really needed for your class hierarchy

`getNum()` is a public member of `Base`. However, due to the private access specifier, it becomes a private member of `Derived`.

We cannot call a `private` member function of the class from `main()`

Relationship Between Constructors

- A derived class constructor **always** calls the constructor for its base class **first** to initialize the base class members.
  - If the derived-class constructor is omitted, the derived class default constructor calls the base class default constructor.
- As the classes may have several constructors defined, you need to specify explicitly the relationship between constructors

Example:

```
Derived(int a, float b) : Base(a) { derivedB = b; }
```

Calls the base class constructor with one parameter

Relationship Between Constructors

```
class Base {
   private:
         int num;
   public:
         Base( int n = 0 ) : num(n) { cout<<“Base is called”; }
         int getNum();
};
```

```
class Derived : public Base {
   private:
         float val;
   public:
         Derived( int n = 0, float v=0.0  );
};

// definition of the constructor for Derived class
Derived::Derived(int n, float v) : Base(n), val(v)
{ cout<<“Derived is called”<<endl; }
```

```
Derived der1(3, 4.5); // will result in calling Base(int) then Derived(int,float)
```

Functions with Similar Names

- A base class and a derived class may have public functions with the same name and the same list of parameters

```
class Base
{
    public:
      void print() { cout<<"Base"<<endl;   }
};

class Derived : public Base
{
    public:
      void print() {  cout<<"Derived"<<endl; }
};
```

- Which print will be called in this case?

```
int main()
{
   Derived obj;
   obj.print();
   return 0;
}
```

Functions with Similar Names

- A derived class may not have its own function, but it may be inherited from the base class.

```
class Base
{
    public:
      void print() { cout<<"Base"<<endl;   }
};

class Derived : public Base
{
    public:
};
```

- Will `print()` from the base class be called in this case, or there will be a compilation error?

```
int main()
{
   Derived obj;
   obj.print();
   return 0;
}
```

Which member function gets invoked?

- When any class member function is called, the following steps take place:

  1. The compiler looks for a matching function in the class of the object. If it's found, it is called.
  2. If no name match is found in this class, the compiler looks for a matching function in the parent class.
  3. If no match is found in the parent class, the compiler continues up the inheritance hierarchy, looking at the parent of the parent, until the base class is reached.
  4. If no match is found in any class, this results in a compilation error.

Function Name Resolution

- How to call `print()` that belongs to the base class from a derived class?

```
class Base
{
    public:
      void print() { cout<<"Base"<<endl;   }
};

class Derived : public Base
{
    public:
      void print() {
          cout<<"Derived"<<endl;

          ~~print();~~              // this a trivial call of itself – an infinite loop

          Base::print();    // you need to use the scope resolution operator
      }
};
```

Function Overloading

- How to reuse functions implemented in the parent class?

```
class Base
{
    private:
       int baseData;
    public:
       void setData(int b) { baseData = b;  }
};

class Derived : public Base
{
   private:
      float derData;
   public:
      void setData(float d) { derData = d;  }
      void setData( int bs, float dr )
      {
         Base::setData(bs);   // use the scope resolution to call a base class function
         derData = dr;
      }
};
```

As a class may have overloaded functions, always use the scope resolution operator to call a parent class function

`static` Members of Classes

- A number of independent objects can be created from one class

```
int main()
{
   EnergyBill customer1, customer2, customer3, customer4;
   . . .
   return 0;
}
```

- Is it possible for objects to share common data?

