Constant And Static

P.S. These slides are useless if you do not attend classes

Constant Variables

The keyword `const` be used to declare constant variables.

They must be initialized when they are declared and cannot be modified later.

Using constant variables to specify array size makes program more scalable.

Constant variables are also called `named constants` or `read-only variables`.

Constant Variables & Constant Parameters

```
int main()
{
    const int a = 5;
    const int b;  // will cause error
    b = 10;       // will cause error

    const int arr[] = {1, 2, 3, 4, 5};
    arr[0] = 10;  // will cause error
}
```

```
int func(const int a, int b)
{
    a += 10;     //will cause error
    b += 20;
}
```

Constant Member Functions

Constant member function is the function that cannot modify the data members.

To declare a constant member function, write the const keyword after the closing parenthesis of the parameter list. If there is separate declaration and definition, then the const keyword is required in both the declaration and the definition.

Constant member functions are used, so that accidental changes to objects can be avoided. A constant member function can be applied to a non-const object.

Keyword, const can't be used for constructors and destructors because the purpose of a constructor is to initialize data members, so it must change the object. Same goes for destructors.

Constant Member Functions

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

class Demo {
int val;

public:

Demo(int x = 0) {
val = x;
}

int getValue() const {
val=2; //[Error] assignment of member 'Demo::val' is read-only object
return val;
}

};
```

```
int main() {
Demo d1(8);
cout << "\nThe value using object d1 : " <<
d1.getValue();
return 0;
}
```

Constant Objects

As with normal variables we can also make class objects constant so that their value can't change during program execution. Constant objects can only call constant member functions. The reason is that only constant member function will make sure that it will not change value of the object. They are also called as read only objects. To declare constant object just write const keyword before object declaration.

Constant Objects

```
class Demo {
public:
int val;
Demo(int x = 0) {
val = x;}
int getValue() const {
return val; }
int getValue1() {
return val;}
};
```

```
int main() {
const Demo d(28);
cout << "The value using constant object d
: " << d.getValue();
cout << "\nThe value using object d non
const func : " << d.getValue1();//error
d.val=10;//error, can't modify const
objects
return 0;
}
```

Constant Objects

A constructor must be a non-const member function but it can still be used to initialize a const object) shows that it calls another non-const member function.

Invoking a nonconst member function from the constructor call as part of the initialization of a const object is allowed. The “constness” of a const object is enforced from the time the constructor completes initialization of the object until that object’s destructor is called.

Member Initialization List

Constant class members can only be initialized through constructor's member initialization list.

```
class A{
    const int x;
    const int y;
    public:
    A ( int val1 , int val2 ) {
    x=val1; // error
    x=val2;} // error
};
int main() {
    A a ( 5 , 10 );}
```

```
class A{
    const int x;
    const int y;
    public:
    A ( int val1 , int val2 ) : x ( val1 ) , y ( val2 )
    {}

};
int main() {
    A a ( 5 , 10 );}
```

Constant with Pointers

There are four ways to use const with pointers:

- Non-constant pointers to non-constant data
- Non-constant pointers to constant data
- Constant pointers to non-constant data
- Constant pointers to constant data

Non-constant Pointers to Non-constant Data

The highest access is granted by a non-constant pointer to non-constant data.

Data can be modified through pointer, and pointer can be made to point to other data.

```
int main()
{
    int a = 10;
    int b = 50;

    int* pA = &a;
    *pA = 20;
    pA = &b;
}
```

Non-constant Pointers to constant Data

Pointer can be modified to point to any other data, but the data to which it points cannot be modified through that pointer.

```
const  int * pVal;
```

```
int main()
{
    int a = 10;
    int b = 50;

    const int* pA = &a;
    *pA = 20;        // this line will cause error
    pA = &b;
}
```

Constant Pointers to Non-constant Data

Always points to the same memory location, but the data at that location can be modified through the pointer.

