# CSCI251/CSCI851 Autumn-2020
# Advanced Programming (**S4a**)

## Programming with Class I:[†]
## Fundamental syntax and some introductory UML

[†] A reference to a textbook by Dr. Neil Gray

# Outline

- Introducing classes.
- Encapsulation and access specifiers.
- A warning on private.
- Designing classes.
- The `this` pointer.

- With UML throughout…

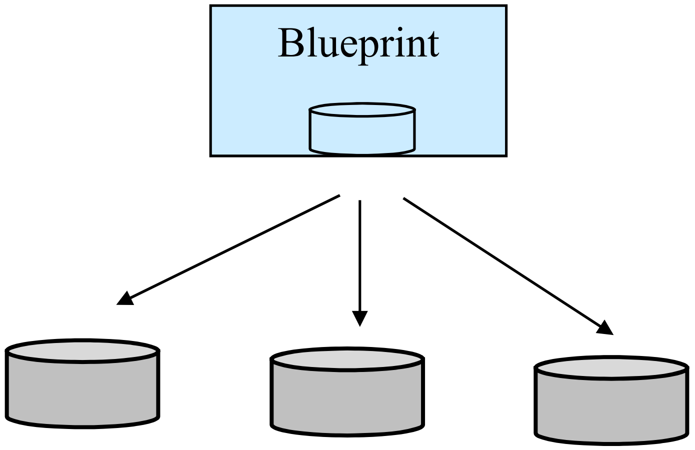
# Introducing classes

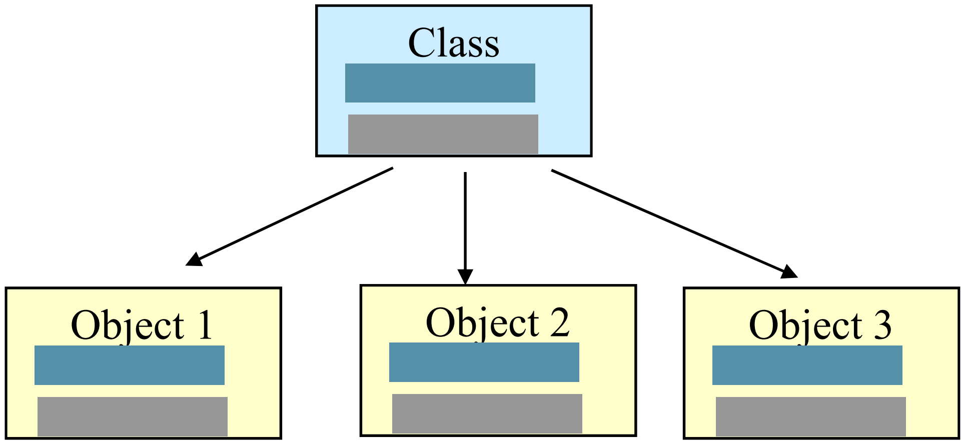
- A **class** is a user defined data type:
  - Classes provide a description for building objects.
  - Classes provide prototypes or blueprints for objects.
  - Classes provide a convenient way to group related data *and* the functions which are used to process the data.
  - When you create an object from the class, you automatically create all the related fields.
- **Abstract data type (ADT):** a type you define.

# Objects and Classes

Parts of a technical system are produced based upon their description: blueprints.

Parts of a software system are generated based upon their description: classes.





- A class is an abstraction, it is an abstract data type.
- An object is an instance generated and placed in computer memory.

# Handling complexity: …

- Object oriented design tends towards managing richer and broader scenarios than procedural programming.
- The systems modelled are typically complicated and require a fair bit of pre-coding thought, i.e. design.
- We are also often wanting to think about writing code that will support multiple related scenarios, rather one very specific problem.
- When you start thinking about coding something one activity that you might carry out is sketching down how components relate to each other, and …
- … UML provides a language for this kind of sketching, and, because it is standardised, it allows others to understand what you are doing.

# UML

- UML (Unified Modelling Language) provides an unambiguous mechanism to describe technical systems.
- We model to identify components, check if something has any chance of working before we build it, and to confirm with our clients that the system is going to do what they want.
  - It's standardised so others can understand what we have sketched.
  - Others? Clients are unlikely to be software engineers but some parts of UML provide accessible representations of some of what is going to happen.

# UML strengths

- It is formal with a strongly defined meaning.
  - UML models cannot be misinterpreted, if interpreted according to the rules.
- It is concise.
- It is comprehensive and can describe any aspect of a software system.
- It is scalable and can be used for large or small projects.
- It is an international standard.
  - Current version 2.5.1, 05-December-2017.
  - https://www.omg.org/spec/UML/2.5.1

# Types of UML models

- In one representation, there are three basic types of models:
  - **Functional view**:
    - Answers: What should the system should do?
    - Perspective: User.
    - Use cases and use case diagrams.
  - **Structural view**:
    - Answers: How must the system be implemented?
    - Perspective: System architect.
    - Class diagrams and object diagrams.
  - **Behavioural view**:
    - Answers: How does the system operate?
    - Interaction diagrams and state diagrams.

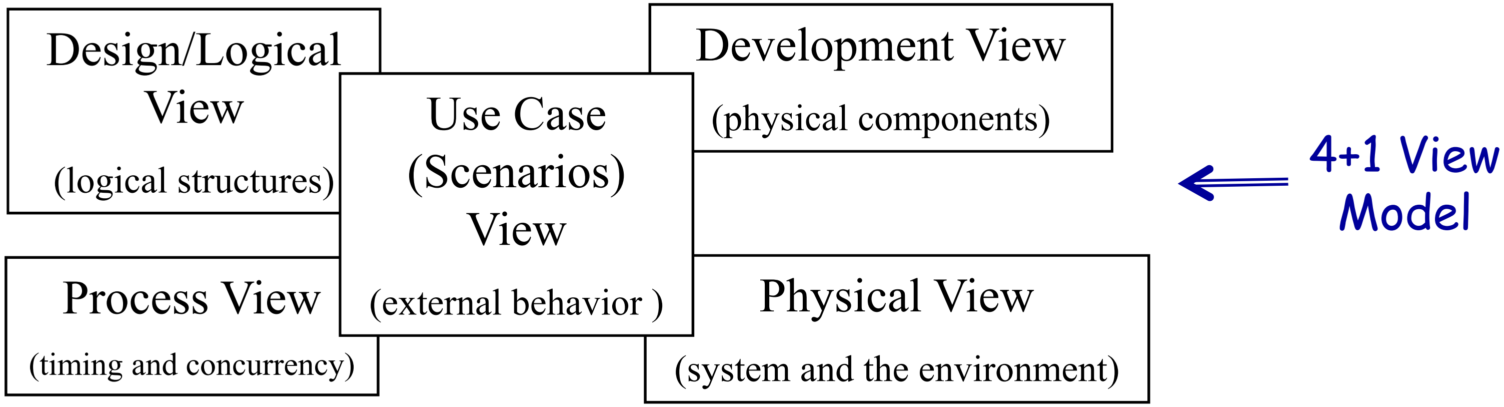
- Another representation reorganises the modelling into three different classes:
  - **Class model**: Structural description.
    - Class and object diagrams.
  - **State model**: Describes evolution of objects of one class/type, by modelling transitions.
    - State diagrams.
  - **Interaction model**: Describes how objects interact.
    - Use cases, sequence diagrams and activity diagrams.

# Relating the models

- The events in a state diagram relate directly to operations on objects in the class model.
  - We might see the operation as a link on an object diagram but the state diagram describes the control associated with that relationship.
- While state diagrams are for a single class, we might need to record references between state diagrams, basically between objects/classes.
  - This is done using the interaction descriptions of the interaction model.
- The state and interaction models act on the data/behaviour framework described by the class model.

# Another representation : 4 + 1 view

- How can an information system be described?

