THE UNIVERSITY OF WARWICK

C++ Classes

Tom Latham

(based on material from Matt Williams & Ben Morgan)

Introduction

- We've seen that a `struct` can be used to bundle data together into a single object
- However, as well being able to store state, it would be even more useful if objects can do things
- We therefore need something that can both:
  - Organise data
  - Present services using this data to the user
- We're going to use the idea of an Employee at a company as our illustrative case

Object instances

- As you might have guessed, the 'thing' we need is an ***Object***, or more precisely, an ***Object Instance***
- What is an Object Instance?
  - "A *self contained meaningful software agent*"
- An instance provides a ***well defined, related set of services***
- An instance has a ***persistent, personal state***
  - It owns a set of variables
  - Its services all access this state
- An instance ***exhibits identity***
  - The state of an instance can change, but it's still the same instance
    - `int a{3}; a++;`
  - Several instances can possess the same state, but they are distinct instances
    - `int a{3}; int b{3};`

Where do objects come from?

- Objects are the latest evolutionary stage of software 'chunks'
- A chunk is simply a piece of code that represents data or operates on data, e.g.

Data:
- Symbolic word (assembler)
- Variables, e.g. int a;
- Data structures

Operations:
- Opcode (assembler)
- Statements, e.g. a=b*c+d
- Functions e.g. double force(double m, double a);

- We want chunks to be
  - ***Powerful***: provide more and better services
  - ***Cohesive***: perform their task and nothing more (simplicity)
  - ***Loosely coupled***: changes do not impact clients of the chunk

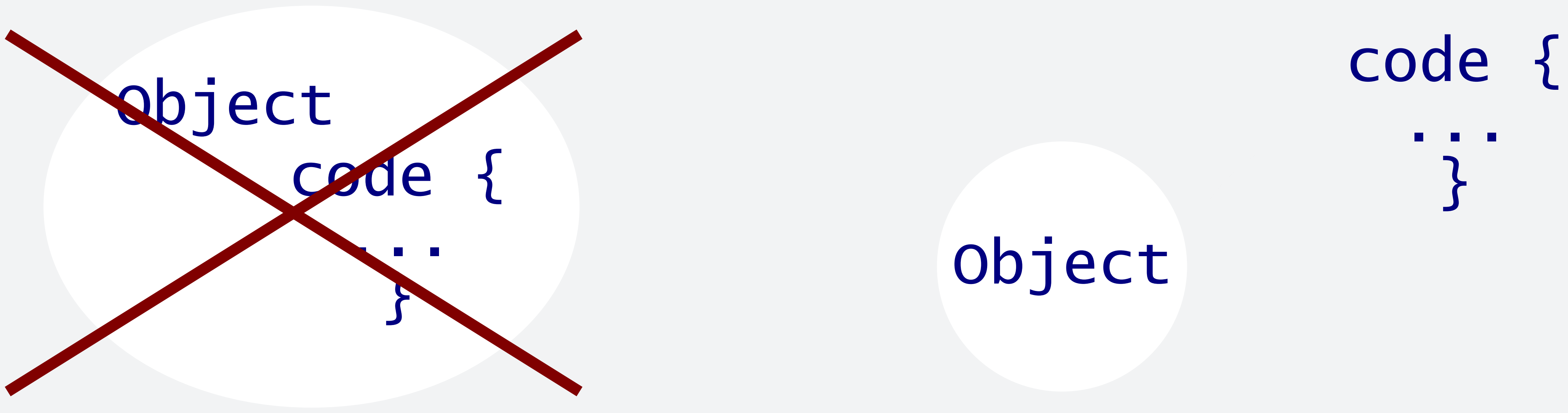
Inversion

- Traditionally, our chunks are data structures and functions
    - Functions interact by calling each other and passing data
    - Functions must manage and pass the data they require
- In object orientation these responsibilities are ***inverted***
- Now our chunks are ***object instances*** that
    - Encapsulate pieces of related data
    - Provide services that can operate on this data
- Object instances interact with each other by requesting services: ***messaging***
    - Objects are responsible for managing both the data and the code required to handle the messages they might receive
- Can be thought of as the data being in charge – ***the data knows how to process itself***!

