5.11 STATIC DATA MEMBERS

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are :

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- 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.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects. Program 5.4 illustrates the use of a static data member.

Program 5.4 Static Class Member

```
#include <iostream>

using namespace std;

class item
{
    static int count;
    int number;
  public:
    void getdata(int a)
    {
      number = a;
      count ++;
    }
    void getcount(void)
    {
      cout << "count: ";
      cout << count << "\n";
    }
};

// int item :: count;  → should reference memory for the

int main()
{
    item a, b, c;          // count is initialized to zero
    a.getcount();          // display count
    b.getcount();
    c.getcount();

    a.getdata(100);        // getting data into object a
    b.getdata(200);        // getting data into object b
    c.getdata(300);        // getting data into object c

    cout << "After reading data" << "\n";

    a.getcount();          // display count
    b.getcount();
    c.getcount();
    return 0;
}
```

The output of the Program 5.4 would be:

```
count: 0
```

```
After reading data
count: 3
count: 3
count: 3
```

Note that the type and scope of each **static** member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as *class variables*.

The **static** variable **count** is initialized to zero when the objects are created. The count is incremented whenever the data is read into an object. Since the data is read into objects three times, the variable count is incremented three times. Because there is only one copy of count shared by all the three objects, all the three output statements cause the value 3 to be displayed. Figure 5.4 shows how a static variable is used by the objects.

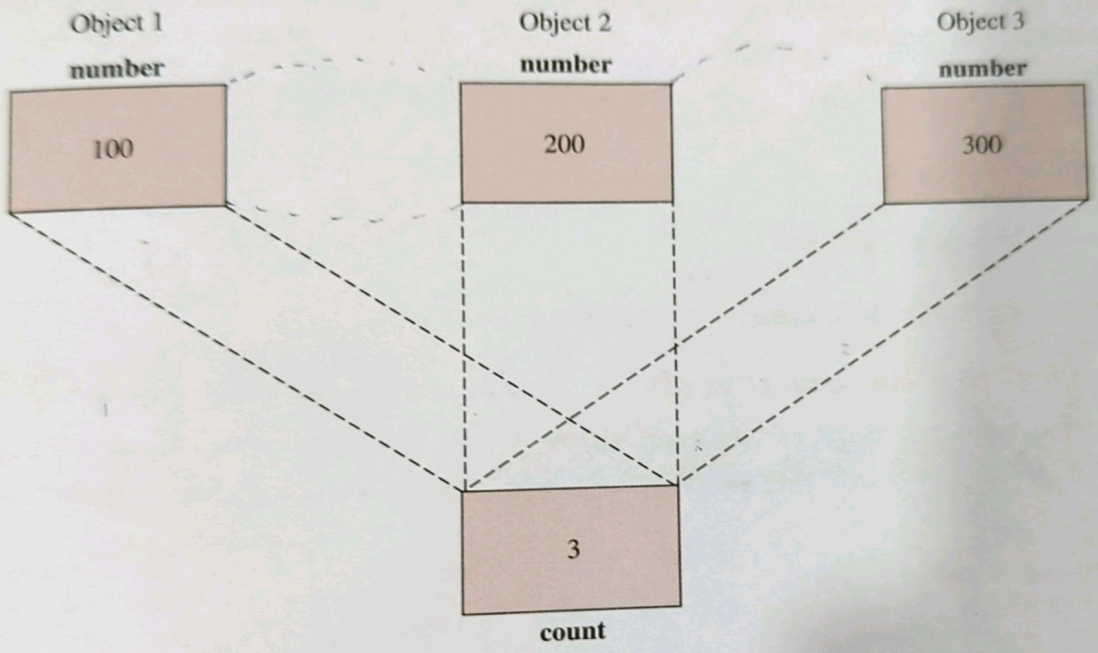

Fig. 5.4 *Sharing of a static data member*

Static variables are like non-inline member functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives count the initial value 10.

```
int item :: count = 10;
```

5.12 STATIC MEMBER FUNCTIONS

Like **static** member variable, we can also have **static** member functions. A member function that is declared **static** has the following properties:

- A **static** function can have access to only other static members (functions or variables) declared in the same class.
- A **static** member function can be called using the class name (instead of its objects) as follows:

```
class-name :: function-name;
```

Program 5.5 illustrates the implementation of these characteristics. The **static** function **showcount()** displays number of objects created till that moment. A count of number of objects created is maintained by the **static** variable.

The function **showcode()** displays the code number of each object.