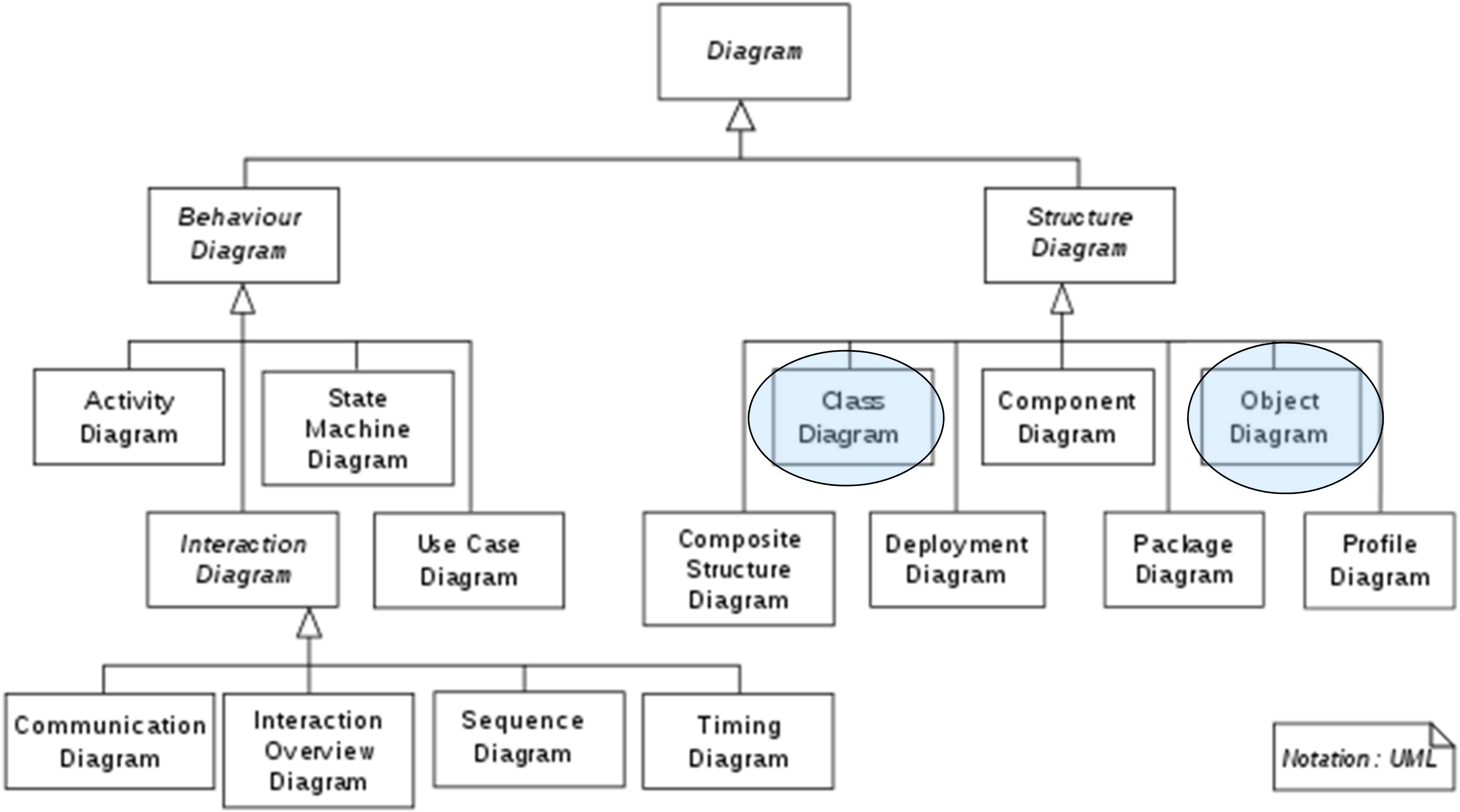


- **Design/logical view**: An abstract description of parts which the system is consists of, visible end-user functionality – UML class, object, state and interaction diagrams.
- **Process view**: Describes system processes – UML activity diagrams.
- **Development view**: Describes how system parts are combined into modules – UML component diagram.
- **Physical view** : Describes how abstract parts of the system are mapped into the deployed system – UML deployment diagrams.
- **Use case view** : describes user-system interaction – Use Case diagrams.

# 4 + 1: Why use different views?

- Different views complement each other.
  - They show different information, or represent the same information from a different perspective.
- This information redundancy is useful for:
  - **Error checking**: The consistency can be checked.
  - **Clarity**: A complex concept may appear significantly simpler from one of the viewpoints, or some people may find it easier to understand description in a particular format.
  - **Completeness**: You are less likely to miss an important feature if you have to think about the system from several different perspectives.

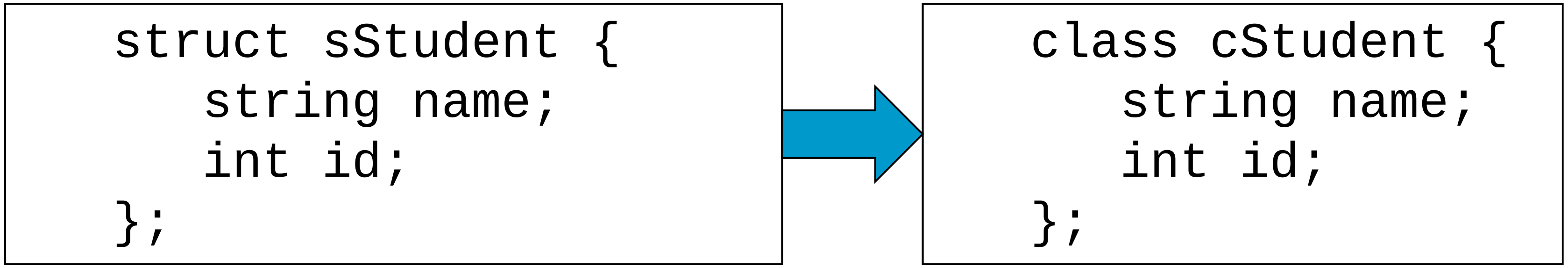
# The diagrams



https://en.wikipedia.org/wiki/Unified_Modeling_Language
See also http://www.uml-diagrams.org/uml-25-diagrams.html

# Back to C++ Classes: Fundamental syntax

- We have seen the basic syntax for structs and unions, and the really basic syntax for the C++ class is the same.



```
struct sStudent {
    string name;
    int id;
};
```

```
class cStudent {
    string name;
    int id;
};
```

- We are just getting started, and a good place to start is the difference between structs and classes mentioned earlier …

- We declare an instance/variable of each type, i.e. objects, as follows:

```
sStudent studentA;
cStudent studentB;
```

- But if we try to set the id values …

```
studentA.id = 123;
studentB.id = 124;
```

- … the struct object will succeed but the class object will fail.
- Members of a class are private by default, whereas they are public by default for a struct.

- The access specifiers/modifiers for data and methods have three different types:
  - **Public** can be directly accessed from outside the object.
  - **Private** can only be directly accessed by internal methods.
  - **Protected** can be directly accessed by objects of subclasses, but not by arbitrary external objects.
- We will look at protected later, but private and public allow us to capture the idea of encapsulation.

# Encapsulating Class Components

- To **encapsulate** components is to hide them in a container:
  - Encapsulation is an example of making something a black box.
- To access some of the hidden components an **interface** should be provided.
- Often we want users of an object to only see the “surface” of the object.

# Public Functions on Private Data

```
class Classname
{
     public:
     // assorted user-accessible functions,etc → Interfaces
     private:
     // data (& functions) hidden from the user
};
```

- This is one of our primary design considerations, effectively following the principle of least privilege.
- If a data member is private we can properly control it through member functions :
  - A variable may have a limited range.
    - Using a private function we can ensure that it always stays within that range after manipulations.
  - A variable may require specific output formatting.
    - Again, we can have a public member function for displaying data which uses the appropriate format.

- Data might sometimes be public, but usually only if it’s a `const`, likely a `static const`.
- This makes sense for fixed values:
  - π, e, ħ …

# Access specifiers: Code syntax

- Access specifiers are mutually exclusive flags in C++, exactly one is always on and everything after one access specifier and before another has that access.

```
class Student {
  private:
    string name;
    int age;
    string password;
    bool verifyPassword();
  public:
    void display();
    bool login( string pswd );
};
```

This is only the definition, we will get to an implementation soon.

# Implementing Class Functions

- When we construct a class, we need two parts:
  - **Definition section:** Contains the class name, variables (attributes), and function prototypes.
  - **Implementation section:** Contains the member function definitions.
- We need to use both the class name and the scope resolution operator (::) when we define class functions in the implementation section.

```
void Student :: display() {
   cout << name << " : " << age << endl;
}
```

- Typically the definition will appear in a `.h` header file, and the implementation in a `.cpp` file.
- However we sometimes put the implementation into the definition.

```
void display() {cout << name << " : " << age << endl;}
```

- Functions defined in the class in this way are implicitly inline, so we don’t need the keyword `inline`.