More on object instances

- Instances have access to code blocks they can use to provide their services

- It's important to note that instances don't carry around their own copies of the service code:



- Service code blocks are managed for object instances by ***classes***

Classes

- structs and classes are very similar behind the scenes but are used to implement different concepts:
- A `struct` should be used when you simply want to bundle together some related data (e.g. our command-line arguments)
- A `class` should be used when the data inside are correlated and you want to provide an interface to that e.g. a std::vector keeps track of the elements themselves, the number of elements, its current capacity, etc.

Defining a class

- A `class` is defined in a similar way to a `struct`

```
//Employee.hpp
class Employee {
 public:
  std::string name; ///< The name of the Employee
  std::string niNumber; ///< The NI number of the Employee
  int salary; ///< Total salary of the Employee in pounds
};
```

- NB the `public` access specifier
- By default a `class`'s members are hidden to the outside – more on this later…
- For now, you need to explicitly make them available

Constructors

- A constructor is a special function that is used to setup the initial state of an object when it is created
- It is named the same as the class name
- It has no return type
- It is declared inside the class

```
class Employee {
 public:
  Employee(const std::string& empName, const std::string& empNI);

  std::string name; ///< The name of the Employee
  std::string niNumber; ///< The NI number of the Employee
  int salary; ///< Salary of the Employee in pounds
};
```

Constructors

- A constructor is defined like any other function
- Put the definition in the .cpp file
- You need to specify the scope (i.e. that we're implementing the function called Employee within the Employee class) using the scope-resolution operator ::
- The function can access the members of the class directly:

```
Employee::Employee(const std::string& empName, const std::string& empNI)
{
  name = empName;
  niNumber = empNI;
  salary = 0;
}
```

- This way of initialising the member variables is equivalent to doing:

```
int salary; //Declare
salary = 0; //Initialise
```

Constructors

- As we've said before, it is always safer (and sometimes necessary) to initialise variables at definition

```
Employee::Employee(const std::string& empName, const std::string& empNI)
 : name{empName}, niNumber{empNI}, salary{0}
{
}
```

- Which is more like:

```
std::string name {empName}; //Declare and initialise
std::string niNumber {empNI}; //Declare and initialise
int salary {0}; //Declare and initialise
```

- Data members declared as `const` *must* be initialised this way

Default values for members

- You can set default values of data members directly in the class declaration

```cpp
class Employee {
 public:
  Employee(const std::string& empName, const std::string& empNI);

  std::string name; ///< The name of the Employee
  std::string niNumber; ///< The NI number of the Employee
  int salary {0}; ///< Salary of the Employee in pounds
};
```

- This defines the default value used for initialisation in case one is not given in the constructor

Constructing an object

- The std::string and std::vector objects that we've already been using are defined as classes

- So, we can use a similar syntax as we used to construct those:

```cpp
#include "Employee.hpp"

int main() {
  //Calls the contructor we declared
  Employee jane {"Jane", "BG123456A"};
  std::cout << jane.name << "\t" << jane.niNumber  << "\n";
}
```

Aside: naming conventions

- Most software projects have naming conventions
- In this course we use:
- UpperCamelCase for class and struct names
- lowerCamelCase for function and variable names
    - NB that constructors are an exception since they must have exactly the same name as the class
- lowerCamelCaseWithTrailingUnderscore_ for data members
- This makes it easier to see, at a glance, the context of a name in the source code

Exercise 2: skeleton of a CaesarCipher class

1. Write the declaration of a CaesarCipher class
   - This should go in a new header file src/MPAGSCipher/CaesarCipher.hpp
   - The key should be the only member variable (for now)
   - The constructor should take the key as its single argument
2. Implement the constructor in the file src/MPAGSCipher/CaesarCipher.cpp

