

CSI333 Lecture 2

Goals-Know how to work things and how things work-Learn to Think productively, creatively, “out of the box”

1. Machine/Assembly and C Languages under Unix: Instruction Set Architecture is instructions & datatypes of computer **hardware**. C related to machine features, comprehensive view of C.
2. C/C++/Java hardware/software/system interface topics: type implementation, modularization with multiple source file programs, functions, pointers, strings, arguments, preprocessor, etc.
3. Computer performance (i.e., **speed** of a program's execution and it's observation).
4. Larger, more mature projects and software structures. Tools (make, debugger, binary editors and object analyzers, version control repositories).
5. See Patt/Patel (PP, BLUE hardcover :(book) and Bryant/O'Hallaron) and writings of top SW developers for detailed motivation.

Project 1

Your job is to understand the algorithm based on these ideas and then implement it (for Boolean, not numeric expressions) by a C++ or Java program. HOMEWORK: practice it on paper.

Observations:

1. The value or operator accessed or removed from the corresponding “box” is ALWAYS the LAST ONE put in the box but not yet removed. Those boxes can be *stacks*, i.e., pushdown or Last-In-First-Out (LIFO) stores.
2. The input is processed left-to-right.
3. The decision to PUSH, or REDUCE, or REMOVE a “(” is based only on the current input token and the top of the Operator stack.
4. When two operators affect the PUSH or REDUCE decision, the decision is based on their relative *precedence* (and associativity).

Associativity

Associativity where it really matters to the IRS and bosses:

$$\text{Salary} - \text{Deduction1} - \text{Deduction2}$$

means

$$(\text{Salary} - \text{Deduction1}) - \text{Deduction2}$$

NOT

$$\begin{aligned} &\text{Salary} - (\text{Deduction1} - \text{Deduction2}) \\ &= (\text{Salary} - \text{Deduction1}) + \text{Deduction2} \end{aligned}$$

(The IRS receives deductions and bosses don't pay them to you!)
Subtraction denoted by $-$ is a *binary (Here, 2 operand) LEFT* associative operator.

In Project 1, operators $\&$ and $|$ are left-associative.

Precedence

$\text{Price1} + \text{Price2} * \text{DiscountFactor2}$

$\text{BoolOne} | \text{BoolTwo} \& \text{BoolThree}$

mean

$\text{Price1} + (\text{Price2} * \text{DiscountFactor2})$

$\text{BoolOne} | (\text{BoolTwo} \& \text{BoolThree})$

NOT

$(\text{Price}-1 + \text{Price}-2) * \text{Discount}-2$

$(\text{Bool}-1 | (\text{Bool}-2) \& \text{Bool}-3$

It has been a social convention among nerds for hundreds of years that multiplication and division beat addition and subtraction. Precedence and associativity are needed to decide between PUSH or REDUCE when there are 2 operators involved. You figure it out and debug it.

It is really a CS1-CS2 Review:

1. Implement and use two stacks.
(Stacks recur over and over in this course...how function linkage works is a topic.)
2. Organize the program so the BoolExpression class holds the method to do the evaluation. (Computer hardware is built of gates that do logical, i.e., *Boolean* operations on bits.)
3. Implement and use a “Dictionary” –data structure that stores
(1) some of the letters from A-Z and (2) for each letter stored, a Boolean (truth) value “true” or “false”. Also, implement operations to
 - 3.1 Print the row of 0’s and 1’s for the letters in alphabetical order.
 - 3.2 Calculate the next row of 0’s and 1’s, for example:
ABC
000
001
010
011 etc
 - 3.3 Find the current Boolean value for a given letter whenever evaluate() needs it.

Getting started.. C++ choice

Class definition in BoolExpression.h

```
#ifndef BoolExpression_h_included
#define BoolExpression_h_included
#include <string>
#include "charStack.h" //YOU code these .h files
#include "boolStack.h" //YOU code these .h files
//#include "symbolTable.h" //Procrastinate here!
//Handle expressions with 0's and 1's BUT NO LETTERS first
class BoolExpression
{
public:
    //public means the body of ANY KIND of function
    // can access or call these members.
    ... to be continued
```

public members of class BoolExpression

```
public:  
    BoolExpression(); //Default constructor inputs expression  
  
    //Be simple (gift): exit or throw exception on failure.  
    //Constr. might initialize stacks and the symbol table.  
  
    //Extendable: OVERLOAD with another constructor  
    // which has a string parameter.  
    // Good idea???  
  
    void printTruthTable(); //Achieve application goal.
```

To be continued...

BoolExpression.h Continued

```
private: //means ONLY BoolExpression's methods can
        //access or call these.

std::string expr; //need std:: since using namespace
                //is BAD in header files.
//Hold the expression.
// i-th char is expr[i]. 0 <= i <= expr.length()-1
// end char is '\0'
//use getline(cin, expr) to read it.

bool evaluate( ); //get value for ONE TRUTH TABLE LINE
// body has the code for the algorithm
// outlined in the assignment.
```

Private part of class BoolExpression cont'd ...

```
charStack *pOpStack; //use new to allocate the
boolStack *pValStack; //stack objects we point to
//symbolTable *pletterValueTable;
//helpers...
bool valAtom( char x);
bool isAtom( char x);
void reduce( ); //The famous reduce step..
// pop 1 operator, 1-2 values, calculate and push result
bool figure_out_precedence( char x, char y);
// void fill_symbol_table(); //scan expr and put the let
// ??? whatever else you need.
}; //REMEMBER THE ; at the end of a class definition
#endif //match the #ifndef include guard
```

