5. Namespaces

In large projects the correct handling of the names represents an important problem of the programming activity. The variables and objects declared inside the blocks does not present any danger, because they are local elements and for them memory is allocated on the stack part of the program. The global variables declared outside functions have the advantage to be visible in the body of all functions defined in the same file as the variables, and for them memory is allocated in the static data part of the program.

In the C language there is a problem with global variables because the namespace of a program is global to the whole program; in other words, two variables with the same name defined in distinct files of a program exists in the same namespace and this fact can generate conflicts for the compiler. For example, for a program developed by two programming teams, if a variable is declared with the same name in two different files at file level by the two teams, the compiler will generate an error (multiple defined variables). Because every programming team manages its own names, a coordination action at the whole project level concerning global variables must exist.

A useful way for resolving this type of conflicts is represented by static variables. A static variable defined at file level becomes visible only in the respective file and it is hidden for the others program modules. The C++ language, in addition to this technique, allows the utilization of some distinct **namespaces** in the same program. So the global space of the program name can be divided in many namespaces, using the ***namespace*** facility.

5.1 Defining namespaces

The definition of a namespace can be realized by using the following syntax:

```
namespace [<identifier>]
{
  <declarations>
}
```

where <*identifier*> represents the name associated to the respective namespace.

Such a definition produces a new namespace, all the declarations between braces being local to the respective space.

Example:

```
namespace sp1 {
  int a, b;
  struct point { double x, y; }
}

namespace sp2 {
  double a, b;
  int point, point1, point2;
```

```
}

int main {
  // ...
}
```

From above example one can observe that a `namespace` definition does not end with ";" as for `struct`, `class` and `union`.

Remark: A `namespace` definition can appear only at global level, outside any function definition. In addition it is allowed the inclusion of a namespace into another space. Usually `namespace` definitions are placed in the header files and not in the implementation files.

It is possible for a certain `namespace` definition to contain a lot of declarations, case when it can be extended in more header files. In this case, one of these files represents a `namespace` definition and the others represent only *completions* of the definition (they do not redefine the same namespace). A usually technique used in this case is the definition in cascade of these header files. If the header files f_1, f_2, … , f_k are used for defining the same namespace, then in each file f_i, the file f_{i-1} must be included.

Example 5.1:

```
// header1.h file
#ifndef HEAD1
#define HEAD1
namespace spa   // 'spa' namespace definition
{
  int m, n;
  int f(int, int);
  // ...
}
#endif

// header2.h file
#ifndef HEAD2
#define HEAD2
#include "head1.h"
namespace spa  // 'spa' namespace completion; is not an error
{
  double x, y, z;
  double g(double);
  // ...
}
#endif

// header3.h file
#ifndef HEAD3
#define HEAD3
#include "head2.h"
namespace spa  // 'spa' namespace completion
{
  char s1, s2;
  char h(const char *);
  // ...
```

```
}
#endif

// pr.cpp file
#include "head3.h"
main()
{
  // ...
}
```

In conclusion it is allowed the using of more namespace constructions which refers the same namespace in the same file: the first represents the definition of a namespace, while the others are completions of the first namespace.

Remark: Outside the explicit defined space by the programmers, every compiling unit has associated by default an *anonymous namespace* (without any name). An anonymous namespace is unique for a compiling unit and the variables declared in this space do not have to be qualified when they are used. For adding variables at the default namespace of the compiling unit, in the respective module must be added a namespace constructions without a name.

Example 5.2. In the next file:

```
//mod1.cpp file
namespace
{
  class A {
    // ...
  };
  class B {
    // ...
  };
  double p,q;
}

void processing()
{
  // ...
}
```

the declaration of classes *A* and *B* and also of the variables *p* and *q* belong to the default namespace of the file *mod1.cpp*.

An anonymous namespace is unique for every compiling unit, so all names from this space are *local* to the respective module, without being necessary to be declared `static`. The C++ language encourages the anonymous namespace utilization, which replace the method with static allocation for names and hiding them within files.

The only operation which can be used with a namespace is defining *aliases* for a namespace. The syntax for associating an alias to a namespace is the following:

```
namespace <namespace 1> = <namespace 2>;
```

Example:

```
namespace spa1
{
  int a, b, c;
  // ...
}
namespace spa2 = spa1;
```

In the previous example, *spa1* and *sp2* represents the same namespace (initial defined by *spa1*).

A relevant example of using an alias for a namespace represents the situation when it is used a namespace already defined but whose name is too long and difficult to be used for prefixing the elements defined in it.

