



# Const and Classes

- const can be used with function arguments to keep a function from modifying a variable passed to it by reference.



# const Member Functions

- A const member function guarantees that it will never modify any of its class's member data.

```
class aClass
{
private:
int alpha;
public:
void nonFunc() //non-const member function
{ alpha = 99; } //OK
void conFunc() const //const member function
{ alpha = 99; } //ERROR: can't modify a member
};
```



# const Member Functions

- A function is made into a constant function by placing the keyword `const` after the declarator but before the function body. If there is a separate function declaration, `const` must be used in both declaration and definition.
- **A Distance Example**



# const Member Function Arguments

- ▶ A function is made into a constant function by placing the keyword `const` after the declarator but before the function body. If there is a separate function declaration, `const` must be used in both declaration and definition.
- ▶ **A Distance Example**



# const Objects

- When an object is declared as const, you can't modify it. It follows that you can use only const member functions with it, because they're the only ones that guarantee not to modify it.



## Q

Imagine a tollbooth at a bridge. Cars passing by the booth are expected to pay a 50 cent toll. Mostly they do, but sometimes a car goes by without paying. The tollbooth keeps track of the number of cars that have gone by, and of the total amount of money collected.

Objects and Classes Model this tollbooth with a class called tollBooth. The two data items are a type unsigned int to hold the total number of cars, and a type double to hold the total amount of money collected. A constructor initializes both of these to 0. A member function called payingCar() increments the car total and adds 0.50 to the cash total. Another function, called nopayCar(), increments the car total but adds nothing to the cash total. Finally, a member function called display() displays the two totals. Make appropriate member functions const.

Include a program to test this class. This program should allow the user to push one key to count a paying car, and another to count a nonpaying car. Pushing the Esc key should cause the program to print out the total cars and total cash and then exit.



# This pointer

The member functions of every object have access to a sort of magic pointer named `this`, which points to the object itself. Thus any member function can find out the address of the object of which it is a member.

```
#include <iostream>
using namespace std;
////////////////////////////////////
class where
{
private:
char charray[10]; //occupies 10 bytes
public:
void reveal()
{ cout << "\nMy object's address is " << this; }
};
////////////////////////////////////
int main()
{
where w1, w2, w3; //make three objects
w1.reveal(); //see where they are
w2.reveal();
w3.reveal();
cout << endl;
return 0;
}
```

The `main()` program in this example creates three objects of type `where`. It then asks each object to print its address, using the `reveal()` member function. This function prints out the value of the `this` pointer. Here's the output:

```
My object's address is 0x8f4effec
My object's address is 0x8f4effe2
My object's address is 0x8f4effd8
```

# Accessing Member Data with this

When you call a member function, it comes into existence with the value of this set to the address of the object for which it was called. The this pointer can be treated like any other pointer to an object, and can thus be used to access the data in the object it points to,

```
#include <iostream>
using namespace std;
class what
{
private:
int alpha;
public:
void tester()
{
this->alpha = 11; //same as alpha = 11;
cout << this->alpha; //same as cout << alpha;
}
};
.
```

```
int main()
{
what w;
w.tester();
cout << endl;
return 0;
}
```

This program simply prints out the value 11. Notice that the tester() member function accesses the variable alpha as

this->alpha

This is exactly the same as referring to alpha directly. This syntax works, but there is no reason for it except to show that this does indeed point to the object.



```
#include <string>
#include <iostream>
using namespace std;
class personType
{
public:
    personType()
    {}
    void print() const;
    void setName(string first, string last);
    personType setFirstName(string first);
    personType setLastName(string last);
    personType(string a, string b)
    { firstName=a; lastName=b; }
private:
    string firstName; //variable to store the first name
    string lastName; };
    personType personType::setLastName(string last)
    { lastName = last; return *this; }
    personType personType::setFirstName(string first)
    { firstName = first; return *this; }
    void personType::print() const
    {cout << firstName << " " << lastName;}
}
```

```
int main()
{
    personType student1 ("Angela", "Smith"); //Line 1
    personType student2;
    personType student3;
    student1.print(); //Line 5
    student2.setFirstName("Shelly").setLastName("Malik");
    student2.print();
    student3.setFirstName("Chelsea");
    student3.print();
    student3.setLastName("Tomek");
    student3.print();
}
```



