CL1002 – Programming Fundamentals Lab

Lab # 7

Static Keyword

Instructor: Muhammad Saad Rashad

Email: saad.rashad@nu.edu.pk

Department of Computer Science,

National University of Computer and Emerging Sciences FAST

Peshawar

1. **Static Keyword:**

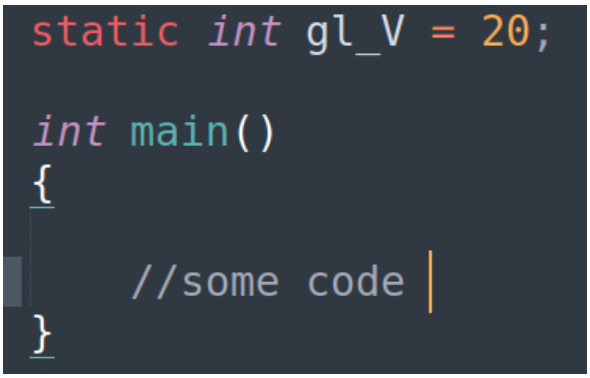
- The **static** keyword in C++ is used to define variables or functions that have a unique storage duration and scope. Depending on its context, it can mean:
    - **Static variable in a function:**
        - Retains its value between function calls
        - Initialized only once.

          Example 1:

```
void counter()
{
   static int st_v=0;
   st_v++;
   cout<<st_v<<"\n";
}
void func()
{
   counter();
}
int main()
{
   counter();
   func();
}
```

    - **Static global variable:**
        - Restricts variable scope to the file it is declared in (internal linkage).

          Example 2:



- **Static function:**
  - Limits function visibility to the same translation unit (same `.cpp` file)

Example 3:

- Let's create a header file in named "utils.h" a header file created by used has .h extension

  utils.h

```
#pragma once

static void helperFunction();
```

- This tells the compiler that `helperFunction()` exists somewhere but does not define it.

  pragma once:
  is a preprocessor directive that ensures a header file is included only once during compilation, preventing multiple inclusion errors.

- Now create a .cpp file to give definition to helperFunction.

  utils.cpp: This file defines the function declared in `utils.h`

```
#include <iostream>
#include "utils.h" //Includes the declaration

static void helperFunction() { // Internal linkage
    std::cout << "This function is private to this file.\n";
}
```

- Now create a main file which wants to use the utilities of our helper function.

  main.cpp

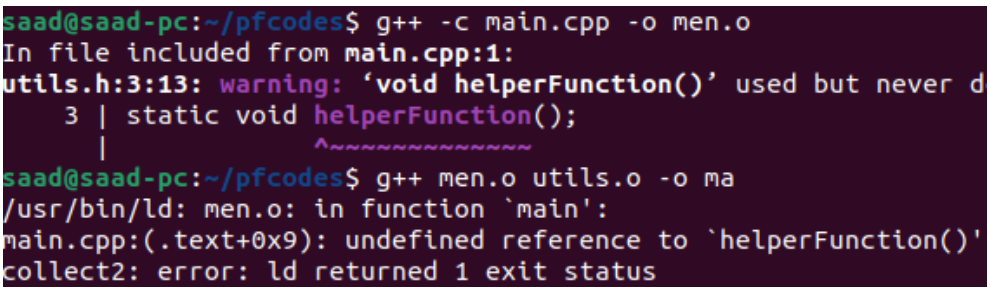
```
#include "utils.h"

int main()
{
    helperFunction();
}
```

`#include "utils.h"` allows the compiler to know that **`helperFunction()`** exists somewhere.

If you are using g++ compiler link the files like this:

- g++ -c utils.cpp -o utils.o
- g++ -c main.cpp -o main.o
- g++ main.o utils.o -o newfile
- ./newfile

The compiler will throw these errors when we try to execute a function that is internally linked even though we have included a header file.



