Module 2:
User defined Data Types
&
Operations upon them

In this module we will cover

- **Inadequacies of in-built data types**
- **User defined data types (classes and objects)**
- **Operations on objects (methods)**
- **Public and Private members of an object**
- **The first steps to data encapsulation**

Aims of the module

In this module we show that the simple in-built data types are not generally convenient for representing the types of entity you find in even a simple programming situation. In general you need several integers, floats, characters ...etc.. to do this in raw form would lead to excessively messy and cumbersome code.

This leads us to introduce the idea of user defined data types which contain variables for all the attributes of an entity. In C++ this means the use of **`classes`**

We introduce the word **"object"** to mean **"instance of a class"**

We then show you how you can manipulate the member variables of such objects.

However we then convince you that actually manipulating the member variables of such objects is bad news. Not only is it still cumbersome, it is fundamentally dangerous as it means the innerds of the class can then never be changed without upsetting users.

To avoid this we introduce a set of functions to separate the user from the implementer. We call these "Methods of the class". These Methods are the only things allowed to access the variables in the class.

Finally we consolidate the ideas we have introduced by taking an abstract look at Data Encapsulation

2.1 Inadequacies of in-built data types

We are going to start this module with an exercise in order to make some points

Student exercise

- Modify the code you wrote earlier to find the angle between two vectors.

-This time encapsulate the calculation of:

(a) the magnitude of a a vector
(b) the dot product of two vectors

inside two functions.

- Use these functions to simplify the code which performs the calculation of the angle between the vectors.

The points which it is intended to illustrate are:

- **The vectors cannot be handled as a single item -they have to be passed around as three disconnected numbers. This is cumbersome.**

- **Functions are becoming silly. The dot product already needs to have 6 arguments !**

- **What would we do if we wanted to add a "magnitude" number to the representation of a vector?. We would have to go and edit every function to have 4 arguments per vector instead of 3!**

  ⇒ completely un-maintainable code

What would we do for an "entity" which was more complex than a vector ?

Here is a good, more abstract, example:

`BankAccount`

How would we represent this of we were writing some financial software ?

Entity type: BankAccount

Attributes:
holdersName
currentBalance
overdraftLimit
jointAccount
yearsHeld
...

Well, we could just write a lot of simple data types:

...but this has all the problems we have just illustrated.

```
// One (bad) realisation of a BankAccount


string holdersName ;
float  currentBalance ;
float  overdraftLimit ;
int    jointAccount ;
int    yearsHeld ;
```

Note: I pulled a fast one here with

string holdersName ;

This is a much easier way to handle strings of characters than using the in-built

char

variable.

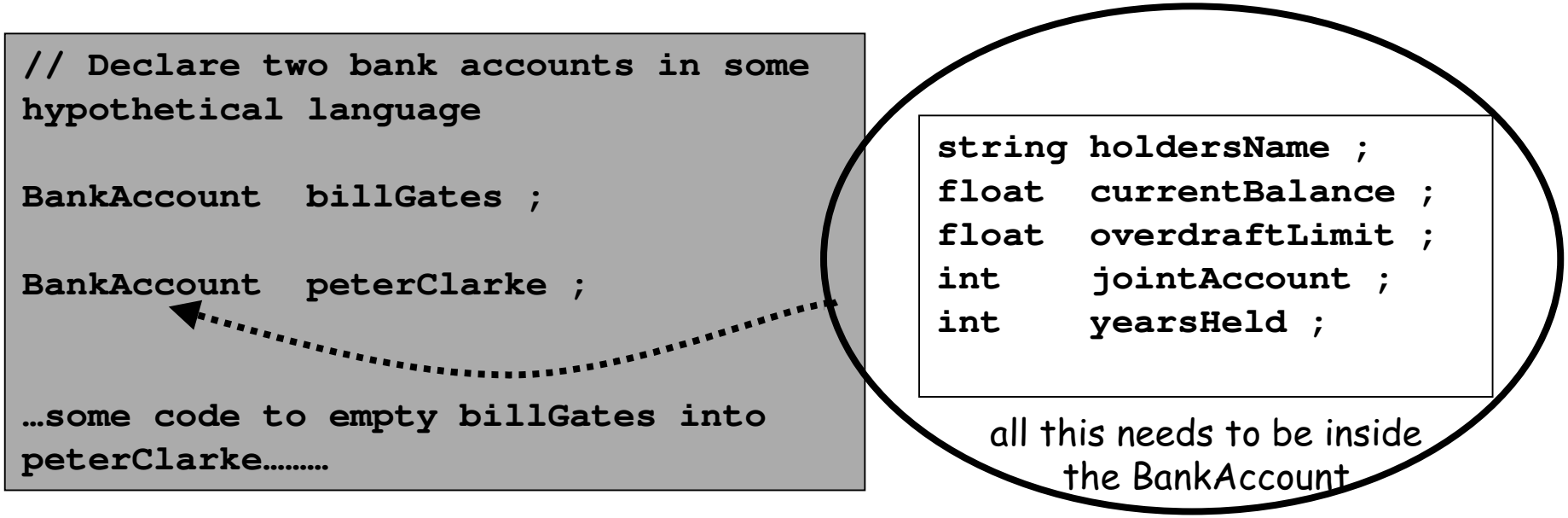
More on this later

```
// One (bad) realisation of a BankAccount

string holdersName ;
float  currentBalance ;
float  overdraftLimit ;
int    jointAccount ;
int    yearsHeld ;
```

The problem is that we have lots of separate bits of information which just happen to be written next to each other in the file.

It is clear that in any sensible language you need a way of referring to the whole collection with a single variable name.

What you really need is a variable of type : `BankAccount`



2.2 User defined Data Types (classes)

In any sensible language you can define

"User Defined Data Types"

to represent the

"Entities"

in your problem

Entity type: BankAccount
Attributes: holdersName currentBalance overdraftLimit jointAccount yearsHeld ...