```
int * const pVal = &val;
```

```
int main()
{
    int a = 10;
    int b = 50;

    int* const pA = &a;
    *pA = 20;
    pA = &b;     // this line will cause error
}
```

Constant Pointers to Constant Data

Always points to the same memory location, and the data at that location cannot be modified via the pointer.

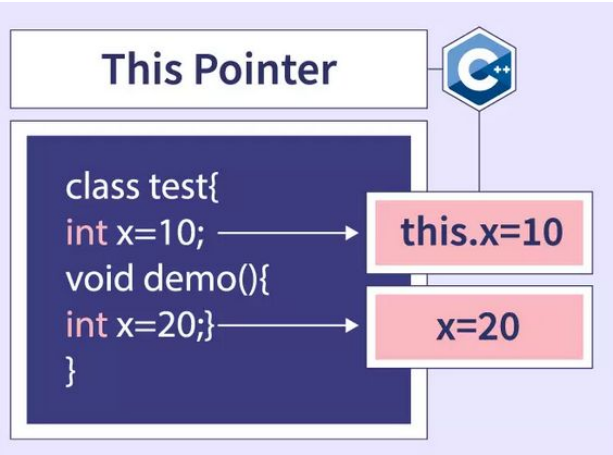
```
const int * const pVal = &val;
```

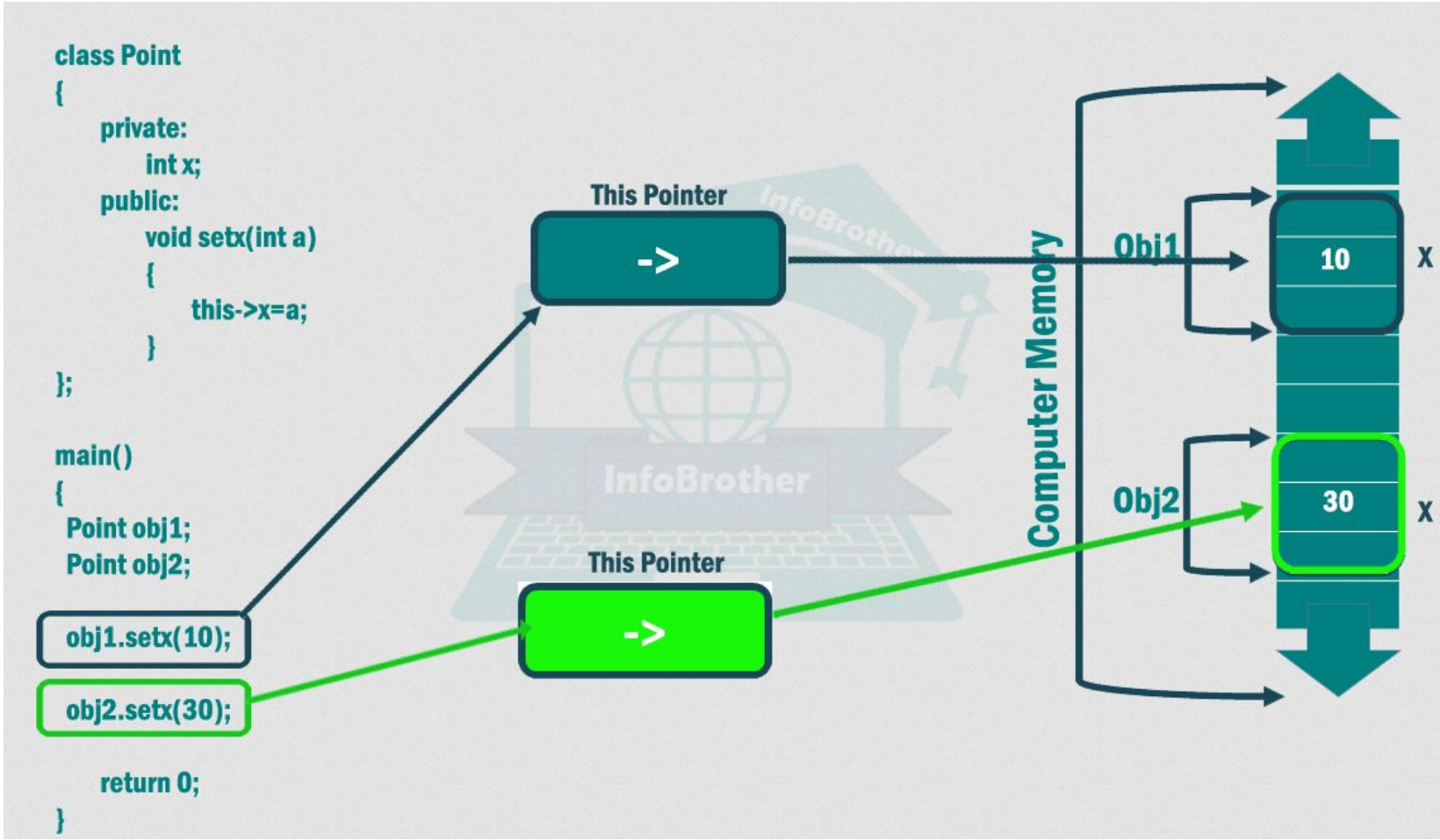
```
int main()
{
    int a = 10;
    int b = 50;

