C++ Programming Essentials
Unit 2: Structured Programming

CHAPTER 8: SCOPE, LIFETIME, AND MORE ON FUNCTIONS

DR. ERIC CHOU IEEE SENIOR MEMBER

LECTURE 1

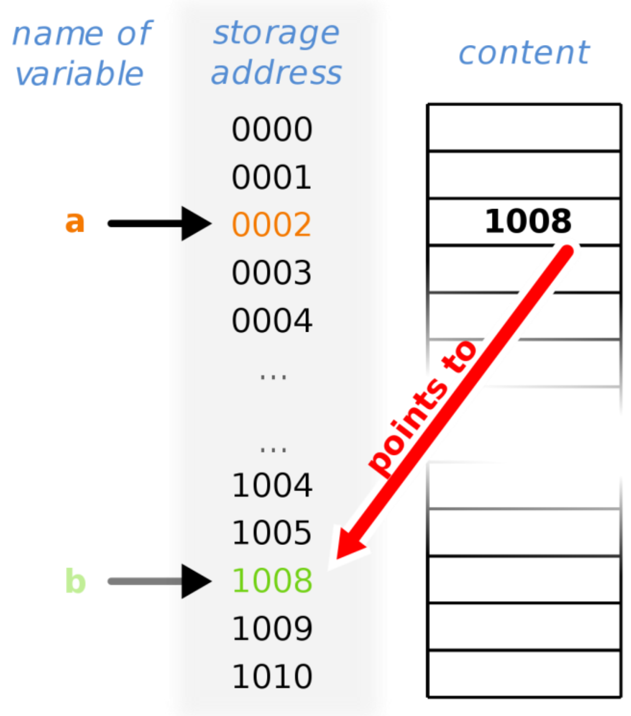
Scope and Binding of Variables

Chapter 8 Topics

- Local Scope vs. Global Scope of an Identifier
- Detailed Scope Rules to Determine which Variables are Accessible in a Block
- Determining the Lifetime of a Variable
- Writing a Value-Returning Function for a Task
- Some Value-Returning Functions with Prototypes in Header Files cctype and cmath
- Creating and Using a Module Structure Chart
- Stub Testing a Program

Binding of Variable to Its Memory Location

- Binding of a variable is the association of a variable to its physical memory location.
- Variable a is bound to the memory location addressed at 0002.
- The time that a is bound to 0002 is called the scope of variable a.



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 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 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.

LECTURE 2

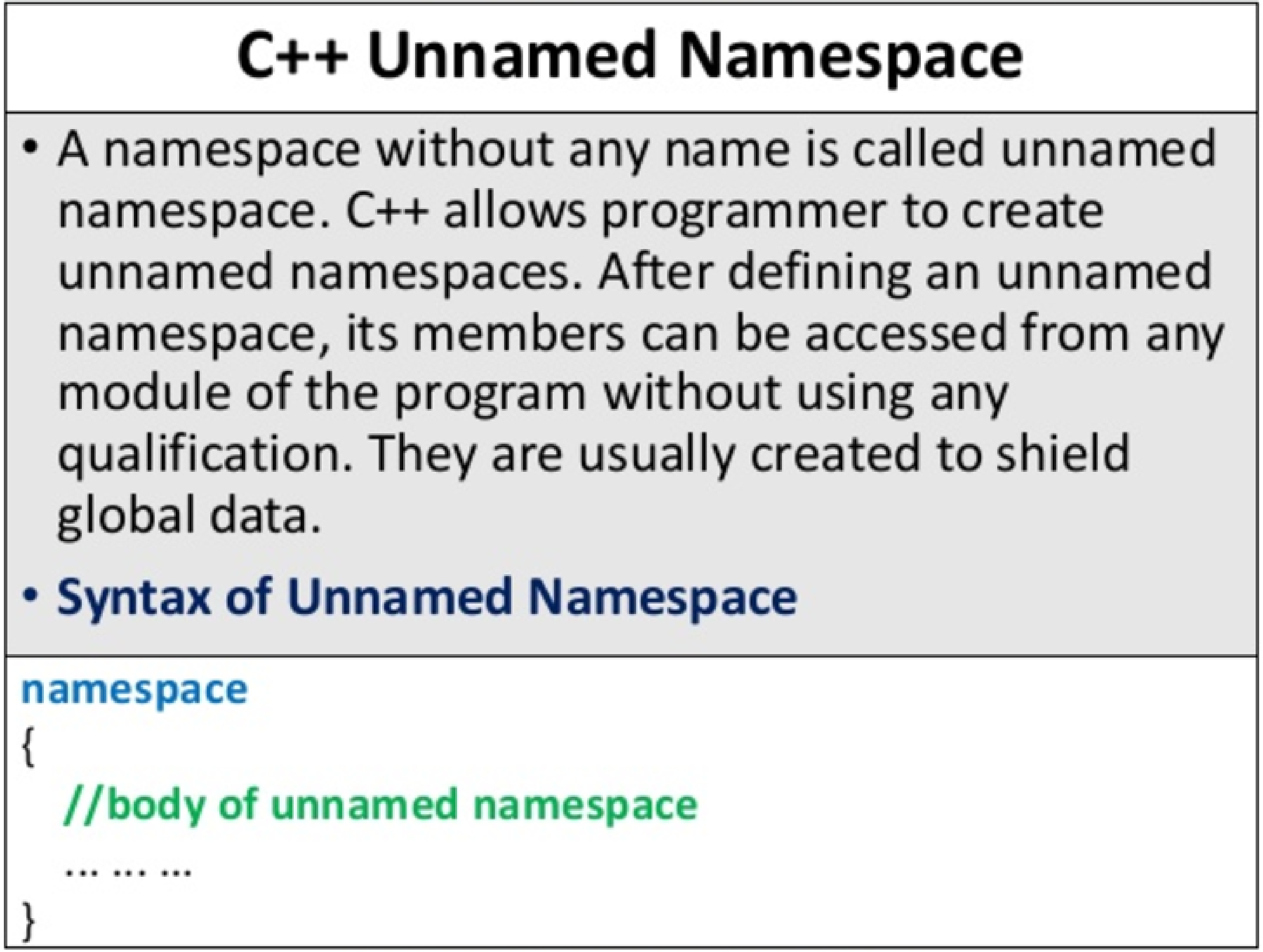
namespace

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

Other Labeled Namespaces

- Namespaces that are just namespaces:
  - C++ **`namespace`**
  - Modula-3 **`module`**
  - Ada **`package`**
  - Java **`package`**
- Namespaces that serve other purposes too:
  - Class definitions in class-based object-oriented languages



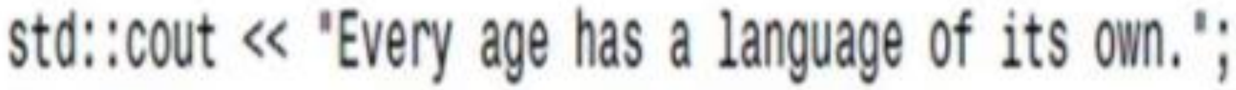
C++ Unnamed Namespace

- A namespace without any name is called unnamed namespace. C++ allows programmer to create unnamed namespaces. After defining an unnamed namespace, its members can be accessed from any module of the program without using any qualification. They are usually created to shield global data.
- **Syntax of Unnamed Namespace**

```
namespace
{
   //body of unnamed namespace
   ... ... ...
}
```

The using Directive

- Various program components such as cout are declared within this namespace.
- We can use it the other way, For example like this



```
std::cout << "Every age has a language of its own.";
```

without using directive

3 Ways to Use Namespace Identifiers

- use a qualified name consisting of the namespace, the scope resolution operator :: and the desired the identifier

```
alpha  =  std :: abs( beta ) ;
```

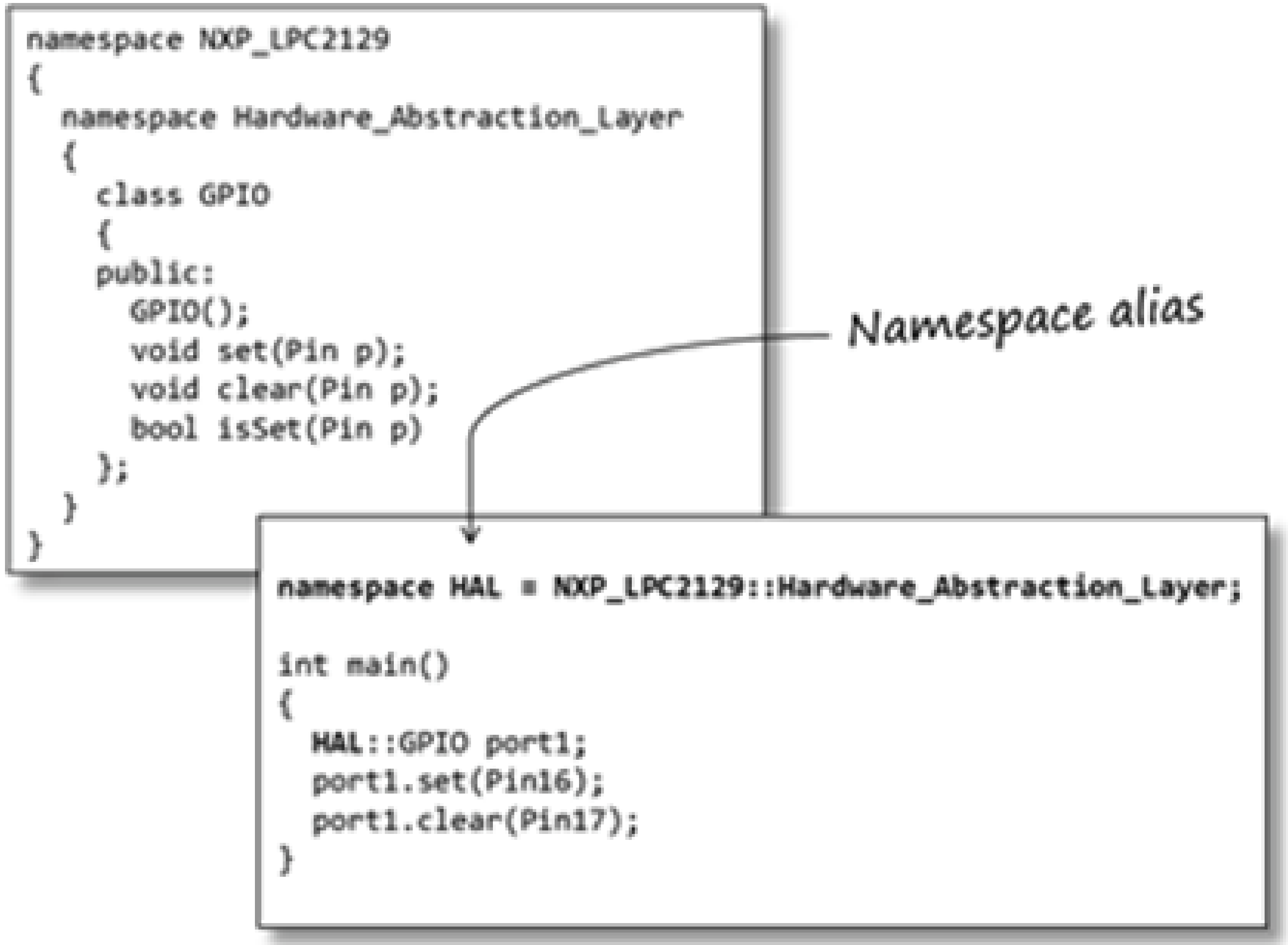
- write a using declaration

```
using  std::abs ;
alpha  = abs( beta );
```

- write a using directive locally or globally

```
using  namespace  std ;
alpha  =  abs( beta );
```

namespace Alias



```
namespace NXP_LPC2129
{
  namespace Hardware_Abstraction_Layer
  {
    class GPIO
    {
    public:
      GPIO();
      void set(Pin p);
      void clear(Pin p);
      bool isSet(Pin p)
    };
  }
}
```

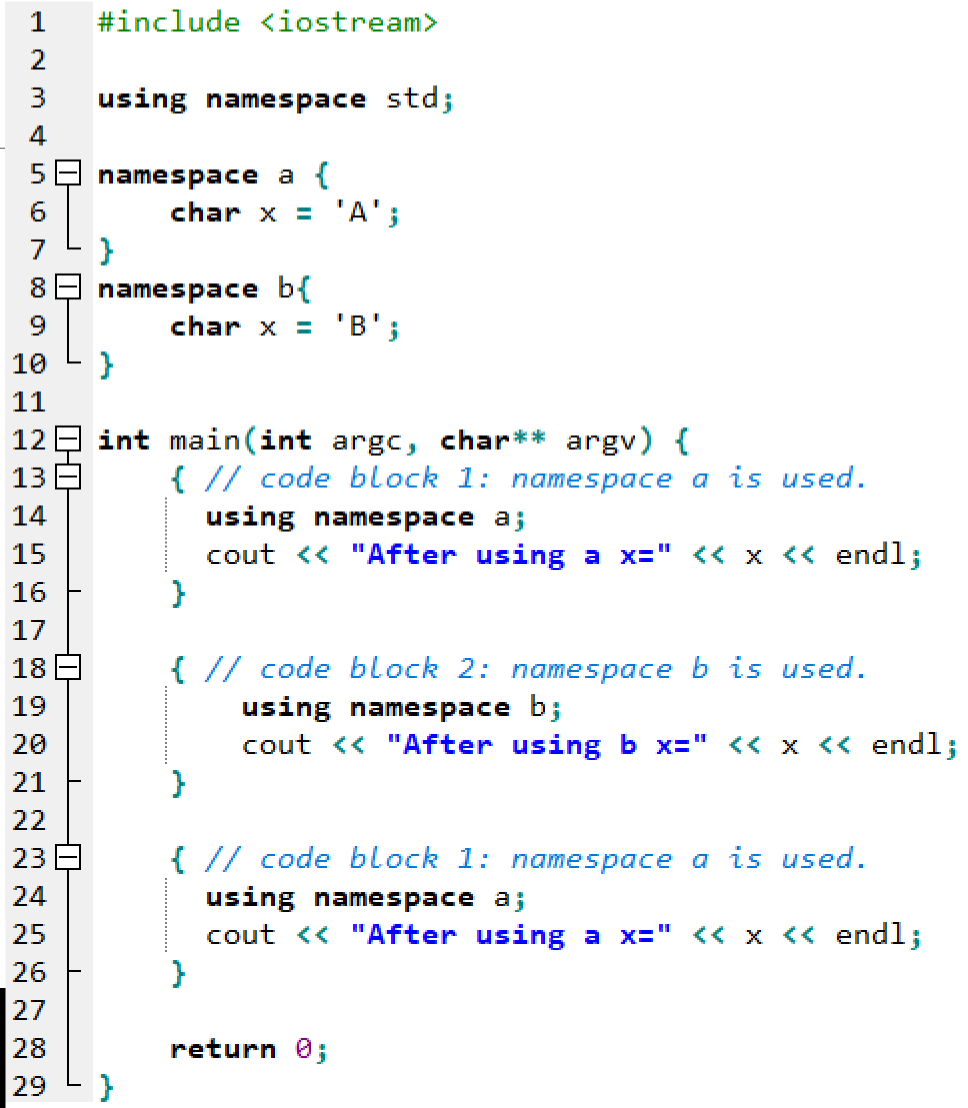
Namespace alias

```
namespace HAL = NXP_LPC2129::Hardware_Abstraction_Layer;

int main()
{
  HAL::GPIO port1;
  port1.set(Pin16);
  port1.clear(Pin17);
}
```

