10.1 Namespace

A namespace declaration identifies and assigns a name to a declarative region. The identifier in a namespace declaration must be unique in the declarative region in which it is used. The identifier is the name of the namespace and is used to reference its members. The declarative region of a namespace declaration is its namespace-body. A namespace is a mechanism for expression logical grouping. That is, if some declarations logically belong together according to some criteria, they can be put in a common namespace to express that fact.

They are used to avoid name clashes. Namespaces are intended to express logical structure. The simplest such structure is the distinction between code written by one-person v/s code written by someone else. When we use only a single global scope, it is unnecessarily difficult to compose a program out of separate parts. The problem is that the separate parts each define the same names. When combined into the same program, these names clash.

This would not be a problem, if the program were tested separately, however during the application integration, such errors would surface.

C++ language provides a single global namespace. This can cause problems with global name clashes. For instance, consider these two C++ header files:

```
// one.h
char func(char);
class String {};

// somelib.h
class String {};
```

With these definitions, it is impossible to use both header files in a single program; the String classes will clash.

A namespace is a declarative region that attaches an additional identifier to any names declared inside it. The additional identifier makes it less likely that a name will conflict with names declared elsewhere in the program. It is possible to use the same name in separate namespaces without conflict even if the names appear in the same translation unit. As long as they appear in separate namespaces, each name will be unique because of the addition of the namespace identifier. For example:

```
// one.h
namespace one
{
   char func(char);
   class String {};
}

// somelib.h
namespace SomeLib
{
   class String {};
}
```

Now the class names will not clash because they become one **::** String and SomeLib **::** String, respectively. Declarations in the file scope of a translation unit, outside all namespaces, are still members of the global namespace.

Members of a namespace may be defined within that namespace. For example:

```
namespace X
{
   void f() {}
}
```

Members of a named namespace can be defined outside the namespace in which they are declared by explicit qualification of the name being defined. However, the entity being defined must already be declared in the namespace. In addition, the definition must appear after the point of declaration in a namespace that encloses the declaration's namespace. For example:

```
namespace Q
{
   namespace V
   {
      void f();
   }

   void V :: f() {}         // ok
   void V :: g() {}         // error, g() is not yet a member of V

   namespace V
   {
      void g();
   }
}
```

If we give our namespace short names, the names of different namespaces will clash.

```
namespace A {}

A::string s1 = "hello";
A::string s2 = "world";
```

However, long namespaces can be impractical in real application code:

```
namespace genesis_insoft_limited {}

genesis_insoft_limited :: string s3 = "hello";
genesis_insoft_limited :: string s4 = "world";
```

This dilemma can be resolved by providing a short alias for a longer namespace name:

```
namespace GIS = genesis_insoft_limited;
GIS :: string s3 = "hello";
GIS :: string s4 = "world";
```

A namespace is a scope. The standard scope rules hold for namespaces, so if a name is previously declared in the namespace or in an enclosing scope, it can be used without any problem. A name from another namespace can be used when qualified by the name of its namespace.

A namespace-alias-definition declares an alternate name for a namespace. The identifier is a synonym for the qualified-namespace-specifier and becomes a namespace-alias.

A namespace-name cannot be identical to any other entity in the same declarative region. In addition, a global namespace-name cannot be the same as any other global entity name in a given program.

