NAMESPACES IN C++

Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area if they live in different area or their mother or father name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz and there is another library available which is also having same function xyz. Now the compiler has no way of knowing which version of xyz function you are referring to within your code.

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

Defining a Namespace:

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name {
   // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
name::code;  // code could be variable or function.
```

Let us see how namespace scope the entities including variable and functions:

```
#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;
   }
}
int main ()
{

   // Calls function from first name space.
   first_space::func();

   // Calls function from second name space.
   second_space::func();

   return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Inside first_space
Inside second_space
```

The using directive:

You can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code:

```
#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 first_space;
int main ()
{

   // This calls function from first name space.
   func();

   return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Inside first_space
```

The using directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows:

```
using std::cout;
```

Subsequent code can refer to cout without prepending the namespace, but other items in the **std** namespace will still need to be explicit as follows:

```
#include <iostream>
using std::cout;

int main ()
{

   cout << "std::endl is used with std!" << std::endl;

   return 0;
}
```

If we compile and run above code, this would produce the following result:

```
std::endl is used with std!
```

Names introduced in a **using** directive obey normal scope rules. The name is visible from the point of the **using** directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

Discontiguous Namespaces:

A namespace can be defined in several parts and so a namespace is made up of the sum of its separately defined parts. The separate parts of a namespace can be spread over multiple files.

So, if one part of the namespace requires a name defined in another file, that name must still be declared. Writing a following namespace definition either defines a new namespace or adds new elements to an existing one:

```
namespace namespace_name {
   // code declarations
}
```

Nested Namespaces:

Namespaces can be nested where you can define one namespace inside another name space as follows:

```
namespace namespace_name1 {
   // code declarations
   namespace namespace_name2 {
      // code declarations
   }
}
```

You can access members of nested namespace by using resultion operators as follows:

```
// to access members of namespace_name2
using namespace namespace_name1::namespace_name2;

// to access members of namespace:name1
using namespace namespace_name1;
```

In the above statements if you are using namespace_name1, then it will make elements of namespace_name2 available in the scope as follows:

```
#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 first_space::second_space;
int main ()
{

   // This calls function from second name space.
   func();

   return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Inside second_space
```