Implementing Associations & Aggregations

- There are two basic ways in which associations and aggregations are implemented
  - Objects contain objects (fixed aggregation)
  - Objects contain pointers to objects (variable aggregations and simple associations)

Aggregation v Inheritance

- The main advantage multiple inheritance has over aggregation is that it allows objects of the derived class to be dynamically bound in the same hierarchy as its base classes. It is also necessary where virtual methods must be overridden by the derived class – an aggregate cannot do this.

Fixed aggregations:

- Fixed aggregations are implemented by using a class type as an attribute type in the aggregate class.

```
#include "chapter.h"
class book
{    private:
     chapter c1,c2,c3,c4;
     public:
     book(int pages);
     int get_book_pages();
}
```

Class chapter

The class chapter must be defined in a header file and imported in to the file which defines book.

```
class chapter
{       private:
        int no_of _pages;
        public:
        chapter(int pages);
        int get_pages();
}
```



```
chapter::chapter(int pages)
  {
   no_of_pages = pages;
  }
int chapter ::get_pages()
  {
   return no_of_pages;
  }
```


The constructor for book calls the chapter constructor:

```
book(int pages):
c1(pages),c2(pages),c3(pages),c4(pages)
   {
   }
```

The body of the constructor is empty as the book has no other attributes to instantiate.

Note that this generates a book where each chapter has the same number of pages.

If each chapter had different numbers of pages (which is more realistic) then a parameter for each chapter would be sent into the book construcor i.e.

```
book(int pages1, int pages2,
        int pages3, int pages4):
             c1(pages1), c2(pages2),
             c3(pages3), c4(pages4)
{
}
```

The method get_book_pages() will call the get_pages method of chapter:

```
int get_book_pages()
{
 return (c1.get_pages() + c2.get_pages()
           + c3.get_pages() +
                c4.get_pages())
}
```

Note that the constructor and method of chapter, when used within book, are called with an instance of the class chapter e.g. c1.get_pages(), c1(pages1)

Aggregation versus Inheritance

- In aggregation an instance of the component class (e.g. chapter) must exist in order for the aggregate class to exist(e.g. book)
- In inheritance a base class constructor is called from a derived class as follows:

```
Derived_class_name::
  Derived_class_constructor_name(types and
  parameters):
  base_class_construcor_name(parameters)
{ set parameters which are not set by the
  base class constructor;
}
```


In aggregation a component class constructor is called from an aggregate class as follows:

```
Aggregate_class_name::
  Aggregate_class_construcor_name(types and
  parameters):
  Component_class_instance(parameters)
{
  set parameters which are not set by the
  component class constructor;
}
```


Variable Aggregations

A component may not always be present at runtime. Therefore, we may represent it using a pointer to an object rather than by an object itself.

Example:

A lecturer may be assigned a course to teach. That lecturer may be removed from that course at anytime. The relationship is one that may change. Hence a fixed aggregation is unsuitable for representing this relationship. Variable aggregation is more suitable.

```
#include "lecturer.h"
class course
{ private:
 lecturer * L // L is a pointer to an
object of type lecturer
   public:
 course();
 void addlecturer();
 void removelecturer();
  }
```

The class lecturer is defined in the header file lecturer.h.

- The course constructor should initialise the pointer L to NULL as no lecturer is associated with the class when it is set up.
- The method addlecturer should put the pointer L pointing at a new object of type lecturer:

```
void course::addlecturer()
{
 L = new lecturer;
}
```


- The method removelecturer should delete the object L is pointing at and set L pointing to NULL:

```
void course::removelecturer()
{
      delete L;
      L = NULL;
}
```


If we don't want to create / delete a new object (lecturer) each time we add/remove we could define the above methods as follows:

```
void course::addlecturer(lecturer
*teacher)
{
 L = teacher;
}
void course::removelecturer()
{
 L = NULL;
}
```

Implementing 1:1 Associations in one direction.

1:1 associations in one direction are implemented as above with a pointer to a class.

Example:

In the class "Button" the association Button changes Light is modelled by providing a pointer to a light object as an attribute of Button.

The Light Class

```
const int OFF = 0;
const int ON = 1;

class Light
{ private:
  int light_state;
   public:
  Light();
  void change_state();
  void show_state();
}
```



```
Light::Light()
{ light_state = OFF;
}
void Light::change_state()
{
  if(light_state==OFF)
  {     light_state=ON;
  }
  else  {    light_state==OFF;
        }
}
```



```
void Light::show_state()
{
  if(light_state==OFF)
  {
   cout<<"Light is off" << endl;
  }
  else
  {
   cout<<"Light is on" << endl;
  }
}
```

The Button class with association

```
class Button
{ private:
  Light* light_bulb;  // 1:1 association
   public:
  Button(Light* bulb);
  void press();
};
  Button::Button(Light* bulb)
  {
   light_bulb = bulb;
  }
```



```
void Button::press()
  {
   light_bulb -> change_state();

  }
```


Implementing 1:1 Bi-directional Associations

In 1:1 bi-directional associations both classes must include a reference(via a pointer) to the linked object.


```
class Person
  { private:
   char name[40];
   Person * partner;

   public:
   Person(char * name_in)
   char* getname();
   void marry(Person *spouse);
   void divorce();
   void showpartner();
  }
```

```
void Person::marry(Person *spouse)
{
  partner = spouse
}

void Person::divorce()
{
  partner = NULL
}
```



```
void Person::showpartner()
{
  if (partner != NULL)
  {
    cout << name << "is married to"<<partner->getname() <<
      endl;
  }
   else
  { cout << name << "is single." << endl;
  }
}
```



```
void main ()
{Person * Ross = new Person ("Ross");
  Person * Rachel = new Person ("Rachel");

   Ross -> showpartner();
   Rachel -> showpartner();
   Ross -> marry(Rachel);
   Rachel -> marry(Ross );

   Ross -> showpartner();
   Rachel -> showpartner();
   Ross -> divorce ();
   Rachel -> divorce();
}
```


The *this* operator

- Problem: User has to update both ends of the relationship.
- In C++ this may be avoided if a link is created in one direction and modified in the opposite direction using the operator ***this***.
- Using the class course defined earlier we can model a 1:1 association between a lecturer and course as follows:

A lecturer teaches one course and a course is taught by one lecturer.

```
class course
{private:
        lecturer * L // L is a pointer to
        an object of type lecturer
 public:
        course(int code);
        void addlecturer();
        void removelecturer();
}
```



```
class lecturer
  { private:
        course * C // C is a pointer to an
        object of type course
    public:
        lecturer(int number):
        void addcourse();
        void removecourse();
    }
```



```
void course::addlecturer(lecturer *teacher)
{
  L = teacher;
  L -> addcourse(this);
}

void lecturer::addcourse(course *module)
{
  C = module;
}
```



```
void main()
{
   course * SE613 = new course(1);
   lecturer *Rosemary = new lecturer(12);

   SE613->addlecturer(Rosemary);
  // this operator => the association is
  updated for both objects.
}
```

N:M associations are implemented using arrays of pointers from one class to another.