```cpp
//////// Lecture 89: User Defined Literals
/* NOTES
A literal is a fixed value that appears directly in the code.
C++ supports different types of literals - integer, floating point, boolean, string.

Some of these literals can be modified through prefixes and suffixes.
-> 14u (unsigned int)
In C++11, we can define OUR OWN SUFFIX
-> Can be applied to integer, floating point, character and string literals.

-> ADVANTAGE: Represent values more clearly by Creating custom literals: syntactic shortcuts and increase type safety.
*/
//For example, if we had a class called Temperature and we want to create an instance of that class by specifying a value of 82.5
Temperature temp {82.5}; // Fahrenheit or celsius?
// Now internally, the temperature class may store the value as a celsius, but what if we want to specify the temperature as Fahrenheit?

// in that case, we can create a user-defined literal which will help us represent the temperature in a different unit.

// SYNTAX:
// To create a user-defined literal, you have to define a function using the operator  keyword.
return_type operator ""_literal (arguments){};
/*
- The operator"" defines a literal operator function. The keyword operator"" is followed by a pair of empty double quotes.
- The return type can be any type, INCLUDING VOID, and the literal is a name that always starts with UNDERSCORE. (underscore is part of the name - literals without the underscore are reserved for the
standard library.
- The arguments of the literal operator function can be of the following types only:
=> to use the integer argument, the argument type should be unsigned long long
=> to use the floating point argument, the argument type should be long double.
=> to use the character argument, the argument type should be char, wchar_t, char16_t, char32_t
=> to use the string argument, the argument type should be a const char*.

// these types are used because they can hold the LARGEST VALUE of that category type.
*/

#include <iostream>
class Distance{
    long double m_Kilometres;
public:
    Distance (long double km) : m_Kilometres{km} {};
    long double GetKm() const {
        return m_Kilometres;
    }
    void setKm(long double km) {
        m_Kilometres = km;
    }
};
// What if we specified miles instead of km? we would have to convert the miles into km.
// Instead of doing that manually, we can add a User Defined Literal Function.
Distance operator"" _mi (long double val){
    return Distance {val * 1.6};
}
// This is just a syntactic sugar. The literal operator function will create an object of distance which has been initialised with the value that you have specified here, but that has been suitably converted into km. Using custom literals, we can make our code more expressive and reduce the chance of errors.
```

```cpp
// now we may also add one more operator - for meters - and we can create the instance like this.
Distance operator"" _m (long double val){
    return Distance {val / 1000};
}
int main(){
    Distance dist {32.0_mi}; // 32 miles
    Distance dist {32.0_m}; // 32m
    std::cout << d2.setKm() << std::endl;
}
/*
IMPORTANT POINTS:
- Custom literals should always begin with an underscore
=> Literals without underscore are reserved for the standard library.
=> if you do try to create a UDF Literal without an underscore, the compiler may flag it as a warning or an error.

- it is not possible to redefine the meaning of built in literal suffixes.

- Only following types can be suffixed to make user defined literals:
=> unsigned long long, long double, const char*, char

- Literal operator functions CANNOT BE MEMBER FUNCTIONS.
=> They will always be global functions.
*/
```