**Scope, Visibility and Lifetime– Part 1**

**Slide 1**

This is the first part of a three part presentation on scope, visibility and lifetime.

In this part, we will discuss the relationship between scope and visibility in programming languages.

**Slide 2**

We begin by defining scope and two related terms, which are visibility and lifetime.

Scope is a compile time concept, which means that at compile time, the compiler can determine the scope of a variable, or more generally any identifier. A variable’s scope is the portion of the program in which it is potentially accessible.

Visibility is the inverse of scope. A variable is visible at a particular point in the program if it can be accessed. Normally the two are complete inverses.

There is one situation in which that is not true. That situation is when a variable is redeclared. Unless it is redeclared, it is visible at any point in the program that is within its scope. But when a variable is redeclared in an inner scope, the variable of the same name declared in the outer scope is no longer visible, at least not without some qualification.

The last of these three terms is *lifetime*. Lifetime and scope are often confused. Lifetime refers to a time period while the program is running. Specifically, it is the time period during the program execution that the variable exists. Although scope and lifetime are related, they are not exactly the same.

**Slide 3**

To illustrate these terms, let’s examine some examples from C++. We have chosen C++ because it makes a clear distinction between local and global variables. We begin with an example to illustrate the scope of global variables. The variable that is now highlighted is a global variable.

It is global because it is declared outside of any function. Its scope is from its declaration to the end of the file.

**Slide 4**

Next we will examine the local variables in the same example.

The first local variable that we are going to consider is another declaration of i. But this declaration of i is a local because it is declared inside the function f1.

The scope of local variables is from their declaration to the end of the block in which they are declared. This block is the complete function so its scope is from its declaration to the end of the function.

Next is a third declaration of i in an inner block. This concept of nested blocks in block structured languages originated with Algol. The scope rules that were devised then still apply to languages that allow the redeclaration of local variables with overlapping scopes. Java is one exception because it prohibits the declaration of variables of the same name in overlapping scopes.

Its scope is from its declaration to the end of the block in which it is declared.

**Slide 5**

Next we will look at the difference between scope and visibility.

The assignment to i shown in red modifies the innermost i, the one that is declared immediately before it.

This second assignment to i changes the global i Referencing the global is achieved by qualifying the name with ::, which is called the scope resolution operator. The distinction is often made between direct visibility and visibility by qualification. In this case we would say that although the global i is not directly visible here, it is visible when qualified by the scope resolution operator.

Referencing i here modifies the second declaration of i, the one that is inside f1 but outside the inner block. Notice that this variable could not have been accessed from the inner block. Although the inner block is within the scope of that variable, it is neither directly visible there, nor is it visible by qualification.

Finally because f2 does not contain a redeclaration of the global variable, this assignment is changing the global i. It is directly visible here, so it can be referenced without qualification by the scope resolution operator.