SCOPE RESOLUTION OPERATOR

::

Different Functions …

ACTIVITY

```
Value of the Variable a: 0
Calling from the Func Function:
Variable a is 100
Variable b is 67
```

```
1  #include<iostream>
2  using namespace std;
3
4   int a=3;
5
6  void func (int x) {
7       int b, a=100;
8           b=x;
9               cout<<"Calling from the Func Function: "<<endl;
10              cout<<"Variable a is "<<a<<endl;
11              cout<<"Variable b is "<<b<<endl;
12  }
13
14 int main()
15 {
16      int a=0;
17      cout<<"Value of the Variable a: " <<a<<endl;
18      func(67);
19 }
```

SCOPE OF VARIABLES

- The extent of the program code within which the variable can be accessed or declared or worked with.
- **Local Variables**
  - Variables defined within a function or block are local to those functions.
    - Declared inside a block.
  - Local variables **do not exist** outside the block
    - **Can not** be accessed or used outside that block.
- **Global Variables**
  - Can be accessed from any part of the program.
  - Available throughout the lifetime of a program.
  - Declared at the top of the program outside all of the functions or blocks.

SCOPE OF VARIABLES...

- Usually two variables with the same name are NOT allowed to be defined.
  - Within the SCOPE
- If the variables are defined in different scopes then the compiler allows it.
- If there is a local variable and a global variable with the same name
  - The **compiler gives precedence to the local variable**
- **How can you ACCESS the global variable???**
  - Use the **scope resolution operator**.
  - SCOPE RESOLUTION OPERATOR ■ ■ ■ ■

TO ACCESS THE GLOBAL VARIABLE

Scope Resolution Operator

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

 int a=3;

 void func (int x)
 {
    int b, a=100;
        b=x;
            cout<<"Calling from the Func Function: "<<endl;
            cout<<"Variable a is "<<a<<endl;
            cout<<"Variable b is "<<b<<endl;
 }

int main()
{
    int a=0;

    cout<<"Value of the Local Variable a in Main: " <<a<<endl;
    cout<<"Value of the Global Variable a " <<::a<<endl;
    func(67);
}
```

```
Value of the Local Variable a in Main: 0
Value of the Global Variable a 3
Calling from the Func Function:
Variable a is 100
Variable b is 67
```

ACTIVITY

ACTIVITY

```
40
3
X-a = 2

1
```

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

int a=3;

class X{
    int a;

    public:

    void setValue(int x){
        a=x;
    }

    int increment_a(){
        return (a++);
    }

    void printX(){
        cout<<"X-a = "<<a<<endl;}
}b;
```

```
int main()
{
    int a=40;
    b.setValue(0);

    cout<<a<<endl;
    cout<<::a<<endl;

    a=b.increment_a();
    a=b.increment_a();
    b.printX();

    cout<<endl<<a<<endl;
}
```

SCOPE RESOLUTION OPERATOR

Used to

- **access** a **global variable** when there is a local variable with the same name,
- **define** a function **outside a class**.

Activity

- **access** elements having the same name inside **two namespaces**
  - Use the namespace name with the scope resolution operator to refer to that class

    using namespace std; & cout<< OR

    std::cout

TO ACCESS ELEMENTS HAVING THE SAME NAME INSIDE TWO NAMESPACES

Scope Resolution Operator

NAMESPACE

- A space where we can define or declare identifiers i.e. variables, methods, and classes.
- Can define Functions, classes, variables having the same name in different libraries.
- Syntax:

```
namespace  namespace_name
{
    // code declarations i.e. variable  (int a;)
    method (void add();)
    classes ( class student{};)
}
```

Calling the namespace

namespace_name: :code; // code could be variable , function or class.

ACTIVITY

```
Inside second_space
Inside first_space
```

```cpp
#include <iostream>
using namespace std;
// first name space
namespace first_space{
    void func(){
        cout << "Inside first_space" << endl;
    }
}

// second name space
namespace second_space{
    void func(){
        cout << "Inside second_space" << endl;
    }
}

using namespace second_space;
int main ()
{
    func();
    first_space::func();
}
```