Loosely:

"Entity" is the abstract notion of something in your problem

"Data Type" is a representation of this in some system (here a language)

Here is how:

```
// C++ code to define a BankAccount class


class BankAccount {
  public:
    string holdersName ;
    float  currentBalance ;
    float  overdraftLimit ;
    int    jointAccount ;
    int    yearsHeld ;

} ;
```

This is called a "class"

The variables inside it are called "class member variables"

If you include this definition in your program, you can now use **`BankAccount`** as if it were an in-built data type

Aside: note the keyword **public:**

```
// C++ code to define a BankAccount class


class BankAccount {
  public:
    string holdersName ;
    float  currentBalance ;
    float  overdraftLimit ;
    int    jointAccount ;
    int    yearsHeld ;

} ;
```

This means that all of the variables following it will be "publicly available" for any other bit of code to use.

At this point this wont mean much to you. Just ignore it for now. All will become clear very shortly.

Here is an example of how you declare some instances of the BankAccount class, and access the "members variables"

This is how you make two instances of the BankAccount class

Note the use of the . to refer to a "member" of a BankAccount

This is how you manipulate members of a BankAccount

```
// Declare two accounts

BankAccount billGates ;
BankAccount peteClarke ;


// Print out my balance

float balance ;

balance = peteClarke.currentBalance ;

std::cout << "They don't pay me enough" << balance ;


// Do a transfer

peteClarke.currentBalance = billGates.currentBalance;

billGates.currentBalance   = 0 ;
```

Here is an example of how you would pass a "BankAccount" to a function

Note that you treat the BankAccount just like you would an int or float

```
// Declare a bank account

BankAccount peteClarke ;

// Use a funtion to make a deposit in it

float amount = 500 ;

makeDeposit( peteClarke, amount ) ;

....
```

The arguments are declared just as normal

... and here is the function itself:

```
// function to  make a deposit in a BankAccount

makeDeposit( BankAccount accountToUse, float amt )
{
    accountToUse.currentBalance += amt ;
    return ;
}
```

Student exercise

-Write a class to represent the "3-vector" we have used in previous examples

- Call it ThreeVector

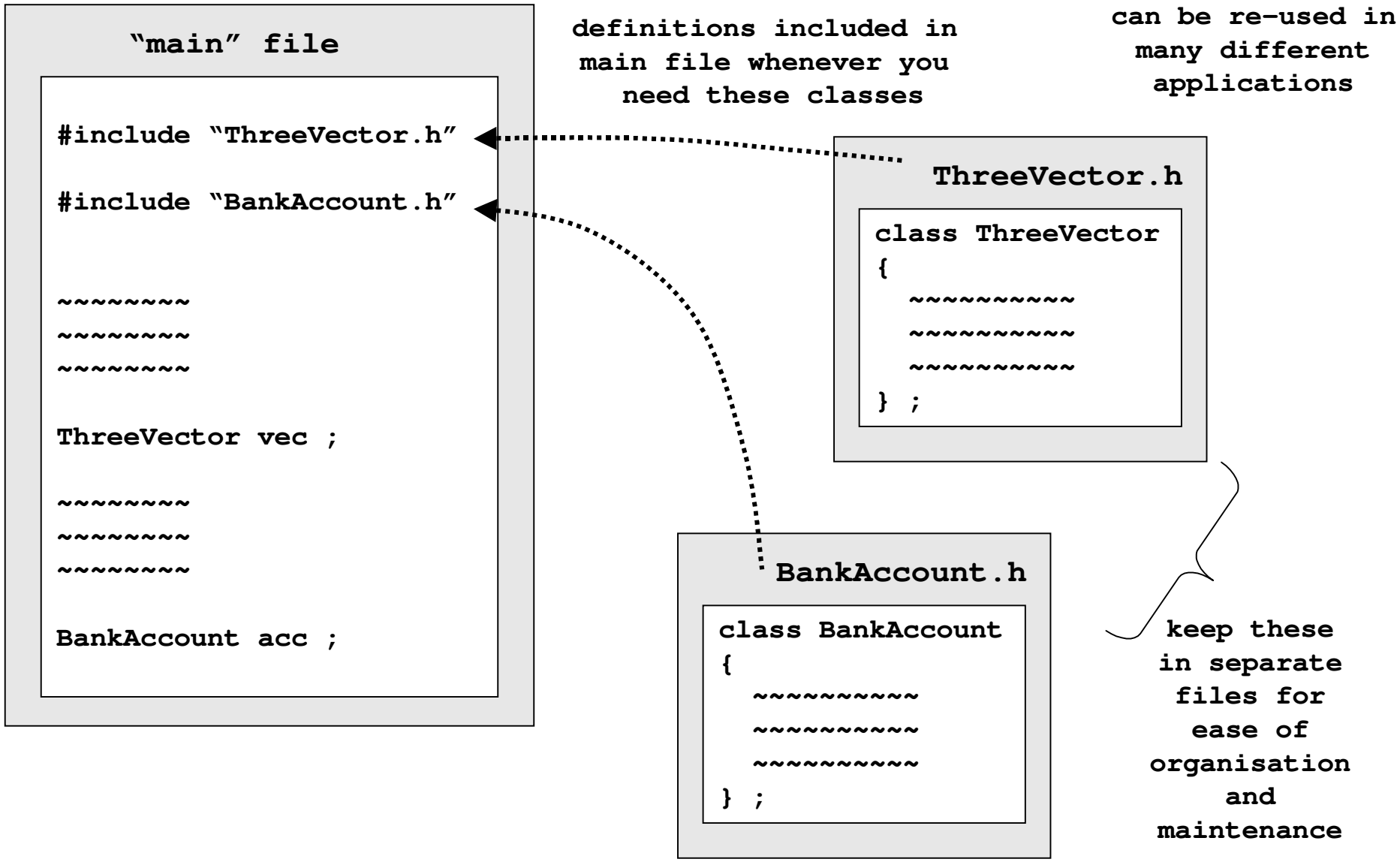
-Modify the code you have written earlier (to find the angle between two vectors) to use the ThreeVector class you have defined.