# In-class initialisers

```
class thing {
  public:
    string name;
    int value1;
    int value2;
};
```

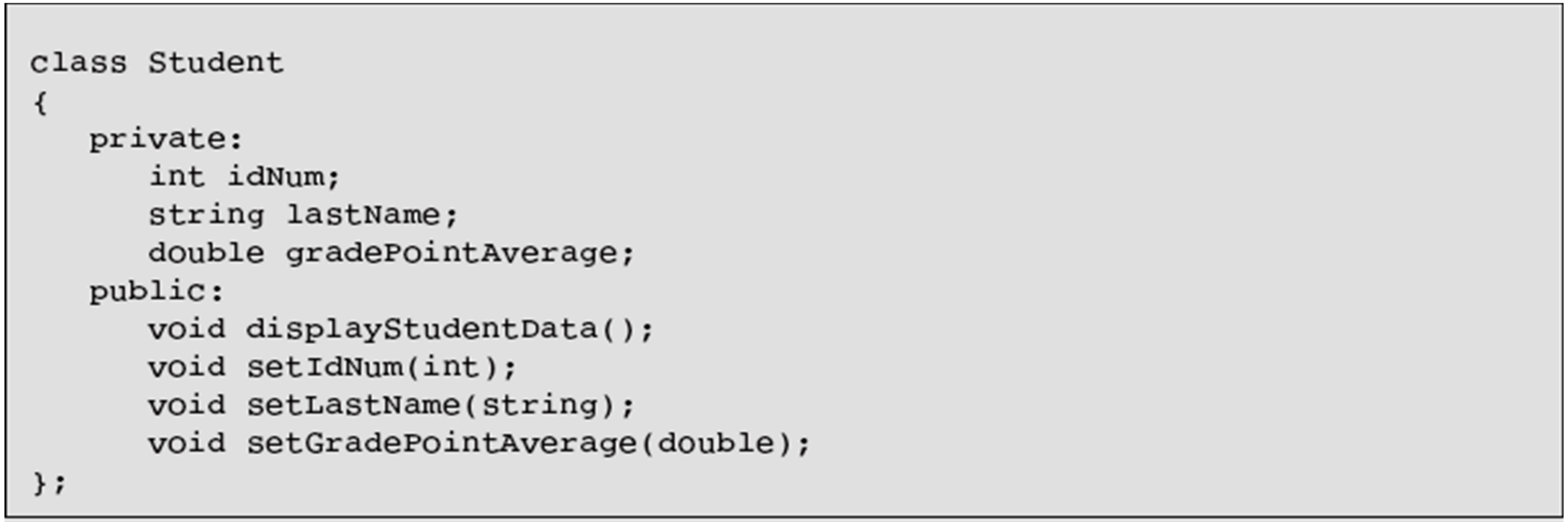
- In our definition example we didn’t initialise the values of the variables …
- … and for non-const you couldn’t pre C++11.

```
12:20:35 $ g++ -std=c++03 test.cpp
test.cpp:8:17: warning: non-static data member initializers only available with -
std=c++11 or -std=gnu++11
    int value2 = 5;
```

- But you can with C++11.

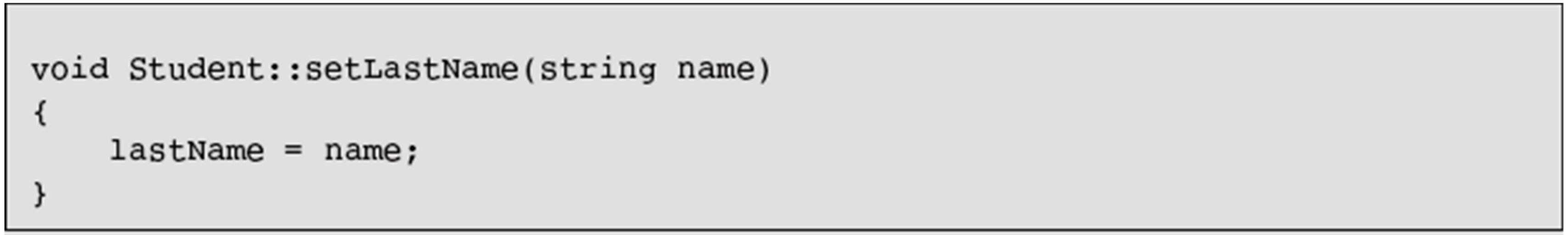
```
class thing {
  public:
    string name;
    int value1;
    int value2 = 5;
};
```

# A complete sample class: From Joyce Farrell, Object Oriented Programming Using C++, 3rd/4th Edition, Thomson Learning.



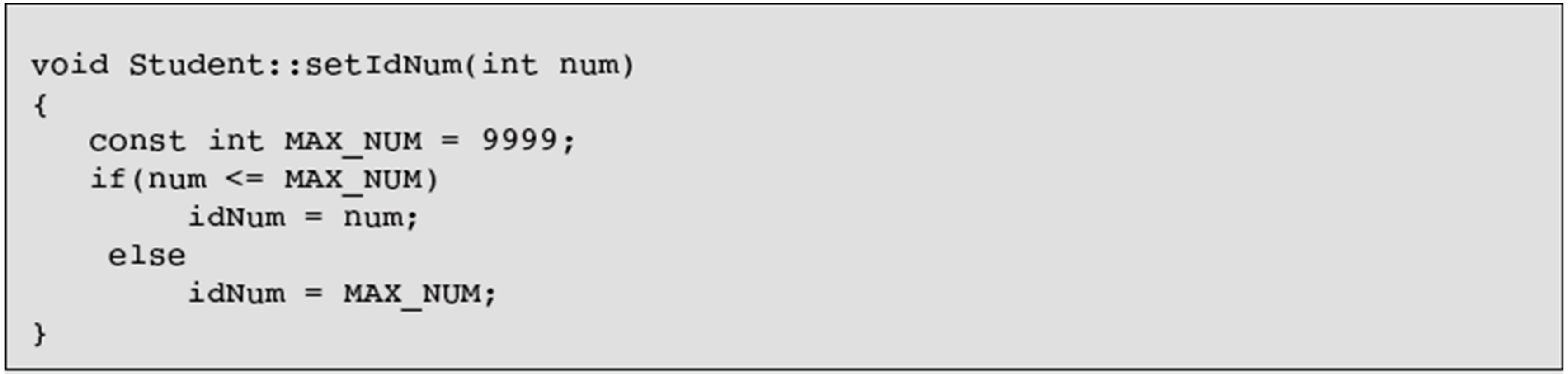
```
class Student
{
   private:
      int idNum;
      string lastName;
      double gradePointAverage;
   public:
      void displayStudentData();
      void setIdNum(int);
      void setLastName(string);
      void setGradePointAverage(double);
};
```

**Figure 7-5** `Student` class with set functions for private data



```
void Student::setLastName(string name)
{
    lastName = name;
}
```

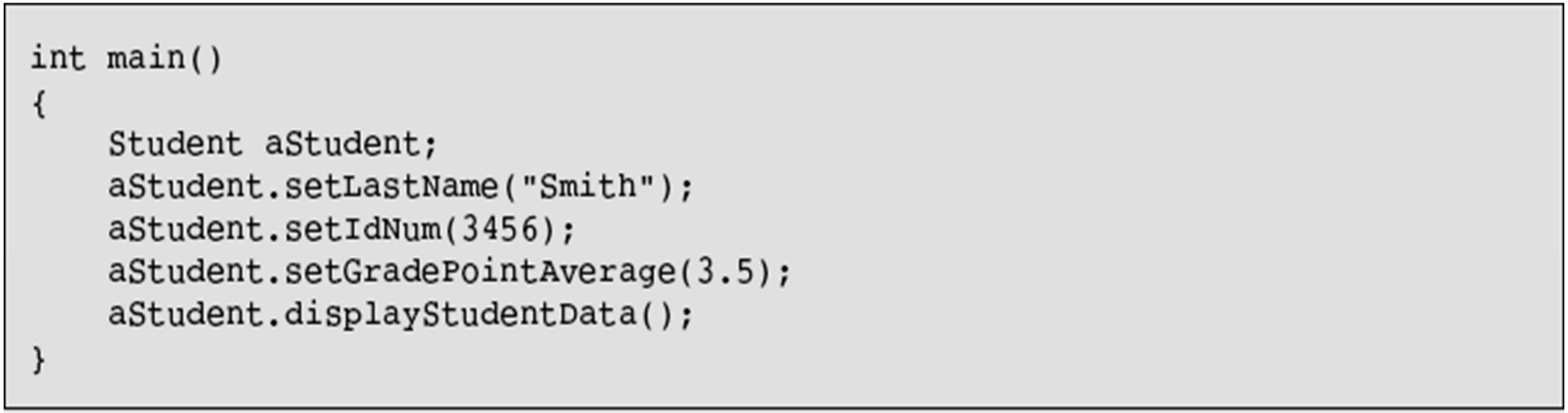
**Figure 7-6** The `setLastName()` function



```
void Student::setIdNum(int num)
{
   const int MAX_NUM = 9999;
   if(num <= MAX_NUM)
        idNum = num;
    else
        idNum = MAX_NUM;
}
```