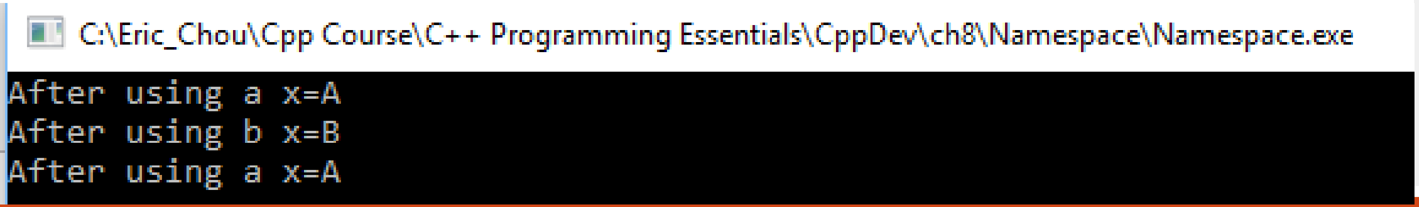
Demo Program:

namespaces.cpp

Go Dev C++!!!



```cpp
#include <iostream>

using namespace std;

namespace a {
    char x = 'A';
}
namespace b{
    char x = 'B';
}

int main(int argc, char** argv) {
    { // code block 1: namespace a is used.
      using namespace a;
      cout << "After using a x=" << x << endl;
    }

    { // code block 2: namespace b is used.
        using namespace b;
        cout << "After using b x=" << x << endl;
    }

    { // code block 1: namespace a is used.
      using namespace a;
      cout << "After using a x=" << x << endl;
    }

    return 0;
}
```



C:\Eric_Chou\Cpp Course\C++ Programming Essentials\CppDev\ch8\Namespace\Namespace.exe

```
After using a x=A
After using b x=B
After using a x=A
```

LECTURE 3

Precedence of Scoping Rules (Binding)

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

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

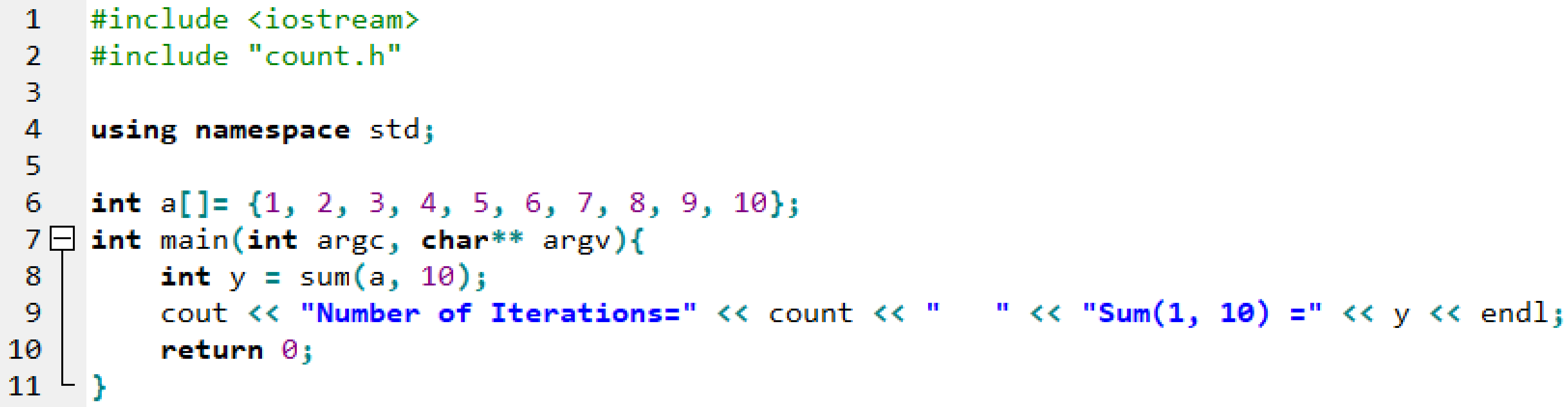
```
int  Square (int n) ;                // function prototype



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

Demo Program:

external.cpp (external means public, static means private)

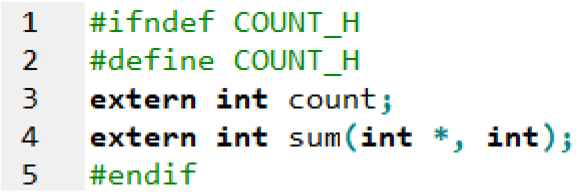
Go Dev C++!!!

external.cpp



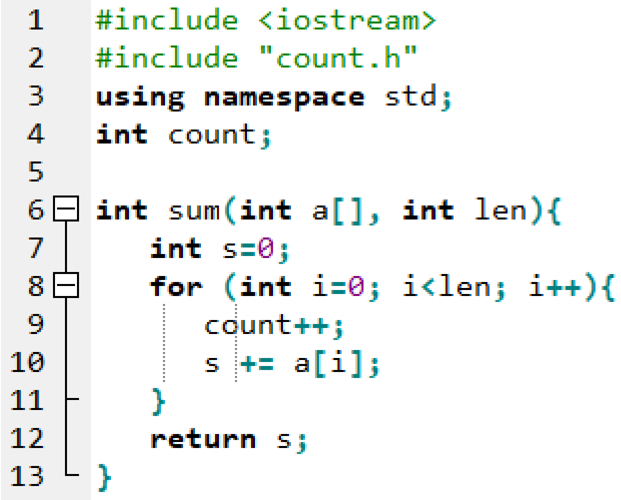
```
#include <iostream>
#include "count.h"

using namespace std;

int a[]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int main(int argc, char** argv){
    int y = sum(a, 10);
    cout << "Number of Iterations=" << count << "    " << "Sum(1, 10) =" << y << endl;
    return 0;
}
```

count.h

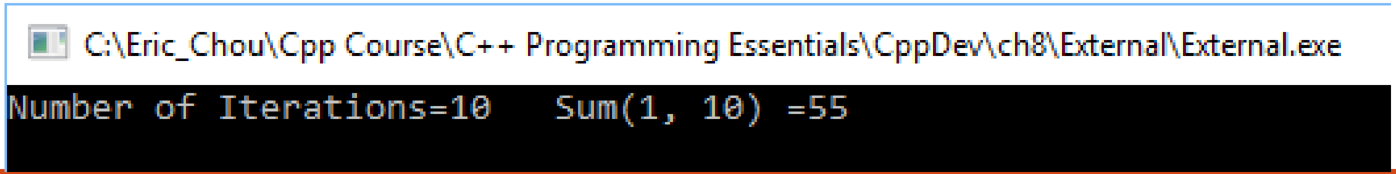


```
#ifndef COUNT_H
#define COUNT_H
extern int count;
extern int sum(int *, int);
#endif
```

count.cpp



```
#include <iostream>
#include "count.h"
using namespace std;
int count;

int sum(int a[], int len){
    int s=0;
    for (int i=0; i<len; i++){
        count++;
        s += a[i];
    }
    return s;
}
```



C:\Eric_Chou\Cpp Course\C++ Programming Essentials\CppDev\ch8\External\External.exe

```
Number of Iterations=10    Sum(1, 10) =55
```

Lifetime of a Variable

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

Lifetime of Local Automatic Variables

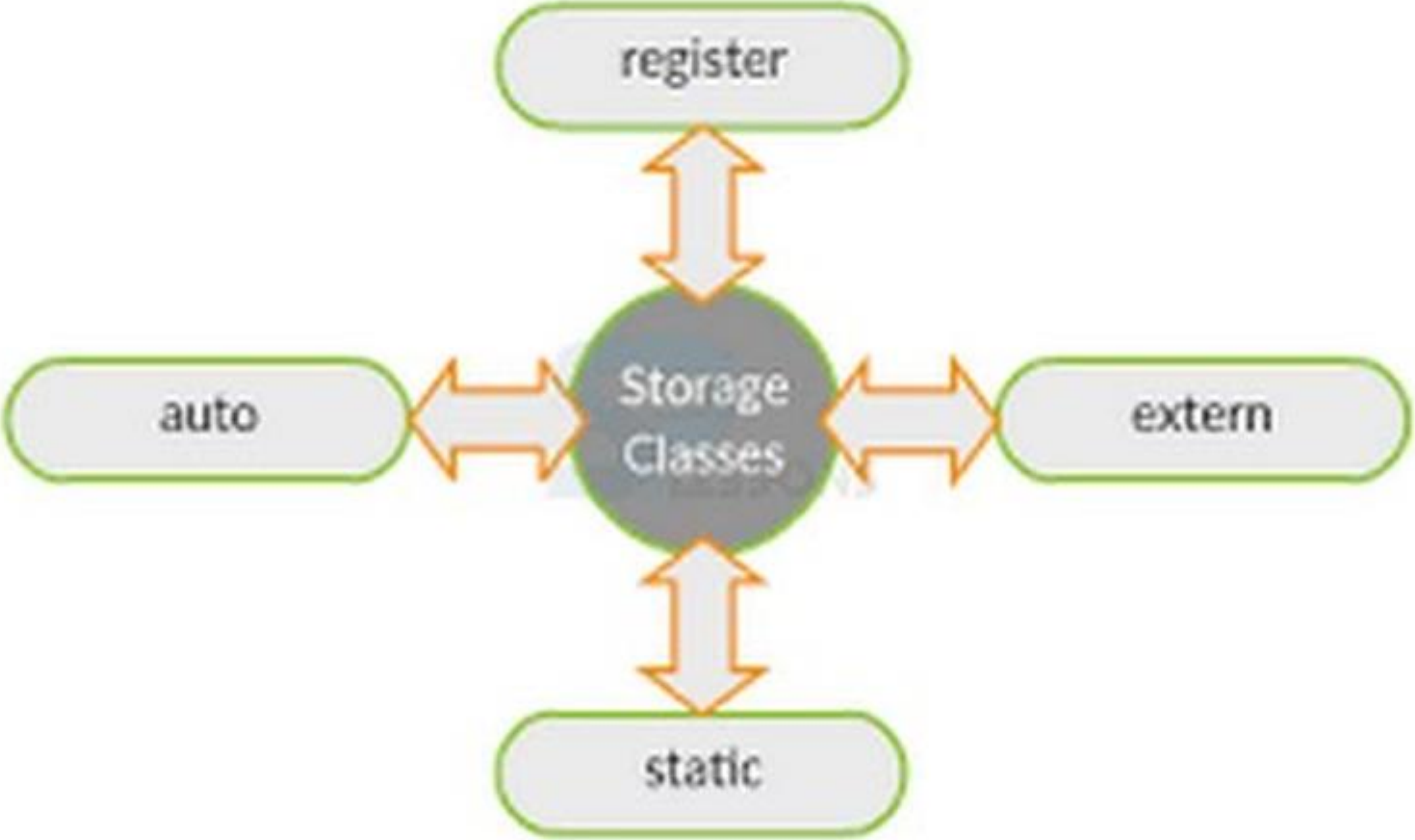
- their storage is created (allocated) when control enters the function
- **local variables are "alive" while function is executing**
- their storage is destroyed (deallocated) when function exits

Lifetime of Global Variables

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

LECTURE 4

Storage Classes: auto and static variables


register
auto
Storage Classes
extern
static

Automatic vs. Static Variable

storage for automatic variable is allocated at block entry and deallocated at block exit

storage for static variable remains allocated throughout execution of the entire program

Note: **static** variable is stored at global static region. **auto** variables are stored in call-stack (in defined in function).

C/C++ Memory Spaces

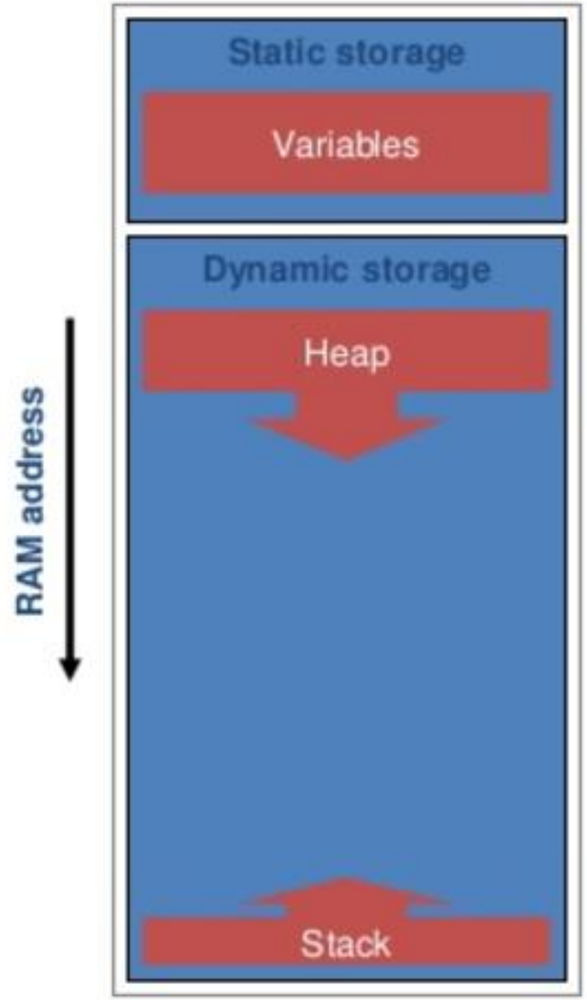
Static memory

- variables outside of functions
- static internal variables
  - keyword static
- program sections [FGA]

Automatic variables

- stack
- register
- keyword auto

Heap

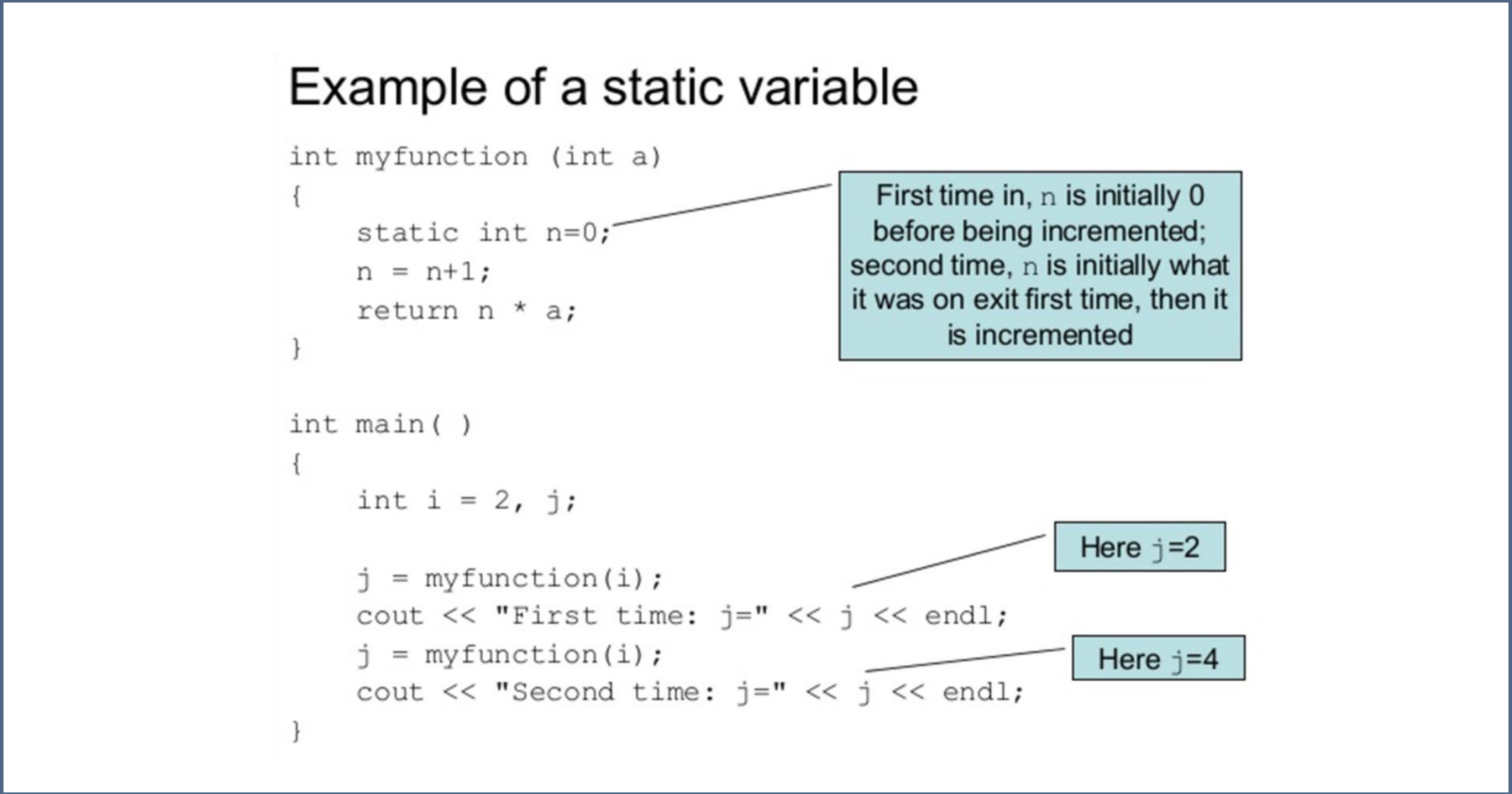


By default

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

Static and Automatic Local Variables