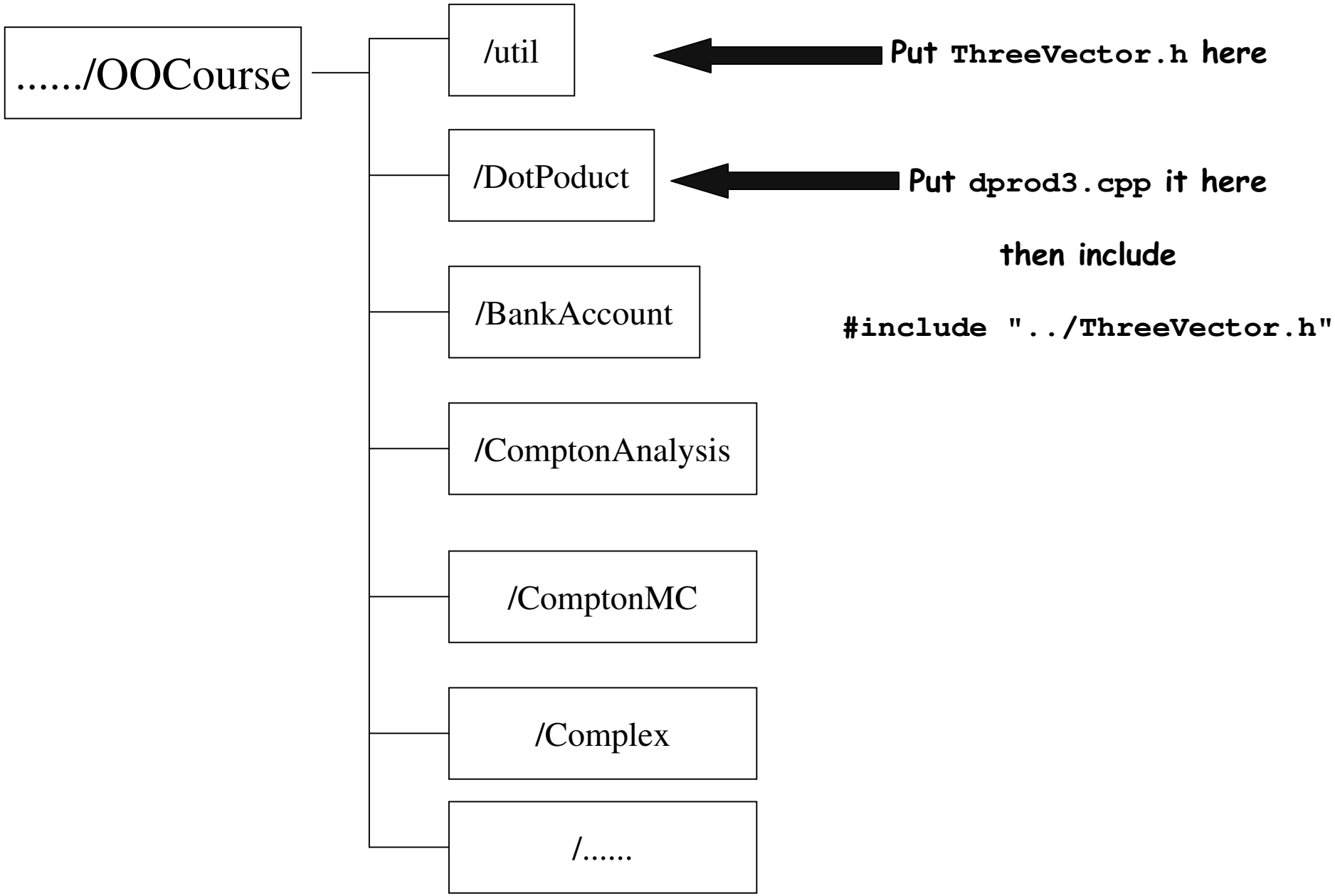
- This will require changing the arguments of any functions you have written to accept instances of the new class type.

util/
ThreeVector_firstgo.h

DotProduct/
dprod3.cpp

Note on organisation of code for classes




...../OOCourse
/util
Put ThreeVector.h here
/DotPoduct
Put dprod3.cpp it here
then include
#include "../ThreeVector.h"
/BankAccount
/ComptonAnalysis
/ComptonMC
/Complex
/.....

We are so happy with what we have done that we are going to invent a new word to celibrate:

object

When we say we have

"we have made an instance of a class"

we will say

we have made an object

The
class
is the type

Just like you would use
int
or
float
as a type.

```
// Instances of types

BankAccount peteClarke ;
```

The
object
is an instance of that
class

The object can hold
values of its internal
member variables

We are going to spend a little time at this point to ensure that you fully understand what has gone on in the last section.

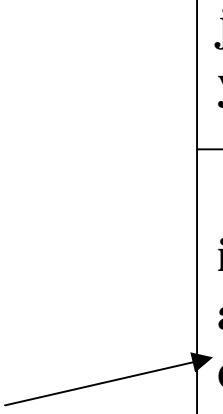
2.3 Operations on objects using methods

We are now going togo much further

We want to separate the user from the details of the "member variables" in a class definition.

We are going to INSIST that a user only interacts with an object via a set of specially written functions which perform all the things we ever need to do to it.

Opposite you see some functions to operate on BankAccount objects



class: BankAccount
Member variables: holdersName currentBalance overdraftLimit jointAccount yearsHeld
Methods: initialise( ) availableFunds( ) deposit( ) withdrawl( ) printStatus( )

We could do this just like you did in the previous example, i.e.

⇒ just write a plain old function for each operation

⇒ each such function would be passed a BankAccount just like you passed a ThreeVector to dotProduct(..)

However if we did this, these functions would really be disconnected from the class.

In other words it could only be by "agreement" that users would use them to operate on the class.

