Chapter 2

Logic Simulation with C/C++ Programming Language

Zainalabedin Navabi

Logic Simulation with C/C++

- Procedural Languages for Hardware Modeling
- Types and Operators for Logic Modeling
- Basic Logic Simulation
  - Logic functions
  - Function overloading
  - Passing logic functions
  - Using default values
  - Building higher level structures
  - Handling 4-value logic
  - Logic vector
  - Sequential circuit modeling
  - Using pointers for logic vectors
- Enhanced logic simulation with timing
  - Using struct for timing and logic
  - Gates that handle timing
  - Utility functions
  - Timing in logic structures
  - Overloading logical operators
  - Using Boolean expressions
- More Functions for Wires and Gates
  - Gate classes
  - Carrier generic modeling
  - Compatible scalar and vector

Logic Simulation with C/C++

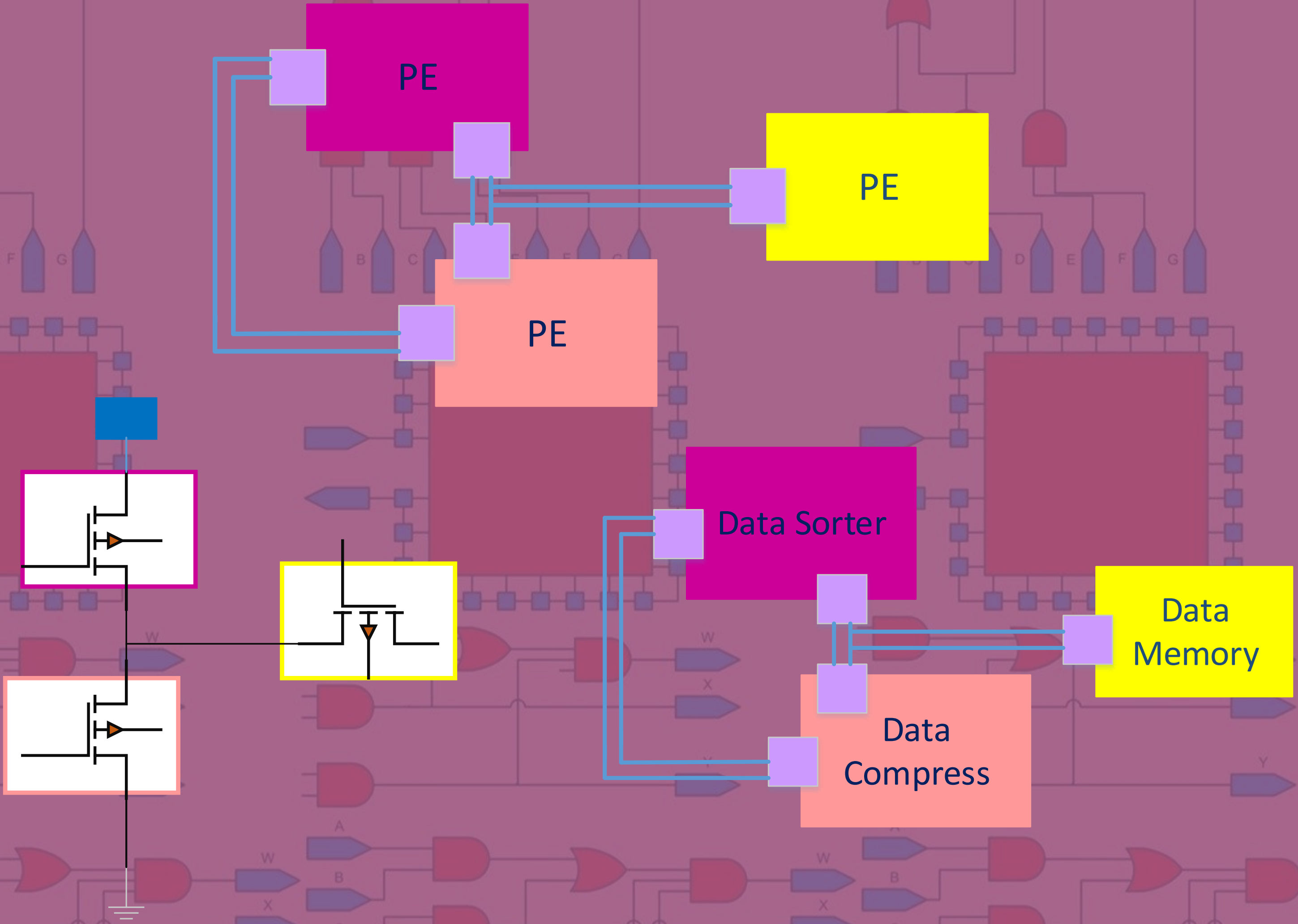
- Containing Event Based Timing
  - To include in wires
  - To include in gates
- Gate-based structures
- Gate pointers and objects
- Wire and gate vectors

- Inheritance in Logic Structures
  - A generic gate definition
  - Gates to include timing
  - Building structures from objects

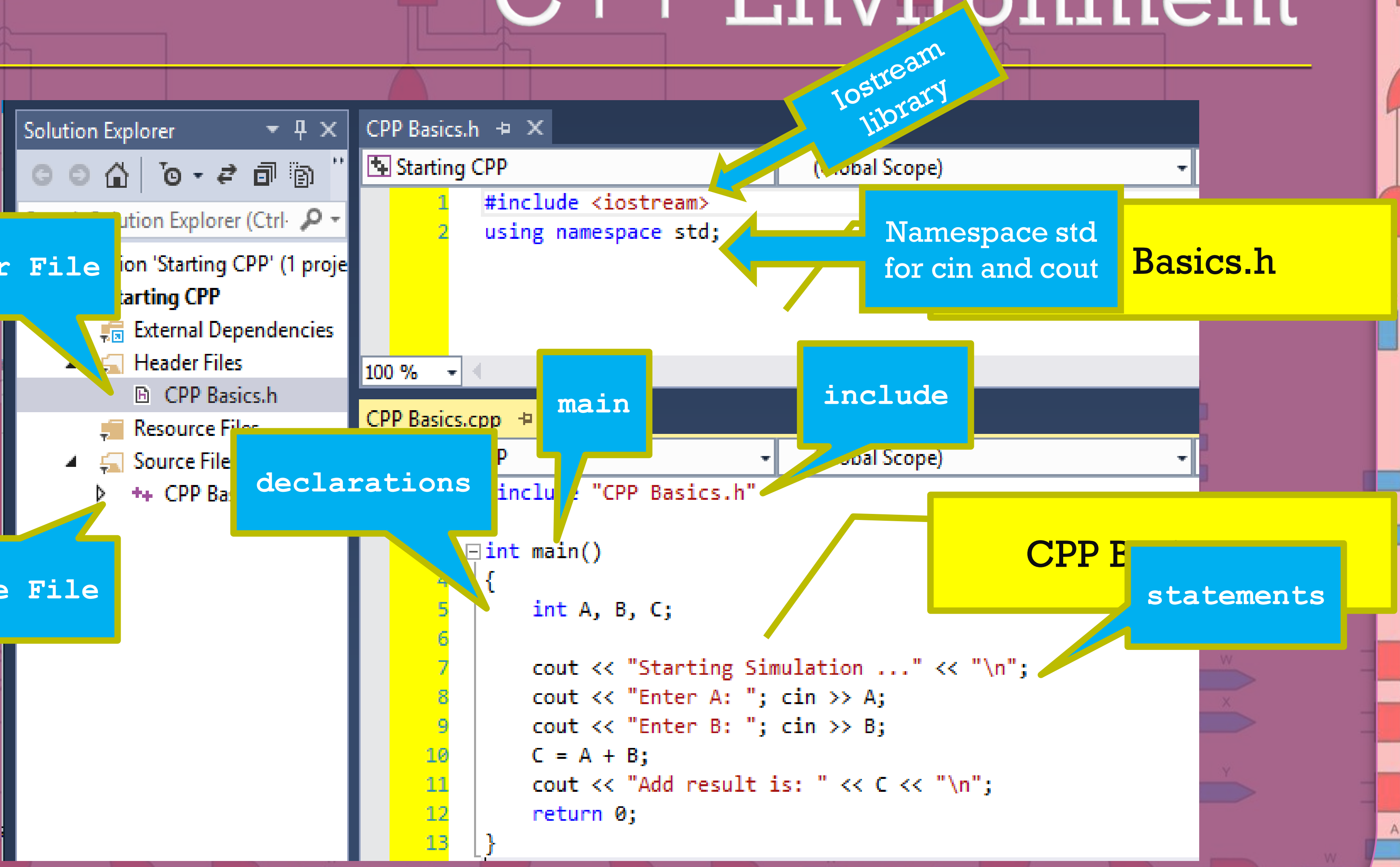
- Hierarchal Modeling of Digital Components
  - Wire functionalities
  - Gate functionalities
  - Polymorphic gate base
  - Virtual functions
  - Functions overwriting
  - Flip flop description hierarchal

Logic Simulation with C/C++

Hardware Modeling



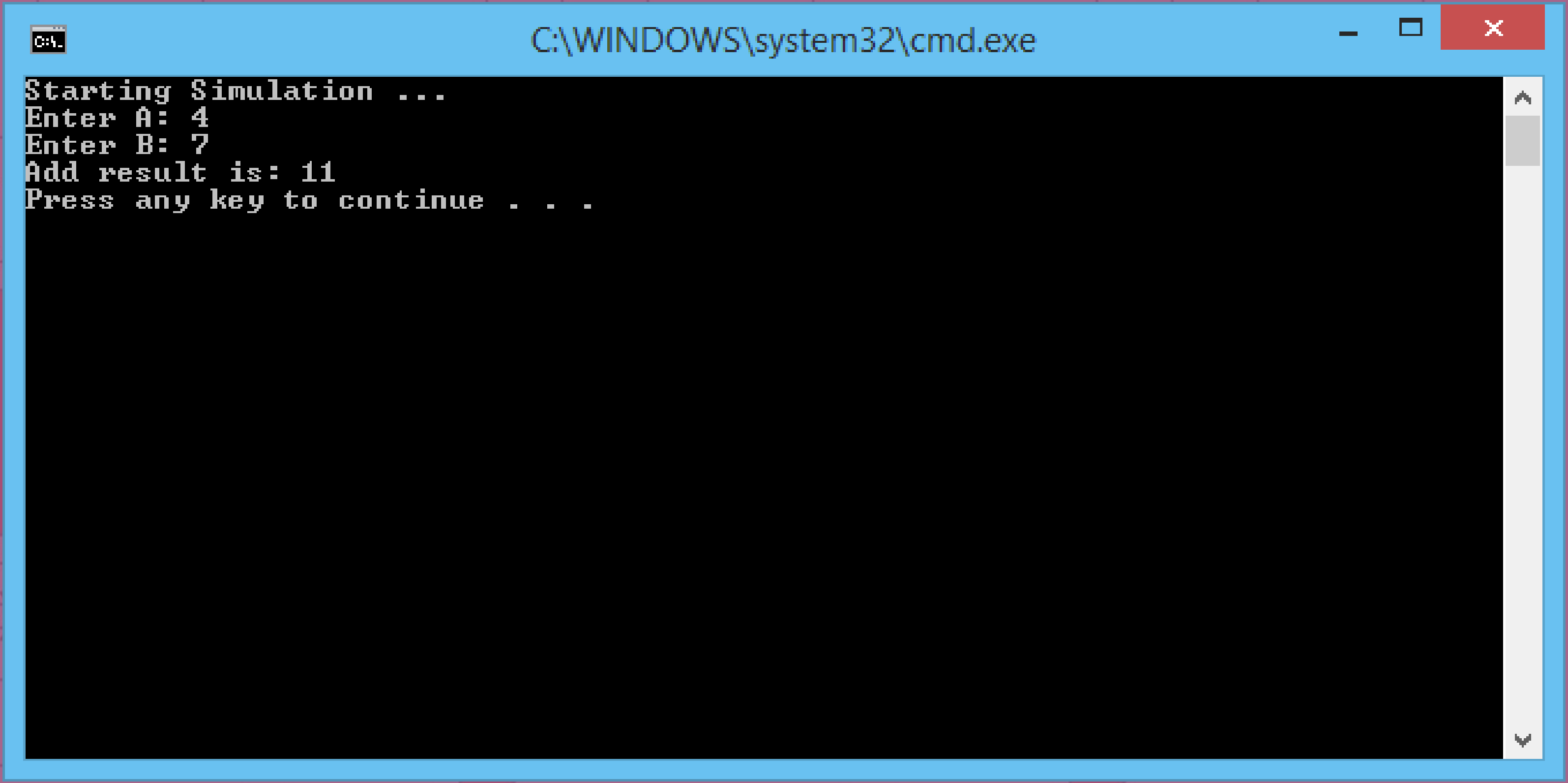
C++ Environment



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

```
#include "CPP Basics.h"

int main()
{
    int A, B, C;

    cout << "Starting Simulation ..." << "\n";
    cout << "Enter A: "; cin >> A;
    cout << "Enter B: "; cin >> B;
    C = A + B;
    cout << "Add result is: " << C << "\n";
    return 0;
}
```

C++ Environment



```
Starting Simulation ...
Enter A: 4
Enter B: 7
Add result is: 11
Press any key to continue . . .
```

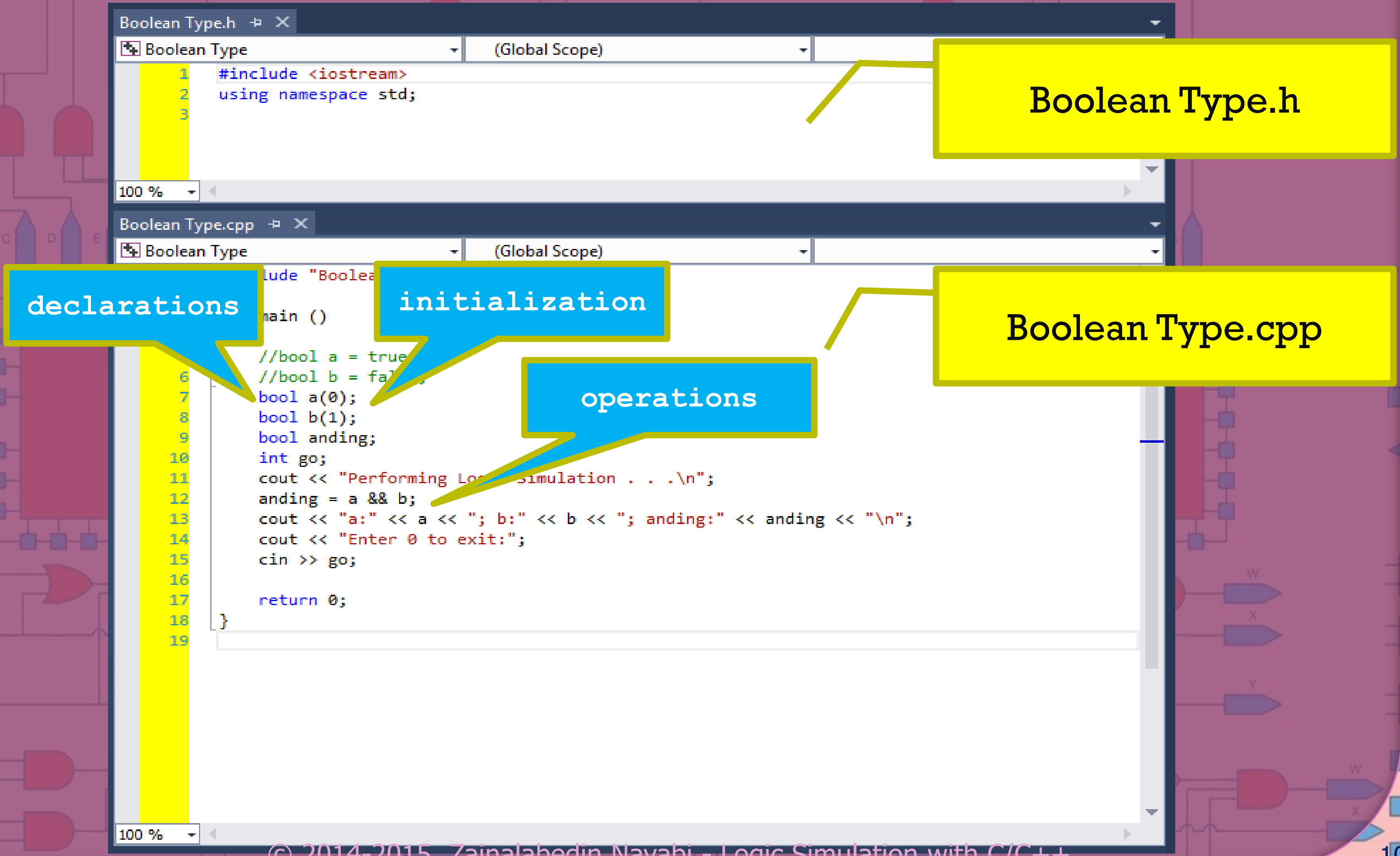
Logic Simulation with C/C++

- Procedural Languages for Hardware Modeling
- Types and Operators for Logic Modeling
- Basic Logic Simulation
  - Logic functions
  - Function overloading
  - Passing logic functions
  - Using default values
  - Building higher level structures
  - Handling 4-value logic
  - Logic vector
  - Sequential circuit modeling
  - Using pointers for logic vectors
- Enhanced logic simulation with timing
  - Using struct for timing and logic
  - Gates that handle timing
  - Utility functions
  - Timing in logic structures
  - Overloading logical operators
  - Using Boolean expressions
- More Functions for Wires and Gates
  - Gate classes
  - Carrier centric modeling
  - Compatible scalar and vector

Types and Operators for logic Modeling

Group	Type names	Note on size/Precision
Character Types	Char	Exactly one byte in size. At least 8 bits
Integer Types (signed)	Signed Char	Same size as char. At least 8 bits
	Signed Int	At least 16 bits
Integer Types (unsigned)	Unsigned Char	Same size as char. At least 8 bits
	Unsigned Int	At least 16 bits
Floating-point Type	Float	
	Double	Precision not less than float
	Long Double	Precision not less than float
Boolean Type	Bool	
Void Type	Void	No storage

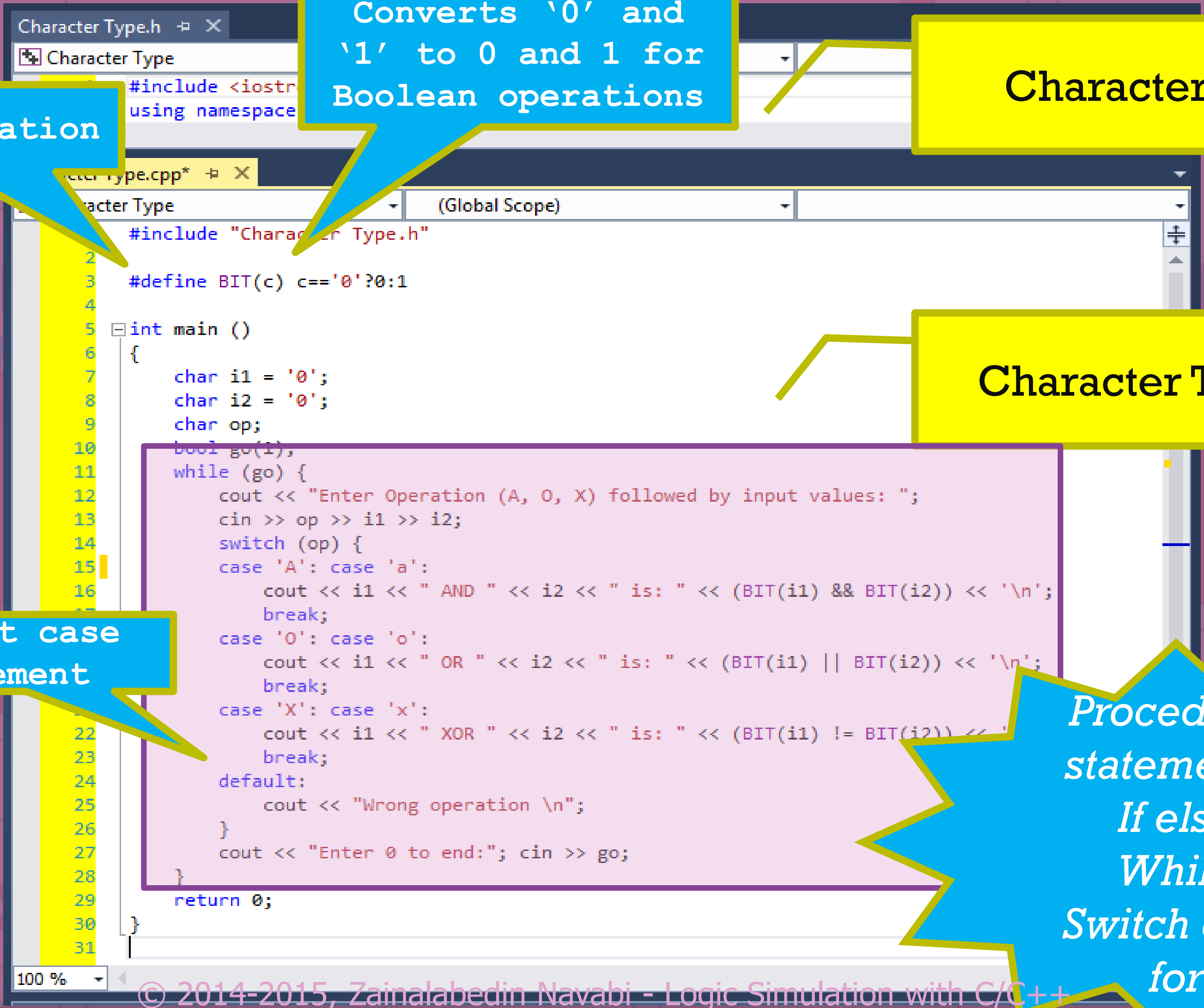
Using Boolean Type



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

```
lude "Boolea

ain ()

    //bool a = true
    //bool b = fa
    bool a(0);
    bool b(1);
    bool anding;
    int go;
    cout << "Performing Lo      imulation . . .\n";
    anding = a && b;
    cout << "a:" << a << "; b:" << b << "; anding:" << anding << "\n";
    cout << "Enter 0 to exit:";
    cin >> go;

    return 0;
}
```

Types and Operators for Logic Modeling



```
#include <iostr
using namespace
```

```
#include "Character Type.h"

#define BIT(c) c=='0'?0:1

int main ()
{
    char i1 = '0';
    char i2 = '0';
    char op;
    bool go(1);
    while (go) {
        cout << "Enter Operation (A, O, X) followed by input values: ";
        cin >> op >> i1 >> i2;
        switch (op) {
        case 'A': case 'a':
            cout << i1 << " AND " << i2 << " is: " << (BIT(i1) && BIT(i2)) << '\n';
            break;
        case 'O': case 'o':
            cout << i1 << " OR " << i2 << " is: " << (BIT(i1) || BIT(i2)) << '\n';
            break;
        case 'X': case 'x':
            cout << i1 << " XOR " << i2 << " is: " << (BIT(i1) != BIT(i2)) <<
            break;
        default:
            cout << "Wrong operation \n";
        }
        cout << "Enter 0 to end:"; cin >> go;
    }
    return 0;
}
```

Macro declaration

Converts '0' and '1' to 0 and 1 for Boolean operations

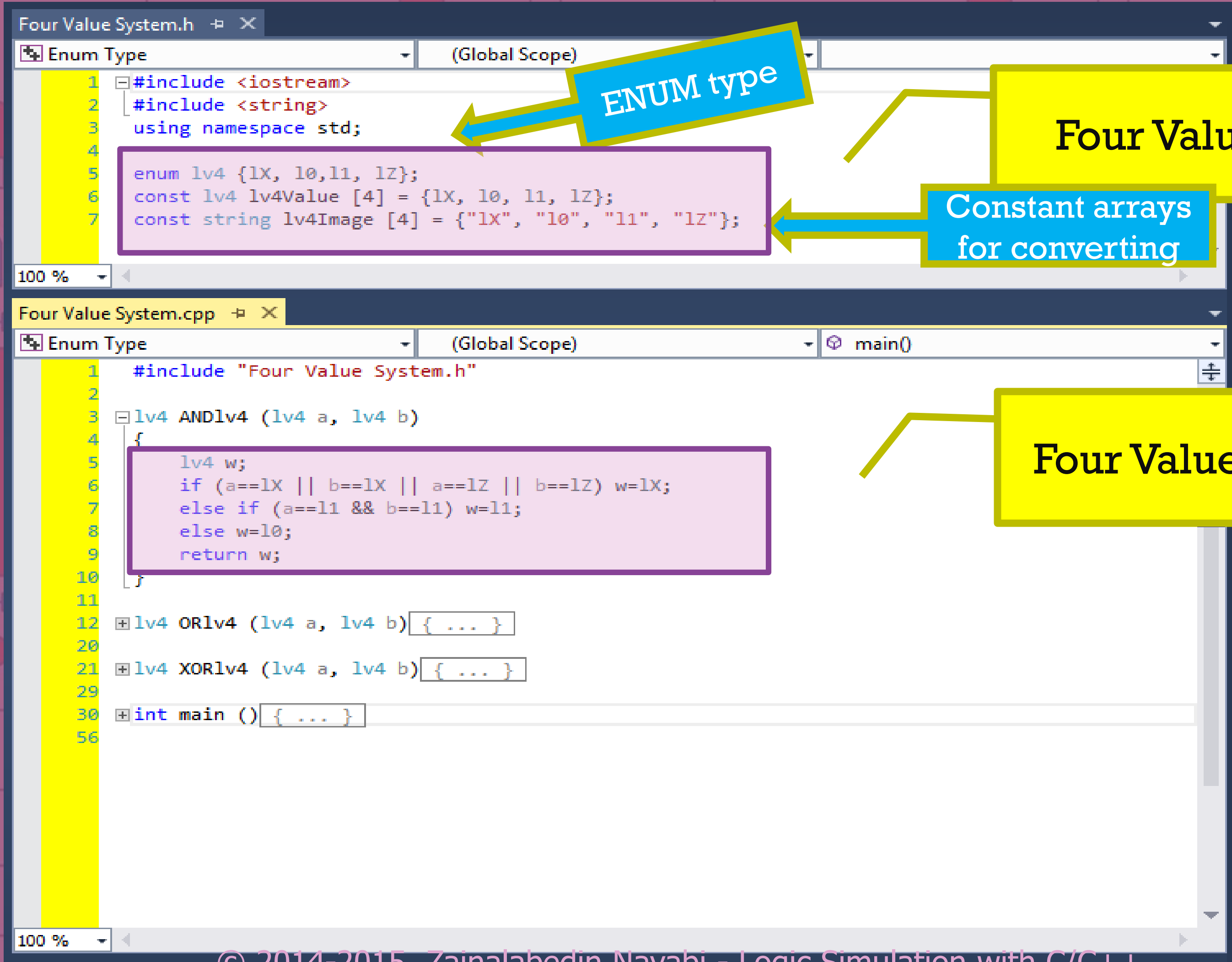
Character Type.h

Character Type.cpp

Default case statement

Procedural statements:
If else
While
Switch case
for

Using Enumerators



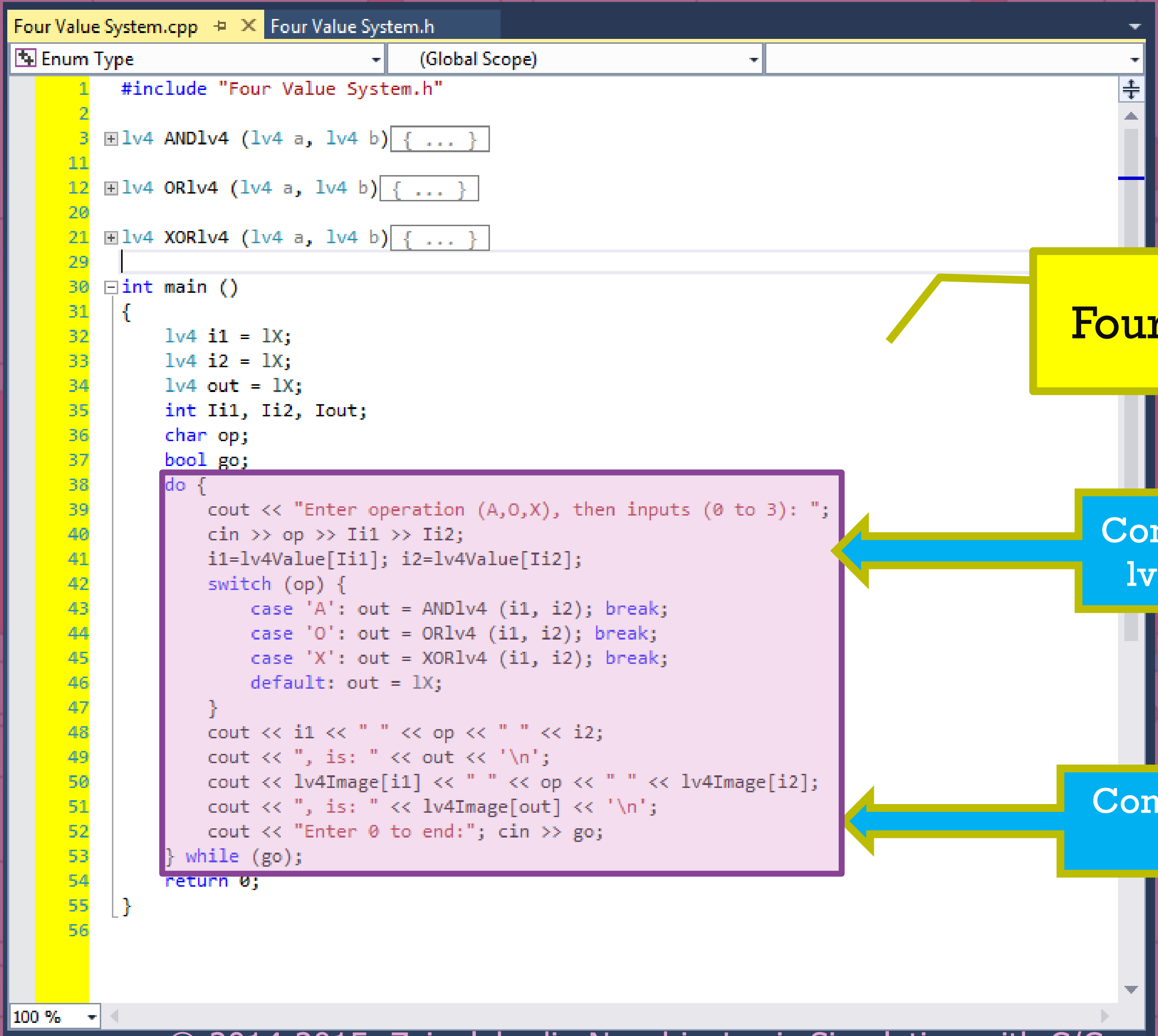
Four Value System.h

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

enum lv4 {lX, l0,l1, lZ};
const lv4 lv4Value [4] = {lX, l0, l1, lZ};
const string lv4Image [4] = {"lX", "l0", "l1", "lZ"};
```

Four Value System.cpp

```
1   #include "Four Value System.h"
2
3   lv4 ANDlv4 (lv4 a, lv4 b)
4   {
5       lv4 w;
6       if (a==lX || b==lX || a==lZ || b==lZ) w=lX;
7       else if (a==l1 && b==l1) w=l1;
8       else w=l0;
9       return w;
10  }
11
12  lv4 ORlv4 (lv4 a, lv4 b){ ... }
20
21  lv4 XORlv4 (lv4 a, lv4 b){ ... }
29
30  int main (){ ... }
56
```

Using Enumerators



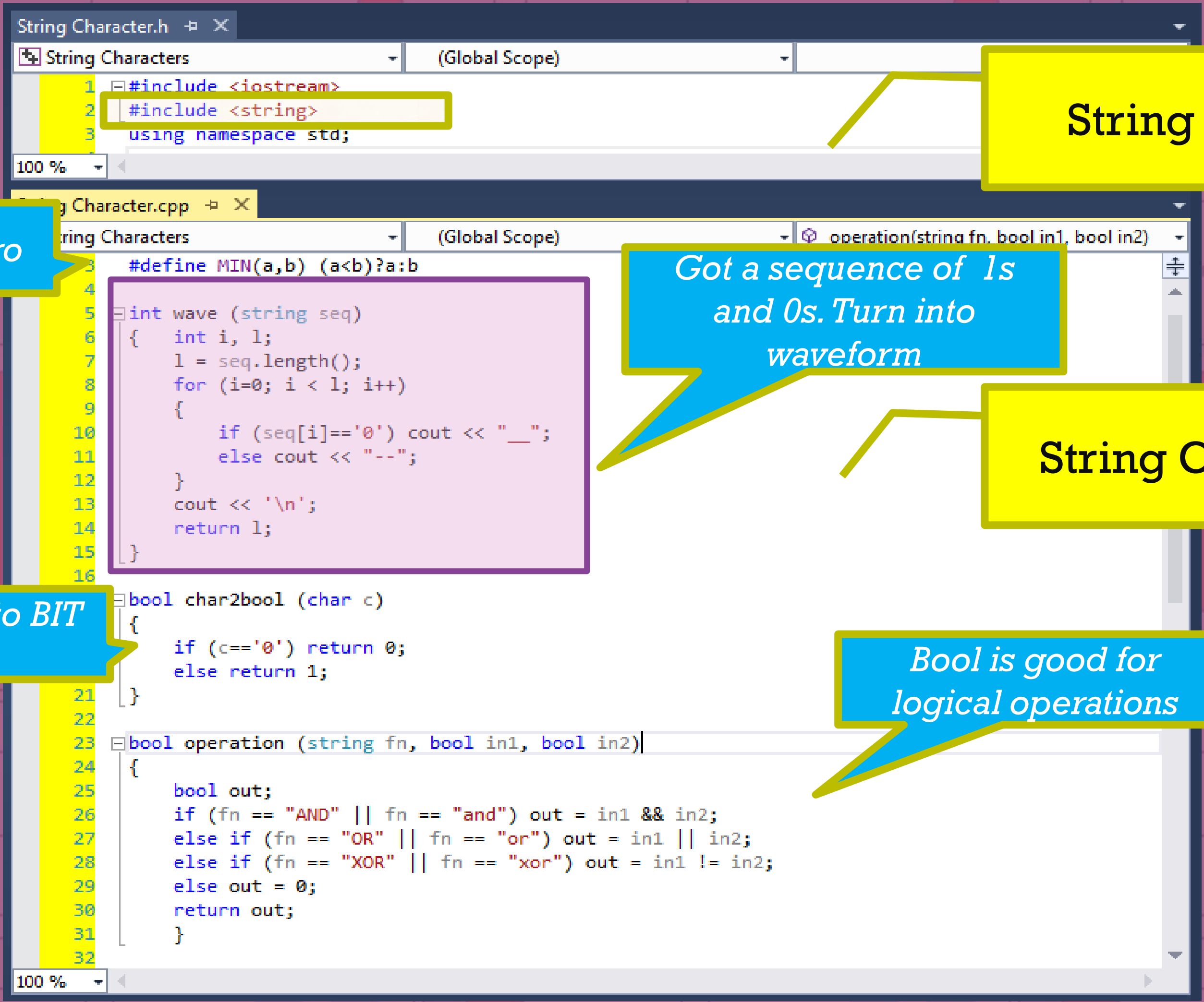
```cpp
#include "Four Value System.h"

lv4 ANDlv4 (lv4 a, lv4 b){ ... }

lv4 ORlv4 (lv4 a, lv4 b){ ... }

lv4 XORlv4 (lv4 a, lv4 b){ ... }

int main ()
{
    lv4 i1 = lX;
    lv4 i2 = lX;
    lv4 out = lX;
    int Ii1, Ii2, Iout;
    char op;
    bool go;
    do {
        cout << "Enter operation (A,O,X), then inputs (0 to 3): ";
        cin >> op >> Ii1 >> Ii2;
        i1=lv4Value[Ii1]; i2=lv4Value[Ii2];
        switch (op) {
            case 'A': out = ANDlv4 (i1, i2); break;
            case 'O': out = ORlv4 (i1, i2); break;
            case 'X': out = XORlv4 (i1, i2); break;
            default: out = lX;
        }
        cout << i1 << " " << op << " " << i2;
        cout << ", is: " << out << '\n';
        cout << lv4Image[i1] << " " << op << " " << lv4Image[i2];
        cout << ", is: " << lv4Image[out] << '\n';
        cout << "Enter 0 to end:"; cin >> go;
    } while (go);
    return 0;
}
```

Four Value System.cpp

Convert Integer value to lv4 enumeration type

Conversion to string for printing

Waveform Generation



String Character.h

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

String Character.cpp

```
#define MIN(a,b) (a<b)?a:b

int wave (string seq)
{   int i, l;
    l = seq.length();
    for (i=0; i < l; i++)
    {
        if (seq[i]=='0') cout << "__";
        else cout << "--";
    }
    cout << '\n';
    return l;
}

bool char2bool (char c)
{
    if (c=='0') return 0;
    else return 1;
}

bool operation (string fn, bool in1, bool in2)
{
    bool out;
    if (fn == "AND" || fn == "and") out = in1 && in2;
    else if (fn == "OR" || fn == "or") out = in1 || in2;
    else if (fn == "XOR" || fn == "xor") out = in1 != in2;
    else out = 0;
    return out;
    }
```

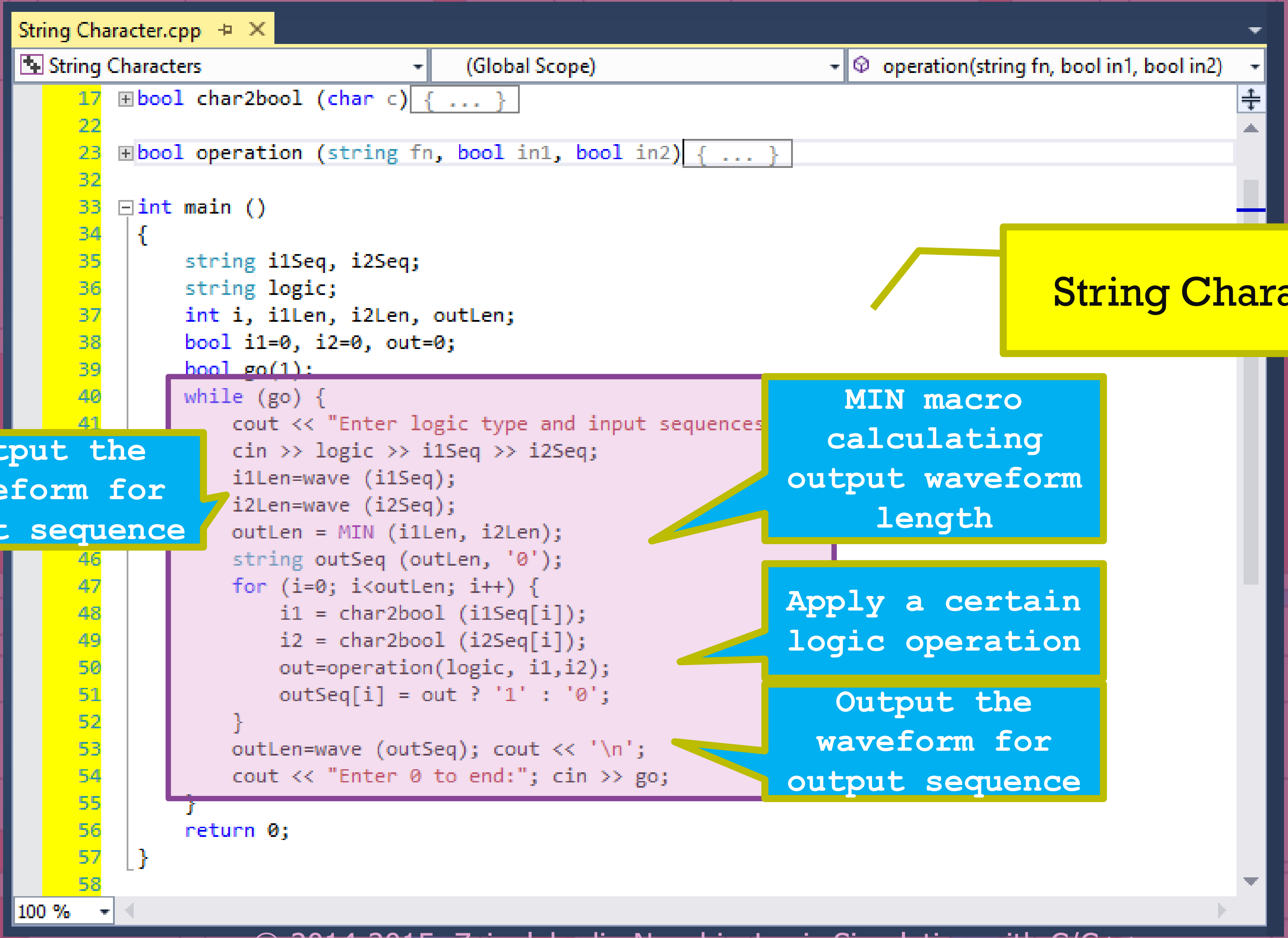
MIN Macro

Got a sequence of 1s and 0s. Turn into waveform

Equivalent to BIT macro

Bool is good for logical operations

Types and Operators for Logic Modeling



```
bool char2bool (char c){ ... }

bool operation (string fn, bool in1, bool in2){ ... }

int main ()
{
    string i1Seq, i2Seq;
    string logic;
    int i, i1Len, i2Len, outLen;
    bool i1=0, i2=0, out=0;
    bool go(1);
    while (go) {
        cout << "Enter logic type and input sequences
        cin >> logic >> i1Seq >> i2Seq;
        i1Len=wave (i1Seq);
        i2Len=wave (i2Seq);
        outLen = MIN (i1Len, i2Len);
        string outSeq (outLen, '0');
        for (i=0; i<outLen; i++) {
            i1 = char2bool (i1Seq[i]);
            i2 = char2bool (i2Seq[i]);
            out=operation(logic, i1,i2);
            outSeq[i] = out ? '1' : '0';
        }
        outLen=wave (outSeq); cout << '\n';
        cout << "Enter 0 to end:"; cin >> go;
    }
    return 0;
}
```

String Character.cpp

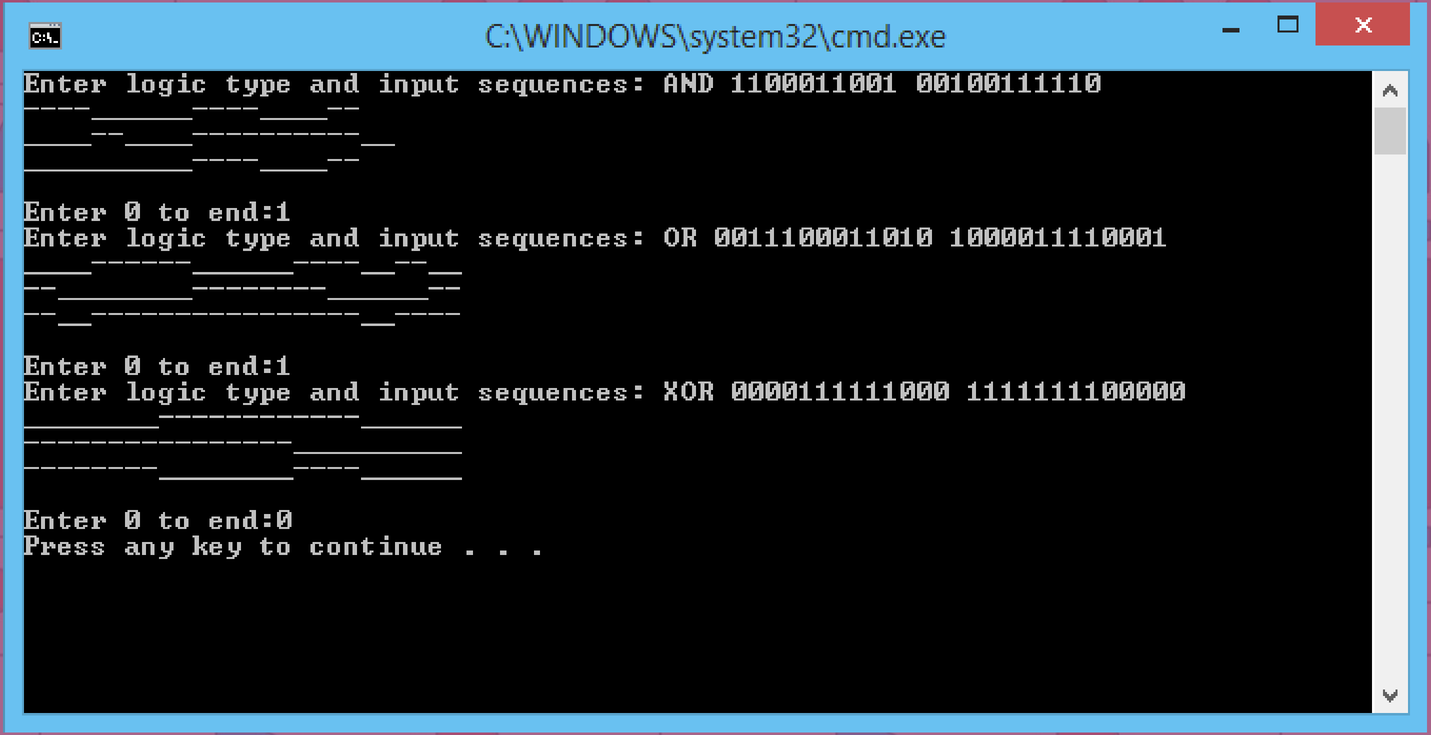
Output the waveform for input sequence

MIN macro calculating output waveform length

Apply a certain logic operation

Output the waveform for output sequence

Types and Operators for Logic Modeling

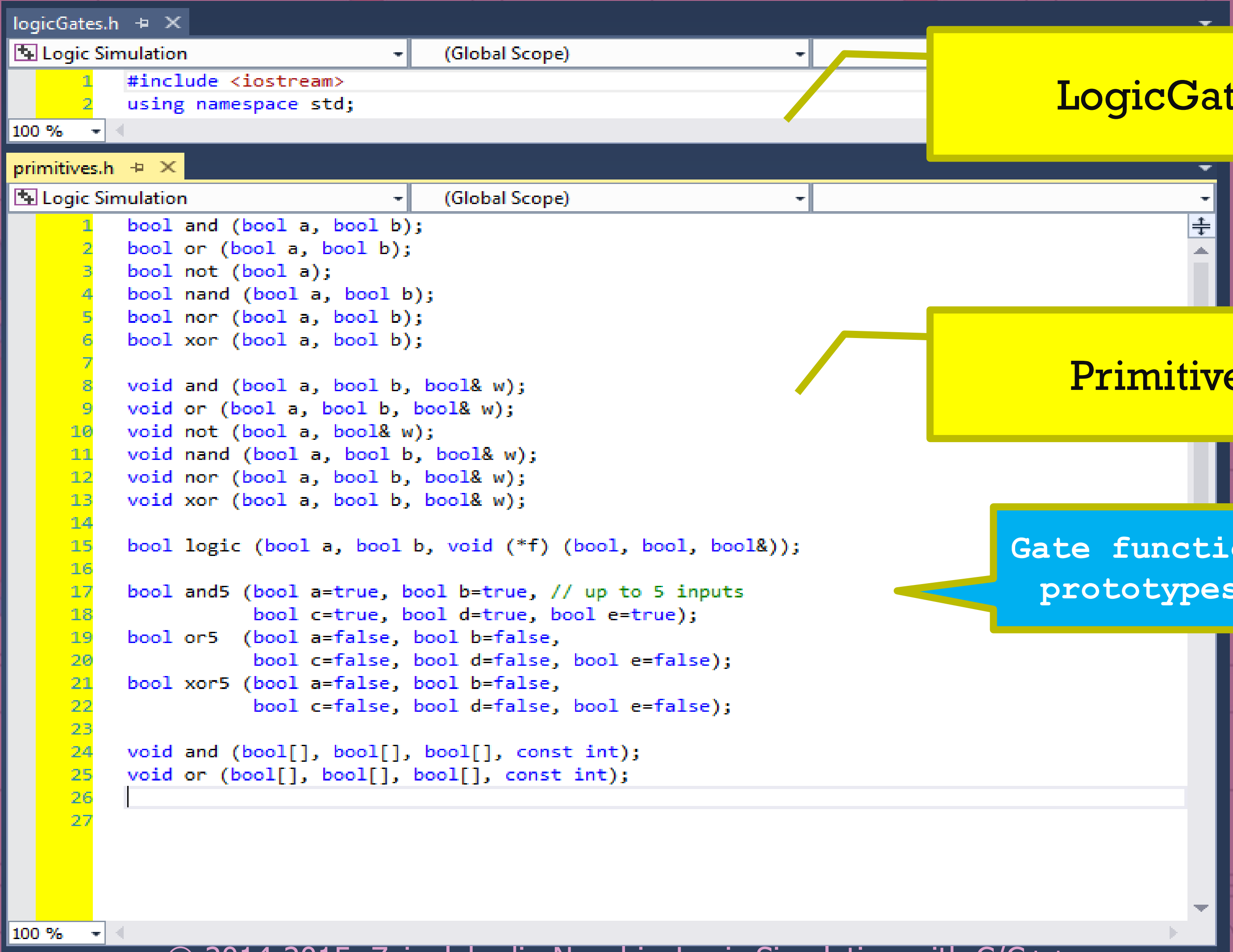


```
C:\WINDOWS\system32\cmd.exe
Enter logic type and input sequences: AND 1100011001 00100111110
----      ----    --
    ______    ____
    --    ----------
____  ____          __
          ----    --
__________    ____

Enter 0 to end:1
Enter logic type and input sequences: OR 0011100011010 1000011110001
    ------      ----  --
____      ______    __  __
--        --------      --
  ________        ______
--  ----------------  ----
  __                __

Enter 0 to end:1
Enter logic type and input sequences: XOR 0000111111000 1111111100000
        ------------
________            ______
----------------
                __________
--------        ----
        ________    ______

Enter 0 to end:0
Press any key to continue . . .
```

Logic Simulation with C/C++

- Procedural Languages for Hardware Modeling
- Types and Operators for Logic Modeling
- Basic Logic Simulation
  - Logic functions
  - Function overloading
  - Passing logic functions
  - Using default values
  - Building higher level structures
  - Handling 4-value logic
  - Logic vector
  - Sequential circuit modeling
  - Using pointers for logic vectors
- Enhanced logic simulation with timing
  - Using struct for timing and logic
  - Gates that handle timing
  - Utility functions
  - Timing in logic structures
  - Overloading logical operators
  - Using Boolean expressions
- More Functions for Wires and Gates
  - Gate classes
  - Carrier centric modeling
  - Compatible scalar and vector

Basic Logic Simulation



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

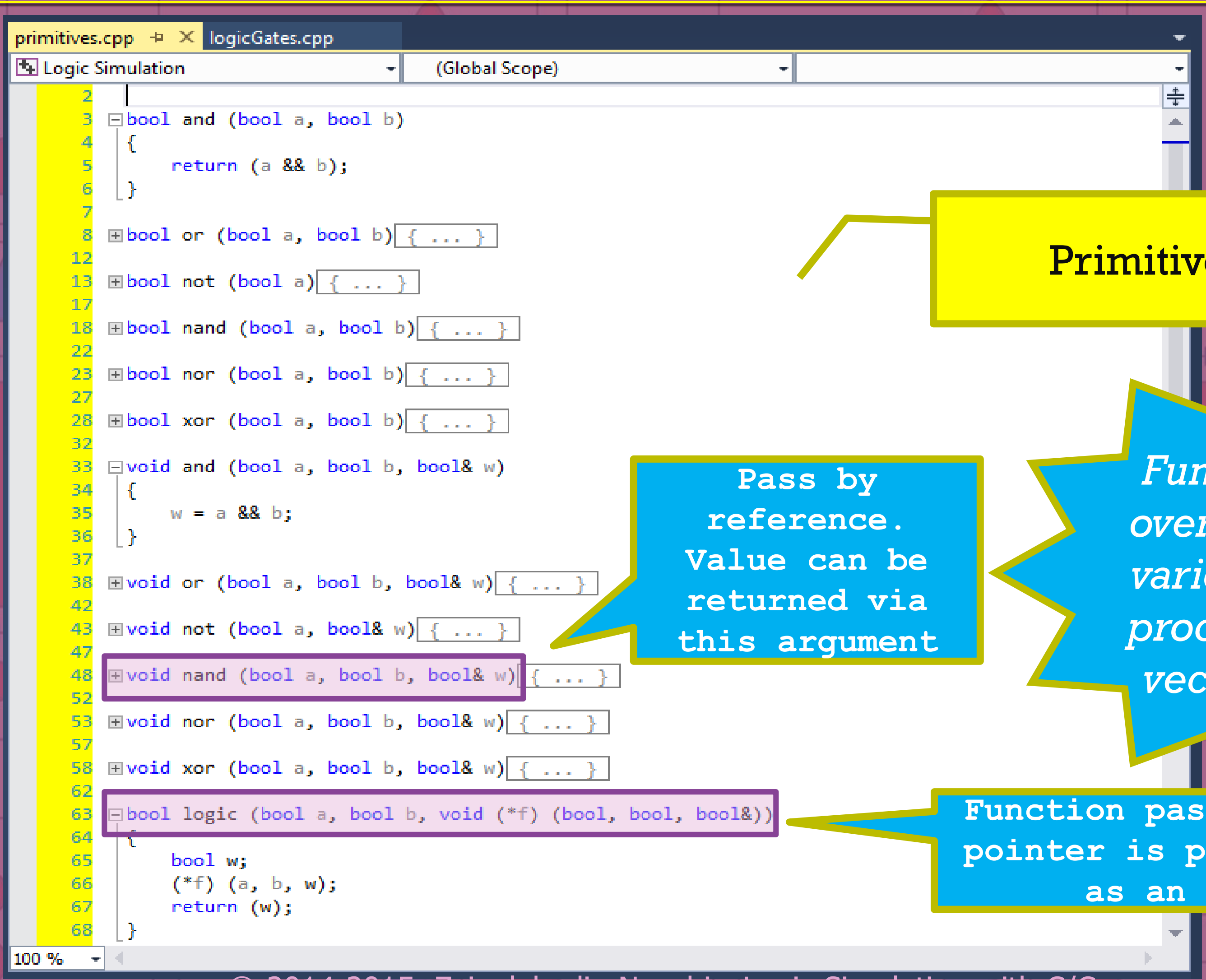
```
bool and (bool a, bool b);
bool or (bool a, bool b);
bool not (bool a);
bool nand (bool a, bool b);
bool nor (bool a, bool b);
bool xor (bool a, bool b);

void and (bool a, bool b, bool& w);
void or (bool a, bool b, bool& w);
void not (bool a, bool& w);
void nand (bool a, bool b, bool& w);
void nor (bool a, bool b, bool& w);
void xor (bool a, bool b, bool& w);

bool logic (bool a, bool b, void (*f) (bool, bool, bool&));

bool and5 (bool a=true, bool b=true, // up to 5 inputs
           bool c=true, bool d=true, bool e=true);
bool or5  (bool a=false, bool b=false,
           bool c=false, bool d=false, bool e=false);
bool xor5 (bool a=false, bool b=false,
           bool c=false, bool d=false, bool e=false);

void and (bool[], bool[], bool[], const int);
void or (bool[], bool[], bool[], const int);
```

LogicGates.h

Primitives.h

Gate function prototypes

Logic Functions



primitives.cpp | logicGates.cpp

Logic Simulation | (Global Scope)