SCOPE RESOLUTION OPERATOR

Used to

- **access** a **global variable** when there is a local variable with same name,
- **define** a function **outside a class**.
- **access** elements having the same name inside **two namespaces**
  - Use the namespace name with the scope resolution operator to refer to that class

    using namespace std; & cout<< OR

    std::cout
- **refer** to a **class inside** another **class**

ACTIVITY

```
1  // Use of scope resolution operator: class inside another class.
2  #include<iostream>
3  using namespace std;
4
5  class outside
6  {
7  public:
8      int x=0;
9
10     class inside
11     {
12     public:
13             int x = 1;
14             int y = 56;
15             int foo(){
16                 return x;
17             }
18     };
19 };
```

```
21 int main(){
22     outside A;
23     outside::inside B;
24     cout<<B.foo()<<endl;
25     cout<<B.y<<endl;
26 }
```

```
1
56
```

TO REFER TO A CLASS INSIDE ANOTHER CLASS

Scope Resolution Operator

ACTIVITY

```
2   #include<iostream>
3   using namespace std;
4
5   class outside
6   {
7   public:
8       int x=0;
9
10      class inside
11      {
12      public:
13              int x = 1;
14              int y = 56;
15              int foo(){
16                  return x;
17              }
18      };
19  };
20
```

```
21  int main(){
22      outside A;
23      outside::inside B;
24          cout<<B.foo()<<endl;
25          cout<<B.y<<endl;
26          cout<<A.y;
27  }
28
```

9 - ...	In function 'int main()':
Obj...	[Error] 'class outside' has no member named 'y'

ACTIVITY

```
 2  #include<iostream>
 3  using namespace std;
 4
 5  class outside
 6  {
 7  public:
 8      int x=0;
 9
10      class inside
11      {
12      public:
13              int x = 1;
14              static int y;
15              int foo(){
16                  return x;
17              }
18      };
19  };
```

```
22  outside C; //Global object
23
24  int outside::inside::y = 5;
25
26  int main(){
27
28      outside A;  //local object
29      outside::inside B;
30
31      cout<<B.foo()<<endl;
32      cout<<outside::inside::y<<endl;
33      cout<<B.y<<endl;
34      cout<<C.x<<endl;
35  }
```

```
1
5
5
0
```

SCOPE RESOLUTION OPERATOR

Used to

- **access** a **global variable** when there is a local variable with same name,
- **define** a function **outside a class**.
- **access** elements having the same name inside **two namespaces**
  - Use the namespace name with the scope resolution operator to refer to that class

    using namespace std; & cout<< OR

    std::cout
- **refer** to a **class inside** another **class**
- **access** a class's **static variables**.
- **access variables** with the same name in two ancestor classes in **multiple inheritance**.

ACTIVITY

```
1  #include<iostream>
2  using namespace std;
3
4  class Enclosing {
5  int x=0;
6  /* start of Nested class declaration */
7  class Nested {
8      int y=1;
9  }; // declaration Nested class ends here
10
11 void EnclosingFun(Nested *n) {
12         cout<<n->y;
13         cout<<x<<endl;
14         cout<<y<<endl;
15 }
16 }; // declaration Enclosing class ends here
17
18 int main()
19 {
20
21 }
```

DYNAMIC MEMORY ALLOCATION

DYNAMIC MEMORY ALLOCATION

- Static Memory allocation: Global and Local variables
  - The memory space is allocated by the compiler.
  - Done before the program executes.
- Can one explicitly allocate and deallocate space at *run-time?*
  - *YES / WHY?*
  - Advantages:
    - No need to know how much amount of memory is needed beforehand.
    - *Dynamically* allocate exactly the correct amount of space for the variables
    - Use the memory space more efficiently…etc.
  - Use a pre-defined keyword

    new to allocate storage and

    delete to deallocate the storage

DYNAMIC MEMORY ALLOCATION: *NEW/DELETE*

```
int main()
{
  // Allocated memory  dynamically.
  int *intPtr1 = new int;
  int *intArrayPtr2 = new int[10];

  // Dynamically allocated memory is deallocated
  delete intPtr1;
  delete [] intArrayPtr2;
}
```

'new' allocates storage for the object and returns a pointer to the allocated storage

Good Coding Practise: Free dynamic memory when no longer required. Use the **delete** operator to release dynamic memory

//Don't forget the square brackets [] for an array

DYNAMIC MEMORY ALLOCATION: *NEW/DELETE*

- Always check whether the memory has been successfully allocated

```
#include <cassert>
Time *timearrayPtr = new Time[10];
assert(timeArrayPtr != 0);
```

- **new** returns a null pointer if the allocation is unsuccessful