Instead we are now going to take a step which you will not have seen in any procedural language:

> **⇒ We are going to place the functions inside the definition of the class itself.**

Just like we had "class member variables" we now will have "class member functions"

you could put this another way and say that the functions are going to "belong" to the class.

These special functions are called "member functions" or equivalently "methods"

Here is how you write member functions (methods) as part of a class:

```
// C++ code to define a BankAccount class
// Now with two member functions (methods) added

class BankAccount
{
  public:

    string holdersName ;
    float  currentBalance ;
    float  overdraftLimit ;
    int    jointAccount ;
    int    yearsHeld ;

    void withdrawl( )
     {
       .... write function code here...
     }

    float availableFunds( )
     {
       .... write function code here ....
     }

     .... other methods similarly written ....
} ;
```

This method is written just like a normal function, but inside the body of the class

Here are the details of a trivial method: **availableFunds( )**

```
float availableFunds( )
{

  // This is a method of BankAcount

  // Its function is to return the funds which are
  // available for use by the account holder

   float funds ;

   funds = currentBalance + overdraftLimit ;

   return funds ;

}
```

It doesnt need any arguments.

The method uses the member variables of the `BankAccount`

Here is the **withdrawl( )** method

This is how you declare that it doesn't return anything

Note the **if{}** and **else{}** statements, and the use of the **<** operator. We will return to these later in more detail.

```
void withdrawl( float amount )
{
  // This is a method of BankAcount
  //
  //Its function is to make a withdrawal if
  //this is possible.

  // check whether this withdrawal can be made

  float limit ;
  limit = currentBalance+overdraftLimit ;

  if( amount < limit )
  {
    // Ok to withdraw
    currentBalance -= amount ;
  }
  else
  {
    // Not enough money - tough luck
  }

  return ;
}
```

Student exercise

- A partially written file is provided for you called:

`BankAccount_for_students.h`

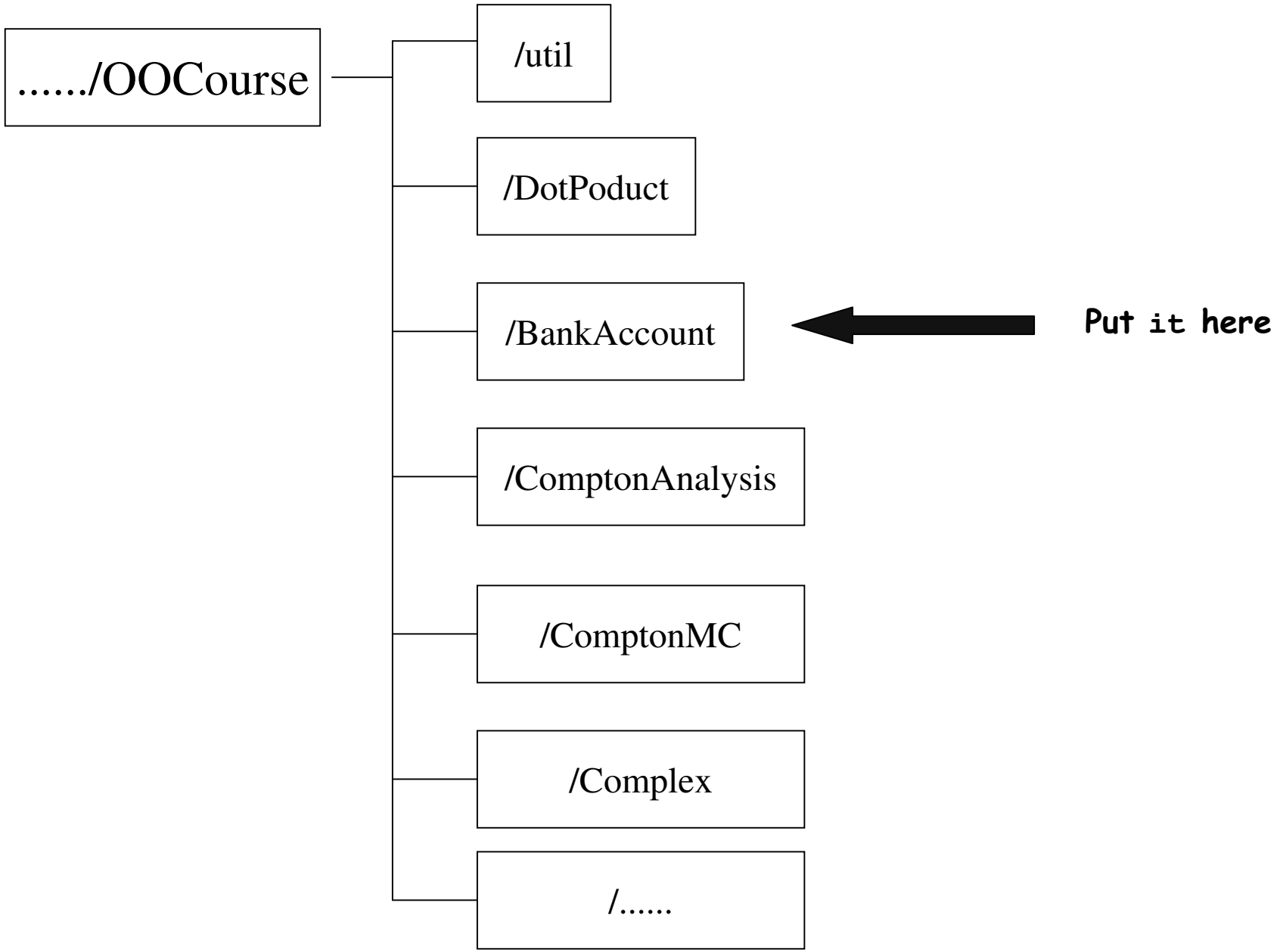
this includes the methods we have already looked at. Copy this file and rename it to BankAccount.h

- Complete this file by adding the other methods listed earlier, i.e.

```
initialise( ... )
deposit( ... )
printStatus( )
```