Multiple Constructors

- A class can have more than one Constructor
- For example, we might want to be able to specify the starting salary of our Employee or perhaps we don't yet know their NI number so just want to provide the name:

```
class Employee {
 public:
  explicit Employee(const std::string& name);

  Employee(const std::string& name, const std::string& niNumber);

  Employee(const std::string& name, const std::string& niNumber,
           int startingSalary);
```

explicit Constructors

- In the previous example we had the `explicit` keyword in front of the first constructor:

```
explicit Employee(const std::string& name);
```

- This is because this constructor takes only a single argument and such single-argument constructors can, in general, be used for implicitly converting between different types
- For example, without the `explicit` keyword, code like the following would compile without warning:

```
void addToTeam(const Employee& emp);

// in some other function
{
  std::string name{"Jane"};
  addToTeam(name);
```

- So, a temporary Employee object would be created and added to the team, which is probably not what was intended here
- Using the explicit keyword indicates that we don't want this constructor to be used in such a way

Exercise 3: a 2nd constructor for CaesarCipher

1. Add a second constructor to your CaesarCipher class that takes the key as a string

   - You can move the code that does the conversion from a string to an unsigned integer from your main function into this constructor
     - Make sure to first set a default value of the key_ member variable
     - Then, if the checks are successful, you can overwrite it with the supplied value

   - Your main function can then be simplified to use this new constructor instead

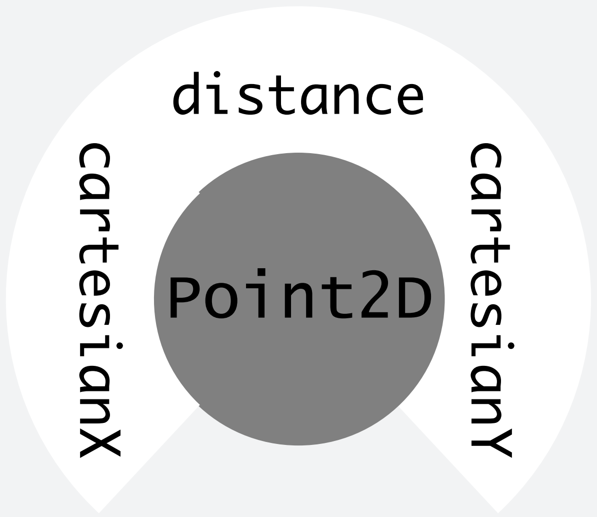
2. Make sure that both of your single-argument constructors are declared explicit

Encapsulation

- Encapsulation is an important concept in OO and is the separation of the interface to something from its underlying implementation
  - Interface: how we ***access*** something, e.g. call to a function
  - Implementation: how that thing ***performs the task***, e.g. the actual code in the function
- This concept is also referred to as '***programming by contract***'
  - You don't care ***how*** a service is provided, just that it is provided
- The “stuff” encapsulated by an object is its knowledge of how to be itself
  - It knows how its services are provided
  - It knows how to store its state
  - It encapsulates this knowledge inside a services interface

Encapsulation Example 1

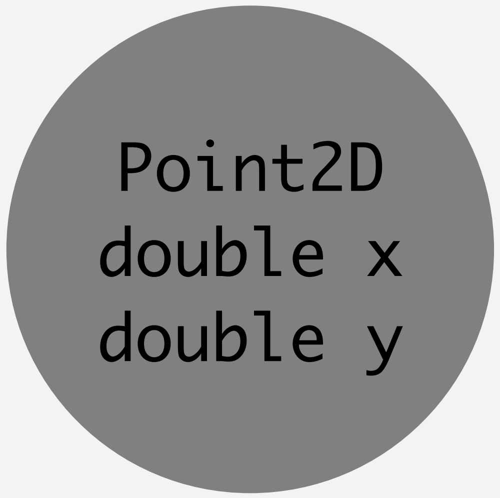
- Consider an object to represent a point in 2D space, lets call it Point2D
- What services might it provide?
  - find cartesian x coordinate
  - find cartesian y coordinate
  - find distance from origin
- So we could diagrammatically represent an instance as:

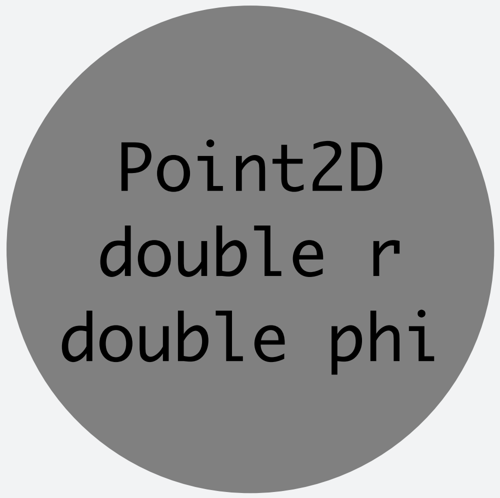


Message Point2D instance to obtain its cartesian y coordinate.

Encapsulation Example 2

- The important thing to note is that we've said ***nothing*** about ***how*** the Point2D instance will obtain, say, the distance
- That's fine – we the client ***do not and should not care***
- Point2D's state could be represented by two double variables using cartesian ***or*** polar coordinates:





- The implementation of, e.g., the distance service will be different in each case, but the client ***will not notice***

Information Hiding

- Encapsulation differentiates the 'outside' of an object (its service interface) from the 'inside' (its state)
- OO languages provide syntax to express this inside/outside structure, and can check the security of this boundary
- By doing this, we can hide the state information inside from clients of the object – why hide this?
    - Client has no need to know what code object uses or how it represents its state, only care that a service can be messaged
    - Client is not impacted by nature of the code used by the object or its variables
    - Client will not be impacted by changes to code or variables
- Encapsulating data and hiding it behind a service interface, making the object a 'black box', helps to:
    - Increase cohesion (information doesn't 'leak' elsewhere)
    - Decrease coupling (only link is messaging services)

Encapsulation & information hiding

- For example, the name of our Employee is currently stored as a single std::string
- However, in the future we may decide instead to store it as a pair of strings for first and last name
- Making the data member private and only allowing use/manipulation of it via services allows the details of the implementation to be hidden

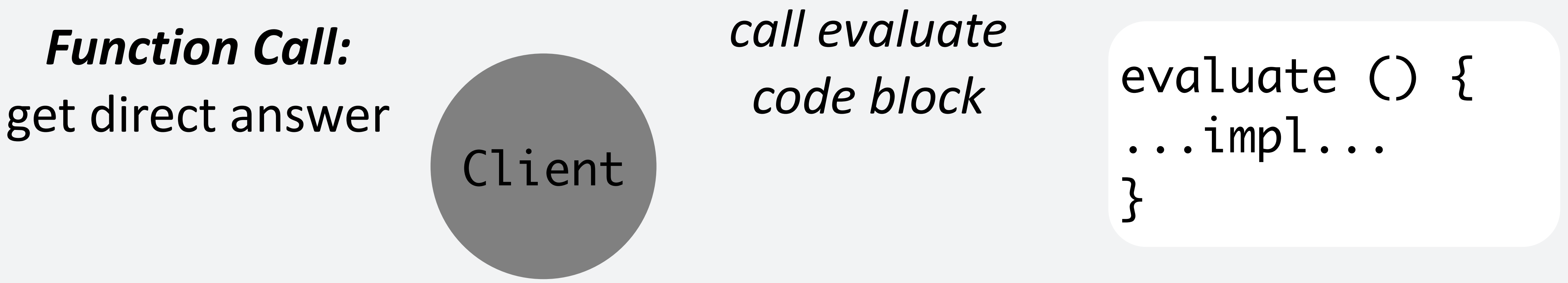
```
class Employee {
 public:
  void setName(const std::string& newName);
  std::string name() const;
 private:
  std::string name_; ///< The name of the Employee
};
```

