Module 8:
More on methods:

In this module we will cover:

- **Overloading of methods**
- **Private methods and self messaging**
- **const methods**
- **const arguments**

Aims of this module

We have previously introduced the concept of a "class" as a construct which combines member variables and methods in a formal way.

So far, however, we have only looked at very simple aspects of methods of a class. In particular we have only written the methods using simple features (i.e those which you might be familiar with from simple C or Fortran)

In this module you will learn about some features of C++ which allow you to use methods of objects in a much more flexible and controlled way.

8.1: Overloading of methods

You can have several methods of a class having the same name provided that their argument list is different.

This feature is very simple but very useful

As example consider you are designing some code to administer Library stock What sort if items do you have in a library ?

Books

Magazines

Journals

Videos

...etc

Each of these is characterised in a different way:

Books : Subject code, Author, Title, ISBN....

Magazines: Title, Year, Volume, Issue number...

...

You would probably decide that it made sense to define a different class to represent each of these.

```
class Book {
   int subject ;
   string author ;
   string title ;
   int isbn ;
};
```

```
class Magazine {
   string title ;
   int year;
   int vol ;
   int issueno ;
};
```

```
class Journal {
   ...


};
```

Next you might decide that you need a class called `Catalogue` which must store all library items.

What services (i.e. methods) should `Catalogue` provide ?

-Presumably you want to be able to add different items to the catalogue.

Therefore you might write the `Catalogue` class like this:

```
class Catalogue {
 private:

   vector<Book> booksInStock;         // Vecotr to hold books

   vector<Magazine> ferretWeekly;     // To hold issues of FW
   vector<Magazine> arcWeldersWorld; // To hold issues of AWW

 public:
   void addABook( Book ) ;
   void addAMagazine( Magazine ) ;
};
```

You need one method name to add a Book:

and a different method name to add a Magazine:

```
// Here are the Catalogue methods

// Add a book method
void Catalogue::addABook( Book newBook )
{
  booksInStock.push_back( newBook ) ;
}


// Add a magazine method
void Catalogue::addAMagazine( Magazine mag )
{
  if( mag.title() == “ferretWeekly” )
  {
    ferretWeekly.push_back( mag );
  }
  else if( mag.title() == “arcWeldersWorld” )
  {
    arcWeldersWorkd.push_back( mag ) ;
  }
}
```

Why do you need two different method names ?

As far as the user of a `Catalogue` is concerned :

- **they only care that they want to add something to the catalogue**
- **they don't care about the details of what `Catalogue` does about it**

Thus a user only really wants a single method called

`add(..)`

In OO languages this is allowed !

You declare your class like this:

```
class Catalogue {

 private:
   vector<Book> booksInStock;         // Vecotr to hold books
   vector<Magazine> ferretWeekly;     // To hold issues of FW
   vector<Magazine> arcWeldersWorld; // To hold issues of AWW

 public:
   void add( Book ) ;
   void add( Magazine ) ;
};
```

...and here is the add method for a Book:

... and the add method for a Magazine:

Note that they have the same name.

The compiler is smart enough to work out which one you want from the argument you pass to it

```
// Here are the Catalogue methods

// Add a book method
void Catalogue::add( Book newBook )
{
  booksInStock.push_back( newBook ) ;
}


// Add a magazine method
void Catalogue::add( Magazine mag )
{
  if( mag.title() == “ferretWeekly” )
  {
    ferretWeekly.push_back( mag );
  }
  else if( mag.title() == “arcWeldersWorld” )
  {
    arcWeldersWorkd.push_back( mag ) ;
  }
}
```

Here is a program which shows how we would use these two different methods

```
// Program fragment to demonstrate overloaded
// methods

// Make a Catalogue
Catalogue ugLibrary;

// Make a book and set its author ..etc
Book b ;
b.initialise( “Ian M Banks”, 12345 ) ;

// Make a magazine
Magazine m ;
m.initialise( “ferretWeekly”, 2000, 6  ) ;


// Now add them to the catalogue

uglibrary.add( b ) ;
ugLibrary.add( m )  ;
```

This feature is called: "method overloading"

You may have as many methods with the same name as you like, provided their argument list is different.

The compiler works out which method to actually invoke.

It is used extensively to keep user code simple.

In fact we have already seen this used for constructors. Remember that you can have many constructors with different argument lists. This is possible as the constructors are overloaded.

8.2: Private methods & self messaging

We have already covered this but re-cover it here just for completeness.

It may well be (very likely) that within some method of a class you need to do something which uses another method of the same class.

We saw this in BankAccount

Here the method

statusCheck(..)