    const int* const pA = &a;
    *pA = 20;       // cannot do this
    pA = &b;    // cannot do this as well
}
```

This Pointer

You can access class members using By default, the compiler provides each member function of a class with an implicit parameter that points to the object through which the member function is called. The implicit parameter is this pointer.




class Point
{
private:
int x;
public:
void setx(int a)
{
this->x=a;
}
};
main()
{
Point obj1;
Point obj2;
obj1.setx(10);
obj2.setx(30);
return 0;
}
This Pointer
->
This Pointer
->
Computer Memory
Obj1
Obj2
10
30
X
X
InfoBrother

This Pointer

```
#include <iostream>
using namespace std;
class example{
private:
int x;
public:

void set(int x){
(*this).x = x;}

void printAddressAndValue(){
cout<<"The address is "<<this<<" and the value is "<<(*this).x<<endl;}
};

main(){
example e;
e.set(3);
e.printAddressAndValue();}
```

This Pointer

```
#include <iostream>
using namespace std;
class example{
private:
int x;
public:

void set(int x){
this->x = x;}

void printAddressAndValue(){
cout<<"The address is "<<this<<" and the value is "<<this->x <<endl;
}

};

main(){
example e;
e.set(3);
e.printAddressAndValue();}
```

Inline Functions

Placing the qualifier inline before a function's return type in definition "advises" the compiler to generate a copy of the function's code in place (when appropriate) to avoid a function call.

Compiler usually ignores this request unless the function does not have too much code.

```
inline void square(int a)
{
    cout << "Square of given number is: " << a * a;
}

int main()
{
    int a = 2;
    square(a);
}
```

Inline Functions

Compiler does not perform inlining when:

1) If a function contains a loop

2) If a function contains static variables

3) If a function is recursive

4) If a function return type is other than void, and the return statement doesn’t exist in function body

5) If a function contains switch or goto statement

Advantages of Inline Functions

1) Function call overhead doesn’t occur

2) It also saves overhead of a return call from a function

3) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return

Disadvantages of Inline Functions

1) The added variables from the inline function consumes additional registers.

2) If you use too many inline functions then the size of the binary executable file will be large, because of

the duplication of same code.

3) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled.

4) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

Storage Classes

To fully define a variable one needs to mention not only its ‘type’ but also its ‘storage class’.

There are basically two kinds of locations in a computer where such a value may be kept Memory and CPU registers.

Moreover, a variable’s storage class tells us:

(a) Where the variable would be stored.

(b) What will be the initial value of the variable.(i.e. the default initial value).

(c) What is the scope of the variable.

(d) What is the life of the variable; i.e. how long would the variable exist.

C++ Storage Class

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extem	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage

Static Storage Classes

Storage - Memory.

Default initial value - Zero.

Scope - Local to the block in which the variable is defined.

Life - Value of the variable persists between different function calls.

Static Storage Classes

```
void staticDemo()
{
    int val = 0;
    ++val;
    cout << “val = "  << val << endl;
}

int main()
{
    staticDemo();   // prints val = 1
    staticDemo();   // prints  val = 1
    staticDemo();   // prints val  = 1
}
```

```
void staticDemo()
{
    static int val = 0;
    ++val;
    cout << “val = "  << val << endl;
}