```
bool and (bool a, bool b)
{
    return (a && b);
}

bool or (bool a, bool b){ ... }

bool not (bool a){ ... }

bool nand (bool a, bool b){ ... }

bool nor (bool a, bool b){ ... }

bool xor (bool a, bool b){ ... }

void and (bool a, bool b, bool& w)
{
    w = a && b;
}

void or (bool a, bool b, bool& w){ ... }

void not (bool a, bool& w){ ... }

void nand (bool a, bool b, bool& w){ ... }

void nor (bool a, bool b, bool& w){ ... }

void xor (bool a, bool b, bool& w){ ... }

bool logic (bool a, bool b, void (*f) (bool, bool, bool&))
{
    bool w;
    (*f) (a, b, w);
    return (w);
}
```

100 %

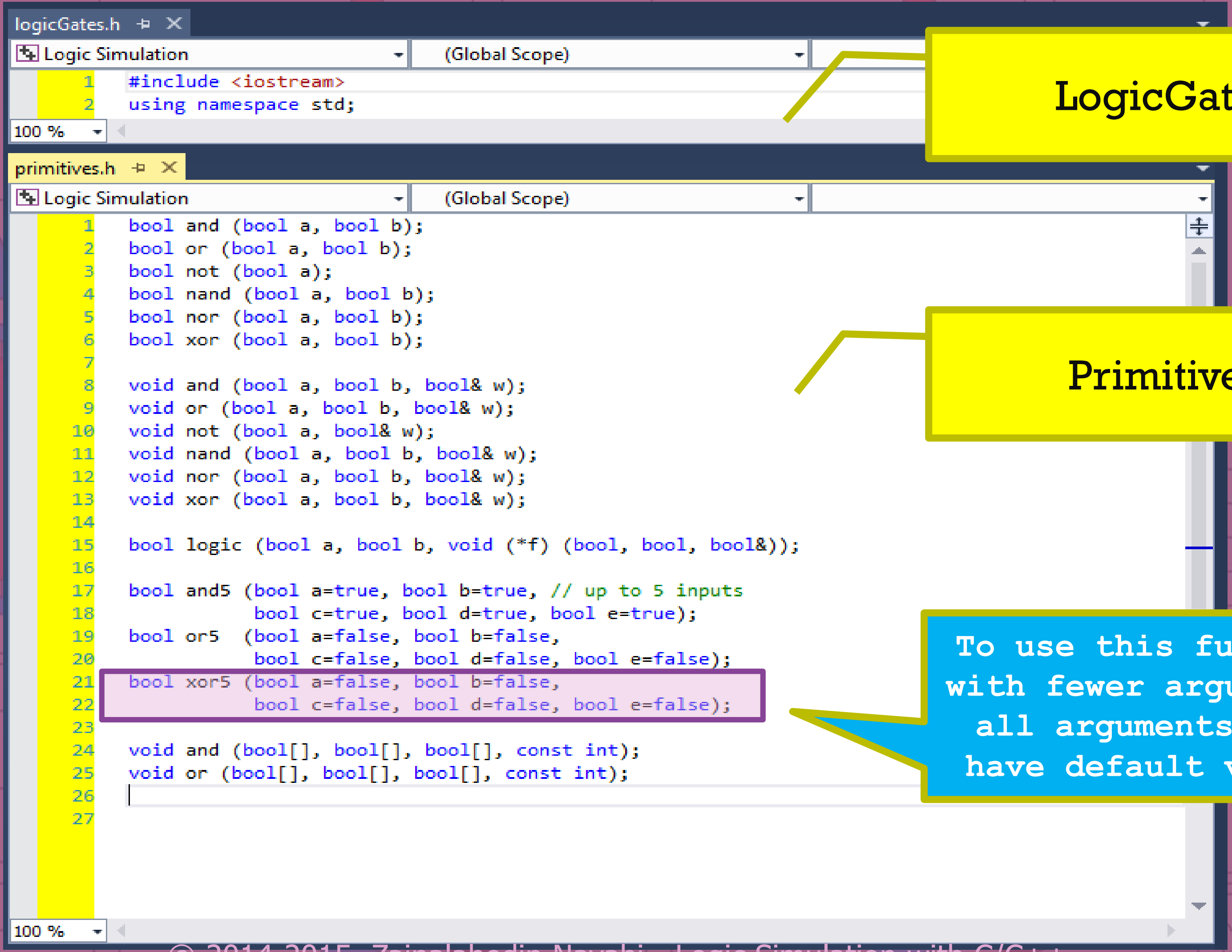
Primitives.cpp

Pass by reference. Value can be returned via this argument

Functions are overloaded for various type of procedure and vector format

Function passing. Function pointer is passed to logic as an argument

Using Default Values



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

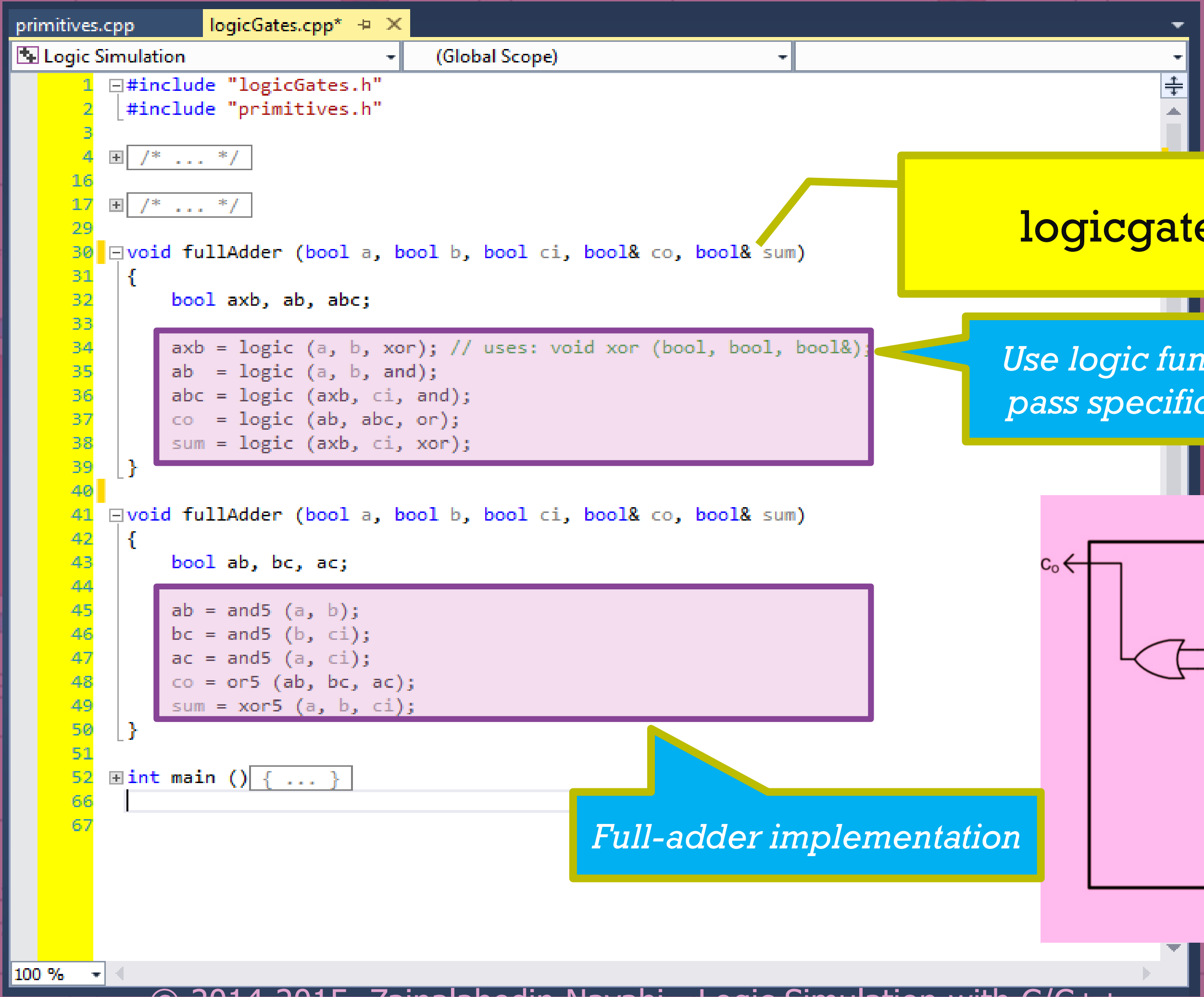
```
bool and (bool a, bool b);
bool or (bool a, bool b);
bool not (bool a);
bool nand (bool a, bool b);
bool nor (bool a, bool b);
bool xor (bool a, bool b);

void and (bool a, bool b, bool& w);
void or (bool a, bool b, bool& w);
void not (bool a, bool& w);
void nand (bool a, bool b, bool& w);
void nor (bool a, bool b, bool& w);
void xor (bool a, bool b, bool& w);

bool logic (bool a, bool b, void (*f) (bool, bool, bool&));

bool and5 (bool a=true, bool b=true, // up to 5 inputs
           bool c=true, bool d=true, bool e=true);
bool or5  (bool a=false, bool b=false,
           bool c=false, bool d=false, bool e=false);
bool xor5 (bool a=false, bool b=false,
           bool c=false, bool d=false, bool e=false);

void and (bool[], bool[], bool[], const int);
void or (bool[], bool[], bool[], const int);
```

LogicGates.h

Primitives.h

To use this function with fewer arguments, all arguments must have default values

Building Higher Level Structures

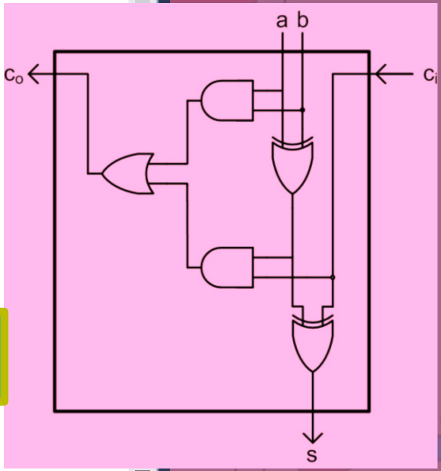


```
#include "logicGates.h"
#include "primitives.h"

/* ... */

/* ... */

void fullAdder (bool a, bool b, bool ci, bool& co, bool& sum)
{
    bool axb, ab, abc;

    axb = logic (a, b, xor); // uses: void xor (bool, bool, bool&);
    ab  = logic (a, b, and);
    abc = logic (axb, ci, and);
    co  = logic (ab, abc, or);
    sum = logic (axb, ci, xor);
}

void fullAdder (bool a, bool b, bool ci, bool& co, bool& sum)
{
    bool ab, bc, ac;

    ab = and5 (a, b);
    bc = and5 (b, ci);
    ac = and5 (a, ci);
    co = or5 (ab, bc, ac);
    sum = xor5 (a, b, ci);
}

int main () { ... }
```

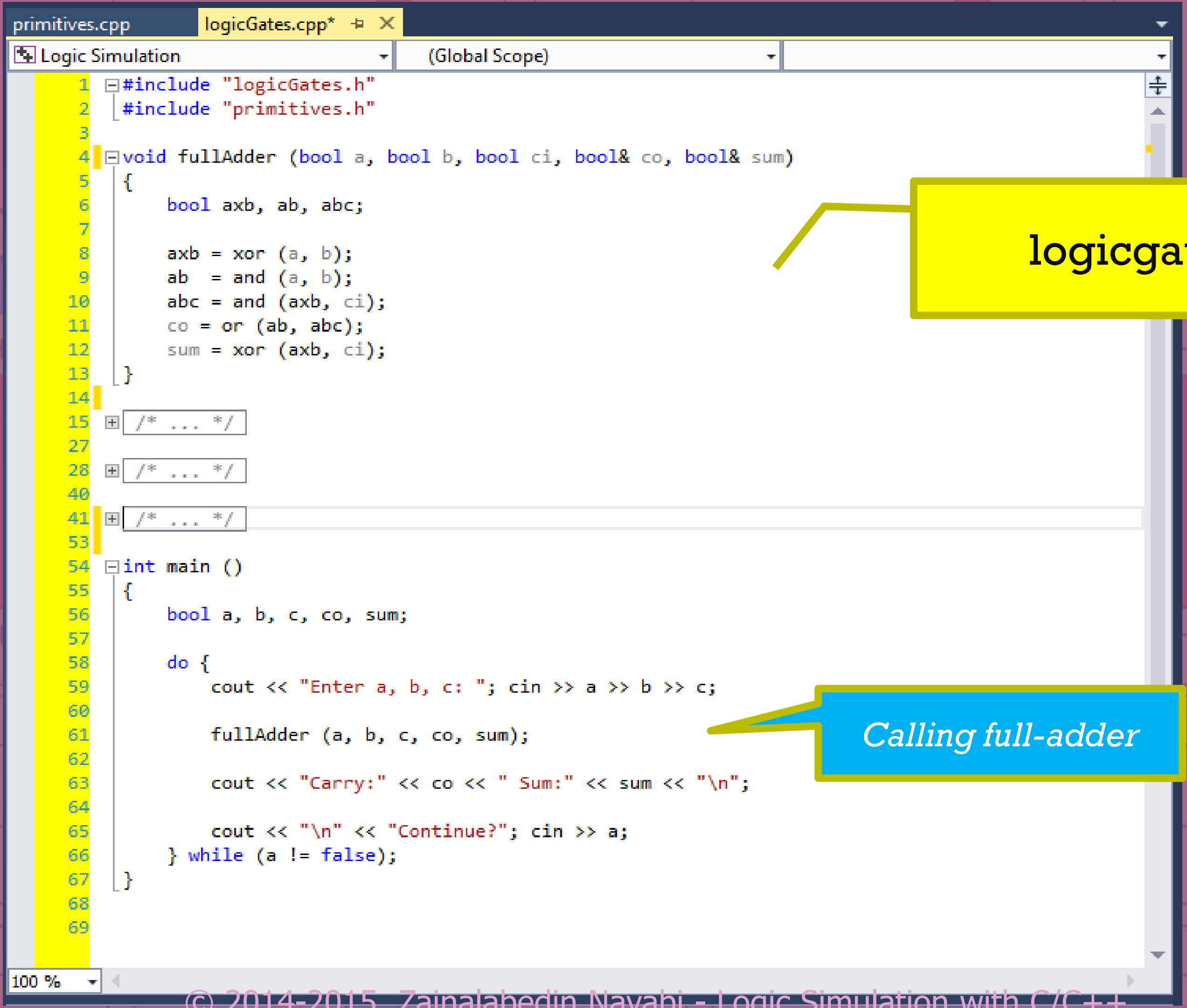
logicgates.cpp

Use logic function and pass specific function

Full-adder implementation



Building Higher Level Structures



primitives.cpp | logicGates.cpp*

Logic Simulation | (Global Scope)

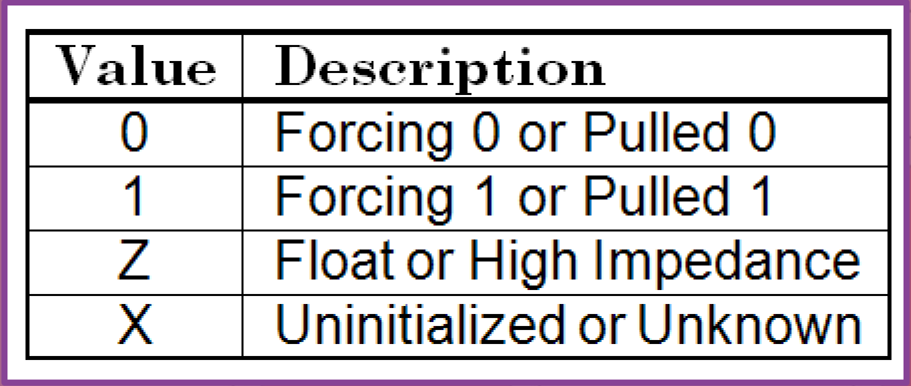
```
#include "logicGates.h"
#include "primitives.h"

void fullAdder (bool a, bool b, bool ci, bool& co, bool& sum)
{
    bool axb, ab, abc;

    axb = xor (a, b);
    ab  = and (a, b);
    abc = and (axb, ci);
    co = or (ab, abc);
    sum = xor (axb, ci);
}

/* ... */

/* ... */

/* ... */

int main ()
{
    bool a, b, c, co, sum;

    do {
        cout << "Enter a, b, c: "; cin >> a >> b >> c;

        fullAdder (a, b, c, co, sum);

        cout << "Carry:" << co << " Sum:" << sum << "\n";

        cout << "\n" << "Continue?"; cin >> a;
    } while (a != false);
}
```

100 %

logicgates.cpp

Calling full-adder

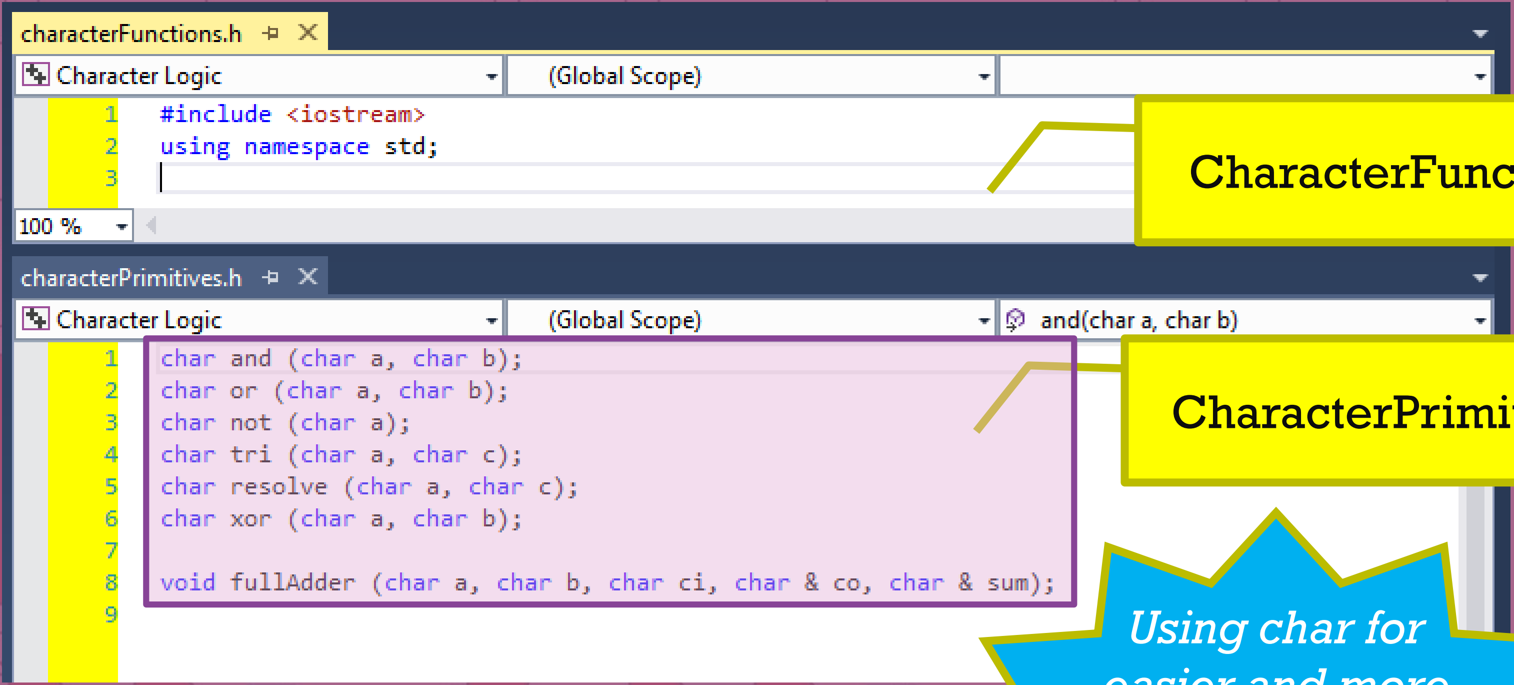
4-value Logic



Value	Description
0	Forcing 0 or Pulled 0
1	Forcing 1 or Pulled 1
Z	Float or High Impedance
X	Uninitialized or Unknown

- **Four-Value Logic System**

Handling 4-value Logic



characterFunctions.h

Character Logic | (Global Scope)

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

```

100 %

CharacterFunctions.h

characterPrimitives.h

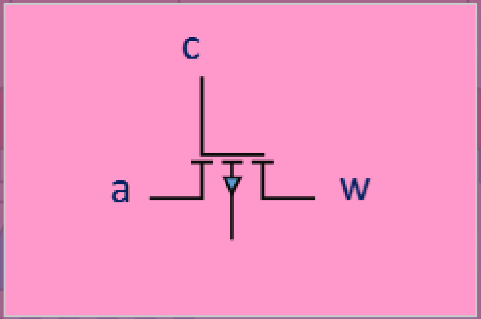
Character Logic | (Global Scope) | and(char a, char b)

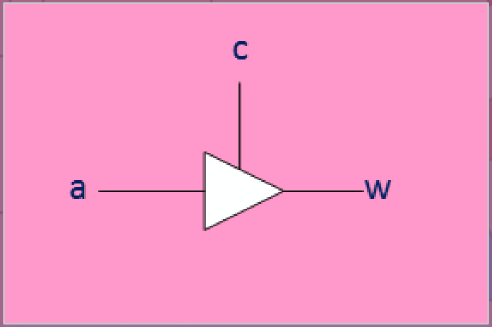
```
char and (char a, char b);
char or (char a, char b);
char not (char a);
char tri (char a, char c);
char resolve (char a, char c);
char xor (char a, char b);

void fullAdder (char a, char b, char ci, char & co, char & sum);
```

CharacterPrimitives.h

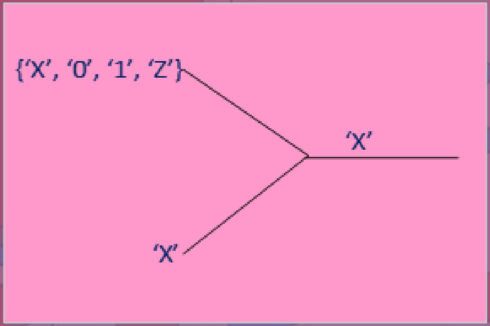
Using char for easier and more expressive in and out instead of directly input logic value

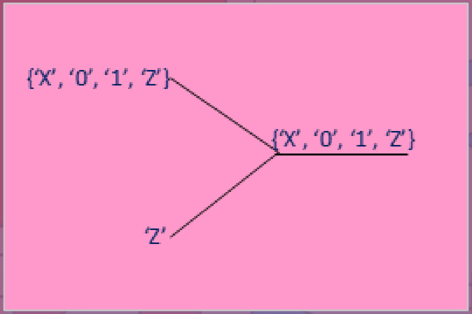
Handling 4-value Logic





Tri-state

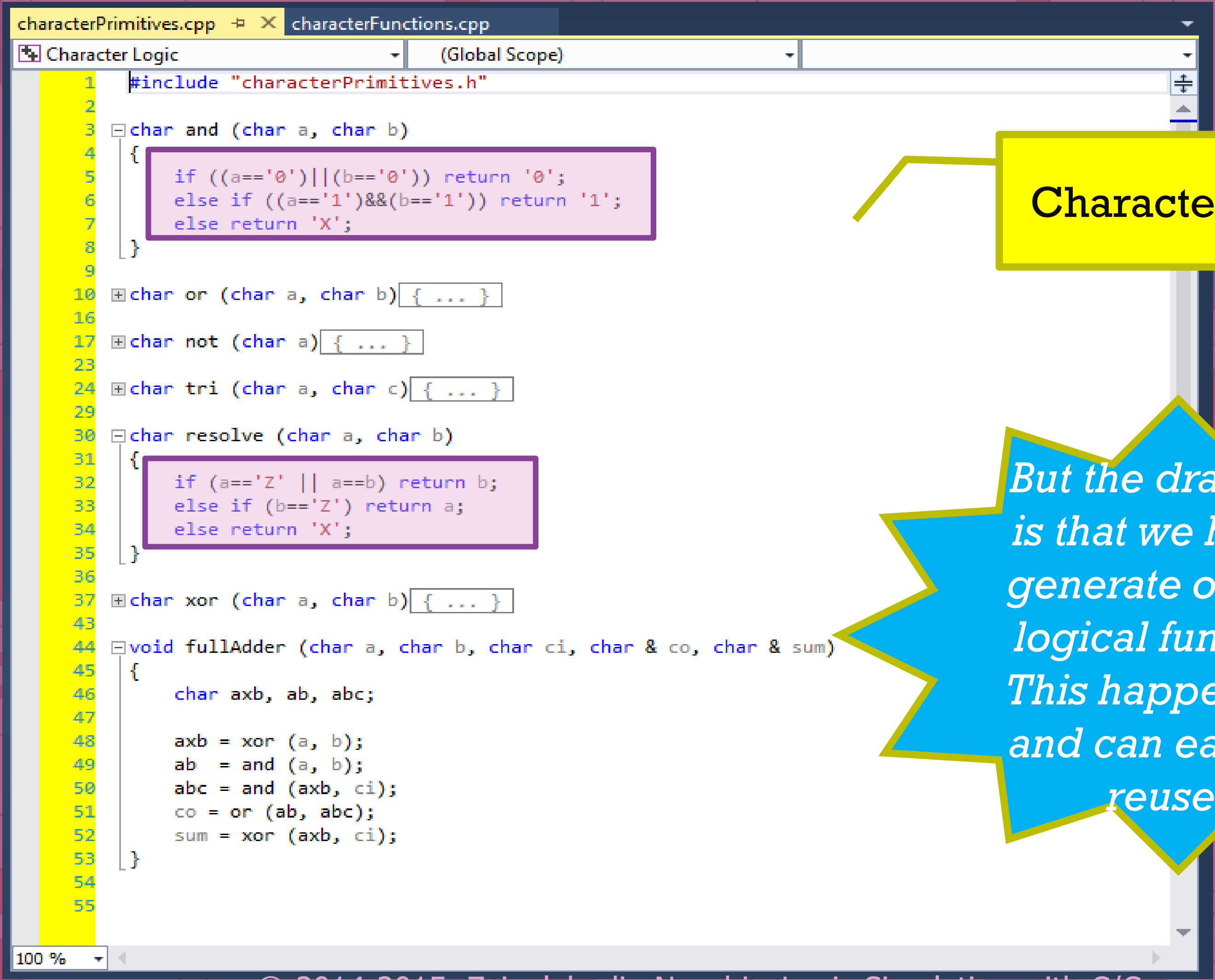




And resolution function

	x	0	1	z
x	x	x	x	x
0	x	0	x	0
1	x	x	1	1
z	x	0	1	z

Handling 4-value Logic



characterPrimitives.cpp | characterFunctions.cpp

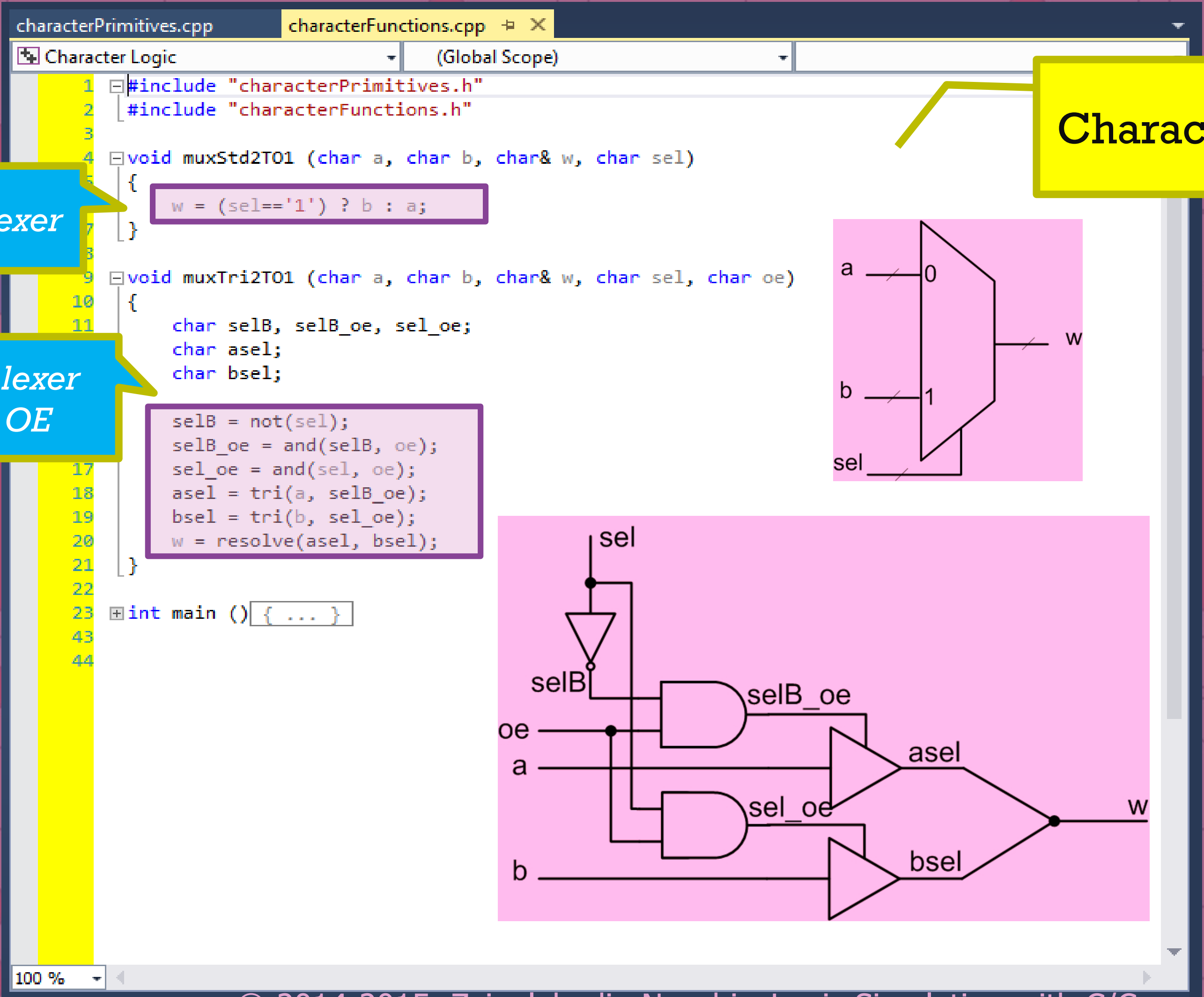
Character Logic | (Global Scope)

```
#include "characterPrimitives.h"

char and (char a, char b)
{
    if ((a=='0')||(b=='0')) return '0';
    else if ((a=='1')&&(b=='1')) return '1';
    else return 'X';
}

char or (char a, char b){ ... }

char not (char a){ ... }

char tri (char a, char c){ ... }

char resolve (char a, char b)
{
    if (a=='Z' || a==b) return b;
    else if (b=='Z') return a;
    else return 'X';
}

char xor (char a, char b){ ... }

void fullAdder (char a, char b, char ci, char & co, char & sum)
{
    char axb, ab, abc;

    axb = xor (a, b);
    ab  = and (a, b);
    abc = and (axb, ci);
    co = or (ab, abc);
    sum = xor (axb, ci);
}
```

CharacterPrimitives.cpp

But the drawback is that we have to generate our own logical functions. This happens one and can easily be reused.

Handling 4-value Logic



characterPrimitives.cpp | characterFunctions.cpp

Character Logic | (Global Scope)

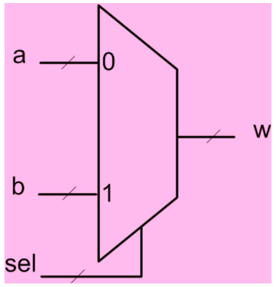
```
#include "characterPrimitives.h"
#include "characterFunctions.h"

void muxStd2TO1 (char a, char b, char& w, char sel)
{
    w = (sel=='1') ? b : a;
}

void muxTri2TO1 (char a, char b, char& w, char sel, char oe)
{
    char selB, selB_oe, sel_oe;
    char asel;
    char bsel;

    selB = not(sel);
    selB_oe = and(selB, oe);
    sel_oe = and(sel, oe);
    asel = tri(a, selB_oe);
    bsel = tri(b, sel_oe);
    w = resolve(asel, bsel);
}

int main () { ... }
```

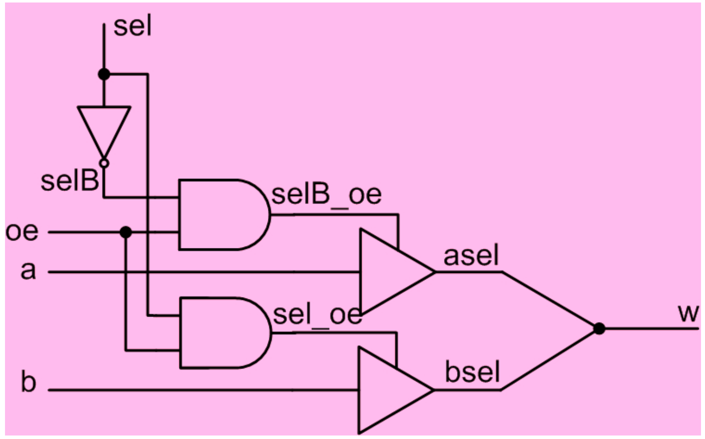
100 %

multiplexer

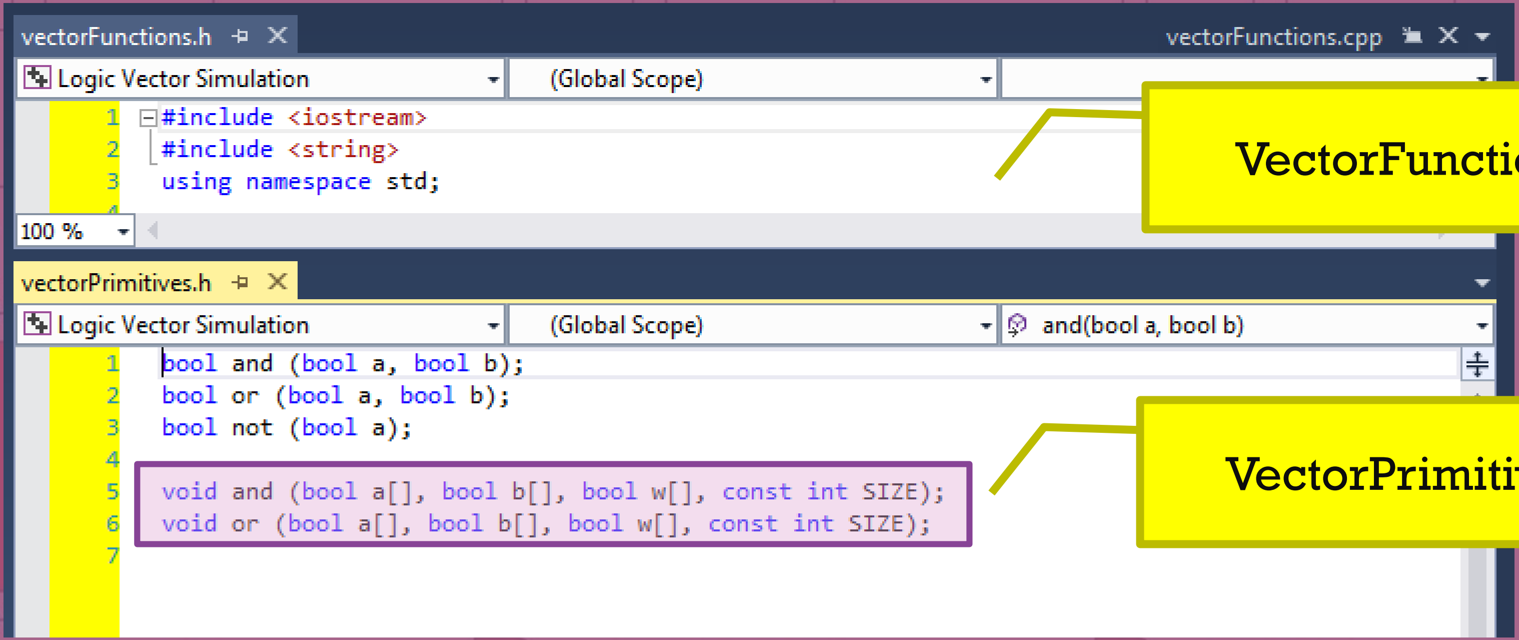
Multiplexer with OE

CharacterFunctions.cpp





Logic Vector



vectorFunctions.h

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

vectorPrimitives.h

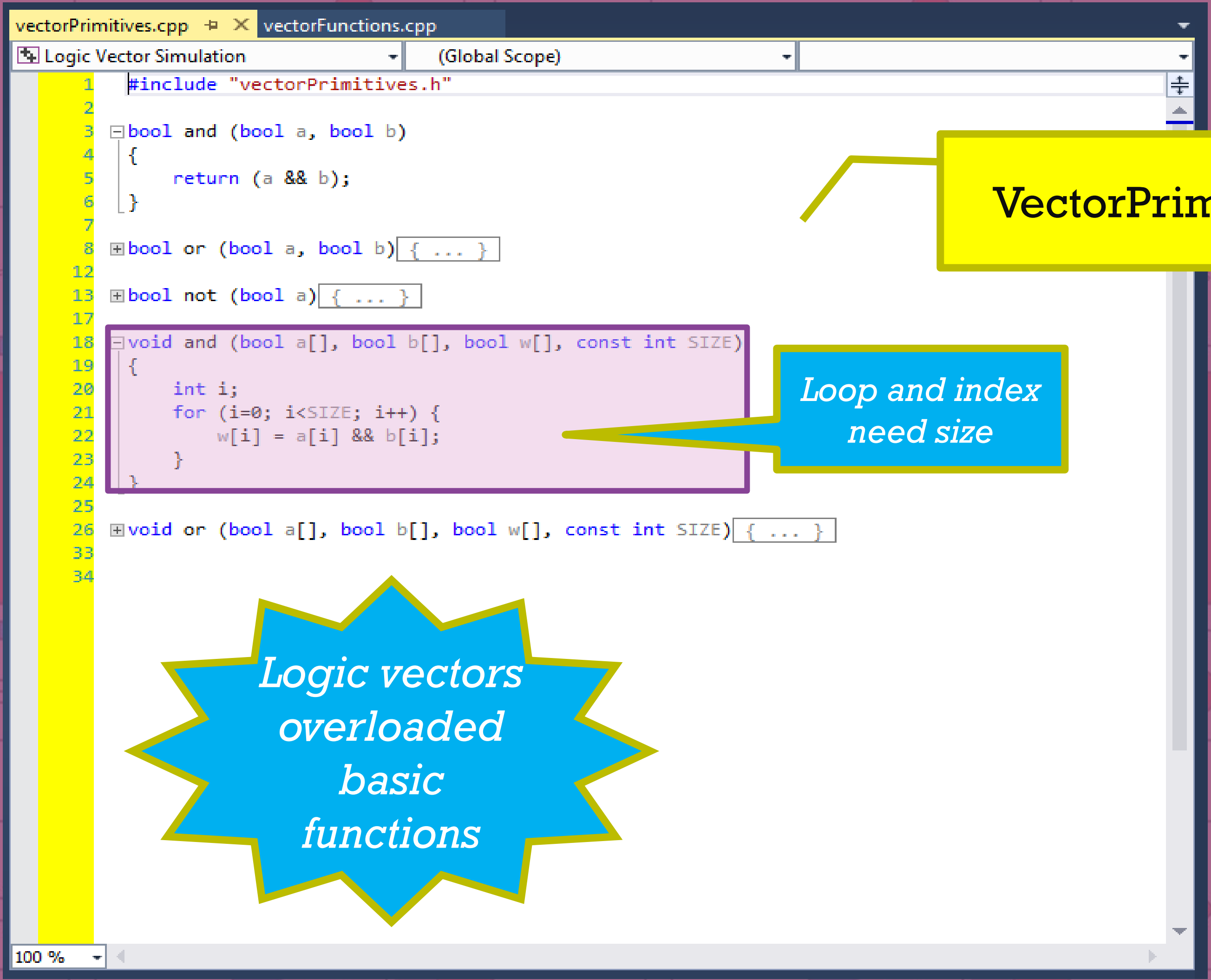
```
bool and (bool a, bool b);
bool or (bool a, bool b);
bool not (bool a);

void and (bool a[], bool b[], bool w[], const int SIZE);
void or (bool a[], bool b[], bool w[], const int SIZE);
```

VectorFunctions.h

VectorPrimitives.h

Arrays are passed by reference to first location

Logic Vector



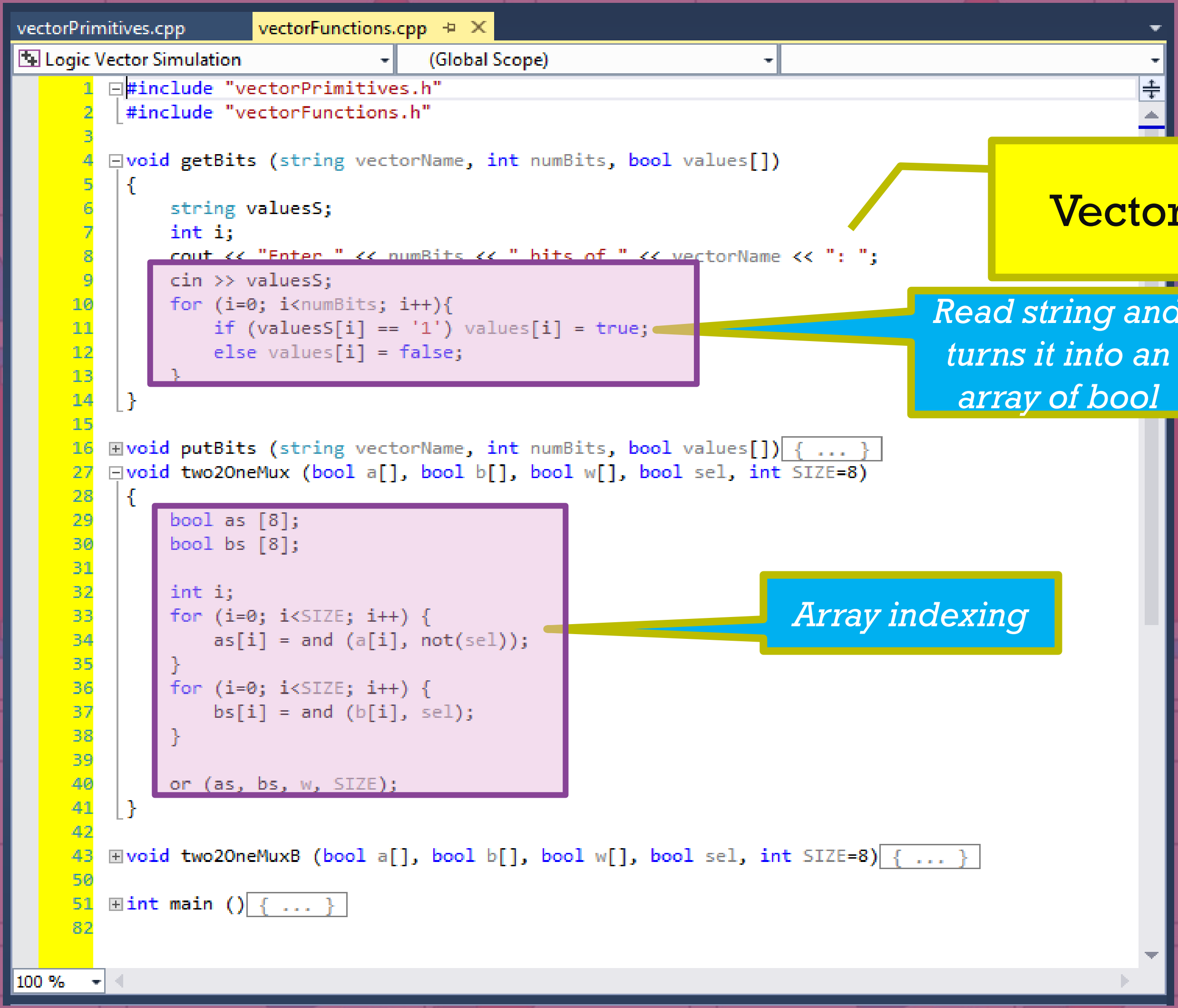
```
#include "vectorPrimitives.h"

bool and (bool a, bool b)
{
    return (a && b);
}

bool or (bool a, bool b){ ... }

bool not (bool a){ ... }

void and (bool a[], bool b[], bool w[], const int SIZE)
{
    int i;
    for (i=0; i<SIZE; i++) {
        w[i] = a[i] && b[i];
    }
}

void or (bool a[], bool b[], bool w[], const int SIZE){ ... }
```

VectorPrimitives.cpp

Loop and index need size

Logic vectors overloaded basic functions

Logic Vector


vectorPrimitives.cpp | vectorFunctions.cpp

Logic Vector Simulation | (Global Scope)

```cpp
#include "vectorPrimitives.h"
#include "vectorFunctions.h"

void getBits (string vectorName, int numBits, bool values[])
{
    string valuesS;
    int i;
    cout << "Enter " << numBits << " bits of " << vectorName << ": ";
    cin >> valuesS;
    for (i=0; i<numBits; i++){
        if (valuesS[i] == '1') values[i] = true;
        else values[i] = false;
    }
}

void putBits (string vectorName, int numBits, bool values[]){ ... }
void two2OneMux (bool a[], bool b[], bool w[], bool sel, int SIZE=8)
{
    bool as [8];
    bool bs [8];

    int i;
    for (i=0; i<SIZE; i++) {
        as[i] = and (a[i], not(sel));
    }
    for (i=0; i<SIZE; i++) {
        bs[i] = and (b[i], sel);
    }

    or (as, bs, w, SIZE);
}

void two2OneMuxB (bool a[], bool b[], bool w[], bool sel, int SIZE=8){ ... }

int main (){ ... }
```

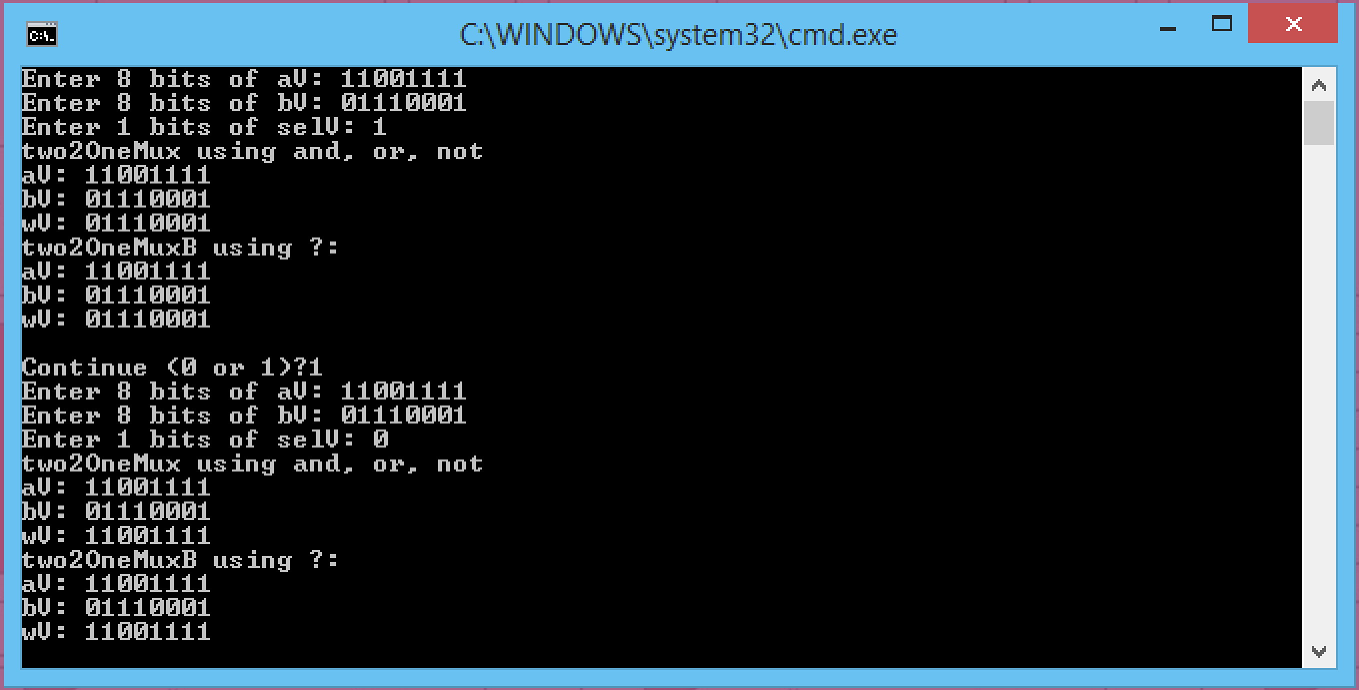
100 %

VectorFunctions.cpp

Read string and turns it into an array of bool

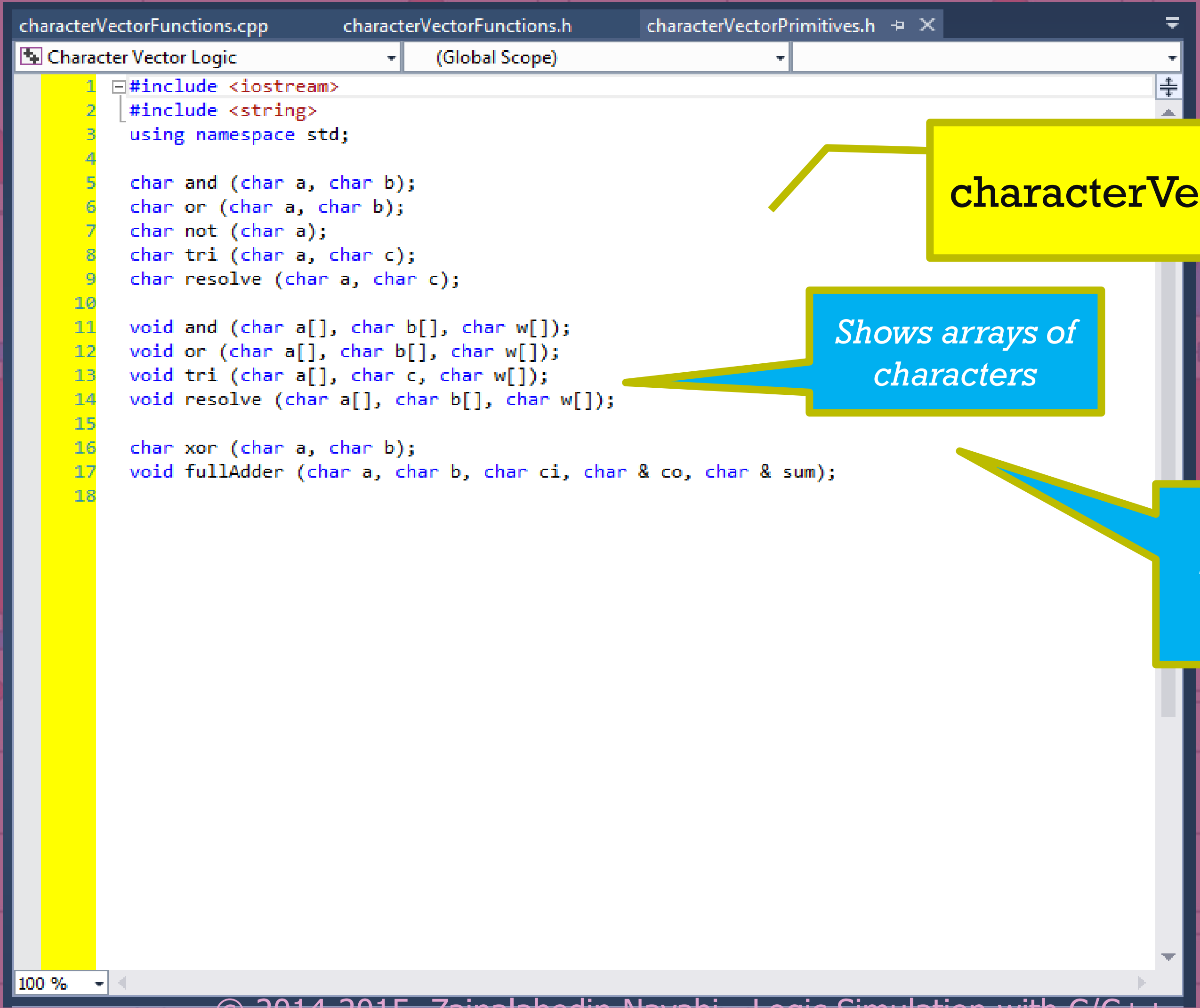
Array indexing

Logic Vector


C:\WINDOWS\system32\cmd.exe

```
Enter 8 bits of aV: 11001111
Enter 8 bits of bV: 01110001
Enter 1 bits of selV: 1
two2OneMux using and, or, not
aV: 11001111
bV: 01110001
wV: 01110001
two2OneMuxB using ?:
aV: 11001111
bV: 01110001
wV: 01110001

Continue (0 or 1)?1
Enter 8 bits of aV: 11001111
Enter 8 bits of bV: 01110001
Enter 1 bits of selV: 0
two2OneMux using and, or, not
aV: 11001111
bV: 01110001
wV: 11001111
two2OneMuxB using ?:
aV: 11001111
bV: 01110001
wV: 11001111
```