```
int  popularSquare( int  n)
{
    static  int   timesCalled = 0 ;          // initialized only once
    int       result  =  n * n ;             // initialized each time
                                             // when this function is called.
    timesCalled  =  timesCalled + 1 ;
    cout  << “Call # “  << timesCalled  << endl ;
    return  result ;
}
```



Example of a static variable

```
int myfunction (int a)
{
    static int n=0;
    n = n+1;
    return n * a;
}

int main( )
{
    int i = 2, j;

    j = myfunction(i);
    cout << "First time: j=" << j << endl;
    j = myfunction(i);
    cout << "Second time: j=" << j << endl;
}
```

First time in, `n` is initially 0 before being incremented; second time, `n` is initially what it was on exit first time, then it is incremented

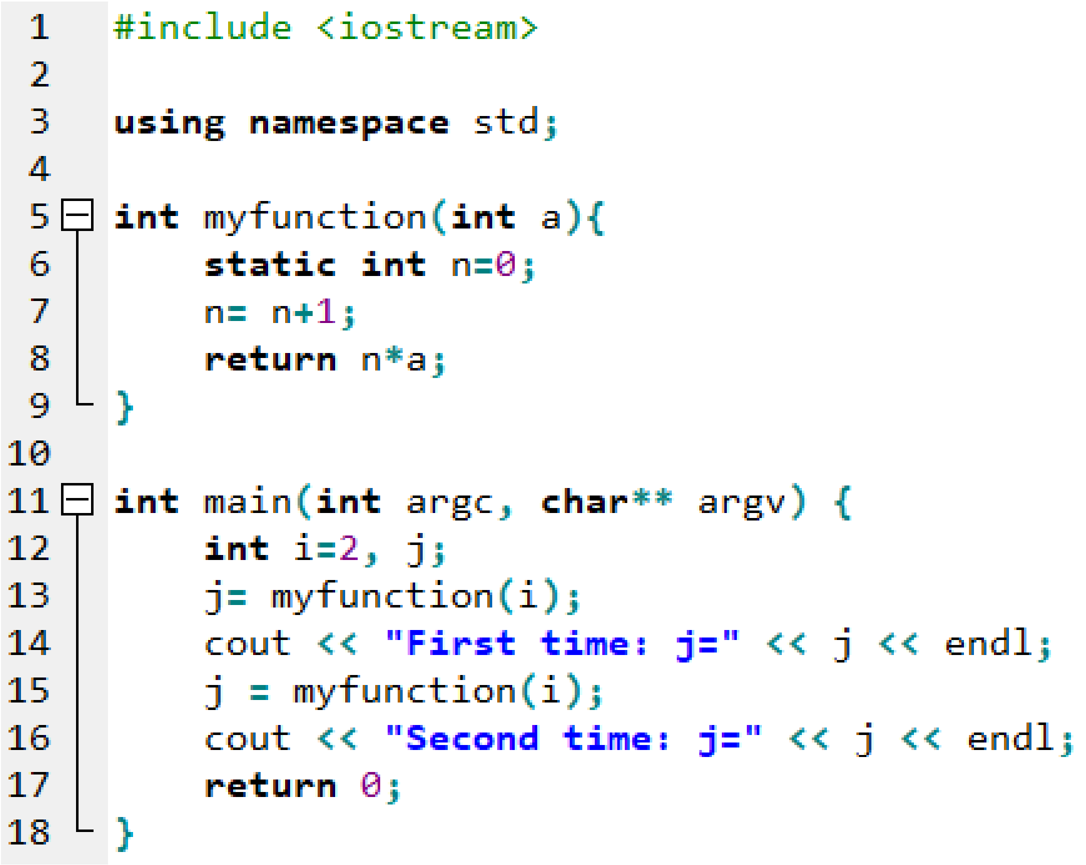
Here `j`=2

Here `j`=4

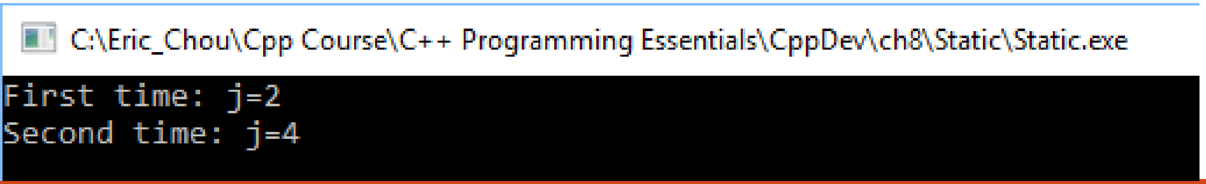
Demo Program:

static.cpp

Go Dev C++!!!



```cpp
#include <iostream>

using namespace std;

int myfunction(int a){
    static int n=0;
    n= n+1;
    return n*a;
}

int main(int argc, char** argv) {
    int i=2, j;
    j= myfunction(i);
    cout << "First time: j=" << j << endl;
    j = myfunction(i);
    cout << "Second time: j=" << j << endl;
    return 0;
}
```

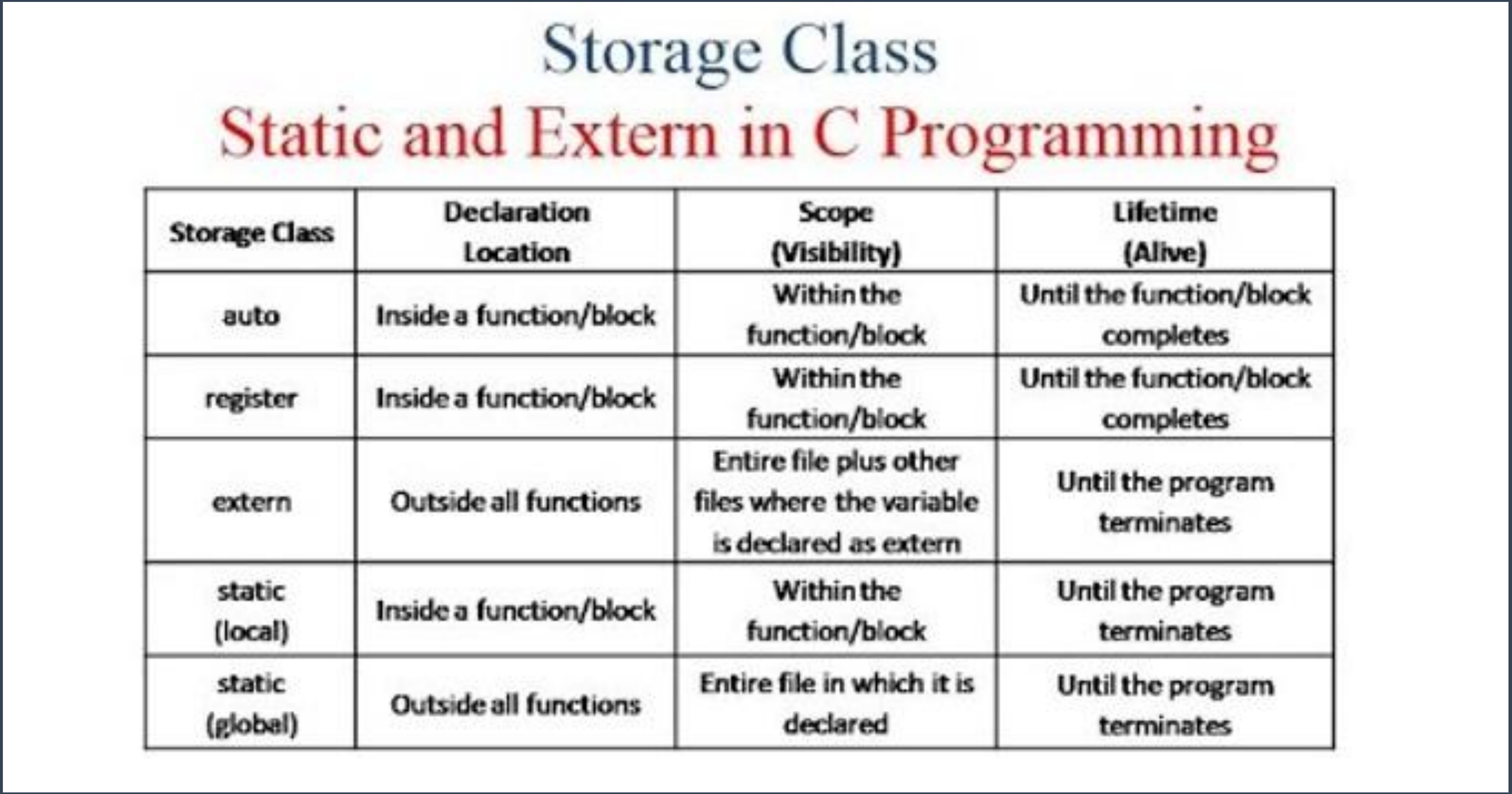


C:\Eric_Chou\Cpp Course\C++ Programming Essentials\CppDev\ch8\Static\Static.exe

