```
////// Lecture 57: Copy Elision
// Copy Elision is a technique used by the compiler to eliminate temporary objects.
// To explain the concept of copy elision, consider the following code:
//////// main.cpp
#include "Integer.h"
#include <iostream>
Integer Add(int a, int b) // modified Add function
{
    Integer temp(a+b);
    return temp; // returns sum via temp object.
}
int main()
{
    Integer a = 3; // initialise the integer with a literal
    // This can work - because we've already added the copy assignment and move assignment functions to the integer class. Even though we have not learnt how to implement these, the copy assignment and move assignment is important in understanding how copy elision works. When those functions are invoked, we will see the output on the console.

    // You may guess that in this case the copy constructor would be invoked, but in fact it is not invoked, but only the parameterised constructor is invoked. This is because of copy elision.

    // ELIDE: Elide is a verb and simply means to omit or leave out, and this is used to represent omission of vowel consonant in pronunication.
    // ELISION: The other word is elision, this is a noun, and it simply means the act or an instance of omission.
    // We are using the assignment operator to initialise a user defined object with a primitive type. These ARE NOT COMPATIBLE.
    // so what the compiler expanded the line above to
```

```
    be was:
    Integer a = Integer(3);
    // And so a temporary object will be constructed
    out of this literal, and THAT will be copied
    into a.
    // But because integer class has a parameterised
    constructor that accepts an integer, the
    compiler ELIDES this object so the copy is not
    created. Instead, the parameterised constructor
    of the integer class is DIRECTLY INVOKED and
    three is passed as an argument.
    // So because of the copy elision, this is what
    the compiler actually did:
    Integer a(3);
    // This is how a compiler has performed copy
    elision, this means it has omitted or left out
    the copy that would have been other wise created
    here.

    // We are able to turn copy elision off in GCC,
    using the flag
    //                      -fno-elide-constructors
    //          g++ -fno-elide-constructors -std=c++17
    ... <filenames>
    // And when you do this, then you will see the
    call to the MOVE constructor.
    // This is because Integer(3) is a temporary
    object (R-value), and instead of being copied
    into a , it gets moved into a.
    return 0;
}

/////// COPY ELISION VS MOVE ELISION
// Depending on whether the expression requires copy
  or move, the compiler may use either use copy
  elision or move elision.
// In copy elision, the copy constructor is omitted,
  and in move elision the move constructor is omitted.
  By default, all modern compilers use copy or move
  elision, this is why we never see the temporary
  getting created either through copy or move.
// So whenever an expression requires a temporary, the
```

```
    compiler MAY implement copy elision. In this Add()
    function, since we are returning an object by value,
    the compiler may implement copy elision here. Let's
    understand this better with an example.
Integer Add(int a, int b) // modified Add function
{
    Integer temp(a+b);
    return temp; // returns sum via temp object.
}
int main()
{
    // Invoke Add() here
    Integer a = Add(3,5);
    // An object temp is created, and because the
      function is return by value, another object is
      created and then that object is copied or moved
      into a.
    // So in all we should see 3 constructor calls in
      this 1 line of code
    // The first will be for the temp object,
    // the second will be a move constructor during
      the return by value
    // the third will be a move constructor moving the
      contents of the temporary into the object a.

    // If we turn on copy elision, we will see that
      the compiler has turned on copy elision. The
      parameterised constructor call is for the temp
      obeject, now we don't see the call to the move
      constructor when the temp is returned by value
      (this has been elided), and even the move from
      the temp object into a has been elided. Because
      of elision, unnecesary objects are not created.
}

////////// NAMED RETURN VALUE OPTIMISATION

Integer Add(int a, int b){
    // 1 version:
    Integer temp(a+b);
    // The copy of the return value is elided by the
      compiler, this is called NAMED return value
```

```
    optimisation.
    return temp;
    // The copy is also elided in the below
    alternative:
    return Integer (a+b); // return value optimisation
    (not named)
}
/* NOTES:
- for copy elision to work, the class must have the
  copy and move constructors.
- In C++17 there is a change, but we will discuss this
  change later.
*/
```