Lecture Thirteen

Namespace, Conversion function and Miscellaneous Topics

Ref: Herbert Schildt, Teach Yourself C++, Third Ed^{n} (Chapter 13)

© Dr. M. Mahfuzul Islam
Professor, Dept. of CSE, BUET

Lecture Overview

- Creating own namespace
- Creating conversion function
- Static member of a class
- const and mutable
- explicit and implicit function
- Linkage specifiers
- Array based I/O

Namespace

- **Namespaces are a relatively recent addition to localize the names of identifiers to avoid name collisions.**
- **Problems of Global namespaces:**
  - **In C++ programming environment, there has been an exploration of variables, functions and class names. All these names competed for slots in the global namespaces.**
  - **If a function, say toupper(), is defined in a program (depending upon argument list) it override the standard library functions toupper() because both are stored in global namespace.**
  - **Name collisions are compounded when two or more third-party libraries were used by the same program.**
- **Namespace localizes visibility of names declared within it, allows same name to be used in different context without giving rise to conflicts.**
- **C++ library is defined within its own namespace, std, which reduces the chance of name collisions.**
- **The general form of defining namespace is shown as:**

```
namespace name{
}
```

Namespace

- The **using** statement **alleviates the problem** of specifying the **namespace** and **scope resolution operator**. There are **two general form** of using statement:

```
using namespace name;
using name::member;
```

- There can be **more than one** namespace declaration of the **same name** in the **same file** or **different files**.
- There is a **special type of namespace**, called an **unnamed namespace** that establish **unique identifiers** that are known only within the **scope of a single file**.

```
namespace{
}
```

- You need **not** create a namespace for most **small or medium-sized programs**. However, if you create a library of **reusable code** or if you want to ensure the **widest portability**, you should **wrap your code** within a namespace.

Namespace

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

namespace firstNS {

    class demo{
          int i;
    public:
        demo(int x) { i = x; }
        void seti(int x) { i = x; }
        int geti() { return i; }
    }

    char str() = “Illustrating namespaces\n”;
    int counter;
}

namespace secondNS {
    int x, y;
}
```

```
int main(){
      firstNS::demo ob(10);

      firstNS::ob.seti(99);
      cout << firstNS::ob.geti() << endl;

      using firstNS::str;
      cout << str;

      using namespace  firstNS;
      for( counter = 10; counter; counter--)
      cout << counter << “ “;
      cout << endl;

      secondNS::x = 10;
      secondNS::y = 20;

      cout << secondNS::x << secondNS::y << endl;

      return 0;
}
```

Namespace

- To use a member of standard library, you need to either include a using namespace std or qualify each reference to a library member with std::.

```
#include <iostream>

int main(){
	double val;

	std::cin >> val;
	std::cout << val;

	return 0;
}
```

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

int main(){
	double val;

	cin >> val;
	cout << val;

	return 0;
}
```

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

int main(){
	double val;

	cin >> val;
	cout << val;

	return 0;
}
```

Namespace

➢**There can be more than one namespace declaration of the same name in the same file or different files.**

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

namespace Demo {
       int a;
}

int x;

namespace Demo {
       int b;
}
```

```
int main(){
       using namespace Demo;

       a = b = x = 100;
       cout << a <<" " << b << " " << x<< endl;

        return 0;
}
```

Creating a Conversion Function

- A **conversion function automatically converts an object into a value that is compatible with the type of the expression.**
- **The general form of a conversion function is shown as**

operator type() { return value;}

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

class coord {
      int x, y;
public:
      coord( int i, int j) { x = i; y = j;}
      operator int() { return x*y; }
}
```

```
int main(){
      coord o1(2,3), o2(4, 3);
      int i;

      i = o1;            // type conversion
      cout << i << endl;

      i = 100 + o2;  // type conversion
      cout << i << endl;

      return 0;
}
```

Creating a Conversion Function

- **Another Example.**

  strcpy() has the prototype

  char *strcpy(char *s1, const char *s2);

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

class strtype {
      char str[80];
       int len;
public:
      strtype(char *s) {
            strcpy(str, s);
            len = strlen(s);
      }
      operator char *() { return str; }
}
```

```
int main(){
      strtype s("This is a test\n");
      char *p, s2[80];

       p = s;           // type conversion
       cout << p << endl;

       strcpy(s2, s);   // type conversion
       cout << s2 << endl;

       return 0;
}
```

