Lecture-3

Object Lifetime and Dynamic Objects

Scope in C++

- In C++ scope is delimited by the { }
- Any variables declared within these sections are only valid within the braces

```
#include <iostream>
#include <cstdlib>

int main()
{
  for(int i=0; i<10; ++i)
  {
    std::cout<<i<<"␣";
  }

  std::cout<<i<<std::endl;

  return EXIT_SUCCESS;
}
```

Object Lifetimes and Dynamic Objects

- External (Global) Objects
  - Persistent (in existence) throughout the lifetime of a program.
- Automatic (local) Objects
  - Persistent and visible only throughout the local scope (where they were created)
- Static Objects
  - Persistent throughout a program but only visible within local scope
- Dynamic Objects
  - Lifetime may be controlled within a particular scope

What was that all about ?

- So far we have only used objects with specific names
    - these are called 'automatic' objects
- These can be determined at compile time so we give them set names
- These may be defined as external, static, or automatic depending upon their use
- However dynamic objects can't easily be defined at compile time so we have to use a different mechanism

External (global) objects

- External Objects are persistent and visible throughout a program module
  - i.e it's scope is an entire module (source file)
- It may also be visible to other source files
- These have identities and attribute which remain constant throughout the program
- External objects are always global to the module they are declared in
- and may become global to other objects by using the `extern` keyword

Automatic Objects

- Automatic Objects exist in a predictable manner for a particular period of time
- However Automatic objects are instantiated within a given scope in a program
  - that is, declared within the braces {} of either main or a function
- Automatic Objects are automatically destroyed once they fall out of the scope in which they were declared
- Therefore automatic objects are persistent and visible within the scope they are declared in.

Static Objects

- C++ allows for a variable or Objects to have the scope of an automatic object but the lifetime of an external Object
- This means that an object could be created once and only once, but with it's visibility delimited by the scope in which it is declared.
- This means that the object's state will persist even when it is not in scope (and therefore not visible)
- Therefore the Object can be said to 'remember' it's state when it comes

www.cplusplus.com/doc/tutorial/pointers/

Pointers

- variables are seen as memory cells that can be accessed using their identifiers.
- The memory of a computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte).
- These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one.
- This way, each cell can be easily located in the memory because it has a unique address and all the memory cells follow a successive pattern

www.cplusplus.com/doc/tutorial/pointers/

Reference operator (&)

- As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address).
- The address that locates a variable within memory is what we call a reference to that variable.
- This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". **`x = &book`**
- The variable that stores the reference to another variable (like ted in the previous example) is what we call a pointer.

Pointers

www.cplusplus.com/doc/tutorial/pointers/

- variable which stores a reference to another variable is called a pointer. Pointers are said to "point to" the variable whose reference they store
- Using a pointer we can directly access the value stored in the variable which it points to. To do this, simply precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".
- Notice the difference between the reference and dereference operators:
  - **&** is the reference operator and can be read as "address of
  - ***** is the dereference operator and can be read as "value pointed by"

```
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;

  return 0;

}
```

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed by p1 = 10
  *p2 = *p1;  // value pointed by p2 = value pointed by p1
  p1 = p2;    // p1 = p2 (value of pointer is copied)
  *p1 = 20;   // value pointed by p1 = 20

  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

Dynamic Objects

- Dynamic objects are used when the object to be created is not predictable enough
- This is usually when we cannot determine at compile time:
  - Object Identities
  - Object Quantities
  - Object Lifetimes
- Therefore when creating Dynamic Objects they cannot be given unique names, so we have to give them some other identities
- In this case we use pointers

Creating Dynamic Objects

- Dynamic Objects use dynamic memory allocation
- In C++ a pointer can be directed to an area of dynamically allocated memory at runtime
- This can then be made to point to a newly created object.
- To do this we use the **new** operator in the following way

```
Point2D *point1;
point1 = new Point2D();
```

- First we create a pointer to the Point2D Class using the *
- Then we use the new operator to construct a new instance to the class
-

Using Dynamic Objects

- To call a user defined constructor we use the following syntax

```
Point2D *point1 = new Point2D(100.0,100.0);
```

- To send a message (i.e. use a method) we use the -> 'pointed to' delimiter as follows

```
cout << point1->GetY();
```

- Apart from these distinctions dynamic Objects behave in the same as static

Destroying a Dynamic Object

- To create an Object we use a constructor, conversely to destroy an object we use a destructor.
- As there is a default constructor which the programmer does not define, there is also a default destructor which the programmer does not define.
- However it is also possible to define a default destructor
- This will either be called by the programmer or by the program when the object falls out of scope

Destroying a Dynamic Object

- To destroy a dynamic Object the destructor must be called.
- This is done by using the **delete** operator as follows

```
Point2D *point = new Point2D(100.0,12.0);

// now do some thing with the object

delete point; // now we destroy the object
```

Example

create a pointer to a Colour object

use new to create a new instance of the Colour class

invoke methods by using -> to de-reference

delete (call dtor) the object when finished

do it all again but with a different object (new memory location)

```
#include <iostream>
#include <cstdlib>

#include "Colour.h"

int main()
{

Colour *current;

current = new Colour(1,0,0);
current->Print();

delete current;

current = new Colour(1,1,1);
current->Print();

delete current;



}
```