Program 5.5 Static Member Function

```
#include <iostream>
using namespace std;
class test
{
    int code;
    static int count;                    // static member variable
public:
    void setcode(void)
    {
    code = ++count;
    }
    void showcode(void)
    {
    cout << "object number: " << code << "\n";
    }
    static void showcount(void)          // static member function
    {
    cout << "count: " << count << "\n";
    }
};
int test :: count;
int main()
{
    test t1, t2;

    t1.setcode();
    t2.setcode();

    test :: showcount();                 // accessing static function

    test t3;
    t3.setcode();

    test :: showcount();

    t1.showcode();
    t2.showcode();
    t3.showcode();

    return 0;
}
```

The output of Program 5.5 would be:

```
count: 2
count: 3
object number: 1
object number: 2
object number: 3
```

Note *Note that the statement*

```
code = ++count;
```

is executed whenever **setcode( )** *function is invoked and the current value of* **count** *is assigned to* **code**. *Since each object has its own copy of* **code**, *the value contained in* **code** *represents a unique number of its object.*

Remember, the following function definition will not work:

```
static void showcount()
{
        cout << code;    // code is not static
}
```

4.7 DEFAULT ARGUMENTS

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a *default* val to the parameter which does not have a matching argument in the function call. Default values are specified when t function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts t program for possible default values. Here is an example of a prototype (i.e., function declaration) with default values

static variable can be used to set the value at the starting of the program which value can be changed later on.
// It can be used in the shopping cart program where u no longer need to make seperate function to set the initial item list to 0.


```
float amount(float principal,int period,float rate=0.15);
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument **rate**. A subsequent function call like

```
value = amount(5000,7);        // one argument missing
```

passes the value of 5000 to **principal** and 7 to **period** and then lets the function use default value of 0.15 for rate. The call

```
value = amount(5000,5,0.12);   // no missing argument
```

passes an explicit value of 0.12 to **rate**.

A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from *right to left*. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

In function declaration: right to left
In function call: left to right.

format

```
int mul(int i, int j=5, int k=10);        // legal
int mul(int i=5, int j);                  // illegal
int mul(int i=0, int j, int k=10);        // illegal
int mul(int i=2, int j=5, int k=10);      // legal
```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation. Program 4.2 illustrates the use of default arguments.

4.10 FUNCTION OVERLOADING

As stated earlier, *overloading* refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as *function polymorphism* in OOP.

Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded **add()** function handles different types of data as follows:

```
// Declarations
int add(int a, int b);                    // prototype 1
int add(int a, int b, int c);             // prototype 2
double add(double x, double y);           // prototype 3
double add(int p, double q);              // prototype 4
double add(double p, int q);              // prototype 5

// Function calls
cout << add(5, 10);                       // uses prototype 1
cout << add(15, 10.0);                    // uses prototype 4
cout << add(12.5, 7.5);                   // uses prototype 3
cout << add(5, 10, 15);                   // uses prototype 2
cout << add(0.75, 5);                     // uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

   ```
   char to int
   float to double
   ```

   to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

   ```
   long square(long n)
   double square(double x)
   ```

   A function call such as

   ```
   square(10)
   ```

   will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square()** should be used.
4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Program 4.5 illustrates function overloading.

Program 4.5 Function Overloading

```
//Function area() is overloaded three times
#include<iostream>

//Declaration of Function Prototypes
int area (int);
int area (int, int);
float area (float);

int main()
{
    cout<< "Calling the area() function for computing the area of a
    (side = 5) - " <<area(5)<<"\n";
    cout<< "Calling the area() function for computing the area of a
    rectangle (length = 5, breadth = 10) - "<<area(5,10)<<"\n";
    cout<< "Calling the area() function for computing the area of a
    (radius = 5.5) - " <<area(5.5);
return 0;
}

int area (int side)                          //Area of square
{
  return(side*side);
}

  int area (int length, int breadth)     //Area of rectangle
{
  return(length*breadth);
}

  float area (float radius)
{                                          //Area of circle
  return(3.14*radius*radius);
}
```

The output of Program 4.5 would be:

```
Calling the area() function for computing the area of a square
(side = 5) - 25
Calling the area() function for computing the area of a rectangle (length =
5, breadth = 10) - 50
Calling the area() function for computing the area of a circle
(radius = 5.5) - 94.99
```

Classes

C++ also permits us to define another user-defined data type known as **class** which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming. Other details about classes are discussed in Chapter 5.

