Object Lifetimes

Types of Object

External (Global) Objects

- Exist through the program lifetime
- Have file scope

Automatic (Local) Objects

- Exist throughout the local scope in which they are created

Static Objects

- Exist through the program lifetime
- Only visible within local scope

Dynamic Objects

- Lifetime controlled within a particular scope

External (Global) Objects

In C++…

```
BankAccount account1;
void main()
{…
}
```

The declaration of objects is all that we can do outside the scope of {}

We cannot call methods of account1 outside {}

External Linkage

In a system containing several program modules, such objects may be declared as extern in other modules if their visibility needs to extend beyond the file scope in which they are defined.

Prog1.ccp	Prog1.ccp	Prog1.ccp
Manager Joe;	extern Manager Joe;	extern Manager Joe;
void main()	…	…
{ …		
}		

Automatic (Local) Objects

In C++…

- Declared inside the body of main.
- Visible anywhere inside main and persists until main finishes.

Example:

```
void main()
{
BankAccount account1;
…
}
```


Static Objects

A static object is declared, like an automatic object, within a local scope. However it is only initialised once, regardless of the number of times it falls in and out of scope.

In C++

```
void  function()
{ BankAccount account1;
  static BankAccount account2;
}
```


Both objects (account1 & account2) come into scope when the function is called, but the automatic object will be created and destroyed each time. The static object will be instantiated once and will persist to the end of the program.

Dynamic Objects

If Objects are unpredictable they must be represented dynamically.

We cannot give dynamic objects unique names
=>we must reference them using pointers.

In c++ a pointer can be directed to an area of dynamically allocated memory at run time in order to reference a newly created object. In this way the constructor is called via a pointer.

Constructor: called when an object is created, the default constructor is called if the user doesn't define one.

Destructor: used to destroy the object by freeing up the memory used by the object (clean up).
In dynamic objects there is no default destructor – the user must call it explicitly.

Destructor methods

- Have the same name as the class preceded by ~
- Cannot take arguments
- Cannot return a value

Creating dynamic objects

classname *pclassname = new classname;
e.g. counter *pcounter = new Counter;

Pcounter is not the name of an object but the name of a pointer to an object of type counter.

Dynamic object do not have names – they simply occupy a memory space which may be referenced by a pointer.

Calling the methods of a dynamic object:
pclassname -> method(parameters);
e.g. pcounter -> increment();

In order to call a method of a dynamic object, the pointer must be dereferenced before the objects methods may be accessed.
The '**->** ' operator must be used instead of '**.**'

```
delete pclassname;
delete pcounter;
```

This cleans up memory by explicitly calling the destructor which destroys the object currently referenced by the pointer.
Delete destroys the object NOT the pointer.
The pointer is still available to point to other objects of the class.

A pointer which is not pointing to an object may be pointing to any random area of memory. These pointers should be directed to '**NULL**'.

NULL is a constant which is declared in standard header files stddef.h, stdlib.h.

Example

```
#include <stddef.h>
void main()
{int *x = NULL;
  counter *pcounter = NULL;
   // insert code here
  if (pcounter ==NULL)
  {cout<<" No object referenced";
  }
  else
 {cout<<" Object referenced";
  }
}
```


Car.h

```
#include <string.h>
class car
{private:
  char colour[10];
public:
  car(char * colour_in);
  char* getColour();
};
car::car(char* colour_in)
{strncpy(colour,colour_in,9);
 colour[9] = '\0';}
char* car::getColour()
{ return colour;}
```

```
#include"car.h"
#include<iostream.h>
void main()
{
  car* garage = NULL;
  int menu_choice = 0;
 char temp_colour[10];
```



```
while(menu_choice !=3)
  {
   cout <<"Enter 1 to put the car in
  garage, 2 to remove it, 3 to quit";
   cin >>menu_choice;

   if (menu_choice == 1 && garage == NULL)
   {
        cout <<"Enter colour of car";
        cin >>temp_colour;

        garage = new Car(temp_colour);
        cout <<"Colour is" <<
             garage->getColour()<<endl;
  }
```



```
else if (menu_choice == 2 && garage !=
  NULL)
{  delete garage;
   garage = NULL;
   cout <<"Car removed" << endl;
}
}
if (garage != NULL)
{ delete garage;
}
}
```


The Metaclass

The MetaClass

- The class holds the attributes and methods which apply to objects of the class – it is the class of the objects. A class acts as a kind of blueprint or skeleton for objects of the class.
- The metaclass holds the attributes and methods which apply to the class itself – it is therefore the class of the class.
- Every class has one (and only one) metaclass, and the metaclass contains those parts of a class which are not appropriate to be duplicated for every object.

Class Attributes

Sometimes we may want to have a physical representation of something about a whole class of objects rather that about one object in particular – suppose we want to know how many objects of a class exist at one particular time – where do we store this information?

Store it in the class?

If we store this attribute in each object it gets duplicated for every object that we create. To keep the value of that attribute correct we would have to change its value for **every** object each time we create/destroy an object – this would be unnecessary duplication of the same data which could lead to inconsistencies.

The Metaclass

- To solve this problem we need a single attribute for the whole class.
- We could have a simple counting variable outside the class itself but this would break the rules of encapsulation by making the attribute a global variable accessible from outside the class.
- We cannot put the attribute in the class as this would make it an attribute for every instance of that class.
- The answer is to put the attribute in the metaclass.

A Metaclass is …?

The metaclass is a repository for class attributes which only have one instance regardless of how many objects of the class exist. Class attributes tell us about the class as a whole as opposed to objects of the class.

Class attributes are just as visible as object attributes to an object. However, the value of object attributes (an objects state) remain invisible to the class.

Class Methods

- Class methods also exist in the metaclass.
- These methods are used to access the class attributes. Like attributes, class methods are just as visible as object methods to an object.

Therefore, it is possible to access the value of a class attribute using an objects method.

Class Methods

- What if there are no objects of the class instantiated and we want to return the value of the class attributes?
- If an object is not created we cannot use it to access the class attributes. Hence, we need another way of accessing the class attributes – class methods.
- Class methods may be called by the class directly as well as by an object.

Note :

Constructors and Destructors belong to the metaclass!


```
#include <string.h>
class BankAccount
{
private:
// object attributes
  int account_number;
  char account_holder[20];
  float current_balance;
// a class (static) attribute to count
  bank accounts
  static int account_total;
```

```
public:
  BankAccount(float start_balance = 0.00);
  int getAccountNumber();
  char* getAccountHolder();
  float getCurrentBalance();
  void setAccountNumber(int number_in);
  void setAccountHolder(char* holder_in);
  void deposit(float amount);
  void withdrawal(float amount);
// a class (static) method to return the
  number of accounts
  static int getAccountTotal();
};
```



```
BankAccount::BankAccount(float
  start_balance)
{
  current_balance = start_balance;
// increment the class attribute when an
  object is constructed
  account_total++;
}
```



```
// the definition of the class method
int BankAccount::getAccountTotal()
{
  return account_total;
}

// reserve storage for the class attribute
int BankAccount :: account_total = 0;
```


Using Metaclass Attributes

```
#include <iostream.h>
#include "BankAccount.h"
void main()
{
// create a number of bank accounts
  BankAccount acc1, acc2, acc3, acc4, acc5;
// test the class method
  cout << "Total number of accounts is: "
  << BankAccount::getAccountTotal() <<
  endl;
}
```