Logic Vector


characterVectorFunctions.cpp
characterVectorFunctions.h
characterVectorPrimitives.h
Character Vector Logic
(Global Scope)
100 %

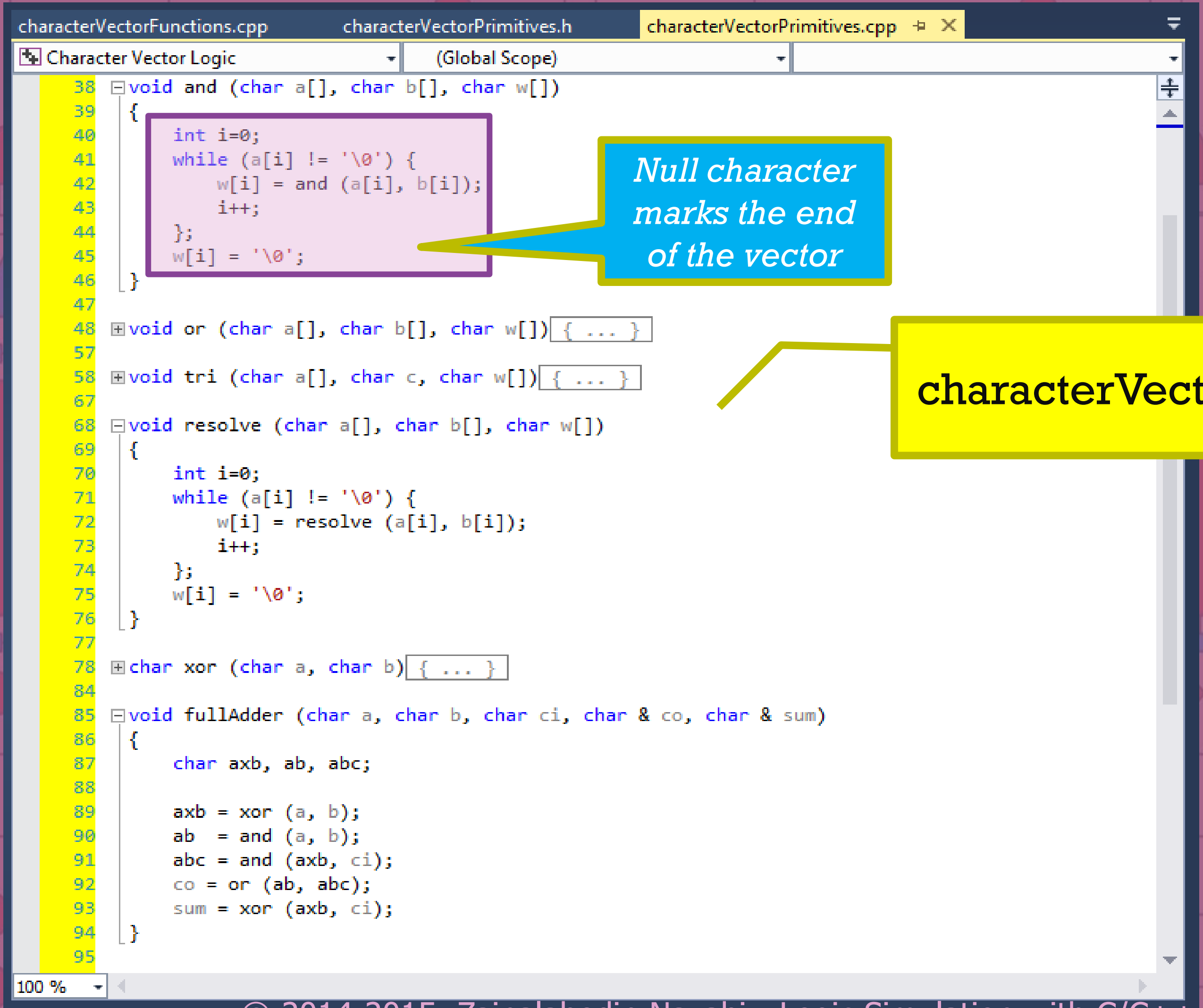
```
#include <iostream>
#include <string>
using namespace std;

char and (char a, char b);
char or (char a, char b);
char not (char a);
char tri (char a, char c);
char resolve (char a, char c);

void and (char a[], char b[], char w[]);
void or (char a[], char b[], char w[]);
void tri (char a[], char c, char w[]);
void resolve (char a[], char b[], char w[]);

char xor (char a, char b);
void fullAdder (char a, char b, char ci, char & co, char & sum);
```

characterVectorPrimitives.h

Shows arrays of characters

Char-based primitives and their vector overloading

Logic Vector

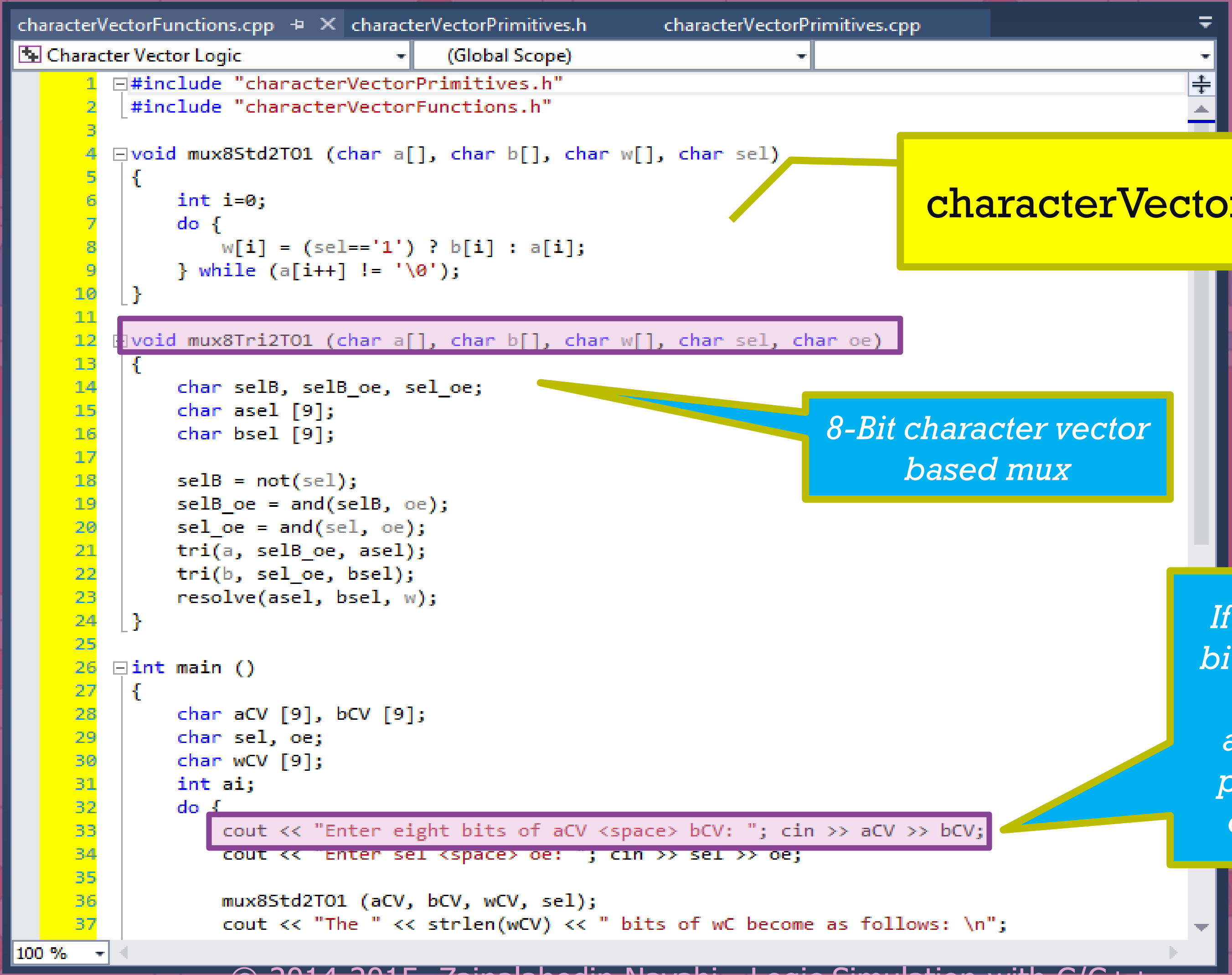

characterVectorFunctions.cpp
characterVectorPrimitives.h
characterVectorPrimitives.cpp
Character Vector Logic
(Global Scope)
100 %

```
void and (char a[], char b[], char w[])
{
    int i=0;
    while (a[i] != '\0') {
        w[i] = and (a[i], b[i]);
        i++;
    };
    w[i] = '\0';
}

void or (char a[], char b[], char w[]){ ... }

void tri (char a[], char c, char w[]){ ... }

void resolve (char a[], char b[], char w[])
{
    int i=0;
    while (a[i] != '\0') {
        w[i] = resolve (a[i], b[i]);
        i++;
    };
    w[i] = '\0';
}

char xor (char a, char b){ ... }

void fullAdder (char a, char b, char ci, char & co, char & sum)
{
    char axb, ab, abc;

    axb = xor (a, b);
    ab  = and (a, b);
    abc = and (axb, ci);
    co = or (ab, abc);
    sum = xor (axb, ci);
}
```

Null character marks the end of the vector

characterVectorPrimitives.cpp

Logic Vector


characterVectorFunctions.cpp
characterVectorPrimitives.h
characterVectorPrimitives.cpp
Character Vector Logic
(Global Scope)
100 %

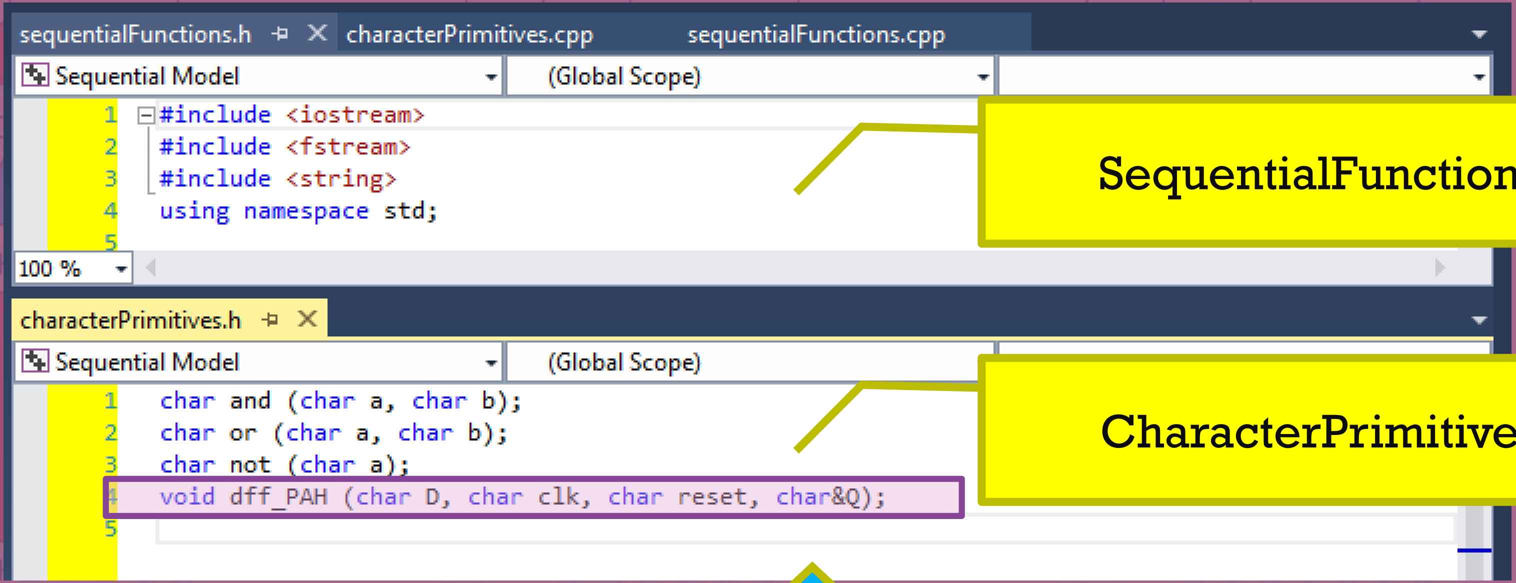
```
#include "characterVectorPrimitives.h"
#include "characterVectorFunctions.h"

void mux8Std2TO1 (char a[], char b[], char w[], char sel)
{
    int i=0;
    do {
        w[i] = (sel=='1') ? b[i] : a[i];
    } while (a[i++] != '\0');
}

void mux8Tri2TO1 (char a[], char b[], char w[], char sel, char oe)
{
    char selB, selB_oe, sel_oe;
    char asel [9];
    char bsel [9];

    selB = not(sel);
    selB_oe = and(selB, oe);
    sel_oe = and(sel, oe);
    tri(a, selB_oe, asel);
    tri(b, sel_oe, bsel);
    resolve(asel, bsel, w);
}

int main ()
{
    char aCV [9], bCV [9];
    char sel, oe;
    char wCV [9];
    int ai;
    do {
        cout << "Enter eight bits of aCV <space> bCV: "; cin >> aCV >> bCV;
        cout << "Enter sel <space> oe: "; cin >> sel >> oe;

        mux8Std2TO1 (aCV, bCV, wCV, sel);
        cout << "The " << strlen(wCV) << " bits of wC become as follows: \n";
```

characterVectorFunctions.cpp

8-Bit character vector based mux

If fewer than 8-bits are entered, using cin automatically puts '\0' at the end of string

Sequential Circuit Modeling



sequentialFunctions.h | characterPrimitives.cpp | sequentialFunctions.cpp

Sequential Model | (Global Scope)

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

characterPrimitives.h

Sequential Model | (Global Scope)

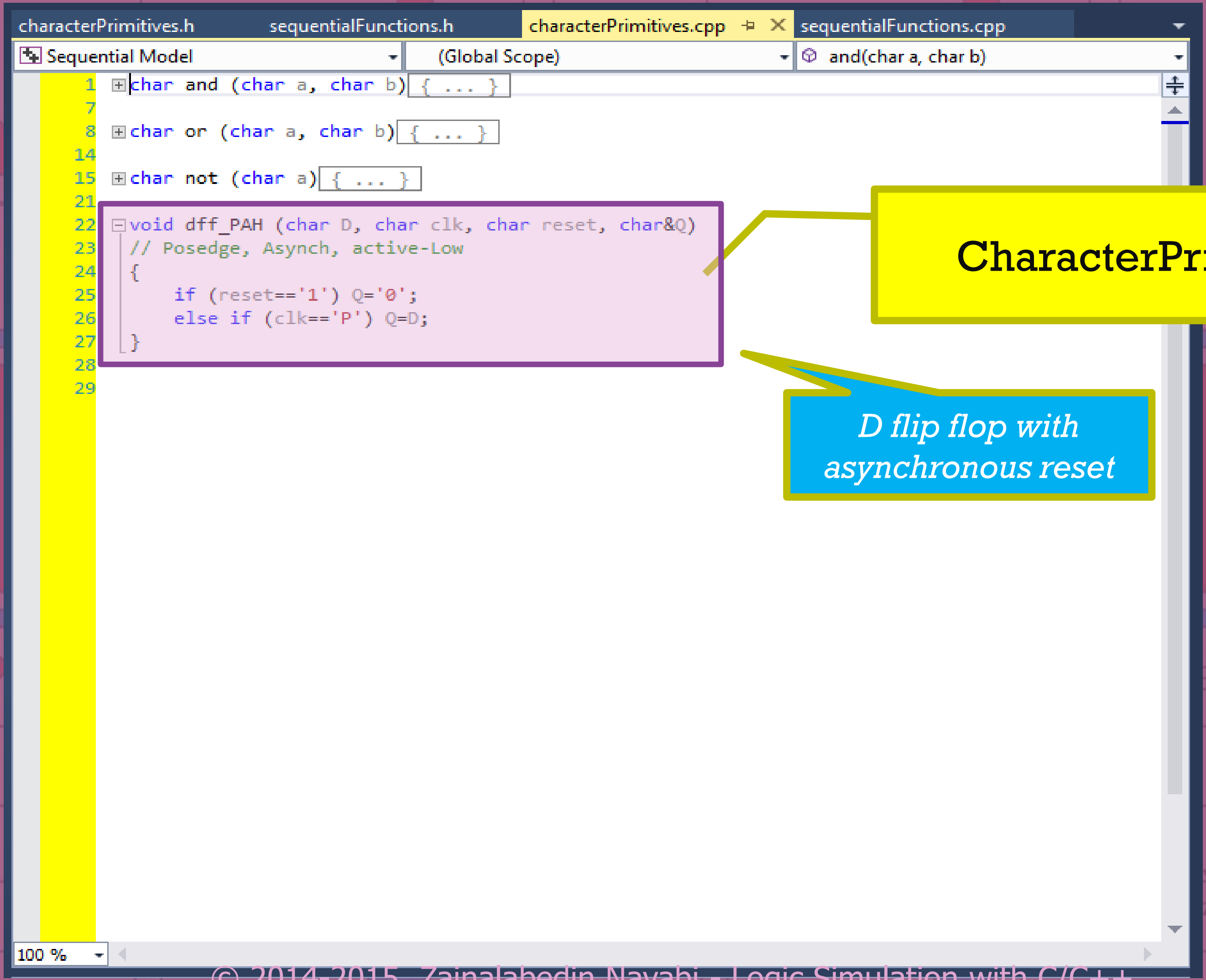
```
char and (char a, char b);
char or (char a, char b);
char not (char a);
void dff_PAH (char D, char clk, char reset, char&Q);
```

SequentialFunctions.h

CharacterPrimitives.h

DFFaLRhE = D filp flop active low reset active high enable

Sequential Circuit Modeling

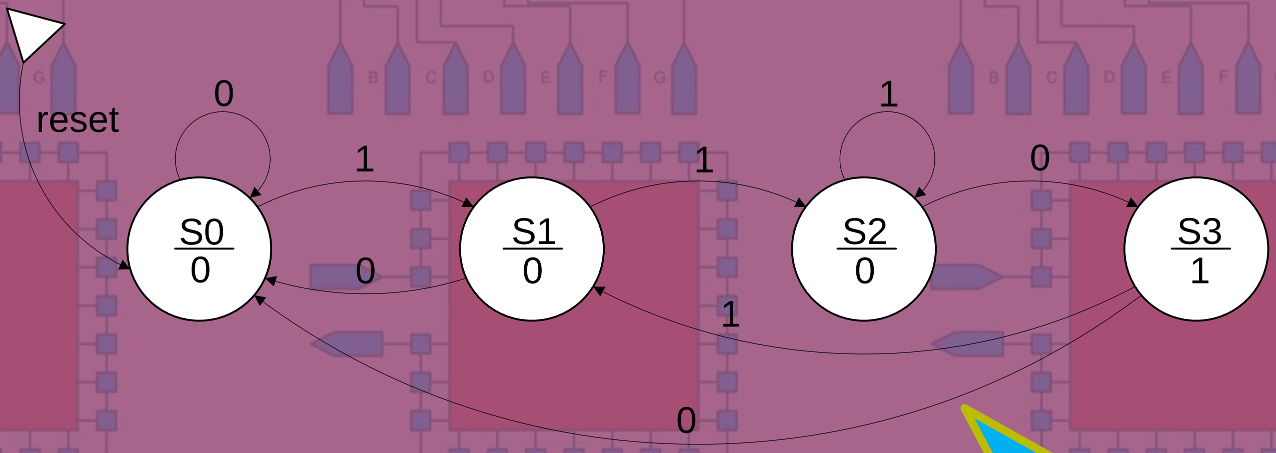


```
char and (char a, char b){ ... }

char or (char a, char b){ ... }

char not (char a){ ... }

void dff_PAH (char D, char clk, char reset, char&Q)
// Posedge, Asynch, active-Low
{
    if (reset=='1') Q='0';
    else if (clk=='P') Q=D;
}
```

CharacterPrimitives.cpp

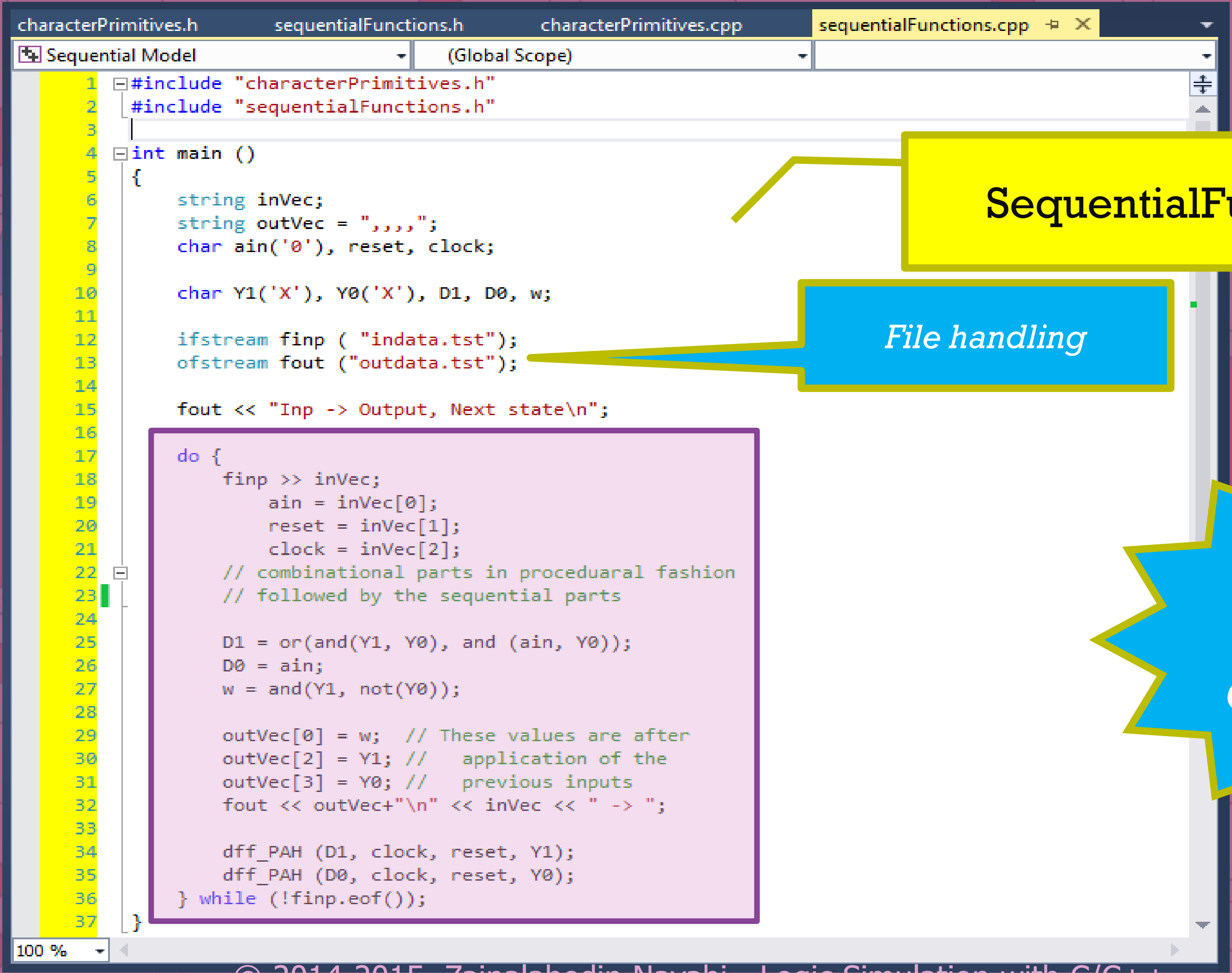
D flip flop with asynchronous reset

Sequential Circuit Modeling



Moore 110 sequence detector

Sequential Circuit Modeling



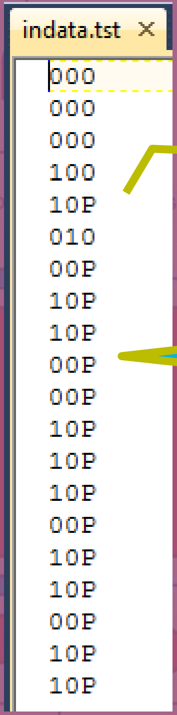
```
#include "characterPrimitives.h"
#include "sequentialFunctions.h"

int main ()
{
    string inVec;
    string outVec = ",,,,";
    char ain('0'), reset, clock;

    char Y1('X'), Y0('X'), D1, D0, w;

    ifstream finp ( "indata.tst");
    ofstream fout ("outdata.tst");

    fout << "Inp -> Output, Next state\n";

    do {
        finp >> inVec;
            ain = inVec[0];
            reset = inVec[1];
            clock = inVec[2];
        // combinational parts in proceduaral fashion
        // followed by the sequential parts

        D1 = or(and(Y1, Y0), and (ain, Y0));
        D0 = ain;
        w = and(Y1, not(Y0));

        outVec[0] = w;  // These values are after
        outVec[2] = Y1; //   application of the
        outVec[3] = Y0; //   previous inputs
        fout << outVec+"\n" << inVec << " -> ";

        dff_PAH (D1, clock, reset, Y1);
        dff_PAH (D0, clock, reset, Y0);
    } while (!finp.eof());
}
```

SequentialFunctions.cpp

File handling

Convert string to char. Operations in char

Sequential Circuit Modeling



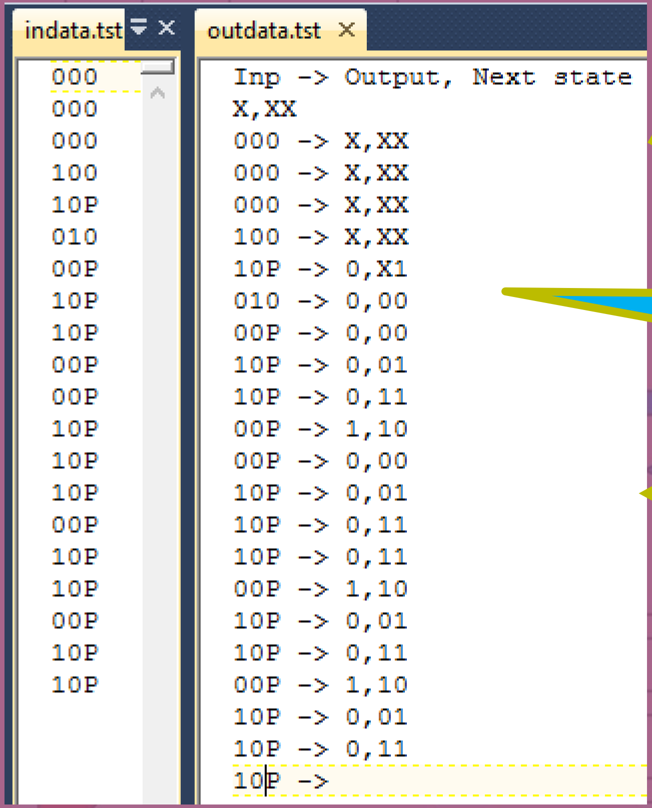
indata.tst

```
000
000
000
100
10P
010
00P
10P
10P
00P
00P
10P
10P
10P
00P
10P
10P
00P
10P
10P
```

Indata.tst

Ain, reset, clock

Sequential Circuit Modeling



indata.tst

```
000
000
000
100
10P
010
00P
10P
10P
00P
00P
10P
10P
10P
00P
10P
10P
00P
10P
10P
```

outdata.tst

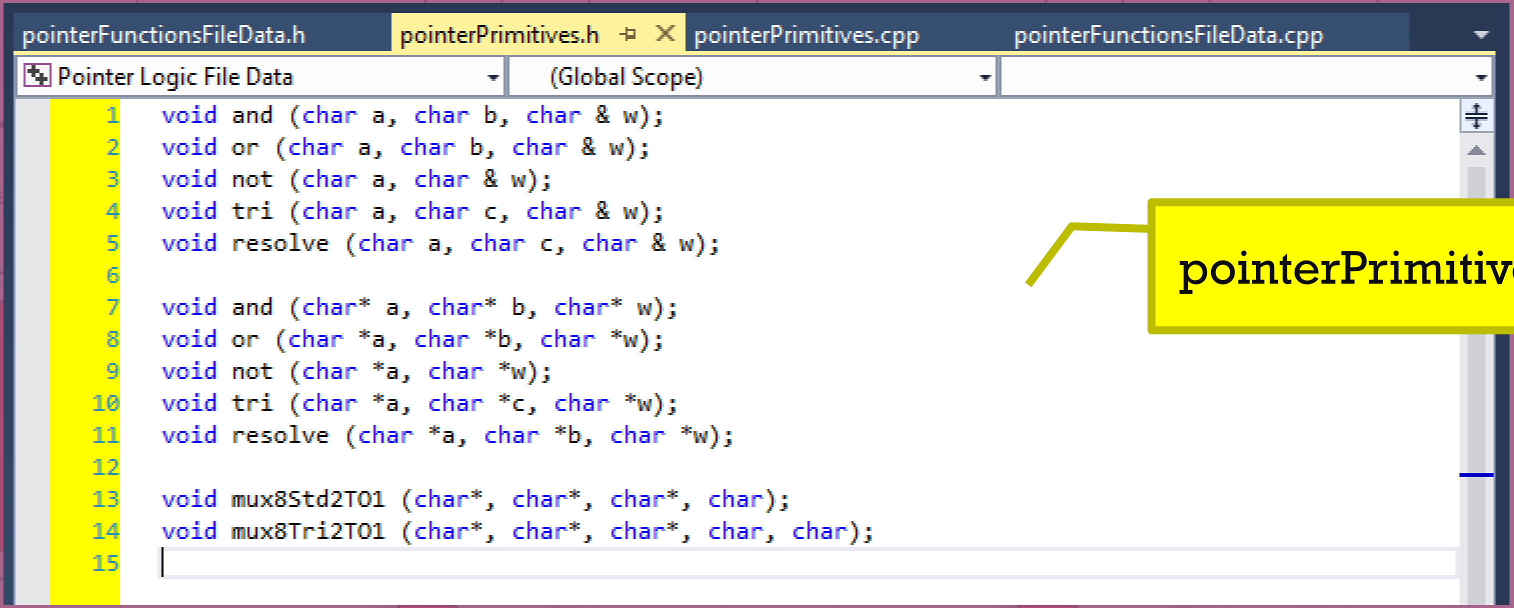
```
Inp -> Output, Next state
X,XX
000 -> X,XX
000 -> X,XX
000 -> X,XX
100 -> X,XX
10P -> 0,X1
010 -> 0,00
00P -> 0,00
10P -> 0,01
10P -> 0,11
00P -> 1,10
00P -> 0,00
10P -> 0,01
10P -> 0,11
10P -> 0,11
00P -> 1,10
10P -> 0,01
10P -> 0,11
00P -> 1,10
10P -> 0,01
10P -> 0,11
10P ->
```

outdata.tst

W and 2 bits of states

Clock by clock output

Using Pointers for Logic Vectors

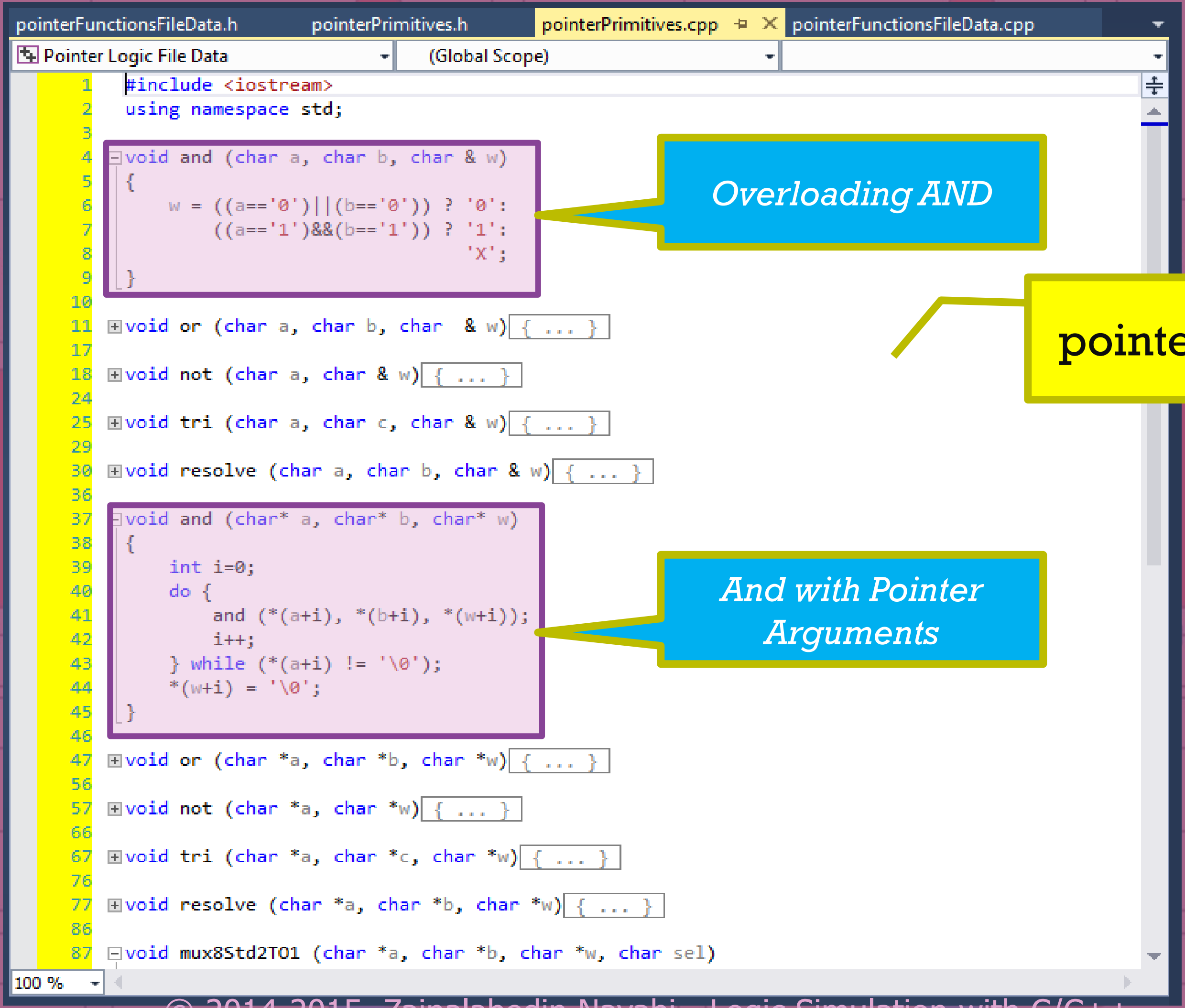


```
void and (char a, char b, char & w);
void or (char a, char b, char & w);
void not (char a, char & w);
void tri (char a, char c, char & w);
void resolve (char a, char c, char & w);

void and (char* a, char* b, char* w);
void or (char *a, char *b, char *w);
void not (char *a, char *w);
void tri (char *a, char *c, char *w);
void resolve (char *a, char *b, char *w);

void mux8Std2TO1 (char*, char*, char*, char);
void mux8Tri2TO1 (char*, char*, char*, char, char);
```

pointerPrimitives.h

Pointers instead of arrays

Using Pointers for Logic Vectors



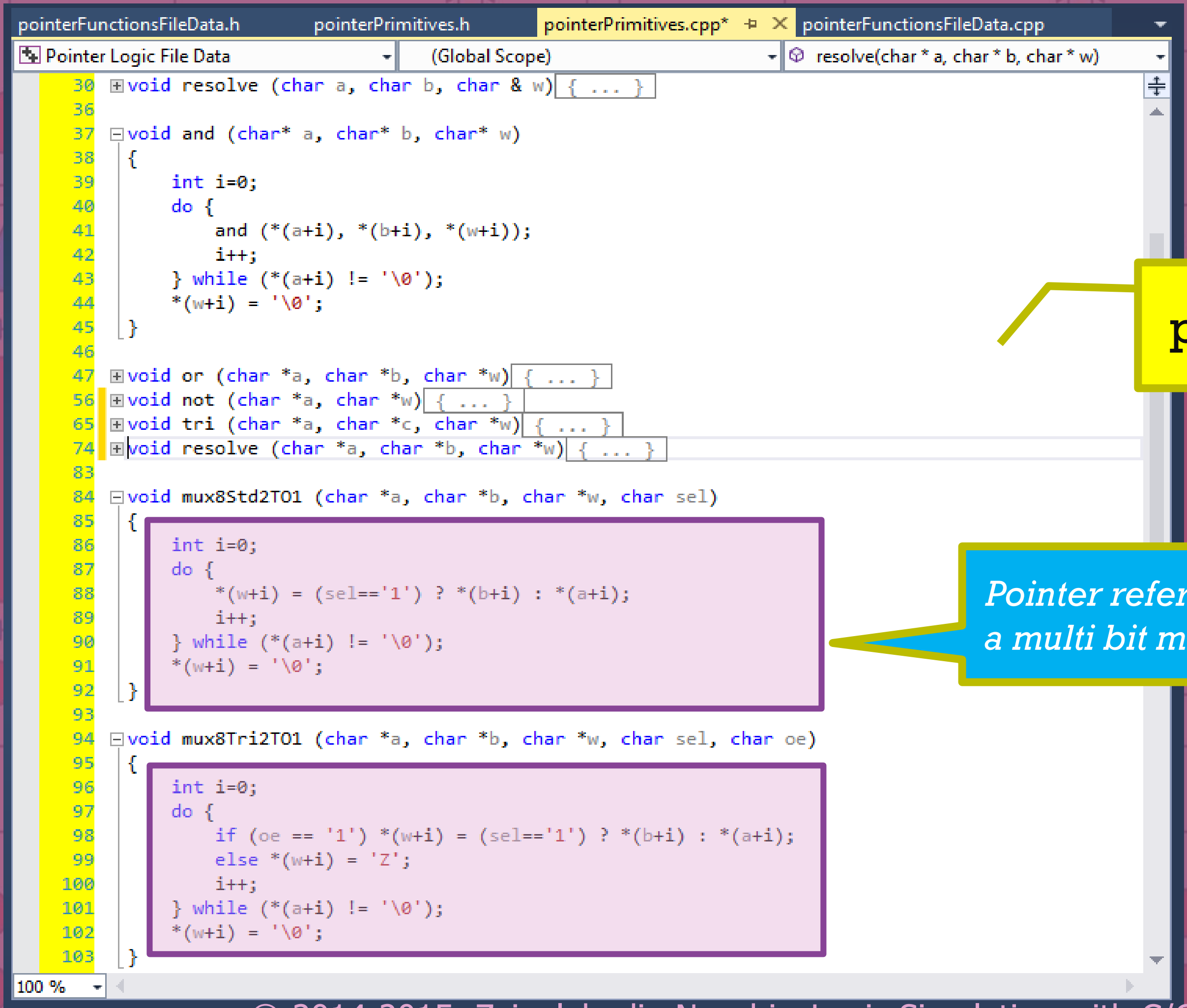
```
#include <iostream>
using namespace std;

void and (char a, char b, char & w)
{
    w = ((a=='0')||(b=='0')) ? '0':
        ((a=='1')&&(b=='1')) ? '1':
                               'X';
}

void or (char a, char b, char  & w){ ... }
void not (char a, char & w){ ... }
void tri (char a, char c, char & w){ ... }
void resolve (char a, char b, char & w){ ... }
void and (char* a, char* b, char* w)
{
    int i=0;
    do {
        and (*(a+i), *(b+i), *(w+i));
        i++;
    } while (*(a+i) != '\0');
    *(w+i) = '\0';
}

void or (char *a, char *b, char *w){ ... }
void not (char *a, char *w){ ... }
void tri (char *a, char *c, char *w){ ... }
void resolve (char *a, char *b, char *w){ ... }
void mux8Std2TO1 (char *a, char *b, char *w, char sel)
```

Overloading AND

pointerPrimitives.cpp

And with Pointer Arguments

Using Pointers for Logic Vectors

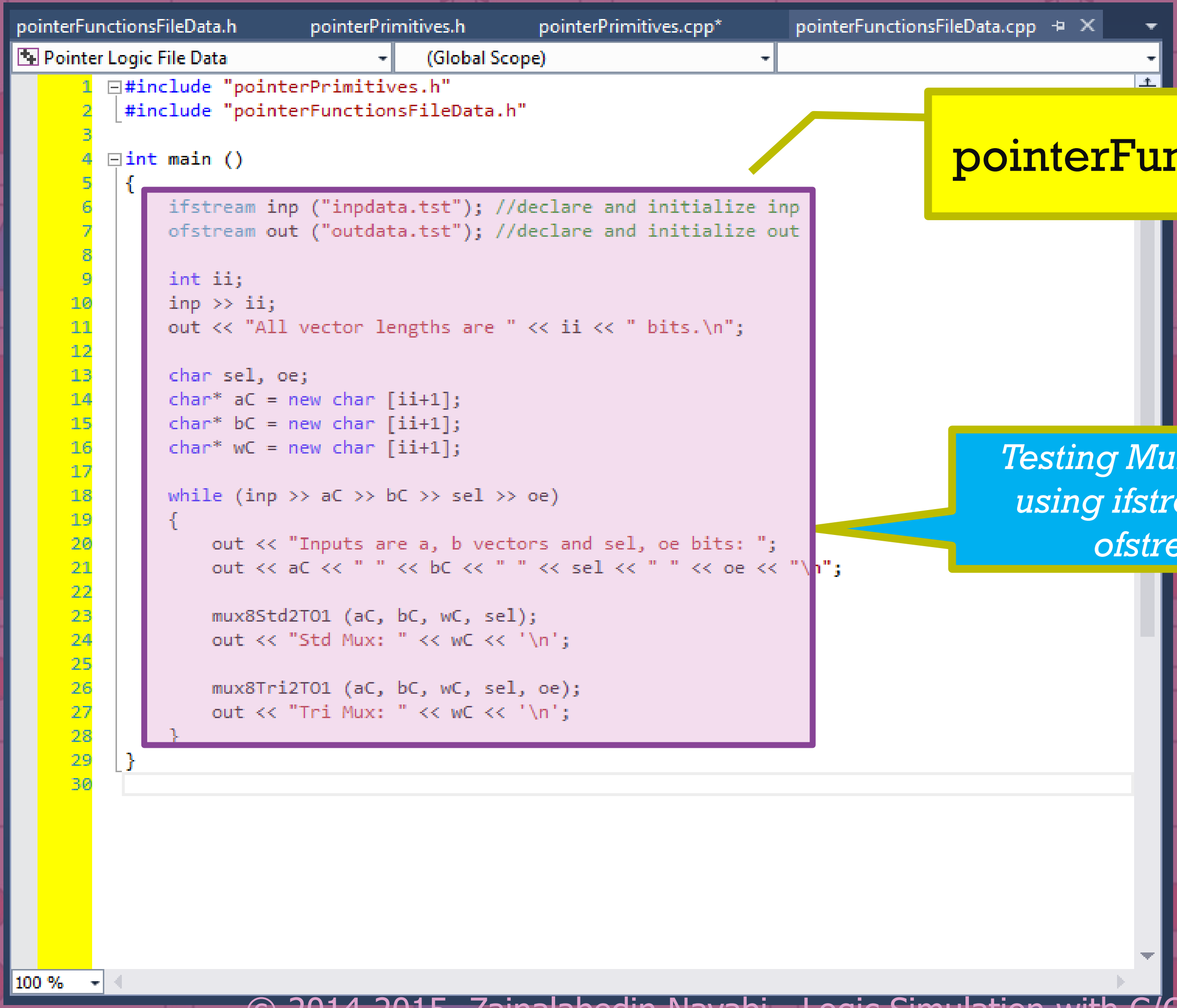

pointerFunctionsFileData.h
pointerPrimitives.h
pointerPrimitives.cpp*
pointerFunctionsFileData.cpp
Pointer Logic File Data
(Global Scope)
resolve(char * a, char * b, char * w)
100 %

```
void resolve (char a, char b, char & w) { ... }

void and (char* a, char* b, char* w)
{
    int i=0;
    do {
        and (*(a+i), *(b+i), *(w+i));
        i++;
    } while (*(a+i) != '\0');
    *(w+i) = '\0';
}

void or (char *a, char *b, char *w) { ... }
void not (char *a, char *w) { ... }
void tri (char *a, char *c, char *w) { ... }
void resolve (char *a, char *b, char *w) { ... }

void mux8Std2TO1 (char *a, char *b, char *w, char sel)
{
    int i=0;
    do {
        *(w+i) = (sel=='1') ? *(b+i) : *(a+i);
        i++;
    } while (*(a+i) != '\0');
    *(w+i) = '\0';
}

void mux8Tri2TO1 (char *a, char *b, char *w, char sel, char oe)
{
    int i=0;
    do {
        if (oe == '1') *(w+i) = (sel=='1') ? *(b+i) : *(a+i);
        else *(w+i) = 'Z';
        i++;
    } while (*(a+i) != '\0');
    *(w+i) = '\0';
}
```

pointerPrimitives.cpp

Pointer referencing in a multi bit multiplexer

Using Pointers for Logic Vectors

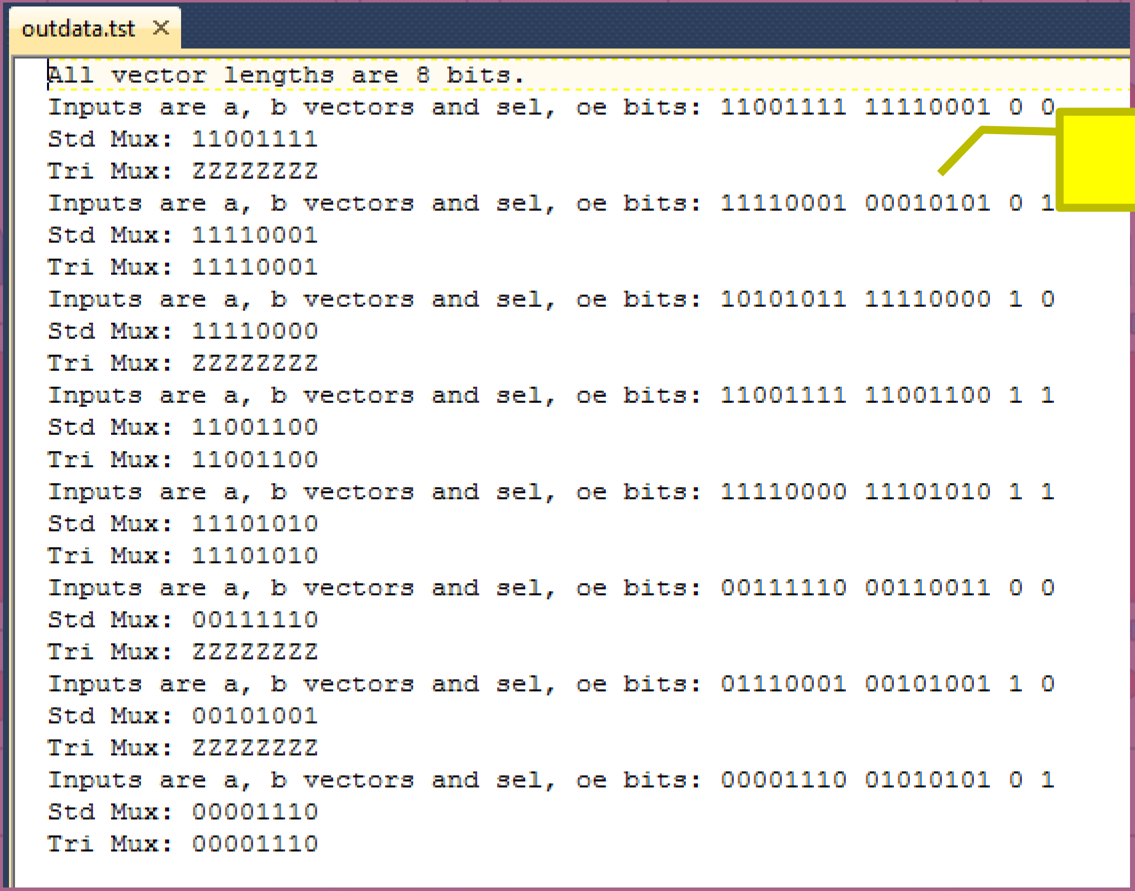


```
#include "pointerPrimitives.h"
#include "pointerFunctionsFileData.h"

int main ()
{
    ifstream inp ("inpdata.tst"); //declare and initialize inp
    ofstream out ("outdata.tst"); //declare and initialize out

    int ii;
    inp >> ii;
    out << "All vector lengths are " << ii << " bits.\n";

    char sel, oe;
    char* aC = new char [ii+1];
    char* bC = new char [ii+1];
    char* wC = new char [ii+1];

    while (inp >> aC >> bC >> sel >> oe)
    {
        out << "Inputs are a, b vectors and sel, oe bits: ";
        out << aC << " " << bC << " " << sel << " " << oe << "\n";

        mux8Std2TO1 (aC, bC, wC, sel);
        out << "Std Mux: " << wC << '\n';

        mux8Tri2TO1 (aC, bC, wC, sel, oe);
        out << "Tri Mux: " << wC << '\n';
    }
}
```

pointerFunctionsFileData.cpp

Testing Multiplexers using ifstream and ofstream

Using Pointers for Logic Vectors



outdata.tst

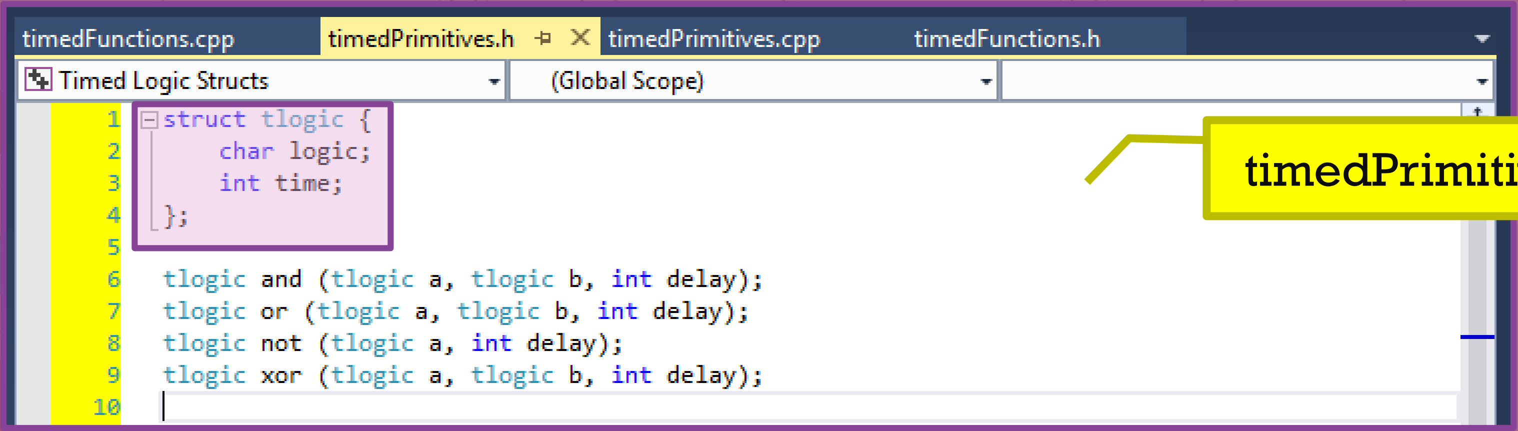
```
All vector lengths are 8 bits.
Inputs are a, b vectors and sel, oe bits: 11001111 11110001 0 0
Std Mux: 11001111
Tri Mux: ZZZZZZZZ
Inputs are a, b vectors and sel, oe bits: 11110001 00010101 0 1
Std Mux: 11110001
Tri Mux: 11110001
Inputs are a, b vectors and sel, oe bits: 10101011 11110000 1 0
Std Mux: 11110000
Tri Mux: ZZZZZZZZ
Inputs are a, b vectors and sel, oe bits: 11001111 11001100 1 1
Std Mux: 11001100
Tri Mux: 11001100
Inputs are a, b vectors and sel, oe bits: 11110000 11101010 1 1
Std Mux: 11101010
Tri Mux: 11101010
Inputs are a, b vectors and sel, oe bits: 00111110 00110011 0 0
Std Mux: 00111110
Tri Mux: ZZZZZZZZ
Inputs are a, b vectors and sel, oe bits: 01110001 00101001 1 0
Std Mux: 00101001
Tri Mux: ZZZZZZZZ
Inputs are a, b vectors and sel, oe bits: 00001110 01010101 0 1
Std Mux: 00001110
Tri Mux: 00001110
```

Outdata.tst

Logic Simulation with C/C++

- Procedural Languages for Hardware Modeling
- Types and Operators for Logic Modeling
- Basic Logic Simulation
  - Logic functions
  - Function overloading
  - Passing logic functions
  - Using default values
  - Building higher level structures
  - Handling 4-value logic
  - Logic vector
  - Sequential circuit modeling
  - Using pointers for logic vectors
- Enhanced logic simulation with timing
  - Using struct for timing and logic
  - Gates that handle timing
  - Utility functions
  - Timing in logic structures
  - Overloading logical operators
  - Using Boolean expressions
- More Functions for Wires and Gates
  - Gate classes
  - Carrier centric modeling
  - Compatible scalar and vector