5.2 Using namespaces

In order to be able to use the names defined in a namespace they names must be attached at their defining namespace. There exist three methods of names utilization: to use the ***resolution operator***, the ***`using` directive***, and the ***`using` declaration.***

Because a namespace represents the scope for its all intern declarations, the classic method of specifying names from these namespaces is represented by prefixing them with the name of the namespace using the resolution operator, as in the case of members of a class.

Example 5.3.

```
// spa.h file
namespace spa {
  class A {
    int n;
    int f();
    // ...
  }
  double x, y;
  class B;
  // ...
}

class spa::B {
  char c1, c2;
  B(char);
  // ...
};

// pr.cpp file
#include "spa.h"
int spa::A::f() { return n; }
spa::B::B(char c) { c1 = c2 = c; }
void processing () {
  spa::x = 7.5;
```

```
  // ...
}
```

The disadvantage of this method is that of all the used names must be prefixed by the namespace from which they belong, so it becomes hard the write programs. A more easy method is that of importing all the names from a namespace with the help of a `using` *directive*. In this way, the name from the respective namespace can be direct used, without prefixing.

The syntax of the `using` directive is:

```
using namespace <name>;
```

The effect of the `using` directive consists in importing of all names from the respective namespace in the place where the directive `using` appears. For example, if the directive is used at file level, all the names from the specified space become global names in the file.

Example:

```
// pr0.cpp file
#include "spa.h"
using namespace spa;

int A::f() {return n;}
B::B(char c) {c1 = c2 = c;}

void processing () {
  x = 7.5;
  // ...
}
```

The scope of the names imported with the `using` directive is given by the place where this directive is put: at file level, or inside on a block. When the directive `using` is used inside on a block, the names imported from the respective space become locales in the block containing the `using` directive.

Example:

```
// pr1.cpp file
#include "spa.h"

void h() {
  using namespace spa;
  x = y = h;
  A a;
  int k = a.f();
  // ...
}
```

The `using` directive can be used in another namespace, respecting the same visibility rules.

Because the `using` directive imports all the names defined in a namespace, it is possible as a certain name to be redefined in the respective place. There are two distinct situations for redefining names: by explicit definition of another object with the same name as the existent one, or by using two `using` directives referring two namespace which contains certain common names.

In the first case the object explicit defined covers the one defined in namespace.

Example:

```
// pr2.cpp file
#include "spa.h"
using namespace spa;

float x = 7; // spa::x is covered by x
spa::X = 8.5;
// ...
```

In the second case can exist the possibility of a name collision. But the ambiguity appears effectively when the name is referred, not in the place of the `using` directive.

Example:

```
// spa1.h file
#ifndef SPA1
#define SPA1
namespace spa1 {
  int x;
  // ...
}

// pr3.cpp file
#include "spa.h"
#include "spa1.h"
using namespace spa;
// Is not an ambiguity when defining a namespace
using namespace spa1;
x = 3; // Error! Ambiguity
spa1::x = 3.5; // Correct!
```

The third method of using the names from namespaces is constituted by the `using` *declaration*. A `using` declaration does not import all the names from a space (as the `using` directive does), but only individual names.

The syntax is:

```
using <space name >::<name>;
```

One can observe that the `using` declaration does not specify the data type associated to the respective name, but only the namespace to which it belongs.

The names which appear in a `using` declaration do not have to be qualified in the scope containing the declaration, as for the case of the directive `using`.

A `using` declaration can appear in a program in the same places as a common declaration. Because it is a declaration, it can overload an object with the same name imported from another namespace with a `using` directive.

Example:

```
// pr4.cpp file
#include "spa.h"
#include "spa1.h"

void g() {
  using namespace spa;
  using spa1::X;
  x = 4;              // spa1::X
  spa::x = 4.3;       // 'spa' must be explicitly specified
  // ...
}
```

Because in a `using` declaration it is specified only the name of an identifier, and not its type, in the case of overloading functions a single declaration relatively to the name of a function allows the loading of all the functions with the same name.

Example:

```
namespace spa3
{
  void f(int,int);
  double f(double);
  int f(int);
  // ...
}

void spa3::f(int a, int b)
  { cout<<a<<b; }

double spa3::f(double x)
  { return x * x; }

int spa3::f(int n)
  { return 2 * n; }

void pr4() {
  using spa3::f;
  f(3, 4);
  double y = f(7.5);
  int k = f(2);
  // ...
}
```