int main()
{
    staticDemo();   // prints val = 1
    staticDemo();   // prints  val = 2
    staticDemo();   // prints val  = 3
}
```

Static Data Member

A data member of a class can be qualified as static. It is initialized to zero when the first object of its class is created.

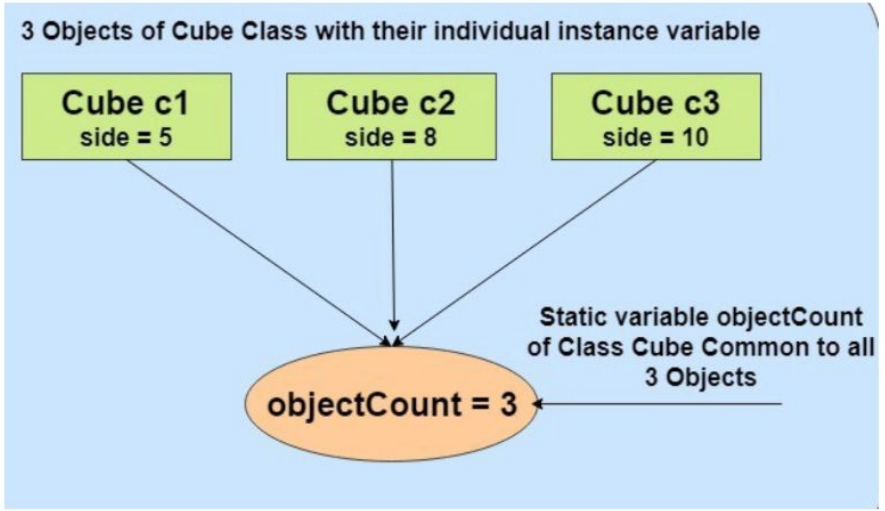
Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

It is visible only within the class, but its lifetime is the entire program.

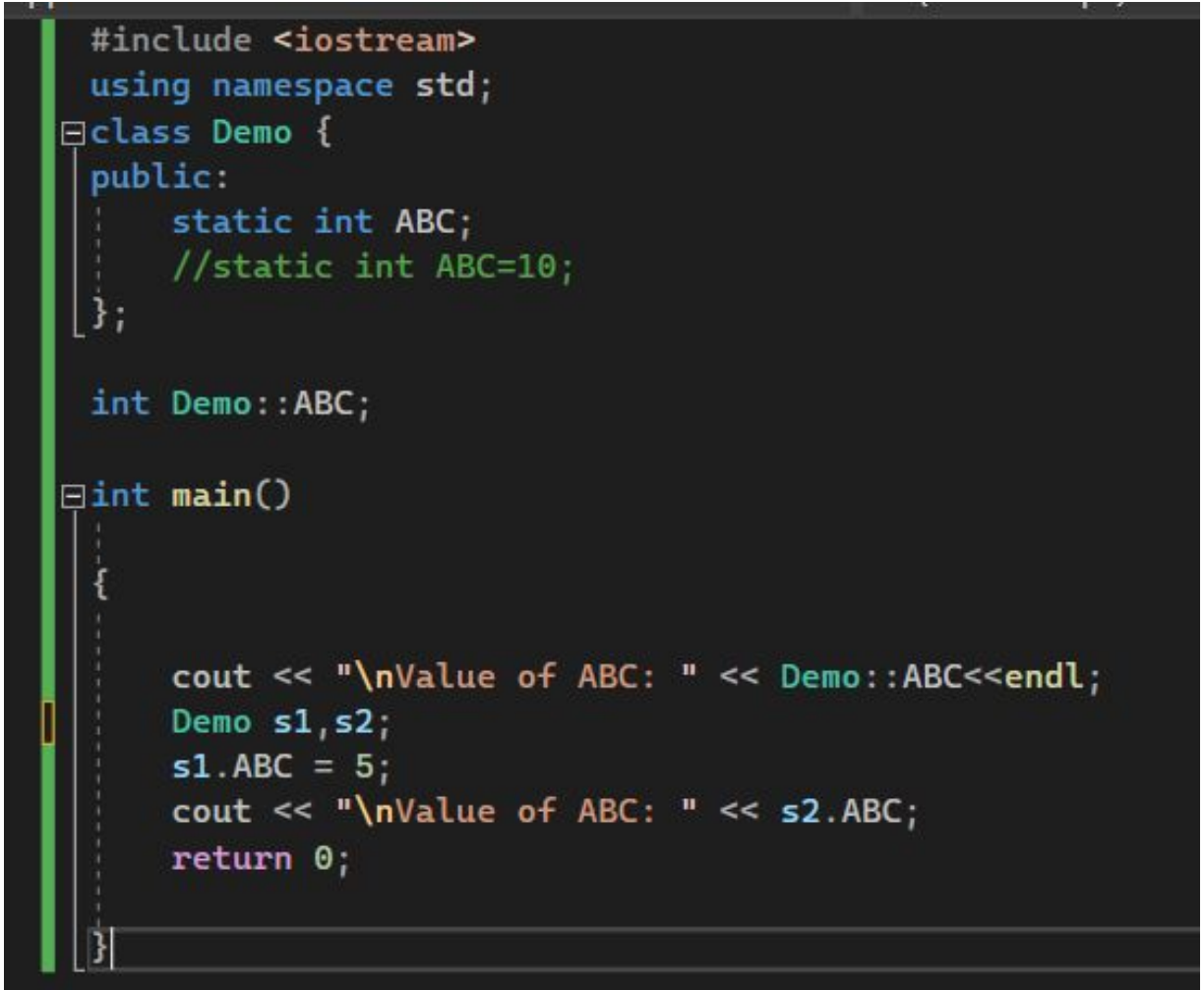
The type and scope of each static member variable must be declared outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object.

Static Data Member

It should be defined outside of the class following this syntax:




```
data_type class_name :: member_name =value;
```


#include <iostream>
using namespace std;
class Demo {
public:
static int ABC;
//static int ABC=10;
};
int Demo::ABC;
int main()
{
cout << "\nValue of ABC: " << Demo::ABC<<endl;
Demo s1,s2;
s1.ABC = 5;
cout << "\nValue of ABC: " << s2.ABC;
return 0;
}

Static Member Function

By declaring a function member as static, you make it independent of any particular object of the class.

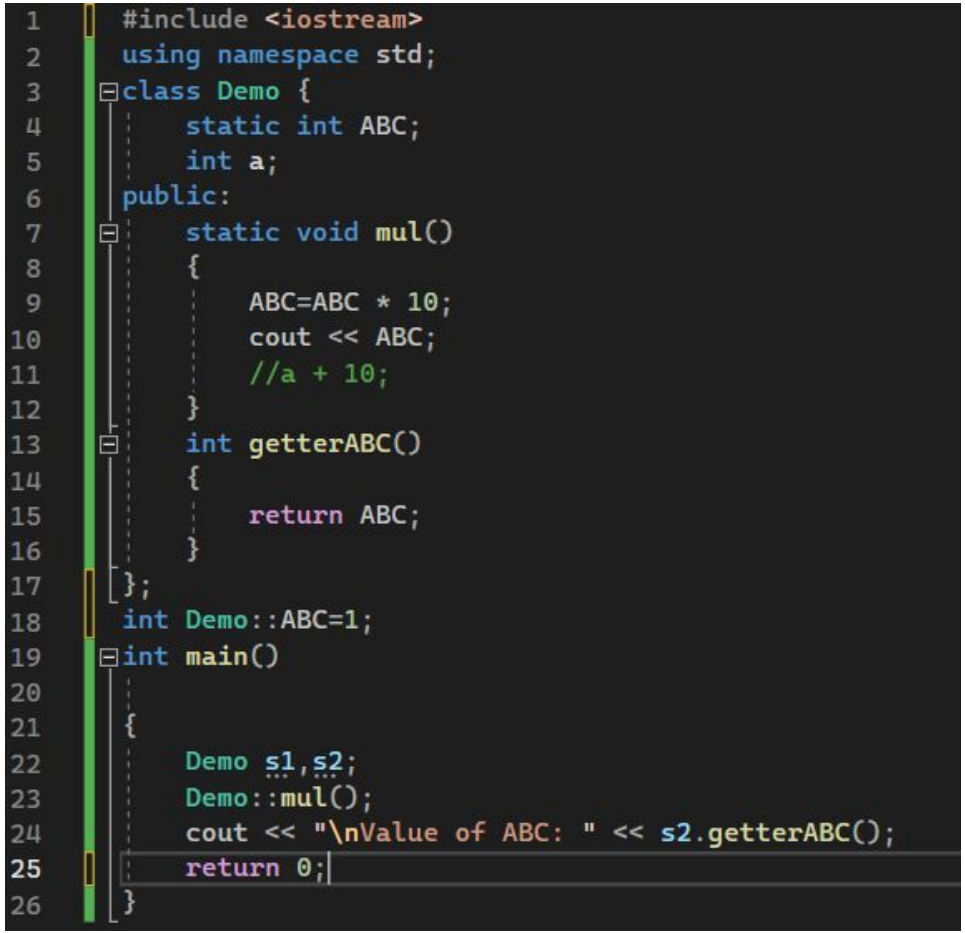
A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator ::.

