- Example

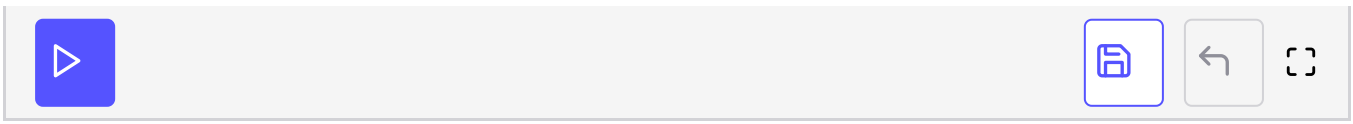
In this lesson, we will look at an example of user-defined literals.

WE'LL COVER THE FOLLOWING

Example

```
//userDefinedLiteral.cpp
#include <iostream>
#include <ostream>

namespace Distance{
  class MyDistance{
    public:
      MyDistance(double i):m(i){}

      friend MyDistance operator +(const MyDistance& a, const MyDistance& b){
        return MyDistance(a.m + b.m);
      }
      friend MyDistance operator -(const MyDistance& a, const MyDistance& b){
        return MyDistance(a.m - b.m);
      }

      friend std::ostream& operator<< (std::ostream &out, const MyDistance& myDist){
        out << myDist.m << " m";
         return out;
      }
    private:
      double m;

  };

  namespace Unit{
    MyDistance operator "" _km(long double d){
      return MyDistance(1000*d);
    }
    MyDistance operator "" _m(long double m){
      return MyDistance(m);
    }
    MyDistance operator "" _dm(long double d){
      return MyDistance(d/10);
    }
```

```
    MyDistance operator "" _cm(long double c){
      return MyDistance(c/100);
    }
  }
}

using namespace Distance::Unit;

int main(){

  std:: cout << std::endl;

  std::cout << "1.0_km: " << 1.0_km << std::endl;
  std::cout << "1.0_m: " << 1.0_m << std::endl;
  std::cout << "1.0_dm: " << 1.0_dm << std::endl;
  std::cout << "1.0_cm: " << 1.0_cm << std::endl;


  std::cout << std::endl;
  std::cout << "1.0_km + 2.0_dm +  3.0_dm - 4.0_cm: " << 1.0_km + 2.0_dm +  3.0_dm - 4.0_cm <
  std::cout << std::endl;

}
```



Explanation

- The literal operators are implemented in the namespace `Distance::unit`. We should use namespaces for user-defined literals due to two reasons. First, the suffixes are typically quite short. Second, the suffixes typically stand for units that are already established abbreviations. In the program, we used the suffixes `km`, `m`, `dm`, and `cm`.

- For the class `MyDistance`, we overloaded the basic arithmetic operators (lines 10 - 15) and the output operator (lines 17 - 19). The operators are global functions, and can use - thanks to their friendship - the internals of the class. We store the output distance in the `private` variable `m`.

- We display, in lines 48 - 51, the various distances. In lines 27 to 37, we calculate the meter in various resolutions. The literal operators take as argument a long double and return a `MyDistance` object. `MyDistance` is automatically normalized to meters.

- The last test looks quite promising: `1.0_km + 2.0_dm + 3.0_dm + 4.0_cm` is `1000.54` m (line 55). The compiler takes care of the calculations with all units.

Get a better understanding of this topic with a few exercises in the next lesson.