Arrays of Objects

```
#include <iostream.h>

class c1 {
   int i;
public:
   void set_i(int j) {i=j;}
   int get_i(){return i;}
};

main(){
   c1 ob[3];
   int i;
   for (i=0; i<3; i++) ob[i].set_i(i+1);

   for(i=0; i<3; i++) cout << ob[i].get_i() << "\n";
}
```

Output:
1
2
3

Arrays of Objects

- You may initialize each object in an array by specifying an initialization list like you do for other types of arrays.
- Each value in the list is simply passed to the constructor function as each element in the array is created.
- Here is a different version of the program that uses an initialisation.


```
#include <iostream.h>

class c1 {
   int i;
public:
   c1(int j) {i=j;} // constructor
   int get_i() {return i;}
};

void main(){
   c1 ob[3] = {1,2,3};   //initializer
   int i;
   for(i=0; i<3; i++)
      cout << ob[i].get_i() << "\n";
}
```


Overloading the constructor

- The constructer function defined in c1 now requires a parameter.
- This implies that any array declared of this type will be initialised.
- So, we can no longer say:

```
c1 a[9]; //error,
         //constructor needs initialisers
```

- To solve, we need to overload the constructor with one that takes no parameters.
- In this way, arrays that are initialized and those that are not initialized are allowed.


```
#include <iostream.h>

class c1 {
   int i;
public:
   c1(){i=0;}           //Called for non-initialized arrays
   c1(int j){i=j;}      //Called for initialized arrays
   int get_i() {return i;}
};
```

Given this class, both of the following are possible:

```
c1 a1[3] = {3,5,6};
c1 a2[34];
```


Arrays of objects

- What if the constructor takes more than 1 parameter?
- Or if it's overloaded?
- How do we declare an array of objects then?

```
#include <iostream.h>

class Numbers{
private:
   float floatNumber;
   int intNumber;
   short shortNumber;
public:
      Numbers(){  shortNumber = 0;
                  floatNumber = 0;
                  intNumber = 0;
                  cout << "\nNone";}
      Numbers(short s, float f, int i){
                  shortNumber = s;
                  floatNumber = f;
                  intNumber = i;
                  cout << "\nAll";}
```



```
      Numbers(float f){ shortNumber = 0;
                        floatNumber = f;
                        intNumber = 0;
                        cout<< "\nfloat";}
      Numbers(int i){   shortNumber = 0;
                        floatNumber = 0;
                        intNumber = i;
                        cout<<"\nint";}
      Numbers(short s){ shortNumber = s;
                        floatNumber = 0;
                        intNumber = 0;
                        cout<<"\nshort";}
};
```



```
void main()
{
Numbers myNums[5] = {   Numbers((short)5),
                        Numbers((float)5.5),
                        Numbers((int)5555555555),
                        Numbers(),
                        Numbers(5,5,5)};
}
```

OUTPUT
short
float
int
None
All

Pointers to Objects

- Just as you can have pointers to other types of variables, you can have pointers to objects.
- When accessing members of a class given a pointer to an object, use the arrow `->` operator instead of the dot operator


```
#include <iostream.h>

class c1{
   int i;
public:
   c1(int j){i=j;}
   int get_i(){return i;}
};
void main()
{
   c1 ob(88), *p;
   p = &ob;             //get address of ob
   cout << ob.get_i();  // use . to call get_i()
   cout << p->get_i();  //use -> to call get_i()
 }
```


Pointers to Objects

- When a pointer is incremented, it points to the next element of its type. (e.g. the next integer).
- In general, all pointer arithmetic is relative to the type fo data that the pointer is declared as pointing to.
- The same is true of pointers to objects.

```
#include <iostream.h>

class c1{
   int i;
public:
   c1(){i=0;}
   c1(int j) {i=j;}
   int get_i(){return i;}
};

void main()
{
   c1 ob[3] = {1,2,3};
   c1 *p;
   int i;

   p = ob;                          // get start of array
   for (i=0; i<3; i++)
      {
      cout << p->get_i() << "\n";
      p++;
      }
}
```


The `this` Pointer

- When a member function is called, it is automatically passed an argument that is a pointer to the object that generated the call
- i.e. it is passed a pointer to the object that invoked it.
- This pointer is called **`this`**.


```
#include <iostream.h>

/*This class computes the result of a
  number raised to some power*/

class pwr{
   double b;
   int e;
   double val;
public:
   pwr(double base, int exp);
   double get_pwr(){return val;}
};
```



```
pwr::pwr(double base, int exp)
{
   b = base;
   e = exp;
   val = 1;
   if (exp==0) return;
   for( ;exp>0; exp--)val = val*b;
}

void main()
{
   pwr x(4.0,2), y(2.5,1), z(5.7,0);
   cout << x.get_pwr() << " ";
   cout << y.get_pwr() << " ";
   cout << z.get_pwr() << "\n";
}
```


- Within a member function, the members of a class can be accessed directly
- So, **`b = base`**, means that the copy of b associated with the object that generated the call will be assigned the value
- We could also write: **`this->b = base`**
- In fact, **`b=base`** is simply shorthand for **`this->b = base`**

```
pwr::pwr(double base, int exp)
{
   this->b = base;
   this->e = exp;
   this->val = 1;
   if (exp==0) return;
   for( ;exp>0; exp--)
          this->val = this->val*this->b;
}
```


- Obviously, no sane programmer would *normally* write pwr() as just shown because nothing is gained, and the shorthand form is easier.
- However, the **`this`** pointer is very important when we start overloading operators
- It can also aid in the management of certain types of linked lists
- Note: static member functions do not get passed a **`this`** pointer.