Introduction to C++

- Namespace
- Preprocessor

Namespace

Namespace:

Namespace is a technique of expressing logical grouping of all elements into a single translation unit in order to avoid the name collision or conflict.

In other words,Namespace is a container for identifiers.It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace.

A namespace is a scope.In general, local scopes,global scope & classes are namespaces.

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

```
namespace identifier
{
 // code declarations
}
```

where identifier is any valid identifier

Namespace

Example:

```
namespace X
{
int a;
}
namespace Y
{
float a;
}
```

In order to access these variables from outside X & Y namespace, we use scope resolution operator. We can access 'a' variable in the following manner:

```
X::a;
Y::a;
```

Namespace

Simple Example using Namespace:

```
#include<iostream>
using namespace std;
namespace X
{
  int a=10;
}
namespace Y
{
  float a=20.5;
}
int main()
{
  cout<<X::a<<endl;
  cout<<Y::a<<endl;
}
```

Namespace

Example:

```
#include <iostream>
using namespace std;
namespace first_space         // first namespace
{   void func()
  {
    cout << "Inside first_space" << endl;
  }
}
namespace second_space        // second namespace
{   void func()
  {
    cout << "Inside second_space" << endl;
  }
}
int main()
{
first_space::func();          // Calls function from first name space.
second_space::func();         // Calls function from second name space.
return 0;
}
```

Namespace

Using a namespace:

There are three ways to use a namespace in program:

- Scope Resolution
- The Using Directive
- The Using Declaration

Namespace

1. With Scope Resolution:

Any name (identifier) declared in a namespace can be explicitly specified using the namespace's name and the scope resolution :: operator with the identifier.

Example:

```
#include<iostream>
using namespace std;
namespace X
{
  int a=10;
}
namespace Y
{
  float a=20.5;
}
int main()
{
  cout<<X::a<<endl;
  cout<<Y::a<<endl;
}
```

Namespace

The Using Directive:

using keyword allows us to import an entire namespace into our program with a global scope. It can be used to import a namespace into another namespace.

Example:

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

Namespace

```
using namespace first_space;
int main ()
{
   // This calls function from first name space.
   func();
   return 0;
}
```

Namespace

The Using Declaration:

When we use using directive, we import all the names in the namespace and they are available throughout the program, that is they have global scope.
But with using declaration, we import one specific name at a time which is available only inside the current scope.

In using declaration, we never mention the argument list of a function while importing it.

Example:

```
#include <iostream>
using namespace std;
namespace first_space        // first name space
{
  void func1()
  {
    cout << "Inside first_space func1" << endl;
  }
  void func2()
  {
          cout<<"Inside first_sapce func2"<<endl;
  }
}
```

Namespace

```
// second name space
namespace second_space
{
   void func1()
   {
      cout << "Inside second_space func1" << endl;
   }
   void func2()
   {
            cout<<"Inside second_space func2"<<endl;
   }
}
using namespace first_space;        //using Directive
int main ()
{
   using second_space::func1;        //using Declaration
   func1();
   func2();
   return 0;
}
```

Preprocessor

Preprocessor:

The preprocessors are the directives, which give instruction to the compiler to preprocess the information before actual compilation starts.
All preprocessor directives begin with #. Preprocessor directives are not C++ statements so they do not end in a semicolon (;).
We already have seen a #include directive in all the examples.
This macro is used to include a header file into the source file.

There are number of preprocessor directives supported by C++ like #include, #define, #if, #else, #line, etc.

Preprocessor

The #define directive:

The #define preprocessor directive creates symbolic constants. The symbolic constant is called a macro and the general form of the directive is:

#define macro-name replacement-text

When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled.

Preprocessor

Example:

```
#include <iostream>
using namespace std;
#define PI 3.14159
int main ()
{
cout << "Value of PI :" << PI << endl;
return 0;
}
```

In this case before the program is compiled, all occurrences of PI will be replaced by 3.14159

Preprocessor

Functions like Macros:

We can use #define to define a macro which will take argument as follows:

Example:

```
#include <iostream>
using namespace std;
#define MIN(a,b) (((a)<(b)) ? a : b)
int main ()
{
int i, j;
i = 100;
j = 30;
cout <<"The minimum is " << MIN(i, j) << endl;
return 0;
}
```

Preprocessor

Conditional Compilation:

There are several directives, which can use to compile selectively portions of our program's source code. This process is called conditional compilation.
The conditional preprocessor construct is much like the if selection structure.

General syntax is as follows:

```
#ifndef NULL
#define NULL 0
#endif
```

Preprocessor

Example:

```
#include <iostream>
using namespace std;
#define DEBUG
#define MIN(a,b) (((a)<(b)) ? a : b)
int main ()
{
  int i, j;
  i = 100;
  j = 30;
#ifdef DEBUG
  cout <<"Trace: Inside main function" << endl;
#endif
cout <<"The minimum is " << MIN(i, j) << endl;
#ifdef DEBUG
  cout <<"Trace: Coming out of main function" << endl;
#endif
   return 0;
}
```

Preprocessor

Output:

Trace: Inside main function
The minimum is 30
Trace: Coming out of main function

Preprocessor

The # and ## operators:

The # operator causes a replacement-text token to be converted to a string surrounded by quotes.

The ## operator is used to concatenate two tokens.

Example:

```
#include <iostream>
using namespace std;
#define MKSTR( x ) #x
#define concat(a, b) a ## b
int main ()
{
   int xy = 100;
   cout<< MKSTR(HELLO C++) << endl;
   cout << concat(x, y);
   return 0;
}
```

Output:

```
Hello
100
```

Discussions