Enumerated Data Type

An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The **enum** keyword (from C) automatically enumerates a list of words by assigning them values 0,1,2, and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the **struct** statement. Examples:

```
enum shape{circle, square, triangle};
enum colour{red, blue, green, yellow};
enum position{off, on};
```

The enumerated data types differ slightly in C++ when compared with those in ANSI C. In C++, the tag names **shape**, **colour**, and **position** become new type names. By using these tag names, we can declare new variables. Examples:

```
shape ellipse;          // ellipse is of type shape
colour background;      // background is of type colour
```

ANSI C defines the types of **enums** to be **ints**. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an **int** value to be automatically converted to an **enum** value. Examples:

```
colour background = blue;          // allowed
colour background = 7;             // Error in C++
colour background = (colour) 7;    // OK
```

However, an enumerated value can be used in place of an **int** value.

```
int c = red;      // valid, colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can over-ride the default by explicitly assigning integer values to the enumerators. For example,

```
enum colour{red, blue=4, green=8};
enum colour{red=5, blue, green};
```

are valid definitions. In the first case, **red** is 0 by default. In the second case, **blue** is 6 and **green** is 7. Note that the subsequent initialized enumerators are larger by one than their predecessors.

C++ also permits the creation of anonymous **enums** (i.e., **enums** without tag names). Example:

```
enum{off, on};
```

Here, **off** is 0 and **on** is 1. These constants may be referenced in the same manner as regular constants. Examples:

```
int switch_1 = off;
int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a **switch** statement. Example:

```
enum shape
{
    circle,
```

```
    rectangle,
    triangle
};
int main()
{
    cout << "Enter shape code:";
    int code;
    cin >> code;
    while(code >= circle && code <= triangle)
    {
        switch(code)
        {
            case circle:
            ......
            ......
            break;
            case rectangle:
            ......
            ......
            break;
            case triangle:
            ......
            ......
            break;
        }
        cout << "Enter shape code:";
        cin >> code;
    }
    cout << "BYE \n";

    return 0;
}
```

Fact: Use system("cls"); to clear console in C++ and C, but in order to do so you may need a header file in C ⟶ <stdlib.h>, in C++ ⟶ <cstdlib>. stdlib stands for Standard library.

Circle → 0
Rectangle → 1
Triangle → 2

Expected output

```
Enter shape code:0
 Circle
Enter shape code:1
 Rectangle
Enter shape code:2
 triangle
Enter shape code:3
 Bye
```

cin.ignore(); is used under <cstdlib> header file use to clear input buffer

ANSI C permits an **enum** to be defined within a structure or a class, but the **enum** is globally visible. In C++, an **enum** defined within a class (or structure) is local to that class (or structure) only.

3.7 STORAGE CLASSES

In addition to the data type, a variable, also has a storage class associated with it. The storage class of a variable specifies the lifetime and visibility of a variable within the program. Lifetime refers to the longevity of a variable or the duration till which a variable remains active during program execution. Visibility, on the other hand, signifies the scope of a variable, i.e., in which program modules the variable is accessible. There are four types of storage classes, as explained below:

auto It is the default storage class of any type of variable. Its visibility is restricted to the function in which it is declared. Further, its lifetime is also limited till the time its container function is executing. That is, it is created as soon as its declaration statement is encountered and is destroyed as soon the program control leaves its container function block.

↳ Local variable

Tip : Dynamic Allocation means the value itself given example memory address.

extern As the name suggests, an external variable is declared outside of a function but is accessible inside the function block. Also called global variable, its visibility is spread all across the program. This means, it is accessible by all the functions present in the program. The lifetime of an external variable is same as the lifetime of a program.

static A static variable has the visibility of a local variable but the lifetime of an external variable. This means, once declared inside a function block, it does not get destroyed after the function is executed, but retains its value so that it can be used by future function calls. → function call variable.

register Similar in behavior to an automatic variable, a register variable differs in the manner in which it is stored in the memory. Unlike, automatic variables that are stored in the primary memory, the register variables are stored in CPU registers. The objective of storing a variable in registers is to increase its access speed, which eventually makes the program run faster. However, it may be noted that it is not an obligation for a compiler to store a register type variable only in CPU registers, but if there are no registers vacant to accommodate the variable then it is stored just like any other automatic variable. ↳ register based variable.

Table 3.4 gives a summary of the four storage classes.

Table 3.4 *Storage classes*

	auto	*extern*	*static*	*register*
Lifetime	Function block	Entire program	Entire program	Function block
Visibility	Local	Global	Local	Local
Initial Value	Garbage	0	0	Garbage
Storage	Stack segment	Data segment	Data segment	CPU registers
Purpose	Local variables used by a single function	Global variables used throughout the program	Local variables retaining their values throughout the program	Variables using CPU registers for storage purpose
Keyword	auto	extern	static	Register