A Scope Example

- The following example shows how several Objects are created and go in and out of scope

```
#include <string>
class Object
{
private :
         std::string name;
  static int ObjectCount;
public :
    Object(std::string namein);
  ~Object();
    int  getObjCount(void){return ObjectCount;}
};
```

A Scope Example

```
Object::Object(char * namein)
{
ObjectCount ++;
name=namein;
cout << "constructor called for "<< name << "
"<< ObjectCount << " Object Instances" <<endl;
}

Object::~Object()
{
delete [] name;
ObjectCount --;
cout << "destructor called for " << name << "
"<< ObjectCount<< " Object Instances" <<endl;
}

int Object::ObjectCount=0;
```

A Scope Example

```
#include <iostream.h>
#include "scope.h"

Object externalObject("external Object");

int main(void)
{
cout << "beginning of Main" << endl;
        {
        cout <<"now within the { " <<endl<<endl;
        Object AutoObject("Auto Object");
        static Object StaticObject("Static Object");
        }
cout <<"now out of the  } " << endl << endl;
Object * Objectptr = new Object("Dynamic Object");
delete Objectptr;
cout <<"end of main"<<endl;
}
```

Program Output

```
constructor called for external Object 1 Object
beginning of Main
now within the {
constructor called for Auto Object 2 Object
constructor called for Static Object 3 Object
destructor called for Auto Object 2 Object
now out of the  }
constructor called for Dynamic Object 3 Object
destructor called for Dynamic Object 2 Object
end of main
destructor called for Static Object 1 Object
destructor called for external Object 0 Object }
```

Why a destructor?

- In the previous example the destructor just printed out that it had been called
- The following example will show the real reason for the destructor
- The class allocates a block of dynamic memory when it is created
- When it is destroyed we need to free this memory so the destructor does this
- We also implement a “deep copy” constructor

mem.h

```
#ifndef __MEM_H__
#define __MEM_H_
class Mem
{
   public :
     Mem(
          int _size
        );
      Mem(
           const Mem &_m
        );
     ~Mem();
      void Print();
      void Set(int _offset,int _value);

  private :
     int *m_mem;
     int m_size;

};  #endif
```

mem.cpp

```
Mem::Mem(
        int _size
       )
{
  std::cout<<"ctor␣called\n";
  // allocate a new block of memory
  m_mem = new int[_size];
  // retain the size of the allocated block
  m_size=_size;
}



Mem::~Mem()
{
  std::cout<<"dtor␣called\n";
  if(m_mem !=0)
  {
    delete [] m_mem;
  }
}
```

mem.cpp

```
void Mem::Print()
{
  for(int i=0; i<m_size; ++i)
  {
    std::cout<<m_mem[i]<<std::endl;
  }
}




void Mem::set(
               int _offset,
               int _value
             )
{
  assert(_offset<m_size);
  m_mem[_offset]=_value;
}
```

The Singleton Pattern

- The singleton pattern defines an object that can only exist once
- This is done by implementing the code in a particular way with the object knowing if it has been created.
- If it has not it will create an instance of itself
- If it has been created it will return the instance.
- This pattern is a good way of storing global state data within a program

```
class Global
{
  public :
    /// @brief to return the instance of the class
    /// @returns the constructed class
    static Global* Instance();
    /// @brief mutator to set the name
    /// @param _name the value to set
    void SetName(
                    const std::string &_name
                  );
    /// @brief accesor for the name at
    /// @returns the name attribute
    std::string GetName();
    /// @brief mutator to set the age
    /// @param _age the age value to s
    void SetAge(
                  int _age
                 );
    /// @brief accesor for the age attribute
    /// @returns the age attribute
    int GetAge();

  private :
    /// @brief the instance of the class
    static Global* m_pinstance;
    /// @brief the name to be stored
    std::string m_name;
    /// @brief the age to be stored.
    int m_age;
    /// @brief private ctor so it can't
    inline ~Global(){;}
    /// @brief private dtor so it can't
    inline Global(){;}
    /// @brief private copy ctor so it can't be called
    inline Global(
                    const Global &_g
                  ){;}

};

#endif
```

Static instance of class

ctor's and dtor private

set the instance pointer to 0

```cpp
Global* Global::m_pinstance = 0;// initialize pointer

Global* Global::Instance()
{
  if (m_pinstance == 0)   // is it the first call?
  {
    m_pinstance = new Global; // create sole instance
  }
  return m_pinstance; // address of sole instance
}
```

if class doesn’t exist create

```cpp
void SetData()
{
  Global *data=Global::Instance();
  data->SetName("Jon");
  data->SetAge(40);
}

void PrintData()
{
  Global *data=Global::Instance();
  std::cout<<"Name = "<<data->GetName()<<"\n";
  std::cout<<"Age = "<<data->GetAge()<<"\n";
}

int main()
{
PrintData();
SetData();
PrintData();
return EXIT_SUCCESS;
}
```

In both functions
we grab the instance of
the class to use