Using Struct for Logic and Timing

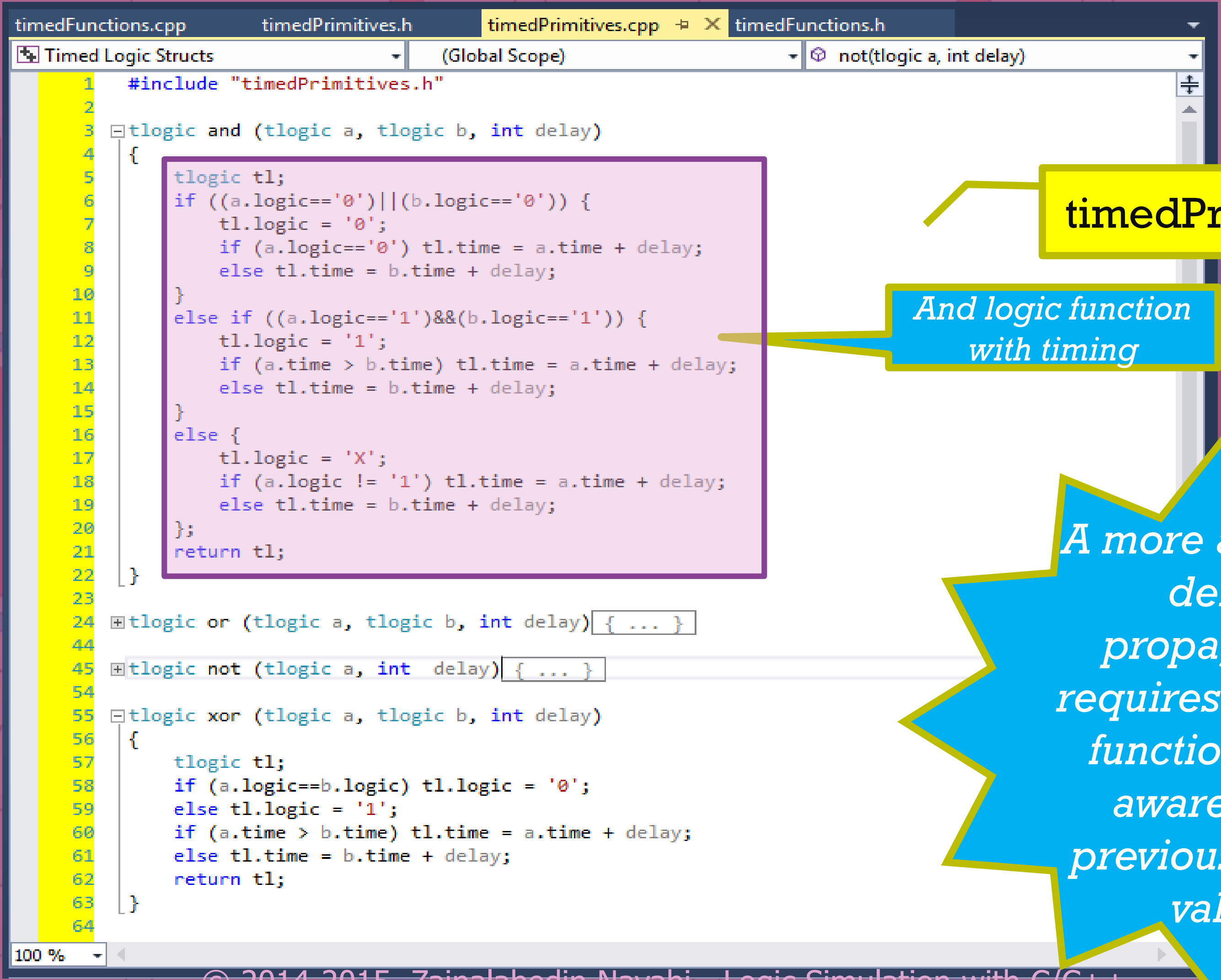


```
struct tlogic {
    char logic;
    int time;
};

tlogic and (tlogic a, tlogic b, int delay);
tlogic or (tlogic a, tlogic b, int delay);
tlogic not (tlogic a, int delay);
tlogic xor (tlogic a, tlogic b, int delay);
```

timedPrimitives.h

Structure to accommodate time as well as logic

Gates that Handle Timing



timedFunctions.cpp timedPrimitives.h timedPrimitives.cpp timedFunctions.h

Timed Logic Structs (Global Scope) not(tlogic a, int delay)

```
#include "timedPrimitives.h"

tlogic and (tlogic a, tlogic b, int delay)
{
    tlogic tl;
    if ((a.logic=='0')||(b.logic=='0')) {
        tl.logic = '0';
        if (a.logic=='0') tl.time = a.time + delay;
        else tl.time = b.time + delay;
    }
    else if ((a.logic=='1')&&(b.logic=='1')) {
        tl.logic = '1';
        if (a.time > b.time) tl.time = a.time + delay;
        else tl.time = b.time + delay;
    }
    else {
        tl.logic = 'X';
        if (a.logic != '1') tl.time = a.time + delay;
        else tl.time = b.time + delay;
    };
    return tl;
}

tlogic or (tlogic a, tlogic b, int delay){ ... }

tlogic not (tlogic a, int  delay){ ... }

tlogic xor (tlogic a, tlogic b, int delay)
{
    tlogic tl;
    if (a.logic==b.logic) tl.logic = '0';
    else tl.logic = '1';
    if (a.time > b.time) tl.time = a.time + delay;
    else tl.time = b.time + delay;
    return tl;
}
```

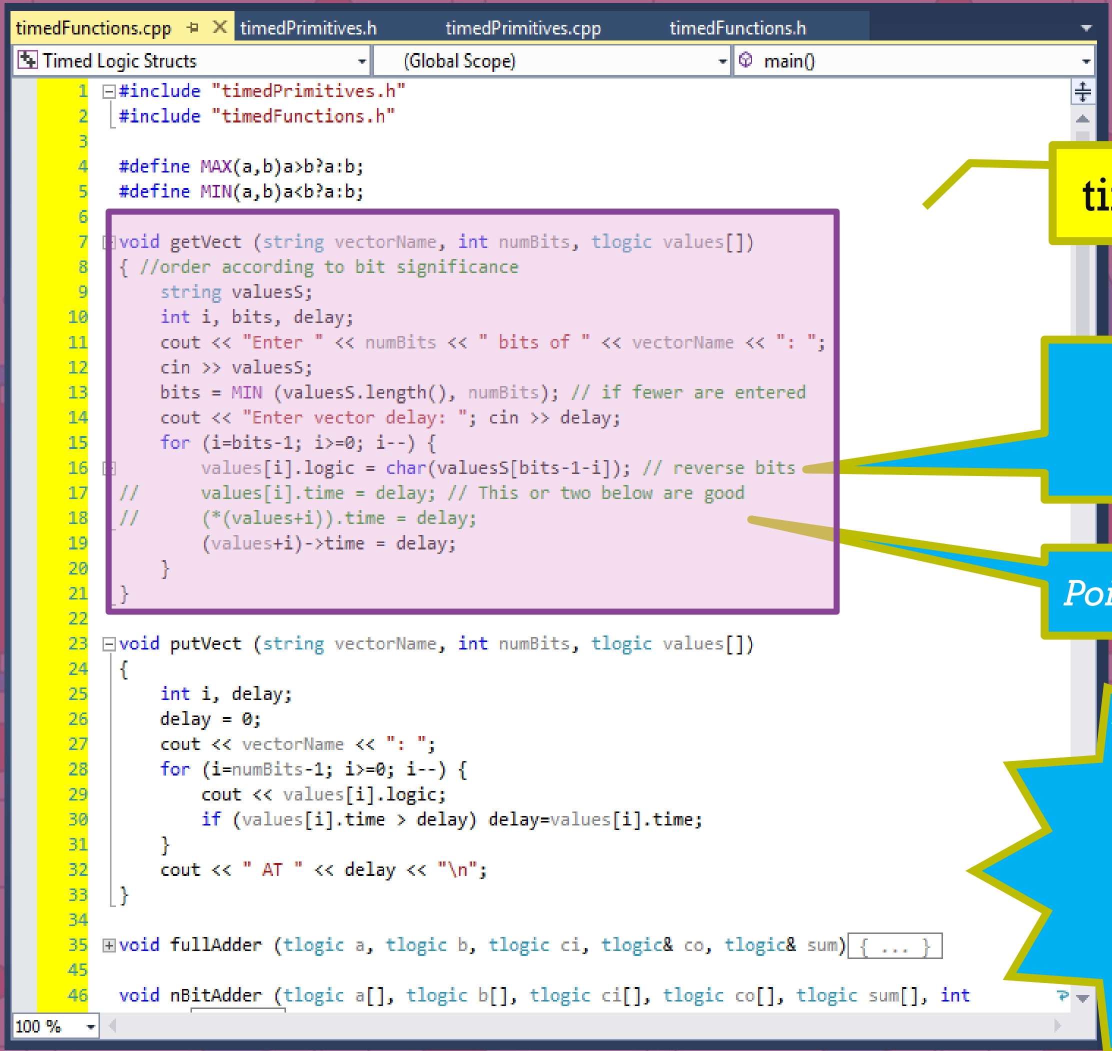
100 %

timedPrimitives.cpp

And logic function with timing

A more accurate delay propagation requires the gate function to be aware of its previous output value

Utility Functions



```
#include "timedPrimitives.h"
#include "timedFunctions.h"

#define MAX(a,b)a>b?a:b;
#define MIN(a,b)a<b?a:b;

void getVect (string vectorName, int numBits, tlogic values[])
{ //order according to bit significance
    string valuesS;
    int i, bits, delay;
    cout << "Enter " << numBits << " bits of " << vectorName << ": ";
    cin >> valuesS;
    bits = MIN (valuesS.length(), numBits); // if fewer are entered
    cout << "Enter vector delay: "; cin >> delay;
    for (i=bits-1; i>=0; i--) {
        values[i].logic = char(valuesS[bits-1-i]); // reverse bits
//      values[i].time = delay; // This or two below are good
//      (*(values+i)).time = delay;
        (values+i)->time = delay;
    }
}

void putVect (string vectorName, int numBits, tlogic values[])
{
    int i, delay;
    delay = 0;
    cout << vectorName << ": ";
    for (i=numBits-1; i>=0; i--) {
        cout << values[i].logic;
        if (values[i].time > delay) delay=values[i].time;
    }
    cout << " AT " << delay << "\n";
}

void fullAdder (tlogic a, tlogic b, tlogic ci, tlogic& co, tlogic& sum) { ... }

void nBitAdder (tlogic a[], tlogic b[], tlogic ci[], tlogic co[], tlogic sum[], int
```

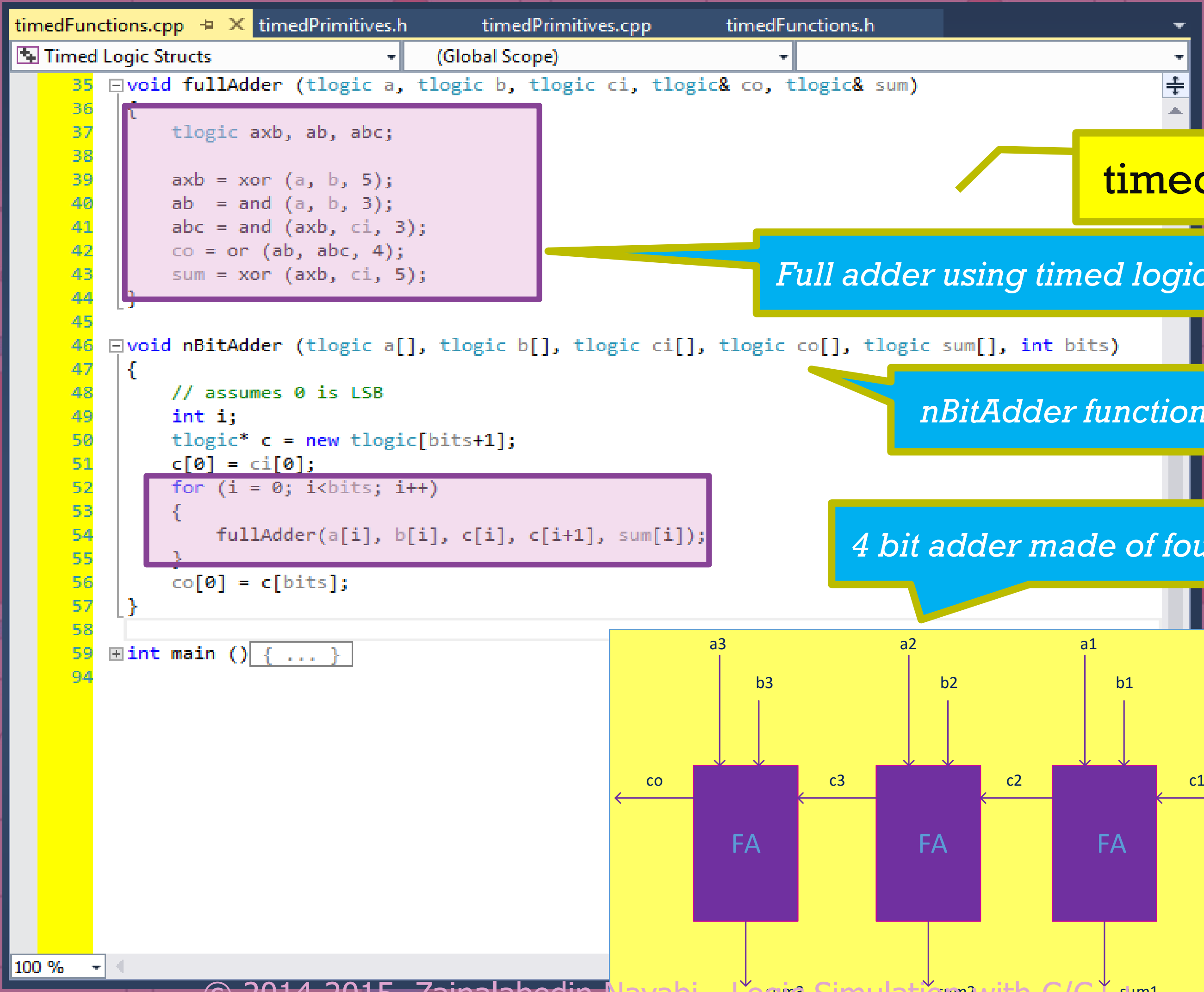
timedFunctions.cpp

Entered: 1011
ValuesS: 1011
Values: 1101

Pointer referencing

This method starts from bit 0 and treat bit 0 as logical LSB value

Timing in Logic Structures



timedFunctions.cpp | timedPrimitives.h | timedPrimitives.cpp | timedFunctions.h

Timed Logic Structs | (Global Scope)

```
void fullAdder (tlogic a, tlogic b, tlogic ci, tlogic& co, tlogic& sum)
{
    tlogic axb, ab, abc;

    axb = xor (a, b, 5);
    ab  = and (a, b, 3);
    abc = and (axb, ci, 3);
    co = or (ab, abc, 4);
    sum = xor (axb, ci, 5);
}

void nBitAdder (tlogic a[], tlogic b[], tlogic ci[], tlogic co[], tlogic sum[], int bits)
{
    // assumes 0 is LSB
    int i;
    tlogic* c = new tlogic[bits+1];
    c[0] = ci[0];
    for (i = 0; i<bits; i++)
    {
        fullAdder(a[i], b[i], c[i], c[i+1], sum[i]);
    }
    co[0] = c[bits];
}

int main (){ ... }
```

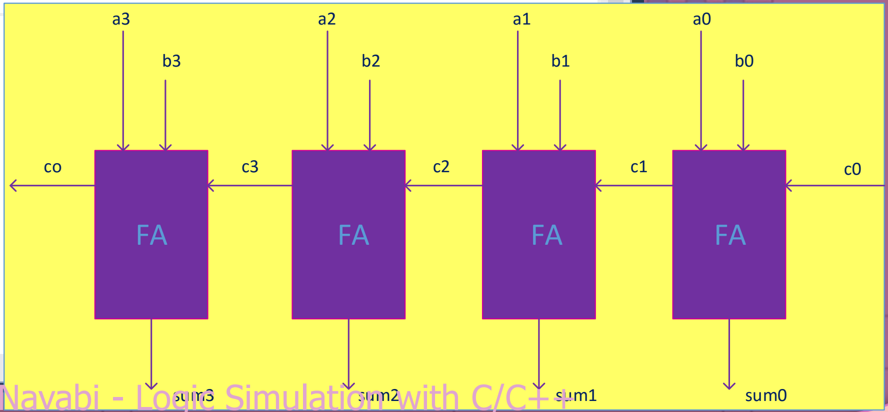
100 %

timedFunctions.cpp

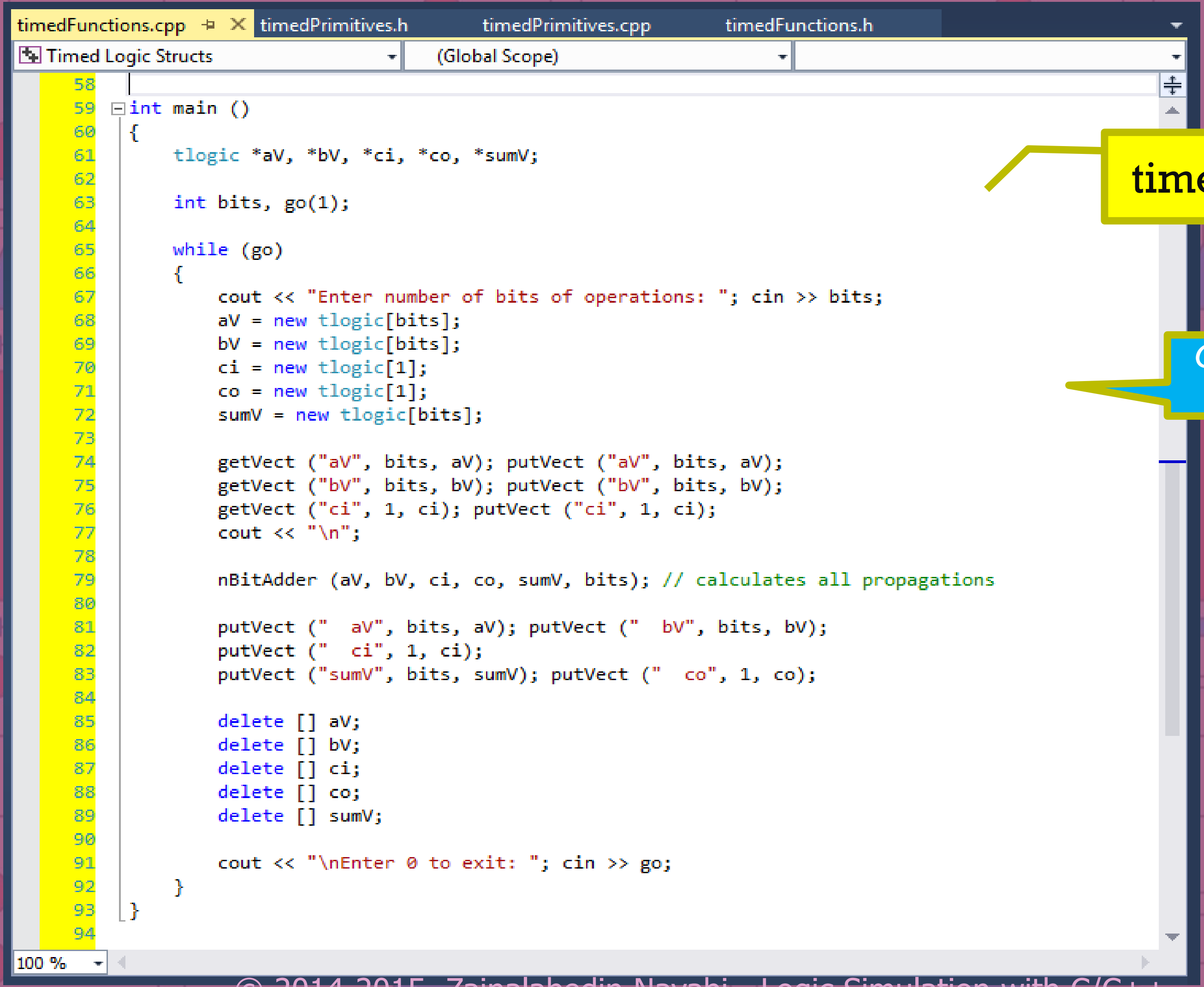
Full adder using timed logic

nBitAdder function using timed logic

4 bit adder made of four FAs



Timing in Logic Structures

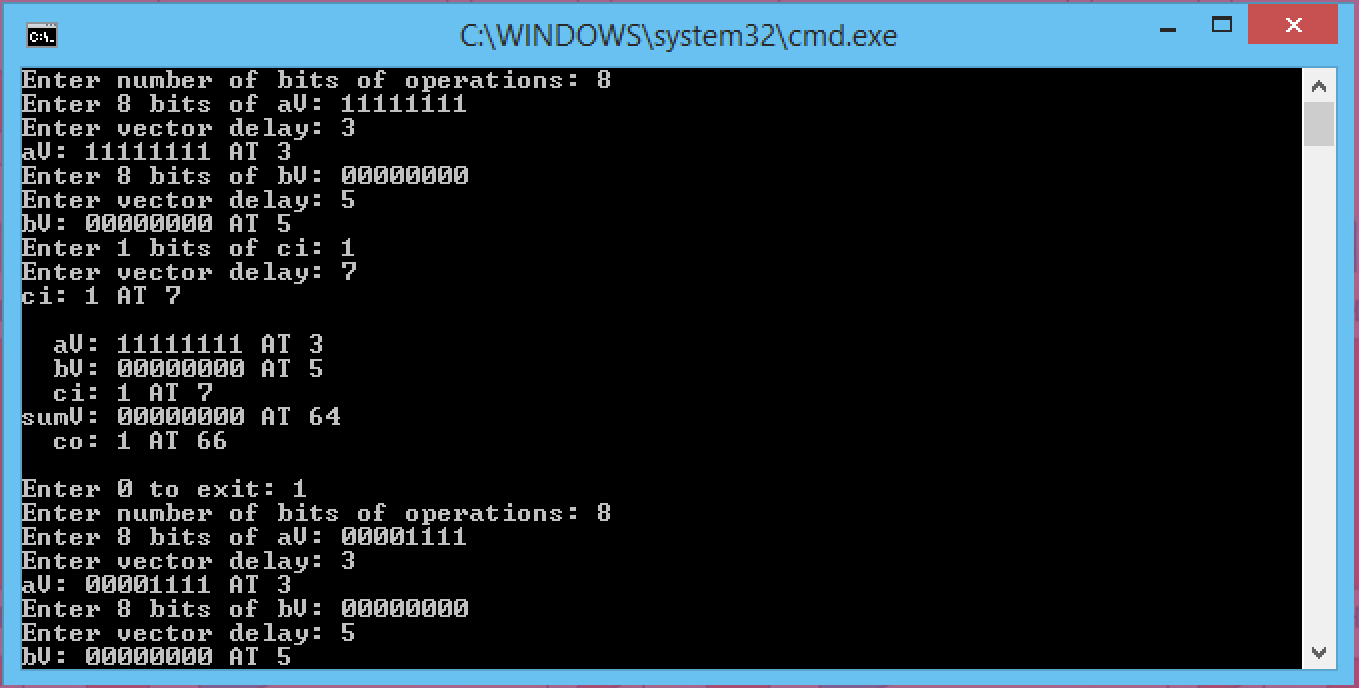


```
int main ()
{
    tlogic *aV, *bV, *ci, *co, *sumV;

    int bits, go(1);

    while (go)
    {
        cout << "Enter number of bits of operations: "; cin >> bits;
        aV = new tlogic[bits];
        bV = new tlogic[bits];
        ci = new tlogic[1];
        co = new tlogic[1];
        sumV = new tlogic[bits];

        getVect ("aV", bits, aV); putVect ("aV", bits, aV);
        getVect ("bV", bits, bV); putVect ("bV", bits, bV);
        getVect ("ci", 1, ci); putVect ("ci", 1, ci);
        cout << "\n";

        nBitAdder (aV, bV, ci, co, sumV, bits); // calculates all propagations

        putVect ("  aV", bits, aV); putVect ("  bV", bits, bV);
        putVect ("  ci", 1, ci);
        putVect ("sumV", bits, sumV); putVect ("  co", 1, co);

        delete [] aV;
        delete [] bV;
        delete [] ci;
        delete [] co;
        delete [] sumV;

        cout << "\nEnter 0 to exit: "; cin >> go;
    }
}
```

timedFunctions.cpp

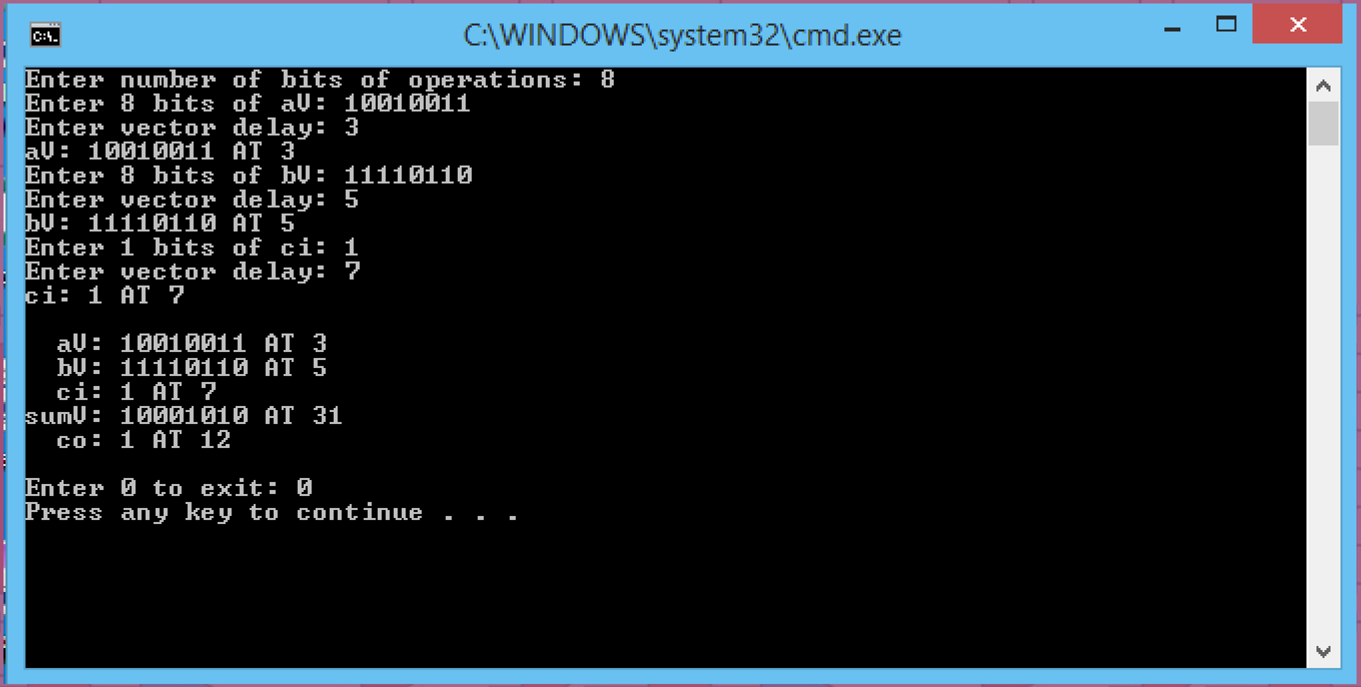
Char based adder with tlogic

Timing in Logic Structures



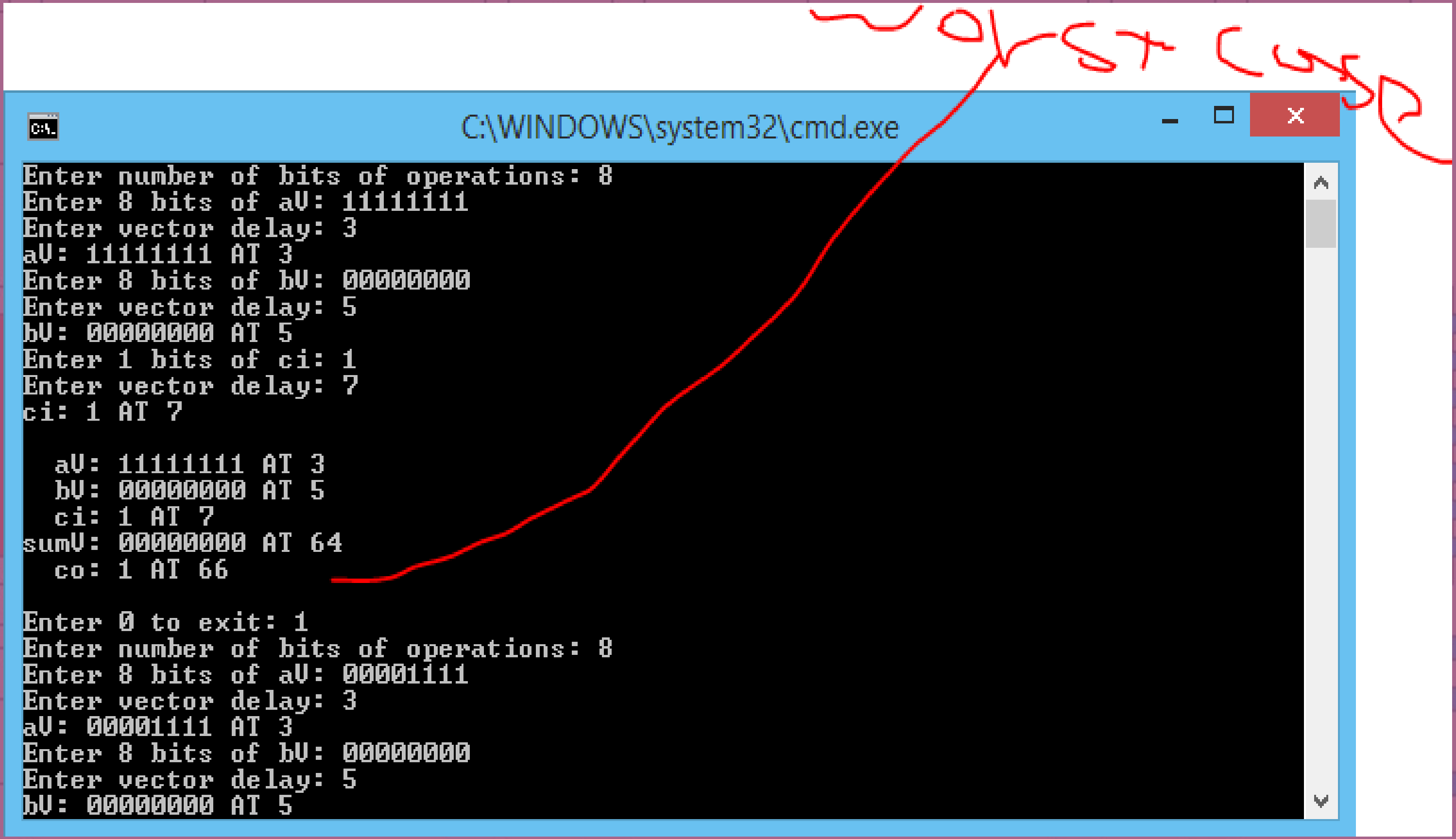
```
Enter number of bits of operations: 8
Enter 8 bits of aV: 11111111
Enter vector delay: 3
aV: 11111111 AT 3
Enter 8 bits of bV: 00000000
Enter vector delay: 5
bV: 00000000 AT 5
Enter 1 bits of ci: 1
Enter vector delay: 7
ci: 1 AT 7

  aV: 11111111 AT 3
  bV: 00000000 AT 5
  ci: 1 AT 7
sumV: 00000000 AT 64
  co: 1 AT 66

Enter 0 to exit: 1
Enter number of bits of operations: 8
Enter 8 bits of aV: 00001111
Enter vector delay: 3
aV: 00001111 AT 3
Enter 8 bits of bV: 00000000
Enter vector delay: 5
bV: 00000000 AT 5
```

Timing in Logic Structures



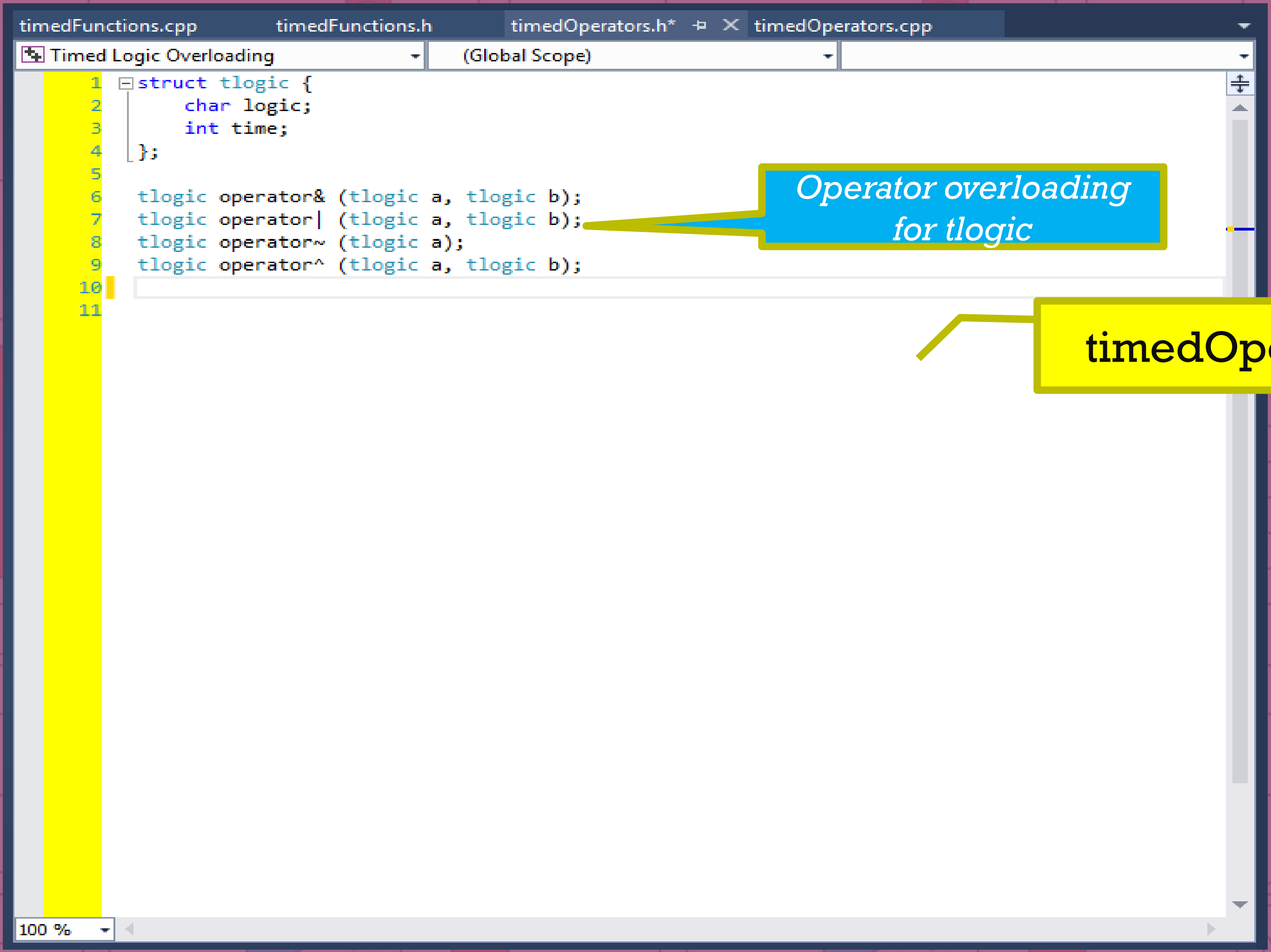
```
Enter number of bits of operations: 8
Enter 8 bits of aV: 10010011
Enter vector delay: 3
aV: 10010011 AT 3
Enter 8 bits of bV: 11110110
Enter vector delay: 5
bV: 11110110 AT 5
Enter 1 bits of ci: 1
Enter vector delay: 7
ci: 1 AT 7

   aV: 10010011 AT 3
   bV: 11110110 AT 5
   ci: 1 AT 7
sumV: 10001010 AT 31
   co: 1 AT 12

Enter 0 to exit: 0
Press any key to continue . . .
```

Timing in Logic Structures



```
Enter number of bits of operations: 8
Enter 8 bits of aV: 11111111
Enter vector delay: 3
aV: 11111111 AT 3
Enter 8 bits of bV: 00000000
Enter vector delay: 5
bV: 00000000 AT 5
Enter 1 bits of ci: 1
Enter vector delay: 7
ci: 1 AT 7

  aV: 11111111 AT 3
  bV: 00000000 AT 5
  ci: 1 AT 7
sumV: 00000000 AT 64
  co: 1 AT 66

Enter 0 to exit: 1
Enter number of bits of operations: 8
Enter 8 bits of aV: 00001111
Enter vector delay: 3
aV: 00001111 AT 3
Enter 8 bits of bV: 00000000
Enter vector delay: 5
bV: 00000000 AT 5
```

Overloading Logical Operators

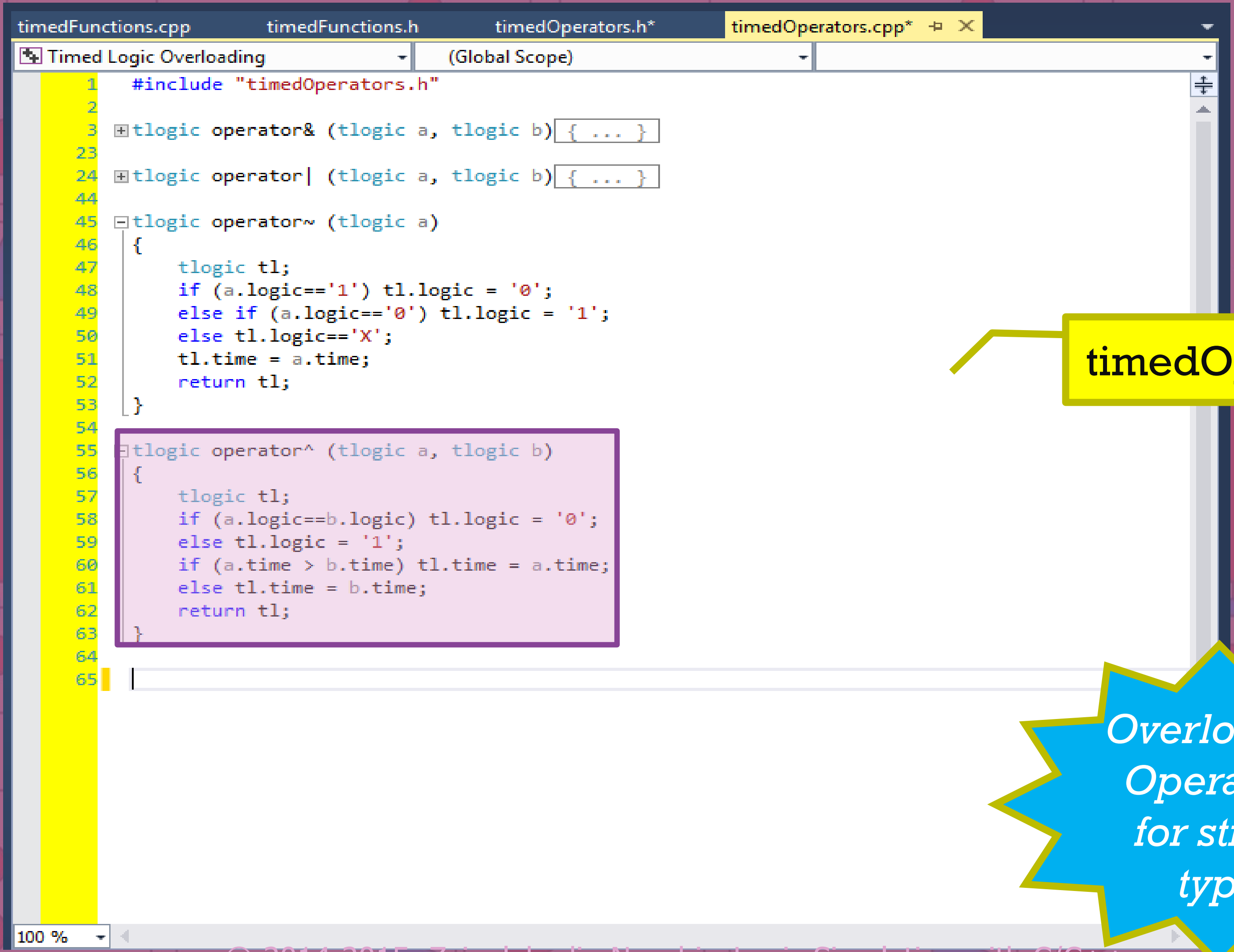


```
struct tlogic {
    char logic;
    int time;
};

tlogic operator& (tlogic a, tlogic b);
tlogic operator| (tlogic a, tlogic b);
tlogic operator~ (tlogic a);
tlogic operator^ (tlogic a, tlogic b);

```

Operator overloading for tlogic

timedOperators.h

Overloading Logical Operators



timedFunctions.cpp timedFunctions.h timedOperators.h* timedOperators.cpp*

Timed Logic Overloading (Global Scope)

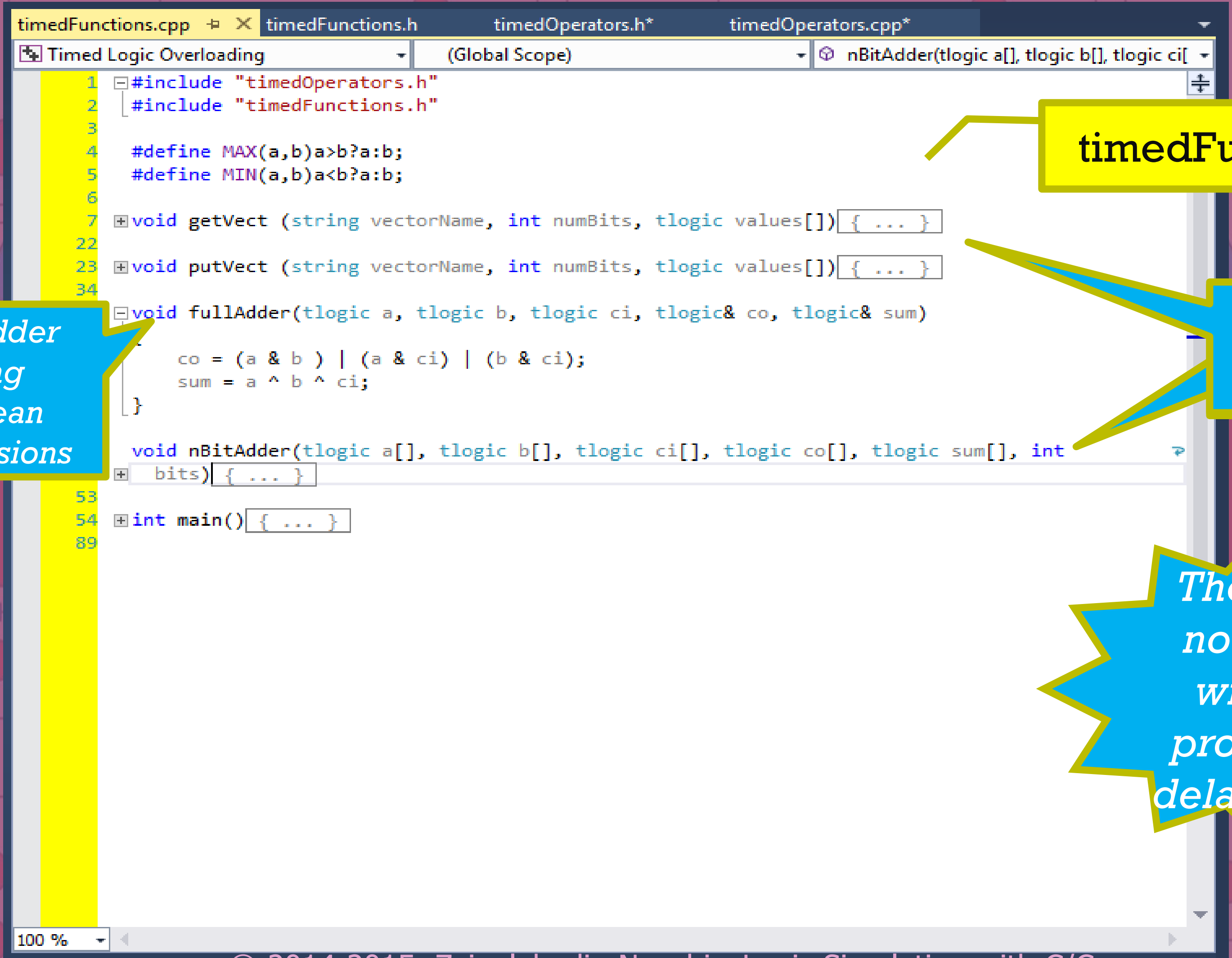
```
#include "timedOperators.h"

tlogic operator& (tlogic a, tlogic b){ ... }

tlogic operator| (tlogic a, tlogic b){ ... }

tlogic operator~ (tlogic a)
{
    tlogic tl;
    if (a.logic=='1') tl.logic = '0';
    else if (a.logic=='0') tl.logic = '1';
    else tl.logic=='X';
    tl.time = a.time;
    return tl;
}

tlogic operator^ (tlogic a, tlogic b)
{
    tlogic tl;
    if (a.logic==b.logic) tl.logic = '0';
    else tl.logic = '1';
    if (a.time > b.time) tl.time = a.time;
    else tl.time = b.time;
    return tl;
}

```

100 %

timedOperators.cpp

Overloaded Operators for struct type

Using Boolean Expressions



```
#include "timedOperators.h"
#include "timedFunctions.h"

#define MAX(a,b)a>b?a:b;
#define MIN(a,b)a<b?a:b;

void getVect (string vectorName, int numBits, tlogic values[]){ ... }

void putVect (string vectorName, int numBits, tlogic values[]){ ... }

void fullAdder(tlogic a, tlogic b, tlogic ci, tlogic& co, tlogic& sum)
{
    co = (a & b ) | (a & ci) | (b & ci);
    sum = a ^ b ^ ci;
}

void nBitAdder(tlogic a[], tlogic b[], tlogic ci[], tlogic co[], tlogic sum[], int
  bits){ ... }

int main(){ ... }
```

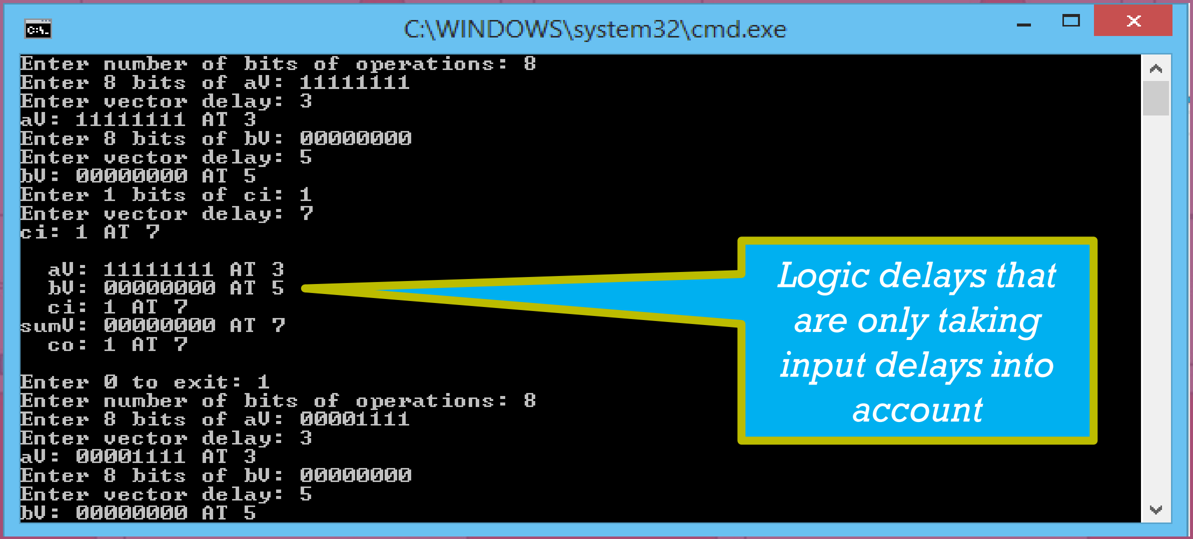
timedFunction.cpp

Full adder using Boolean expressions

Full adder considers logic and timing

There are no inside wires to propagate delay values

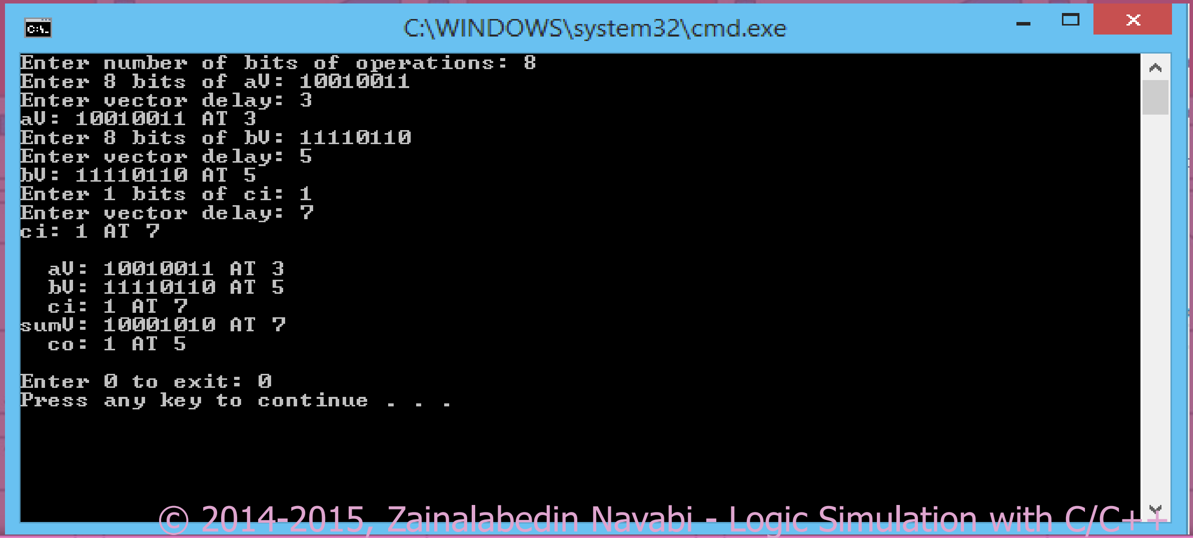
Using Boolean Expressions



C:\WINDOWS\system32\cmd.exe

```
Enter number of bits of operations: 8
Enter 8 bits of aV: 11111111
Enter vector delay: 3
aV: 11111111 AT 3
Enter 8 bits of bV: 00000000
Enter vector delay: 5
bV: 00000000 AT 5
Enter 1 bits of ci: 1
Enter vector delay: 7
ci: 1 AT 7

  aV: 11111111 AT 3
  bV: 00000000 AT 5
  ci: 1 AT 7
sumV: 00000000 AT 7
  co: 1 AT 7

Enter 0 to exit: 1
Enter number of bits of operations: 8
Enter 8 bits of aV: 00001111
Enter vector delay: 3
aV: 00001111 AT 3
Enter 8 bits of bV: 00000000
Enter vector delay: 5
bV: 00000000 AT 5
```

Logic delays that are only taking input delays into account



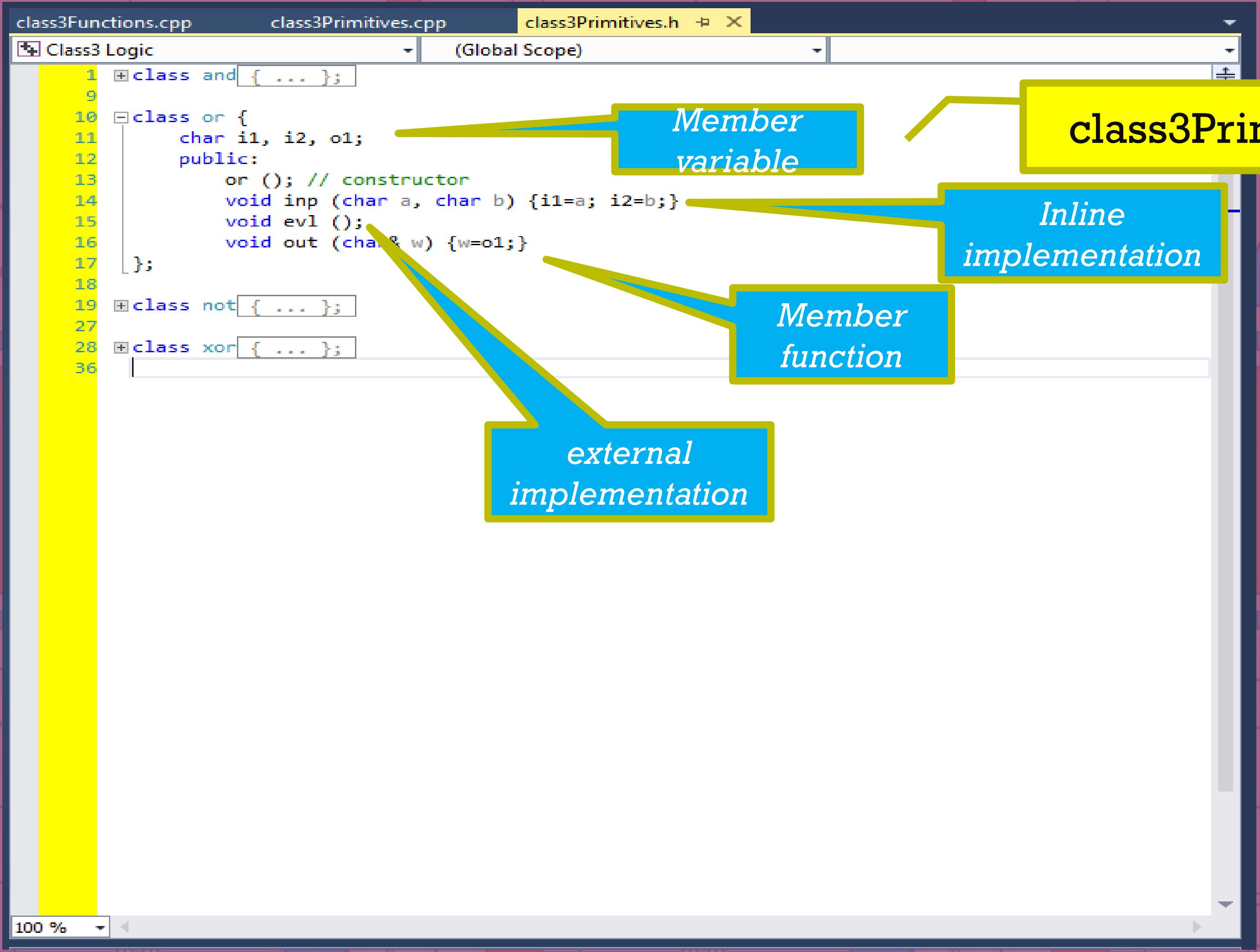
C:\WINDOWS\system32\cmd.exe

```
Enter number of bits of operations: 8
Enter 8 bits of aV: 10010011
Enter vector delay: 3
aV: 10010011 AT 3
Enter 8 bits of bV: 11110110
Enter vector delay: 5
bV: 11110110 AT 5
Enter 1 bits of ci: 1
Enter vector delay: 7
ci: 1 AT 7

  aV: 10010011 AT 3
  bV: 11110110 AT 5
  ci: 1 AT 7
sumV: 10001010 AT 7
  co: 1 AT 5

Enter 0 to exit: 0
Press any key to continue . . .
```