**Figure 7-7** The `Student` class `setIdNum()` function

…

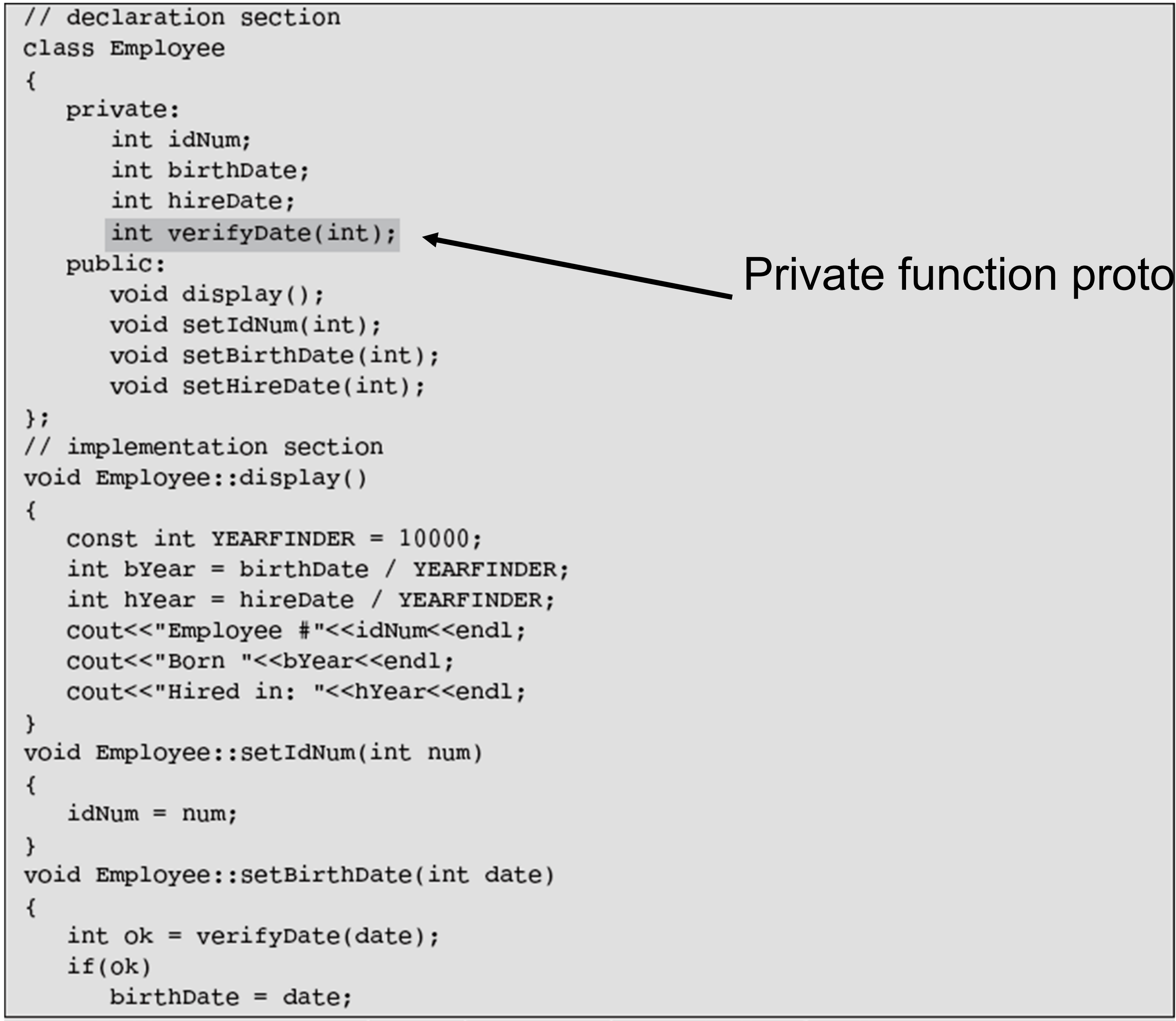


```
int main()
{
    Student aStudent;
    aStudent.setLastName("Smith");
    aStudent.setIdNum(3456);
    aStudent.setGradePointAverage(3.5);
    aStudent.displayStudentData();
}
```

**Figure 7-8** The `Student` class and a demonstration `main()` function

We are restricting the range of `IdNum`.

# Private Functions …



```
// declaration section
class Employee
{
   private:
      int idNum;
      int birthDate;
      int hireDate;
      int verifyDate(int);
   public:
      void display();
      void setIdNum(int);
      void setBirthDate(int);
      void setHireDate(int);
};
// implementation section
void Employee::display()
{
   const int YEARFINDER = 10000;
   int bYear = birthDate / YEARFINDER;
   int hYear = hireDate / YEARFINDER;
   cout<<"Employee #"<<idNum<<endl;
   cout<<"Born "<<bYear<<endl;
   cout<<"Hired in: "<<hYear<<endl;
}
void Employee::setIdNum(int num)
{
   idNum = num;
}
void Employee::setBirthDate(int date)
{
   int ok = verifyDate(date);
   if(ok)
      birthDate = date;
```

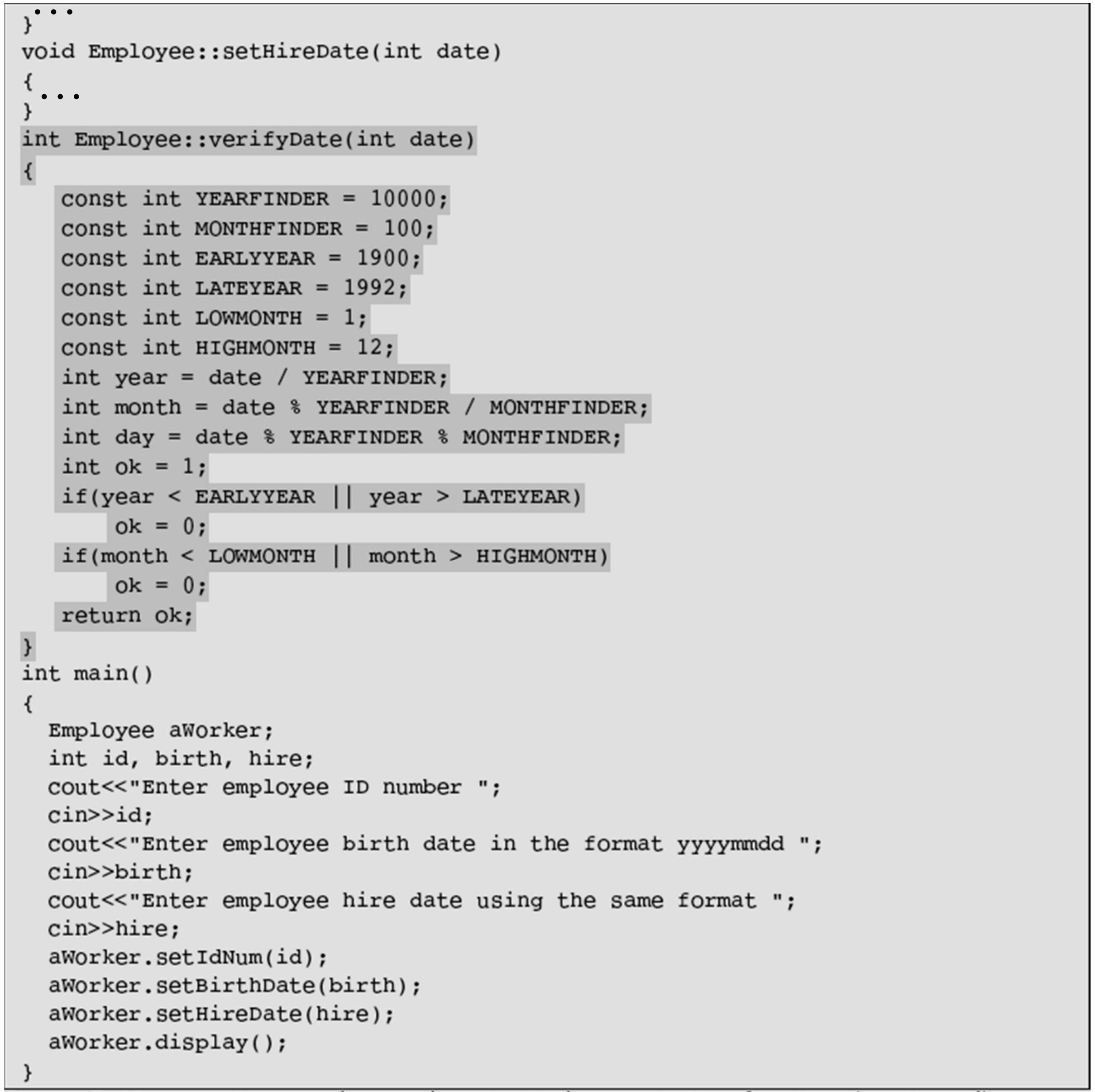
Private function prototype.

**Figure 7-10** Employee class and `main()` demonstration function

This function verifies dates. It is only used by class functions so can be private.

It checks that the dates conform to various rules.