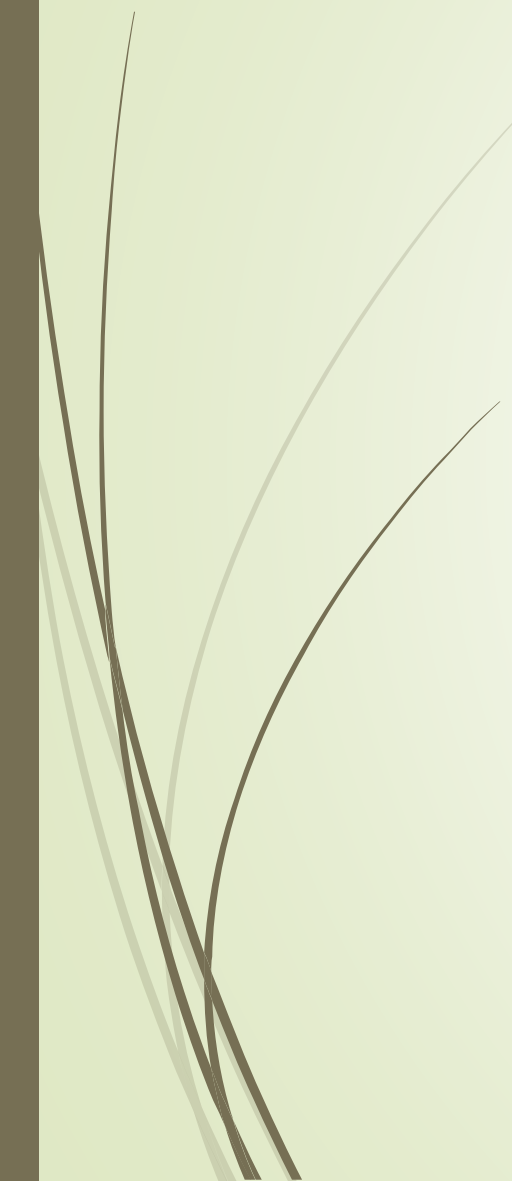

# Using this for Returning Values

- we could not return an object by reference, because the object was local to the function returning it and thus was destroyed when the function returned.
- An object's member functions are created and destroyed every time they're called, but the object itself endures until it is destroyed
- Thus returning by reference the object of which a function is a member is a better bet than returning a temporary object created in a member function. The this pointer makes this easy.

# Cascading Member Functions in C++

```
class Square
{
public:
    int side;
    Square area()
    {
        cout << "Area of the square is :" << side*side << endl;
        return *this;
    }
    Square perimeter()
    {
        cout << "Perimeter of the square is :" << 4*side << endl;
        return *this;
    }
};

int main()
{
    Square sq;
    sq.side = 3;
    //cascading function calls
    sq.area().perimeter();
}
```



As you can see, we have created a class Square and defined functions area() and perimeter() for it. The function area() prints the area of the square and the function perimeter() prints the perimeter of the square. \*this pointer specifies that the functions return a reference to the object which makes cascading possible.

# C++ Inline Functions

- When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

# C++ Inline Functions

- C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.
- The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
    // function code
}
```



# C++ Inline Functions

- Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:
  - If a function contains a loop. (for, while, do-while)
  - If a function contains static variables.
  - If a function is recursive.

## **Inline functions provide following advantages:**

- 1) Function call overhead doesn't occur.
- 2) It also saves the overhead of push/pop variables on the stack when function is called.
- 3) It also saves overhead of a return call from a function.



# C++ Inline Functions

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
    return s*s*s;
}
int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
} //Output: The cube of 3 is: 27
```



# C++ Inline Functions

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

inline void displayNum(int num) {
    cout << num << endl;
}

int main() {
    // first function call
    displayNum(5);

    // second function call
    displayNum(8);