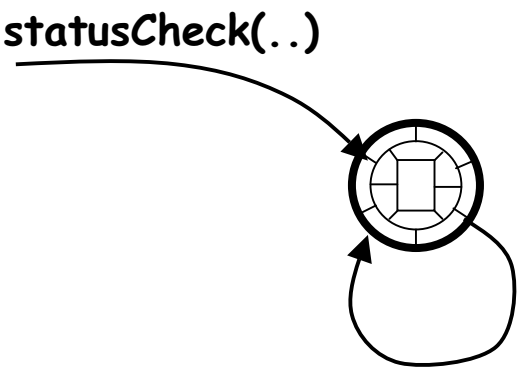
uses the method

availablefunds(..)

```
//Method of BankAccount: statusCheck
void BankAccount::statusCheck( )
{

  if( availableFunds( ) < 0.0 )
  {
    // It seems to have exceeded the limit

    cout << "\n\n Dear Mr/Ms .......

  }

  return ;
}
```

This is known as "self messaging"

This is when an Object sends a message to itself (I.e. calls a method of itself)



Now in this case `availablefunds(..)` is a public method anyway.

However it may be that you want to write a method purely for internal use. I.e you don't want anyone outside the object using it.

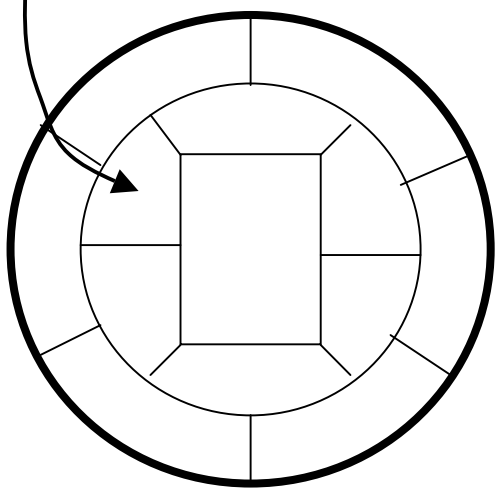
Suppose we don't want an outsider to be able to find out the available funds.

Then we can make this a private method

We do it simply by declaring it in the private section of the class like this:

```
class BankAccount
{

  private:
    string  holderName  ;
    float currentBalance  ;
    float overdraftLimit  ;
    bool  jointAccount    ;
    int   yearsHeld       ;

    void availableFunds( ) ;

  public:
    bool statusCheck( ) ;
    ....
    ...other methods....
    ...
};
```

This is what this bit of the icon symbolises



Use of this-> for self messaging

The following is a "good style" recommendation:

When you self message in a piece of code, always invoke the method using this->

```
//Method of BankAccount: statusCheck
void BankAccount::statusCheck( )
{

  if( this->availableFunds( ) < 0.0 )
  {
    // It seems to have exceeded the limit

    cout << "\n\n Dear Mr/Ms .......
  }

  return ;
}
```

Doing so makes it explicitly clear that you are calling a method of the object, rather than some external function. You soon get very used to doing this.

8.3: const methods

Just as you can have const items you can have const methods

What does this mean ?

It means that you are telling the compiler that this method promises not to change the state of the object in any way.

This is useful to:

(i) ensure that anyone writing or modifying the code of the method doesnt accidentally change anything (it will fail to compile)

(ii) help the compiler generate efficient code

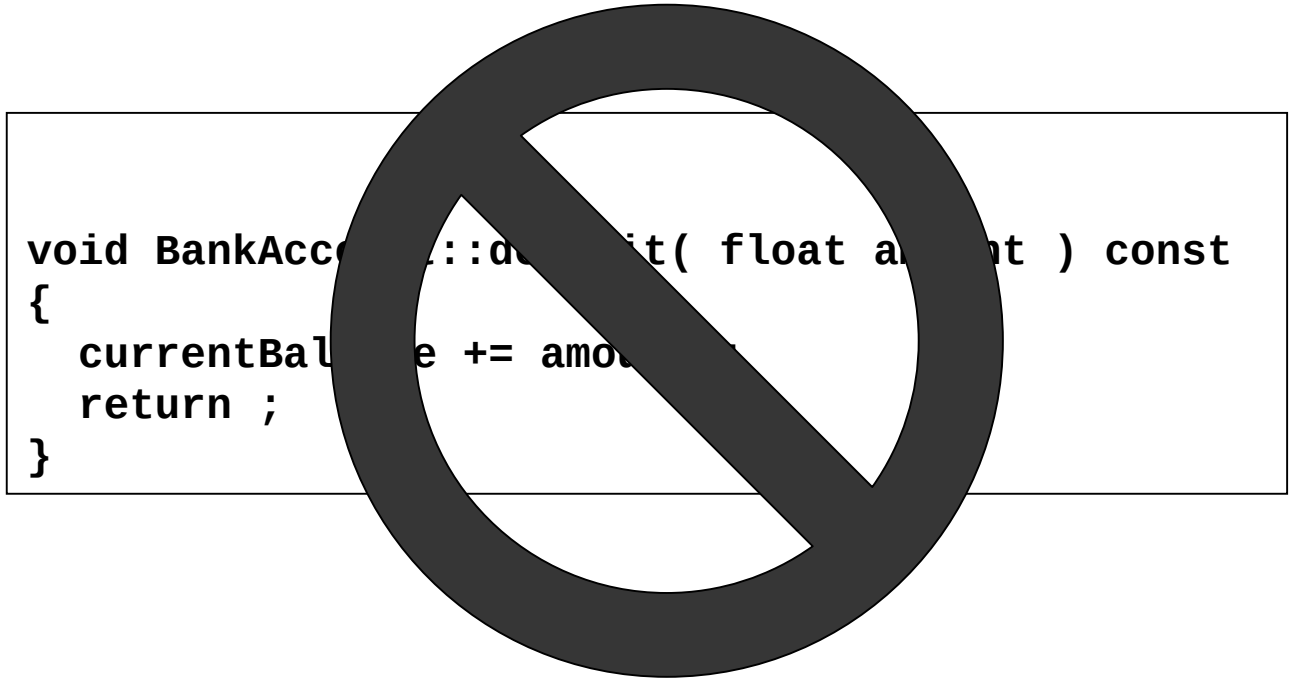
Here is an example of a method which is (and should be!) const

```
//const Method of BankAccount

float BankAccount::availableFunds( ) const
{
  return (currentBalance + overdraftLimit) ;
}
```