```
First time: j=2
Second time: j=4
```

LECTURE 5

Storage Classes: extern and static variables



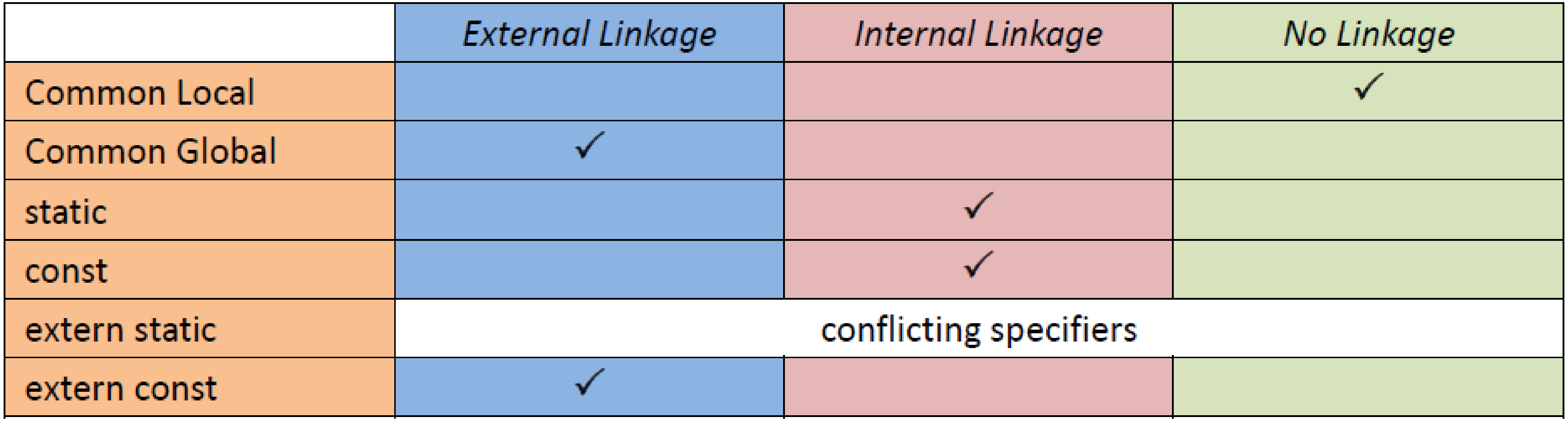
Storage Class

Static and Extern in C Programming

Storage Class	Declaration Location	Scope (Visibility)	Lifetime (Alive)
auto	Inside a function/block	Within the function/block	Until the function/block completes
register	Inside a function/block	Within the function/block	Until the function/block completes
extern	Outside all functions	Entire file plus other files where the variable is declared as extern	Until the program terminates
static (local)	Inside a function/block	Within the function/block	Until the program terminates
static (global)	Outside all functions	Entire file in which it is declared	Until the program terminates

Global static (private) versus extern (public) Variables.

- Definition of static and extern are the same for C and C++.
- For variables, un-specified global variables are considered static. To shared with other module. It must be declared extern at the prototype .h.
- For functions, un-specified functions are considered to be of extern type which is public.
- Private (static) data and public (extern) function are following the rule of data encapsulation.



	External Linkage	*Internal Linkage*	*No Linkage*
Common Local			✓
Common Global	✓		
static		✓	
const		✓	
extern static	conflicting specifiers		
extern const	✓		

Note: in some sense, namespace is used to prevent conflicting variables.

LECTURE 6

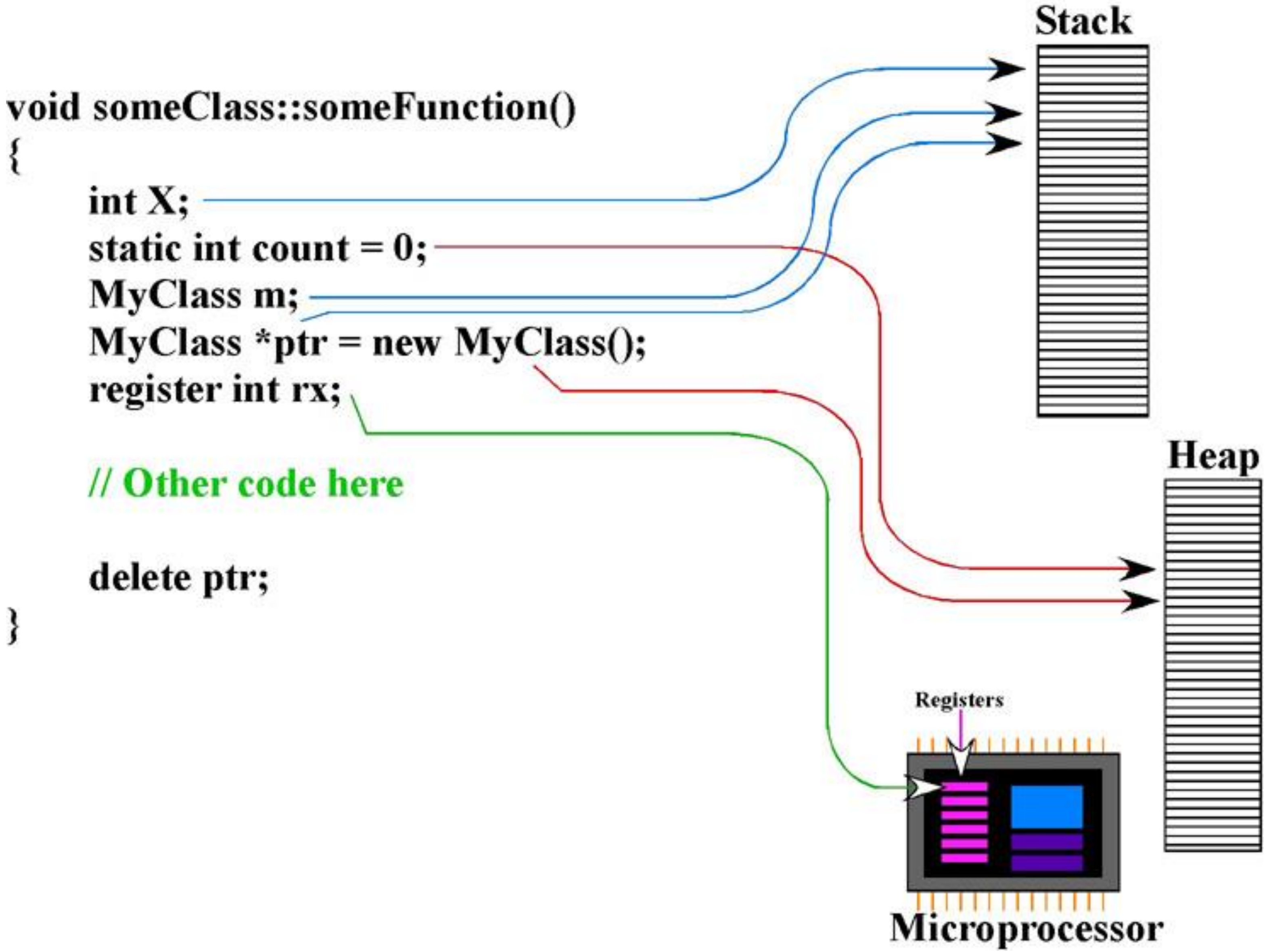
Storage Classes: register

Register variable

- Used to indicate to the compiler that the variable should be stored in a register if possible.
- The scope of register variables is local to the block in which they are declared.
- fast

For example:

```
register int var;
```

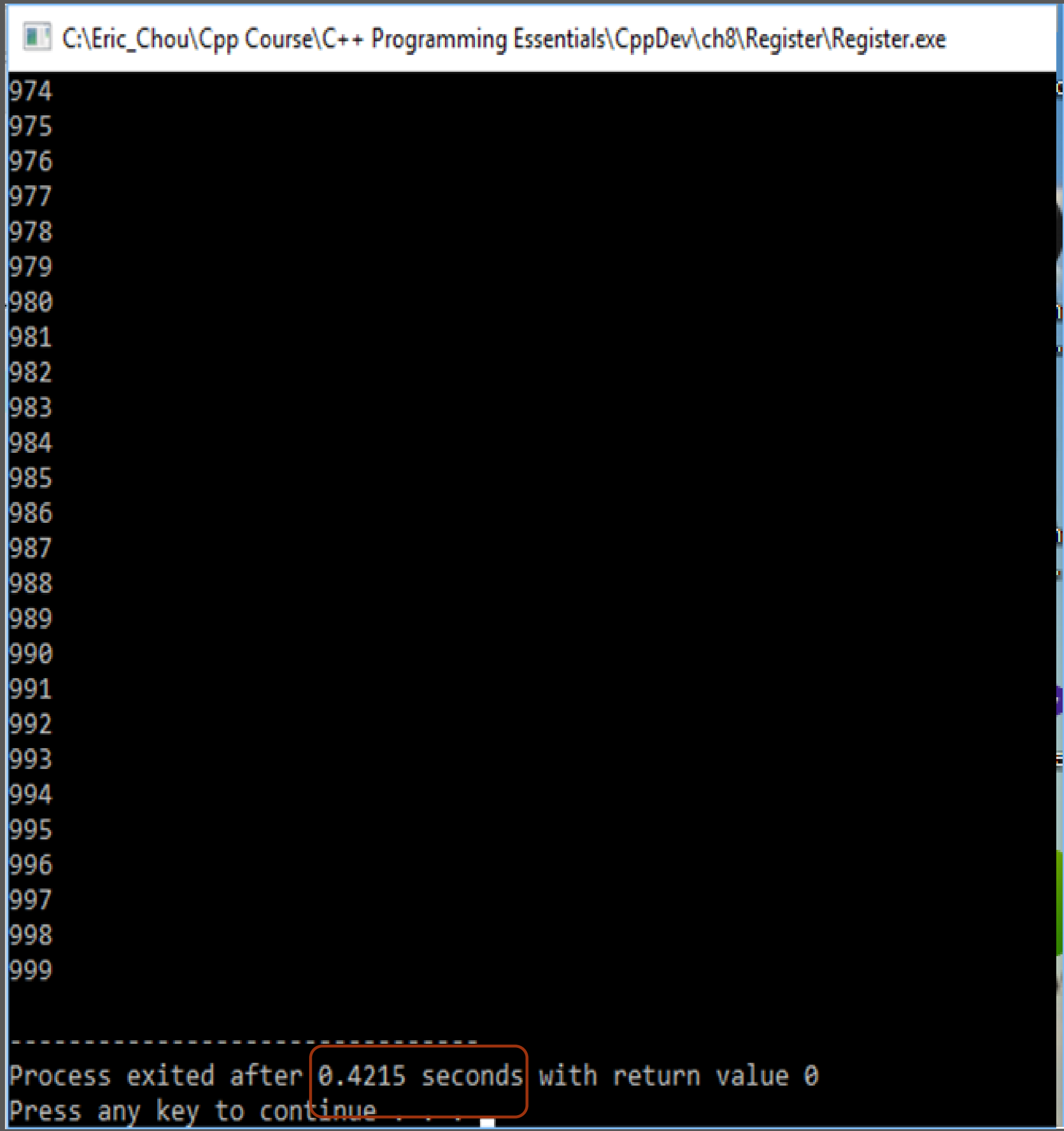


```
void someClass::someFunction()
{
    int X;
    static int count = 0;
    MyClass m;
    MyClass *ptr = new MyClass();
    register int rx;

    // Other code here

    delete ptr;
}
```

Demo Program

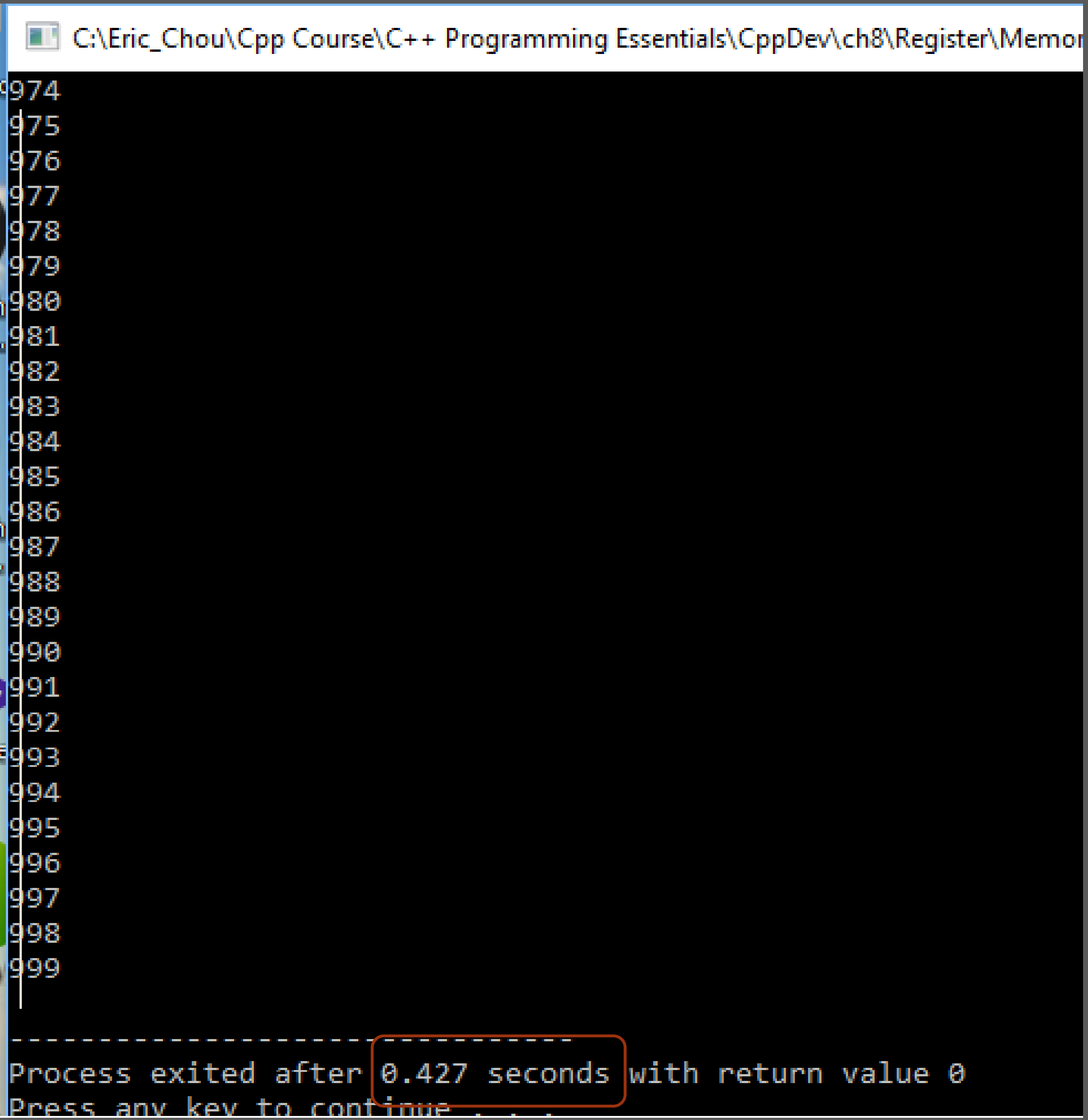
register.cpp

Go Dev C++!!!



C:\Eric_Chou\Cpp Course\C++ Programming Essentials\CppDev\ch8\Register\Register.exe