    // third function call
    displayNum(666);

    return 0;
}
```




# C++ Inline Functions in class


```
class S
{
    public:
        int square(int s); // declare the function
};

inline int S::square(int s) // use inline prefix
{
}
```



# Static variables

- **Static variables in a Function:** When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call.
- 



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

```
void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";



    // value is updated and
    // will be carried to next
    // function calls
    count++;
}
```

```
int main()
{
    for (int i=0; i<5; i++)
        demo();
    return 0;
}
```

**Output:**

**0 1 2 3 4**

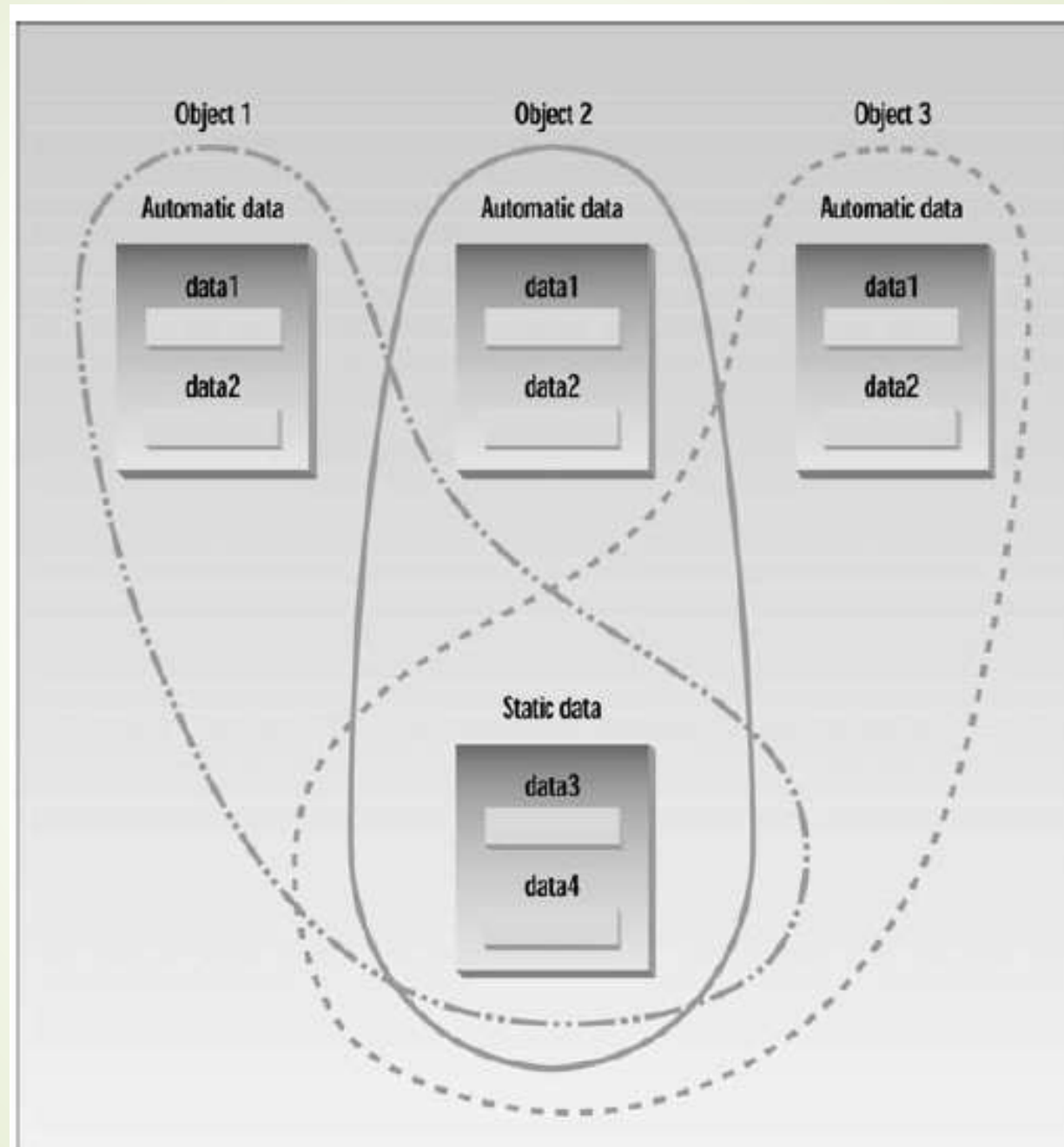


- 
- 
- You can see in the above program that the variable count is declared as static. So, its value is carried through the function calls. The variable count is not getting initialized for every time the function is called.



# Static Class members

- ▶ If a data item in a class is declared as static, 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 a common item of information.
- ▶ It continues to exist even if there are no objects of the class.
- ▶ Static class member data is used to share information among the objects of a class.
- ▶ Static member data requires an unusual format. Ordinary variables are usually declared (the compiler is told about their name and type) and defined (the compiler sets aside memory to hold the variable) in the same statement.
- ▶ Static member data, on the other hand, requires two separate statements. The variable's declaration appears in the class definition, but the variable is actually defined outside the class, in much the same way as a global variable.





# Static Data Members in C++

- ➡ Static data members are class members that are declared using the static keyword. There is only one copy of the static data member in the class, even if there are many class objects. This is because all the objects share the static data member. The static data member is always initialized to zero when the first class object is created.

# Static Data Members in C++