Here is an example of a method which cannot be const

```
//non const Method of BankAccount

void BankAccount::deposit( float amount )
{
  currentBalance += amount ;
  return ;
}
```

If you tried this it would generate a compiler error:



```
void BankAcc  t::de  it( float a  nt ) const
{
  currentBal  e += amo  ;
  return ;
}
```

It is very good practice to methods const wherever possible

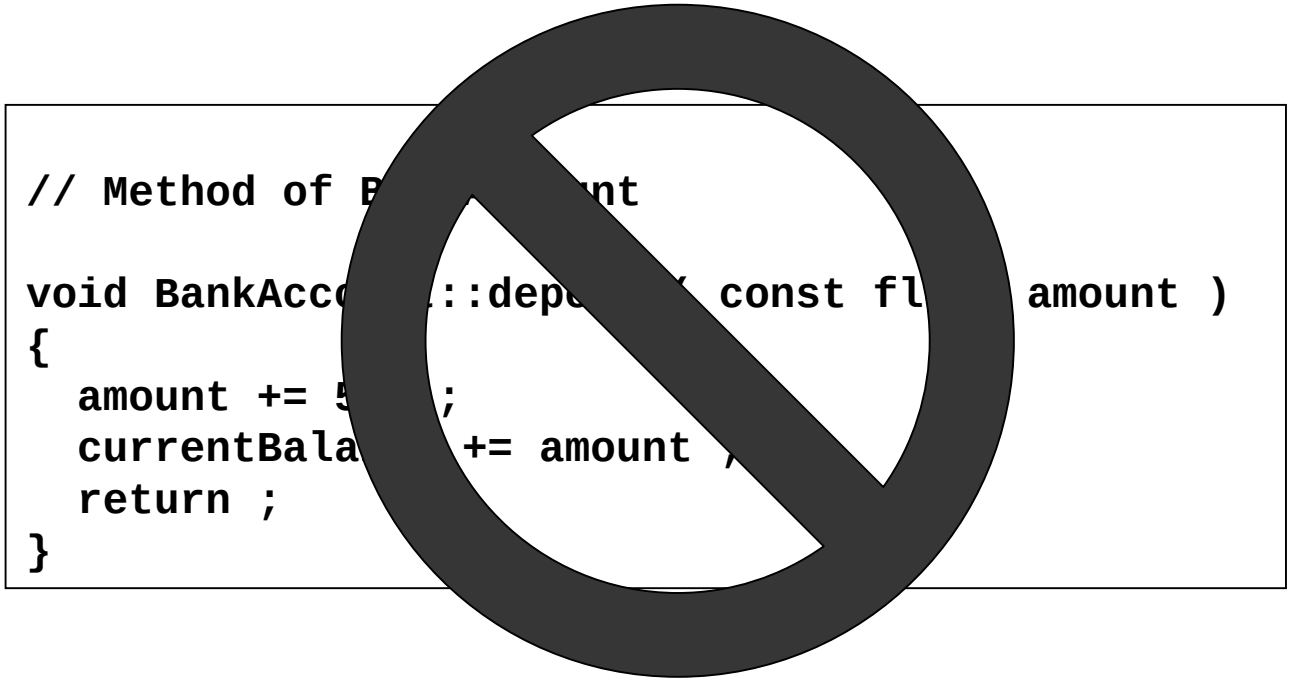
8.4: const arguments

You can also tell the compiler that an argument which you are going to pass will not be changed by the method. This also helps the compiler and protects from user programming errors.

Here is an example of a `const` argument

```
// Method of BankAccount

void BankAccount::deposit( const float amount )
{
  currentBalance += amount ;
  return ;
}
```

If you tried this it would generate a compiler error:



```
// Method of B      nt

void BankAcc    ::dep    ( const fl    amount )
{
  amount += 5    ;
  currentBala    += amount ;
  return ;
}
```

Summary of Module 8:
More on methods:

In this module we have covered the following topics.

- **Overloading of methods**
  - **You can have many different methods with the same name provided they have different argument lists**
- **Private methods and self messaging**
  - **If you want to write a method purely for internal use by the class, then you should make it a private method**
  - **Methods of a class can call other methods of the class.**
- **const methods**
  - **If a method does not change the state of an object then you should always make it const**
- **const arguments**
  - **If a method does not change an argument then make the argument const**