

Chapter 8

Scope, Lifetime and More on Functions

Definitions

- Scope
 - The region of program code where it is legal to reference (use) an identifier
- Three types of Scope
 - Class Scope
 - Local Scope
 - Global Scope

Class Scope

- This term refers to the data type called a class.
- We will not talk about classes until 1704.

Local Scope

- The scope of an identifier declared inside a block extends from the point of declaration to the end of that block.
- Also, function parameters (formal parameter) extends from the point of declaration to the end of the block that is the body of the function.

Global Scope

- An identifier declared outside of all functions and classes extends from the point of declaration to the end of the entire file containing the program code.
- Global scope should be avoided at all cost at this point in your programming career.

Example

```
int gamma;        //global scope
int main()
{
    gamma = 3;
    ...
}
void someFunc()
{
    gamma = 5;
    ...
}
```

Name Precedence

- Identifiers, inside of a nested block with the same name as an identifier in an enclosing block, hide the identifier in the enclosing block.
- Also called *name hiding*.

```
#include <iostream>
using namespace std;
void someFunc( float );
const int a = 17; //global const
int b;           //global variable
int c;           //global variable
int main()
{
    b =4;
    c = 6;
    someFunc ( 42.8 );
    return 0;
}
```

```
void someFunc( float c )
{
    float b; //prevents access to
            //global b
    b = 2.3;
    cout << "a = " << a; // 17
    cout << " b = " << b; // 2.3
    cout << " c = " << c; // 42.8
}
```

Scope Rules!!

- Non-local identifier: With respect to a given block, any identifier declared outside that block
 1. A function name has global scope. Function definitions cannot be nested within function definitions.
 2. The scope of a function parameter is identical to the scope of a local variable declared in the outermost block of the function body.

More Scope Rules!!

3. The scope of a global variable or constant extends from its declaration to the end of the file.
 4. The scope of a local variable or constant extends from its declaration to the end of the block in which it is declared. This scope includes any nested blocks
 5. The scope of an identifier does not include any nested blocks that contain a locally declared identifier with the same name (local identifiers have name precedence).
- Rules taken from page 375 of your book.

```
void block1 ( int, char& );
void block2 ();
int a1; // global
int a2; // global
int main()
{
...
}
```

```
void block1 ( int a1, //prevents
              //access to global a1
              char& b2 )
{
  int c1;    //has same scope as
  int d2;    // b2,c1,d2;
  ...
}
```

```
void block2()
{
  int a1; //blocks global a1
  int b2; //local; no conflict
  while (...)
  {      //block3
    int c1; //local to block3
    int b2; //blocks non-local b2
    ...
  }
}
```

Namespaces

- What is a namespace?
 - Basically a named scope.
 - This allows you to hide identifiers inside of a scope to avoid name clashes.
 - You must use one of the three namespace resolution methods to access the identifiers inside of a namespace

Example

- In `cstdlib`
`namespace std`
`{`
`...`
`int abs(int);`
`...`
`}`

Fourth Scope

- Namespace Scope
 - The scope of an identifier declared in a namespace definition extends from the point of declaration to the end of the namespace body *and* its scope includes the scope of a `using` directive specifying that namespace

Lifetime of a Variable

- Lifetime
 - The period of time during program execution when an identifier has memory allocated to it
- Automatic Variable
 - A variable for which memory is allocated and de-allocated when control enters and exits the block in which it is declared
- Static Variable
 - A variable for which memory remains allocated throughout the execution of the entire program

Example

```
void someFunc( int someParam )
{
    int i = 0;    //initialized each time
    int n = 2;    //initialized each time
    static char ch = 'A';    //Initialized once
    ...
}
```

Interface Design

- Side Effect
 - Any effect of one function on another that is not part of the explicitly defined interface between them.

```

void CountInts();
int count;
int intVal;

int main() {
    count = 0;
    cin >> intVal;
    while ( cin ){
        count++;
        CountInts();
        cin >> intVal;
    }
    cout << count << " lines of input
processed.\n";
    return 0;
}

```

```

void CountInts()
{
    count = 0;
    while ( intVal != 99999 )
    {
        count++;
        cin >> intVal;
    }
    cout << count << " integers on
this line.\n";
}

```

Value-Returning Functions

- Function Value type
 - the data type of the result value returned by a function
- Example

```
int Day ( /* in */ int month,  
         /* in */ int dayOfMonth,  
         /* in */ int year )
```
- Not limited to only numeric returns, can return any data type

Rules of Thumb

1. If the module must return more than one value or modify any of the caller's arguments, do not use a value-returning function
2. If the module must perform I/O, do not use a value return function.
3. If there is only one value returned and it is a Boolean value, use a value returning function.

More Rules of Thumb

4. If there is only one value returned and that value is to be used immediately in an expression, use a value returning function.
5. When in doubt, use a void function.
6. If both a void function and a value-returning function are acceptable, use the one you feel more comfortable with.
 - From page 399 in your book.