

## Section 5.2

# Namespaces

---

Namespaces allow us to group a set of global classes, objects and/or functions under a name. To say it somehow, they serve to split the global scope in sub-scopes known as *namespaces*.

The form to use *namespaces* is:

```
namespace identifier
{
    namespace-body
}
```

Where *identifier* is any valid identifier and *namespace-body* is the set of classes, objects and functions that are included within the *namespace*. For example:

```
namespace general
{
    int a, b;
}
```

In this case, **a** and **b** are normal variables integrated within the **general** *namespace*. In order to access to these variables from outside the namespace we have to use the scope operator `::`. For example, to access the previous variables we would have to put:

```
general::a
general::b
```

The functionality of *namespaces* is specially useful in case that there is a possibility that a global object or function can have the same name than another one, causing a redefinition error. For example:

```
// namespaces
#include <iostream.h>

namespace first
{
    int var = 5;
}

namespace second
{
    double var = 3.1416;
```

```
5
3.1416
```

```

}

int main () {
    cout << first::var << endl;
    cout << second::var << endl;
    return 0;
}

```

In this case two global variables with the **var** name exist, one defined within *namespace first* and another one in *second*. No redefinition errors thanks to *namespaces*.

## using namespace

The **using** directive followed by **namespace** serves to associate the present nesting level with a certain *namespace* so that the objects and functions of that *namespace* can be accesible directly as if they were defined in the global scope. Its utilization follows this prototype:

```
using namespace identifier;
```

Thus, for example:

```

// using namespace example
#include <iostream.h>

namespace first
{
    int var = 5;
}

namespace second
{
    double var = 3.1416;
}

int main () {
    using namespace second;
    cout << var << endl;
    cout << (var*2) << endl;
    return 0;
}

```

```

3.1416
6.2832

```

In this case we have been able to use **var** without having to precede it with any scope operator.

You have to consider that the sentence **using namespace** has validity only in the block in which it is declared (understanding as a block the group of instructions

within key brackets `{}`) or in all the code if it is used in the global scope. Thus, for example, if we had intention to first use the objects of a *namespace* and then those of another one we could do something similar to:

```
// using namespace example
#include <iostream.h>

namespace first
{
    int var = 5;
}

namespace second
{
    double var = 3.1416;
}

int main () {
    {
        using namespace first;
        cout << var << endl;
    }
    {
        using namespace second;
        cout << var << endl;
    }
    return 0;
}
```

```
5
3.1416
```

## alias definition

We have the possibility to define alternative names for *namespaces* that already exist. The form to do it is:

```
namespace new_name = current_name ;
```

## *Namespace* std

One of the best examples that we can find about *namespaces* is the standard C++ library itself. According to ANSI C++ standard, the definition all the classes, objects and functions of the standard C++ library are defined within *namespace std*.

You may have noticed that we have ignored this rule all along this tutorial. I've decided to do so since this rule is almost as recent as the ANSI standard itself

(1997) and many older compilers do not comply with this rule.

Almost all compilers, even those complying with ANSI standard, allow the use of the traditional header files (like `iostream.h`, `stdlib.h`, etc), the ones we have used throughout this tutorial. Nevertheless, the ANSI standard has completely redesigned this libraries taking advantage of the templates feature and following the rule to declare all the functions and variables under the namespace `std`.

The standard has specified new names for these "header" files, basically using the same name for C++ specific files, but without the ending `.h`. For example, `iostream.h` becomes `iostream`.

If we use the ANSI-C++ compliant include files we have to bear in mind that all the functions, classes and objects will be declared under the `std` namespace. For example:

```
// ANSI-C++ compliant hello world
#include <iostream>

int main () {
    std::cout << "Hello world in ANSI-C++\n";
    return 0;
}
```

Hello world in ANSI-C++

Although it is more usual to use `using namespace` and save us to have to use the scope operator `::` before all the references to standard objects:

```
// ANSI-C++ compliant hello world (II)
#include <iostream>
using namespace std;



int main () {
    cout << "Hello world in ANSI-C++\n";
    return 0;
}
```

Hello world in ANSI-C++

The name for the C files has also suffered some changes. You can find more information on the new names for the standard header files in the document [Standard header files](#).

The use of the ANSI-compliant way to include the standard libraries, apart for the ANSI-compliance itself, is highly recommended for STL users.

---

[Previous:](#)   [Next:](#)  
[5-1. Templates.](#) [index](#) [5-3. Exception handling.](#)