- Examples

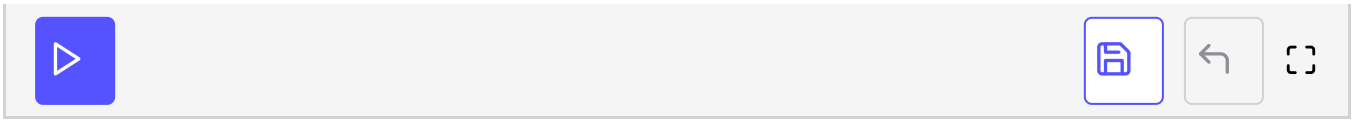
In this lesson, we'll look at examples of what we've learned about literals so far.

WE'LL COVER THE FOLLOWING

User-Defined

```
#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 because name collisions are very likely, for two reasons. First, the suffixes are usually very short; second, the suffixes usually stand for units which 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 9 - 14) and the output operator (lines 16 - 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`.

- In lines 47 - 50, we display the various distances, and in lines 26 - 37, we convert distances in various units to meters. The literal operators take a long double as an argument 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 54). The compiler takes care of the calculations for all units.

Built-In

In the following program, we use some of the built-in suffixes:

```
#include <iostream>
#include <chrono>

using namespace std::literals::chrono_literals;

int main(){

  std::cout << std::endl;

  typedef std::chrono::duration<long long, std::ratio<2700>> hour;
  auto schoolHour = hour(1);
  // auto schoolHour= 45min;

  auto shortBreak = 300s;
  auto longBreak = 0.25h;

  auto schoolWay = 15min;
  auto homework = 2h;

  auto schoolDayInSeconds = 2 * schoolWay + 6 * schoolHour + 4 * shortBreak + longBreak + hom

  std::cout << "School day in seconds: " << schoolDayInSeconds.count() << std::endl;

  std::chrono::duration<double, std::ratio<3600>> schoolDayInHours = schoolDayInSeconds;
  std::chrono::duration<double, std::ratio<60>> schoolDayInMinutes = schoolDayInSeconds;
  std::chrono::duration<double, std::ratio<1, 1000>> schoolDayInMilliseconds= schoolDayInSeco

  std::cout << "School day in hours: " << schoolDayInHours.count() << std::endl;
  std::cout << "School day in minutes: " << schoolDayInMinutes.count() << std::endl;
  std::cout << "School day in milliseconds: " << schoolDayInMilliseconds.count() << std::endl

  std::cout << std::endl;

}
```

Explanation

- The program is totally self-explanatory. The suffixes are expressive enough. Making the correct additions is the job of the compiler. The time literals support basic addition, subtraction, multiplication, division, and modulo operations.

In the next lesson, there are a couple of coding challenges related to literals.