Logic Simulation with C/C++

- Procedural Languages for Hardware Modeling
- Types and Operators for Logic Modeling
- Basic Logic Simulation
  - Logic functions
  - Function overloading
  - Passing logic functions
  - Using default values
  - Building higher level structures
  - Handling 4-value logic
  - Logic vector
  - Sequential circuit modeling
  - Using pointers for logic vectors
- Enhanced logic simulation with timing
  - Using struct for timing and logic
  - Gates that handle timing
  - Utility functions
  - Timing in logic structures
  - Overloading logical operators
  - Using Boolean expressions
- More Functions for Wires and Gates
  - Gate classes
  - Carrier centric modeling
  - Compatible scalar and vector

Gate Classes



```
class and { ... };

class or {
    char i1, i2, o1;
    public:
        or (); // constructor
        void inp (char a, char b) {i1=a; i2=b;}
        void evl ();
        void out (char& w) {w=o1;}
};

class not { ... };

class xor { ... };
```

Member variable

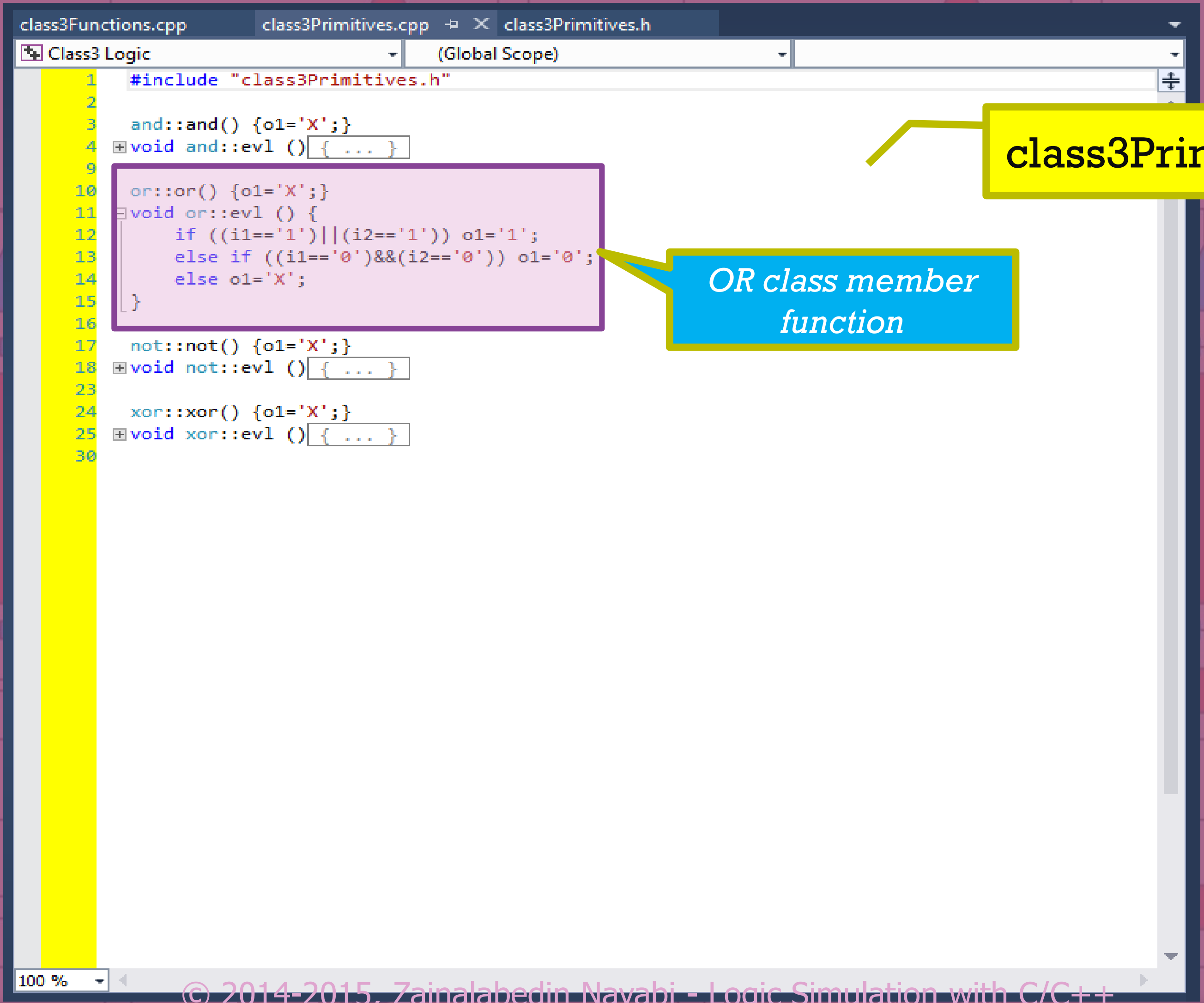
class3Primitives.h

Inline implementation

Member function

external implementation

Gate Classes

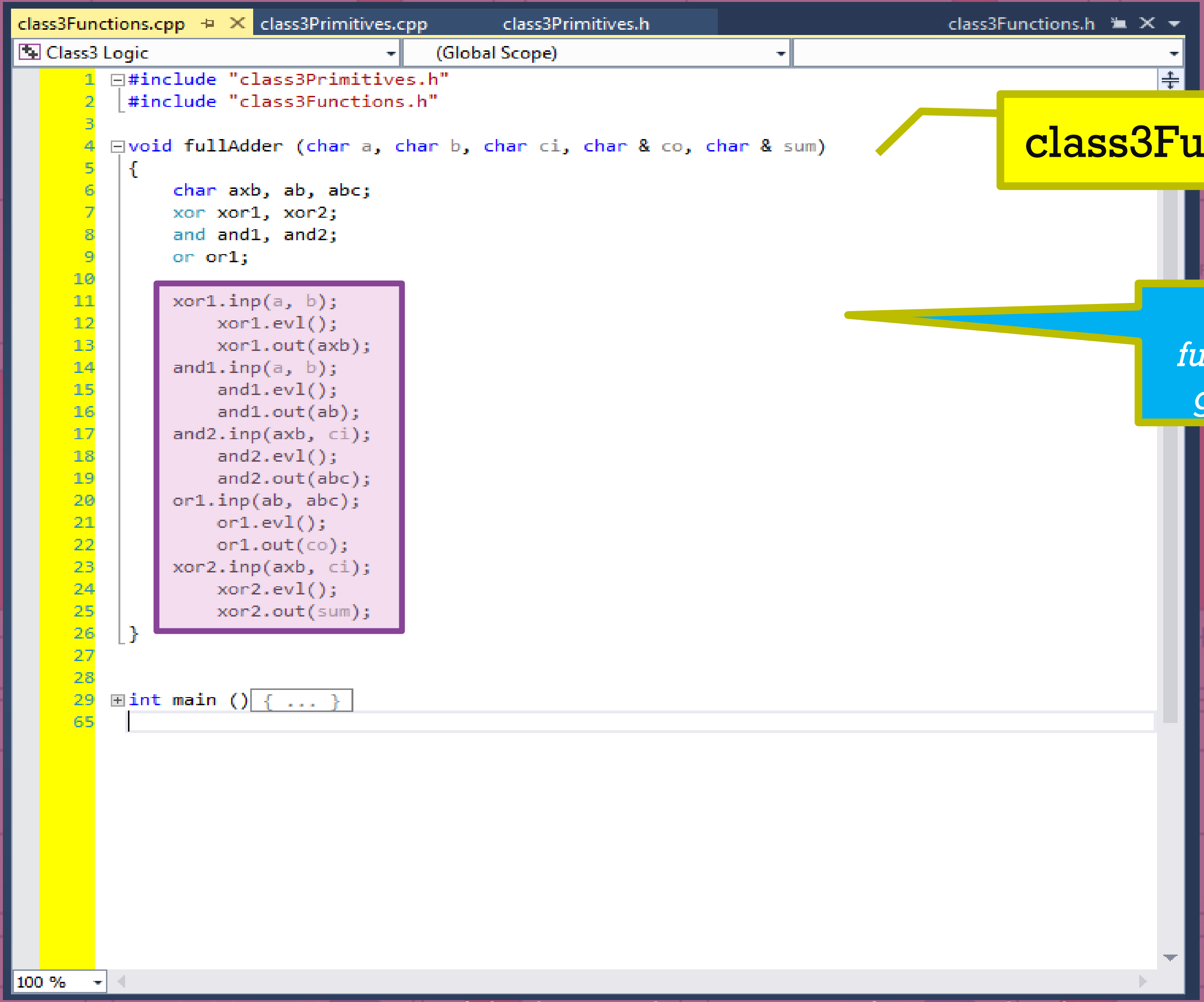


```
#include "class3Primitives.h"

and::and() {o1='X';}
void and::evl () { ... }

or::or() {o1='X';}
void or::evl () {
    if ((i1=='1')||(i2=='1')) o1='1';
    else if ((i1=='0')&&(i2=='0')) o1='0';
    else o1='X';
}

not::not() {o1='X';}
void not::evl () { ... }

xor::xor() {o1='X';}
void xor::evl () { ... }
```

class3Primitives.cpp

OR class member function

Gate Classes

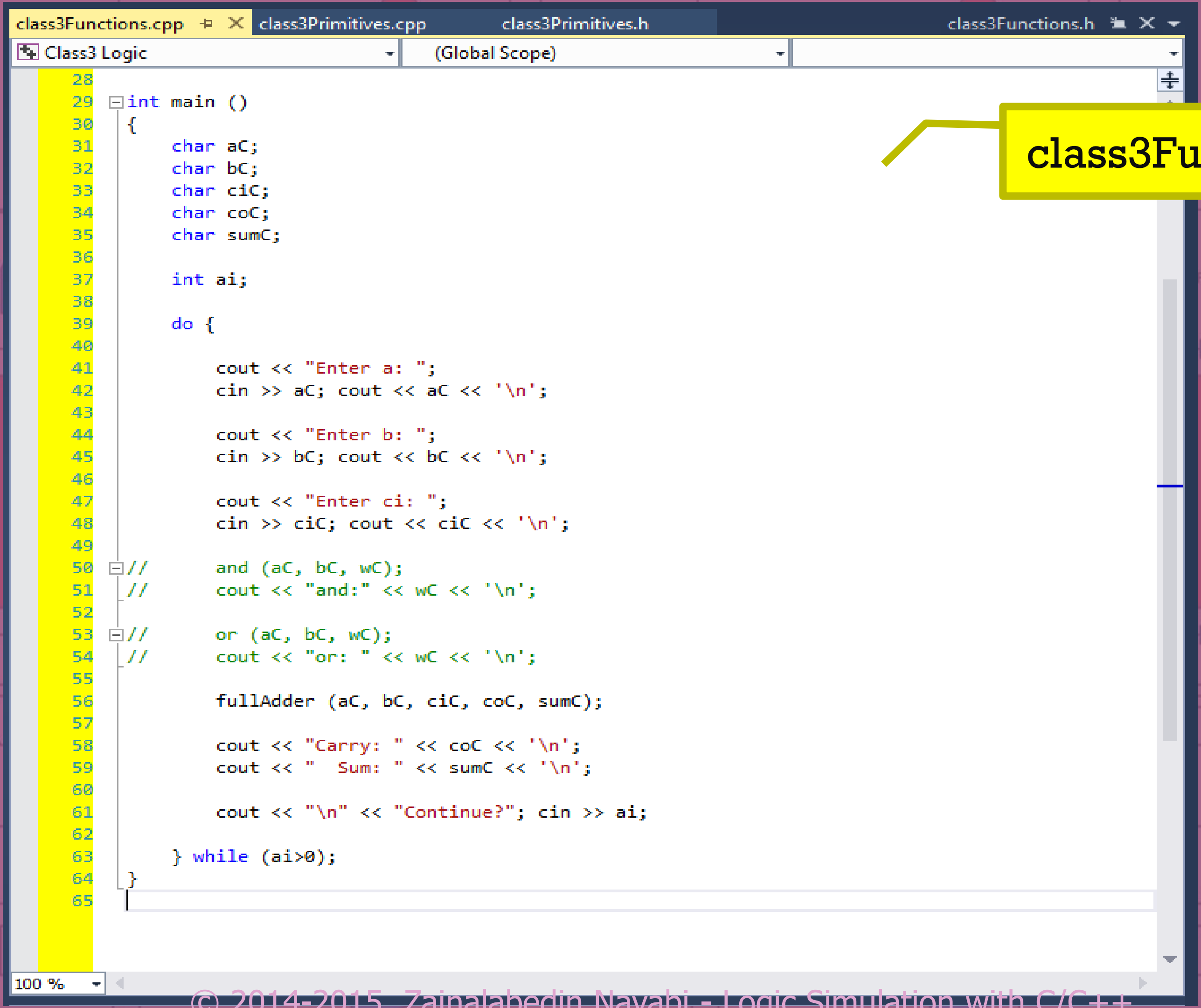


```
#include "class3Primitives.h"
#include "class3Functions.h"

void fullAdder (char a, char b, char ci, char & co, char & sum)
{
    char axb, ab, abc;
    xor xor1, xor2;
    and and1, and2;
    or or1;

    xor1.inp(a, b);
        xor1.evl();
        xor1.out(axb);
    and1.inp(a, b);
        and1.evl();
        and1.out(ab);
    and2.inp(axb, ci);
        and2.evl();
        and2.out(abc);
    or1.inp(ab, abc);
        or1.evl();
        or1.out(co);
    xor2.inp(axb, ci);
        xor2.evl();
        xor2.out(sum);
}


int main () { ... }
```

class3Functions.cpp

Full adder function using gate classes

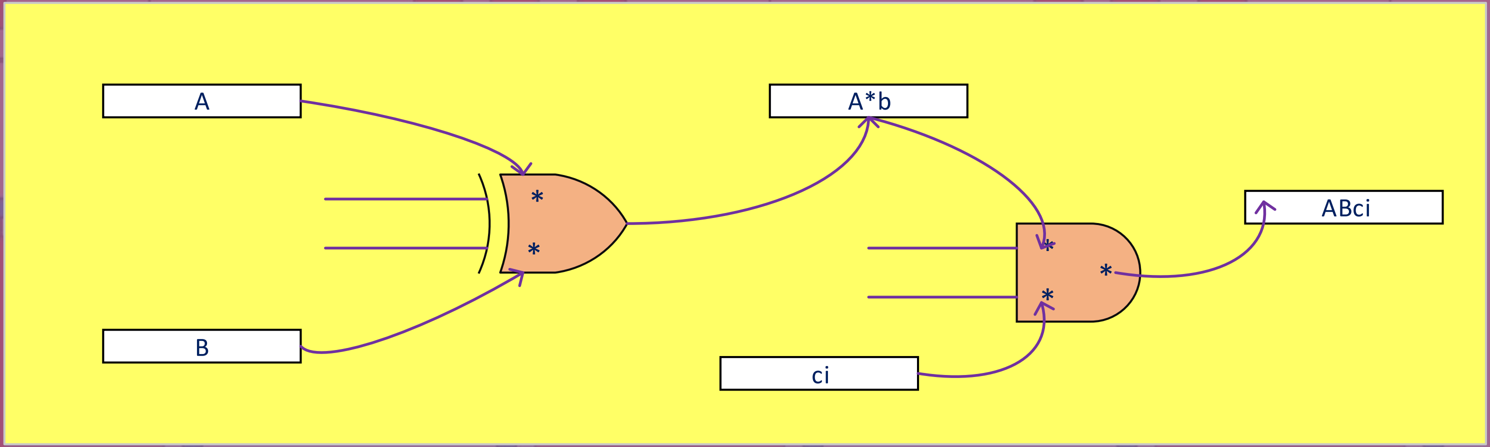
Gate Classes



```
int main ()
{
    char aC;
    char bC;
    char ciC;
    char coC;
    char sumC;

    int ai;

    do {

        cout << "Enter a: ";
        cin >> aC; cout << aC << '\n';

        cout << "Enter b: ";
        cin >> bC; cout << bC << '\n';

        cout << "Enter ci: ";
        cin >> ciC; cout << ciC << '\n';

//      and (aC, bC, wC);
//      cout << "and:" << wC << '\n';

//      or (aC, bC, wC);
//      cout << "or: " << wC << '\n';

        fullAdder (aC, bC, ciC, coC, sumC);

        cout << "Carry: " << coC << '\n';
        cout << "  Sum: " << sumC << '\n';

        cout << "\n" << "Continue?"; cin >> ai;

    } while (ai>0);
}
```

class3Functions.cpp

Carrier Centric Modeling



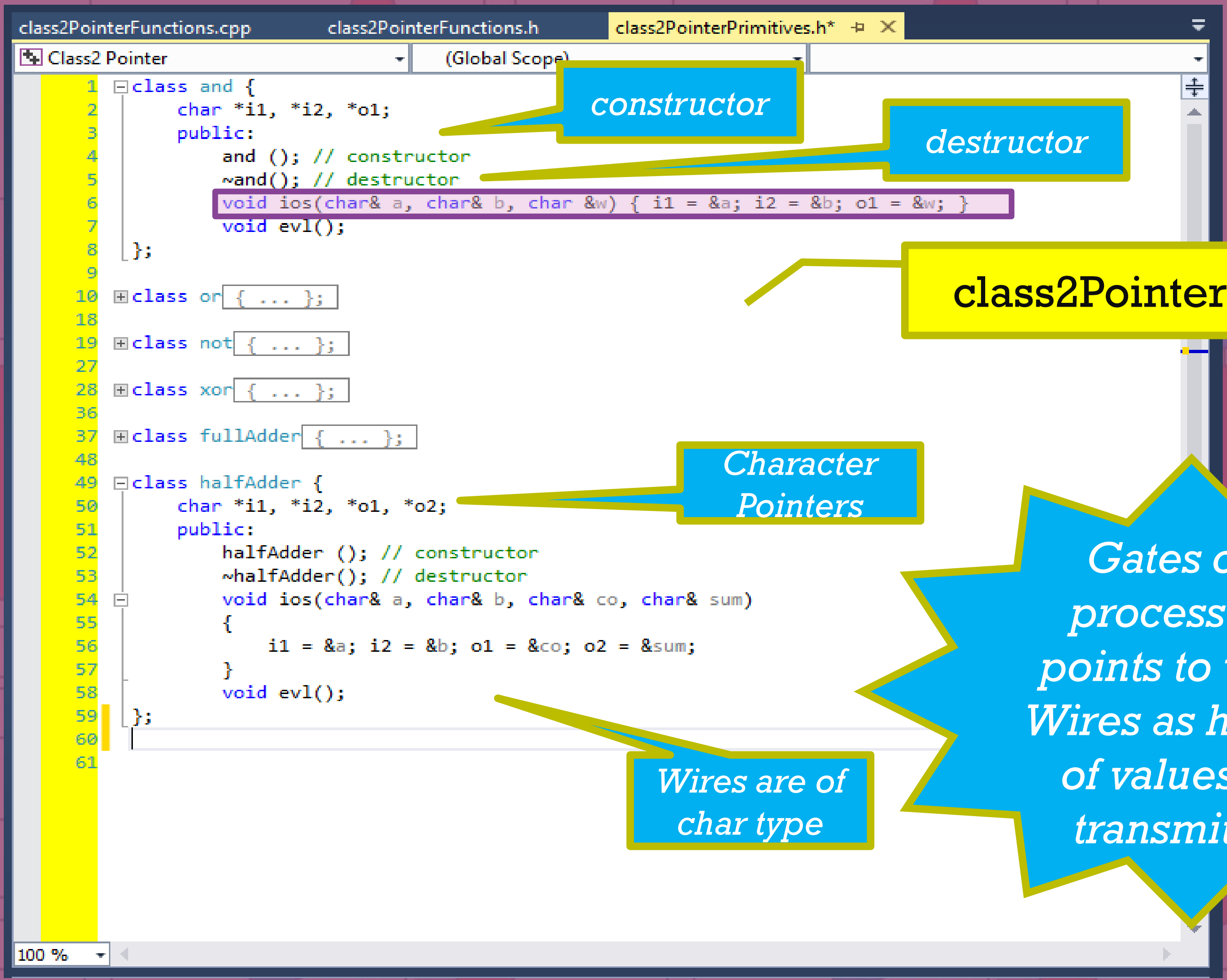
Pointer Based Logic Classes

- Classes do not hold values. Since the lines are just pointers, someone else has to declare them and allocate them.

- evl and out are combined and evl does both. Actually, since the outputs are pointers they will just be updated by evl. Every invocation of evl puts the internal output values on the evl return value.

- Destructor is introduced.

Pointer Based Logic Classes



```
class and {
    char *i1, *i2, *o1;
    public:
        and (); // constructor
        ~and(); // destructor
        void ios(char& a, char& b, char &w) { i1 = &a; i2 = &b; o1 = &w; }
        void evl();
};

class or { ... };

class not { ... };

class xor { ... };

class fullAdder { ... };

class halfAdder {
    char *i1, *i2, *o1, *o2;
    public:
        halfAdder (); // constructor
        ~halfAdder(); // destructor
        void ios(char& a, char& b, char& co, char& sum)
        {
            i1 = &a; i2 = &b; o1 = &co; o2 = &sum;
        }
        void evl();
};
```

constructor

destructor

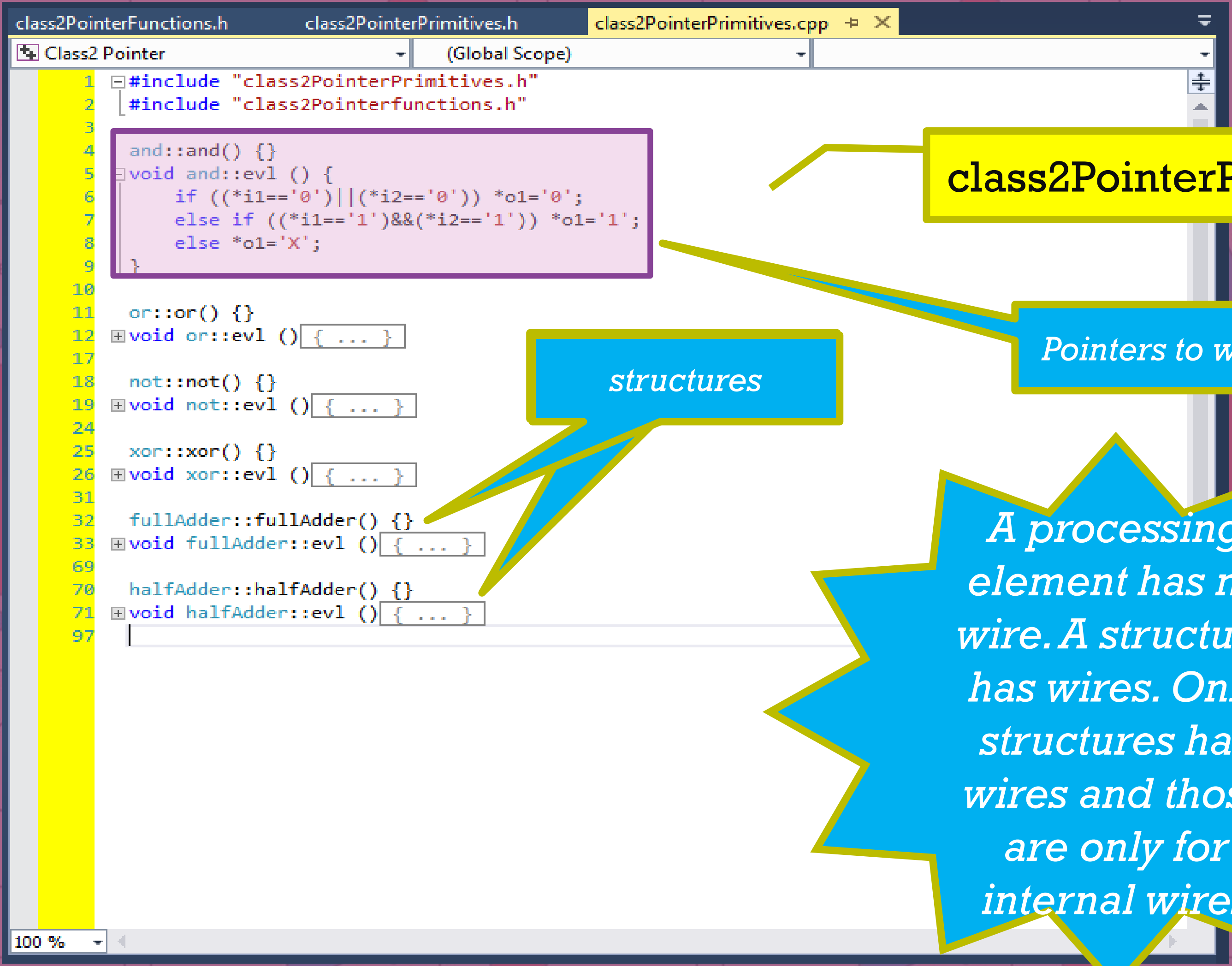
class2PointerPrimitives.h

Character Pointers

Wires are of char type

Gates only process and points to wires. Wires as holders of values and transmitters

Pointer Based Logic Classes



class2PointerFunctions.h class2PointerPrimitives.h class2PointerPrimitives.cpp

Class2 Pointer (Global Scope)

```
#include "class2PointerPrimitives.h"
#include "class2Pointerfunctions.h"

and::and() {}
void and::evl () {
    if ((*i1=='0')||(*i2=='0')) *o1='0';
    else if ((*i1=='1')&&(*i2=='1')) *o1='1';
    else *o1='X';
}

or::or() {}
void or::evl () { ... }

not::not() {}
void not::evl () { ... }

xor::xor() {}
void xor::evl () { ... }

fullAdder::fullAdder() {}
void fullAdder::evl () { ... }

halfAdder::halfAdder() {}
void halfAdder::evl () { ... }
```

100 %

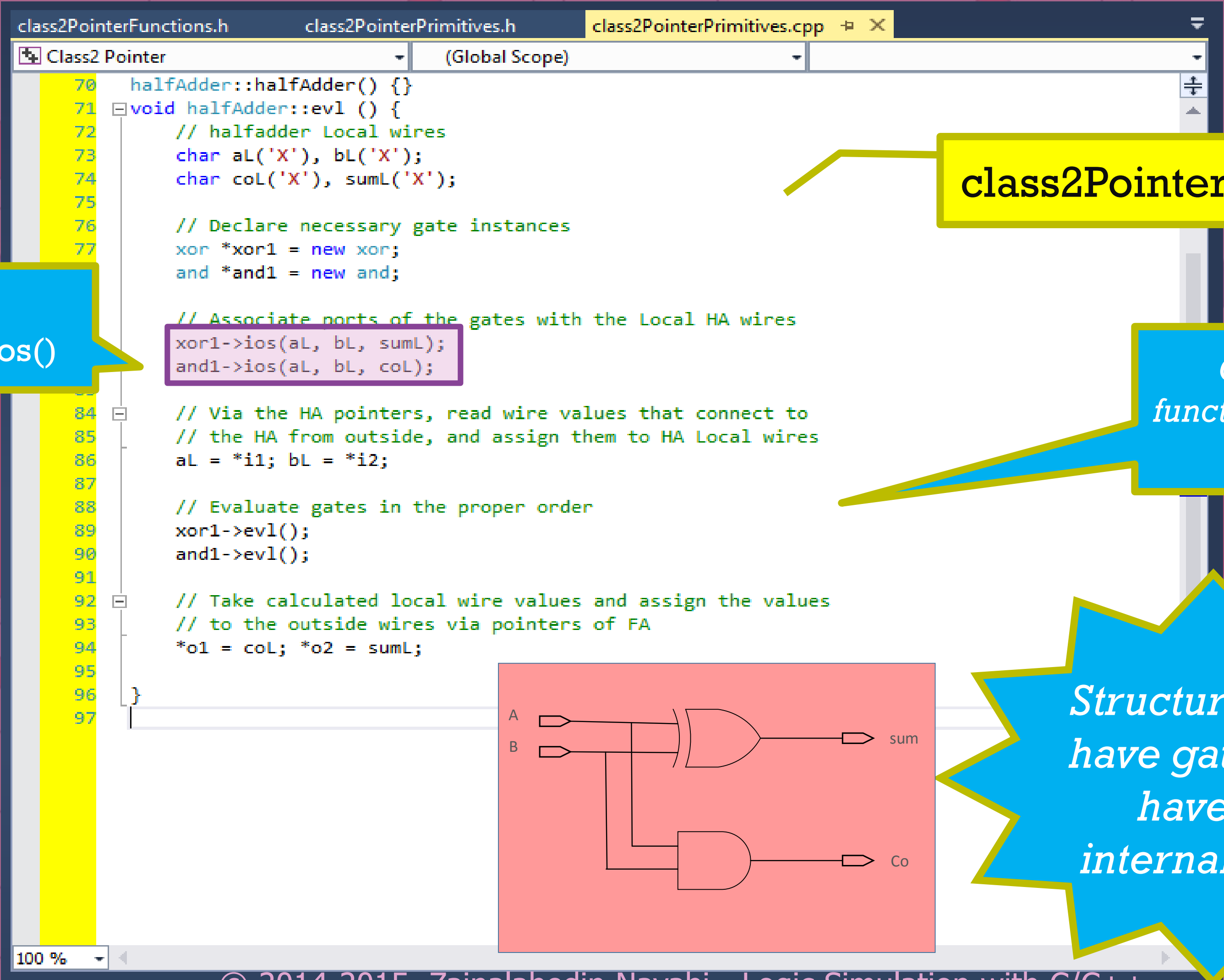
class2PointerPrimitives.cpp

Pointers to wires

structures

A processing element has no wire. A structure has wires. Only structures has wires and those are only for internal wires

Pointer Based Logic Classes

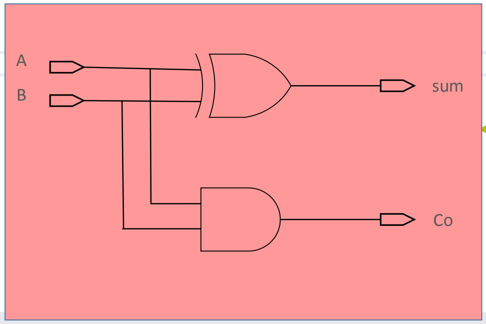


```
halfAdder::halfAdder() {}
void halfAdder::evl () {
    // halfadder Local wires
    char aL('X'), bL('X');
    char coL('X'), sumL('X');

    // Declare necessary gate instances
    xor *xor1 = new xor;
    and *and1 = new and;

    // Associate ports of the gates with the Local HA wires
    xor1->ios(aL, bL, sumL);
    and1->ios(aL, bL, coL);

    // Via the HA pointers, read wire values that connect to
    // the HA from outside, and assign them to HA Local wires
    aL = *i1; bL = *i2;

    // Evaluate gates in the proper order
    xor1->evl();
    and1->evl();

    // Take calculated local wire values and assign the values
    // to the outside wires via pointers of FA
    *o1 = coL; *o2 = sumL;

}
```

class2PointerPrimitives.cpp

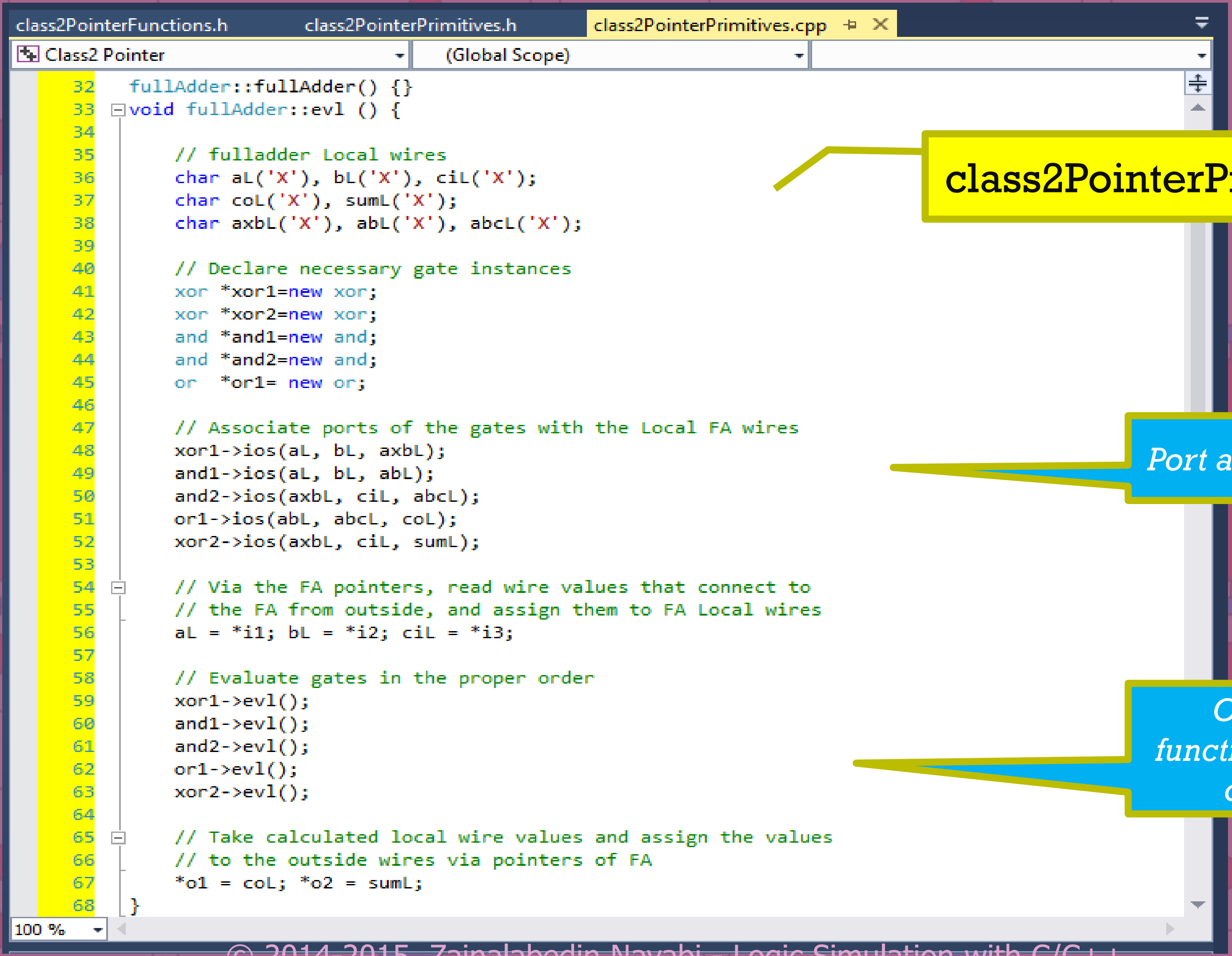
Or
*xor1.ios()

Order evl() functions according to logic



Structures also have gates that have no internal wires

Pointer Based Logic Classes



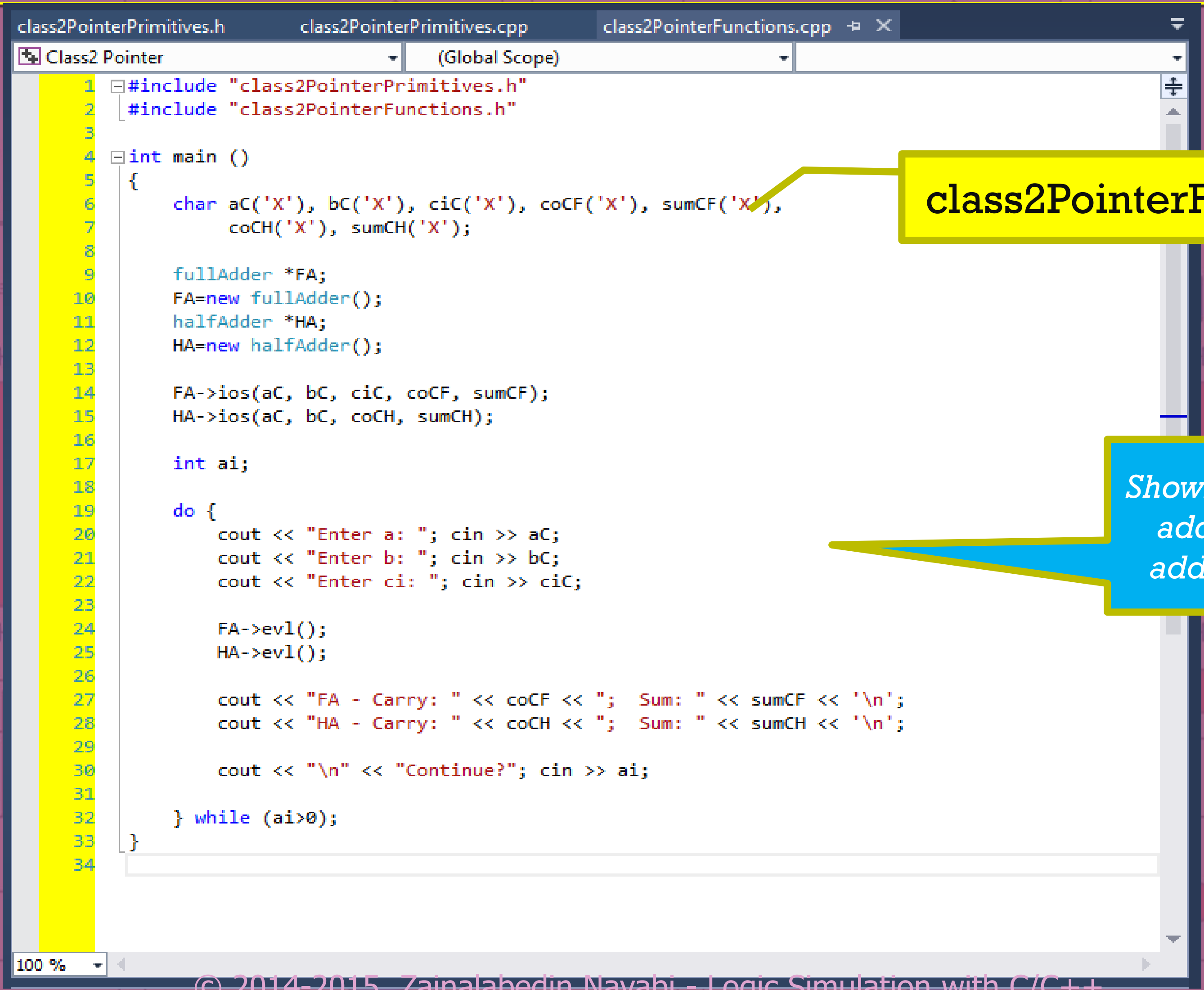
```cpp
fullAdder::fullAdder() {}
void fullAdder::evl () {

    // fulladder Local wires
    char aL('X'), bL('X'), ciL('X');
    char coL('X'), sumL('X');
    char axbL('X'), abL('X'), abcL('X');

    // Declare necessary gate instances
    xor *xor1=new xor;
    xor *xor2=new xor;
    and *and1=new and;
    and *and2=new and;
    or  *or1= new or;

    // Associate ports of the gates with the Local FA wires
    xor1->ios(aL, bL, axbL);
    and1->ios(aL, bL, abL);
    and2->ios(axbL, ciL, abcL);
    or1->ios(abL, abcL, coL);
    xor2->ios(axbL, ciL, sumL);

    // Via the FA pointers, read wire values that connect to
    // the FA from outside, and assign them to FA Local wires
    aL = *i1; bL = *i2; ciL = *i3;

    // Evaluate gates in the proper order
    xor1->evl();
    and1->evl();
    and2->evl();
    or1->evl();
    xor2->evl();

    // Take calculated local wire values and assign the values
    // to the outside wires via pointers of FA
    *o1 = coL; *o2 = sumL;
}
```

class2PointerPrimitives.cpp

Port association

Only evl() functions must be ordered

Pointer Based Logic Classes

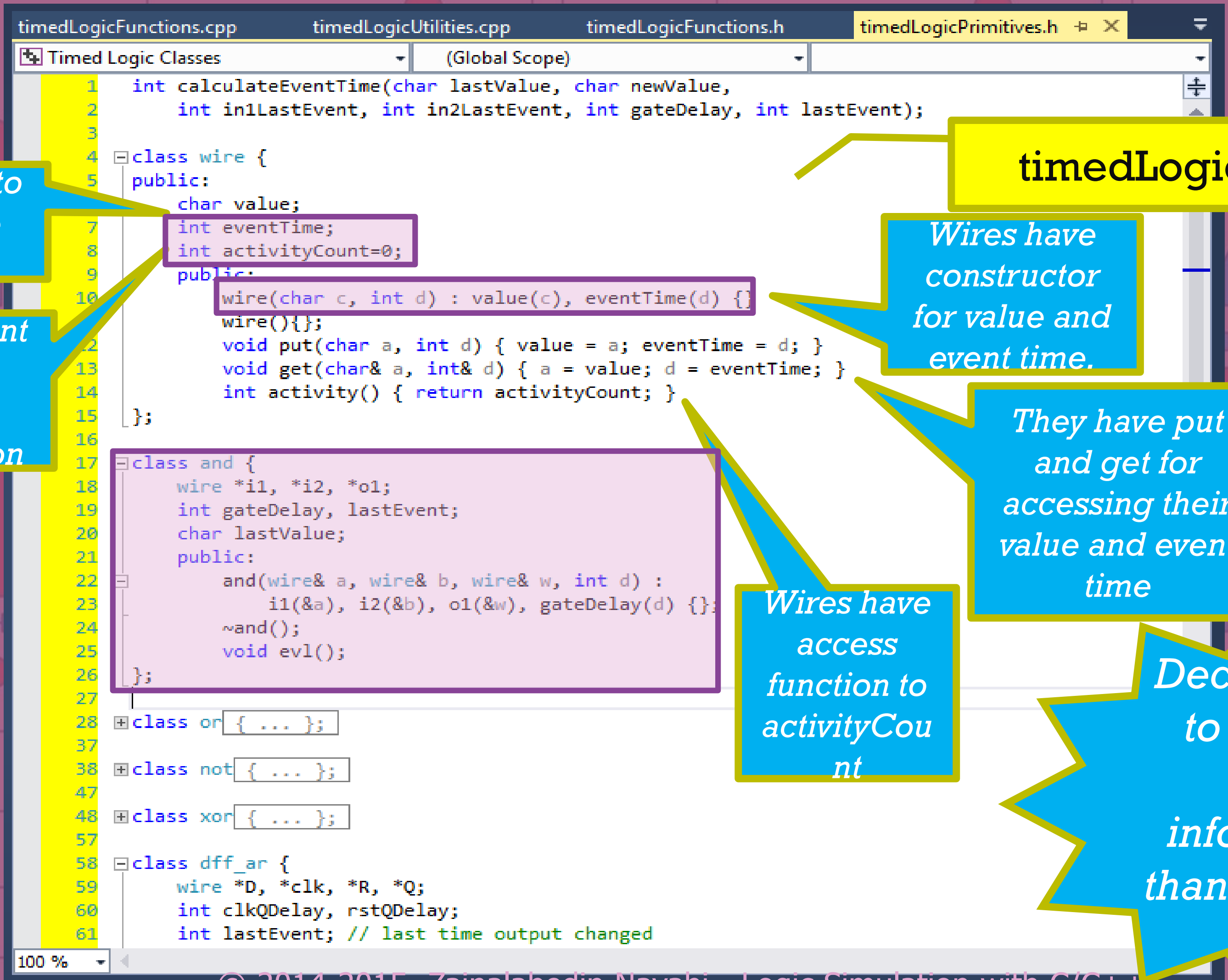

class2PointerPrimitives.h
class2PointerPrimitives.cpp
class2PointerFunctions.cpp
Class2 Pointer
(Global Scope)
100 %

```
#include "class2PointerPrimitives.h"
#include "class2PointerFunctions.h"

int main ()
{
    char aC('X'), bC('X'), ciC('X'), coCF('X'), sumCF('X'),
         coCH('X'), sumCH('X');

    fullAdder *FA;
    FA=new fullAdder();
    halfAdder *HA;
    HA=new halfAdder();

    FA->ios(aC, bC, ciC, coCF, sumCF);
    HA->ios(aC, bC, coCH, sumCH);

    int ai;

    do {
        cout << "Enter a: "; cin >> aC;
        cout << "Enter b: "; cin >> bC;
        cout << "Enter ci: "; cin >> ciC;

        FA->evl();
        HA->evl();

        cout << "FA - Carry: " << coCF << ";  Sum: " << sumCF << '\n';
        cout << "HA - Carry: " << coCH << ";  Sum: " << sumCH << '\n';

        cout << "\n" << "Continue?"; cin >> ai;

    } while (ai>0);
}
```

class2PointerFunctions.cpp

Shows main for full adder and half adder functions

Gate Classes with Power and Timing Calculation



```
int calculateEventTime(char lastValue, char newValue,
    int in1LastEvent, int in2LastEvent, int gateDelay, int lastEvent);

class wire {
public:
    char value;
    int eventTime;
    int activityCount=0;
    public:
        wire(char c, int d) : value(c), eventTime(d) {}
        wire(){};
        void put(char a, int d) { value = a; eventTime = d; }
        void get(char& a, int& d) { a = value; d = eventTime; }
        int activity() { return activityCount; }
};

class and {
    wire *i1, *i2, *o1;
    int gateDelay, lastEvent;
    char lastValue;
    public:
        and(wire& a, wire& b, wire& w, int d) :
            i1(&a), i2(&b), o1(&w), gateDelay(d) {};
        ~and();
        void evl();
};

class or { ... };

class not { ... };

class xor { ... };

class dff_ar {
    wire *D, *clk, *R, *Q;
    int clkQDelay, rstQDelay;
    int lastEvent; // last time output changed
```

timedLogicPrimitives.h

EventTime to propagate delay

ActivityCount to carry power consumption

Wires have constructor for value and event time.

They have put and get for accessing their value and event time

Wires have access function to activityCount

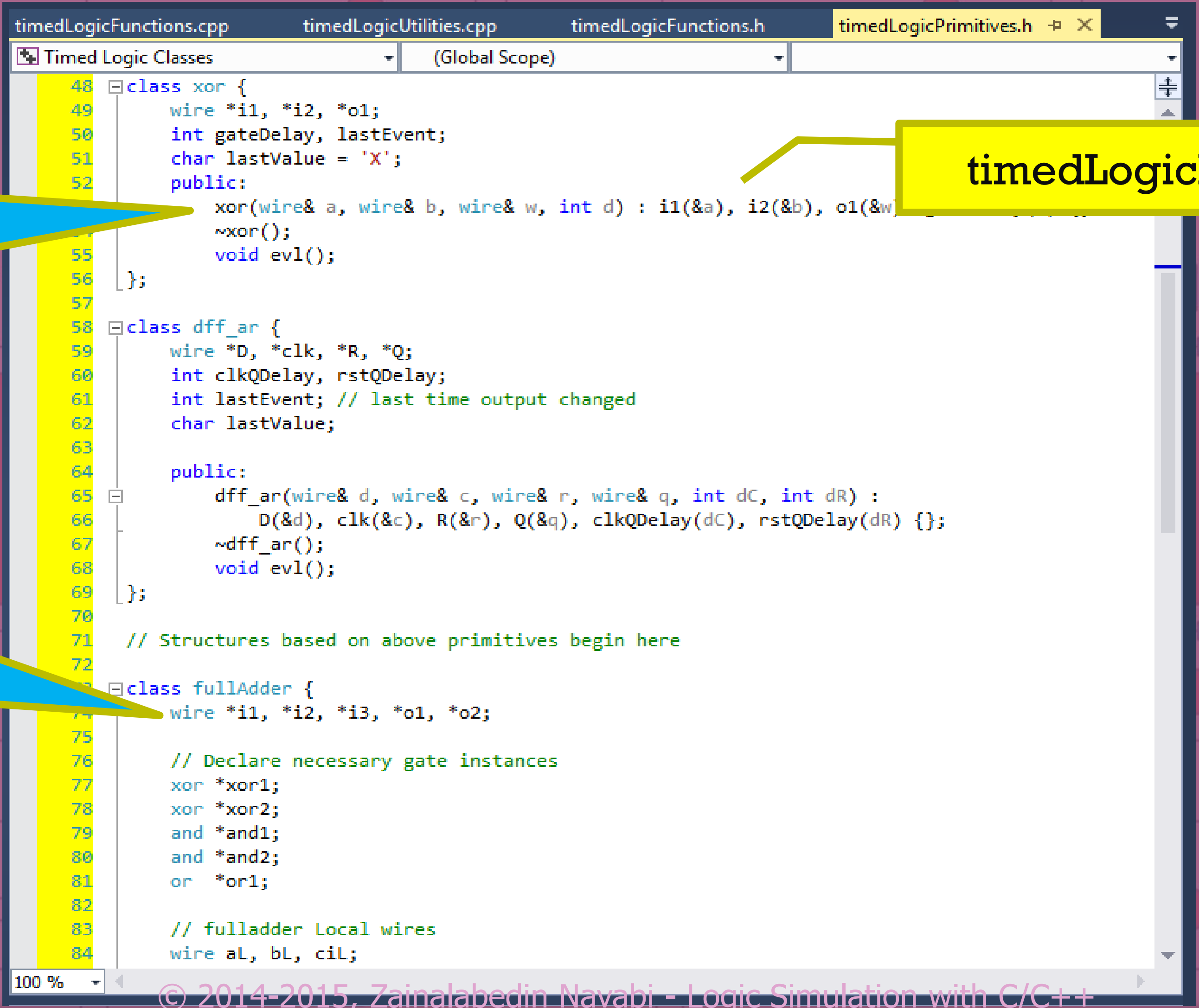
Declare wire to contain more information than just logic value

Gate Classes with Power and Timing Calculation

Xor constructor just ties port pointers to wires

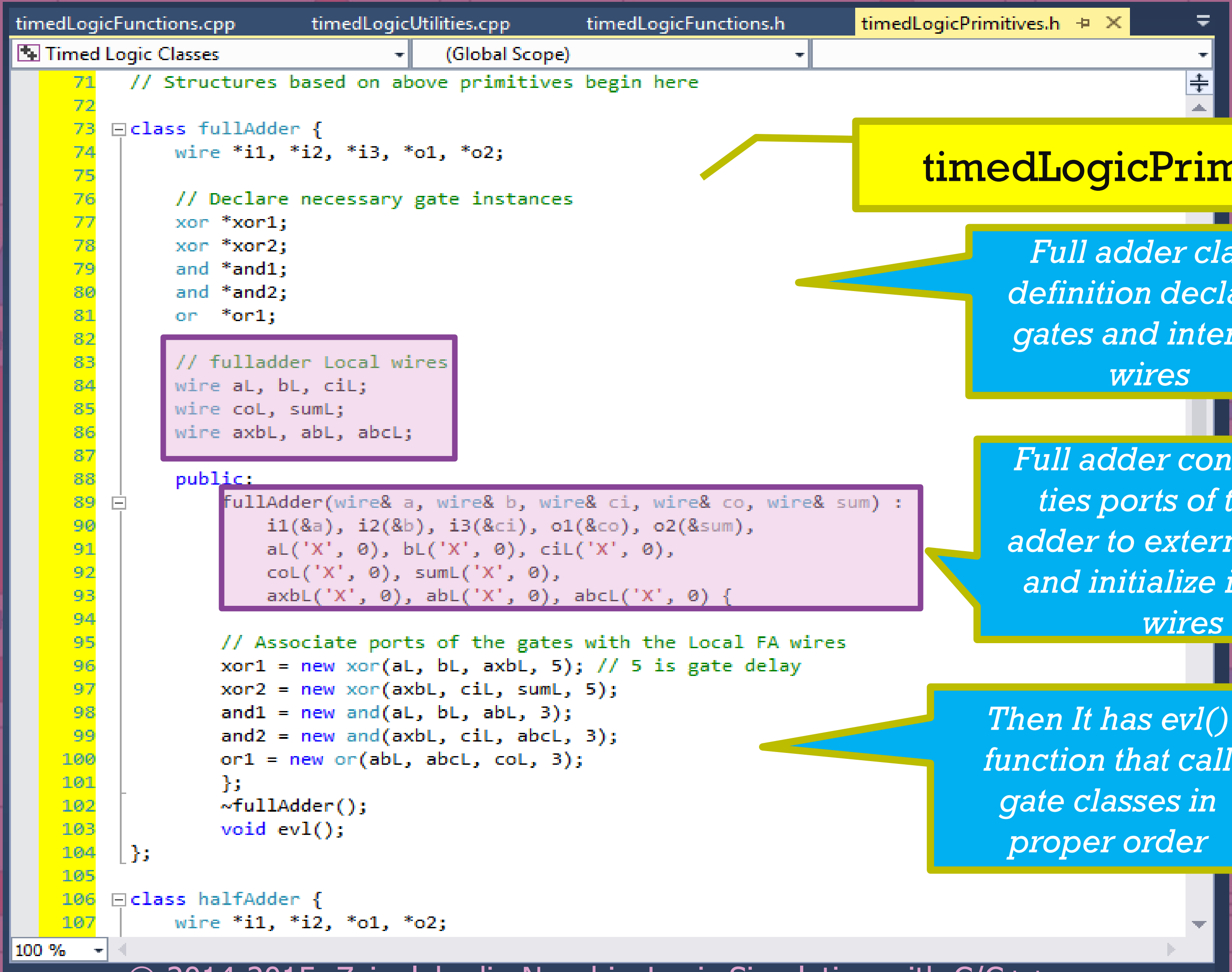
timedLogicPrimitives.h

O1 is a pointer. This pointer is tied to pointer of w



```
class xor {
    wire *i1, *i2, *o1;
    int gateDelay, lastEvent;
    char lastValue = 'X';
    public:
        xor(wire& a, wire& b, wire& w, int d) : i1(&a), i2(&b), o1(&w
        ~xor();
        void evl();
};

class dff_ar {
    wire *D, *clk, *R, *Q;
    int clkQDelay, rstQDelay;
    int lastEvent; // last time output changed
    char lastValue;

    public:
        dff_ar(wire& d, wire& c, wire& r, wire& q, int dC, int dR) :
            D(&d), clk(&c), R(&r), Q(&q), clkQDelay(dC), rstQDelay(dR) {};
        ~dff_ar();
        void evl();
};

// Structures based on above primitives begin here

class fullAdder {
    wire *i1, *i2, *i3, *o1, *o2;

    // Declare necessary gate instances
    xor *xor1;
    xor *xor2;
    and *and1;
    and *and2;
    or  *or1;

    // fulladder Local wires
    wire aL, bL, ciL;
```