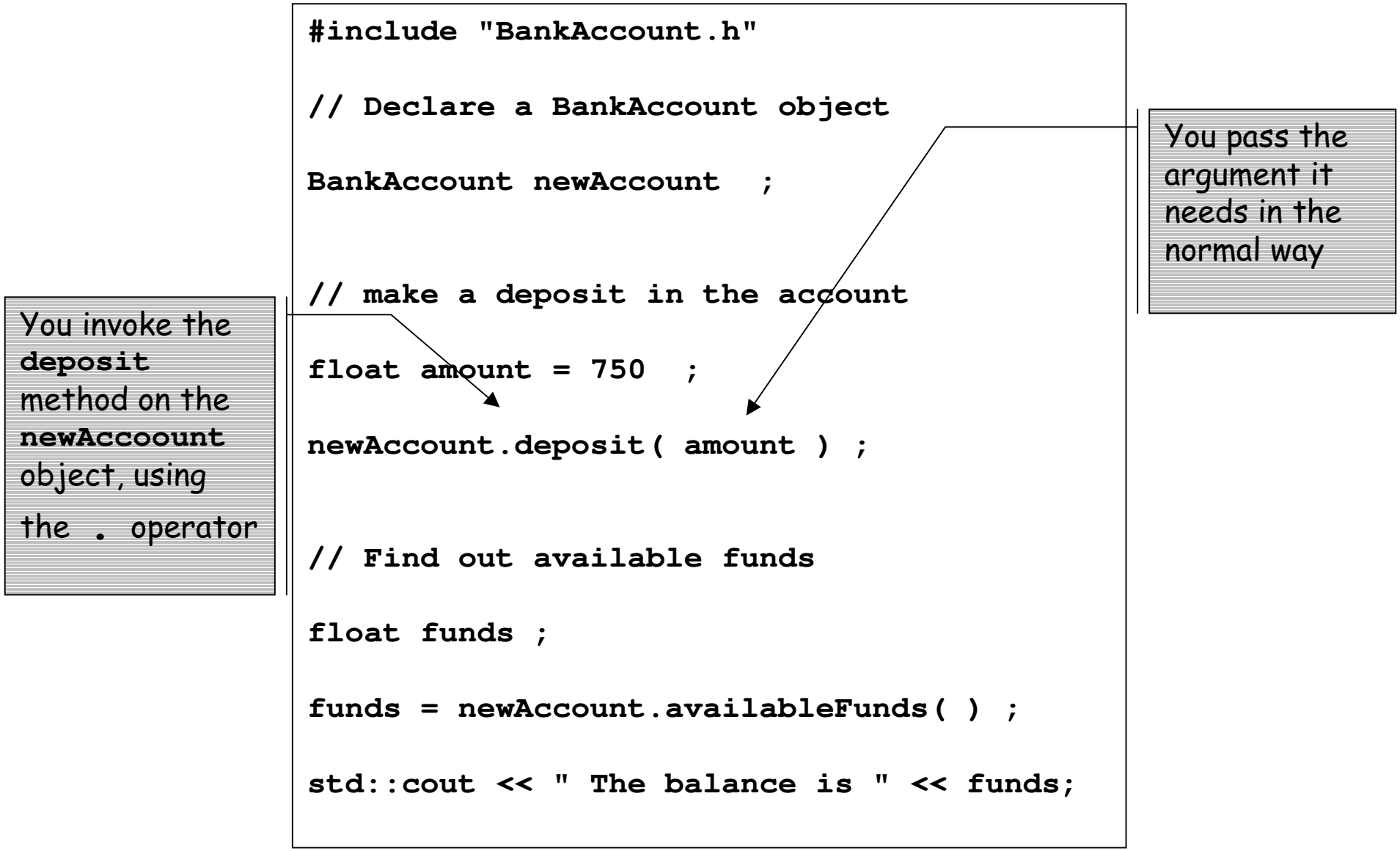
- Modify the `withdrawl( )` method to return a `bool` variable (true or false) to indicate whether the withdrawl was successful.

BankAccount/
BankAccount_inline.h


...../OOCourse
/util
/DotPoduct
/BankAccount
Put it here
/ComptonAnalysis
/ComptonMC
/Complex
/.....

2.4 Using methods

Now we have written the `BankAccount.h` to include some methods, we will see how to use them in a program


#include "BankAccount.h"
// Declare a BankAccount object
BankAccount newAccount ;
// make a deposit in the account
float amount = 750 ;
newAccount.deposit(amount) ;
// Find out available funds
float funds ;
funds = newAccount.availableFunds( ) ;
std::cout << " The balance is " << funds;
You invoke the deposit method on the newAccoount object, using the . operator
You pass the argument it needs in the normal way

Note a key concept here:

```
// Declare a BankAccount object

BankAccount newAccount  ;


// make a deposit in the account

float amount = 750  ;

newAccount.deposit( amount ) ;
```

This syntax is telling the object (`newAccount`) to do something to itelf (`deposit( )` an `amount`)

When the deposit method is run it knows which member variables to operate on ⇒ those of the particular object for which it has been called

It is as if each object has its own personal set of functions which operate only upon its "internal" member variables

In other words:

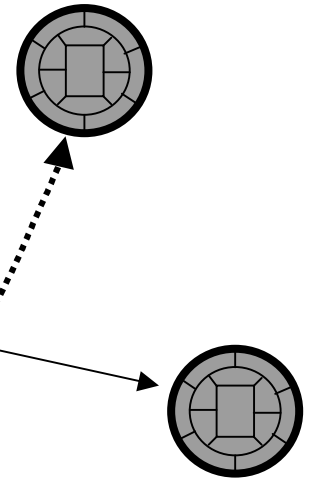
```
// Declare an account

BankAccount newAccount1  ;
BankAccount newAccount2  ;


// make a deposit in account 1

float amount = 750  ;

newAccount1.deposit( amount ) ;



// make a deposit in account 2

amount = 500  ;

newAccount2.deposit( amount ) ;
```

This invokes the **deposit** method to operate on the member variables of the first account object.