The date is entered in yyyymmdd format.



```
 ...
}
void Employee::setHireDate(int date)
{
 ...
}
int Employee::verifyDate(int date)
{
   const int YEARFINDER = 10000;
   const int MONTHFINDER = 100;
   const int EARLYYEAR = 1900;
   const int LATEYEAR = 1992;
   const int LOWMONTH = 1;
   const int HIGHMONTH = 12;
   int year = date / YEARFINDER;
   int month = date % YEARFINDER / MONTHFINDER;
   int day = date % YEARFINDER % MONTHFINDER;
   int ok = 1;
   if(year < EARLYYEAR || year > LATEYEAR)
       ok = 0;
   if(month < LOWMONTH || month > HIGHMONTH)
       ok = 0;
   return ok;
}
int main()
{
  Employee aWorker;
  int id, birth, hire;
  cout<<"Enter employee ID number ";
  cin>>id;
  cout<<"Enter employee birth date in the format yyyymmdd ";
  cin>>birth;
  cout<<"Enter employee hire date using the same format ";
  cin>>hire;
  aWorker.setIdNum(id);
  aWorker.setBirthDate(birth);
  aWorker.setHireDate(hire);
  aWorker.display();
}
```

**Figure 7-10** Employee class and main() demonstration function (continued)

# A warning on private …

- Private is a compiler setting.
- It doesn’t actually stop you from interacting with the location… ☹
- …

```
class thing {
public:
    int value1 = 5;
    void display(){cout << value2 << endl;}
private:
    int value2 = 77;
};

int main()
{
    thing A;
    cout << A.value1 << endl;
    cout << A.value2 << endl;
    cout << *(&A.value1 + 1) << endl;
    A.display();
    cin >> *(&(A.value1) + 1);
    A.display();
    return 0;
}
```

# Not so `const` ☹

- We can do something similar to change `const`...

```
class thing {
public:
    int value1 = 5;
    const int value2 = 77;
};

int main()
{
    thing A;
    cout << A.value1 << endl;
    cout << A.value2 << endl;
    cin >> *(&(A.value1) + 1);
    cout << A.value2 << endl;
    return 0;
}
```

# Designing Classes

- The problem of defining appropriate classes for your project is generally non-trivial.
- If we need to define a class representing, for example, students, we should consider:
  - What shall we call it?
  - What are its attributes?
  - What methods are needed to implement behaviours according to their roles?
- We also need to consider what objects will be interacting with objects of the Student class, and what type of interaction is the most appropriate.
  - Are Student objects used by objects of other classes?
  - Are Student objects used by functions outside the class?

# The big heavy orange cat sat on the mat…

- **A scenario to think about:**
  - What objects are needed and what are their properties?
- **Objects?**
  - Cat, mat
- **Cat:**
  - Data?
    - Big → Size? → Width, Height, Depth?
    - Heavy → Mass
    - Orange → Colour (Ginger?)
  - Methods/Behaviours?
    - Measure Size→ Test for “bigness”
    - Measure Mass→ Test for “heaviness”
    - Show colour
    - Sitting on the mat (interaction)

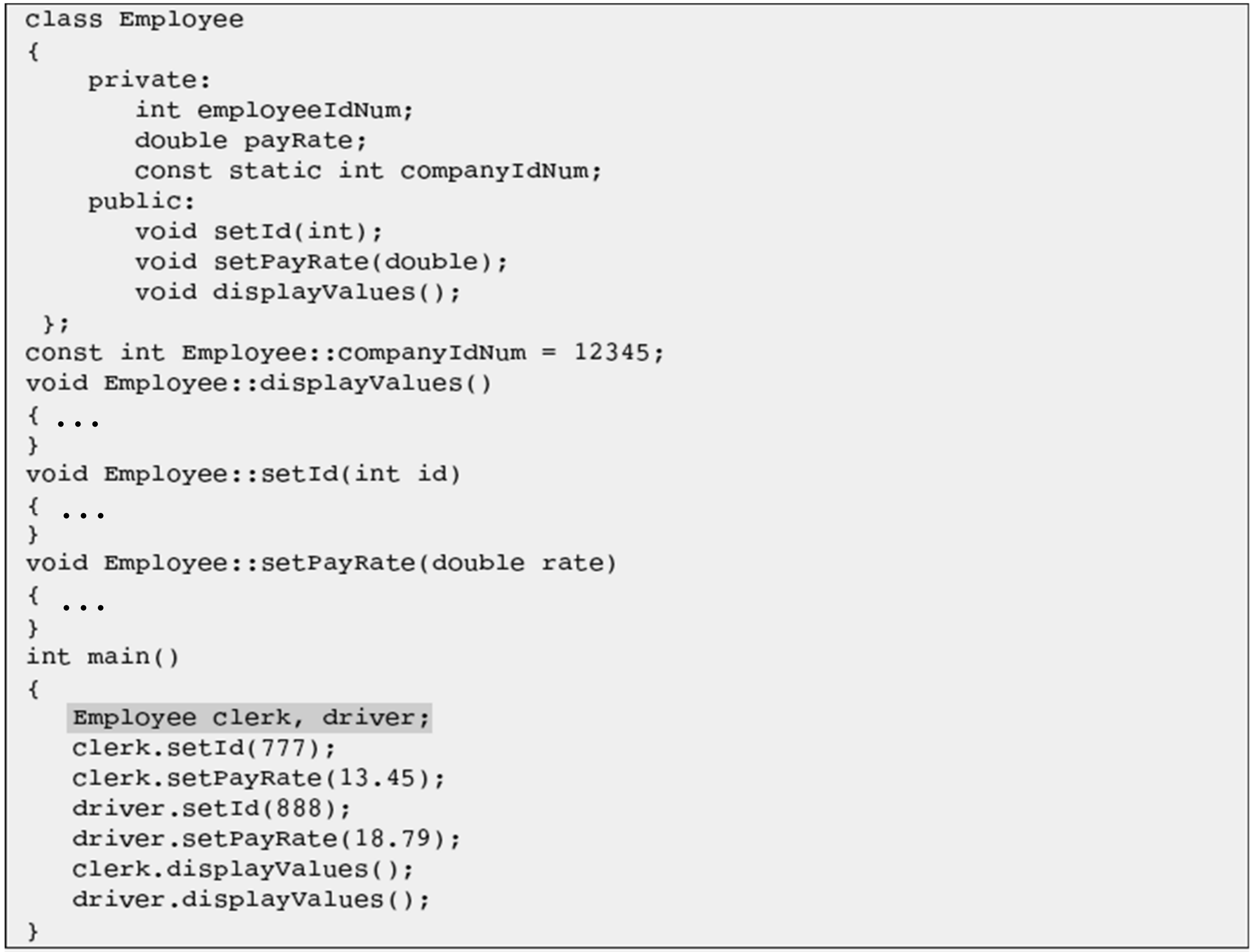
- **Mat:**
  - Behaviours?
    - Can be sat on
- **Classes**: Cat, Mat, … ?

# For class or object: Static…

- To avoid data replication some attributes are shared between instances, using static.
  - These are referred to in UML as having classifier scope.
- Class attributes/variables :
  - Define the data shared by class instances.
- Class methods :
  - Define behaviour of classes that cannot be associated with an instance.
- Instance attributes/variables :
  - Define the data stored by class instances. Instance scope.
- Instance methods :
  - Define behaviour of class instances. Instance scope.

# Static or not: The `this` Pointer

- We only have one copy of each function in a class.
- The this pointer holds the memory address of the current object that is using the function.

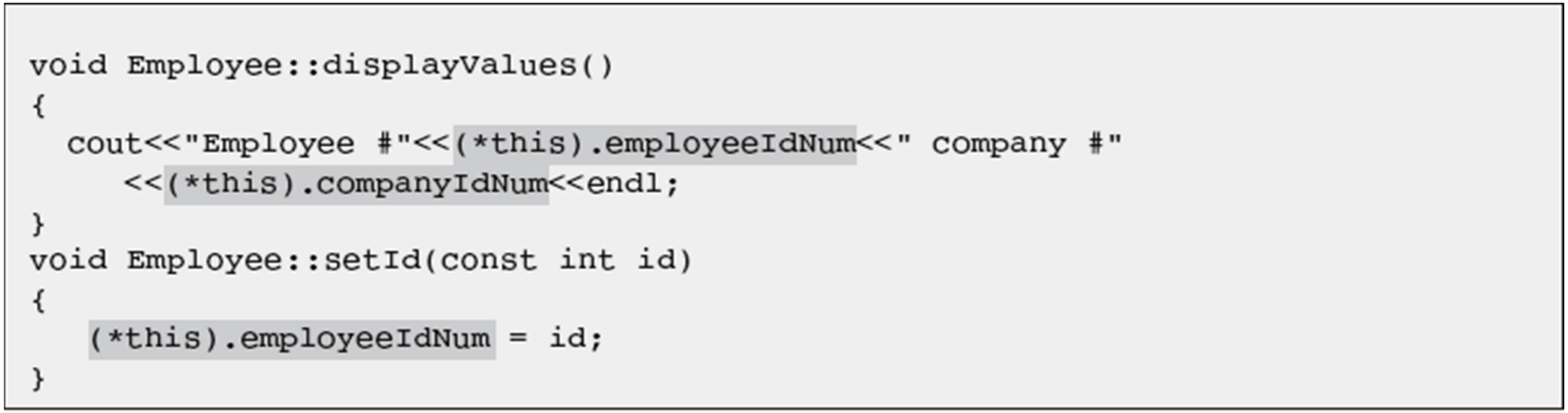


```
class Employee
{
    private:
       int employeeIdNum;
       double payRate;
       const static int companyIdNum;
    public:
       void setId(int);
       void setPayRate(double);
       void displayValues();
 };
const int Employee::companyIdNum = 12345;
void Employee::displayValues()
{ ...
}
void Employee::setId(int id)
{ ...
}
void Employee::setPayRate(double rate)
{ ...
}
int main()
{
   Employee clerk, driver;
   clerk.setId(777);
   clerk.setPayRate(13.45);
   driver.setId(888);
   driver.setPayRate(18.79);
   clerk.displayValues();
   driver.displayValues();
}
```