Gate Classes with Power and Timing Calculation



```
// Structures based on above primitives begin here

class fullAdder {
    wire *i1, *i2, *i3, *o1, *o2;

    // Declare necessary gate instances
    xor *xor1;
    xor *xor2;
    and *and1;
    and *and2;
    or  *or1;

    // fulladder Local wires
    wire aL, bL, ciL;
    wire coL, sumL;
    wire axbL, abL, abcL;

    public:
        fullAdder(wire& a, wire& b, wire& ci, wire& co, wire& sum) :
            i1(&a), i2(&b), i3(&ci), o1(&co), o2(&sum),
            aL('X', 0), bL('X', 0), ciL('X', 0),
            coL('X', 0), sumL('X', 0),
            axbL('X', 0), abL('X', 0), abcL('X', 0) {

        // Associate ports of the gates with the Local FA wires
        xor1 = new xor(aL, bL, axbL, 5); // 5 is gate delay
        xor2 = new xor(axbL, ciL, sumL, 5);
        and1 = new and(aL, bL, abL, 3);
        and2 = new and(axbL, ciL, abcL, 3);
        or1 = new or(abL, abcL, coL, 3);
        };
        ~fullAdder();
        void evl();
};

class halfAdder {
    wire *i1, *i2, *o1, *o2;
```

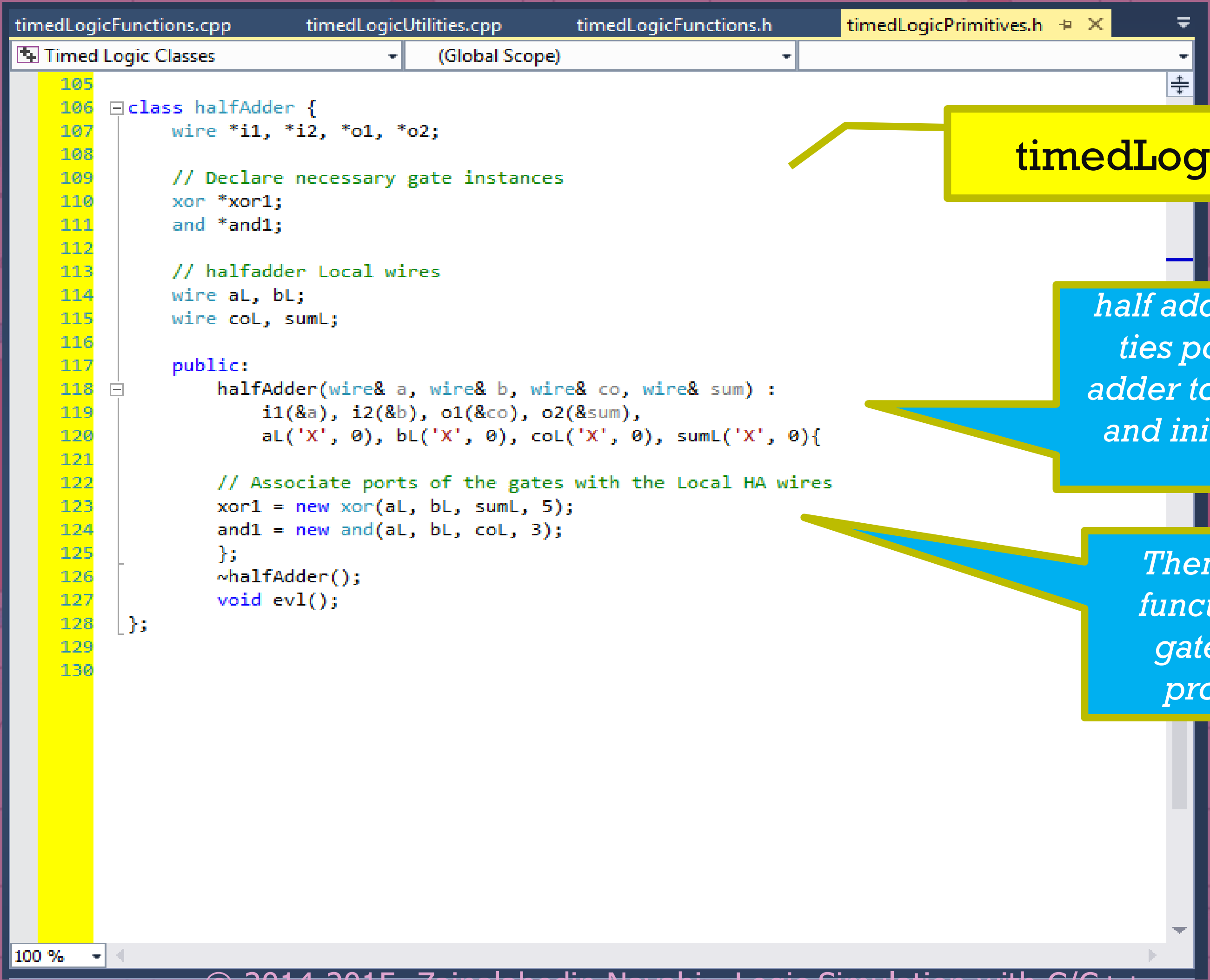
timedLogicPrimitives.h

Full adder class definition declares gates and internal wires

Full adder constructor ties ports of the full adder to external wires and initialize internal wires

Then It has evl() function that call gate classes in proper order

Gate Classes with Power and Timing Calculation



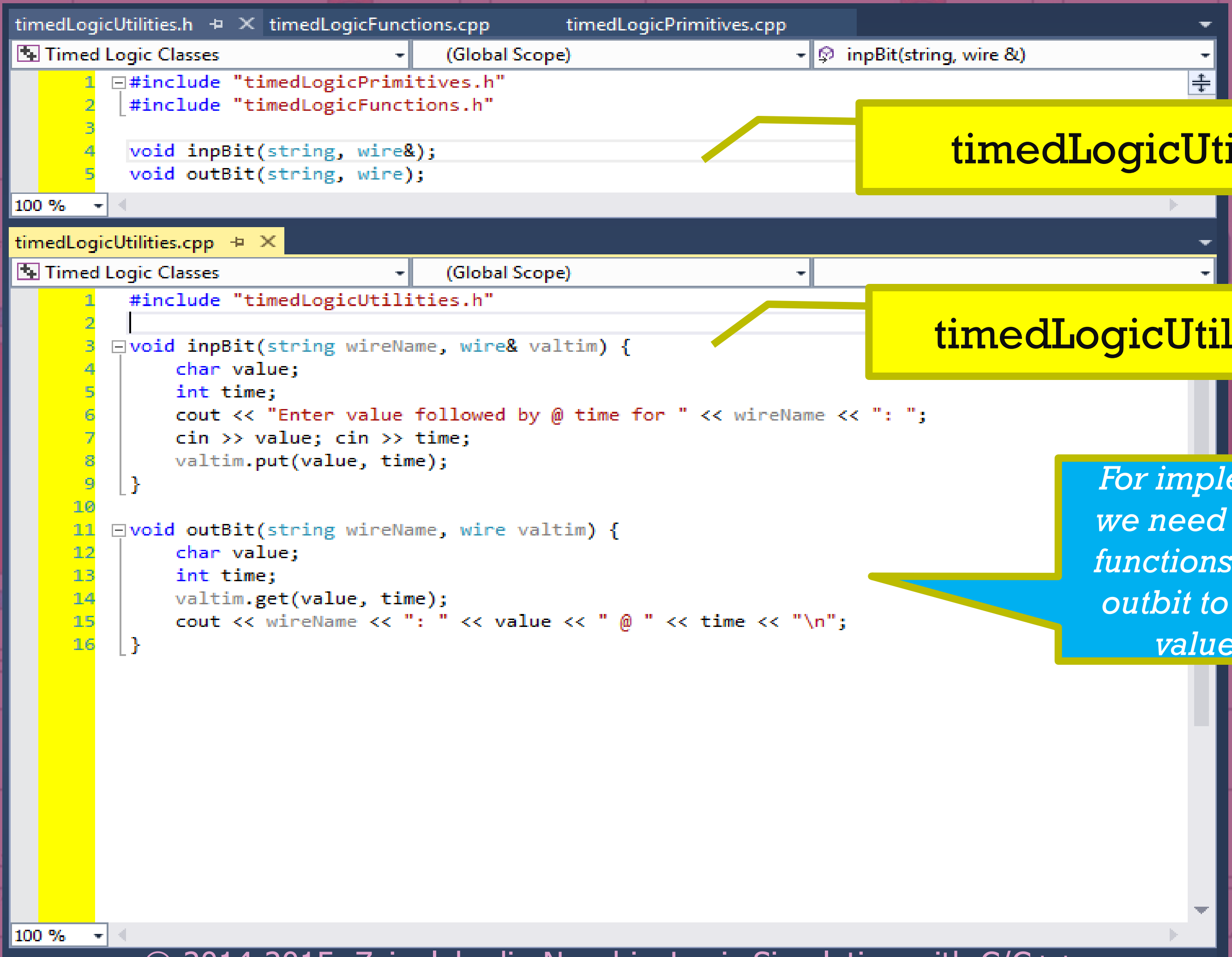
```
class halfAdder {
    wire *i1, *i2, *o1, *o2;

    // Declare necessary gate instances
    xor *xor1;
    and *and1;

    // halfadder Local wires
    wire aL, bL;
    wire coL, sumL;

    public:
        halfAdder(wire& a, wire& b, wire& co, wire& sum) :
            i1(&a), i2(&b), o1(&co), o2(&sum),
            aL('X', 0), bL('X', 0), coL('X', 0), sumL('X', 0){

        // Associate ports of the gates with the Local HA wires
        xor1 = new xor(aL, bL, sumL, 5);
        and1 = new and(aL, bL, coL, 3);
        };
        ~halfAdder();
        void evl();
};
```

timedLogicPrimitives.h

half adder constructor ties ports of the full adder to external wires and initialize internal wires

Then It has evl() function that call gate classes in proper order

Gate Classes with Power and Timing Calculation



```
#include "timedLogicPrimitives.h"
#include "timedLogicFunctions.h"

void inpBit(string, wire&);
void outBit(string, wire);
```

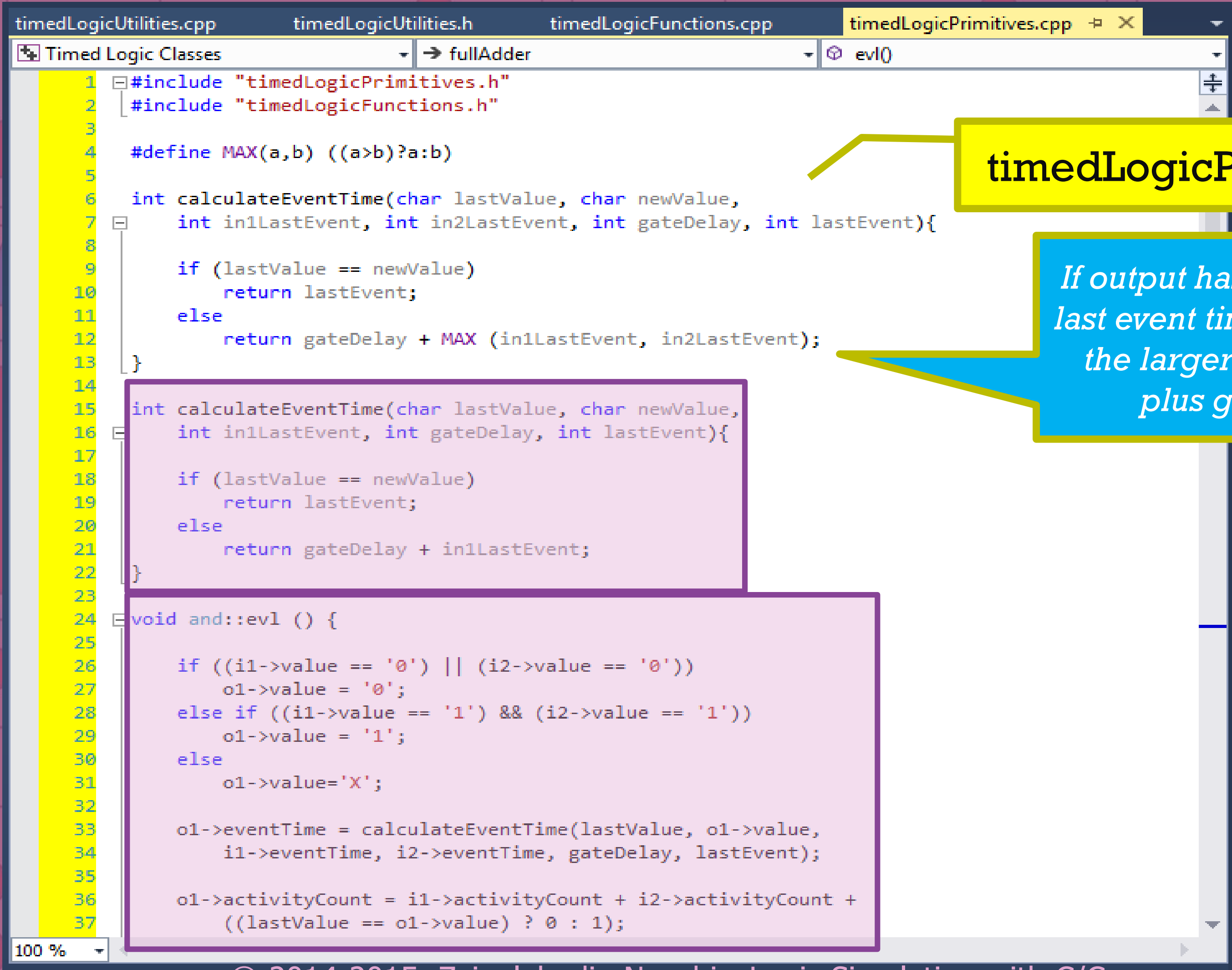
timedLogicUtilities.h

```
#include "timedLogicUtilities.h"

void inpBit(string wireName, wire& valtim) {
    char value;
    int time;
    cout << "Enter value followed by @ time for " << wireName << ": ";
    cin >> value; cin >> time;
    valtim.put(value, time);
}

void outBit(string wireName, wire valtim) {
    char value;
    int time;
    valtim.get(value, time);
    cout << wireName << ": " << value << " @ " << time << "\n";
}
```

timedLogicUtilities.cpp

For implementing this we need several utility functions. For inbit and outbit to get time and value for wires

Gate Classes with Power and Timing Calculation



timedLogicUtilities.cpp timedLogicUtilities.h timedLogicFunctions.cpp timedLogicPrimitives.cpp

Timed Logic Classes | fullAdder | evl()

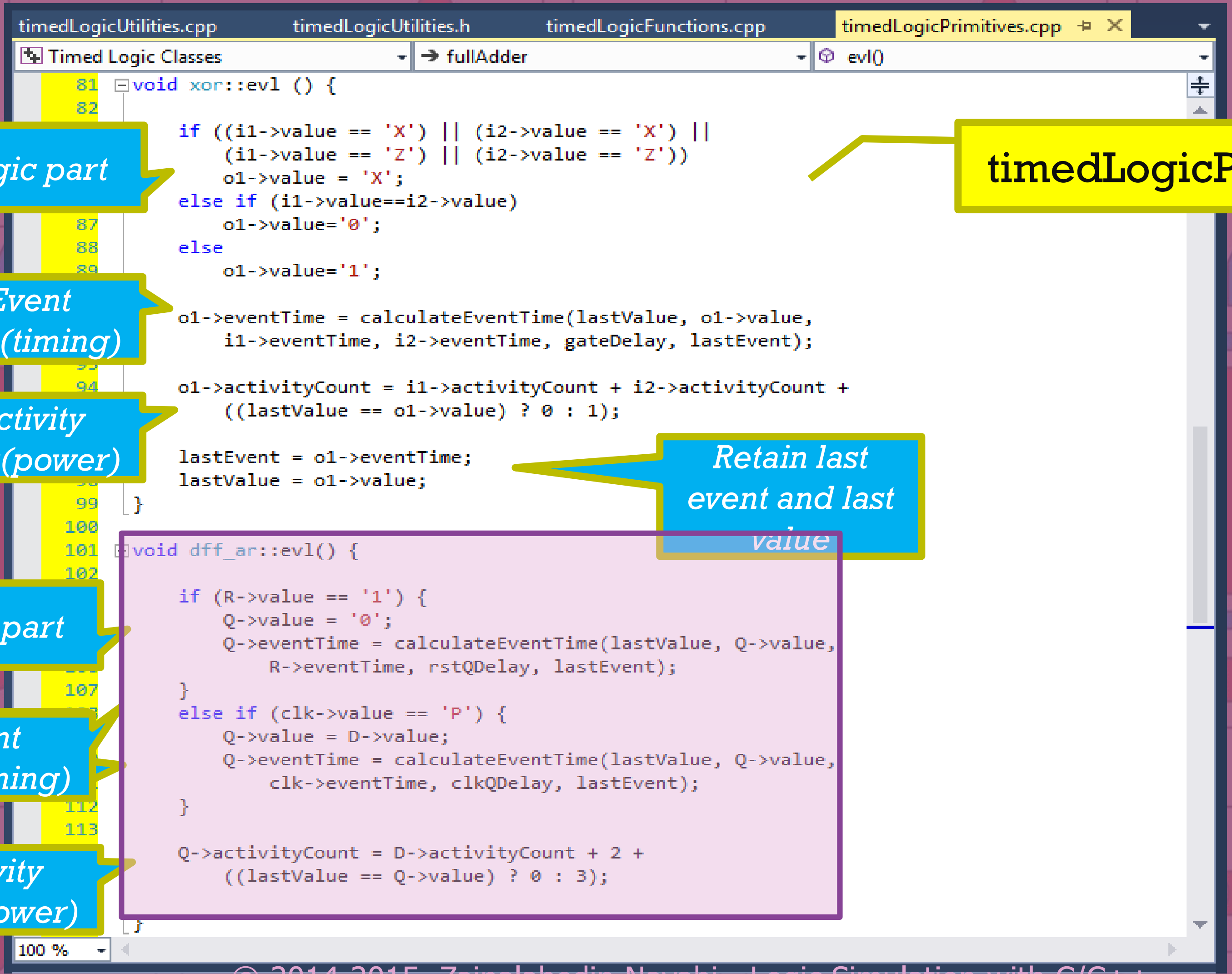
```
#include "timedLogicPrimitives.h"
#include "timedLogicFunctions.h"

#define MAX(a,b) ((a>b)?a:b)

int calculateEventTime(char lastValue, char newValue,
    int in1LastEvent, int in2LastEvent, int gateDelay, int lastEvent){

    if (lastValue == newValue)
        return lastEvent;
    else
        return gateDelay + MAX (in1LastEvent, in2LastEvent);
}

int calculateEventTime(char lastValue, char newValue,
    int in1LastEvent, int gateDelay, int lastEvent){

    if (lastValue == newValue)
        return lastEvent;
    else
        return gateDelay + in1LastEvent;
}

void and::evl () {

    if ((i1->value == '0') || (i2->value == '0'))
        o1->value = '0';
    else if ((i1->value == '1') && (i2->value == '1'))
        o1->value = '1';
    else
        o1->value='X';

    o1->eventTime = calculateEventTime(lastValue, o1->value,
        i1->eventTime, i2->eventTime, gateDelay, lastEvent);

    o1->activityCount = i1->activityCount + i2->activityCount +
        ((lastValue == o1->value) ? 0 : 1);
```

100 %

timedLogicPrimitives.cpp

If output has changed, the last event time on output is the larger of the inputs plus gate delay

Gate Classes with Power and Timing Calculation



```
void xor::evl () {

    if ((i1->value == 'X') || (i2->value == 'X') ||
        (i1->value == 'Z') || (i2->value == 'Z'))
        o1->value = 'X';
    else if (i1->value==i2->value)
        o1->value='0';
    else
        o1->value='1';

    o1->eventTime = calculateEventTime(lastValue, o1->value,
        i1->eventTime, i2->eventTime, gateDelay, lastEvent);

    o1->activityCount = i1->activityCount + i2->activityCount +
        ((lastValue == o1->value) ? 0 : 1);

    lastEvent = o1->eventTime;
    lastValue = o1->value;
}

void dff_ar::evl() {

    if (R->value == '1') {
        Q->value = '0';
        Q->eventTime = calculateEventTime(lastValue, Q->value,
            R->eventTime, rstQDelay, lastEvent);
    }
    else if (clk->value == 'P') {
        Q->value = D->value;
        Q->eventTime = calculateEventTime(lastValue, Q->value,
            clk->eventTime, clkQDelay, lastEvent);
    }

    Q->activityCount = D->activityCount + 2 +
        ((lastValue == Q->value) ? 0 : 3);
}
```

timedLogicPrimitives.cpp

Logic part

Event part(timing)

Activity part(power)

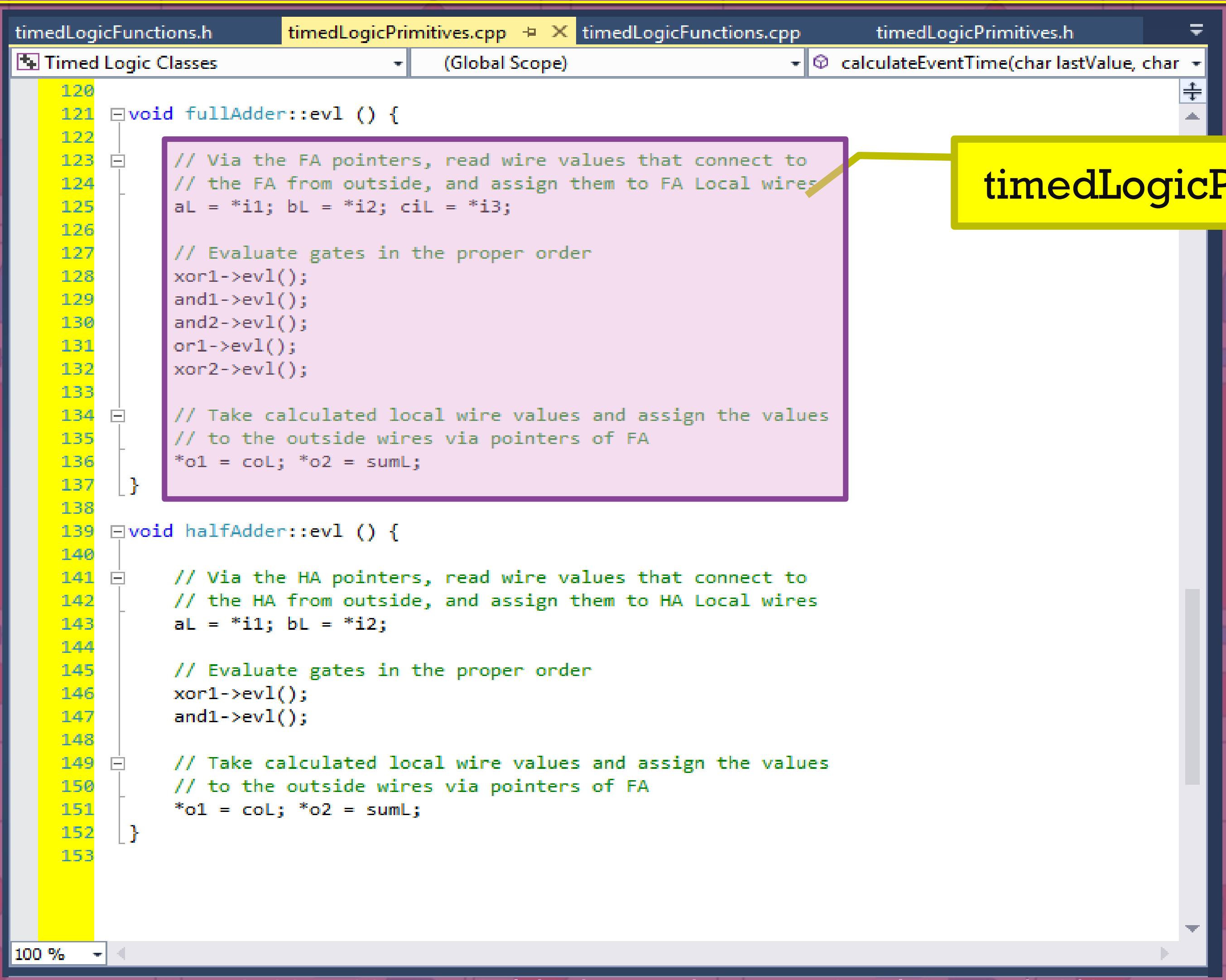
Retain last event and last value

Logic part

Event part(timing)

Activity part(power)

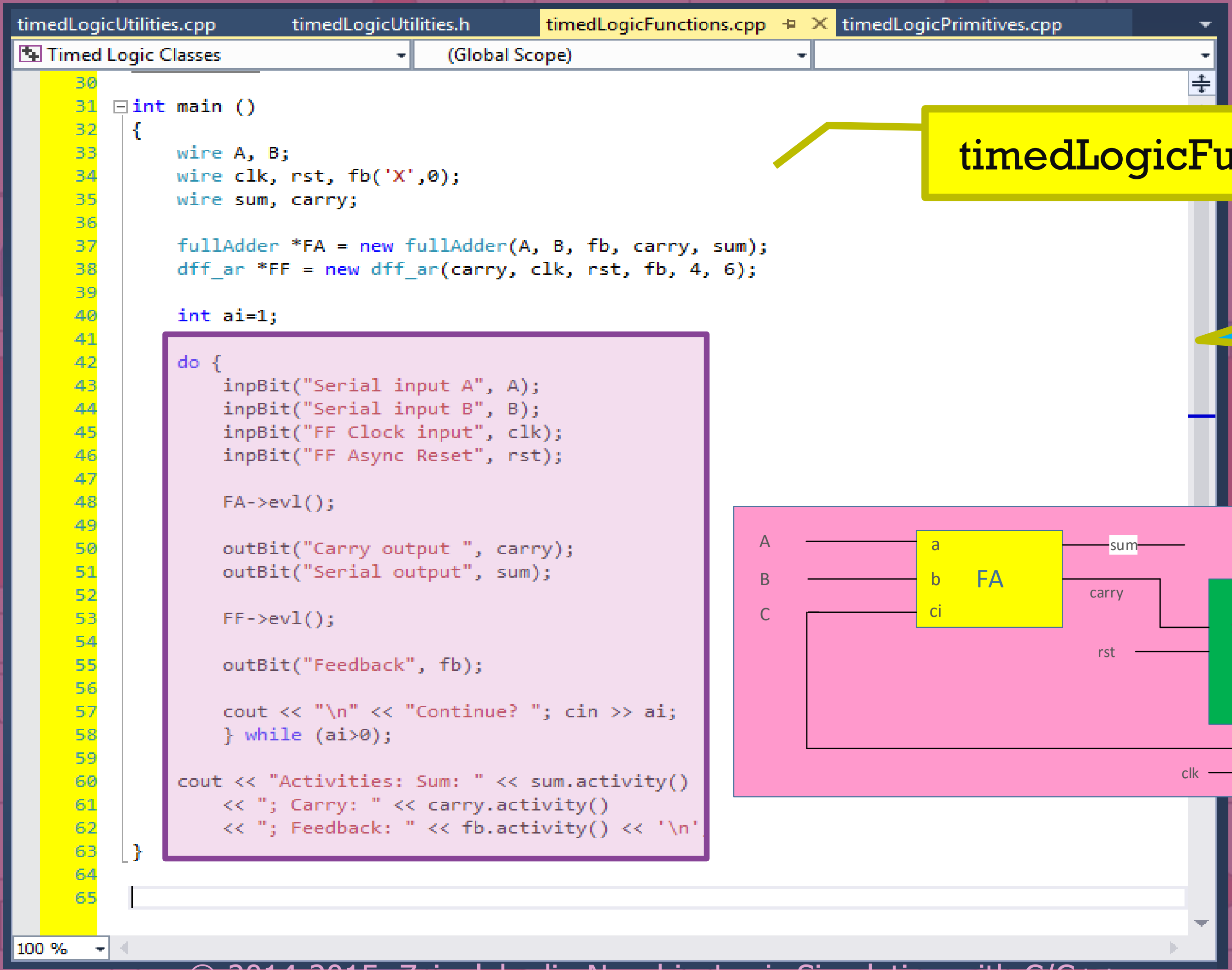
Gate Classes with Power and Timing Calculation



```
void fullAdder::evl () {

    // Via the FA pointers, read wire values that connect to
    // the FA from outside, and assign them to FA Local wires
    aL = *i1; bL = *i2; ciL = *i3;

    // Evaluate gates in the proper order
    xor1->evl();
    and1->evl();
    and2->evl();
    or1->evl();
    xor2->evl();

    // Take calculated local wire values and assign the values
    // to the outside wires via pointers of FA
    *o1 = coL; *o2 = sumL;
}

void halfAdder::evl () {

    // Via the HA pointers, read wire values that connect to
    // the HA from outside, and assign them to HA Local wires
    aL = *i1; bL = *i2;

    // Evaluate gates in the proper order
    xor1->evl();
    and1->evl();

    // Take calculated local wire values and assign the values
    // to the outside wires via pointers of FA
    *o1 = coL; *o2 = sumL;
}
```

timedLogicPrimitives.cpp

Gate Classes with Power and Timing Calculation



timedLogicUtilities.cpp timedLogicUtilities.h timedLogicFunctions.cpp timedLogicPrimitives.cpp

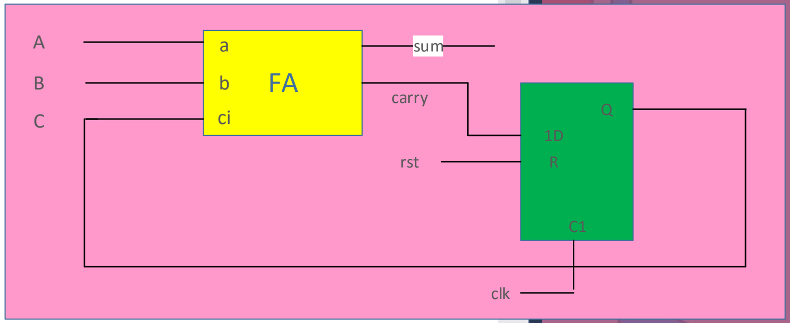
Timed Logic Classes (Global Scope)

```
int main ()
{
    wire A, B;
    wire clk, rst, fb('X',0);
    wire sum, carry;

    fullAdder *FA = new fullAdder(A, B, fb, carry, sum);
    dff_ar *FF = new dff_ar(carry, clk, rst, fb, 4, 6);

    int ai=1;

    do {
        inpBit("Serial input A", A);
        inpBit("Serial input B", B);
        inpBit("FF Clock input", clk);
        inpBit("FF Async Reset", rst);

        FA->evl();

        outBit("Carry output ", carry);
        outBit("Serial output", sum);

        FF->evl();

        outBit("Feedback", fb);

        cout << "\n" << "Continue? "; cin >> ai;
        } while (ai>0);

    cout << "Activities: Sum: " << sum.activity()
        << "; Carry: " << carry.activity()
        << "; Feedback: " << fb.activity() << '\n'
}
```

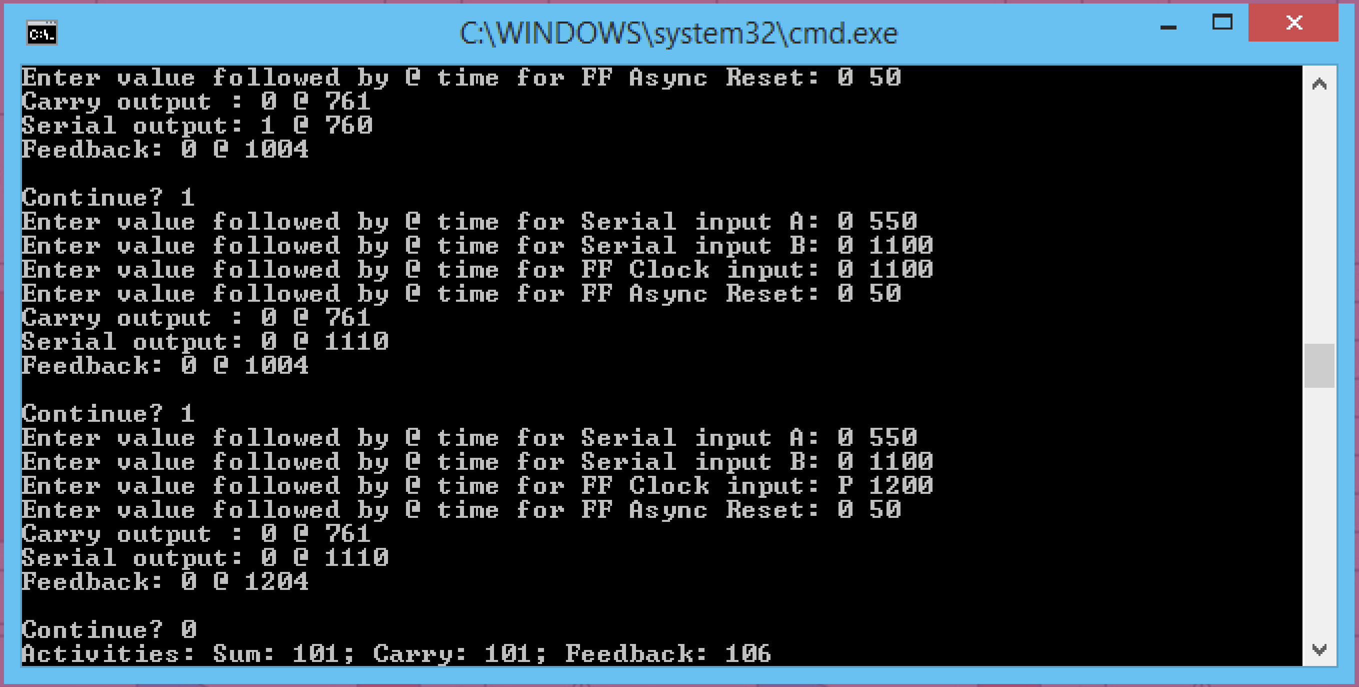
100 %

timedLogicFunctions.cpp

Serial Adder

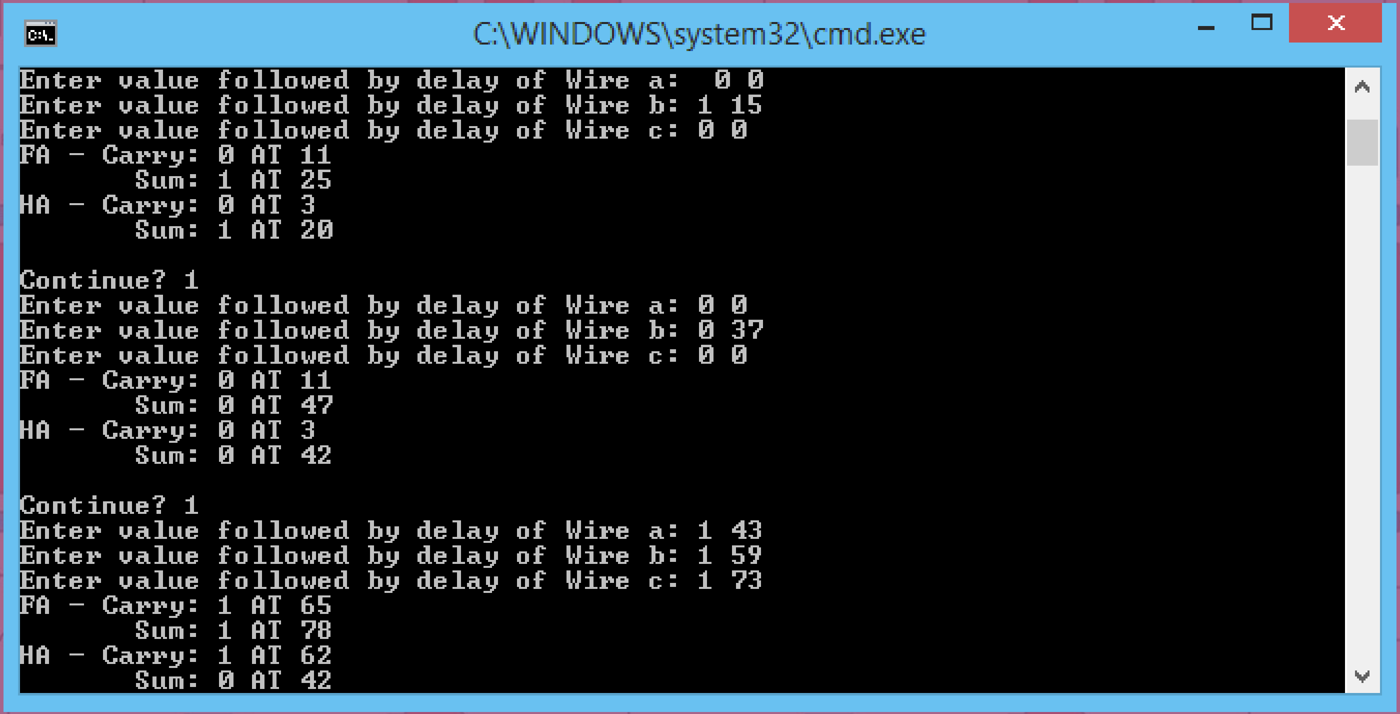


Gate Classes with Power and Timing Calculation



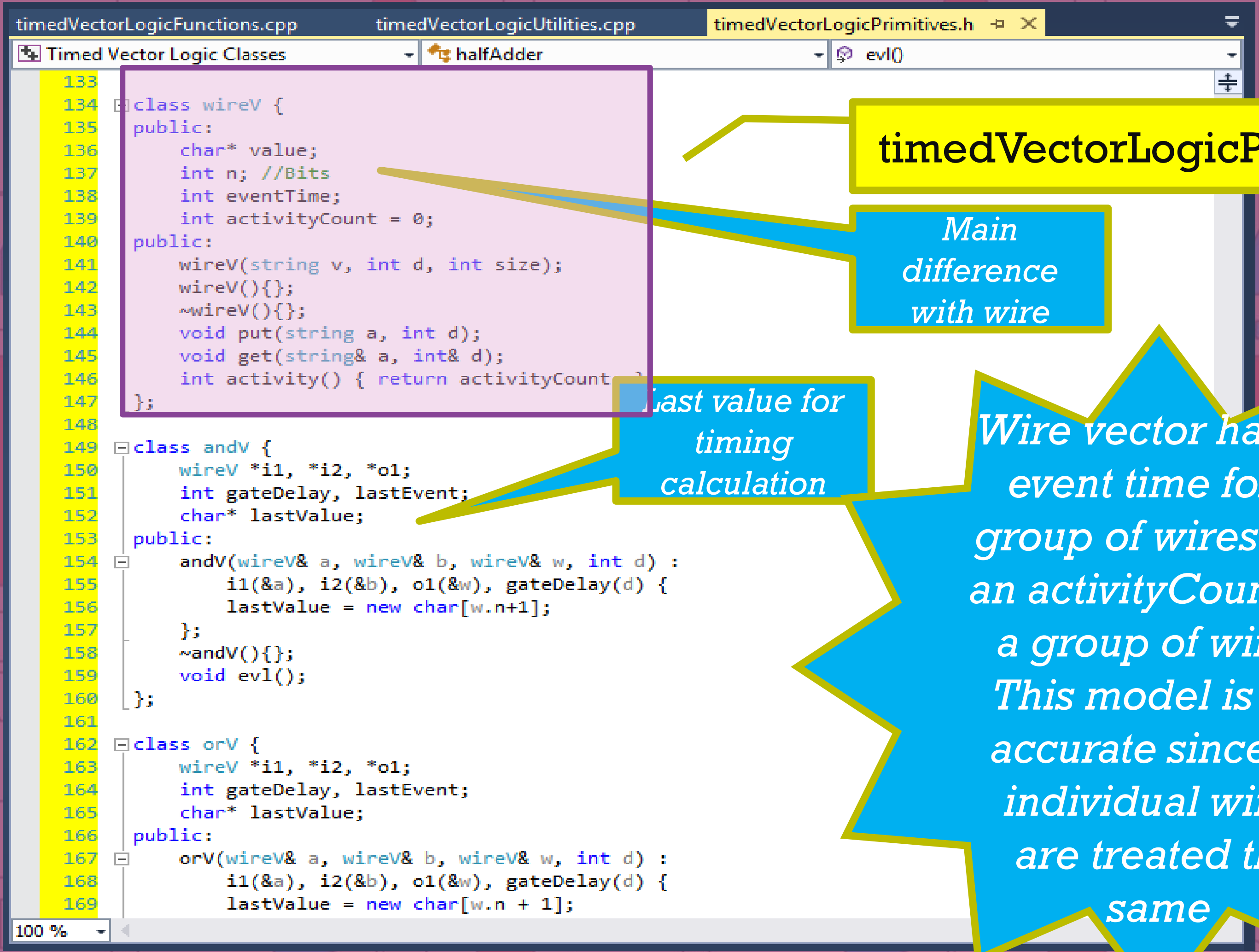
```
Enter value followed by @ time for FF Async Reset: 0 50
Carry output : 0 @ 761
Serial output: 1 @ 760
Feedback: 0 @ 1004

Continue? 1
Enter value followed by @ time for Serial input A: 0 550
Enter value followed by @ time for Serial input B: 0 1100
Enter value followed by @ time for FF Clock input: 0 1100
Enter value followed by @ time for FF Async Reset: 0 50
Carry output : 0 @ 761
Serial output: 0 @ 1110
Feedback: 0 @ 1004

Continue? 1
Enter value followed by @ time for Serial input A: 0 550
Enter value followed by @ time for Serial input B: 0 1100
Enter value followed by @ time for FF Clock input: P 1200
Enter value followed by @ time for FF Async Reset: 0 50
Carry output : 0 @ 761
Serial output: 0 @ 1110
Feedback: 0 @ 1204

Continue? 0
Activities: Sum: 101; Carry: 101; Feedback: 106
```

Gate Classes with Power and Timing Calculation



```
Enter value followed by delay of Wire a:  0 0
Enter value followed by delay of Wire b: 1 15
Enter value followed by delay of Wire c: 0 0
FA - Carry: 0 AT 11
       Sum: 1 AT 25
HA - Carry: 0 AT 3
       Sum: 1 AT 20

Continue? 1
Enter value followed by delay of Wire a: 0 0
Enter value followed by delay of Wire b: 0 37
Enter value followed by delay of Wire c: 0 0
FA - Carry: 0 AT 11
       Sum: 0 AT 47
HA - Carry: 0 AT 3
       Sum: 0 AT 42

Continue? 1
Enter value followed by delay of Wire a: 1 43
Enter value followed by delay of Wire b: 1 59
Enter value followed by delay of Wire c: 1 73
FA - Carry: 1 AT 65
       Sum: 1 AT 78
HA - Carry: 1 AT 62
       Sum: 0 AT 42
```

Wire and Gate Vectors


timedVectorLogicFunctions.cpp
timedVectorLogicUtilities.cpp
timedVectorLogicPrimitives.h
Timed Vector Logic Classes
halfAdder
evl()
100 %

```
class wireV {
public:
    char* value;
    int n; //Bits
    int eventTime;
    int activityCount = 0;
public:
    wireV(string v, int d, int size);
    wireV(){};
    ~wireV(){};
    void put(string a, int d);
    void get(string& a, int& d);
    int activity() { return activityCount
};

class andV {
    wireV *i1, *i2, *o1;
    int gateDelay, lastEvent;
    char* lastValue;
public:
    andV(wireV& a, wireV& b, wireV& w, int d) :
        i1(&a), i2(&b), o1(&w), gateDelay(d) {
        lastValue = new char[w.n+1];
    };
    ~andV(){};
    void evl();
};

class orV {
    wireV *i1, *i2, *o1;
    int gateDelay, lastEvent;
    char* lastValue;
public:
    orV(wireV& a, wireV& b, wireV& w, int d) :
        i1(&a), i2(&b), o1(&w), gateDelay(d) {
        lastValue = new char[w.n + 1];
```

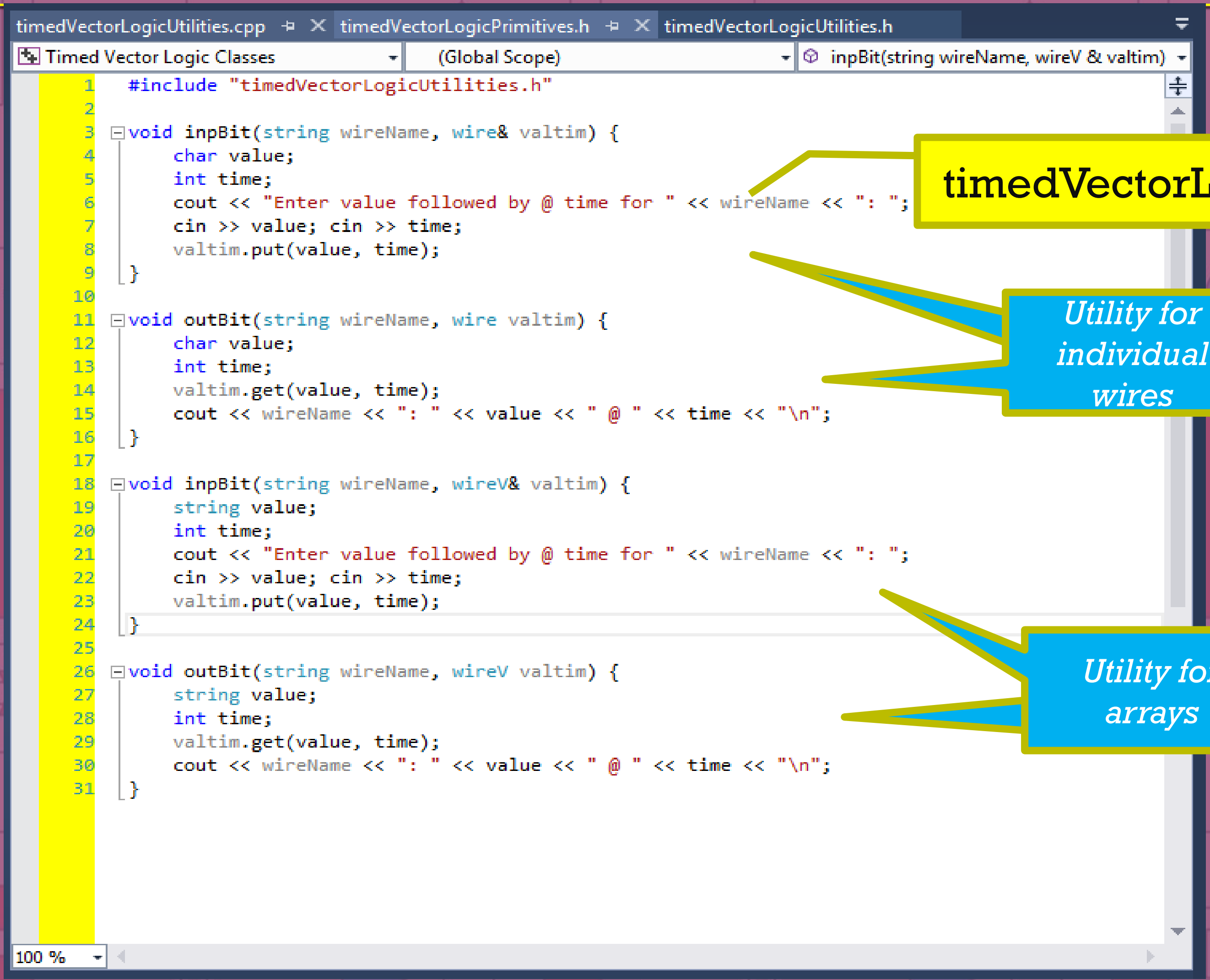
timedVectorLogicPrimitives.h

Main difference with wire

Last value for timing calculation

Wire vector has an event time for a group of wires and an activityCount for a group of wires. This model is not accurate since all individual wires are treated the same

Wire and Gate Vectors


timedVectorLogicUtilities.cpp
timedVectorLogicPrimitives.h
timedVectorLogicUtilities.h
Timed Vector Logic Classes
(Global Scope)
inpBit(string wireName, wireV & valtim)
100 %

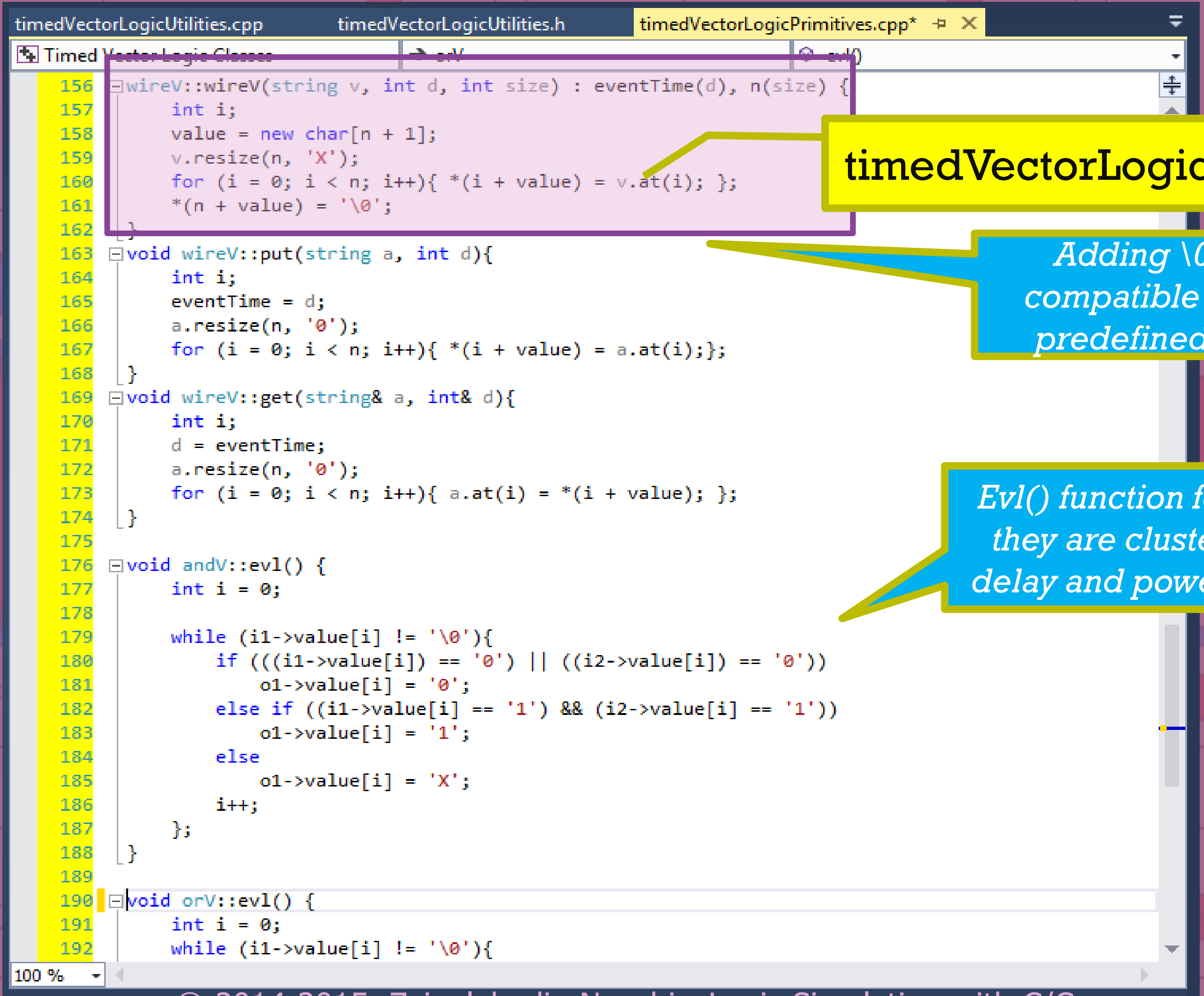
```
#include "timedVectorLogicUtilities.h"

void inpBit(string wireName, wire& valtim) {
    char value;
    int time;
    cout << "Enter value followed by @ time for " << wireName << ": ";
    cin >> value; cin >> time;
    valtim.put(value, time);
}

void outBit(string wireName, wire valtim) {
    char value;
    int time;
    valtim.get(value, time);
    cout << wireName << ": " << value << " @ " << time << "\n";
}

void inpBit(string wireName, wireV& valtim) {
    string value;
    int time;
    cout << "Enter value followed by @ time for " << wireName << ": ";
    cin >> value; cin >> time;
    valtim.put(value, time);
}

void outBit(string wireName, wireV valtim) {
    string value;
    int time;
    valtim.get(value, time);
    cout << wireName << ": " << value << " @ " << time << "\n";
}
```

timedVectorLogicPrimitives.h

Utility for individual wires

Utility for arrays

Wire and Gate Vectors


timedVectorLogicUtilities.cpp
timedVectorLogicUtilities.h
timedVectorLogicPrimitives.cpp*
Timed Vector Logic Classes
orV
evl()
100 %

