**Rules of Scope**

| scope  | (of an identifier) the range of program statements within which the identifier is recognized as a valid name |

**C++ Scope Rules**

1. Every identifier must be declared and given a type before it is referenced (used).
2. The scope of an identifier begins at its declaration.
3. If the declaration is within a compound statement, the scope of the identifier ends at the end of that compound statement. We say the identifier is **local** to that compound statement (or block).
4. If the declaration is not within a compound statement, the scope of the identifier ends at the end of the file. We say the identifier has **global** scope.

---

**Function Declaration vs Definition**

A function name is an identifier, so it must be formally declared before it is used. Unlike a simple variable or constant, a function has a return type and (possibly) a formal parameter list. The function declaration must also specify those.

The function declaration is essentially just a copy of the function header from the function definition. A function declaration must specify the types of the formal parameters, but not the formal parameter names. However, it is good practice to include the formal parameter names in the function declaration.

Function declarations are typically declared at file scope or within a header file, although they may be placed anywhere. The placement determines the scope of the function name, and hence where it may be called.

```cpp
double circleArea(double Radius) {
    const double PI = 3.141592653;
    return (PI * Radius * Radius);
}
```

Many authors refer to a function declaration as a **prototype**.
Pass-by-Value Parameters

Pass-by-Value
- default passing mechanism except for one special case discussed later
- allocate a temporary memory location for each formal parameter (when function is called)
- copy the value of the corresponding actual parameter into that location
- called function has no access to the actual parameter, just to a copy of its value

```cpp
int FindMinimum(int A, int B) {
    if (A <= B)
        return A;
    else
        return B;
}
```

```cpp
int First = 15,
    Second = 42;
int Least = FindMinimum(First, Second);
```

### Variable | Value
--- | ---
First | 
Second | 
Least | 

### Variable | Value
--- | ---
A | 
B | 

Created when call occurs and destroyed on return.

Pass-by-Reference Parameters

Pass-by-Reference
- put ampersand (\&) after formal parameter type in prototype and definition
- forces the corresponding actual and formal parameters to refer to the same memory location;
  that is, the formal parameter is then a synonym or alias for the actual parameter
- called function may modify the value of the actual parameter

```cpp
void SwapEm(int& A, int& B) {
    int TempInt;
    TempInt = A;
    A = B;
    B = TempInt;
}
```

```cpp
int First = 15,
    Second = 42;
SwapEm(First, Second);
```

### Variable | Value
--- | ---
First | 
Second | 

### Variable | Value
--- | ---
A | 
B | 
TempInt | 

©William D McQuain, January 2005
Pass-by-Constant-Reference Parameters

- Precede formal parameter type with keyword `const`, follow it with an ampersand (`&`)
- Forces the corresponding actual and formal parameters to refer to the same primary memory location; just as in pass-by-reference
- But, the called function is not allowed to modify the value of the parameter; the compiler flags such a statement as an error

```
int AddEm(const int& A, const int& B) {
    int Sum;
    Sum = A + B;
    return Sum;
}
```

```
int First = 15,
    Second = 42,
    Third;
Third = AddEm(First, Second);
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td></td>
</tr>
<tr>
<td>Second</td>
<td></td>
</tr>
<tr>
<td>Third</td>
<td></td>
</tr>
</tbody>
</table>

Choosing a Parameter Passing Mechanism

Pass-by-Reference
- Use **only if** the design of the called function requires that it be able to modify the value of the parameter

Pass-by-Constant-Reference
- Use if the called function has no need to modify the value of the parameter, but the parameter is very large (e.g., a string or a structure or an array, as discussed later)
- Use as a safety net to guarantee that the called function cannot be written in a way that would modify the value passed in†

Pass-by-Value
- Use in all cases where none of the reasons given above apply
- Pass-by-value is safer than pass-by-reference

† Note that if a parameter is passed by value, the called function may make changes to that value as the formal parameter is used within the function body. Passing by constant reference guarantees that even that sort of internal modification cannot occur.
See the CS 1044 notes on C++ Functions for more details and examples.