

1 Scope of a Variable

- As you've seen first-hand or read in the book, you cannot define two variables with the same name in C++ (at least not in the same *scope*)
- *examples...*
- In any large program, one typically has many, many variables defined throughout the program
- Additionally, large programs are broken into many *modules* (we'll get to those later) that may span many files
- If we had to make sure that *all* variables *throughout* a program were uniquely named, variable names would become quite cumbersome
- Additionally, one could not develop a single module without knowing about *all the other modules*

- Thankfully, C++ provides a way around this
- In C++, we can limit *where a variable exists* in the file...
- This is called the variable's *scope*
- Outside the variable's scope, it's as if the variable does *not* exist
- C++ provides a few ways of limiting the scope of a variable, but the only one we need to know about for now is called *block scope*

2 Block Scope: “{” and “}”

- In C++, a *block* is defined by using a set of curly braces, “{” and “}”
- The stuff inside the braces is called the *block*
- As you’ve noticed already, we use curly braces quite often
- They appear “around” the `main` function, after control structures (`if`, `else`, and `switch`), and after looping structures (`for`, `while`, and `do-while`).
- *But they can appear on their own... more on that in a second*
- Each block has its own *scope*, which is simply the chunk of the file between the braces
- Any variables defined in that scope *exists only in that scope, and...*

2 Block Scope: “{” and “}”

- In C++, a *block* is defined by using a set of curly braces, “{” and “}”
- The stuff inside the braces is called the *block*
- As you’ve noticed already, we use curly braces quite often
- They appear “around” the `main` function, after control structures (`if`, `else`, and `switch`), and after looping structures (`for`, `while`, and `do-while`).
- *But they can appear on their own... more on that in a second*
- Each block has its own *scope*, which is simply the chunk of the file between the braces
- Any variables defined in that scope *exists only in that scope, and...*
- *...any inherited scopes.*
- *examples...*

2.1 Scopes and “Headless” Blocks

- Although blocks are usually created for functions, control structures, or looping structures, they don't need such additional structure
- That is, we can create “headless” blocks in C++
- *examples...*
- Such blocks are practically useless, but they can be used to illustrate points about scopes

- For example, within an “inner” scope, we can define new variables (**Just as we do in main’s inner scope!**)...
- ... and they don’t exist in *any* outer scope
- *examples...*

- For example, within an “inner” scope, we can define new variables (**Just as we do in main’s inner scope!**)...
- ... and they don’t exist in *any* outer scope
- *examples...*
- That is, *inner blocks inherit their scope from outer blocks*
- but *outer blocks do not inherit their scope from inner blocks*

- Additionally, we can define new *versions* of variables in sub-scopes
- That is, if an outer scope has a variable named “**x**”, an inner scope can define a *new* variable named “**x**”...
- ... *which is a different variable!*
- *examples...*
- When the inner-scope ends, the new version of **x** will cease to exist, and the old version will “take over”

2.2 Using Blocks with various Constructs

- As you've seen by now, blocks are almost always used in conjunction with a function, a control structure, or a looping structure
- For example, with a simple `if` statement,

```
if (<condition>
{
    BLOCK
}
```

- The block that immediately follows the `if` is *bound to* that `if`-statement...

2.2 Using Blocks with various Constructs

- As you've seen by now, blocks are almost always used in conjunction with a function, a control structure, or a looping structure
- For example, with a simple `if` statement,

```
if (<condition>
{
    BLOCK
}
```

- The block that immediately follows the `if` is *bound to* that `if`-statement...
- **Provided there is not semi-colon after the `if`-statement!**
- We call it the *body* of the `if` statement
- If the condition evaluates to `true`, then the body is executed
- The same is true for *all* control structures and looping structures

2.3 Common Error with Blocks and Constructs

- The most common error when using various constructs with blocks is *including a semi-colon* after the construct
- Take the previous example with an `if`-statement followed by a block...
- Only let's add a semi-colon after the `if`-statement

```
if (<condition>) ;  
{  
    BLOCK  
}
```

- What happens?

- *That if-statement ends at the semi-colon!*
- *The following block is **not** bound to that if-statement!*
- As a result, that block is a *headless block*, and always executes!
- This is a common error, and applies to all conditional & looping constructs