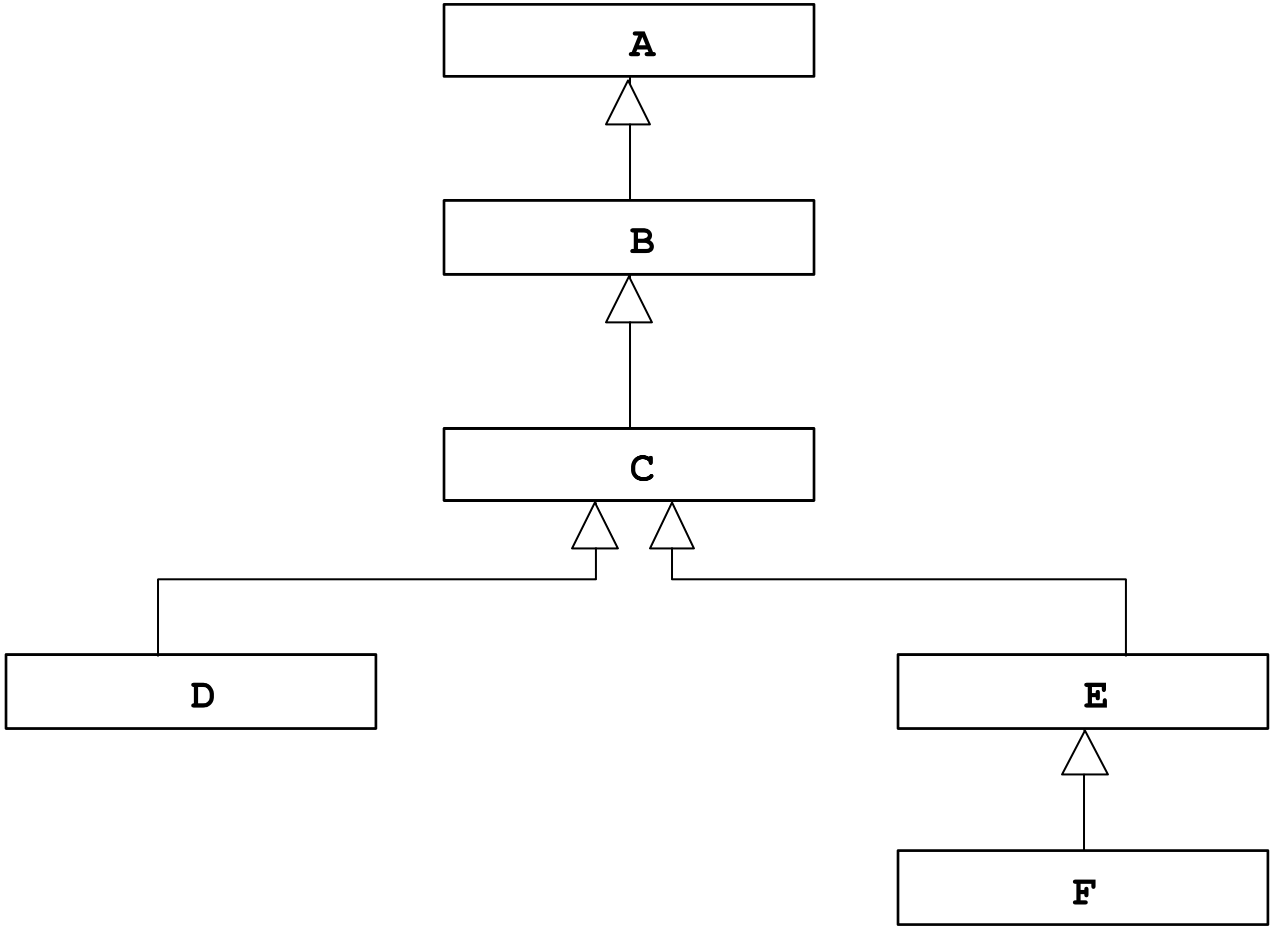
  Example: All energy bills must have the same rate

- All data members which should share exactly the same value among all declared objects should be defined as **static.**
- A static class member is shared by all objects of the class.

