```
1  //////// Lecture 82: Enums (Part I) - Basics
2  // Let's write a function to fill the background with
   •   some colour
3  // We'll use an integer as an argument and we want to
   •   represent the different colours. Instead of using
   •   simple numbers to represent colours, we can use
   •   symbolic constants, so we can either create macros
   •   or we can create constants.
4  // Using macros
5  #define RED 0
6  // Using constants
7  const int RED = 0;
8  void FillColor(int color){
9      // Fill background with some color
10 }
11 // When you want to invoke the FillColor() and want to
   •   specify a colourm, you don't have to remember the
   •   number, instead you can use the name of the colour,
   •   so I can invoke FillColor() with RED as an argument,
   •   by passing the macro or you can use the symbolic
   •   constant.
12 int main()
13 {
14     FillColor(RED);
15 }
16 // but there is a problem with this function. The user
   •   can invoke FilLColor with any number as an argument.
   •   And the function would not know what to do with this
   •   value, as a result, it will have undefined behaviour.
17 // ideally, we should use a restricted range of values
   •   that can be accepted by FilLcolor.
18 // This is where enumerated types will help us.
19
20
21
22 //////// Lecture 83 - Enums (PART II) - Scoped Enums
   •   (C++11)
23 enum Color{
24     RED,
25     GREEN,
26     BLUE
27 };
```

```
// Can also be written as
enum Color{RED, GREEN, BLUE};
void FillColor(Color color)
{
    // Fill background with some colour
    if (color == RED) // do something
    if (color == GREEN) // do something
    if (color == BLUE ) // do something
}

int main(){
    Color c = RED; // you can assign an enum like this!
    FillColor(c);
    // OR
    FillColor(BLUE);
    //because an integer doesn't implicitly convert
      into an enum type, only a specific range of
      values are accepted.
}
// it is still possible to use integers with enums,
  but you will have to apply static_cast, but whatever
  value you cast should fall within the range of
  enumerator values, otherwise the result is undefined.
int main(){
    FillColor(static_cast<Color> (2)); // BLUE!
}

// Let's say we want to create an enum for traffic
  lights
enum TrafficLight{
    RED,
    YELLOW,
    GREEN
};
// and compile it, you will get an error, because
  'RED' and 'GREEN' are being redefined, and that's
  not allowed.
// Enumerators and Enumerated types have the scope in
  accordance to where they are defined, and in this
  code they have a global scope, so the redefinition
  of the same colours 'RED' and 'GREEN' ARE NOT
  ALLOWED. You cannot use these enumerators as
```

```
    variable names or enumerators.
// This may cause compilation errors or it may lead to undefined results at runtime.


///// SCOPED_ENUMS
// C++11 solves this problem by SCOPED ENUMS. A scoped enum is one in which the enumerator has a scope within its enumerated type, so the enumerator is not visible outside the scope of its type.
// To create a scoped enum, you have to use the class keyword with the enum like this:
enum class Color{RED, GREEN, BLUE};
// now color is a scoped enum. Now the variables RED, GREEN, BLUE are not visible in the global scope, so when we want to reference the enumerator, we have to write this instead:
void FillColor(Color color)
{
    if (color == Color::RED ) // do smth
}
// Same thing in main() as well
int main()
{
    FillColor(Color::RED);
}
// now, we can create this traffic light enum as a scoped enum.
enum class TraffficLight{
    RED, YELLOW, GREEN
};
// this way, the names of numerators do not clash. One advantage is that the enumerator is not implicitly convertible to int, so if we try to assign 'RED' to an integer like
int x = Color::RED;
// this code will not compile.
// To resolve this error, you have to EXPLICITLY apply a static_cast like this:
int x = static_cast<int>(Color::RED);
// the underlying type in scoped enums is always int, but you can specify any other integral type as the
```

```
    underlying type, and that can be done by doing this:
enum class TrafficLight : /*underlying type*/  {RED,
    GREEN, YELLOW};
// example:
enum class TrafficLight : char {RED = 'c', GREEN,
    YELLOW};

// Enum types can be initialised with values.
enum class TrafficLight:{RED = 6, GREEN, YELLOW};
// now green will have a value 6, yellow will have a
    value 7.
// In the same way for enumerated type TrafficLight,we
    can assign a different value to RED:
// then  GREEN will have the value 'd' and YELLOW will
    have the value 'e'. note that the ASCII values of
    the characters are stored, not the characters
    themselves. SO RED = 99, GREEN = 100, YELLOW = 101

```