Scope of Identifiers

- Identifiers can be declared anywhere in a program
- **Local variables**
  - declared inside a block - such as the body of a function
  - cannot be accessed outside of that block
- **Scope** (scope of visibility or scope of access) - region of program code where it is legal to reference(use) an identifier.
  - **Class Scope** - described in Chapter 12
  - **Local Scope** - The scope of an identifier declared inside a block extends from the point of declaration to **the end of that block**.
  - **Namespace scope** - Scope of an identifier declared in a namespace definition extending from point of declaration to the end of the namespace body
  - **Global Scope** - The scope of an identifier declared outside all functions, namespaces and classes extends from the point of declaration to the end of the entire file containing the program code
More on Scope of Identifiers

• C++ function names have global scope - Cannot nest a function definition inside another function definition.
• Global variables and constants are declared outside of all functions. *Global Variables are rarely used.*

• Name precedence or name hiding - declaration of a local identifier (with the same name as a global identifier) will hide the global identifier for the scope of the local declaration

• Scope Rules - Rules that determine where in a program an identifier may be accessed
  - Nonlocal identifier - With respect to a given block, any identifier declared outside that block
  - Read and understand scope rules: pages 404-405

Scope Rules

1) *Function names have global scope*
2) *Function parameters* have local scope associated with the outermost block of the function body
3) *Scope of global variables or constants* extend from declaration to the end of the file (see rule 5)
4) *Local variable scope* extends from declaration to end of the block in which it is declared. This scope includes any nested blocks except as noted in rule 5
5) *Local identifiers have name precedence* - the scope of an identifier does not include any nested block that contains a locally declared identifier with the same name

Note: Functions cannot access identifiers in other functions - even functions called by a function cannot access the identifiers of the calling function

See Ch_09_01.cpp and Ch_09_02.cpp for examples of variable scope
Variable Declarations and Definitions

- Recall - function prototypes are a declaration only and do not cause memory to be reserved, and function definitions include the body of the function and cause memory to be reserved

- A variable declaration becomes a variable definition if memory is reserved - what we have been doing up to this point is variable definition

- The standard `int num;` is a declaration and definition

Namespaces

- In the line: `using namespace std;` `using` is a directive, and namespace is another way of saying scope. In this line, `std` is the identifier that indicates the scope to use

- Namespace definition:
  ```
  namespace name_of_scope
  {
    statements
  }
  ```
  - `name_of_scope` is an identifier for the namespace being created
  - Look at pages 407-409
Namespaces (continued)

- Identifiers declared in a namespace can be accessed outside the body (scope) of the namespace by one of three methods only
  a) Using a qualified name
  b) Using a declaration
  c) Using a directive - all identifiers are available, but only within the scope in which it appears. We have been placing “using namespace std;” at the top of the program for global coverage

---

Namespace Example

```cpp
#include <iostream>

int main ()
{
    std::string line = "Hello\n";  // Data types are no longer defined, so must use the
    // qualified name method
    std::cout << "a qualified name: " << line;
    if (true)
    {
        using std::cout;  // allows cout to be used, but only within the if block
        cout << "a using declaration: " << line;
    }
    // cout << "test"; this line will not work
    // since cout is not defined for main, but in the if statement only
    using namespace std;
    cout << "a using directive: " << line;
    return 0;
}
```

See Ch_09_03.cpp for C++ examples of this program
Lifetime of a Variable

- **Lifetime** - The period of time during program execution when an identifier has memory allocated to it

- **Automatic variable** - storage is allocated at block entry and deallocated at block exit. By default, variables declared within a block are automatic variables

Lifetime of a Variable - Continued

- **Lifetime** - The period of time during program execution when an identifier has memory allocated to it
  
  - **Static variable** - storage remains allocated for the duration of the entire program
    
    - All global variables are static variables
    
    - Use the keyword `static` to make local variables static variables
    
    - Lifetime persists from function call to function call
    
    - Usually better to declare a local variable as `static` than to use a global variable. Why? Keeps other functions in the program from changing the value unexpectedly
    
    - If the initialization of a static variable occurs in the definition, the initialization only occurs once – the first time the definition is encountered
Static Variable Example

```cpp
#include <iostream>
using namespace std;

void function1();

int main ()
{
    for (int i = 10; i <= 15; i++)
    {
        function1();
    }
    return 0;
    // Note that the i defined in the for loop has scope only in that loop.
    // The i defined in the function has scope only in the function block,
    // and this i takes name precedence over the i in the for loop.
}

void function1()
{
    static int i = 0;  // i retains its value, initialization only occurs once
    static int j;    // j is not initialized here
    j = 0;           // initialization of j occurs every function call.
    cout << "i is: " << i << " -- j is: " << j << endl;
    i++; j++;        // i and j are modified. i keeps it value from call to call.
}

See Ch_09_04.cpp for a C++ example of static variables
```