**Figure 7-19** `Employee` class and `main()` function that creates two `Employee` objects

- The `this` pointer is automatically supplied when you call a non-static member function of a class:
  - For example, `clerk.displayValues();`
    - … is actually `displayValues(&clerk);`
- The actual argument list used by the compiler for `displayValues()` is `displayValues(Employee *this).`

# Using the `this` Pointer Explicitly

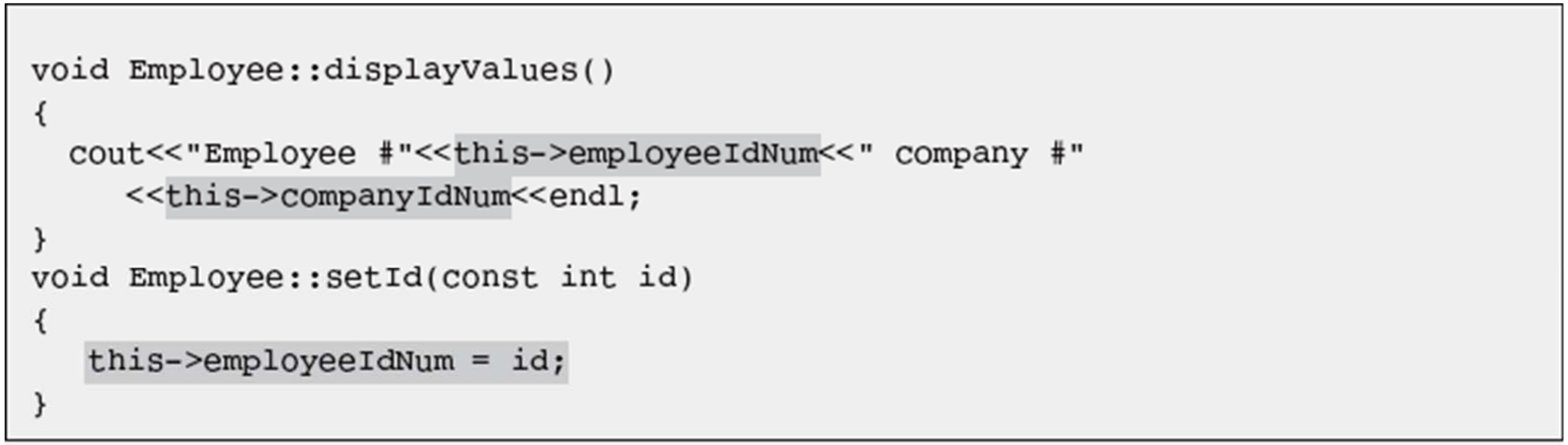


```
void Employee::displayValues()
{
  cout<<"Employee #"<<(*this).employeeIdNum<<" company #"
     <<(*this).companyIdNum<<endl;
}
void Employee::setId(const int id)
{
   (*this).employeeIdNum = id;
}
```

**Figure 7-22** Member functions explicitly using the `this` pointer

- Why do you need the bracket?
- Because `this` is a pointer, `this.employeeIdNum` isn’t and the “`.`” would take precedence over the `*`.

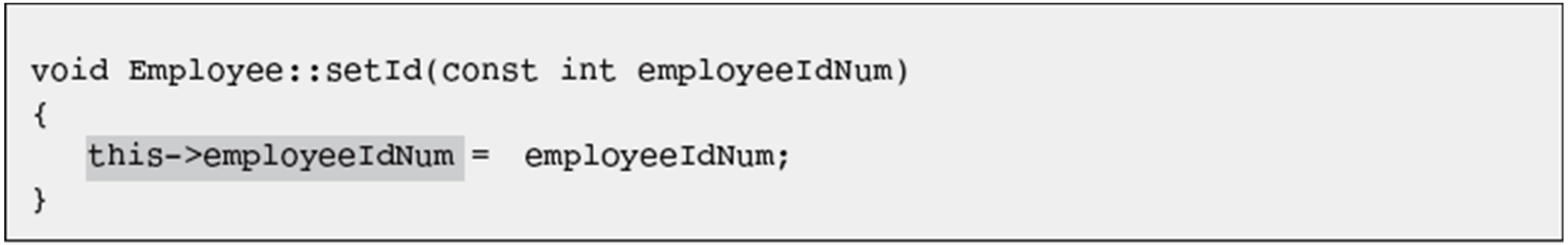
# Using the Pointer-to-Member Operator



```
void Employee::displayValues()
{
  cout<<"Employee #"<<this->employeeIdNum<<" company #"
     <<this->companyIdNum<<endl;
}
void Employee::setId(const int id)
{
   this->employeeIdNum = id;
}
```

Figure 7-23 Explicitly using the `this` pointer with the pointer-to-member operator

- Above is another way of accessing a field of a class.
- Below we use this to automatically identify context.



```
void Employee::setId(const int employeeIdNum)
{
   this->employeeIdNum =  employeeIdNum;
}
```

Figure 7-24 Using the `this` pointer to differentiate between a class field and a local variable with the same identifier

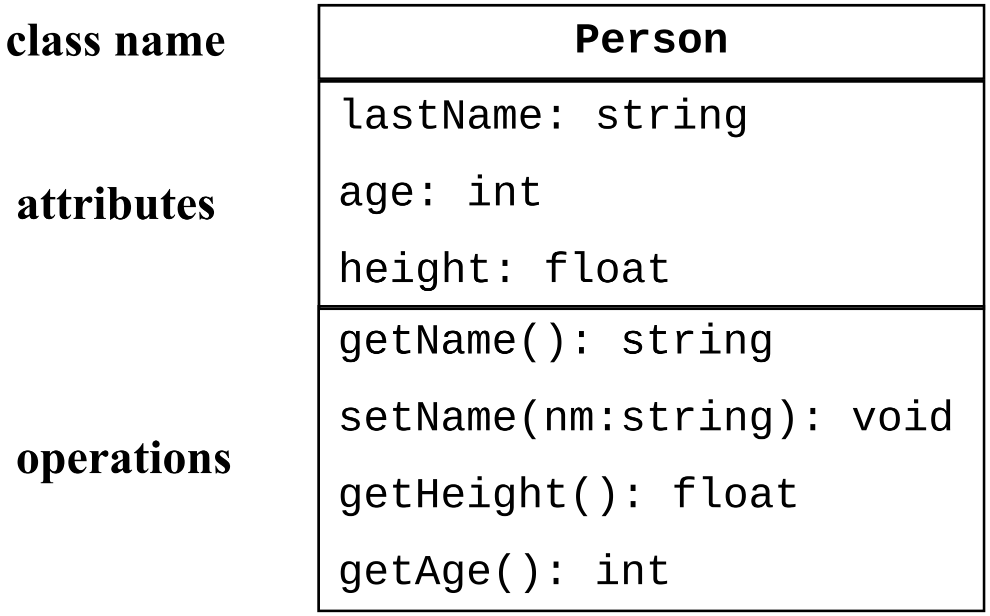
# What is the `->` operator?

- In the previous example we used the `->` operator (pointer-to-member operator) to access member functions of the entity at a particular address, as specified by a pointer.
- `(*this).number` and `this->number` are equivalent.
- In general for a pointer `X` we have `(*X).x` and `X->x`.

# Back to UML: Diagrams

- Class diagrams contain classes and relationships between them.
  - They specify attributes sufficient to describe *any* state of the system.
    - We need this for the state model.
  - They specify methods sufficient to implement system behaviour.
    - We need this for the interaction model.

- The most basic class symbol, so representation of a class, is a box with the class name.
  - It is used when class properties are not important, usually at the initial stages of system design.
- Adding in more details is likely later …
  - Attributes, or data members.
  - Operations, or methods.