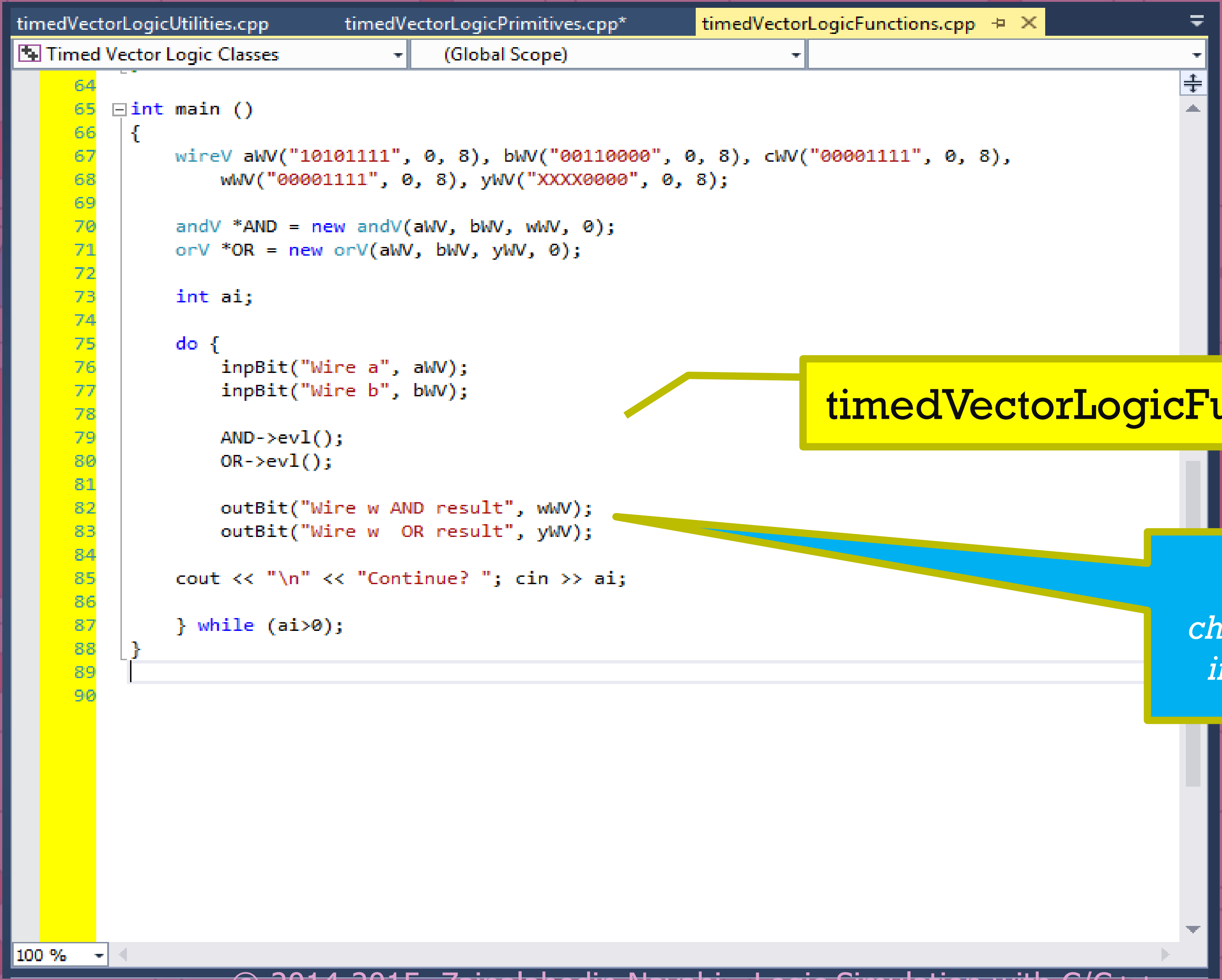
```
wireV::wireV(string v, int d, int size) : eventTime(d), n(size) {
    int i;
    value = new char[n + 1];
    v.resize(n, 'X');
    for (i = 0; i < n; i++){ *(i + value) = v.at(i); };
    *(n + value) = '\0';
}
void wireV::put(string a, int d){
    int i;
    eventTime = d;
    a.resize(n, '0');
    for (i = 0; i < n; i++){ *(i + value) = a.at(i);};
}
void wireV::get(string& a, int& d){
    int i;
    d = eventTime;
    a.resize(n, '0');
    for (i = 0; i < n; i++){ a.at(i) = *(i + value); };
}

void andV::evl() {
    int i = 0;

    while (i1->value[i] != '\0'){
        if (((i1->value[i]) == '0') || ((i2->value[i]) == '0'))
            o1->value[i] = '0';
        else if ((i1->value[i] == '1') && (i2->value[i] == '1'))
            o1->value[i] = '1';
        else
            o1->value[i] = 'X';
        i++;
    };
}

void orV::evl() {
    int i = 0;
    while (i1->value[i] != '\0'){
```

timedVectorLogicPrimitives.cpp

Adding \0 to make it compatible with the c++ predefined string class

Evl() function for wireV. Since they are clusters, individual delay and power do not apply

Wire and Gate Vectors

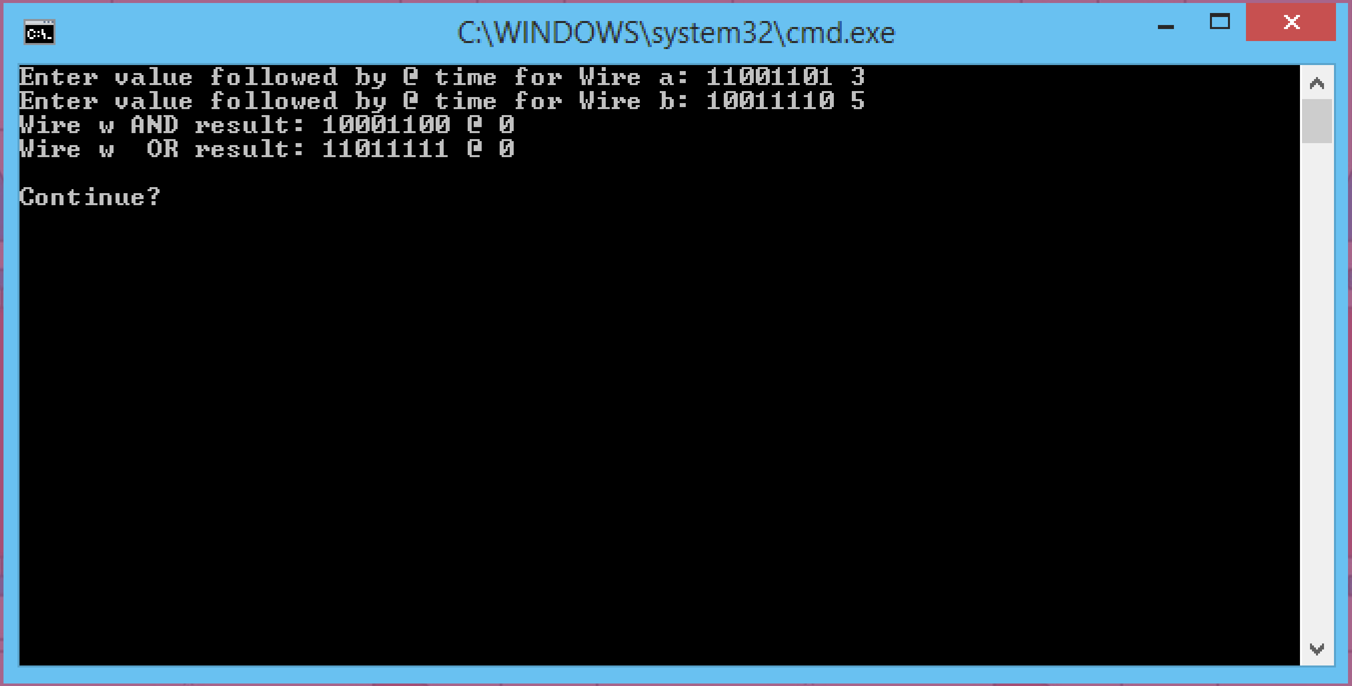

timedVectorLogicUtilities.cpp
timedVectorLogicPrimitives.cpp*
timedVectorLogicFunctions.cpp
Timed Vector Logic Classes
(Global Scope)
100 %

```
int main ()
{
    wireV aWV("10101111", 0, 8), bWV("00110000", 0, 8), cWV("00001111", 0, 8),
        wWV("00001111", 0, 8), yWV("XXXX0000", 0, 8);

    andV *AND = new andV(aWV, bWV, wWV, 0);
    orV *OR = new orV(aWV, bWV, yWV, 0);

    int ai;

    do {
        inpBit("Wire a", aWV);
        inpBit("Wire b", bWV);

        AND->evl();
        OR->evl();

        outBit("Wire w AND result", wWV);
        outBit("Wire w  OR result", yWV);

    cout << "\n" << "Continue? "; cin >> ai;

    } while (ai>0);
}
```

timedVectorLogicFunctions.cpp

Vectors have character pointer instead of char

Wire and Gate Vectors



```
Enter value followed by @ time for Wire a: 11001101 3
Enter value followed by @ time for Wire b: 10011110 5
Wire w AND result: 10001100 @ 0
Wire w  OR result: 11011111 @ 0

Continue?
```

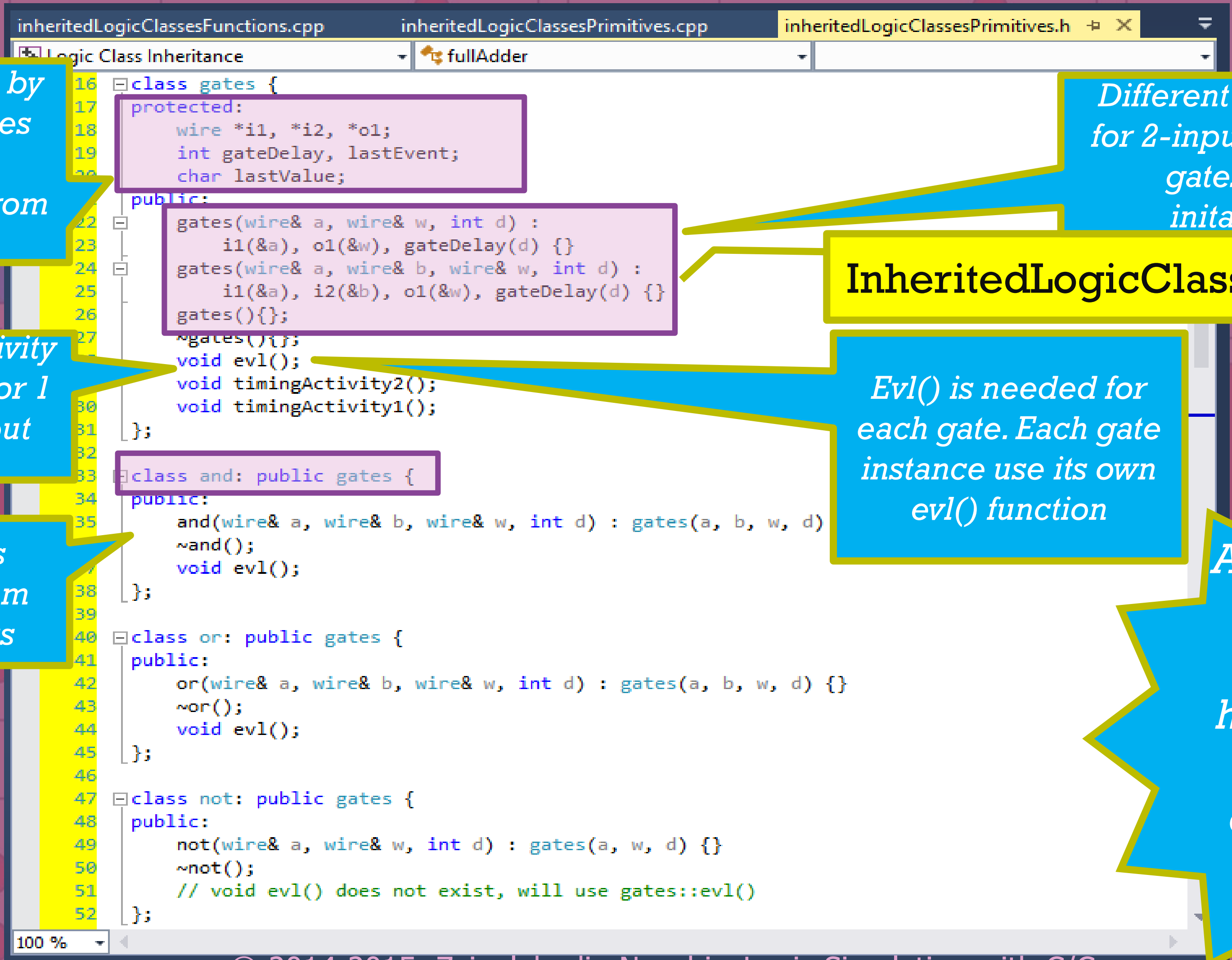
Logic Simulation with C/C++

- Containing Event Based Timing
  - To include in wires
  - To include in gates
- Gate-based structures
- Gate pointers and objects
- Wire and gate vectors

- Inheritance in Logic Structures
  - A generic gate definition
  - Gates to include timing
  - Building structures from objects

- Hierarchal Modeling of Digital Components
  - Wire functionalities
  - Gate functionalities
  - Polymorphic gate base
  - Virtual functions
  - Functions overwriting
  - Flip flop description hierarchal

Inheritance in Logic Functions



inheritedLogicClassesFunctions.cpp | inheritedLogicClassesPrimitives.cpp | inheritedLogicClassesPrimitives.h

Logic Class Inheritance | fullAdder

```
class gates {
protected:
    wire *i1, *i2, *o1;
    int gateDelay, lastEvent;
    char lastValue;
public:
    gates(wire& a, wire& w, int d) :
        i1(&a), o1(&w), gateDelay(d) {}
    gates(wire& a, wire& b, wire& w, int d) :
        i1(&a), i2(&b), o1(&w), gateDelay(d) {}
    gates(){};
    ~gates(){};
    void evl();
    void timingActivity2();
    void timingActivity1();
};

class and: public gates {
public:
    and(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d)
    ~and();
    void evl();
};

class or: public gates {
public:
    or(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d) {}
    ~or();
    void evl();
};

class not: public gates {
public:
    not(wire& a, wire& w, int d) : gates(a, w, d) {}
    ~not();
    // void evl() does not exist, will use gates::evl()
};
```

100 %

Accessible by gate classes that are inherited from gates

Different constructors for 2-input and 1-input gates and no initailization

InheritedLogicClassPrimitives.h

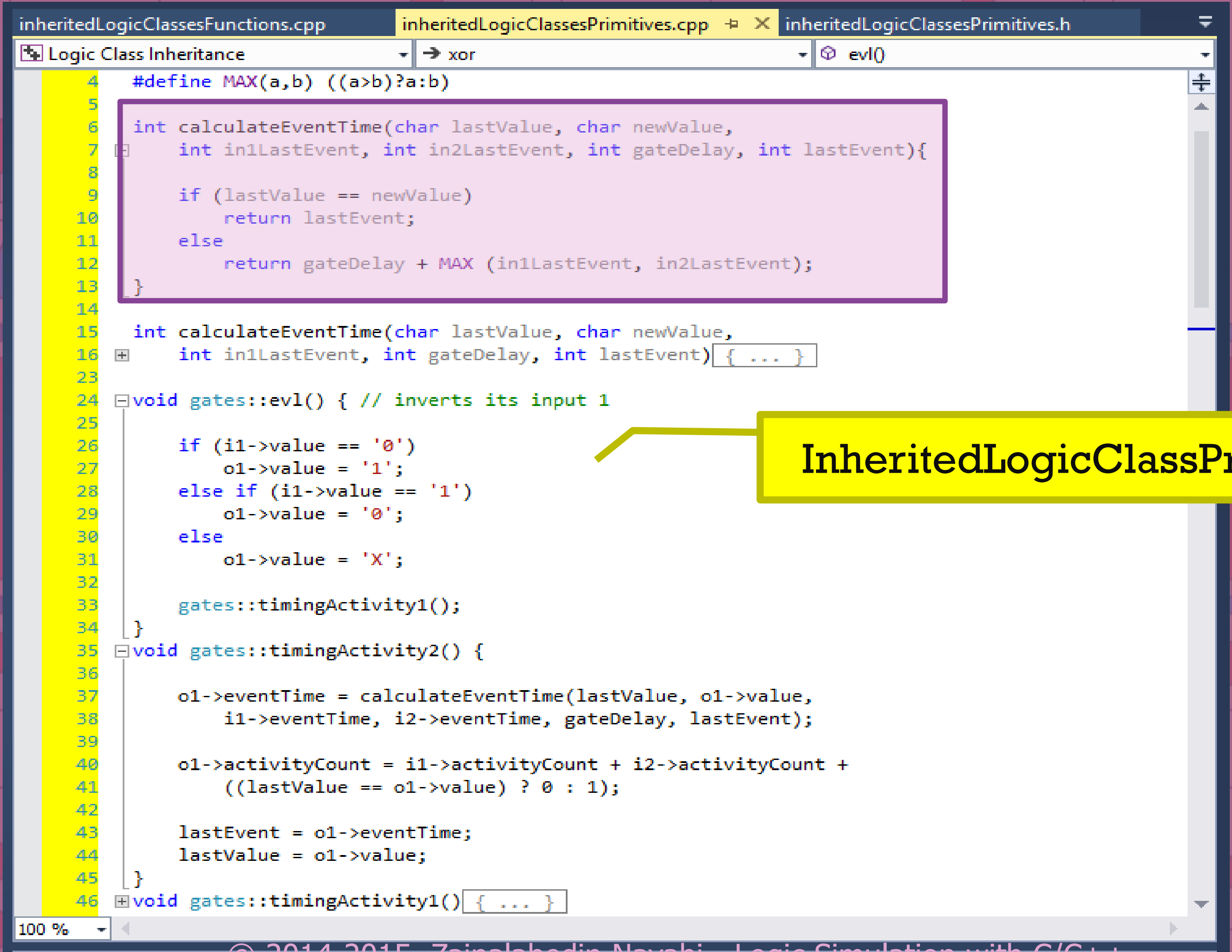
Timing activity functions for 1 and 2 input gates

Evl() is needed for each gate. Each gate instance use its own evl() function

All gates inherit from gate class

An inherited class that does not have its own evl() can depend on the base class

Inheritance in Logic Functions



inheritedLogicClassesFunctions.cpp | inheritedLogicClassesPrimitives.cpp | inheritedLogicClassesPrimitives.h

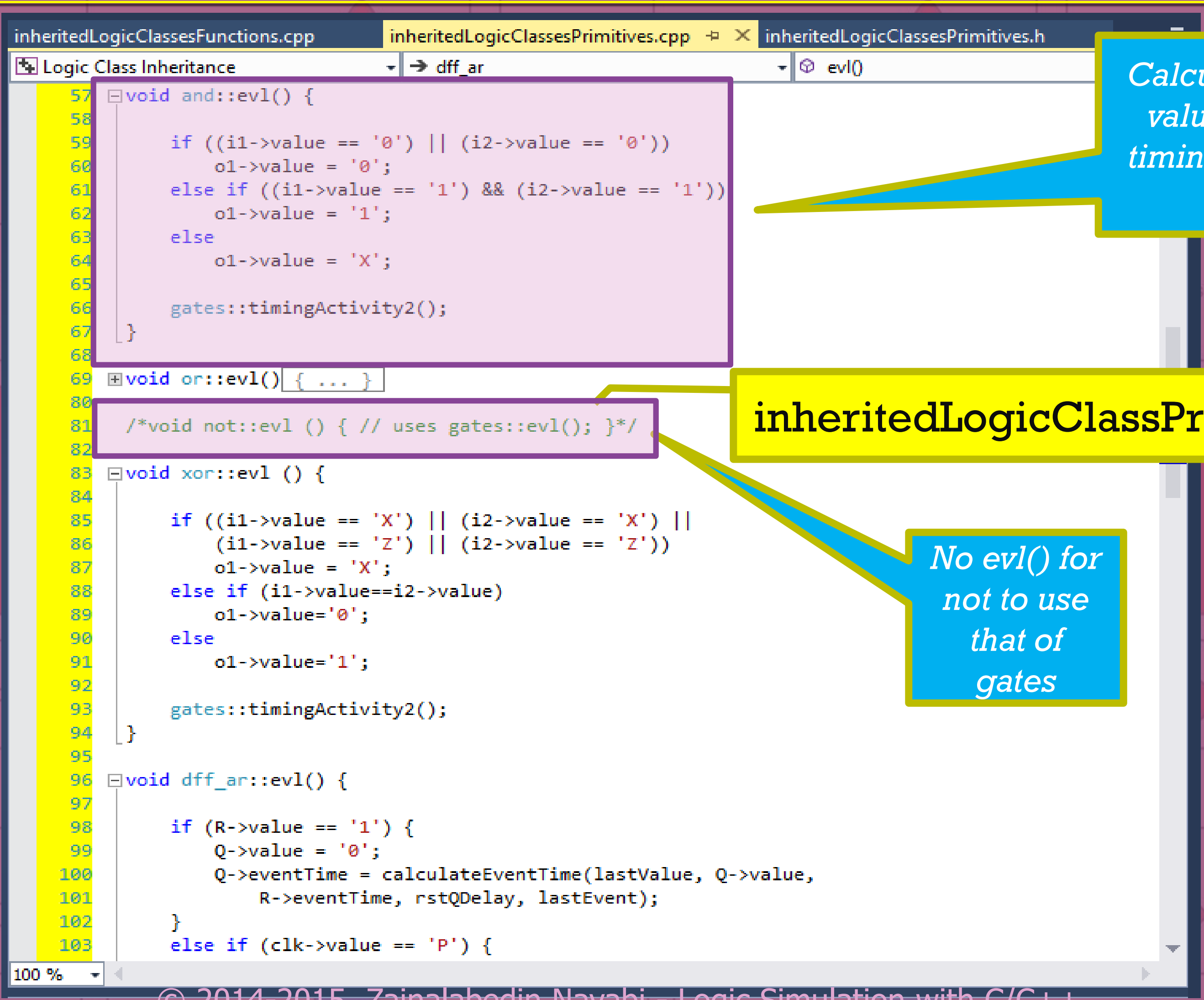
Logic Class Inheritance | xor | evl()

```
#define MAX(a,b) ((a>b)?a:b)

int calculateEventTime(char lastValue, char newValue,
    int in1LastEvent, int in2LastEvent, int gateDelay, int lastEvent){

    if (lastValue == newValue)
        return lastEvent;
    else
        return gateDelay + MAX (in1LastEvent, in2LastEvent);
}

int calculateEventTime(char lastValue, char newValue,
    int in1LastEvent, int gateDelay, int lastEvent){ ... }

void gates::evl() { // inverts its input 1

    if (i1->value == '0')
        o1->value = '1';
    else if (i1->value == '1')
        o1->value = '0';
    else
        o1->value = 'X';

    gates::timingActivity1();
}
void gates::timingActivity2() {

    o1->eventTime = calculateEventTime(lastValue, o1->value,
        i1->eventTime, i2->eventTime, gateDelay, lastEvent);

    o1->activityCount = i1->activityCount + i2->activityCount +
        ((lastValue == o1->value) ? 0 : 1);

    lastEvent = o1->eventTime;
    lastValue = o1->value;
}
void gates::timingActivity1(){ ... }
```

100 %

InheritedLogicClassPrimitives.cpp

Inheritance in Logic Functions



inheritedLogicClassesFunctions.cpp | inheritedLogicClassesPrimitives.cpp | inheritedLogicClassesPrimitives.h

Logic Class Inheritance | dff_ar | evl()

```
void and::evl() {

    if ((i1->value == '0') || (i2->value == '0'))
        o1->value = '0';
    else if ((i1->value == '1') && (i2->value == '1'))
        o1->value = '1';
    else
        o1->value = 'X';

    gates::timingActivity2();
}

void or::evl() { ... }

/*void not::evl () { // uses gates::evl(); }*/

void xor::evl () {

    if ((i1->value == 'X') || (i2->value == 'X') ||
        (i1->value == 'Z') || (i2->value == 'Z'))
        o1->value = 'X';
    else if (i1->value==i2->value)
        o1->value='0';
    else
        o1->value='1';

    gates::timingActivity2();
}

void dff_ar::evl() {

    if (R->value == '1') {
        Q->value = '0';
        Q->eventTime = calculateEventTime(lastValue, Q->value,
            R->eventTime, rstQDelay, lastEvent);
    }
    else if (clk->value == 'P') {
```

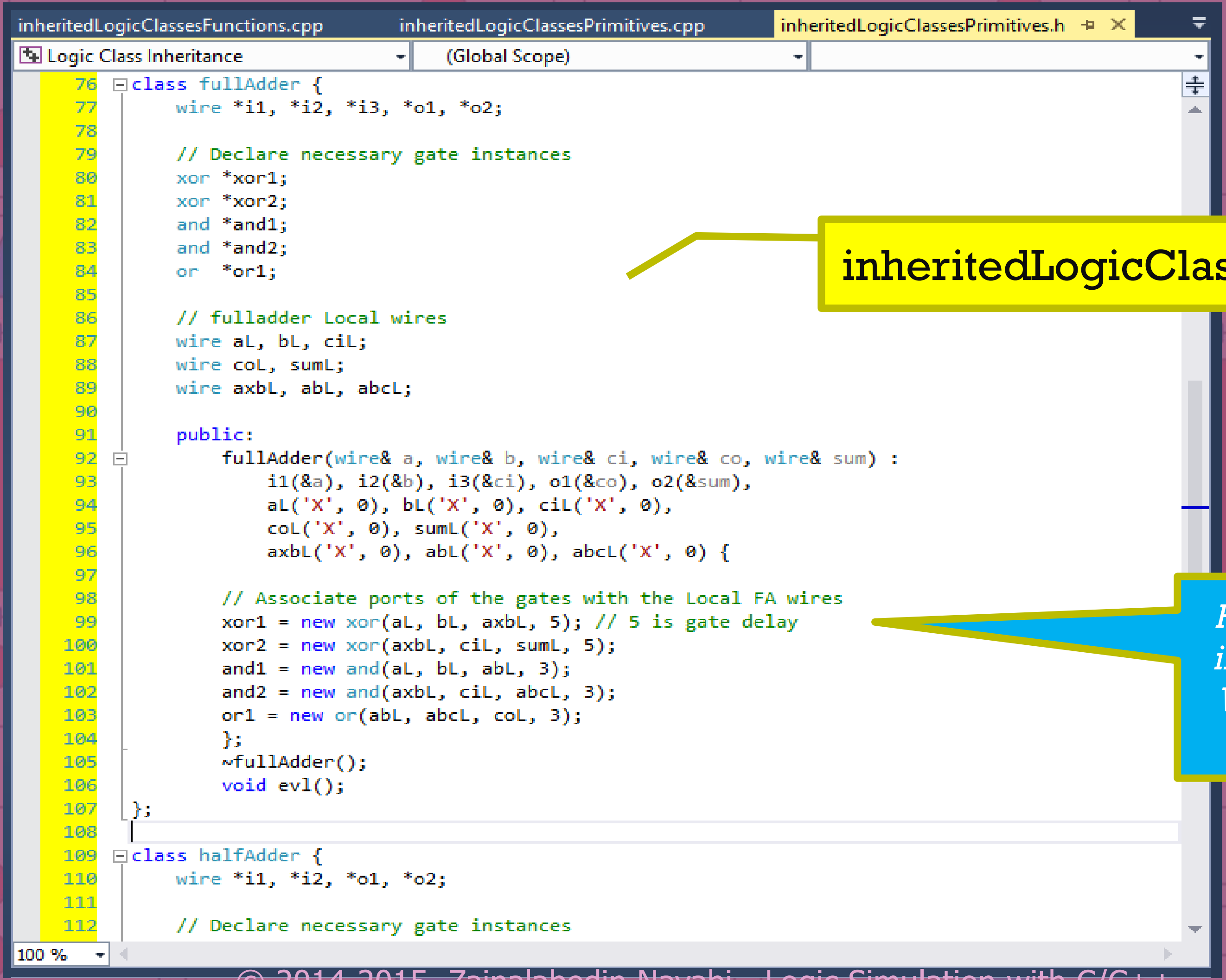
100 %

Calculate output value and call timing activity at gates

inheritedLogicClassPrimitives.cpp

No evl() for not to use that of gates

Structures from Inherited Gates

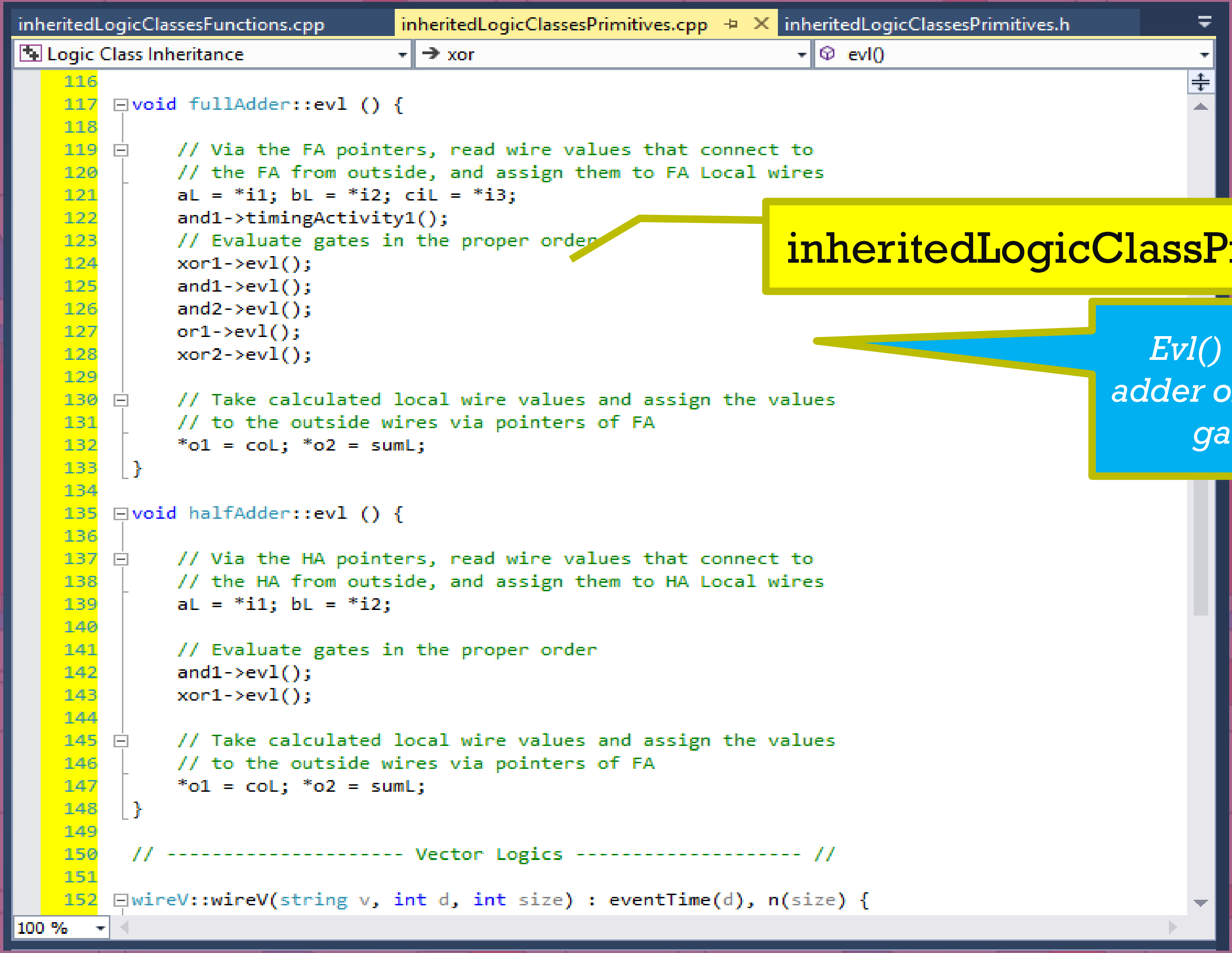

inheritedLogicClassesFunctions.cpp
inheritedLogicClassesPrimitives.cpp
inheritedLogicClassesPrimitives.h
Logic Class Inheritance
(Global Scope)
100 %

```
class fullAdder {
    wire *i1, *i2, *i3, *o1, *o2;

    // Declare necessary gate instances
    xor *xor1;
    xor *xor2;
    and *and1;
    and *and2;
    or  *or1;

    // fulladder Local wires
    wire aL, bL, ciL;
    wire coL, sumL;
    wire axbL, abL, abcL;

    public:
        fullAdder(wire& a, wire& b, wire& ci, wire& co, wire& sum) :
            i1(&a), i2(&b), i3(&ci), o1(&co), o2(&sum),
            aL('X', 0), bL('X', 0), ciL('X', 0),
            coL('X', 0), sumL('X', 0),
            axbL('X', 0), abL('X', 0), abcL('X', 0) {

        // Associate ports of the gates with the Local FA wires
        xor1 = new xor(aL, bL, axbL, 5); // 5 is gate delay
        xor2 = new xor(axbL, ciL, sumL, 5);
        and1 = new and(aL, bL, abL, 3);
        and2 = new and(axbL, ciL, abcL, 3);
        or1 = new or(abL, abcL, coL, 3);
        };
        ~fullAdder();
        void evl();
};

class halfAdder {
    wire *i1, *i2, *o1, *o2;

    // Declare necessary gate instances
```

inheritedLogicClassPrimitives.cpp

Full adder uses inherited gates. Wiring is done here.

Inheritance in Logic Structures

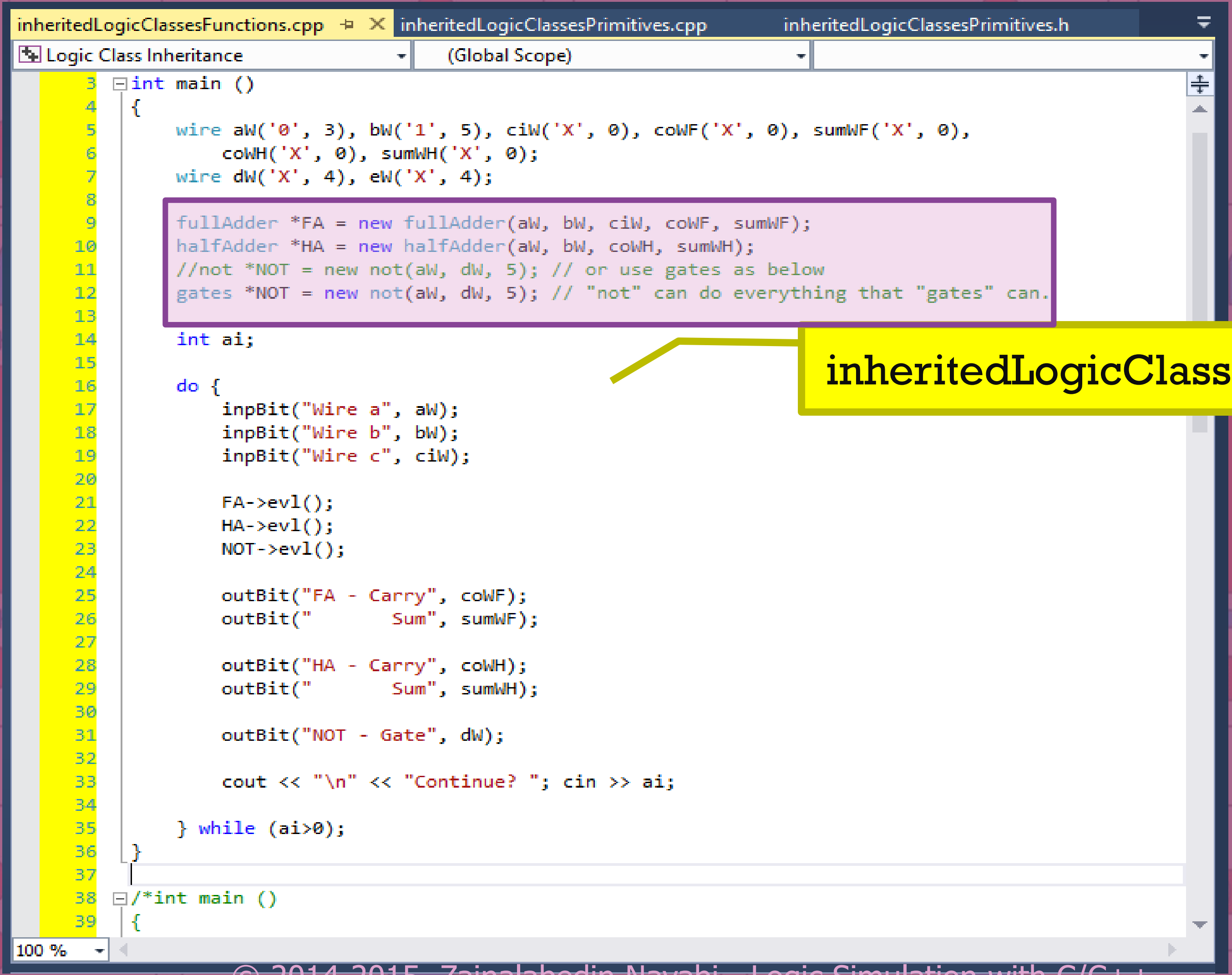

inheritedLogicClassesFunctions.cpp
inheritedLogicClassesPrimitives.cpp
inheritedLogicClassesPrimitives.h
Logic Class Inheritance
xor
evl()
100 %

```
void fullAdder::evl () {

    // Via the FA pointers, read wire values that connect to
    // the FA from outside, and assign them to FA Local wires
    aL = *i1; bL = *i2; ciL = *i3;
    and1->timingActivity1();
    // Evaluate gates in the proper order
    xor1->evl();
    and1->evl();
    and2->evl();
    or1->evl();
    xor2->evl();

    // Take calculated local wire values and assign the values
    // to the outside wires via pointers of FA
    *o1 = coL; *o2 = sumL;
}

void halfAdder::evl () {

    // Via the HA pointers, read wire values that connect to
    // the HA from outside, and assign them to HA Local wires
    aL = *i1; bL = *i2;

    // Evaluate gates in the proper order
    and1->evl();
    xor1->evl();

    // Take calculated local wire values and assign the values
    // to the outside wires via pointers of FA
    *o1 = coL; *o2 = sumL;
}

// -------------------- Vector Logics -------------------- //

wireV::wireV(string v, int d, int size) : eventTime(d), n(size) {
```

inheritedLogicClassPrimitives.cpp

Evl() of full adder order the gates

Inheritance in Logic Structures



inheritedLogicClassesFunctions.cpp   inheritedLogicClassesPrimitives.cpp   inheritedLogicClassesPrimitives.h

Logic Class Inheritance   (Global Scope)

```
int main ()
{
    wire aW('0', 3), bW('1', 5), ciW('X', 0), coWF('X', 0), sumWF('X', 0),
        coWH('X', 0), sumWH('X', 0);
    wire dW('X', 4), eW('X', 4);

    fullAdder *FA = new fullAdder(aW, bW, ciW, coWF, sumWF);
    halfAdder *HA = new halfAdder(aW, bW, coWH, sumWH);
    //not *NOT = new not(aW, dW, 5); // or use gates as below
    gates *NOT = new not(aW, dW, 5); // "not" can do everything that "gates" can.

    int ai;

    do {
        inpBit("Wire a", aW);
        inpBit("Wire b", bW);
        inpBit("Wire c", ciW);

        FA->evl();
        HA->evl();
        NOT->evl();

        outBit("FA - Carry", coWF);
        outBit("       Sum", sumWF);

        outBit("HA - Carry", coWH);
        outBit("       Sum", sumWH);

        outBit("NOT - Gate", dW);

        cout << "\n" << "Continue? "; cin >> ai;

    } while (ai>0);
}

/*int main ()
{
```

100 %

inheritedLogicClassFunctions.cpp

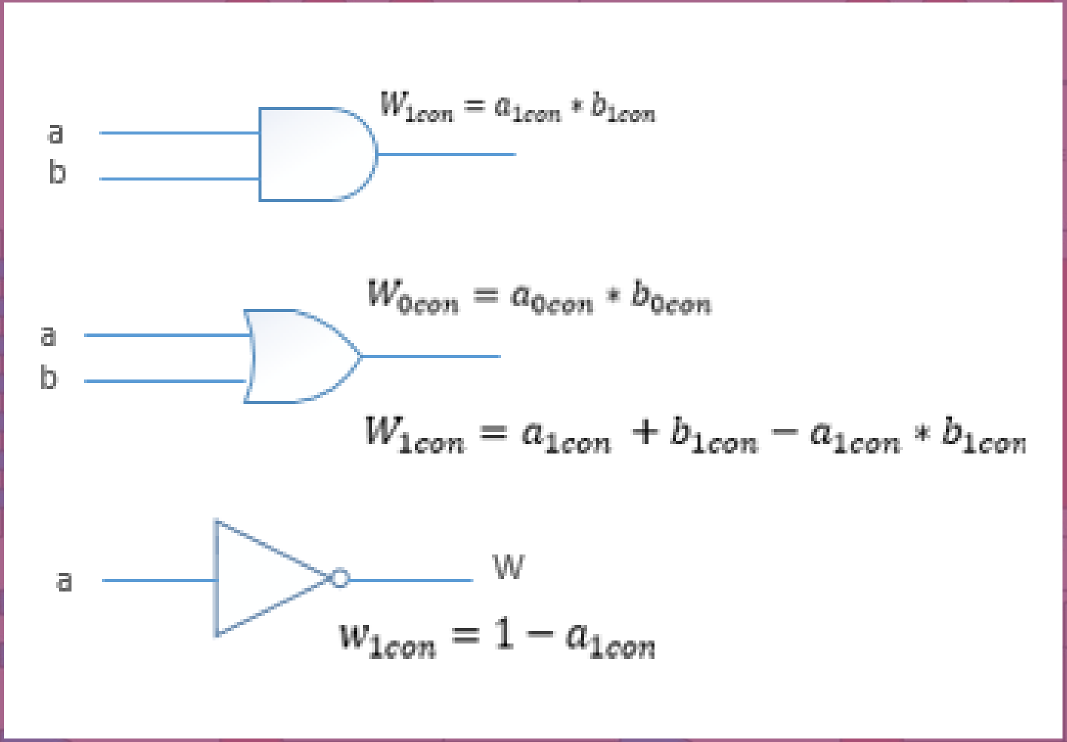
Logic Simulation with C/C++

- Containing Event Based Timing
  - To include in wires
  - To include in gates
- Gate-based structures
- Gate pointers and objects
- Wire and gate vectors

- Inheritance in Logic Structures
  - A generic gate definition
  - Gates to include timing
  - Building structures from objects

- Hierarchal Modeling of Digital Components
  - Wire functionalities
  - Gate functionalities
  - Polymorphic gate base
  - Virtual functions
  - Functions overwriting
  - Flip flop description hierarchal

Logic Testability Analysis



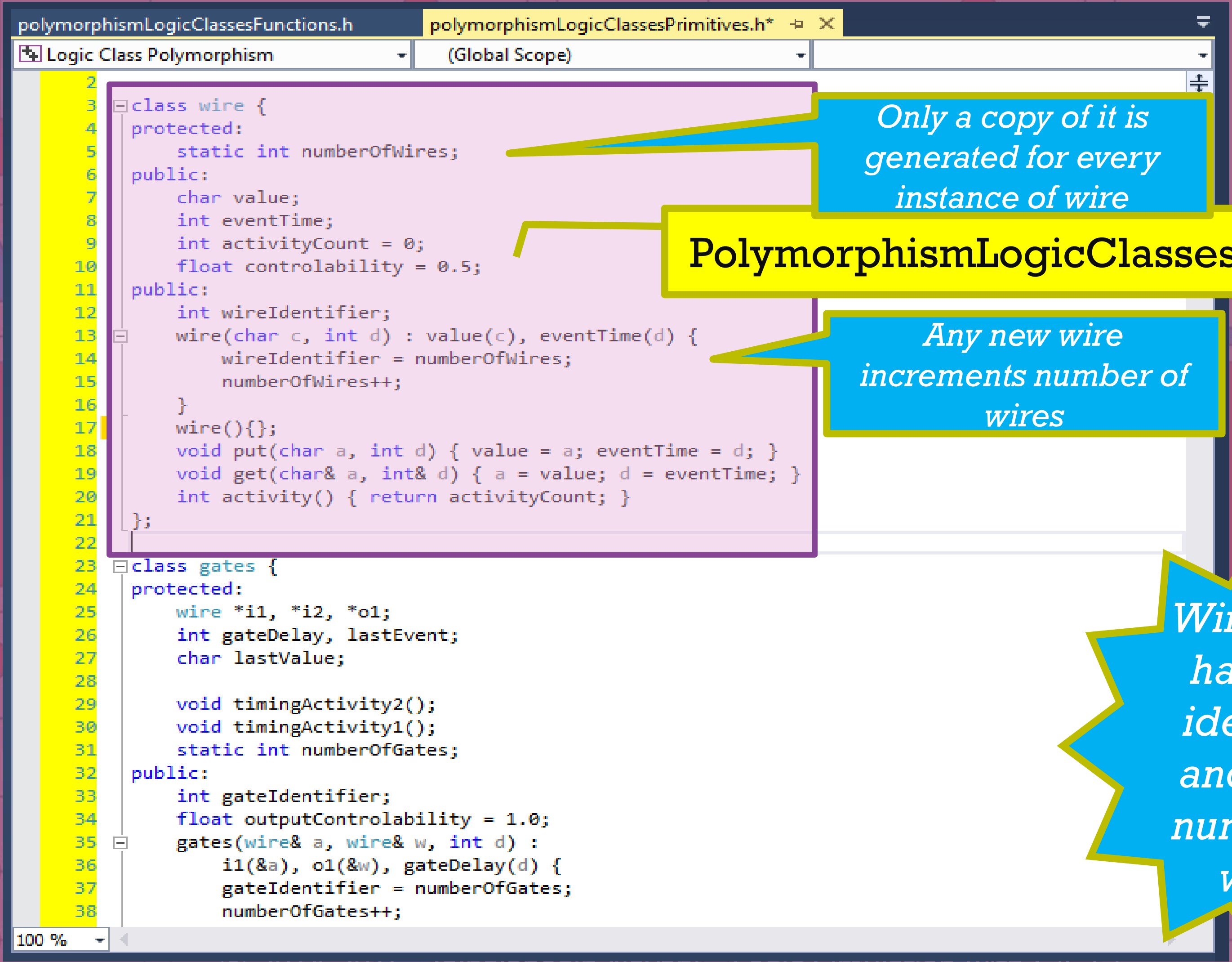
$W_{1con} = a_{1con} * b_{1con}$

$W_{0con} = a_{0con} * b_{0con}$

$W_{1con} = a_{1con} + b_{1con} - a_{1con} * b_{1con}$

$w_{1con} = 1 - a_{1con}$

Wire Functionality



polymorphismLogicClassesFunctions.h
polymorphismLogicClassesPrimitives.h*
Logic Class Polymorphism
(Global Scope)
100 %

```
class wire {
protected:
    static int numberOfWires;
public:
    char value;
    int eventTime;
    int activityCount = 0;
    float controlability = 0.5;
public:
    int wireIdentifier;
    wire(char c, int d) : value(c), eventTime(d) {
        wireIdentifier = numberOfWires;
        numberOfWires++;
    }
    wire(){};
    void put(char a, int d) { value = a; eventTime = d; }
    void get(char& a, int& d) { a = value; d = eventTime; }
    int activity() { return activityCount; }
};

class gates {
protected:
    wire *i1, *i2, *o1;
    int gateDelay, lastEvent;
    char lastValue;

    void timingActivity2();
    void timingActivity1();
    static int numberOfGates;
public:
    int gateIdentifier;
    float outputControlability = 1.0;
    gates(wire& a, wire& w, int d) :
        i1(&a), o1(&w), gateDelay(d) {
        gateIdentifier = numberOfGates;
        numberOfGates++;
```

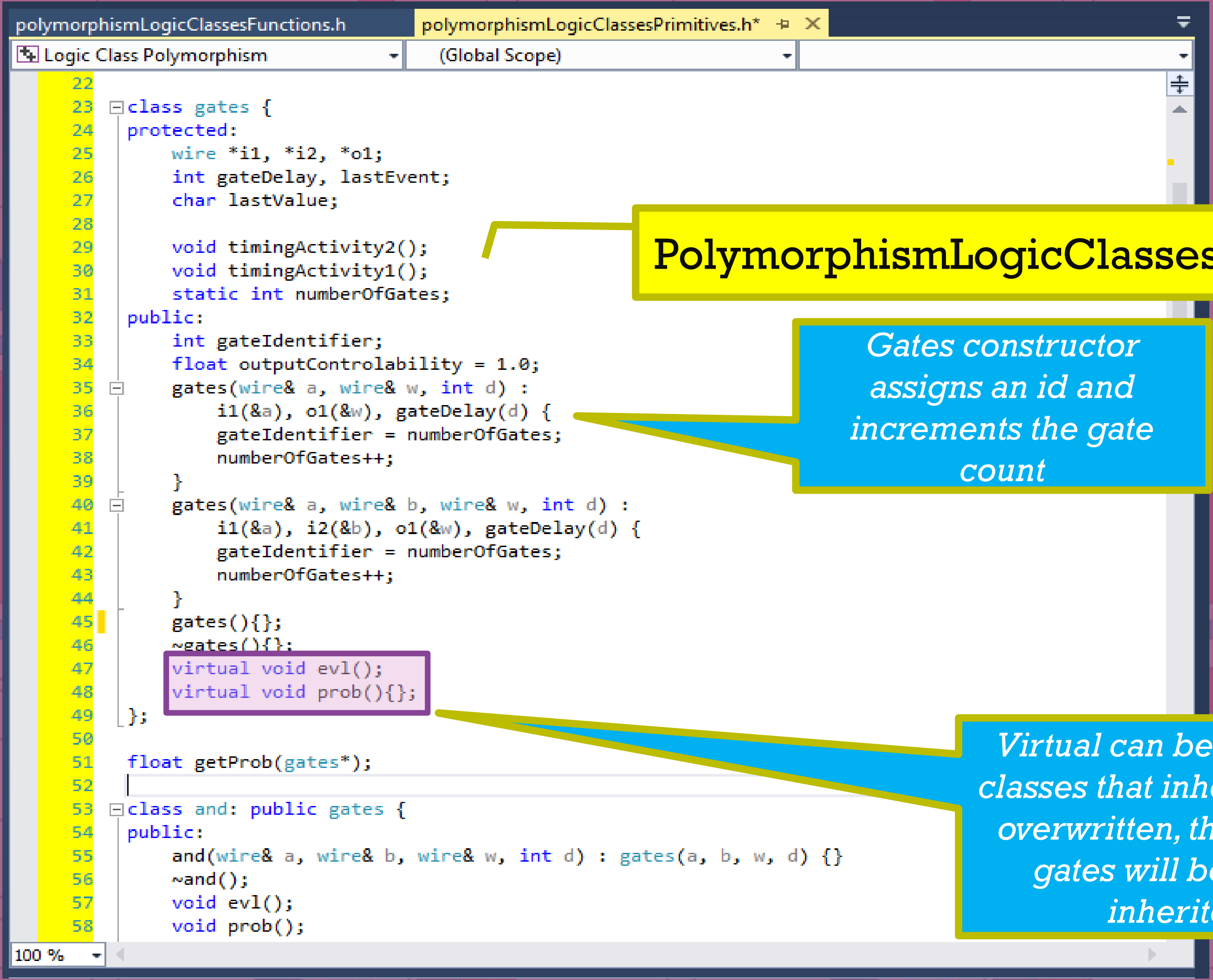
Only a copy of it is generated for every instance of wire

PolymorphismLogicClassesPrimitives.h

Any new wire increments number of wires

Wire class has wire identifier and static number of wires

Polymorphic Gate Base



polymorphismLogicClassesFunctions.h | polymorphismLogicClassesPrimitives.h*

Logic Class Polymorphism | (Global Scope)

```
class gates {
protected:
    wire *i1, *i2, *o1;
    int gateDelay, lastEvent;
    char lastValue;

    void timingActivity2();
    void timingActivity1();
    static int numberOfGates;
public:
    int gateIdentifier;
    float outputControlability = 1.0;
    gates(wire& a, wire& w, int d) :
        i1(&a), o1(&w), gateDelay(d) {
        gateIdentifier = numberOfGates;
        numberOfGates++;
    }
    gates(wire& a, wire& b, wire& w, int d) :
        i1(&a), i2(&b), o1(&w), gateDelay(d) {
        gateIdentifier = numberOfGates;
        numberOfGates++;
    }
    gates(){};
    ~gates(){};
    virtual void evl();
    virtual void prob(){};
};

float getProb(gates*);

class and: public gates {
public:
    and(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d) {}
    ~and();
    void evl();
    void prob();
```

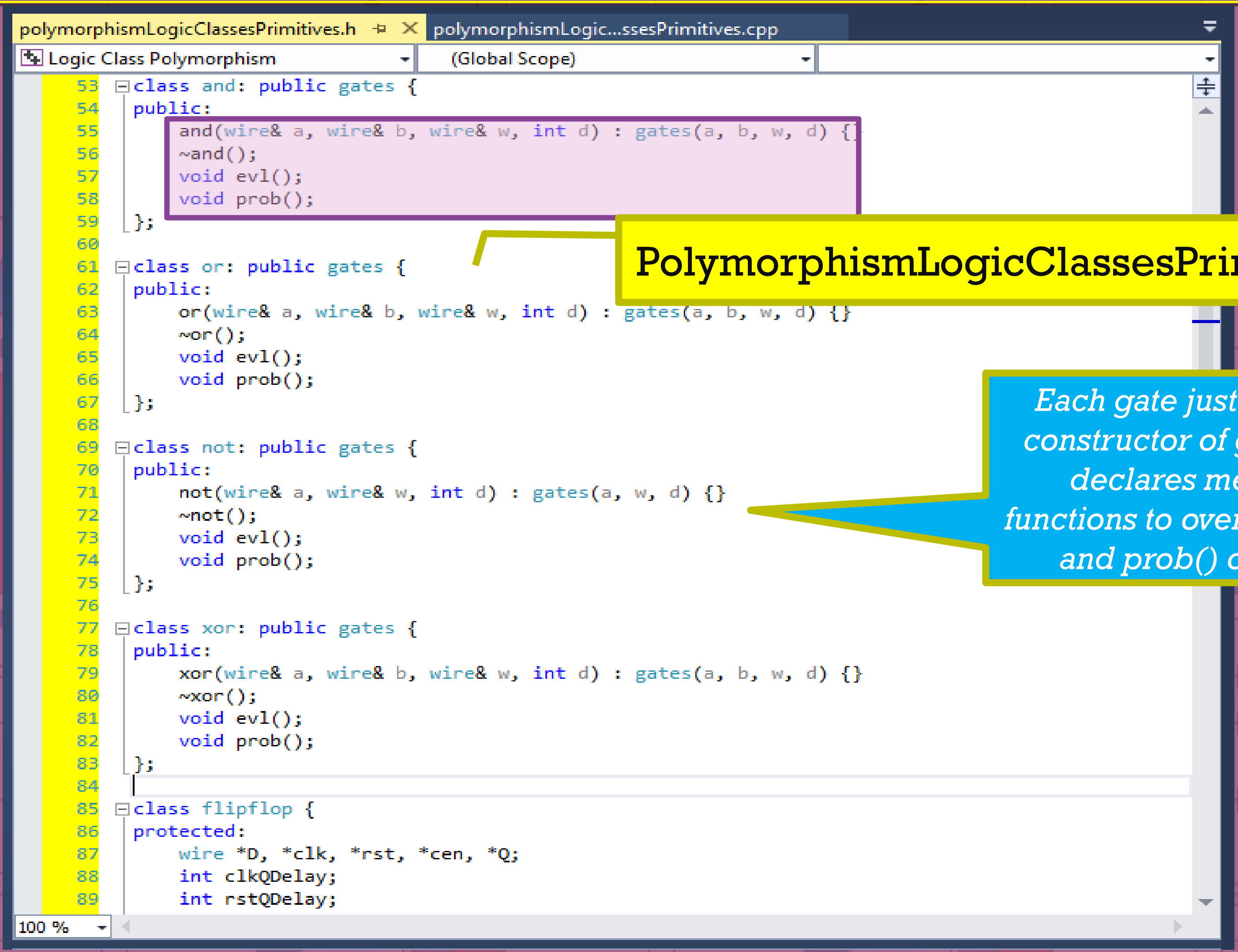
100 %

PolymorphismLogicClassesPrimitives.h

Gates constructor assigns an id and increments the gate count

Virtual can be overwritten by classes that inherit from it. If not overwritten, the same evl() of gates will be used for an inherited class