Messaging 1

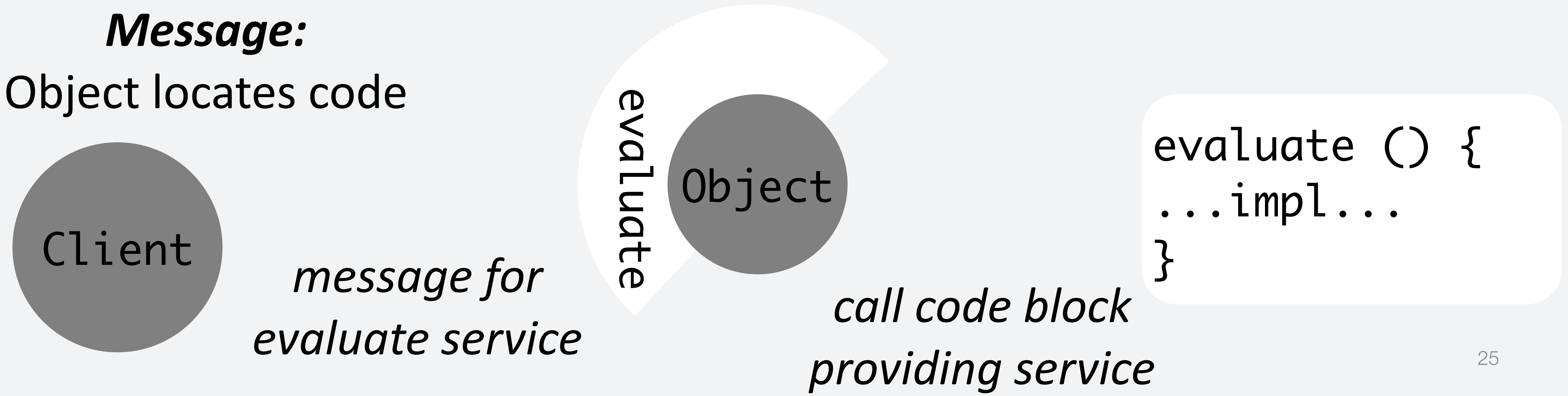
- Objects hide their implementation behind a service interface – we ask 'what can you do?' rather than 'how do you do it?'
  - e.g. you've used vectors, you don't know how they are implemented, but you know you can add elements to them, access elements, etc.
- There is a slight subtlety here, as inversion, encapsulation and information hiding could not happen if the client were directly calling the service code
- OO changes from instructions (***function calls***) to requests (***messages***)
  - e.g. if we message an object for its service 'evaluate' we are not directly calling a function 'evaluate'
- There's an extra layer involved behind the service interface

Messaging 2

- In messaging, the client no longer directly selects the code to be run
- You message an object to provide a service, it decides which code runs
- It's important to understand this subtle distinction between calls and requests – and it's probably easiest to see this diagrammatically:



```
evaluate () {
...impl...
}
```



```
evaluate () {
...impl...
}
```

Identity and Encapsulation

- Recall that earlier we discussed objects exhibiting ***identity***
- Object instances have identity distinct from their state:
  - An object can change state yet still be the same instance
    - e.g. we message a StraightLine object requesting it to change its slope to a supplied value
  - Two objects can possess the same state, but be distinct instances
    - e.g. two StraightLine objects have the same slope and intercept
- Identity is essential for encapsulation to work
  - We must be able to identify a particular object instance without knowing its state

Programming by contract – client point of view

- I want you to provide a *service*
- I don't care *how* you do it
    - The answer is the important thing, not how it was arrived at
- I don't really care if *you* do it
    - The task (or parts of it) can be delegated
- I don't really care *what you are*
    - As long as the server does what you want you don't care about what other things it might do or indeed anything else about it

Programming by contract – server point of view

- I don't know who is messaging or what they will do with the information
  - There is no need to know
- I know one way of functioning
- I don't know who I will be
  - You will be an object instance but you don't know which one, that doesn't matter
- It's up to me how I provide my service

Type

- An object's type is the set of message signatures that it will accept
- Weak typing – only checked at run-time, the errors go to the user rather than the programmer
- Strong typing – the compiler checks that objects are capable of acting in their intended role
    - C++ is an example of a strongly typed language