```
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

--------------------------------
Process exited after 0.4215 seconds with return value 0
Press any key to continue . . .
```



C:\Eric_Chou\Cpp Course\C++ Programming Essentials\CppDev\ch8\Register\Memor

```
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

--------------------------------
Process exited after 0.427 seconds with return value 0
Press any key to continue . . .
```

LECTURE 7

Scope of Argument Variables

Scope of variables

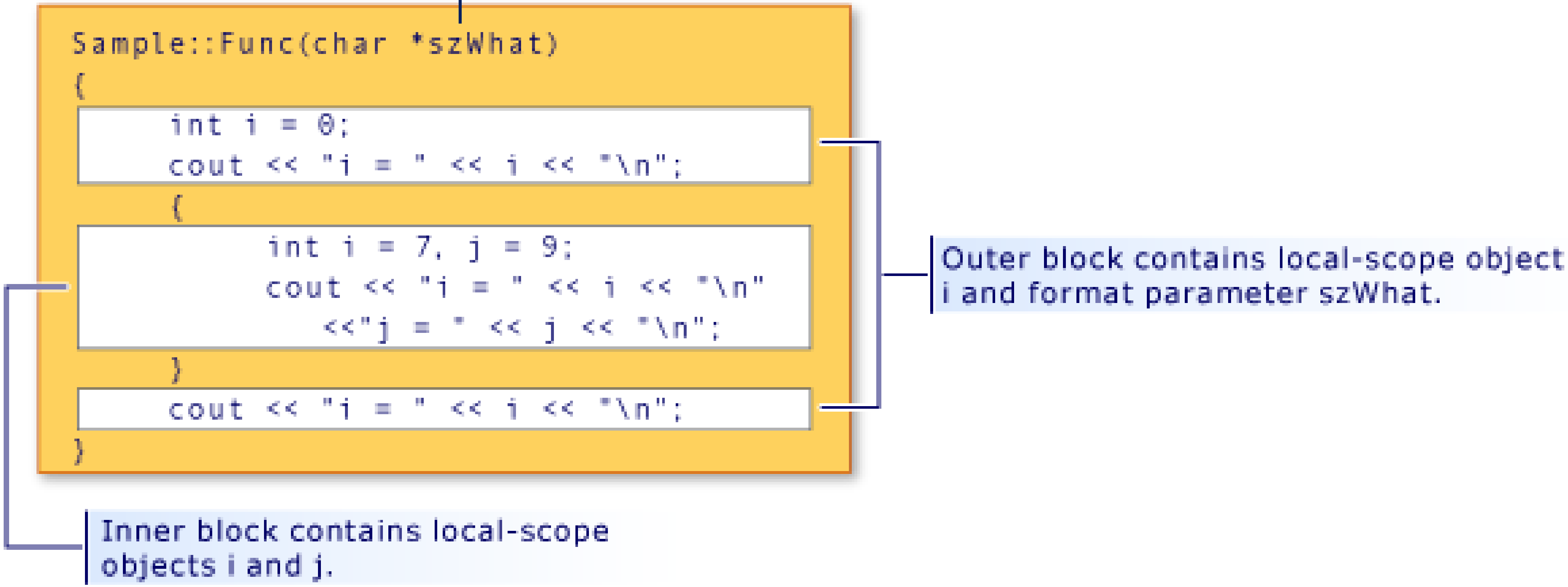
- Arguments to a function and variables declared inside a function are local to that function

```
float calcXcord(float grad1,float Ycept1,float grad2, float Ycept2)
{
   float Xcord;
   Xcord=(Ycept2-Ycept1)/(grad1-grad2);
   return Xcord;
}
int main(int argc, char* argv[])
{
  float gradLine1, gradLine2;
  float Y0Line1, Y0Line2;
  float Xcoord, Ycoord;

  cout<<"input gradient and Y axis value for first line"<<endl;
  cin>>gradLine1>>Y0Line1;
  cout<<"input gradient and Y axis value for second line"<<endl;
  cin>>gradLine2>>Y0Line2;

  Xcoord=calcXcord(gradLine1,Y0Line1,gradLine2,Y0Line2);
}
```

Only known inside function calcXcord

Only known inside function main

Argument Variables’ scope is the whole function (method).



```
Sample::Func(char *szWhat)
{
        int i = 0;
        cout << "i = " << i << "\n";
        {
                int i = 7, j = 9;
                cout << "i = " << i << "\n"
                   <<"j = " << j << "\n";
        }
        cout << "i = " << i << "\n";
}
```

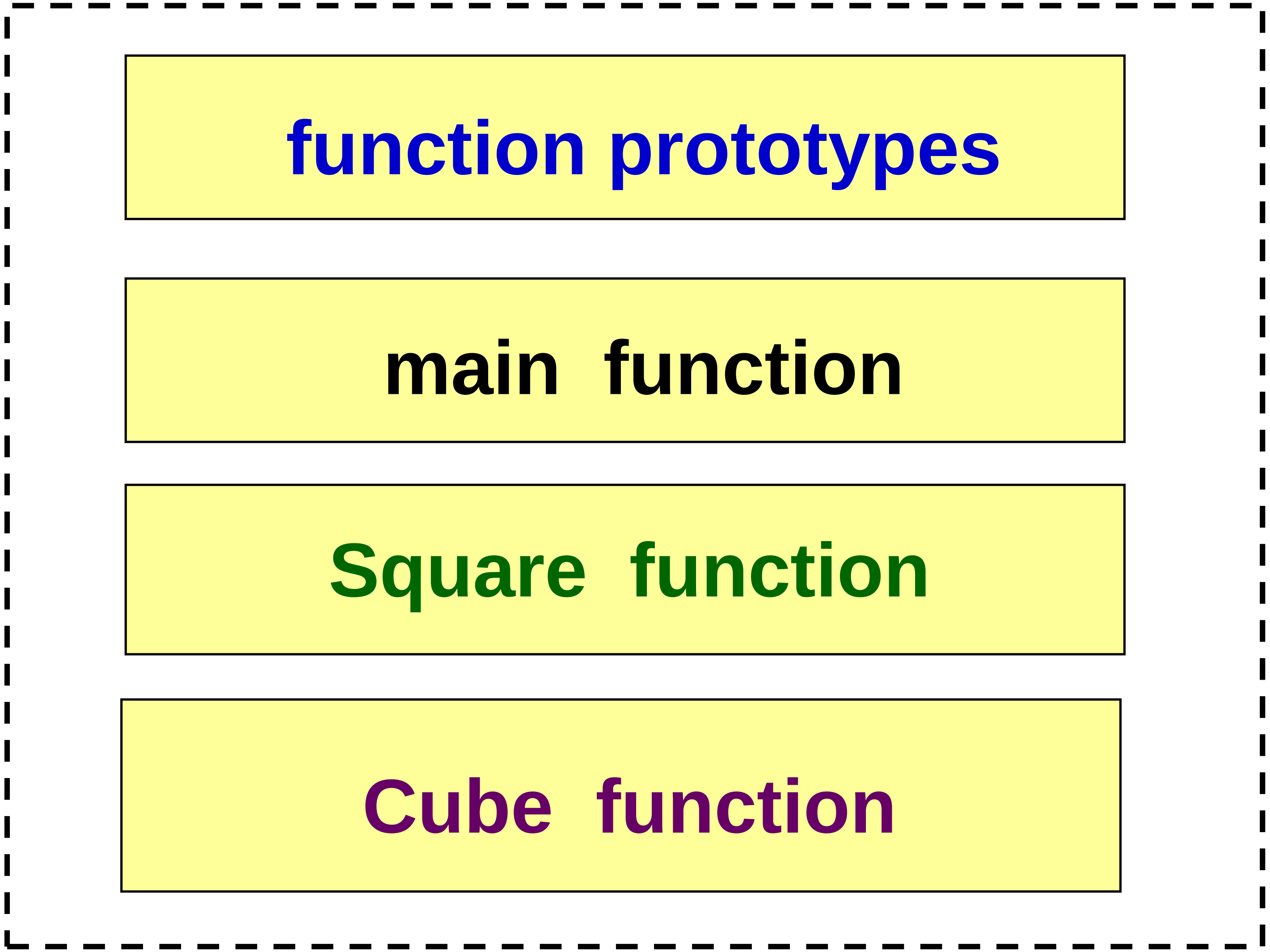
Outer block contains local-scope object i and format parameter szWhat.

Inner block contains local-scope objects i and j.

LECTURE 8

Guidelines for Modular Programming

Program with Several Functions



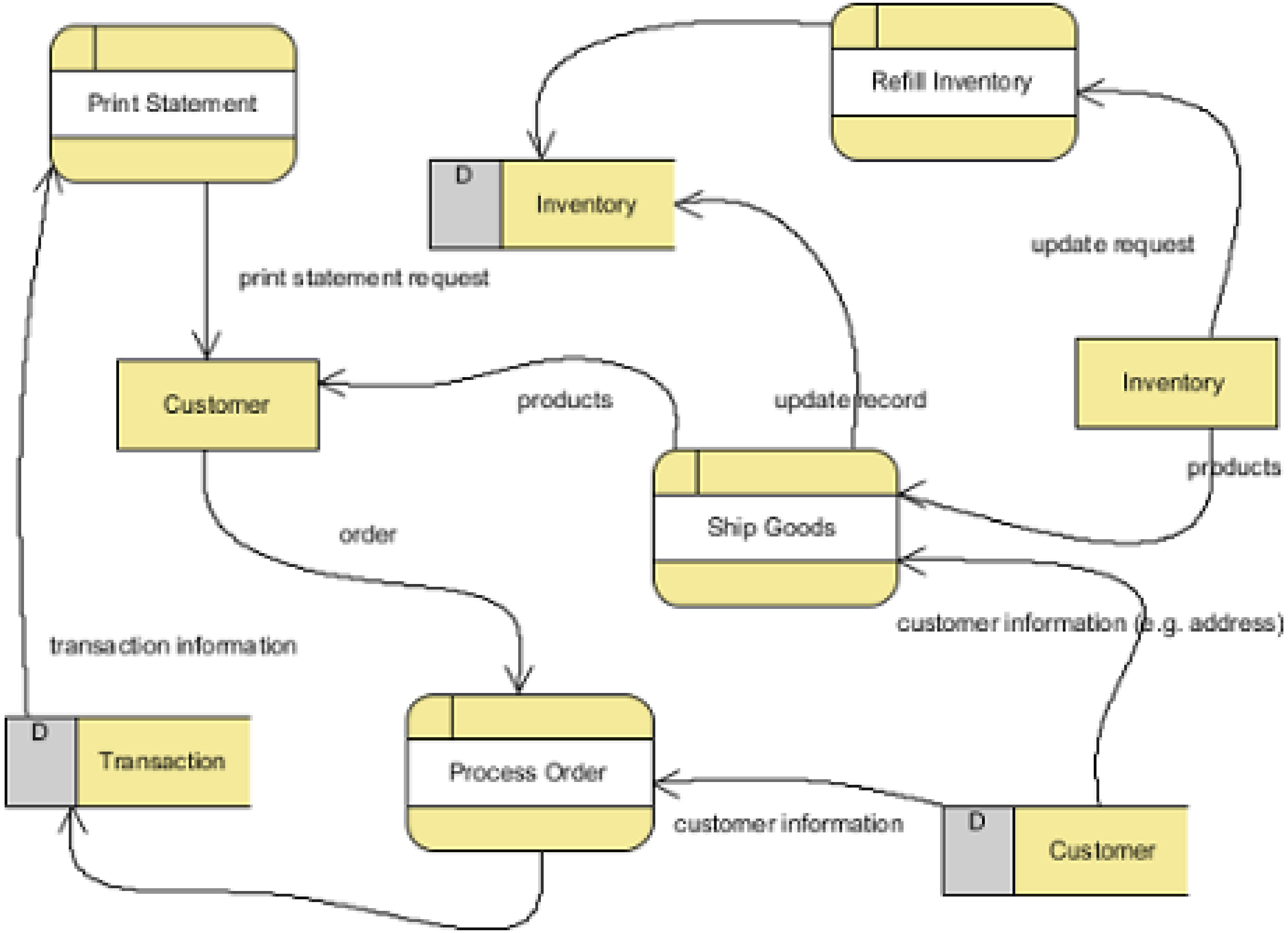
function prototypes

main function

Square function

Cube function

Data Flow Diagram



Data Flow Determines Passing-Mechanism

Passing Mechanism is the Data Flow

Parameter Data Flow	Passing-Mechanism
Incoming /* in */	Pass-by-value
Outgoing /* out */	Pass-by-reference
Incoming/outgoing /* inout */	Pass-by-reference

Value-returning Functions

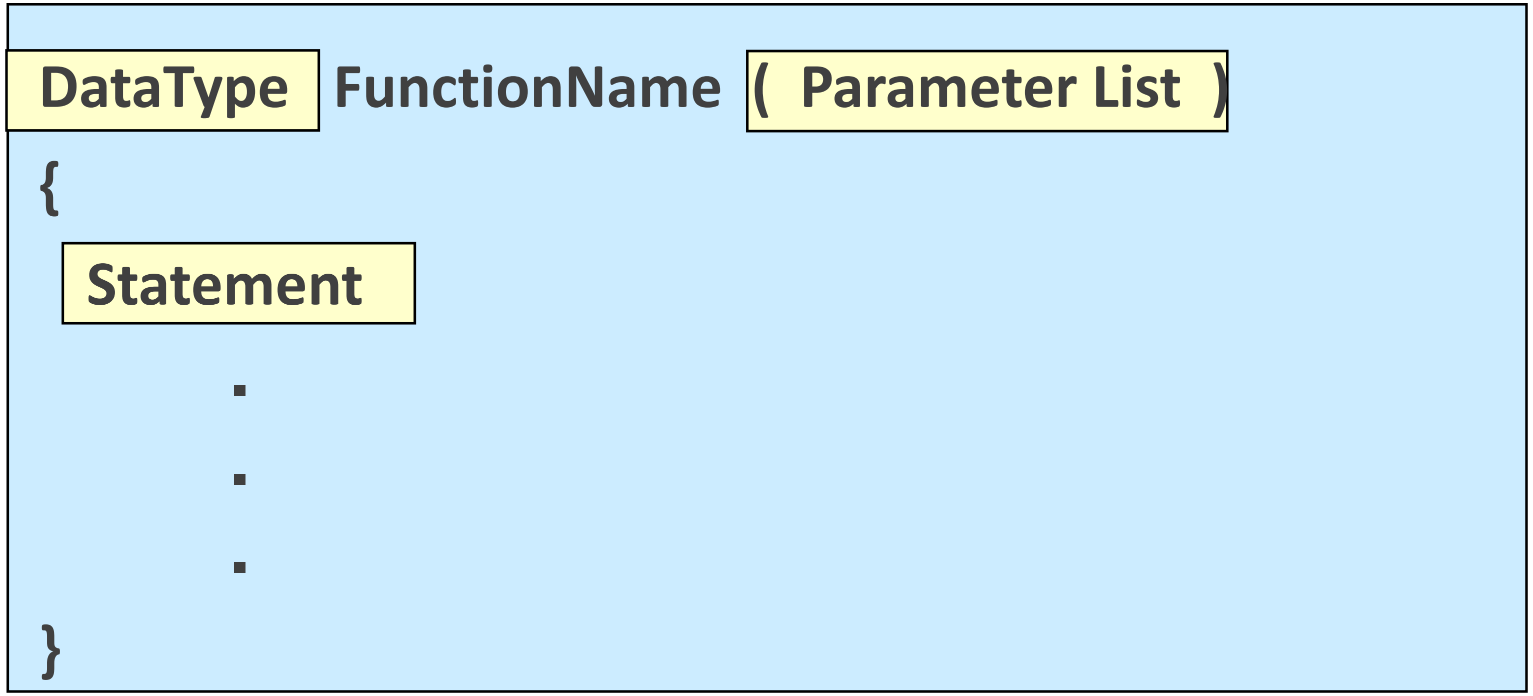
```
#include <iostream>

