Namespaces

- The correct management of names from programs is an important problem of programming activity, especially in the case of large projects.

- There are two methods for hiding names inside the source files:
    - using *static variabes*
    - using ***namespaces***.

- The C++ language 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.

A. Defining namespaces

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

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

- All declarations between braces are local to the respective namespace.

Example:

```
namespace sp1 {
  int a, b;
  struct point { double x, y; }
}
namespace sp2 {
  double a, b;
  int point, point1, point2;
}
```

- 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. A usually technique used in this case is the definition in *cascade* of these header files.

Example:

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

// header2.h file
#ifndef HEAD2
#define HEAD2
#include “header1.h”
// ‘spa’ namespace completion
namespace spa
{
  double x, y, z;
  double g(double);
  // ...
}
#endif

// header3.h file
#ifndef HEAD3
#define HEAD3
#include “header2.h”
// ‘spa’ namespace completion
namespace spa
{
  char s1, s2;
  char h(const char *);
  // ...
}
#endif
```

- Outside the explicit defined space by the programmers, every compiling unit has associated by default an ***anonymous namespace***. 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.

Example.

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

- An *anonymous* `namespace` is unique for every compiling unit, so all names from this space are local to the respective module (is not necessary to be declared `static`). The C++ language encourages the *anonymous* `namespace` utilization, which replace the method with static allocation for names.

- ❑ 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 <namespace1> = <namespace2>;
  ```

Example. In the following example, *spa1* and *sp2* represents the same namespace.

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

B. Using namespaces

- ❑ There are three methods for referring in programs the names defined inside namespaces:
  - ***the resolution operator*** (**::**)
  - ***the directive `using`***
  - ***the declaration `using`***

- ❑ When using the ***resolution operator***, the names must be prefixed with the name of the namespace by using the resolution operator, as in the case of members of a class.

- ❑ 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.

Example:

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

- When using the ***`using` directive***, the names from the respective namespace can be used directly, without prefixing, as in the following syntax:

```
using namespace <nume>;
```

- ❑ The effect of the **`using`** directive consists in importing of all names from the respective namespace in the place where the directive **`using`** appears.

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();
  // ...
}
```

- ❑ 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,
  - 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;
// It is not an ambiguity
using namespace spa1;
x = 3; // Error! Ambiguity
spa1::x = 3.5; // Correct!
```

- *The third method* of using the names from namespaces is the ***`using` declaration***. A `using` declaration imports only individual names from a namespace. The syntax is:

```
   using <nume spatiu>::<nume>;
```

- 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’ must be explicitly specified
  spa::x = 4.3;
  // ...
}
```

- 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);
  // ...
}
```