**class name**

| Person |
|---|
| lastName: string<br>age: int<br>height: float |
| getName(): string<br>setName(nm:string): void<br>getHeight(): float<br>getAge(): int |

**attributes**

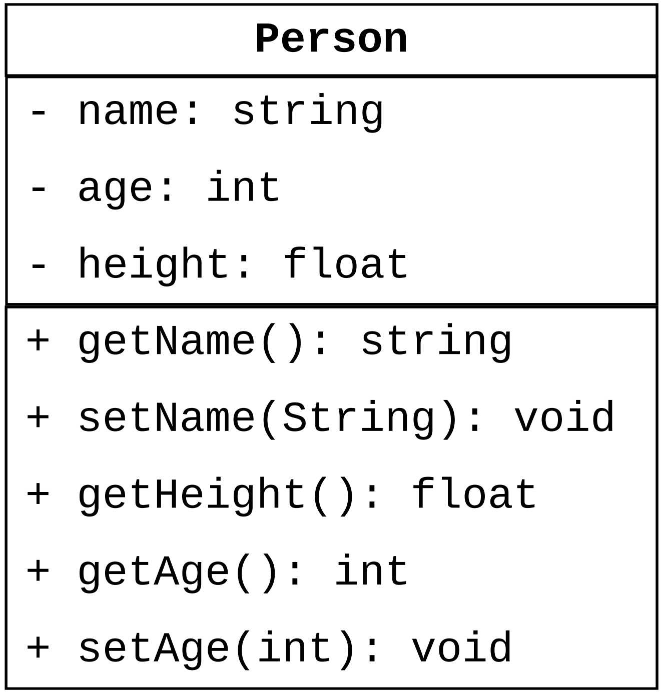
**operations**

C++

```
class Person{
            ...
};
```

- The level of detail depends on who the diagram is intended for.
- Class symbols, i.e. these boxes, are the building blocks of class diagrams.

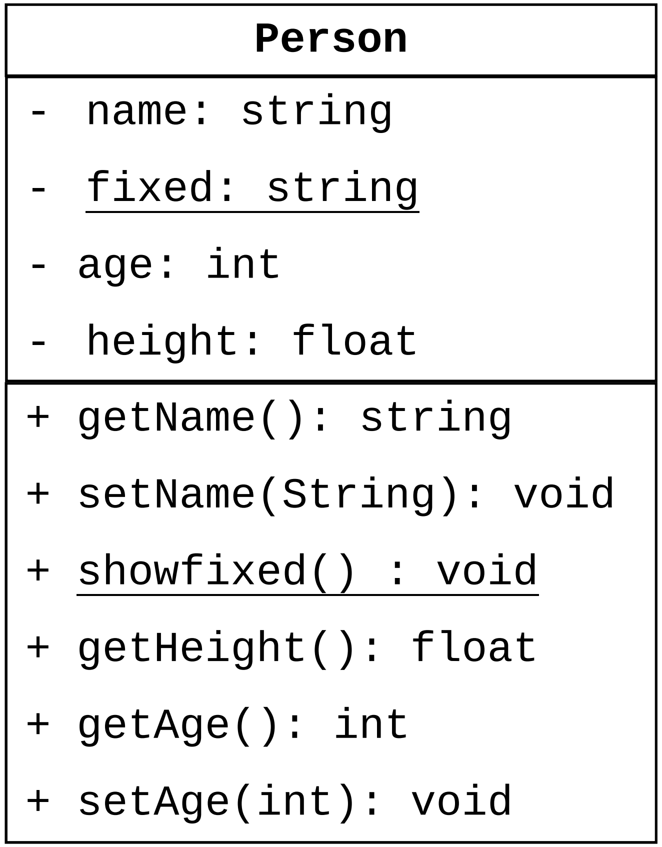
- We can include visibility prefixes corresponding to the access specifiers.



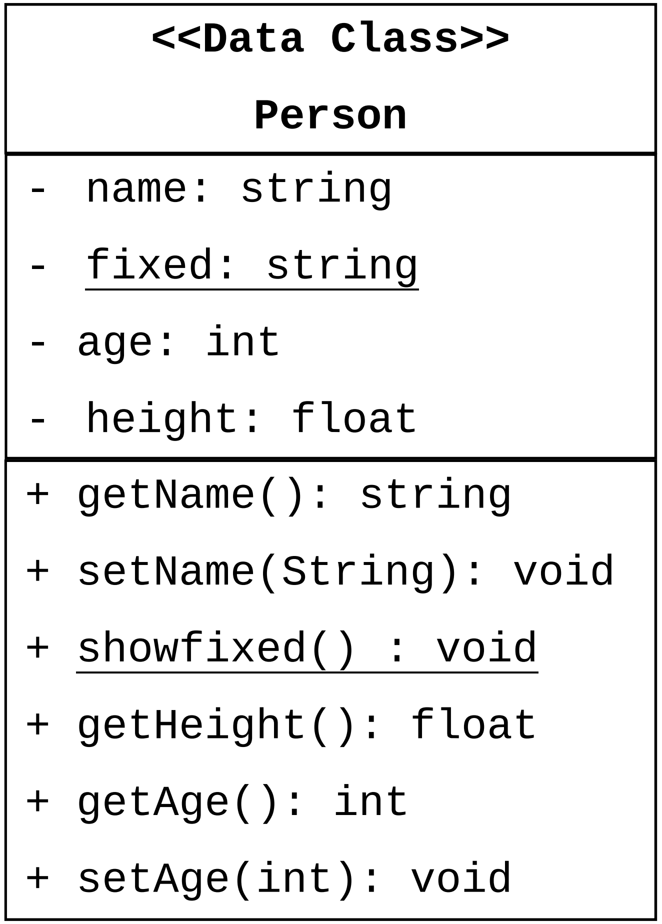
```
Person
- name: string
- age: int
- height: float
+ getName(): string
+ setName(String): void
+ getHeight(): float
+ getAge(): int
+ setAge(int): void
```

| | |
|---|---|
| '+' | public member |
| '-' | private member |
| '#' | protected member |
| '/' | derived member |
| '~' | package member |

- We can identify whether members have class/classifier scope using an underline.
- We can define stereotypes using `<<Stereotype>>`.
  - See http://www.uml-diagrams.org/class.html.

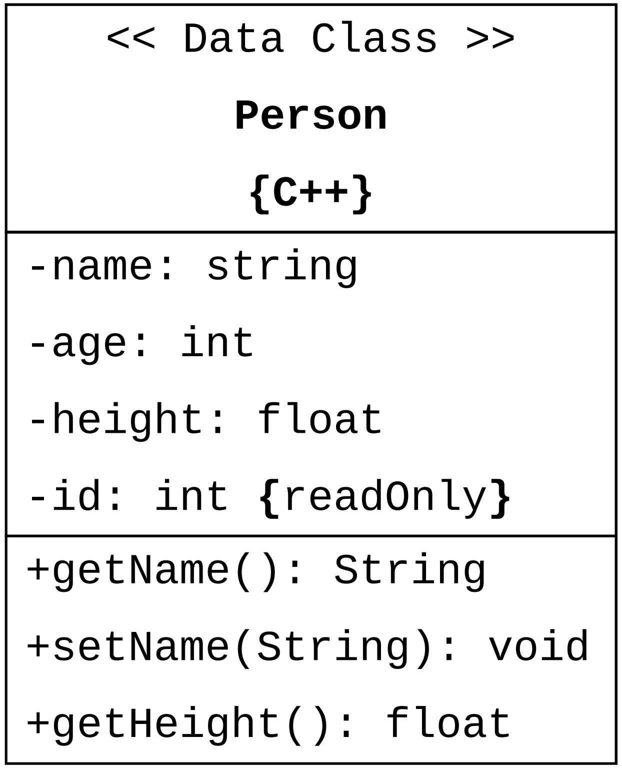


```
Person
-  name: string
-  fixed: string
- age: int
-  height: float
+ getName(): string
+ setName(String): void
+ showfixed() : void
+ getHeight(): float
+ getAge(): int
+ setAge(int): void
```



```
<<Data Class>>
Person
-  name: string
-  fixed: string
- age: int
-  height: float
+ getName(): string
+ setName(String): void
+ showfixed() : void
+ getHeight(): float
+ getAge(): int
+ setAge(int): void
```

- We can also represent constraints using { }, …
  - … such as the implementation language being C++.
  - … or an attribute being constant.