```
Class_name::Function_name();
```

Static Member Function

A static member function can only access static data member, other static member functions and any other functions from outside the class.

A static member function can be called even if no objects of the class exist.



```
#include <iostream>
using namespace std;
class Demo {
    static int ABC;
    int a;
public:
    static void mul()
    {
        ABC=ABC * 10;
        cout << ABC;
        //a + 10;
    }
    int getterABC()
    {
        return ABC;
    }
};
int Demo::ABC=1;
int main()

{
    Demo s1,s2;
    Demo::mul();
    cout << "\nValue of ABC: " << s2.getterABC();
    return 0;
}
```

Concept: A static member function can only access static data member

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

class Student {
    public:
        int rollnumber, age;
        static int m;

    static modifyAge () //Error
    {
        age = 10;
    }
};
```

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

class Student {
    public:
        int rollnumber, age;
        static int m;

    static modifyAge () //Not an Error
    {
        m = 10;
    }
};
```

Concept: A static member function can only access other static member functions

```
1  #include <iostream>
2  using namespace std;
3
4  class Student {
5      public:
6          int rollnumber, age;
7          static int m;
8      void print()
9      {
10         cout<<"Hello"<<endl;
11     }
12     static modifyAge ()
13     {
14         m = 10;
15         print();  //Error
16     }
17 };
```

```
1  #include <iostream>
2  using namespace std;
3
4  class Student {
5      public:
6          int rollnumber, age;
7          static int m;
8      static void print()
9      {
10         cout<<"Hello"<<endl;
11     }
12     static modifyAge ()
13     {
14         m = 10;
15         print();  //Not an Error
16     }
17 };
```

Difference between a static variable and global variable

All global variables are static, but all static variables are not necessarily global.

Static variables have visibility limited to the function in which declared.

What all static variables (including global) have in common is that they have a lifetime that lasts the duration of the program. They are initialized just once, when the program begins.

Class Activity

Create a SavingsAccount class. Use a static data member annualInterestRate to store the annual interest rate for each of the savers. Each member of the class contains a private data member savingsBalance indicating the amount the saver currently has on deposit. Provide member function calculateMonthlyInterest that calculates the monthly interest by multiplying the balance by annualInterestRate divided by 12; this interest should be added to savingsBalance.

Provide a static member function modifyInterestRate that sets the static annualInterestRate to a new value.

Write a driver program to test class SavingsAccount. Instantiate two different objects of class SavingsAccount, saver1 and saver2, with balances of $2000.00 and $3000.00, respectively. Set the annualInterestRate to 3 percent. Then calculate the monthly interest and print the new balances for each of the savers. Then set the annualInterestRate to 4 percent, calculate the next month’s interest and print the new balances for each of the savers.

Automatic Storage Class

The auto storage class is the default storage class for all local variables.

Storage - Memory.

Default initial value - An unpredictable value, which is often called a garbage value.

Scope - Local to the block in which the variable is defined.

Life - Till the control remains within the block in which the variable is defined.

```
int count;   OR      auto int count;
```

Register Storage Class

Storage - CPU registers.

Default initial value - Garbage value.

Scope - Local to the block in which the variable is defined.

Life - Till the control remains within the block in which the variable is defined.

A value stored in a CPU register can always be accessed faster than the one that is stored in memory.

```
register int count;
```

External Storage Class

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to declare a global variable or function in another file.

Mutable Storage Class

It allows a member of an object to override const member function. That is, a mutable member can be modified by a const member function.

Mutable keyword is used with member variables of class, which we want to change even if the object is of const type. Hence, mutable data members of a const objects can be modified.

Mutable Storage Class

```
1  class Zee
2  {
3      int i;
4      mutable int j;
5      public:
6      Zee()
7      {
8          i = 0;
9          j = 0;
10     }
11
12     void fool() const
13     {
14         i++;    // will give error
15         j++;    // works, because j is mutable
16     }
17 };
18
19 int main()
20 {
21     const Zee obj;
22     obj.fool();
23 }
```

Activity

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

class Point
{
    int x, y;

    public:
    Point(int i = 0, int j =0)
    { x = i; y = j;  }

    int getX() const { return x; }

    int getY() {return y;}
};
```

```
int main()
{
    const Point t;
    cout << t.getX() << " ";
    cout << t.gety();
    getchar();
    return 0;
}
```

