ACTIVITY

- Define a class to create a new datatype called Date (Should include day, month and year)
- Define two functions called
  - setDate to initialize values by passing parameters
  - getDate to print the value (Date) on the console
- Create a variable called "today" of data type Date
- Set the value to 28/November/2024
- Display the value on the console

CLASS: DATE

```cpp
1   #include<iostream>
2   using namespace std;
3
4   class date
5   {
6       int day;
7       string month;
8       int year;
9
10      void setDate(int D,string M,int Y){
11          day = D;
12          month = M;
13          year = Y;
14      }
15
16      void getDate(){
17          cout<<"Day:"<<day<<" Month:"<<month<<" Year:"<<year<<endl;
18      }
19  }; // end of class definition
20
```

CLASS AS A DATA TYPE

- Once a class has been declared, it can be used as object/array declarations

Example:

```
Car carObj,            // object of type Car
  Car carArray[10];    // array of Car objects
  Car *newCar;         // Object pointer
```

The class name becomes the new type specifier.

DATA MEMBERS AND MEMBER FUNCTIONS

Accessing Data members / Member functions

- DOT (.) Operator

  - Example: MyObject.variableX;   MyObject.func();

- Refer to a member of a class via a pointer to an object, (->) Operator

A *pointer* variable, or simply a *pointer*, contains the location, or address in the memory, of a memory cell

ACTIVITY

- Define a class to create a new datatype called Date (Should include day, month and year)
- Define two functions called
  - setDate to initialize values by passing parameters
  - getDate to print the value (Date) on the console
- Create a variable called "today" of data type Date
- Set the value to 28/November/2024
- Display the value on the console
- **Create a pointer variable called ptrDate to assign the value of "today"**
- **Display the value of ptrDate**

ACCESS OPERATOR (->): EXAMPLE

```
int main()
{
 date today;
 today.setDate(5,"December",2024);

        date *ptrDate;
        ptrDate=&today;

        cout<<"CHECK POINTER"<<endl;
        ptrDate->getDate();
}
```

Date today;

0x6ffdd0

DATA MEMBERS AND MEMBER FUNCTIONS

Accessing Data members / member functions

- DOT (.) Operator

  - Example: MyObject.variableX;   MyObject.func();

- Refer to a member of a class via a pointer to an object, (->) Operator

- Accessing a data member depends on the access modifier of the data member.

Defining Member functions in classes

- Inside class definition

- Outside class definition

  - use the scope resolution :: operator along with class name and function name.

```cpp
class Date{
    int day;
    string month;
    int year;

    public:
    void setDate(int,string,int);
    void getDate();
};

void Date::setDate(int d,string m,int y){
    day=d;
    month=m;
    year=y;
}

void Date::getDate(){
    cout<<"Day: "<<day<<" Month: "<<month<<" Year: "<<year<<endl;
}
```

DEFINING MEMBER FUNCTIONS

- The scope resolution operator (::) can be used to define member functions outside the class definition.

- Member functions defined inside a class definition are automatically treated as inline functions.

- Member functions that contain more than one statement should be defined outside their class definition.

- Members of a class have class scope – data members can be referred to directly inside the member functions from that class.

SCOPE RESOLUTION OPERATOR

::

Different Functions …

TO ACCESS THE GLOBAL VARIABLE

Scope Resolution Operator

ACTIVITY

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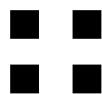
```
Value of the Variable a: 0
Calling from the Func Function:
Variable a is 100
Variable b is 67
```

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 ■ ■

```cpp
1    #include<iostream>
2    using namespace std;
3
4     int a=3;
5
6     void func (int x)
7     {
8         int b, a=100;
9             b=x;
10                cout<<"Calling from the Func Function: "<<endl;
11                cout<<"Variable a is "<<a<<endl;
12                cout<<"Variable b is "<<b<<endl;
13     }
14
15   int main()
16   {
17        int a=0;
18
19        cout<<"Value of the Local Variable a in Main: " <<a<<endl;
20        cout<<"Value of the Global Variable a " <<::a<<endl;
21        func(67);
22   }
```

```
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
```

```cpp
1   #include<iostream>
2   using namespace std;
3
4   int a=3;
5
6   class X{
7       int a;
8
9       public:
10
11      void setValue(int x){
12          a=x;
13      }
14
15      int increment_a(){
16          return (a++);
17      }
18
19      void printX(){
20          cout<<"X-a = "<<a<<endl;}
21  }b;
```

```cpp
24  int main()
25  {
26      int a=40;
27      b.setValue(0);
28
29      cout<<a<<endl;
30      cout<<::a<<endl;
31
32      a=b.increment_a();
33      a=b.increment_a();
34      b.printX();
35
36      cout<<endl<<a<<endl;
37  }
```

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

52

TO ACCESS ELEMENTS HAVING THE SAME NAME INSIDE TWO NAMESPACES

Scope Resolution Operator

SCS1310

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
1   #include <iostream>
2   using namespace std;
3   // first name space
4   namespace first_space{
5       void func(){
6           cout << "Inside first_space" << endl;
7       }
8   }
9
10  // second name space
11  namespace second_space{
12      void func(){
13          cout << "Inside second_space" << endl;
14      }
15  }
16
17  using namespace second_space;
18  int main ()
19  {
20      func();
21      first_space::func();
22  }
```

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**

TO REFER TO A CLASS INSIDE ANOTHER CLASS

56

Scope Resolution Operator

ACTIVITY

```cpp
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   };
```

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

```
1
56
```

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
```

```
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<<A.y<<endl;
34       cout<<B.v<<endl:
35
36   }
```

9 - ...   In function 'int main()':

Obj...   [Error] 'class outside' has no member named 'y'

ACTIVITY

```cpp
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  };
```

```cpp
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

```cpp
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

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

}
```

'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

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