Polymorphic Gate Base

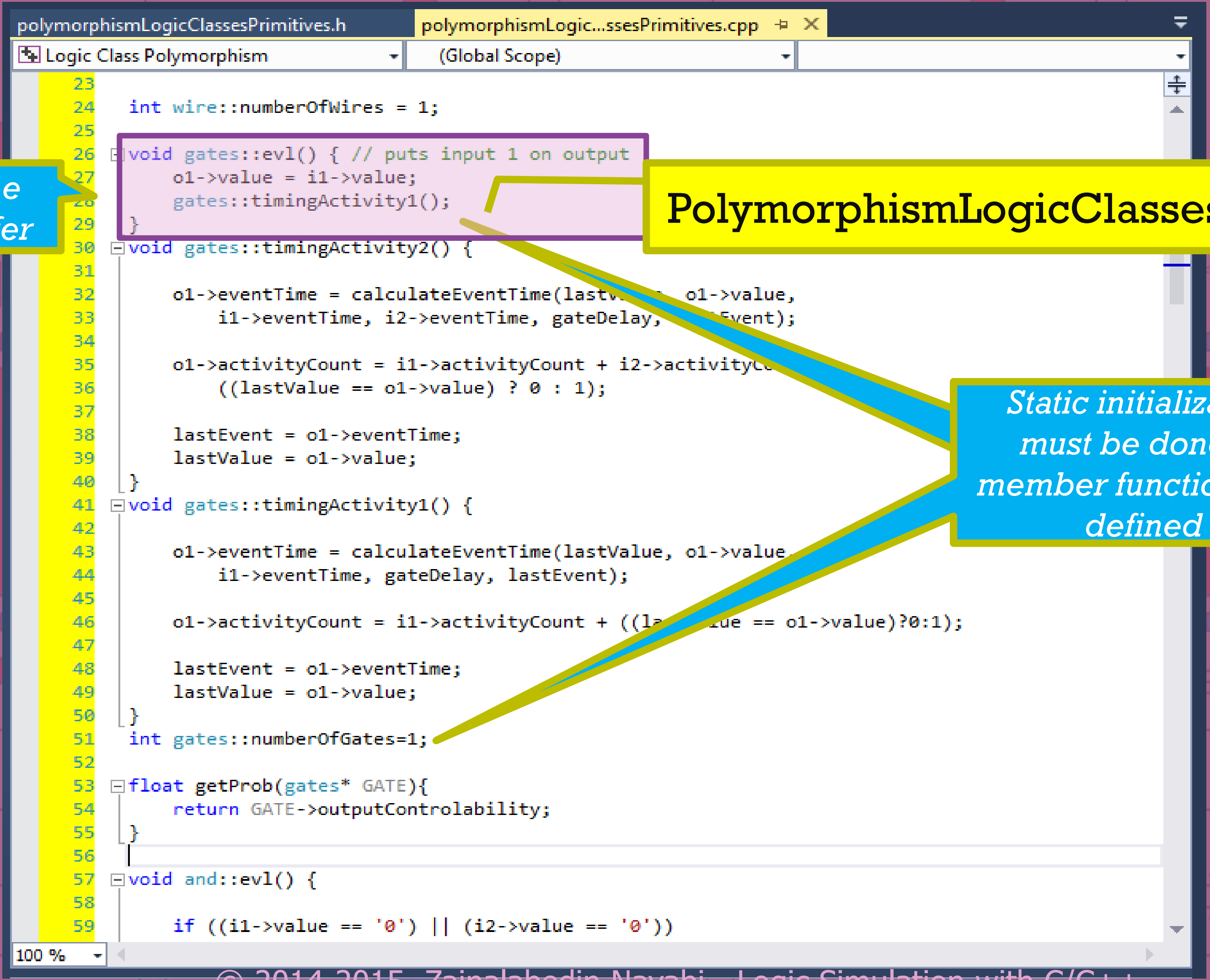

polymorphismLogicClassesPrimitives.h
polymorphismLogic...ssesPrimitives.cpp
Logic Class Polymorphism
(Global Scope)
100 %

```
class and: public gates {
public:
    and(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d) {}
    ~and();
    void evl();
    void prob();
};

class or: public gates {
public:
    or(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d) {}
    ~or();
    void evl();
    void prob();
};

class not: public gates {
public:
    not(wire& a, wire& w, int d) : gates(a, w, d) {}
    ~not();
    void evl();
    void prob();
};

class xor: public gates {
public:
    xor(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d) {}
    ~xor();
    void evl();
    void prob();
};

class flipflop {
protected:
    wire *D, *clk, *rst, *cen, *Q;
    int clkQDelay;
    int rstQDelay;
```

PolymorphismLogicClassesPrimitives.h

Each gate just uses the constructor of gates and declares member functions to overwrite evl() and prob() of gates

Polymorphic Gate Base



polymorphismLogicClassesPrimitives.h | polymorphismLogic...ssesPrimitives.cpp

Logic Class Polymorphism | (Global Scope)

```cpp

int wire::numberOfWires = 1;

void gates::evl() { // puts input 1 on output
    o1->value = i1->value;
    gates::timingActivity1();
}
void gates::timingActivity2() {

    o1->eventTime = calculateEventTime(lastV      o1->value,
        i1->eventTime, i2->eventTime, gateDelay,      Event);

    o1->activityCount = i1->activityCount + i2->activityC
        ((lastValue == o1->value) ? 0 : 1);

    lastEvent = o1->eventTime;
    lastValue = o1->value;
}
void gates::timingActivity1() {

    o1->eventTime = calculateEventTime(lastValue, o1->value
        i1->eventTime, gateDelay, lastEvent);

    o1->activityCount = i1->activityCount + ((la      ue == o1->value)?0:1);

    lastEvent = o1->eventTime;
    lastValue = o1->value;
}
int gates::numberOfGates=1;

float getProb(gates* GATE){
    return GATE->outputControlability;
}

void and::evl() {

    if ((i1->value == '0') || (i2->value == '0'))
```

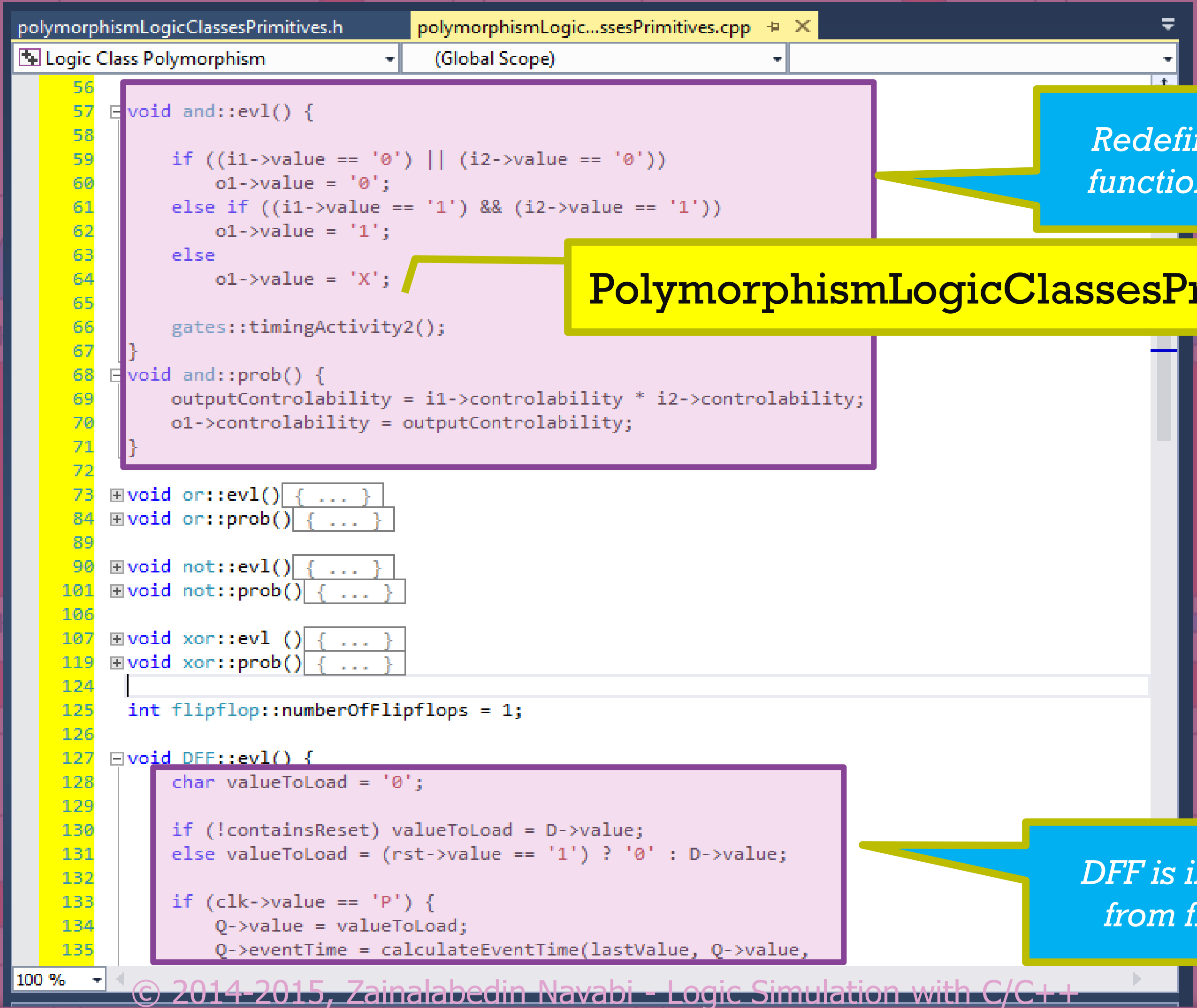
100 %

Like a one input buffer

PolymorphismLogicClassesPrimitives.cpp

Static initialization must be done as member functions are defined

Polymorphic Gate Base



polymorphismLogicClassesPrimitives.h
polymorphismLogic...ssesPrimitives.cpp
Logic Class Polymorphism
(Global Scope)
100 %

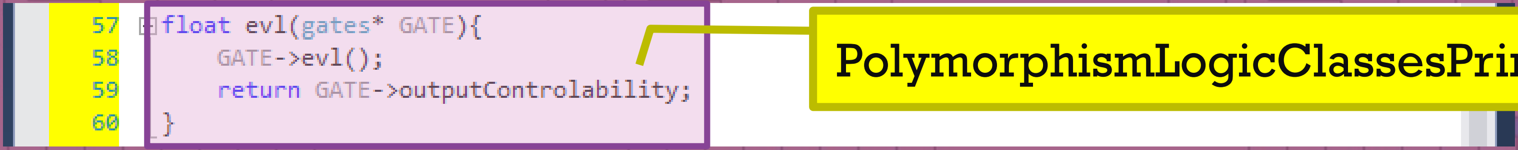
```
void and::evl() {

    if ((i1->value == '0') || (i2->value == '0'))
        o1->value = '0';
    else if ((i1->value == '1') && (i2->value == '1'))
        o1->value = '1';
    else
        o1->value = 'X';

    gates::timingActivity2();
}
void and::prob() {
    outputControlability = i1->controlability * i2->controlability;
    o1->controlability = outputControlability;
}

void or::evl() { ... }
void or::prob() { ... }

void not::evl() { ... }
void not::prob() { ... }

void xor::evl () { ... }
void xor::prob() { ... }

int flipflop::numberOfFlipflops = 1;

void DFF::evl() {
    char valueToLoad = '0';

    if (!containsReset) valueToLoad = D->value;
    else valueToLoad = (rst->value == '1') ? '0' : D->value;

    if (clk->value == 'P') {
        Q->value = valueToLoad;
        Q->eventTime = calculateEventTime(lastValue, Q->value,
```

Redefine virtual functions of gate

PolymorphismLogicClassesPrimitives.cpp

DFF is inherited from flip flop

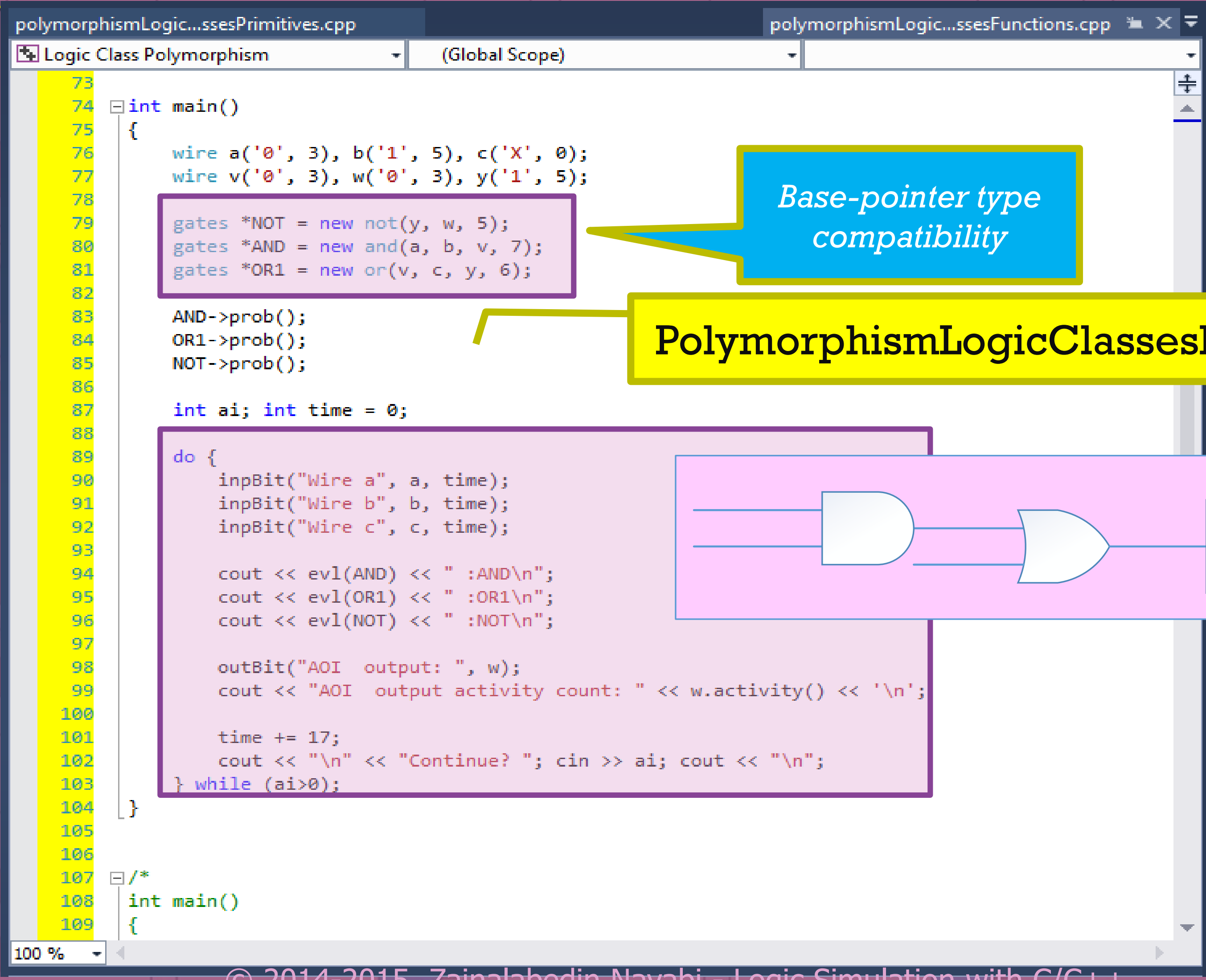
Polymorphic Gate Base



```
float evl(gates* GATE){
    GATE->evl();
    return GATE->outputControlability;
}
```

PolymorphismLogicClassesPrimitives.cpp

Polymorphic Gate Base



polymorphismLogic...ssesPrimitives.cpp

polymorphismLogic...ssesFunctions.cpp

Logic Class Polymorphism

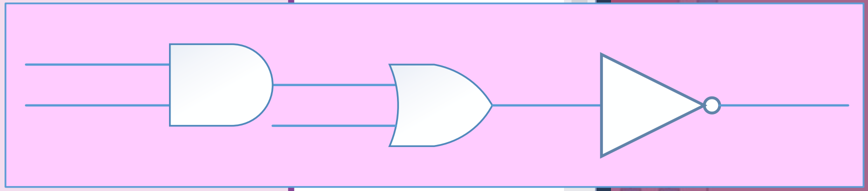
(Global Scope)

```
int main()
{
    wire a('0', 3), b('1', 5), c('X', 0);
    wire v('0', 3), w('0', 3), y('1', 5);

    gates *NOT = new not(y, w, 5);
    gates *AND = new and(a, b, v, 7);
    gates *OR1 = new or(v, c, y, 6);

    AND->prob();
    OR1->prob();
    NOT->prob();

    int ai; int time = 0;

    do {
        inpBit("Wire a", a, time);
        inpBit("Wire b", b, time);
        inpBit("Wire c", c, time);

        cout << evl(AND) << " :AND\n";
        cout << evl(OR1) << " :OR1\n";
        cout << evl(NOT) << " :NOT\n";

        outBit("AOI  output: ", w);
        cout << "AOI  output activity count: " << w.activity() << '\n';

        time += 17;
        cout << "\n" << "Continue? "; cin >> ai; cout << "\n";
    } while (ai>0);
}


/*
int main()
{
```

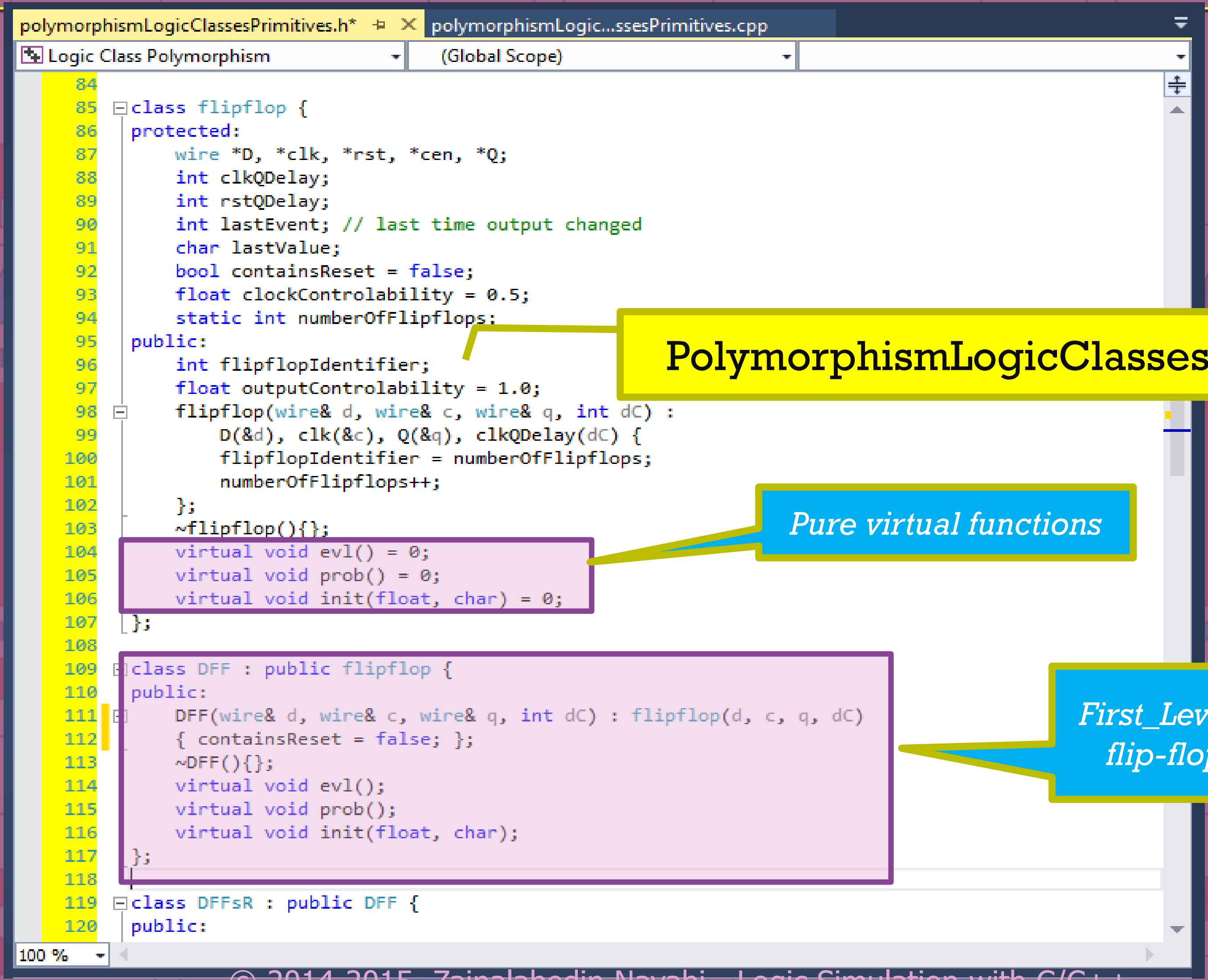
100 %

Base-pointer type compatibility

PolymorphismLogicClassesFunctions.cpp



Polymorphic Gate Base



polymorphismLogicClassesPrimitives.h*
polymorphismLogic...ssesPrimitives.cpp
Logic Class Polymorphism
(Global Scope)
100 %

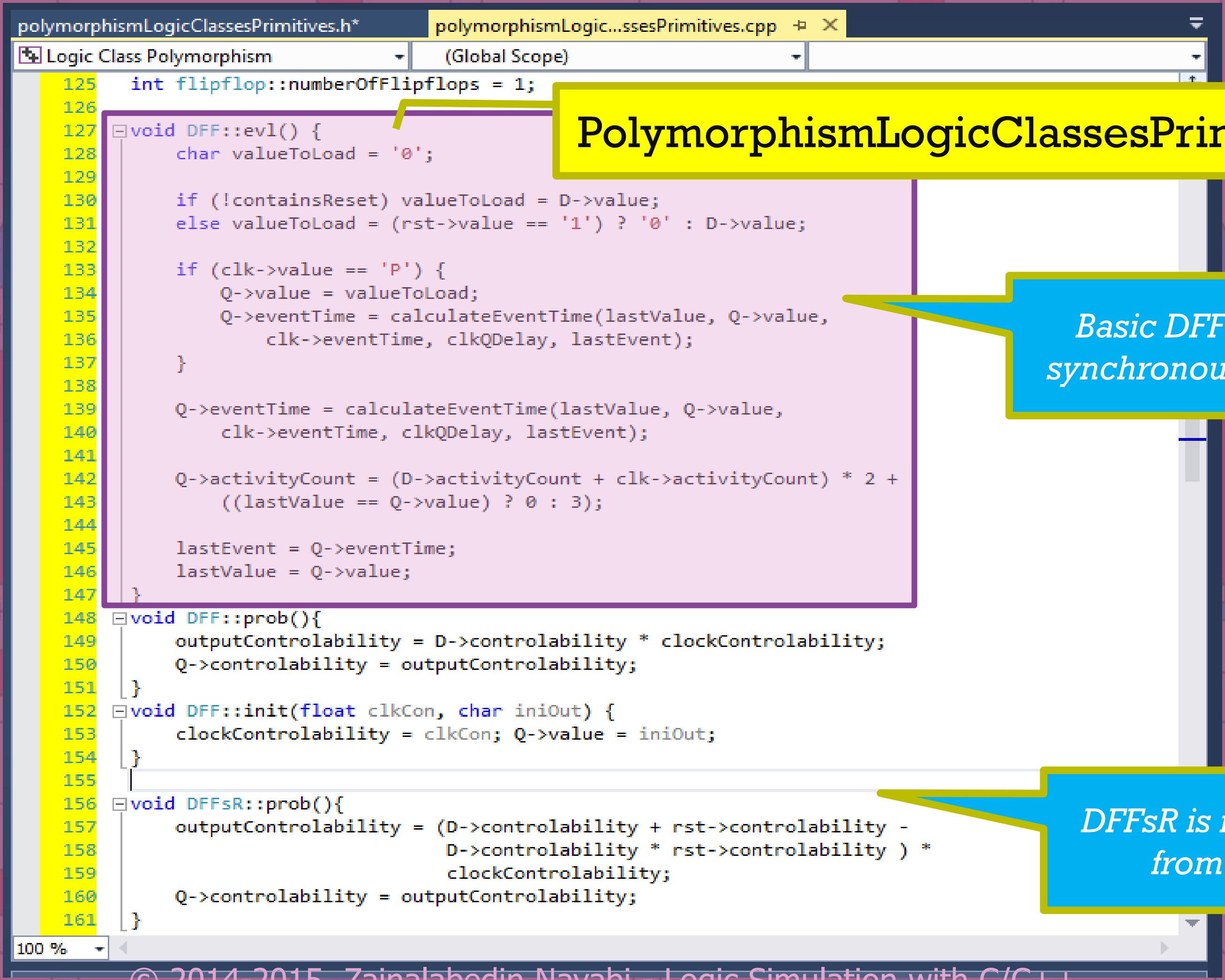
```
class flipflop {
protected:
    wire *D, *clk, *rst, *cen, *Q;
    int clkQDelay;
    int rstQDelay;
    int lastEvent; // last time output changed
    char lastValue;
    bool containsReset = false;
    float clockControlability = 0.5;
    static int numberOfFlipflops;
public:
    int flipflopIdentifier;
    float outputControlability = 1.0;
    flipflop(wire& d, wire& c, wire& q, int dC) :
        D(&d), clk(&c), Q(&q), clkQDelay(dC) {
        flipflopIdentifier = numberOfFlipflops;
        numberOfFlipflops++;
    };
    ~flipflop(){};
    virtual void evl() = 0;
    virtual void prob() = 0;
    virtual void init(float, char) = 0;
};

class DFF : public flipflop {
public:
    DFF(wire& d, wire& c, wire& q, int dC) : flipflop(d, c, q, dC)
    { containsReset = false; };
    ~DFF(){};
    virtual void evl();
    virtual void prob();
    virtual void init(float, char);
};

class DFFsR : public DFF {
public:
```

PolymorphismLogicClassesPrimitives.h

Pure virtual functions

First_Level Derived flip-flop classes

Flip Flop Description Hierarchies



polymorphismLogicClassesPrimitives.h* | polymorphismLogic...ssesPrimitives.cpp

Logic Class Polymorphism | (Global Scope)

```
int flipflop::numberOfFlipflops = 1;

void DFF::evl() {
    char valueToLoad = '0';

    if (!containsReset) valueToLoad = D->value;
    else valueToLoad = (rst->value == '1') ? '0' : D->value;

    if (clk->value == 'P') {
        Q->value = valueToLoad;
        Q->eventTime = calculateEventTime(lastValue, Q->value,
            clk->eventTime, clkQDelay, lastEvent);
    }

    Q->eventTime = calculateEventTime(lastValue, Q->value,
        clk->eventTime, clkQDelay, lastEvent);

    Q->activityCount = (D->activityCount + clk->activityCount) * 2 +
        ((lastValue == Q->value) ? 0 : 3);

    lastEvent = Q->eventTime;
    lastValue = Q->value;
}
void DFF::prob(){
    outputControlability = D->controlability * clockControlability;
    Q->controlability = outputControlability;
}
void DFF::init(float clkCon, char iniOut) {
    clockControlability = clkCon; Q->value = iniOut;
}

void DFFsR::prob(){
    outputControlability = (D->controlability + rst->controlability -
                            D->controlability * rst->controlability ) *
                            clockControlability;
    Q->controlability = outputControlability;
}
```

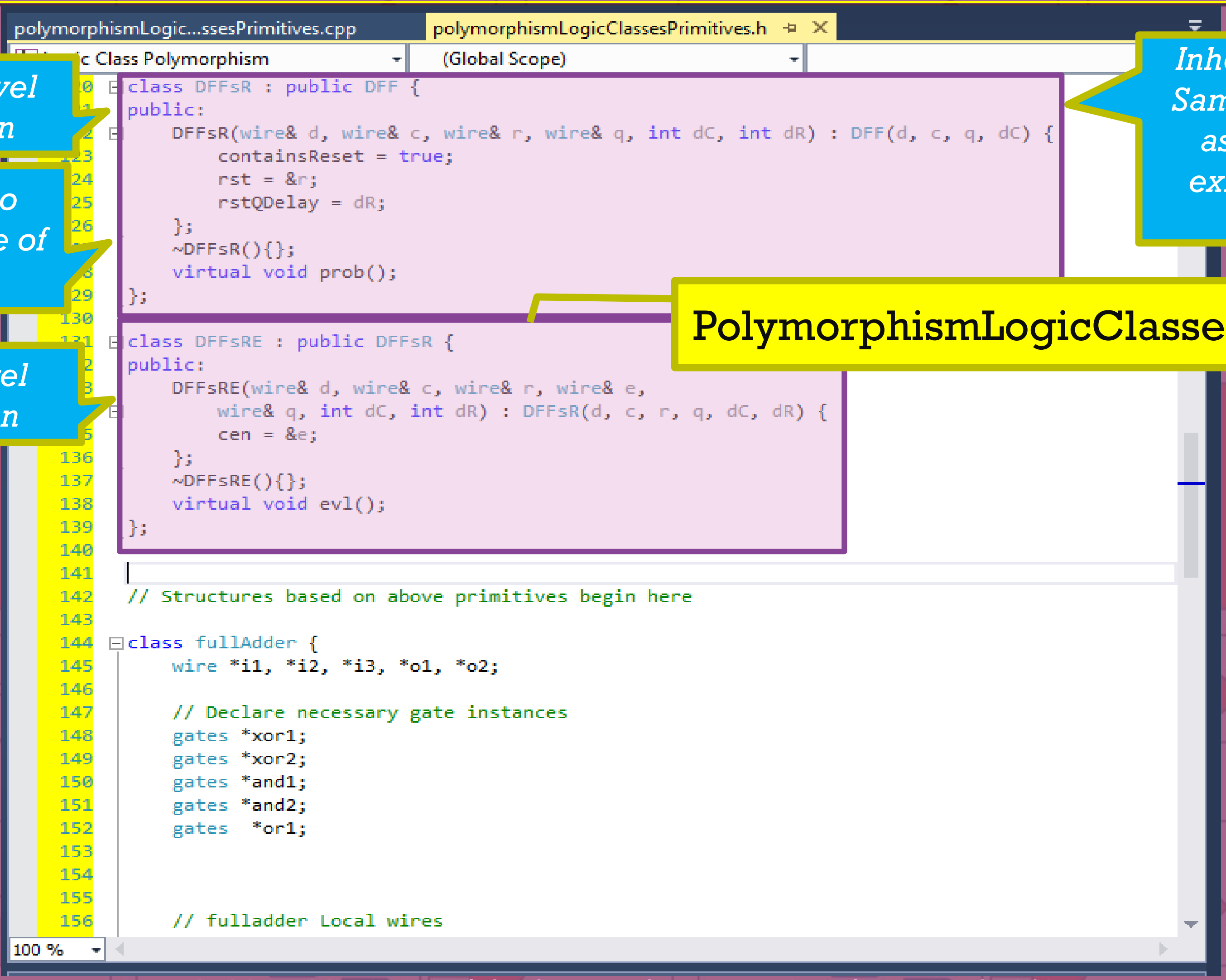
100 %

PolymorphismLogicClassesPrimitives.cpp

Basic DFF with synchronous reset

DFFsR is inherited from DFF

Flip Flop Description Hierarchies


polymorphismLogic...ssesPrimitives.cpp
polymorphismLogicClassesPrimitives.h
...ic Class Polymorphism
(Global Scope)
100 %

```
class DFFsR : public DFF {
public:
    DFFsR(wire& d, wire& c, wire& r, wire& q, int dC, int dR) : DFF(d, c, q, dC) {
        containsReset = true;
        rst = &r;
        rstQDelay = dR;
    };
    ~DFFsR(){};
    virtual void prob();
};

class DFFsRE : public DFFsR {
public:
    DFFsRE(wire& d, wire& c, wire& r, wire& e,
        wire& q, int dC, int dR) : DFFsR(d, c, r, q, dC, dR) {
        cen = &e;
    };
    ~DFFsRE(){};
    virtual void evl();
};

// Structures based on above primitives begin here

class fullAdder {
    wire *i1, *i2, *i3, *o1, *o2;

    // Declare necessary gate instances
    gates *xor1;
    gates *xor2;
    gates *and1;
    gates *and2;
    gates  *or1;



    // fulladder Local wires
```

Second Level Derivation

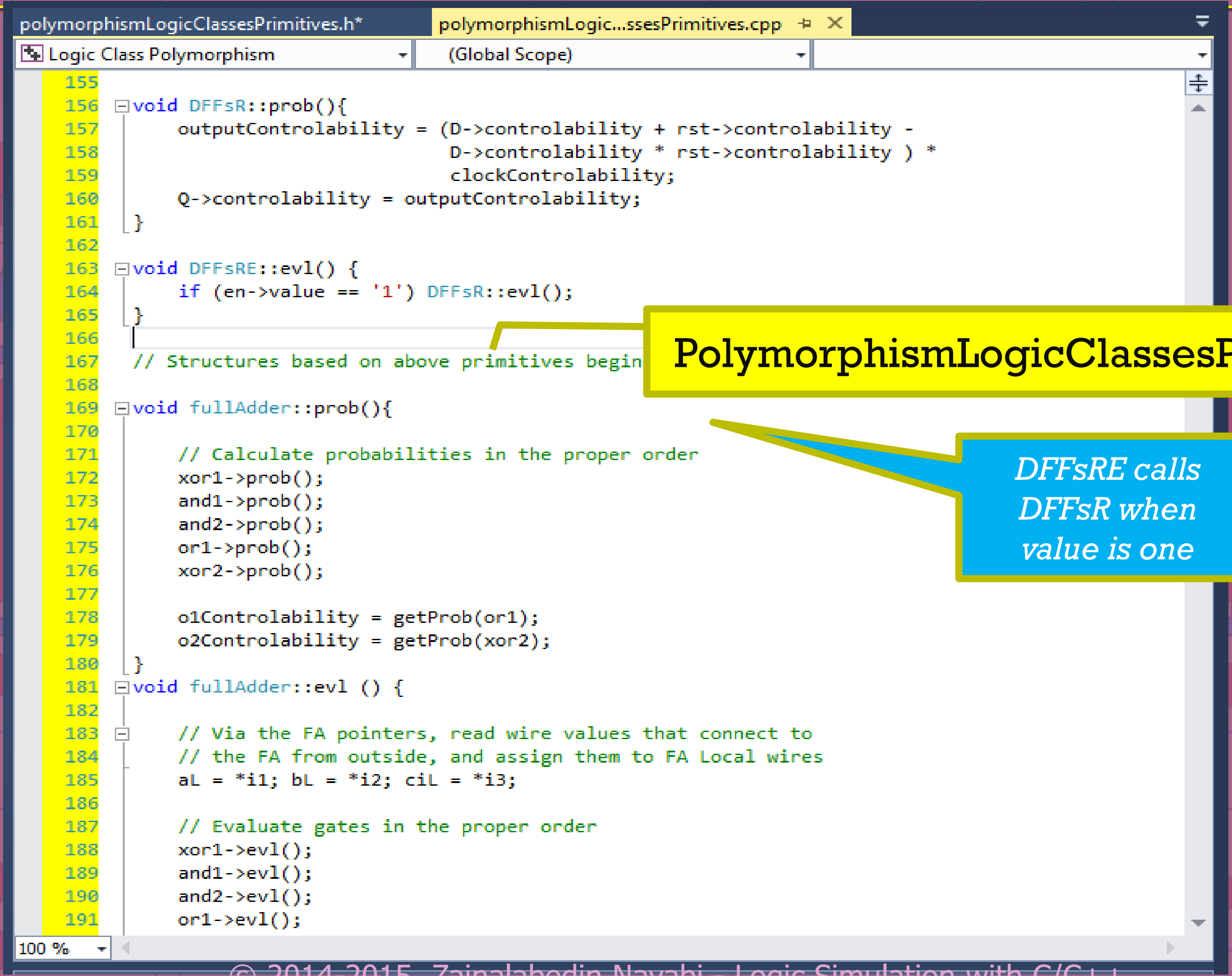
No evl(), so uses the one of DFF

Third Level derivation

Inherited from DFF. Same members but assigns value to existing rst of flip flop

PolymorphismLogicClassesPrimitives.h

Flip Flop Description Hierarchal



polymorphismLogicClassesPrimitives.h* | polymorphismLogic...ssesPrimitives.cpp

Logic Class Polymorphism | (Global Scope)

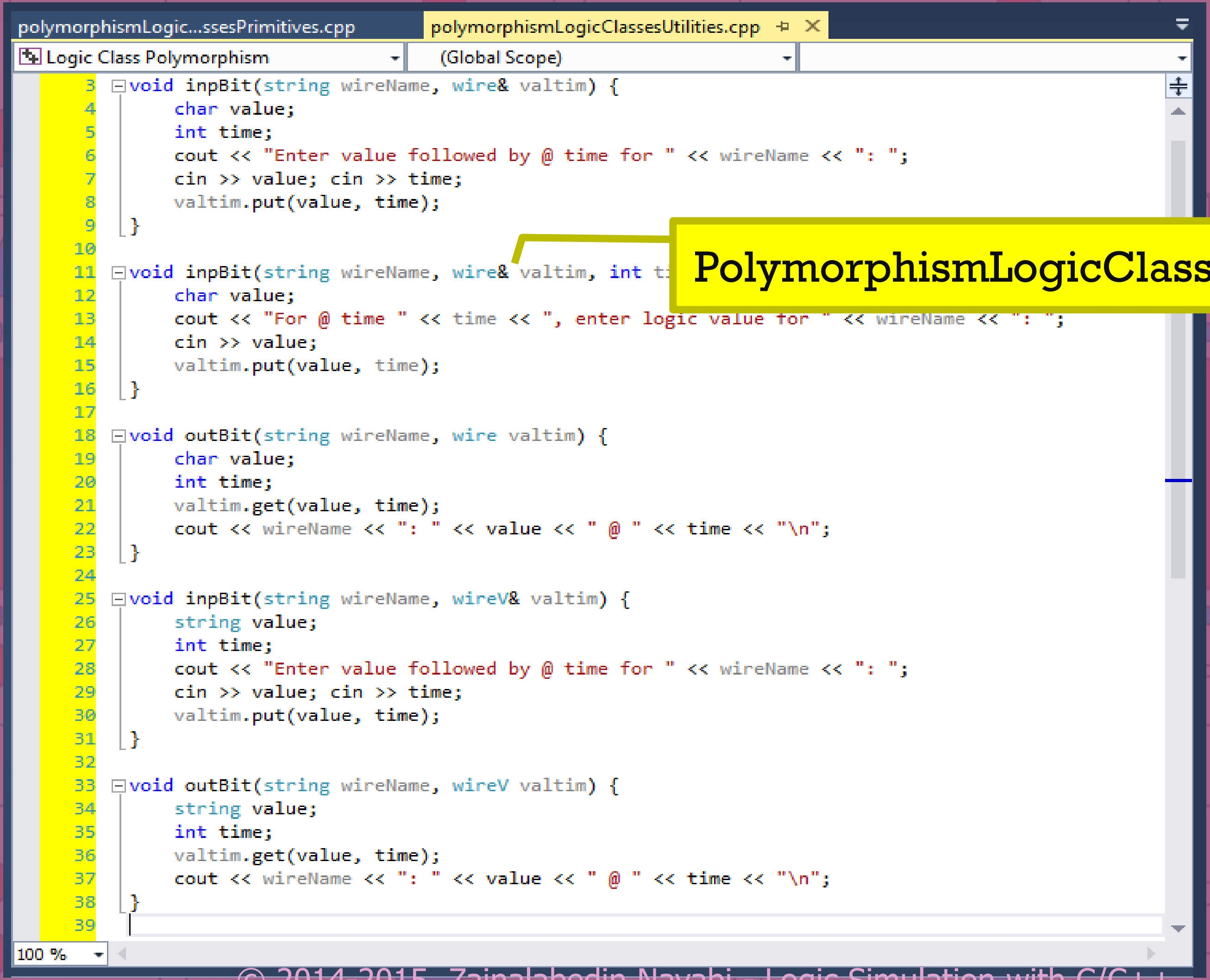
```
void DFFsR::prob(){
    outputControlability = (D->controlability + rst->controlability -
                            D->controlability * rst->controlability ) *
                            clockControlability;
    Q->controlability = outputControlability;
}

void DFFsRE::evl() {
    if (en->value == '1') DFFsR::evl();
}

// Structures based on above primitives begin

void fullAdder::prob(){

    // Calculate probabilities in the proper order
    xor1->prob();
    and1->prob();
    and2->prob();
    or1->prob();
    xor2->prob();

    o1Controlability = getProb(or1);
    o2Controlability = getProb(xor2);
}
void fullAdder::evl () {

    // Via the FA pointers, read wire values that connect to
    // the FA from outside, and assign them to FA Local wires
    aL = *i1; bL = *i2; ciL = *i3;

    // Evaluate gates in the proper order
    xor1->evl();
    and1->evl();
    and2->evl();
    or1->evl();
```

100 %

PolymorphismLogicClassesPrimitives.cpp

DFFsRE calls DFFsR when value is one

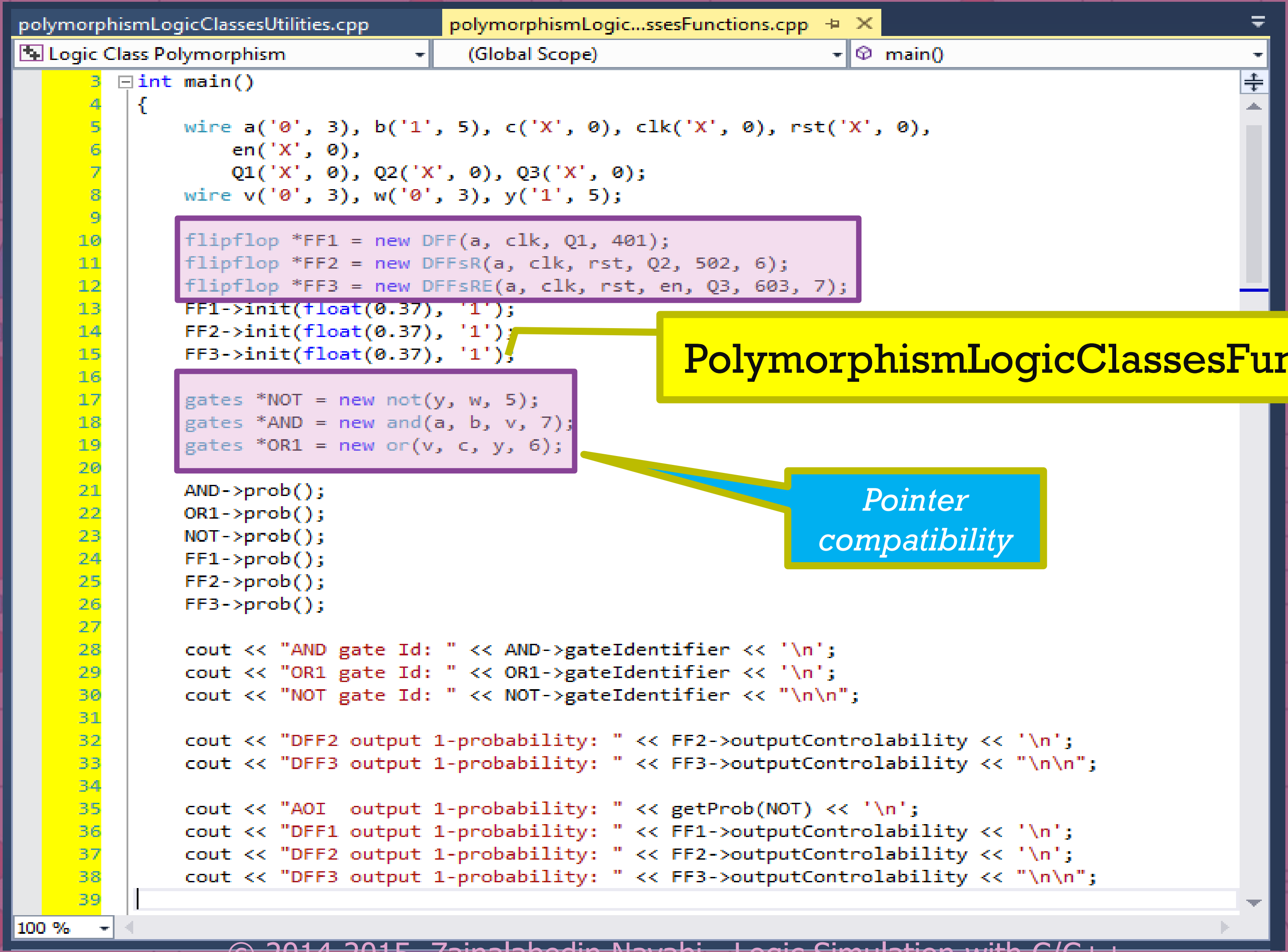
Flip Flop Description Hierarchal


polymorphismLogic...ssesPrimitives.cpp
polymorphismLogicClassesUtilities.cpp
Logic Class Polymorphism
(Global Scope)
100 %

```cpp
void inpBit(string wireName, wire& valtim) {
    char value;
    int time;
    cout << "Enter value followed by @ time for " << wireName << ": ";
    cin >> value; cin >> time;
    valtim.put(value, time);
}

void inpBit(string wireName, wire& valtim, int ti
    char value;
    cout << "For @ time " << time << ", enter logic value for " << wireName << ": ";
    cin >> value;
    valtim.put(value, time);
}

void outBit(string wireName, wire valtim) {
    char value;
    int time;
    valtim.get(value, time);
    cout << wireName << ": " << value << " @ " << time << "\n";
}

void inpBit(string wireName, wireV& valtim) {
    string value;
    int time;
    cout << "Enter value followed by @ time for " << wireName << ": ";
    cin >> value; cin >> time;
    valtim.put(value, time);
}

void outBit(string wireName, wireV valtim) {
    string value;
    int time;
    valtim.get(value, time);
    cout << wireName << ": " << value << " @ " << time << "\n";
}
```

PolymorphismLogicClassesPrimitives.cpp

Flip Flop Description Hierarchal



polymorphismLogicClassesUtilities.cpp | polymorphismLogic...ssesFunctions.cpp

Logic Class Polymorphism | (Global Scope) | main()

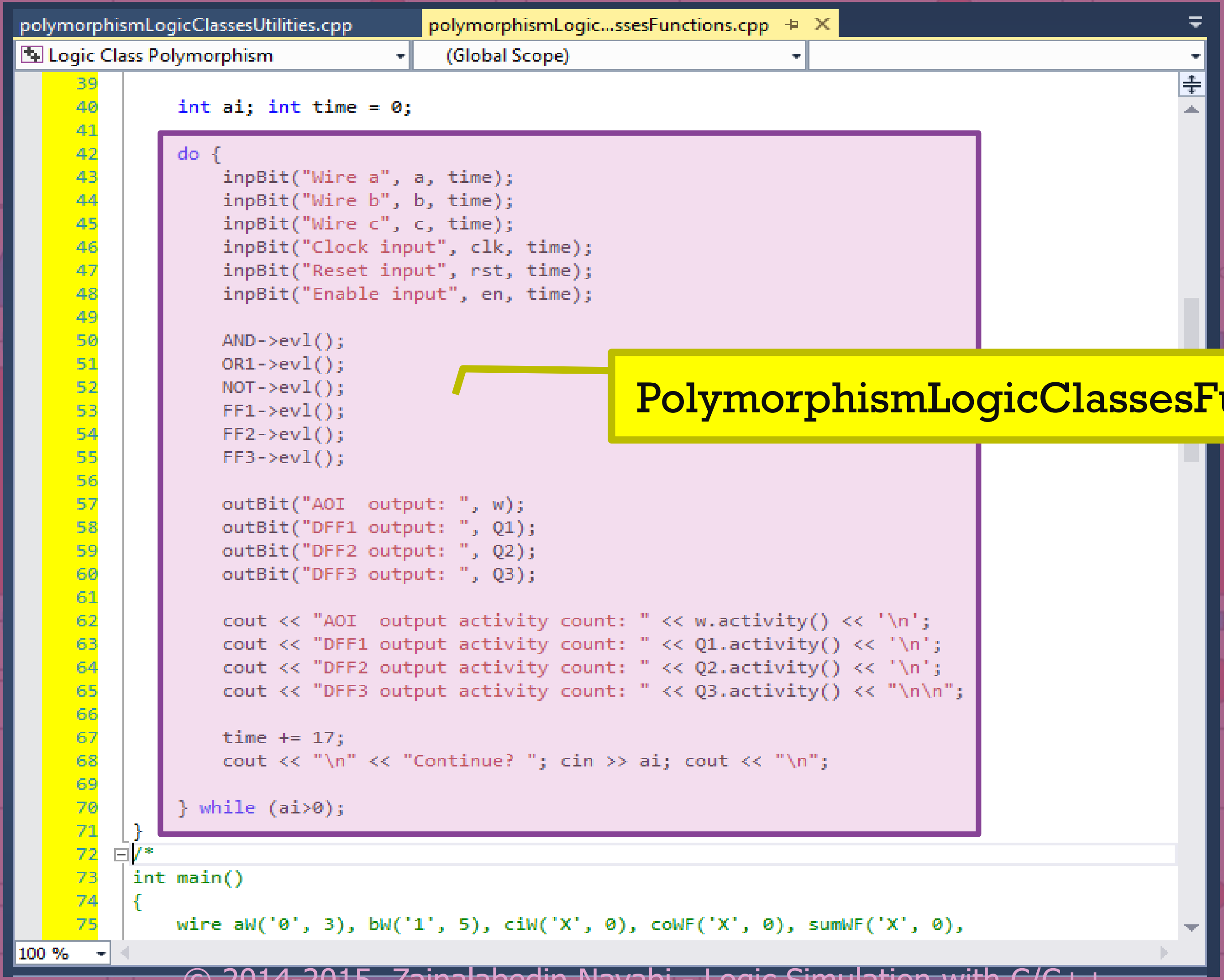
```
int main()
{
    wire a('0', 3), b('1', 5), c('X', 0), clk('X', 0), rst('X', 0),
        en('X', 0),
        Q1('X', 0), Q2('X', 0), Q3('X', 0);
    wire v('0', 3), w('0', 3), y('1', 5);

    flipflop *FF1 = new DFF(a, clk, Q1, 401);
    flipflop *FF2 = new DFFsR(a, clk, rst, Q2, 502, 6);
    flipflop *FF3 = new DFFsRE(a, clk, rst, en, Q3, 603, 7);
    FF1->init(float(0.37), '1');
    FF2->init(float(0.37), '1');
    FF3->init(float(0.37), '1');

    gates *NOT = new not(y, w, 5);
    gates *AND = new and(a, b, v, 7);
    gates *OR1 = new or(v, c, y, 6);

    AND->prob();
    OR1->prob();
    NOT->prob();
    FF1->prob();
    FF2->prob();
    FF3->prob();

    cout << "AND gate Id: " << AND->gateIdentifier << '\n';
    cout << "OR1 gate Id: " << OR1->gateIdentifier << '\n';
    cout << "NOT gate Id: " << NOT->gateIdentifier << "\n\n";

    cout << "DFF2 output 1-probability: " << FF2->outputControlability << '\n';
    cout << "DFF3 output 1-probability: " << FF3->outputControlability << "\n\n";

    cout << "AOI  output 1-probability: " << getProb(NOT) << '\n';
    cout << "DFF1 output 1-probability: " << FF1->outputControlability << '\n';
    cout << "DFF2 output 1-probability: " << FF2->outputControlability << '\n';
    cout << "DFF3 output 1-probability: " << FF3->outputControlability << "\n\n";
```

100 %

PolymorphismLogicClassesFunctions.cpp

Pointer compatibility

Flip Flop Description Hierarchal


polymorphismLogicClassesUtilities.cpp
polymorphismLogic...ssesFunctions.cpp
Logic Class Polymorphism
(Global Scope)
100 %

```

    int ai; int time = 0;

    do {
        inpBit("Wire a", a, time);
        inpBit("Wire b", b, time);
        inpBit("Wire c", c, time);
        inpBit("Clock input", clk, time);
        inpBit("Reset input", rst, time);
        inpBit("Enable input", en, time);

        AND->evl();
        OR1->evl();
        NOT->evl();
        FF1->evl();
        FF2->evl();
        FF3->evl();

        outBit("AOI  output: ", w);
        outBit("DFF1 output: ", Q1);
        outBit("DFF2 output: ", Q2);
        outBit("DFF3 output: ", Q3);

        cout << "AOI  output activity count: " << w.activity() << '\n';
        cout << "DFF1 output activity count: " << Q1.activity() << '\n';
        cout << "DFF2 output activity count: " << Q2.activity() << '\n';
        cout << "DFF3 output activity count: " << Q3.activity() << "\n\n";

        time += 17;
        cout << "\n" << "Continue? "; cin >> ai; cout << "\n";

    } while (ai>0);
}
/*
int main()
{
    wire aW('0', 3), bW('1', 5), ciW('X', 0), coWF('X', 0), sumWF('X', 0),
```

PolymorphismLogicClassesFunctions.cpp

Conclusion

This chapter presented:

- Procedural Languages for Hardware Modeling
- Types and Operators for Logic Modeling
- Basic Logic Simulation
- Enhanced logic simulation with timing
- More Functions for Wires and Gates
- Inheritance in Logic Structures
- Hierarchal Modeling of Digital Components

Copyright and Acknowledgment

- Slides prepared by Hanieh Hashemi, ECE graduate student