Side Effects and Global Constants

- A **side effect** is any effect one function has on another that is not part of the explicitly defined interface between the functions - *i.e., unexpectedly changing the value of a parameter*. Global variables can result in such a situation - read pages 417 - 419

- **Global Constants**
  - are acceptable and work best when several functions need access to the same constant.
  - Provide for ease of change and consistency
  - Should not be used if the constant is required in one function only. In this case, define the constant in the function that requires it
Value-Returning Functions

- Recall that a call to a void function is a complete statement

- A call to a value returning function is part of an expression

- A value returning function returns one function value only by means of a return statement
  - DataType at the beginning of the function heading declares the type of function value the function returns
    - This is called the function value type (or function type)
    - If the DataType is omitted, int is assumed
    - If the DataType of the return expression is different from the declared function value type, the value of the expression is coerced to the correct type

Value Returning Functions (continued)

- Value returning functions
  - must have return Expression;
  - Should be used if one value is to be returned only

- Rule of thumb: never use reference parameters with a value returning function.
  - Use value parameters only
  - A possible exception is when the function returns a Boolean value
Value Returning Functions (continued)

- **Boolean Functions** - Boolean results can be used as a function type for value returning functions

- **Boolean functions are useful for complicated conditional tests** - logical expressions that need to be evaluated in if statements, while loops, etc

Naming of Functions

- Best names for value returning functions are nouns or adjectives so that the name suggests a value and not a command – i.e. Average()

- Best names for void functions are verbs which make the statement sound like a command - i.e. PrintMenu()

- Best names for Boolean functions are adjectives or phrases beginning with “is”. Though the “is” is optional in some cases – i.e. IsDataValid()
When to Use Value-Returning Functions - pg 432

- **Do not use a value returning function if:**
  - the module(function) must return more than one value or modify any of the caller’s arguments
  - If the module must perform I/O (a Boolean value-returning function indicating success or failure is an exception)

- **Use a value returning function if there is one value to be returned only:**
  - and the value is a Boolean value, or
  - the value is to be used immediately in an expression

- When in doubt, use a void function
- If either a void function or value returning function are acceptable, use the one you feel most comfortable implementing

Ignoring the Return Value

- C++ lets you ignore the value returned by a value returning function
- \texttt{sqrt(x)} is a value returning function, both statements are valid:
  - \texttt{sqrt(x);} // returned value is lost/ignored
  - \texttt{answer = sqrt(x);} // return value is used

- Remember - void functions cannot be used in expressions and must be called as a statement:
  - \texttt{void Print\_Data(ofstream&);} // function prototype
  - \texttt{Print\_Data(dataOut);} // valid void function call
  - \texttt{answer = Print\_Data(dataOut);} // Invalid void function call

- Read the Problem-Solving Case Studies: Pages 436-444
Testing and Debugging - Stubs

- A stub is a dummy function that assists in testing part of a program. A stub has the same name and interface as the actual function, but it is usually much simpler.

Example

```cpp
void GetInput (ifstream& DataIn, int number)
{
    // stub for the function
    cout << "Entering GetInput function.\n";
    cout << "No action performed\n";
    return;
}
```

See Example Ch_09_06.cpp for an example of this program

Testing and Debugging -

- A driver is a simple `main` function that is used to call a function being tested and permits direct control of the testing process.

Example

```cpp
float calc_pay(int, float, float); // function to test
int main()
{
    cout << "testing function calc_pay\n";
    cout << "for id 1, wage = 10.5 and \";
    cout << "hours = 45\n";
    cout << "TotalPay = "
    cout << calc_pay(1,10.5,45) << endl;
    // other statements as needed
    return 0;
}
```

See Example Ch_09_06.cpp for an example of this program
Reading

- Read through the Case study programs on pages 436-444. Look at the functional decomposition and how the programs are arranged.

- Read the summary
- Go over the quick check and exam preparation exercises at the end of the chapter.

Use of `exit` in Functions

```cpp
void get_data(ifstream &InFile)
{
    string file_name;
    cout << "enter the name of the file to open: ";
    cin >> file_name;
    InFile.open(file_name.c_str());
    if (!InFile)
    {
        cout << "Error: file: " << file_name << " was not found\n";
        exit (1); // exit the program at this point
        // a return; here would exit the function
        // but not the program
    }
    // other statements
}
```

See Example Ch_09_07.cpp for an example of this program
Boolean Function Example

```cpp
bool OpenInput (ifstream&, string&);
int main ()
{
    ifstream InFile;
    string file_name;
    if (!OpenInput(InFile, file_name))
    {
        cout << "Error in opening input file: " << file_name << endl << endl;
        return 1; // since in main will terminate the program
    }
    // other statements
    return 0;
}
bool OpenInput(ifstream& Input, string& name)
{
    cout << "Enter the name of the file to open for input: ";
    cin >> name;
    Input.open(name.c_str());
    if (!Input)
        return false; // other statements
    return true;
}
See Example Ch_09_07.cpp for an example of this program
```

This if statement can be replaced with the statement: return bool(Input);