This invokes the **deposit** method to operate on the member variables of the second account object

Pictorially:

```
// Declare an account

BankAccount newAccount1  ;
BankAccount newAccount2  ;


// make a deposit in account 1

float amount = 750  ;

newAccount1.deposit( amount ) ;



// make a deposit in account 2

amount = 500  ;

newAccount2.deposit( amount ) ;
```



Look in the following file for a more complete example

Private study

Make sure you are familiar with what is going on in this example:

- a class written in a file and #included

-making variables which are instances of the class

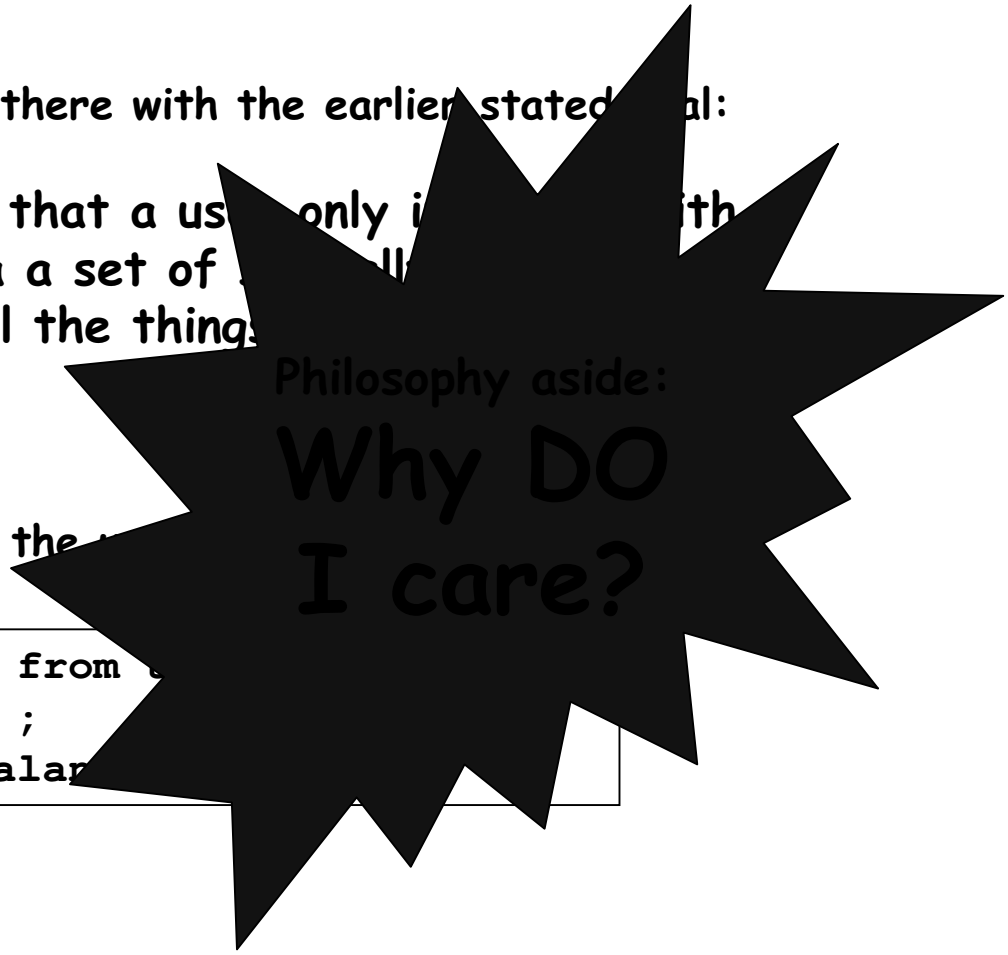
- using methods to operate on each instance

YOU MUST SEEK FURTHER EXPLANATION IF YOU ARE IN DOUBT AT THIS POINT

2.5 Public -vs- Private members

We are nearly, but not quite there with the earlier stated goal:

"We are going to INSIST that a user only interacts with the contents of a class via a set of [illegible] functions which perform all the things [illegible] it."



Up to this point nothing stops the [illegible] varibles directly, i.e:

```
// make a withdrawl from t
float amount = 750  ;
newAccount.currentBalan
```

... rather than

```
// make a withdrawl from the account
float amount = 750  ;
newAccount.withdrawl( amount ) ;
```

C++ allows you to declare member variables to be

"private"

so that ONLY the methods of the class can alter them

Here is how you make things in the class private

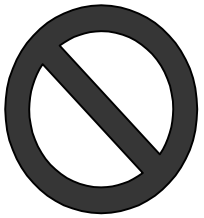
Anything following this can only be used by methods of the class

Anything following this can be used by anyone outside the class

```
class BankAccount
{
  private:

    string holdersName ;
    float  currentBalance ;
    float  overdraftLimit ;
    int    jointAccount ;
    int    yearsHeld ;

  public:

    float initialise( )
     {
       ....
     }

    float availableFunds( )
     {
       ....
     }

     .... other methods  ....
} ;
```

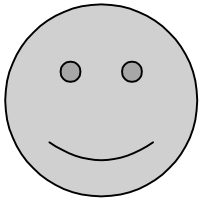
Now this WILL NOT work (the compiler will not allow it)

```
// make a withdrawl from the account
float amount = 750  ;
newAccount.currentBalance -= amount ;
```



Only this WILL WORK

```
// make a withdrawl from the account
float amount = 750  ;
newAccount.withdrawl( amount ) ;
```



2.6 The first steps toward "Data Encapsulation"

Up to this point we have been skirting around one of the fundamental concepts of OO programming - that of

"Data Encapsulation"

