2 Namespaces¶

- **You can download the template for the programming tasks of the module as a `zip` file from `this link`.**

Contents

C++ Primer: Chapter 18.2 (Namespaces)

Namespaces are an essential concept in C++, and important in structuring large software in more manageable units. Namespace groups together different identifiers, for example variable names, types, class names, functions, and so on. that are part of same software module, or for example, library. Unlike in C, which only has a single global namespace (i.e. every function and variable is contained within it), in C++ we can define multiple namespaces that contain different symbols. In particular, every class and struct creates its own namespace that contains the symbols of that class.

Namespaces can also be created explicitly with the **namespace** keyword. This is helpful in large programs, to avoid name collisions (e.g. two functions with the same name) between two parts of code that often may be developed by different independent parties.

The following creates a namespace *mymath* which contains the function *Add*:

```
1// Define a namespace called mymath
2namespace mymath {
3    // Define a function called Add within mymath
4    int Add(int first, int second) {
5        return first + second;
6    }
7} // namespace mymath
```

Note that there is no semicolon at the end of the namespace block.

Namespaces are referred to using a double colon notation; `std::string` refers to some symbol called `string` within the namespace `std`. In our above example, to call the function *Add* from outside the namespace *mymath* one would write

```
mymath::Add(some_number, some_other_number)
```

Namespaces can also be nested by defining a namespace within a namespace:

```
namespace mymath {
namespace constants {
const double kPi = 3.14159265359;
const double kE = 2.71828182845;
} // namespace constants

int Add(int first, int second) {
    return first + second;
}
} // namespace mymath
```

To access the *kPi* from outside the *mymath* namespace, one would write

```
mymath::constants::kPi
```

You may have noticed that the above specified accessing the members from *outside* the mymath namespace. This is because the `namespace_name::` prefix is not necessary from within the namespace:

```
1namespace mymath {
2    namespace constants {
3        const double kPi = 3.14159265359;
4        const double kE = 2.71828182845;
5    } // namespace constants
6
7    // Function is defined inside the mymath namespace
8    double CalculateCircumference(double diameter) {
9        return diameter * constants::kPi;
10    }
11} // namespace mymath
12
13// Function is defined outside the mymath namespace
14double CalculateDiameter(double circumference) {
15    return circumference / mymath::constants::kPi;
16}
```

Here you can see that the *CalculateDiameter* function uses `mymath::constants::kPi` while *CalculateCircumference* uses `constants::kPi` to access the same thing.

Namespace should be used, for example, for a library that can be used by multiple different programs. This prevents name collisions from occurring with the code using the library and other libraries. For example, everything in the gcheck libary, which used on this course to test the exercises, is defined within the namespace *gcheck*.

Because any program or software library typically consists of multiple files, it is possible to specify the same namespace block for the same name in multiple discontinuous locations and multiple files.

The global namespace is implicitly declared for names defined in global scope. If one wants to explicitly refer to such global names, the notation `::one_name` refers to *one_name* under global scope. Note that the *main* function has to be within the global namespace.

The *std* namespace used before as an example is a special one used by all parts of the C++ standard library and, while you technically can, you should not define your own symbols inside it.

using decleration
previous | next

> A lazy writer can use the *using* declaration to bring a name from other namespace to the current namespace. `using std::string;` would allow to use just `string` without the *std* prefix later in the code where the string type from C++ standard library is used.
>
> It is also possible to import entire namespace, for example by `using namespace std;` for the standard library. Although this may save in typing effort, it is rarely a good idea, because it opens the possibility of name collisions between the standard namespace (in this case) and the current name space.

2.1 Namespace aliasing¶

Another thing one can do to ease their typing burden is to alias a namespace. For example, you could do

```
namespace myconstants = mymath::constants;
```

after which *kPi* can be accessed with `myconstants::kPi` instead of `mymath::constants::kPi`.

Hint

More information at [cppreference].

2.2 Using namespaces¶

The following code uses namespaces to show that namespace concept enable efficient name reuse. You can copy/paste this snippet to test its functionality.

```
1#include <iostream>
2
3namespace mymath {
4namespace constants {
5    const double kPi = 3.14159265359;
6    const double kE = 2.71828182845;
7} // namespace constants
8
9// Function is defined inside the mymath namespace
10double CalculateCircumference(double diameter) {
11    return diameter * constants::kPi;
12}
13} // namespace mymath
14
15// Function is defined outside the mymath namespace
16double CalculateDiameter(double circumference) {
17    return circumference / mymath::constants::kPi;
18}
19
20// --> we can reuse the name because we are in a different name space
21const double kE = 2.71;
22
23int main() {
24    double diameter = 100.0;
25    double circumference = mymath::CalculateCircumference(diameter);
26    double calculated_diameter = CalculateDiameter(circumference);
27
28    // The following prints out the information.
29    // How this works is explained in the next section
30    std::cout << "Diameter is " << diameter << std::endl
31        << "Circumference is " << circumference << std::endl
32        << "Diameter calculated from circumference is "
33        << calculated_diameter << std::endl;
34
35    return 0;
36}
```