int  Square ( int ) ;                                        // prototypes
int  Cube ( int ) ;
using namespace std;

int  main (  )
{
     cout  <<  “The square of 27 is “
               <<   Square (27)    <<    endl;     // function call

     cout  <<  “The cube of 27 is “
              <<  Cube (27)     <<    endl;        // function call
    return 0;
}
```

Rest of Program

```
int  Square ( int  n )          // header and body
{
    return   n * n;
}


int  Cube ( int  n )            // header and body
{
   return  n * n * n;
}
```

Syntax Template for Function Definition

Each function should have its own purpose.



```
DataType  FunctionName  ( Parameter List )
{
  Statement
          .
          .
          .
}
```

“What will the function do with your argument?”

The answer determines whether your function parameter should be value or reference as follows . . .

When to Use Value-Returning Functions

1) If it must return more than one value or modify any of the caller's arguments, do not use a value-returning function.

2) If it must perform I/O, do not use a value-returning function.

3) If there is only one value returned, and it is Boolean, a value-returning function is appropriate.

4) If there is only one value returned, and that value will be used immediately in an expression, a value-returning function is appropriate.

5) When in doubt, use a void function. You can recode any value-returning function as a void function by adding an extra outgoing parameter.

6) If both void and value-returning are acceptable, use the one you prefer.

What will the function do with your argument?

IF THE FUNCTION--	FUNCTION PARAMETER IS--
will only use its value	*/* in */* value parameter
will give it a value	*/* out */* reference parameter using &
will change its value	*/* inout */* reference parameter using &

NOTE: I/O stream variables and arrays are exceptions

Some Prototypes in Header File < cmath >

```
double   cos ( double  x );
//  FCTNVAL                    == trigonometric cosine of angle x radians

double   exp ( double  x );
//  FCTNVAL == the value e (2.718 . . .) raised to the power x

double   log ( double x );
//  FCTNVAL == natural (base e) logarithm of x

double   log10 ( double x );
//  FCTNVAL == common (base 10) logarithm of x

double  pow ( double  x, double y );
//  FCTNVAL == x raised to the power y
```

LECTURE 9

Demo Program: Handling Functional Calls (AmoutDue)

Prototype for `float` Function

- called AmountDue( ) with 2 parameters
- The first is type char, the other is type int.

```
float  AmountDue ( char,  int ) ;
```

- This function will find and return the amount due for local phone calls. A char value ‘U’ or ‘L’ indicates Unlimited or Limited service, and the int holds the number of calls made.
- Assume Unlimited service is $40.50 per month.
- Limited service is $19.38 for up to 30 calls, and $.09 per additional call.

```
float  AmountDue (char  kind,   int calls)   // 2 parameters
{
    float    result ;                          // 1 local variable

    const  float  UNLIM_RATE = 40.50,
                  LIM_RATE = 19.38,
                     EXTRA = .09 ;
    if (kind ==‘U’)
       result = UNLIM_RATE ;

    else if ( ( kind == ‘L’ ) && ( calls <= 30) )
        result = LIM_RATE ;

    else
        result = LIM_RATE + (calls - 30) * EXTRA ;

    return result ;
}
```

```
#include  <iostream>
#include  <fstream>
float AmountDue(char, int);                                        // prototype
using namespace std;

int main(void)
{    ifstream  myInfile;
     ofstream  myOutfile;
     int       areaCode, Exchange, calls;
     string    phoneNumber;
     //int       count = 0;
     float     bill;
     char      service;
     myInfile.open("calls.txt"); if (!myInfile.good()) exit(100);
     myOutfile.open("bills.txt");                                  // open files
     while (myInfile >> service >> phoneNumber >> calls){
         bill  =  AmountDue(service, calls) ;                      // function call
         cout << service << " " << phoneNumber << " " << calls << endl;
         myOutfile  <<  phoneNumber << "  "<< bill  << endl;
     }
     myInfile.close();                                             // close files
     myOutfile.close();
     string s;
     ifstream fin;
     fin.open("bills.txt"); if (!fin.good()) exit(100);
     getline(fin, s);
     while (fin){
         cout << s << endl;
         getline(fin, s);
     }
     fin.close();
   return 0;
}
```

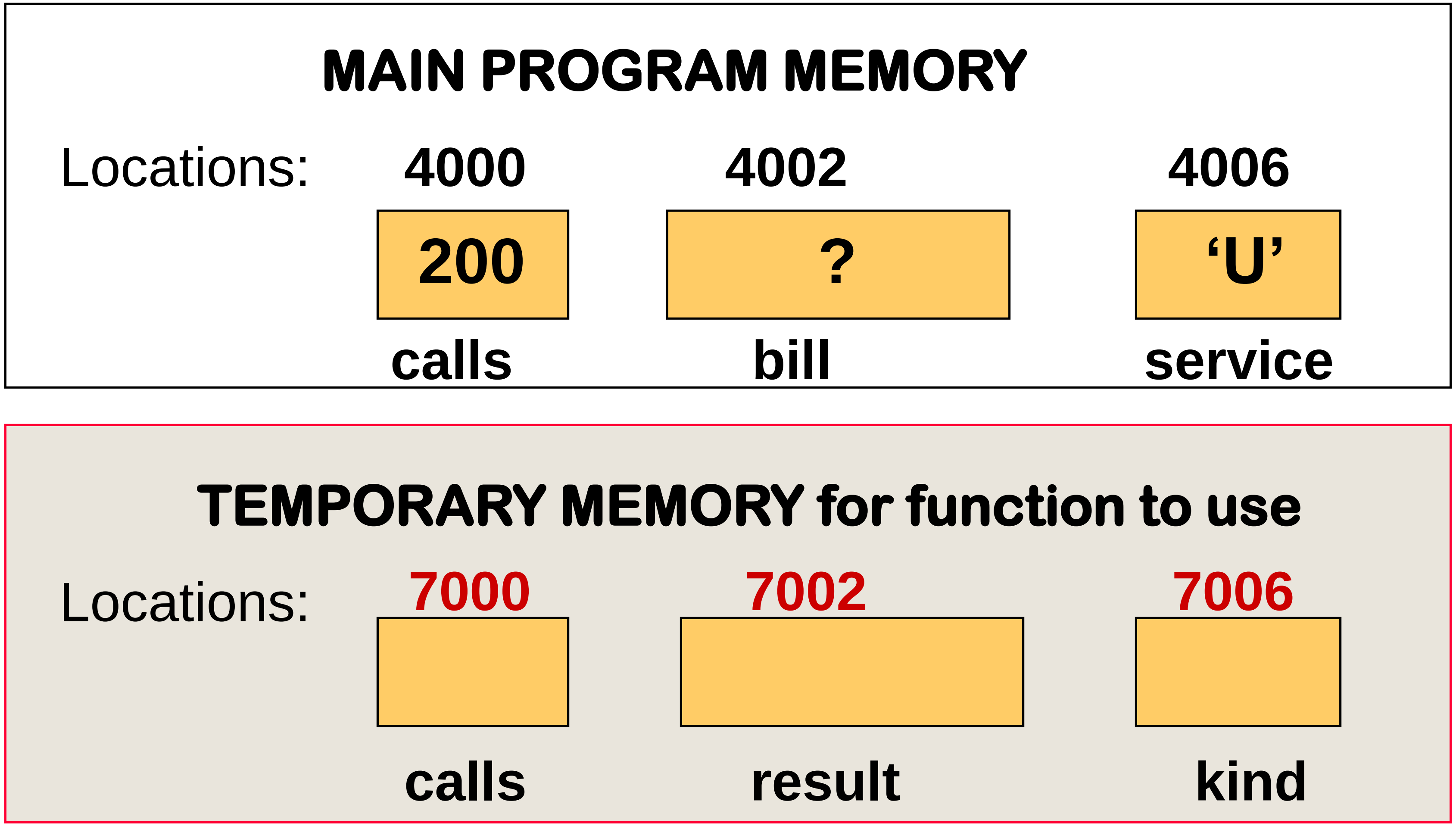
Demo Program

amountdue.cpp

Go Dev C++!!!

To handle the call

```
AmountDue(service, calls)
```



MAIN PROGRAM MEMORY

Locations:	4000	4002	4006
	200	?	‘U’
	calls	bill	service

TEMPORARY MEMORY for function to use

Locations:	7000	7002	7006
	calls	result	kind

Handling Function Call

```
bill = AmountDue(service, calls);
```

- Begins by evaluating each argument
- a copy of the value of each is sent to temporary memory which is created and waiting for it
- the function body determines result
- result is returned and assigned to bill

LECTURE 10

Demo Program: Base Conversion

```
int  Power (    /* in */   int   x ,          // Base number
                /* in */   int   n   )        // Power to raise base to

// This function computes x to the n power

// Precondition:
//    x is assigned  &&  n >= 0  &&  (x to the n) <= INT_MAX
// Postcondition:
//    Function value  ==  x to the n power

{
    int  result ;          // Holds intermediate powers of x
    result  =  1;
    while  ( n > 0 )
    {
     result  =  result  *  x ;
     n-- ;
     }
     return  result ;
}
```

Demo Program:

power.cpp

Go Dev C++!!!

LECTURE 11

Property Check Function: isProperty()

Using `bool` Type with a Loop

```
     . . .
bool  dataOK ;                          // declare Boolean variable
float       temperature ;
     . . .
dataOK = true ;                         // initialize the Boolean variable
while ( dataOK )
{
     .   .   .

   if  ( temperature  >  5000 )
     dataOK = false ;
}
```

A Boolean Function

```
bool   IsTriangle (   /* in */   float  angle1,
                                            /* in */   float  angle2,
                                         /* in */   float  angle3 )
//  Function checks if 3 incoming values add up to 180 degrees,
//  forming a valid triangle
//  PRECONDITION:   angle1, angle2, angle 3 are assigned
//  POSTCONDITION:
//  FCTNVAL         == true, if sum is within 0.000001 of
//                                         180.0 degrees
//                               == false, otherwise
{
  return ( fabs( angle1 + angle2 + angle3 - 180.0 )  <  0.000001 ) ;
}
```

Some Prototypes in Header File < cctype >

```
int   isalpha (char  ch);
//  FCTNVAL          == nonzero, if ch is an alphabet letter
//                              == zero, otherwise
```

```
int   isdigit ( char  ch);
//  FCTNVAL          == nonzero, if ch is a digit ( ‘0’ - ‘9’)
//                              == zero, otherwise
```