Activity

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

class Test
{
    static int x;

    public:

    Test() { x++; }

    static int getX() {return x;}
};

int Test::x = 0;
```

```
int main()
{
    cout << Test::getX() << " ";
    Test t[5];
    cout << Test::getX();
    getchar();
    return 0;
}
```

Pass By Reference Vs pass By Pointer

```
#include<iostream>
using namespace std;
class Test
{
private: int x; int y;
public:
Test(int x = 0, int y = 0)
{
this->x = x;
this->y = y;
}
Test& setX(int a)
{
x = a;

return *this;
}
Test& setY(int b)
{
y = b;
return *this;
}
void print()
{
cout<< "x = " << x << " y = " << y <<endl;
}
};
int main()
{
Test obj1(5, 5);
// Chained function calls. All calls modify the same
object
// as the same object is returned by reference
obj1.setX(10).setY(20).print();
}
```

```
x = 10 y = 20

--------------------------------
Process exited after 0.01273 seconds with return value 0
Press any key to continue . . .
```

Pass By Reference Vs pass By Pointer

```
#include <iostream>
using namespace std;
class thisExample {
    int test;
    char example;
    public:
        thisExample(int test, char example) {
            this->test = test;
            this->example = example;
        }
        void incTest() {
            test = test + 1;
        }

        void displayVars() {
            cout << "test: " << test << "\nexample: " << example <<endl;
        }

        thisExample* returnObj1() {
            return this; //return address of current ob
        }
        thisExample& returnObj2() {
            return *this; // returns refernce of current ob
        }
};
```

Pass By Reference Vs pass By Pointer

```
int main()
{

    thisExample tex(10, 'e');
    thisExample* TE1 = tex.returnObj1();
    TE1->incTest();
    TE1->displayVars();
    tex.displayVars();

    thisExample TE2 = tex.returnObj2();
    TE2.incTest();
    TE2.displayVars();
    tex.displayVars();

    cout << "------------------\n";

    tex.incTest();
    tex.incTest();
    tex.displayVars();
    TE1->displayVars();
    TE2.displayVars();
}
```

```
test: 11
example: e
test: 11
example: e
test: 12
example: e
test: 11
example: e
------------------
test: 13
example: e
test: 13
example: e
test: 12
example: e
```

Pass By Reference Vs pass By Pointer

```
int main()
{

    thisExample tex(10, 'e');
    thisExample* TE1 = tex.returnObj1();
    TE1->incTest();
    TE1->displayVars();
    tex.displayVars();

    thisExample &TE2 = tex.returnObj2();
    TE2.incTest();
    TE2.displayVars();
    tex.displayVars();

    cout << "------------------\n";

    tex.incTest();
    tex.incTest();
    tex.displayVars();
    TE1->displayVars();
    TE2.displayVars();
}
```

```
/tmp/DwBSdk88g2.o
test: 11
example: e
test: 11
example: e
test: 12
example: e
test: 12
example: e
------------------
test: 14
example: e
test: 14
example: e
test: 14
example: e
```

Class Activity

“Hotel Mercato” requires a system module that will help the hotel to calculate the rent of the customers. You are required to develop one module of the system according to the following requirements:

1) The hotel wants such a system that should have the feature to change the implementation independently of the interface. This will help when dealing with changingrequirements.
2) The hotel charges each customer 1000.85/- per day. This amount is being decided by the hotel committee and cannot be changed fulfilling certain complexformalities.
3) The module should take the customer’s name and number of days, the customer has stayed in the hotel as arguments in the constructor. The customer name must be initialized only once when the constructor iscalled. Any further attempts to change the customer’s name shouldfail.
4) The module then analyses the number of days. If the customer has stayed for more than a week in the hotel , he gets discount on the rent. Otherwise, he is being chargednormally.
5) The discounted rent is being calculated after subtracting one day from the total number ofdays.
6) In the end, the module displays the followingdetails:
a. Customer Name
b. Days
c. Rent

Note that, the function used for displaying purpose must not have the ability to modify any data member.

References

https://www.infobrother.com/Tutorial/C++/C++ Pointer Object

https://www.geeksforgeeks.org/passing-by-pointer-vs-passing-by-reference-in-c/