2. **Static Keyword W.R.T class:**

○ **Static Data members:**

- If a data item in a class is defined as static, then only one such item is created for the entire class, no matter how many objects there are
- A static data item is useful when all objects of the same class must share the common item of information.
- Static data member is visible only within the class, but its life time is the entire program.
- A **static data member** of a class is defined outside the class because it belongs to the class itself, **not to any specific object**
- Only One Copy Exists: Since static members belong to the class, they must be allocated once and shared among all objects.
- Unlike normal members, static members are **not automatically initialized by the constructor**. They exist independently of objects, so their memory needs to be allocated explicitly.
  Example 4:

```
class check
{
private:
   static int count;

public:
   check();
   ~check();
   int getcount();
};
int check::count=0;
check::check()
{
   count++;
}
check::~check()
{
   cout<<"\n\n Object Destroyed\n\n";
}
int check::getcount()
{
   return count;
}
int main()
{
   check ob1,ob2,ob3;

   cout<<"count is: "<<ob1.getcount()<<"\n";
   cout<<"count is: "<<ob2.getcount()<<"\n";
   cout<<"count is: "<<ob3.getcount()<<"\n";
}
```

Output:

```
count is: 3
count is: 3
count is: 3

Object Destroyed

Object Destroyed
```

Object Destroyed

If we had used an ordinary automatic variable as opposed to static variable for count then output would have been :

count is 1
count is 1
count is 1

- **Static Member Functions:**

- A static function in a class:
- Belongs to the class, not objects
- Can access only static data members
- Cannot use **`this`** pointer (because it is not tied to any object)
- Can be called without creating an object

Example 5:

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

class check
{
private:
   static int total;
   int id;

public:
   check();
   ~check();
   void showid();
   static void showtotal();
};

int check::total=0;

check::check()
{
   total++;
   id=total;
}
check::~check()
```

```
{
   total--;
   cout<<"\nDestroying ID number: "<<id;
}
void check::showid()
{
   cout<<"\n ID Number is: "<<id;
}
void check::showtotal()
{
   cout<<"\nTotal is: "<<total;
}
int main()
{
   check ob1;
   check::showtotal();

   check ob2;
   check::showtotal();

   check ob3;
   check::showtotal();

   ob1.showid();
   ob2.showid();
   ob3.showid();
}
```

Output:

```
Total is: 1
Total is: 2
Total is: 3

 ID Number is: 1
 ID Number is: 2
 ID Number is: 3

Destroying ID number: 3
Destroying ID number: 2
Destroying ID number: 1
```

- In above program we could have created a dummy object like
  practice dummy;
  dummy.showtotal( );
- But this is rather awkward. We should not need to refer to the object when we are doing something that relates to the entire class.
- It is more reasonable to use the name of the class itself with the scope resolution operator (::). i.e.
  practice :: showtotal( );
- However this won't work if showtotal( ); is a normal member function.

Example 6: Passing a Static Member to a Normal Function

```
class check
{
private:
   static int count;

public:
   check();
   ~check();
   void printcount(int num);
   void display();
};
int check::count=0;
check::check()
{
   count++;
}
check::~check()
{
   cout<<"\nObject Destroyed\n";
}
void check::printcount(int num)
{
   cout<<"Count right now: "<<num;
}
void check::display()
{
   printcount(count);
}

int main()
```

```
{
   check ob1,ob2,ob3;

   ob1.display();cout<<"\n";
   ob2.display();cout<<"\n";
   ob3.display();cout<<"\n";