```
int  islower ( char  ch );
//  FCTNVAL          == nonzero, if ch is a lowercase letter (‘a’ - ‘z’)
//                              == zero, otherwise
```

```
int  isupper ( char ch);
//  FCTNVAL          == nonzero, if ch is an uppercase letter (‘A’ - ‘Z’)
//                              == zero, otherwise
```

LECTURE 12

Program Integration

Date Formats

American: MM/DD/YYYY

British: DD/MM/YYYY

ISO 8601: YYYY-MM-DD

Top-down Design

Start from the top level description of each function.

Break down a big functionality into smaller functions.

Work on the project calling structure (in tree-structure).

ConvertDates Program

```cpp
// *******************************************************
//  ConvertDates program
//  This program reads dates in American form: mm/dd/yyyy
//  from an input file and writes them to an output file
//  in American, British: dd/mm/yyyy, and ISO: yyyy-mm-dd
//  formats. No data validation is done on the input file.
// *******************************************************

#include <iostream>    // for cout and endl
#include <iomanip>     // for setw
#include <fstream>     // for file I/O
#include <string>      // for string type

using namespace std;

void  Get2Digits( ifstream&, string& );  // prototypes
void  GetYear( ifstream&, string& );
void  OpenForInput( ifstream& );
void  OpenForOutput( ofstream& );
void  Write( ofstream&, string, string, string );
```

ConvertDates Continued

```
int  main( )
{
     string    month;  // Both digits of month
     string    day;    // Both digits of day
     string    year;   // Four digits of year
     ifstream  dataIn; // Input file of dates
     ofstream  dataOut; // Output file of dates

     OpenForInput(dataIn);
     OpenForOutput(dataOut);
                  // Check files
     if ( !dataIn  || !dataOut )
      return 1;
                  // Write headings
     dataOut << setw(20) << “American Format”
             << setw(20) << “British Format”
             << setw(20) << “ISO Format”  << endl << endl;
```

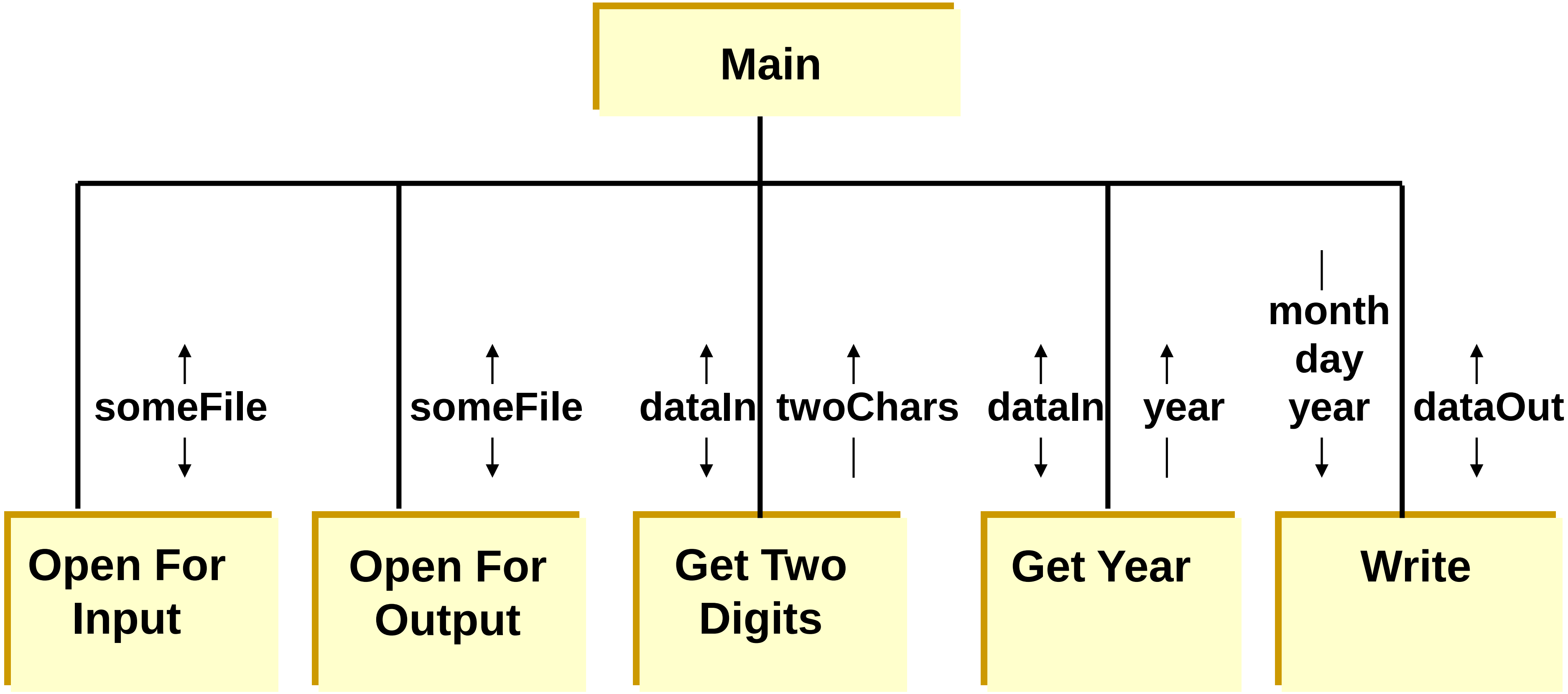
End of main

```
    Get2Digits( dataIn, month ) ;   // Priming read

    while ( dataIn )   // While last read successful
    {
     Get2Digits( dataIn,  day );
     GetYear( dataIn,  year );
     Write( dataOut, month, day, year );
     Get2Digits( dataIn, month );  // Read next data

    }

    return  0;
}
```

Sample Input Data File

```
10/11/1975
1  1  /  2  3    /  1  9  2 6
     5/2/2004
05   /  28    / 1965
7/   3/   19  56
```

Module Structure Chart



LECTURE 13

Bottom-Up Implementation

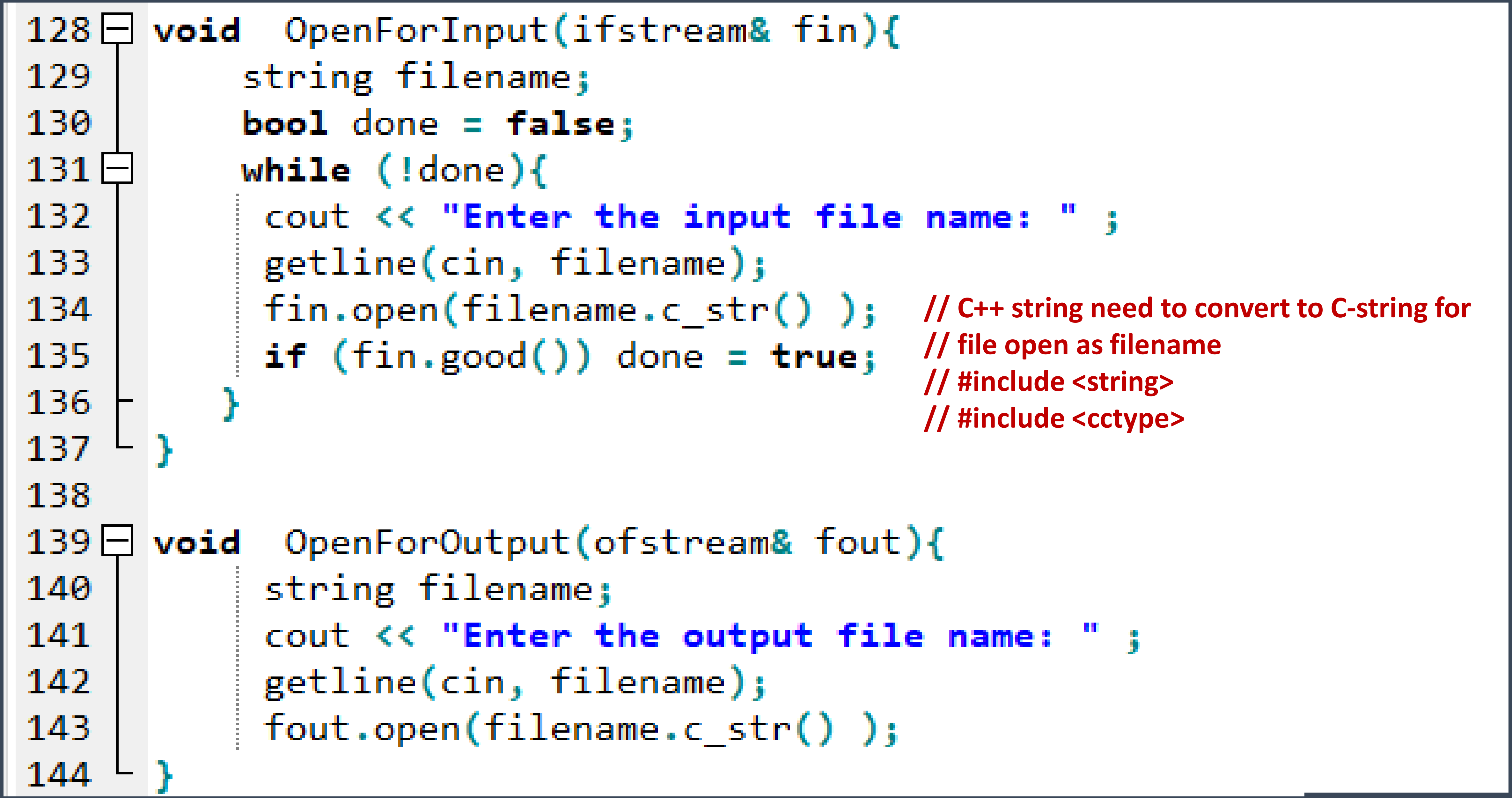
Debug Mode

In the global definition,

```
const bool DEBUG = 1;     // 1 for debug mode, 0 for normal mode
```

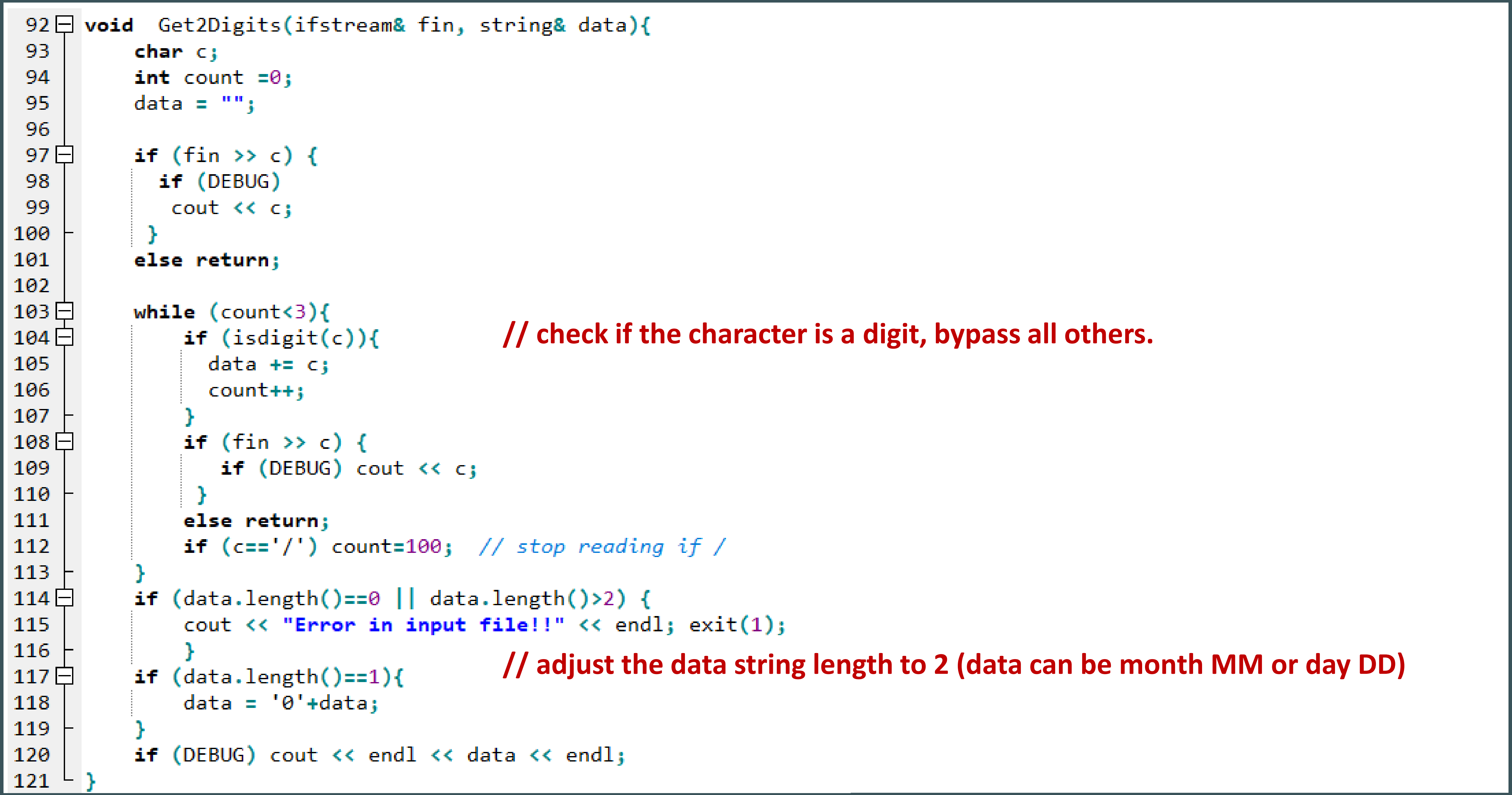
Example:

```
if (dataIn >> c) {                  // get a char c
   if (DEBUG) cout << c;            // echo the char if in debug mode
}
else return;
```