10.1 Code Listing

```
# include <iostream.h>

int Gl_Var = 1;
int& fun(int&);

// A namespace declaration, identifier name is GlobalSpace1
namespace GlobalSpace1
{
   int Gl_Var = 2 ;     // Member of GlobalSpace1
   int fun(int&);       // Member function of GlobalSpace1
}

// A namespace declaration, identifier name is GlobalSpace2
namespace GlobalSpace2
{
   int Gl_Var = 3;      // Member of GlobalSpace2
   int fun(int&);       // Member function of GlobalSpace2
}

// An extension-namespace-definition for GlobalSpace1
namespace GlobalSpace1
{
   int Set_Var(int);
}

// 'Alias1' is an alias name for GlobalSpace1
namespace Alias1 = GlobalSpace1;

// Implementation of member of GlobalSpace1
int Alias1 :: Set_Var(int Set_to_Gl_Var)
{
   return Gl_Var = Set_to_Gl_Var;
}

// Implementation of member of GlobalSpace1
int GlobalSpace1 :: fun(int& Ref_to_Arg)
{
   return Ref_to_Arg++ + Gl_Var;       // Uses GlobalSpace1 :: Gl_Var
}

// Implementation of member of GlobalSpace2
int GlobalSpace2 :: fun (int& Ref_to_Arg)
{
   return Ref_to_Arg++ + ::Gl_Var;     // Uses GlobalSpace :: Gl_Var
}

// Implementation of member of global space
int& fun(int& Ref_to_Arg)
{
   GlobalSpace1 :: Gl_Var += GlobalSpace2 :: Gl_Var + ++Ref_to_Arg;
   return GlobalSpace1 :: Gl_Var;
}

int main()
{
   // Calls Globally declared function 'fun' and argument global variable 'Gl_Var'
   cout << fun(:: Gl_Var) << endl ;
```

```
   // Calls GlobalSpace1 member function 'fun' and argument is GlobalSpace2 member 'Gl_Var'
   cout << GlobalSpace1 :: fun(GlobalSpace2 :: Gl_Var) << endl;

   // Calls GlobalSpace2 member function 'fun' and argument is GlobalSpace1 member 'Gl_Var'
   cout << GlobalSpace2 :: fun (GlobalSpace1 :: Gl_Var) << endl;

   // Calls GlobalSpace1 member function 'Set_Var'
   cout << GlobalSpace1 :: Set_Var(-2) << endl;

   // Calls GlobalSpace1 member function 'fun' since 'Alias1' is an alternative name   to GlobalSpace1
   cout << Alias1 :: fun(Alias1 :: Gl_Var) <<endl;

   return 0;
}
```

10.2 Code Listing

```
# include <iostream.h>

namespace F
{
   float x = 9;
}

namespace G
{
   using namespace F;
   float y = 2.0;
   namespace INNER_G
   {
      float z = 10.01;
   }
}

int main()
{
   using namespace G;              // this directive gives you everything declared in "g"
   using namespace G :: INNER_G;   // this directive gives you only "inner_g"
   float x = 19.1;                 // local declaration takes precedence

   cout << "x = " << x << endl;    // local variable
   cout << "y = " << y << endl;    // In G
   cout << "z = " << z << endl;    // In G :: Inner

   return 0;
}
```

10.3 Code Listing

```
# include <iostream.h>
# include <time.h>

class DATE
{
   private:
      int   Day, Mon, Year;

   public:
      DATE() {}
      DATE(int dd, int mm, int yy) : Day(dd), Mon(mm), Year(yy) {}

      int day()
```

```
      {
         return Day;
      };

      int mon()
      {
         return Mon;
      };

      int year()
      {
         return Year;
      };

      void SetCurrDate();
      friend ostream& operator << (ostream& , DATE &);
};

void DATE :: SetCurrDate()
{
   time_t a;
   struct tm *t;

   a = time(NULL);
   t = localtime(&a);

   Day = t->tm_mday;
   Mon = t->tm_mon + 1;
   Year = t->tm_year + 1900;
}

ostream& operator << (ostream& O, DATE & d)
{
   O << " Date : " << d.Day << "/" << d.Mon<<"/" << d.Year;
   return O;
}

namespace American
{
   void format(DATE &d)
   {
      cout << endl << " American Date : "<< d.mon() << "/";
      cout << d.day() << "/" << d.year();
   }
}

namespace British
{
   void format(DATE &d)
   {
      cout << endl << " British Date : " << d.day();
      cout << "/" << d.mon() << "/" << d.year();
   }
}

void main()
{
   DATE d;
   d.SetCurrDate();

   cout<<endl<<" Current "<<d;

   American :: format(d);
   British :: format(d);
}
```

10.2 Cast operators

C++ supports four new cast operators
- static_cast
- const_cast
- dynamic_cast
- reinterpret_cast

In C, we typecast using the following statement:
(type) expression

In C++, we would write the same as
static_cast <type> (expression)

For example, suppose we want to cast an int to double, using C-style casts, the operation would be as follows:

```
int num1, num2;
double result = ((double)num1) / num2;
```

In C++, it would be written as

```
double result = static_cast <double> (num1) / num2;
```

static_cast has the same power and meaning as the general purpose C-style cast. It has the same kind of restrictions. For example, we can't cast a struct into a double or a double into a pointer using a static_cast.

const_cast is used to cast away the constness or volatileness of an expression. The only thing the const_cast allows us to change is the constness or volatileness of something.
Imagine we have a function init() which takes a non-const object. To this function if we pass a const object it would result in an error. To overcome this problem, we change the constness of the object.

```
void init(account *pa);
const account a1;
```

// Error. Can't pass a const object to a function that takes a non-const object.

```
init(&a1);
```