}
```

3. **Constructor/Destructor Calling control:**

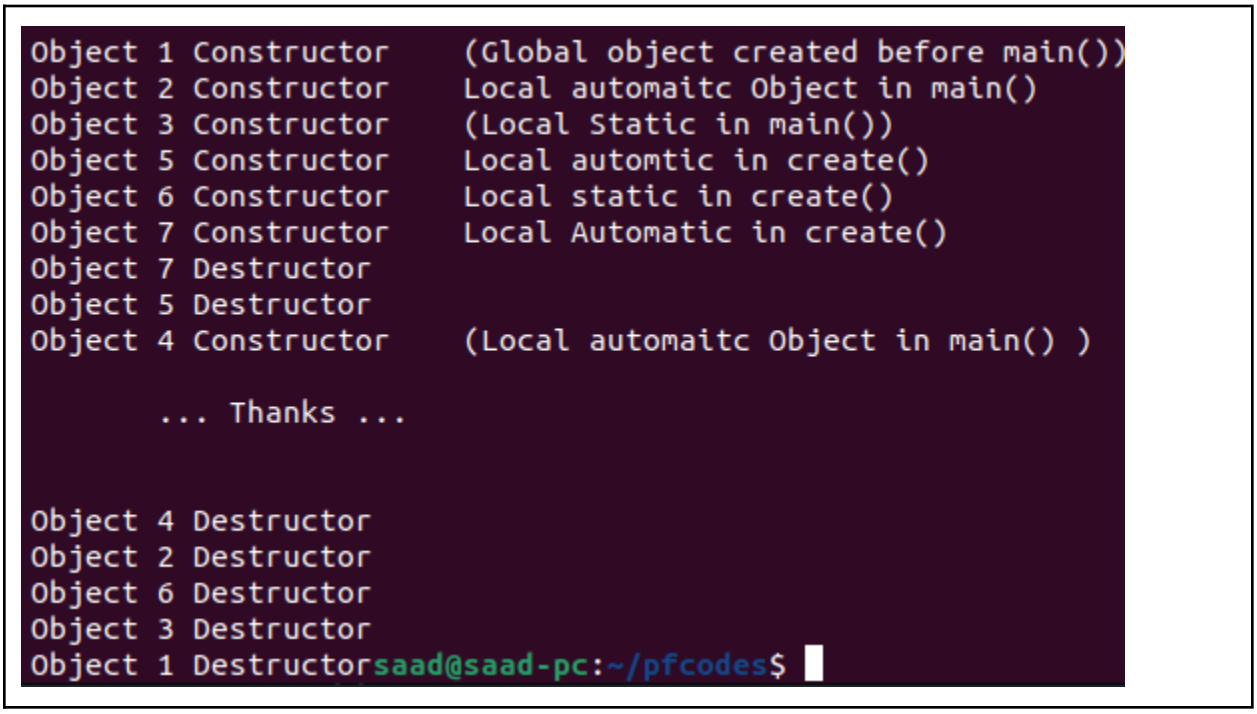
- Generally, destructor calls are made in the reverse order of the constructor calls.
- Constructors are called for objects defined in the global scope before any other function (including main( )) when the file begins execution
- The corresponding destructors for global objects are called when main( ) terminates or an exit( ) function is called.
- Constructors and destructors for automatic objects (not static) are called each time the objects enter and leave the scope.
- Constructors are called for static local objects only once when execution first reaches the point where the objects are defined.
- The corresponding destructors are called when main terminates or the exit( ) function is called in the reverse order.

Example 7:

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

class constdest
{
private:
   int data;
public:
   constdest(int);
   ~constdest();
};
constdest::constdest(int val)
{
   data = val;
```

```
    cout<<"\n Object "<<data<<" Constructor";
}
constdest::~constdest()
{
    cout<<"\nObject "<<data<<" Destructor";
}

void create();

constdest first(1); // Global Object of Class

int main()
{
    cout<<"\t (Global object created before main()) ";

    constdest second(2); //Local Obj of constdest class
    cout<<"\t Local automaitc Object in main()";

    static constdest third(3); //Local Object of constdest class
    cout<<"\t (Local Static in main()) ";

    create();

    constdest fourth(4);
    cout<<"\t (Local automaitc Object in main() ) ";//local obj of class

    cout<<"\n\n\t... Thanks ...\n\n";
}

void create()
{
    constdest fifth(5);
    cout<<"\t Local automtic in create()";

    static constdest sixth(6);
    cout<<"\t Local static in create()";

    constdest seventh(7);
    cout<<"\t Local Automatic in create()";

}
```

Output:



Tasks:

Task 1:

a. Create a Logger class that stores log messages persistently using static.
   - The Logger class will use a static array to store up to 10 messages.
   - A log() function will add messages to the array.
   - A printLogs() function will print all stored messages.

b. You will Create a Logger class with:
   - A **static array** (logs[10]) to store messages.
   - A **static integer** (count) to track the number of logs.

c. Implement log() to store messages in the array.

d. Implement printLogs() to print all logged messages.

e. In main(), log multiple messages and print them.

Task 2:

Imagine a Car class that tracks the maximum speed recorded during a race. The speed tracker should persist across different cars to keep track of the highest speed reached.

a. You will Create a Car class with:

- A **static integer** maxSpeed to store the highest recorded speed.
- A **normal integer** currentSpeed to store the speed of each car.

b. Implement a setSpeed(int speed) function that:

- Updates currentSpeed of the car.
- Updates maxSpeed if the new speed is greater.

c. Implement a getMaxSpeed() function to return the highest recorded speed.
d. In main(), create multiple Car objects and set their speeds.