Classes

- Classes are sources of object instances
- They provide a single definition of their instances
  - The variables (data) – each instance has its own copy of the data that it carries around with it
  - The mechanisms (code) – each instance does not carry a copy of the code around but has access to it

Accessor functions

- In some cases you may want functions that simply get or set the value of a particular data member (so-called 'getters' and 'setters')

- Such functions are generally rather simple:

```cpp
void Employee::setName(const std::string& newName)
{
  name_ = newName;
}

std::string Employee::name() const
{
  return name_;
}
```

- But they can be more complex:

```cpp
std::string Employee::name() const
{
  return firstName_ + " " + lastName_; //for example...
}
```

Member functions

- Member functions can do anything, not just get and set
    - Indeed, a class that consists purely of getters and setters really ought to be a simple struct!
- Classes can perform more complex (inter)actions:

```cpp
class Employee {
 public:
 void promoteToGrade(const Grade& newGrade);
  //...
};

void Employee::promoteToGrade(const Grade& newGrade)
{
  title_ = newGrade.title();
  salary_ = newGrade.startSalary();
}
```

const functions

- You may have noticed that the name() function has a const label at the end
- This means that nothing inside that function can change the object it is acting on

```
std::string Employee::name() const
{
  name_ = "Fran"; //Compiler error
  return name_;
}
```

- It also means you can call it on a const object

```
const Employee bill {"Bill"};
std::cout << bill.name(); //This is fine
bill.setName("John"); //Compiler error
```

- Member functions should be made const unless explicitly needed otherwise

Exercise 4: adding functionality

- Add a member function, called applyCipher, to your CaesarCipher class that encrypts or decrypts a string and returns the resulting string
- You can use the runCaesarCipher function that we implemented last time (you can find it in src/MPAGSCipher/RunCaesarCipher.cpp) as the basis for this but with a few adaptations:
    - You should use the key_ data member as the key
    - You can also make the alphabet another data member
    - The two data members should now be made private
- In your main function, create an instance of this class and then call its applyCipher function instead of using the runCaesarCipher function

Enumerations

- Enumerations (`enum`) provide a way of defining a set of named values
- They are useful when there is a small finite set of possible values and you want to perform different actions depending on the value

Enumerations

- Declaring an enum is similar to declaring a class

```
/// The rank of the employee
enum class Rank {
  Junior, ///< A new person at the company
  Senior, ///< Someone whos been here a while
  Chief   ///< Someone super special
};
```

- It makes a new type, which can be instantiated:

```
Rank personsRank {Rank::Senior};

if (personsRank == Rank::Senior) {
  std::cout << "Senior" << std::endl;
}
```

Enums and switches

- Enums are very useful when mixed with switch statements

```
Rank personsRank {Rank::Senior};

switch(personsRank) {
  case Rank::Junior:
    return 0;
  case Rank::Senior:
    return 3000;
}
```

- The compiler will warn you that you missed one of the entries in the enum (Rank::Chief)

Enum conversions

- Enumerations are represented by integers behind the scenes
- However, C++11 introduced strong typing of enumerations, meaning you can't convert freely between them:

```
enum class Colour {
  Red,
  Blue,
  Green
};

Colour c {Colour::Red};
c = Rank::Senior; //COMPILER ERROR
                  //Rank::Senior is '1', doesn't set 'c' to 'Blue'

if(c == Rank::Junior) { //COMPILER ERROR
  std::cout << "This doesn't make sense" << std::endl;
}
```

Naming convention for enumerations

- In this course we use UpperCamelCase for both the *type* and the *states* of enumerations
- Look back at the previous few slides to see what this looks like in action
- Please stick to this and the other naming conventions as you write your code

Exercise 5: enumerate the cipher mode

- Add an enumeration called CipherMode to designate the encryption mode (Encrypt or Decrypt)
  - This declaration should go in a new file: src/MPAGSCipher/CipherMode.hpp
  - It only needs the two states
- Use this enumeration instead of the 'encrypt' boolean flag in the rest of your code