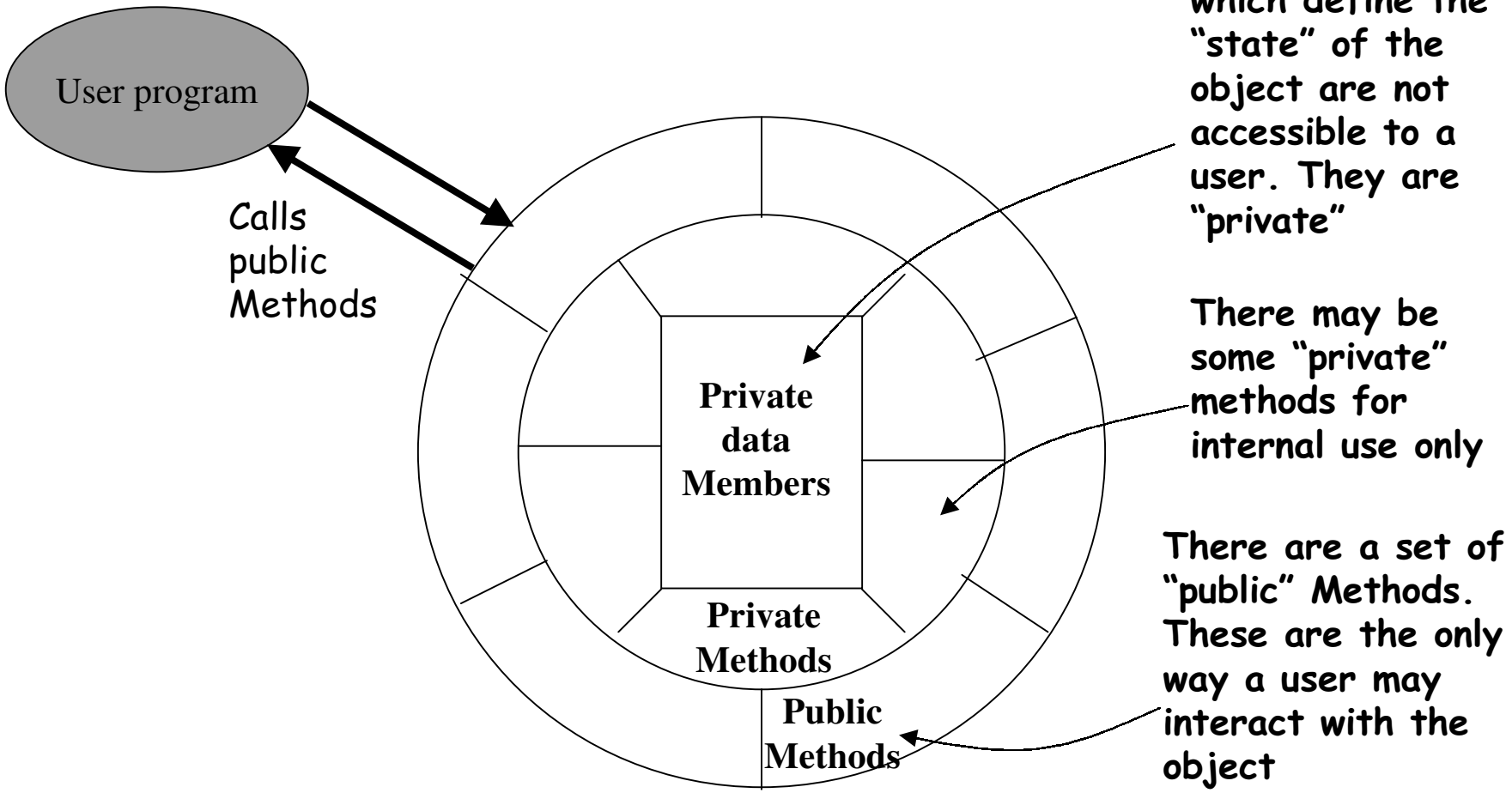
In this section we will spend a few moments consolidating this idea.

(don't worry if you find this a bit abstract at present !)

This is a first look at the idea of an "object"

⇒ something which encapsulates its "state" internally,

⇒ only allows users to interact with it through a public interface

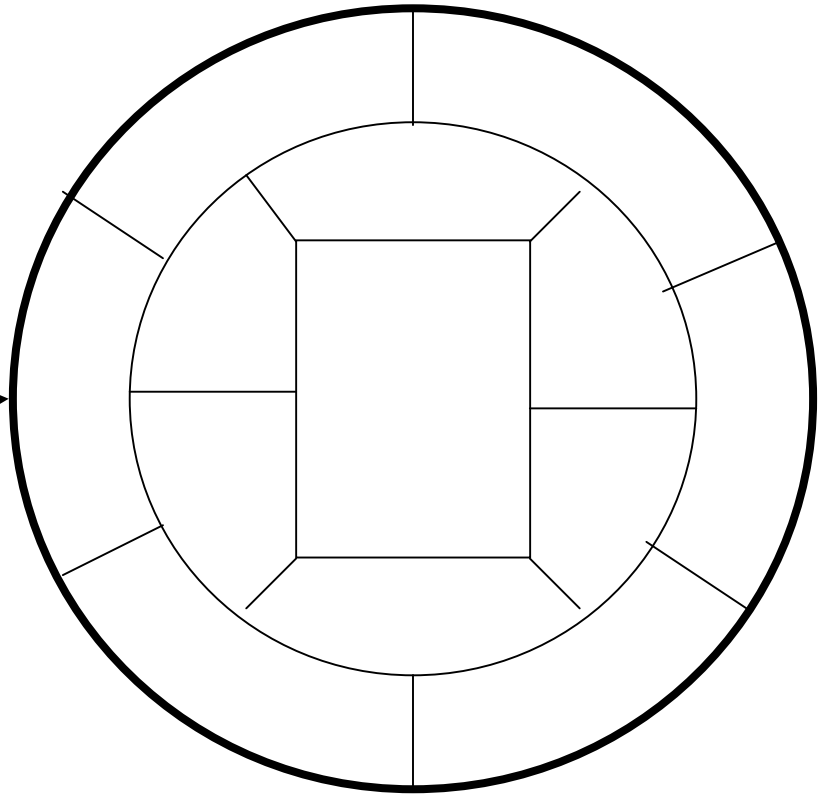


Why bother with all this ?