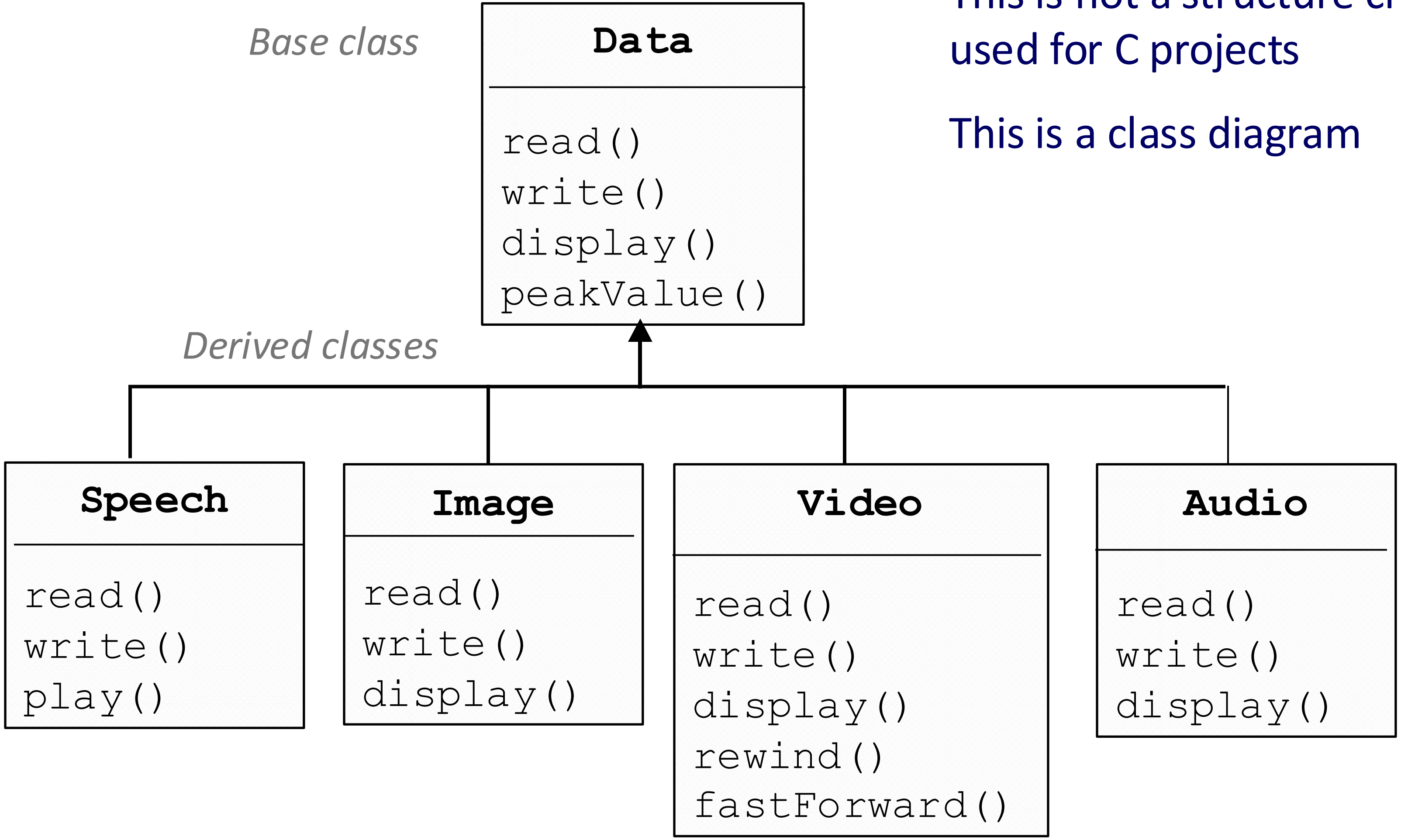
static Members of Classes

```
class EnergyBill {
    private:
        int customerNumber;
        float totalUsage;
        float amountDue;
        static float rate;      // a static data member
    public:
        EnergyBill( int custNum ); // static members are not initialized by constructors
        float calculateAmountDue();
        void updateUsage( float energyUsage );
        void printBill();
        .   .   .
};

double energyBill :: rate = 0.2507; // initialise the static member
 int main()
 {
    // these two objects will always use identical rate
    EnergyBill customer1, customer2;
    . . .
    return 0;
 }
```

Multiple layers of inheritance



Final Example



Base class

Data

```
read()
write()
display()
peakValue()
```

Derived classes

Speech

```
read()
write()
play()
```

Image

```
read()
write()
display()
```

Video

```
read()
write()
display()
rewind()
fastForward()
```

Audio

```
read()
write()
display()
```

This is not a structure chart used for C projects

This is a class diagram