```
#include <iostream>
#include <string.h>
using namespace std;
class Student {
    private:
        int rollNo;
        char name[10];
        int marks;
    Public:
        static int objectCount;
    public:
        Student() {
            objectCount++;
        }
        void getdata() {
            cout << "Enter roll number: " << endl;
            cin >> rollNo;
            cout << "Enter name: " << endl;
            cin >> name;
            cout << "Enter marks: " << endl;
            cin >> marks;
        }
};
```

```
void putdata() {
    cout << "Roll Number = " << rollNo << endl;
    cout << "Name = " << name << endl;
    cout << "Marks = " << marks << endl;
    cout << endl;
}

};
int Student::objectCount = 0;
int main(void) {
    Student s1;
    s1.getdata();
    s1.putdata();
    Student s2;

    s2.getdata();
    s2.putdata();
    Student s3;
    s3.getdata();
    s3.putdata();
    cout << "Total objects created = " <<
    Student::objectCount << endl;
}
```



# Static Data Members in C++

- Static data members are class members that are declared using static keywords. A static member has certain special characteristics. These are:
  - 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 initialized before any object of this class is being created, even before main starts.
  - It is visible only within the class, but its lifetime is the entire program





# Why we use Static variable

- In a road-racing game, for example, a race car might want to know how many other cars are still in the race. In this case a static variable count could be included as a member of the class. All the objects would have access to this variable. It would be the same variable for all of them; they would all see the same count.

# Static Data Members in C++

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

class A
{
public:
    A() { cout << "A's Constructor Called " << endl; }
};

class B
{
    static A a;
public:
    B() { cout << "B's Constructor Called " << endl; }
};

int main()
{
    B b;
}
```

The above program calls only B's constructor, it doesn't call A's constructor. The reason for this is simple, static members are only declared in a class declaration, not defined. They must be explicitly defined outside the class using the scope resolution operator.

# Static Data Members in C++

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

class A
{
    int x;
public:
    A() { cout << "A's constructor called " << endl; }
};

class B
{
    static A a;
public:
    B() { cout << "B's constructor called " << endl; }
};

A B::a; // definition of a

int main()
{
    B b1, b2, b3;
}
```

# Static Member Function

```
#include <iostream>
using namespace std;
class A
{
    int x;
public:
    A() { cout << "A's constructor called " << endl; }
};
class B
{
    static A a;
public:
    B() { cout << "B's constructor called " << endl; }
    static A getA() { return a; }
};
A B::a; // definition of a
int main()
{
    B b1, b2, b3;
    A a = b1.getA();
}
```

# Static Member Function

```
#include <iostream>
using namespace std;
class A
{
    int x;
public:
    A() { cout << "A's constructor called " << endl; }
};
class B
{
    static A a;
public:
    B() { cout << "B's constructor called " << endl; }
    static A getA() { return a; }
};

A B::a; // definition of a
int main()
{
    // static member 'a' is accessed without any object of B
    B::getA();
}
```