<< Data Class >>

**Person**

**{C++}**

-name: string

-age: int

-height: float

-id: int **{**readOnly**}**

+getName(): String

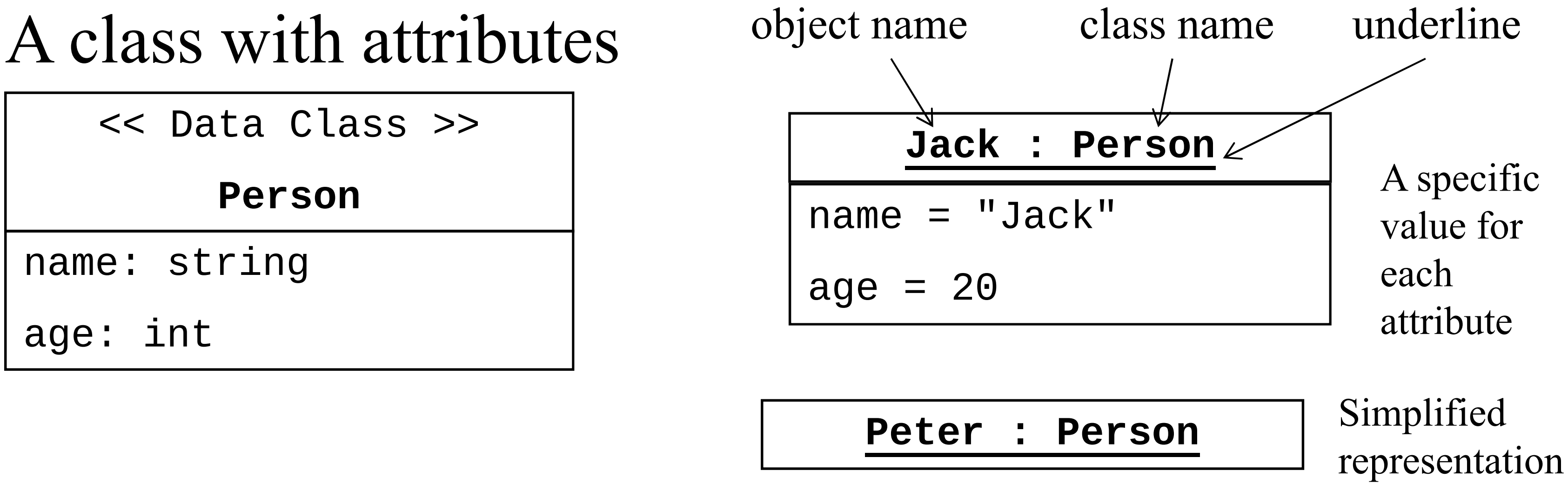
+setName(String): void

+getHeight(): float

```
class Person {
   private:
      string name;
      int age;
      float height;
      const int id = 25;
   public:
      string getName(){…}
      void setName(String){…}
      float getHeight(){…}
};
```

# Object Diagrams

- The building block is the object symbol, each based off a class symbol.



A class with attributes

<< Data Class >>

**Person**

name: string

age: int

object name

class name

underline

**Jack : Person**

name = "Jack"

age = 20

A specific value for each attribute

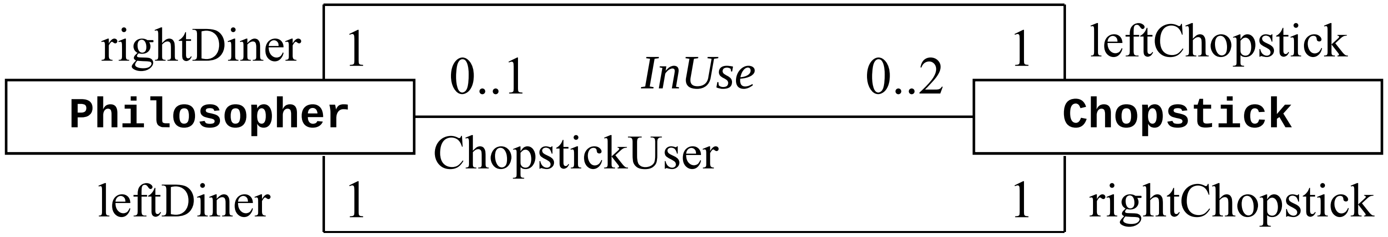
**Peter : Person**

Simplified representation

- Object diagrams can represent a 'snapshot' of the system at a given moment, …
- … although we may leave out the attributes to obtain a time-invariant object symbol.

- To emphasise the difference, we need to jump ahead a bit … for an example anyway.

Class Diagram



rightDiner

1

0..1

*InUse*

0..2

1

leftChopstick

**Philosopher**

**Chopstick**

ChopstickUser

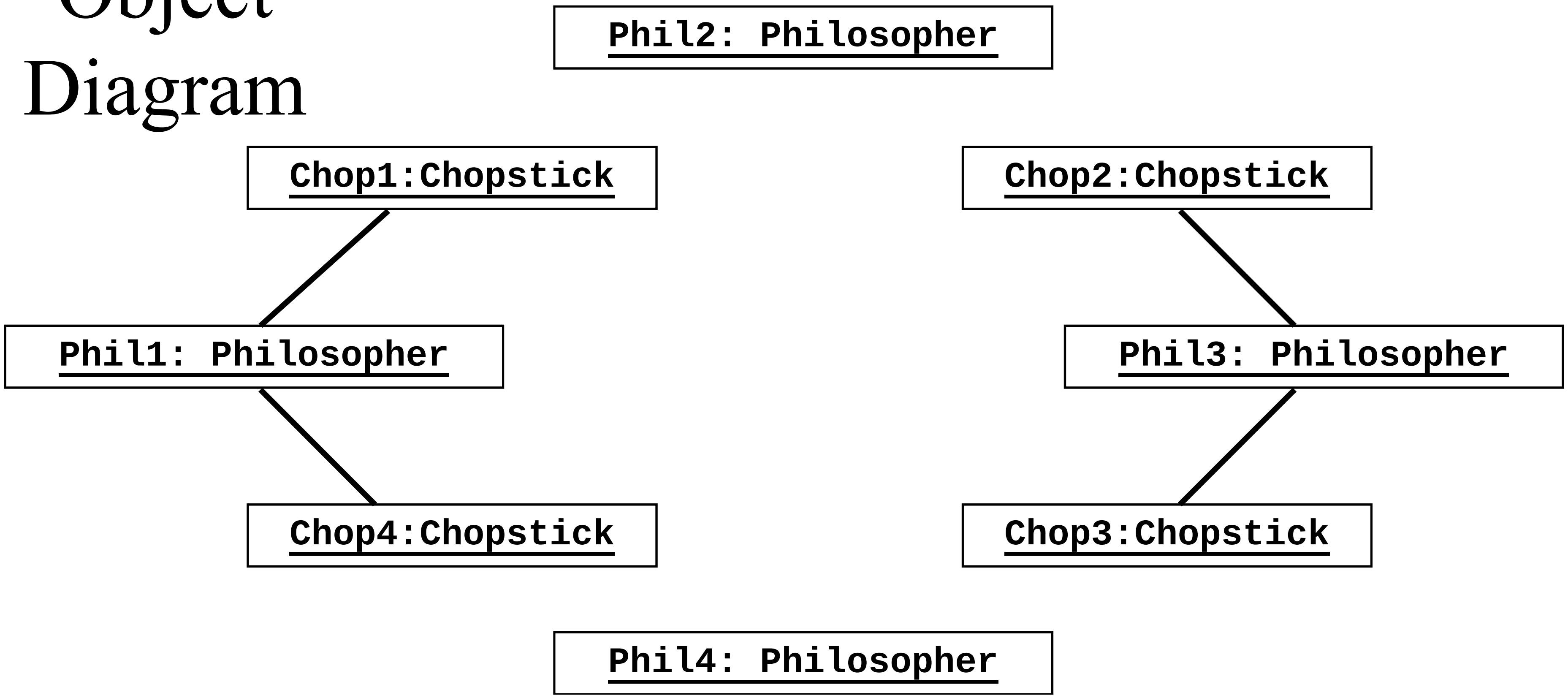
leftDiner

1

1

rightChopstick

Object Diagram



**Phil2: Philosopher**

**Chop1:Chopstick**

**Chop2:Chopstick**

**Phil1: Philosopher**

**Phil3: Philosopher**

**Chop4:Chopstick**

**Chop3:Chopstick**

**Phil4: Philosopher**

# `get()` and `set()` ?

- Why didn’t we include get and set member functions two slides back?
  - The stereotype `<<Data Class>>` should include these methods.
  - We often list only the “more important” member functions and data elements.
- We can omit those identifiers which have no significance in the view of the system we are modelling.

# Get → `const` functions

- Typically get functions return something without making changes to any values.
- If that is the case the function can be `const`.
- This is an instruction to the compiler to say our intent is that nothing is changed.
- The syntax is, for example, …

```
int function() const;
```

- If you declare a function to be `const`, it can be called on any type of object.
- A function that is non-`const` can only be called by non-`const` objects.
- Here goes a complete example of a `const` function in the context of get for a class.

```
class Test
{
private:
        int x;



public:
        int getx() const;
};
```

```
int Test::getx() const
{
//      x=5;
        return x;
}

int main()
{
        Test thing;
        cout << thing.getx() << endl;
}
```

# Continuing …

- We will look more at class diagrams more later, once we see class relationships.
- For now we will return to C++ stand alone classes and delve into the organisation of them some more.