Chapter 4

Functions

Chapter 4 Topics(part 2)

- Scope of an Identifier
  - Local Scope vs. Global Scope
  - Scope Rules
  - Name Precedence
- Lifetime of a Variable
  - Global Variable
  - Automatic Variable
  - Static Variable
- More on Functions

Scope of Identifier

- the scope of an identifier (or named constant) means the region of program code where it is legal to use that identifier for any purpose

Local Scope vs. Global Scope

- the scope of an identifier that is declared inside a block (this includes function parameters) extends from the point of declaration to the end of the block

- the scope of an identifier that is declared outside of all namespaces, functions and classes extends from the point of declaration to the end of the entire file containing program code

```
const   float   TAX_RATE = 0.05 ;        // global constant
float    tipRate ;                       // global variable
void   handle ( int,  float ) ;          // function prototype

using  namespace  std ;

int  main (  )
{
    int      age ;                       // age and bill local to this block
    float   bill ;
    .                                    // a, b, and tax cannot be used here
    .                                    // TAX_RATE and tipRate can be used
    handle (age, bill) ;

    return 0 ;
}

void  handle (int a,  float b)
{
    float  tax ;                         // a, b, and tax local to this block
    .                                    // age and bill cannot be used here
    .                                    // TAX_RATE and tipRate can be used
}
```

Detailed Scope Rules

1. Function name has global scope.
2. Function parameter scope is identical to scope of a local variable declared in the outermost block of the function body.
3. Global variable (or constant) scope extends from declaration to the end of the file, except as noted in rule 5.
4. Local variable (or constant) scope extends from declaration to the end of the block where declared. This scope includes any nested blocks, except as noted in rule 5.
5. An identifier's scope does not include any nested block that contains a locally declared identifier with the same name (local identifiers have name precedence).

Name Precedence (or Name Hiding)

- when a function declares a local identifier with the same name as a global identifier, the local identifier takes precedence within that function

Name Precedence Implemented by Compiler Determines Scope

- When an expression refers to an identifier, the compiler first checks the local declarations.
- If the identifier isn't local, compiler works outward through each level of nesting until it finds an identifier with same name. There it stops.
- Any identifier with the same name declared at a level further out is never reached.
- If compiler reaches global declarations and still can't find the identifier, an error message results.

Chapter 4 Topics(part 2)

- Scope of an Identifier
  - Local Scope vs. Global Scope
  - Scope Rules
  - Name Precedence
- Lifetime of a Variable
  - Global Variable
  - Automatic Variable
  - Static Variable
- More on Functions

Lifetime of a Variable

- the lifetime of a variable is the time during program execution when an identifier actually has memory allocated to it

These allocate memory

```
int  someInt ;                  // for the global variable

int  Square (int n)             // for instructions in body
{
    int  result ;               // for the local variable
    result  =  n * n ;
    return  result ;
}
```

These do NOT allocate memory

```
int  Square (int n) ;          // function prototype

extern  int  someInt ;         // someInt is a global
                               // variable defined in
                               // another file
```

Lifetime of Local Automatic Variables

- local variables are “alive” while function is executing
- their storage is created (allocated) when control enters the function
- their storage is destroyed when function exits

Lifetime of Global Variables

- their lifetime is the lifetime of the entire program
- their memory is allocated when program begins execution
- their memory is destroyed when the entire program terminates

By default

- local variables are automatic
- to obtain a static local variable, you must use the reserved word static in its declaration.

Automatic vs. Static Variable

- storage for automatic variable is allocated at block entry and destroyed at block exit
- storage for static variable remains allocated throughout execution of the entire program

Static and Automatic Local Variables

```
int  popularSquare( int  n)
{
    static  int   timesCalled = 0 ;  // initialized only once
    int       result  =  n * n ;          // initialized each time

    timesCalled  =  timesCalled + 1 ;
    cout  << "Call # " << timesCalled  << endl ;
    return  result ;
}
```

Chapter 4 Topics(part 2)

- Scope of an Identifier
  - Local Scope vs. Global Scope
  - Scope Rules
  - Name Precedence
- Lifetime of a Variable
  - Global Variable
  - Automatic Variable
  - Static Variable
- More on Functions

Inline Function

```
#include <iostream>
using namespace std;
inline int max(int, int, int);            //   function prototype

int main( )
{
    int i=10, j=20, k=30, m;
    m=max(i,j,k);   //   replaced by the body block of max when compiling
    cout<<"max="<<m<<endl;
    return 0;
}

inline int max(int a, int b, int c)       //  define the inline function
{
     if(b>a) a=b;
     if(c>a) a=c;
     return a;                            //  return the largest number
}
```

Function Overloading

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

int sum (int a, int b) { return a + b; }                    // two int parameters
int sum (int a, int b, int c) { return a + b + c; }         // three int parameters
double sum (double a, double b) { return a+b; }             // two double parameters

int main( )
{
    int i=10, j=20, k=30;
    double m=2.1, n=3.5;
    cout<<i<< ' + ' <<j<< ' = ' <<sum(i, j)<<endl;                  //two int arguments
    cout<<i<< ' + ' <<j << ' + ' <<k<< ' = ' <<sum(i, j, k)<<endl; // three int arguments
    cout<<m<< ' + ' <<n<< ' = ' <<sum(m, n)<<endl;       // two double arguments
    return 0;
}
```

Function Template

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

template <class T>              // declare a template with type parameter T
T sum (T a, T b) { return a + b; }

int main( )
{
    int i=10, j=20 ;
    double m=2.1, n=3.5;
    cout<<i<< ' + ' <<j<< ' = ' <<sum(i, j)<<endl;          //replace T by int
    cout<<m<< ' + ' <<n<< ' = ' <<sum(m, n)<<endl;    // replace T by double
    return 0;
}
```