To overcome this problem, we write the same as

```
init(const_cast <account *> (&a1));
```

dynamic_cast is primarily used to perform safe casts down or across the inheritance hierarchy. It is used to cast pointers or references to base class objects into pointers or references to derived or sibling class in such a way that you can determine whether the casts succeeded.

```
class shape {}
class circle : public shape {}
shape *sp;
```

/* We are passing to draw(), a pointer to circle sp. If sp points to one it is fine, otherwise null pointer is passed. */

```
draw(dynamic_cast <circle *>(sp));
```

dynamic_cast are useful for navigating inheritance hierarchy. They cannot be applied to types lacking virtual functions, nor can they cast away constness.

```
int num1, num2;

// Error, as no inheritance is involved
double result = dynamic_cast <double> (num1) / num2;
const account a1;

// Error, as dynamic_cast can't cast away constness
init(dynamic_cast <account *> (&a1));
```

reinterpret_cast is used to perform type conversions whose result is nearly always implementation defined, hence they are rarely portable. It is commonly used to cast between function pointer types. For example, if we have an array of pointers to functions of a particular type:

```
// fp is a pointer to a function, which no arguments and returns void
typedef void (*fp)();
fp fparray[5];
```

Assume a function test(), which is defined as follows:

```
int test();
```

If we perform the following operation, we get an error - type mismatch

```
fparray[0] = &test;
```

Using reinterpret_cast we can solve this problem.

```
fparray[0] = reinterpret_cast <fparray> (&test);
```

10.4 Code Listing

```
#include <iostream.h>

void init(int *x)
{
   cout<< "x value before in init method " << *x << endl;
   *x = 30;
   cout<< "x value after in init method " << *x << endl;
}


void main(int argc,char *argv[])
{
   int num1 = 10;
   int num2 = 3;

   cout<<"result1 is " << static_cast<double>(num1)/num2 <<endl;

   void init(int *pa);
   const int a1 = 10;

   // init(&a1);  //error

   init(const_cast<int*>(&a1));
   cout << "a1 value is " << a1 << endl;
}
```

10.5 Code Listing

```
#include <iostream.h>
#define interface struct

interface IFace1 {
   virtual void Fun1() = 0;
   virtual void Fun2() = 0;
   virtual void Fun3() = 0;
};

struct CImpl {
   virtual void Fun1()
   {
      cout << "Fun1() "<< endl;
   }

   virtual void Fun2()
   {
      cout << "Fun2() "<< endl;
   }

   virtual void Fun3()
   {
      cout << "Fun3() "<< endl;
   }
};

void main(){
   IFace1* pIFace = NULL;
   pIFace = reinterpret_cast<IFace1*> (new CImpl);
   pIFace->Fun1();
   pIFace->Fun2();
   pIFace->Fun3();
}
```

10.5a Code Listing

```
#include <iostream.h>
#define interface struct

interface IFace1 {
   virtual void Fun1() = 0;
   virtual void Fun2() = 0;
   virtual void Fun3() = 0;
};

class CImpl : public IFace1{
   public:
      virtual void Fun1()
      {
         cout << "Fun1() "<< endl;
      }

      virtual void Fun2()
      {
         cout << "Fun2() "<< endl;
      }

      virtual void Fun3()
      {
         cout << "Fun3() "<< endl;
      }
};

void main(){
   IFace1* pIFace = new CImpl;
```

```
   pIFace->Fun1();
   pIFace->Fun2();
   pIFace->Fun3();
}
```

10.6 Code Listing

```
#include <iostream.h>

struct base
{
   virtual void draw() {
      cout << "Base draw" << endl;
   }
};

struct der1 : base
{
   virtual void draw() {
      cout << "Derived 1 draw" << endl;
   }

   void fn1() {
      cout << "function 1" << endl;
   }
};

struct der2: base
{
   void fn2() {
      cout << "function 2" << endl;
   }
};

void main()
{
   base *p[2];
   der1 derobj1;
   der2 derobj2, *pder2;

   p[0] = &derobj1;
   p[1] = &derobj2;

   for(int i = 0; i < 2; i++) {
      p[i]->draw();

      if(dynamic_cast<der1 *>(p[i])) {
         cout << "If block " << endl;
         dynamic_cast<der1 *>(p[i])->fn1();
      }
      else if(pder2 = dynamic_cast<der2 *>(p[i])) {
         cout << "Else if block " << endl;
         pder2->fn2();
      }
   }
}
```

10.3 Exercise

Theory Questions

1. Explain the significance of Namespace?
2. What are the different types of cast operators?
3. What is namespace alias? What is the significance of using keyword?

Problems

1. Using namespace develop an application to display message text according to the language selected?