```
void  OpenForInput(ifstream& fin){
    string filename;
    bool done = false;
    while (!done){
     cout << "Enter the input file name: " ;
     getline(cin, filename);
     fin.open(filename.c_str() );
     if (fin.good()) done = true;
   }
}

void  OpenForOutput(ofstream& fout){
     string filename;
     cout << "Enter the output file name: " ;
     getline(cin, filename);
     fout.open(filename.c_str() );
}
```

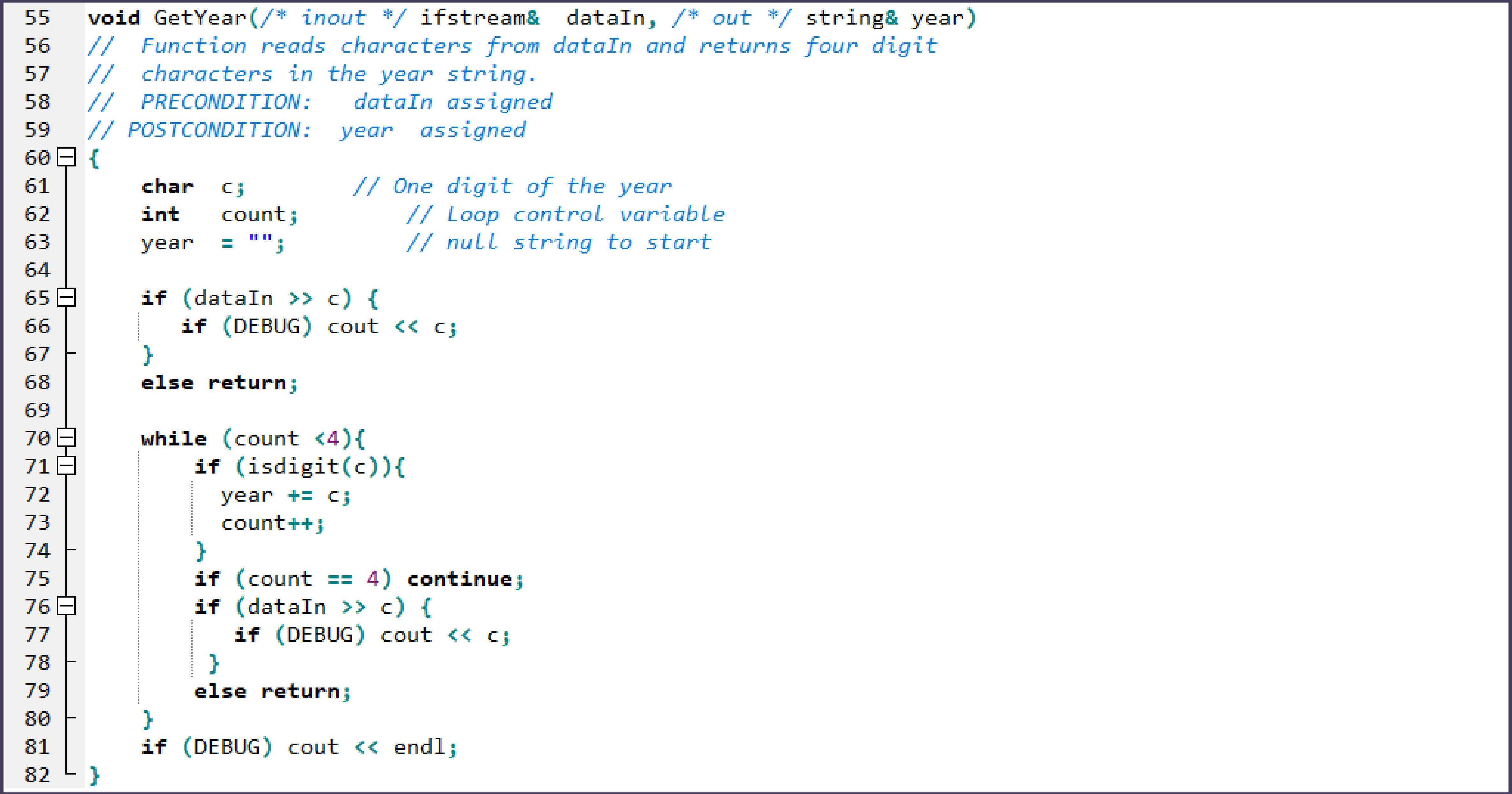
// C++ string need to convert to C-string for
// file open as filename
// #include <string>
// #include <cctype>



```cpp
void  Get2Digits(ifstream& fin, string& data){
    char c;
    int count =0;
    data = "";

    if (fin >> c) {
      if (DEBUG)
       cout << c;
     }
    else return;

    while (count<3){
        if (isdigit(c)){
          data += c;
          count++;
        }
        if (fin >> c) {
           if (DEBUG) cout << c;
         }
        else return;
        if (c=='/') count=100;  // stop reading if /
    }
    if (data.length()==0 || data.length()>2) {
        cout << "Error in input file!!" << endl; exit(1);
        }
    if (data.length()==1){
        data = '0'+data;
    }
    if (DEBUG) cout << endl << data << endl;
}
```

// check if the character is a digit, bypass all others.

// adjust the data string length to 2 (data can be month MM or day DD)



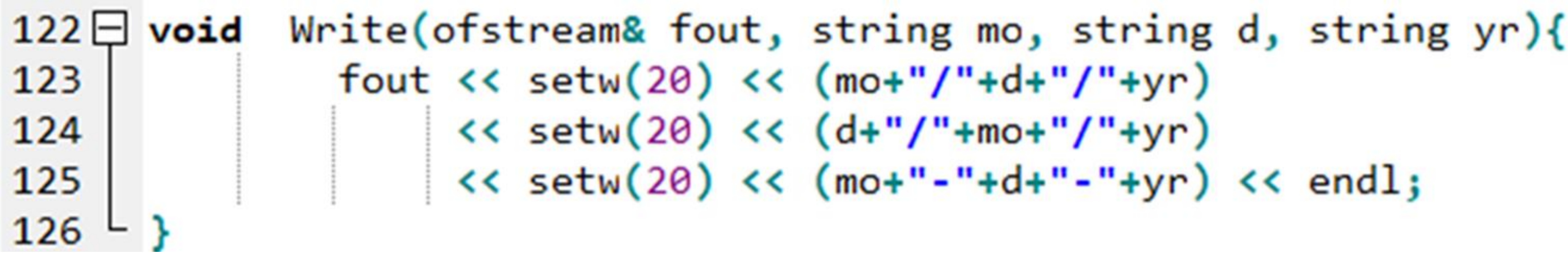
```
void GetYear(/* inout */ ifstream&  dataIn, /* out */ string& year)
//  Function reads characters from dataIn and returns four digit
//  characters in the year string.
//  PRECONDITION:   dataIn assigned
// POSTCONDITION:  year  assigned
{
    char  c;         // One digit of the year
    int   count;         // Loop control variable
    year  = "";          // null string to start

    if (dataIn >> c) {
       if (DEBUG) cout << c;
    }
    else return;

    while (count <4){
        if (isdigit(c)){
          year += c;
          count++;
        }
        if (count == 4) continue;
        if (dataIn >> c) {
           if (DEBUG) cout << c;
         }
        else return;
    }
    if (DEBUG) cout << endl;
}
```

Use Stubs in Testing a Program

- A stub is a dummy function with a very simple body, often just an output statement that this function was reached, and a return value (if any is required) of the correct type.
- Its name and parameter list is the same as a function that will actually be called by the program being tested.

A Stub for Function `GetYear`

```
void   GetYear (   /* inout */   ifstream dataIn,
                            /* out */     string&   year )


//  Stub to test GetYear function in ConvertDates program.
//  PRECONDITION:   dataIn assigned
// POSTCONDITION:  year  assigned
{
  cout  << “GetYear was called.  Returning \”1948\”.”  << endl ;
  year  = “1948” ;
}
```



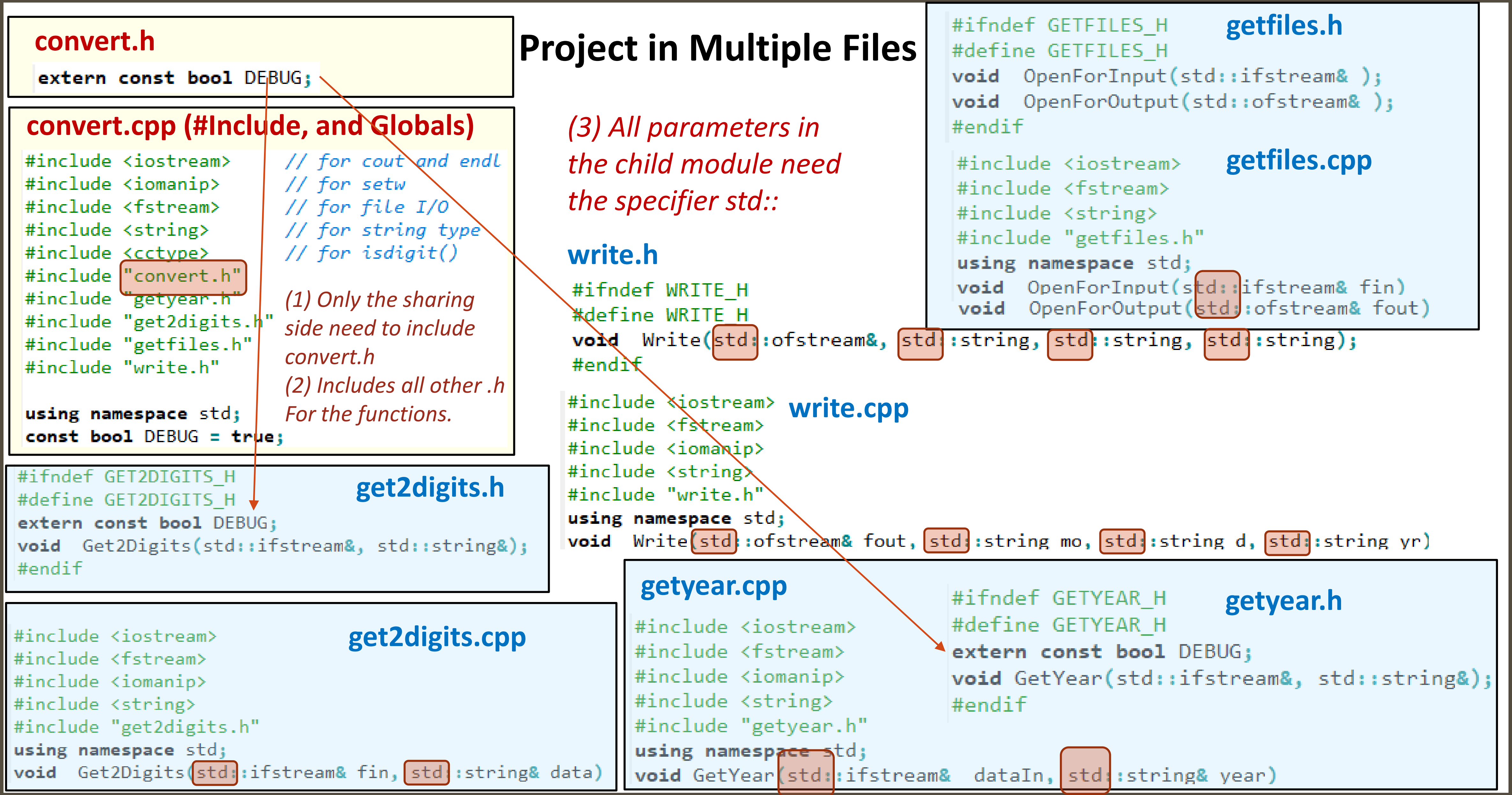
```
void  Write(ofstream& fout, string mo, string d, string yr){
        fout << setw(20) << (mo+"/"+d+"/"+yr)
             << setw(20) << (d+"/"+mo+"/"+yr)
             << setw(20) << (mo+"-"+d+"-"+yr) << endl;
}
```

Demo Program

convertdate.cpp (single module format)

Go Dev C++!!!

Project in Multiple Files



convert.h

```
extern const bool DEBUG;
```

convert.cpp (#Include, and Globals)

```
#include <iostream>      // for cout and endl
#include <iomanip>       // for setw
#include <fstream>       // for file I/O
#include <string>        // for string type
#include <cctype>        // for isdigit()
#include "convert.h"
#include "getyear.h"
#include "get2digits.h"
#include "getfiles.h"
#include "write.h"

using namespace std;
const bool DEBUG = true;
```

(1) Only the sharing side need to include convert.h

(2) Includes all other .h For the functions.

(3) All parameters in the child module need the specifier std::

get2digits.h

```
#ifndef GET2DIGITS_H
#define GET2DIGITS_H
extern const bool DEBUG;
void  Get2Digits(std::ifstream&, std::string&);
#endif
```

get2digits.cpp

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include "get2digits.h"
using namespace std;
void  Get2Digits(std::ifstream& fin, std::string& data)
```

write.h

```
#ifndef WRITE_H
#define WRITE_H
void  Write(std::ofstream&, std::string, std::string, std::string);
#endif
```

write.cpp

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include "write.h"
using namespace std;
void  Write(std::ofstream& fout, std::string mo, std::string d, std::string yr)
```

getfiles.h

```
#ifndef GETFILES_H
#define GETFILES_H
void  OpenForInput(std::ifstream& );
void  OpenForOutput(std::ofstream& );
#endif
```

getfiles.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include "getfiles.h"
using namespace std;
void  OpenForInput(std::ifstream& fin)
void  OpenForOutput(std::ofstream& fout)
```

getyear.cpp

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include "getyear.h"
using namespace std;
void GetYear(std::ifstream&  dataIn, std::string& year)
```

getyear.h

```
#ifndef GETYEAR_H
#define GETYEAR_H
extern const bool DEBUG;
void GetYear(std::ifstream&, std::string&);
#endif
```

Demo Program

convert.cpp (multiple module format)

Go Dev C++!!!