It separates the user from the implementer

⇒ the implementer can do what they like inside, and even change it later

⇒ none of this will affect a user provide the interface is maintained



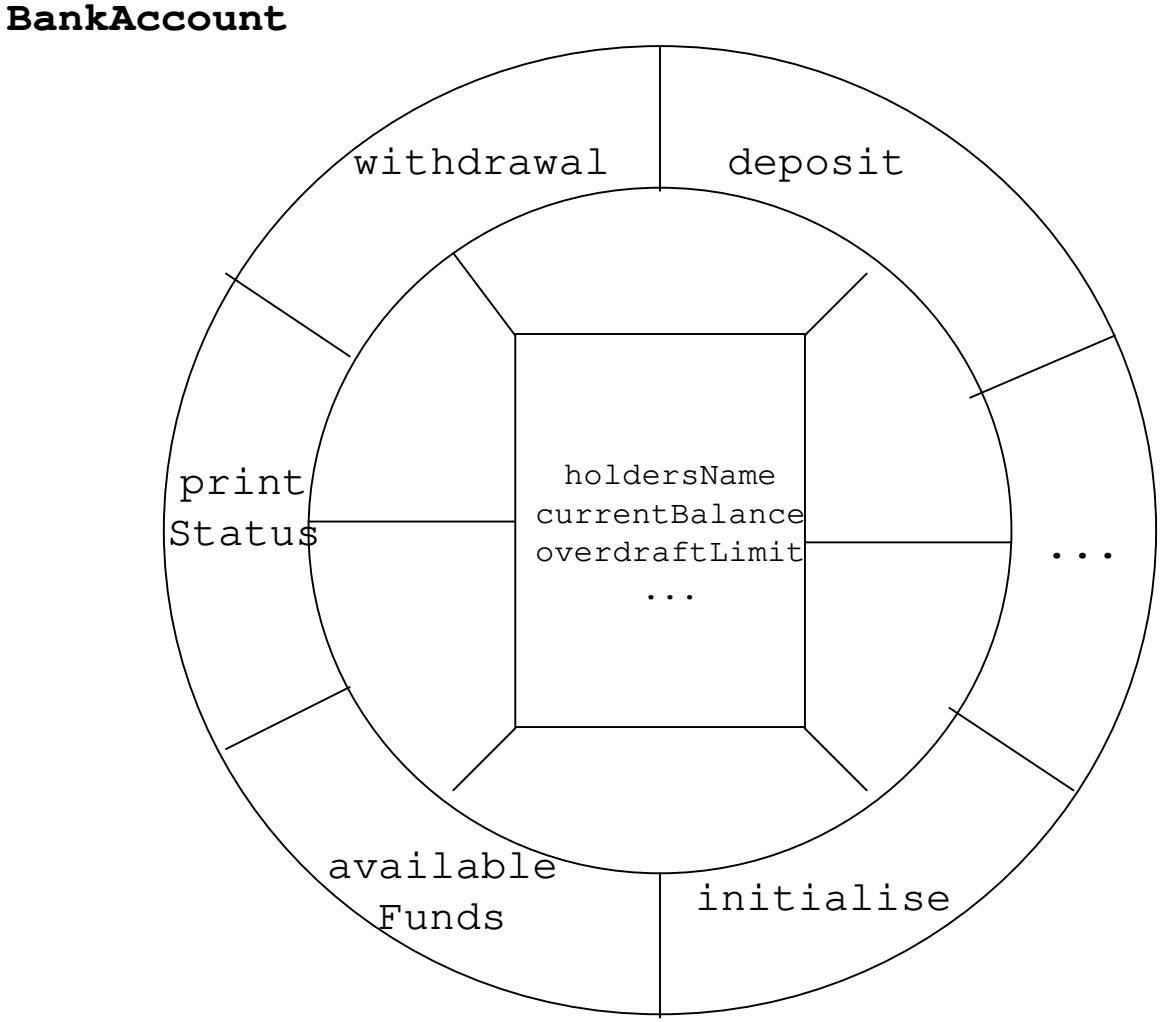
Protective membrane

⇒ re-use

⇒ easier maintenance

....well.. some of these things anyway

We have developed a (not very interesting) example:



You have to get into the mode of thinking about software in terms of

objects

You have to ask:

"What services should an object provide"

You should not care how it does it, nor how it represents itself internally

In other words you must get away from the habit of knowing or caring what the internal member variables are.

Student exercise

-Write the ThreeVector class properly

- You should include the following methods (and any others you think fit)

initialise( ) - to initialise the data members to values specified in arguments.

magnitude( ) - to return the magnitude of the vector

dotProduct( ) - form the dot product between the vector and another one supplied in an argument.

angle( ) - to return the angle between the vector and another one supplied in an argument.

dump( ) - to print out some suitable info on the vector contents

Modify your dot product code to use all these methods, particularly the initialise() method.
The code is much neater now !!!!!!

util/
ThreeVector_
secondgo.h

DotProduct/
dprod4.cpp

Summary of Module 2: User defined Data Types & Operations upon them

- **Inadequacies of in-built types**
  - **limited use**
  - **problems generally contain abstract entities**
  - **difficulty in referring to a single instance**
  - **difficulty in passing a group of variables as arguments**

- **User defined data types**
  - **classes**
  - **declaration of instances of a class (object)**

- **Operations on object member variables:**
  - **access to members of an object**
  - **simple manipulations of members**

- **Operations on objects via methods**
  - **Use of "Methods" to hide the implementation of an object from a user**

- **First steps toward data encapsulation**
  - **Consolidation of the ideas of:**
    - **Object consisting of "state" and "Methods" to use/change it**
    - **"private Members", I.e. hiding data members from user**
    - **"public Methods", I.e the things a user program may use**