Static class members

- For a normal member variable, each time an object is created, a new copy of that variable is created.
- Only one copy of static variable and all the objects of its class share it.
- A static member variable exists before any object of its class is created.
- A static class member is a global variable-
  - Declaring a static member is not defining it, it must be defined elsewhere using the scope resolution operator.
  - It is possible to access a static member variable independent of any object.
- The principal reason static member variables supported by C++ is to avoid the need for global variables.
- A static member function does not have a this pointer.
- Static member functions cannot be declared as const or volatile.

Static class members

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

class myclass {
      static int i;
public:
      void seti( int n) { i = n;}
      int geti() { return i; }
};

int myclass::i;

int main(){
      myclass o1, o2;

      o1.seti(10);
      cout << o1.geti() <<" " << o2.geti() << endl;

      return 0;
}
```

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

class myclass {
public:
      static int i;
      void seti( int n) { i = n;}
      int geti() { return i; }
};

int myclass::i;

int main(){
      myclass o1, o2;

      myclass::i = 100;
      cout << o1.geti() <<" " << o2.geti() << endl;

      return 0;
}
```

const member function and mutable

- **When a member function is declared as const, it cannot modify the object that invokes it.**
- **A const member function can be called by either const or non-const objects.**
- **mutable overrides const-ness. A mutable member can be modified by a const member function.**

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

```
class Demo {
        mutable int i;
        int j;
public:
        int geti() const {return i;}  // ok
        int getj() const {return j;}  // ok
        void seti(int x) const {
                i = x;                  //ok
        }
        void setj(int x) const {
                j = x;                  //wrong
        }
};
```

```
int main(){
        Demo ob;

        ob.seti(100);
        ob.setj(200);
        cout << ob.geti() <<" " << ob.getj() << endl;

        return 0;
}
```

A Final look at Constructors

- **Type conversion from the type of argument to the type of class is of two types: implicit and explicit.**
- **Implicit Type Conversion:**

The declaration, **myclass ob(4);**
Can be written as, **myclass ob = 4;** // not allowed in explicit conversion

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

```
class myclass {
      int a;
public:
      myclass( int x) {a = x;}

      myclass( char *str) { a =atoi(str);}
      int geta() { return a; }
};
```

```
int main(){
      myclass ob1 = 4;
      myclass ob2 = "123";

      cout << ob1.geta() <<" " ;
      cout<< ob2.geta() << endl;

      ob1 ="1776";   //ob1 = myclass("1776");
      ob2 = 2001; //ob2 = myclass(2001);

      cout << ob1.geta() <<" " ;
      cout<< ob2.geta() << endl;

      return 0;
}
```

A Final look at Constructors

➢**To prevent Conversion, program with Explicit Conversion:**

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

class myclass {
	int a;
public:
	explicit myclass( int x) {a = x;}
	explicit myclass( char *str) { a =atoi(str);}
	int geta() { return a; }
};
```

Using linkage specifiers and The asm keyword

- **Linkage specifier** tells the compiler that **one or more functions** in your C++ program will be **linked with another language** that might have a **different approach** to **naming**, **parameter passing**, **stack restoration** and the like.

```
extern "C" int func(int x);                    // link as C functions
extern "C" {
        void f1();
        int f2(int x);
        double t3(double x, int *p);
}
```

- **asm** keyword **embeds assembly language** instructions in C++ source code.

```
void func() {
     asm("mov bp, sp");
     asm("push ax");
     asm("mov cl, 4");
     ----------------
}
```

- **Different languages** accept three **slightly different** forms of asm statement:

```
asm op-code;
asm op-code newline
asm {
        instruction sequence
}
```

Array-based I/O

- **istrstream, ostrstream and strstream should be included which also includes istream, ostream and iostream.**

- **General form of the ostrstream constructor:**

  ostrstream ostr(char *buf, streamsize size, openmode mode=ios::out);

  Example:

  ostrstream ostr(buf, sizeof buf);

- **General form of the istrstream constructor:**

  istrstream istr(const char *buf);

- **General form of the strstream constructor:**

  strstream ostr(char *buf, streamsize size, openmode mode=ios::in |ios::out);