Put the main function in BoolExpression.cpp

```
// file BoolExpression.cpp
#include "BoolExpression.h"
#include <iostream> //need for cin, cout, getline, etc.
#include <string>
using namespace std; //use usings in .cpp files only, not in .h files
int main(int argc, char *argv[]) //command line args NOT USED
{
    BoolExpression myBoolExpression; //constructor reads it in
    // alternative, Java-style:
    // BoolExpression *pmyBoolExpression = new BoolExpression(argv[1]);
    myBoolExpression.printTruthTable();
    // alternative:
    // pmyBoolExpression->printTruthTable();
    return 0; //success return code.
}
```

```
BoolExpression::BoolExpression() //default constructor
{
```

Programming Project 2

- ▶ Start using C on Unix.
- ▶ Input 32-bit words from the user in several different ways.
- ▶ See those bits in two common ways: Hexadecimal and the bits as 0s and 1s.
- ▶ Output (i.e., interpret) the 32-bit words in several different ways.

Loop until commanded to exit:

1. Print a menu and receive an input choice, or command to use currently stored data.
2. (exit, or) Run an input operation according to that choice.
3. Print the input bits.
4. Print a menu and receive an output choice.
5. Run an output operation according to that choice.

Lab this week

Instructions and help material are distributed ... Labs this week cover:

1. Network Knowledge: Real architecture of the elements UAlbany's part of the Internet which you will use in this course. Where are your UAlbany files?
2. How you MUST use MULTIPLE windows on UNIX (with an X-server) and how they work.
3. Getting started with writing, running and saving C programs under UNIX.

Using UAlbany Windows PCs: (copy, print or memorize! FOLLOW THESE EXACTLY for current CSI333 work)

1. Ctrl-Alt-Delete (3-finger salute) and log in with Net-Id and password (BRING THEM TO THE LAB).
2. Dismiss or minimize the Web browser.
3. From bottom of START menu, select XWin32/Step 1: Start X Server.
4. From bottom of START menu, select Step 2: ssh to itsunix.
5. (Make sure X tunnelling is enabled in the ssh client software.)

Bit - Binary Digit

Basic Unit of Information stored and manipulated in our computers.

Computer ¹ hardware stores & manipulates & transmits all data **digitally**: which means with on/off, 0/1, **true/false** (usually) electrical² signals.

A bit (binary digit) is a single unit of memory or transmission that can have only 2 possible values.

¹and much other popular electronic product

²also optical and magnetic

A representation is a coding scheme to give meaning to bit strings (sequences).

Different kinds of data are each represented by different meanings we give to sequences of bits.

Some hardware (printers, keyboards e.g.) embodies particular representations: 01000010 makes standard printers print **B**

The base or radix 2 (binary) number system is used by present computers to represent non-negative integers.

Example: 0001 1100 1000 0110

$$\begin{aligned} &0 \cdot 2^{15} + 0 \cdot 2^{14} + 0 \cdot 2^{13} + 1 \cdot 2^{12} \\ &+ 1 \cdot 2^{11} + 1 \cdot 2^{10} + 0 \cdot 2^9 + 0 \cdot 2^8 \\ &+ 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 \\ &+ 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \end{aligned}$$

$$\begin{aligned} &= (0+0+0+4096)+(2048+1024+0+0)+(256+0+0+0)+(0+4+2+0) \\ &= 7302 \end{aligned}$$

Computer Experts all know their powers of 2:

2^1	2	2^{11}	2048 = 2Ki
2^2	4	2^{12}	4098 = 4Ki
2^3	8	2^{13}	8192 = 8Ki
2^4	16 _{ten} = 0x10	2^{14}	16,384 = 16Ki
2^5	32 _{ten} = 0x20	2^{15}	32,768 = 32Ki
2^6	64 _{ten} = 0x40	2^{16}	65,536 = 64Ki
2^7	128 _{ten} = 0x80	2^8	256
2^9	512	2^{10}	1024 = 1kibi = 1Ki
2^{20}	1,048,576 = 1Mi $\approx 10^6$	2^{30}	1,073,741,824 = 1Gi $\approx 10^9$

“Thus 1024 bytes of storage is officially a kibibyte, not a kilobyte. However, computer professionals generally dislike this unit (they say it sounds like a cat food) so the ambiguity in the size of a kilobyte persists. The prefix is a contraction of “kilobinary.” The symbol Ki-, rather than ki-, was chosen for uniformity with the other binary prefixes (Mi-, Gi-, etc.)” (Russ Rowlett, UNC)

Addition in Decimal

$$\begin{array}{r} 36 \\ 87 \\ \hline \end{array}$$

Addition in Decimal

$$\begin{array}{r} 1 \quad \text{carries} \\ 36 \\ 87 \\ \hline 3 \end{array}$$

Addition in Decimal

$$\begin{array}{r} 11 \text{ carries} \\ 36 \\ 87 \\ \hline 23 \end{array}$$

Addition in Decimal

$$\begin{array}{r} 11 \text{ carries} \\ 36 \\ 87 \\ \hline 123 \end{array}$$

Addition in Binary: Single digits

$$\begin{array}{r} 0 \\ 0 \\ \hline 0 \end{array} \quad \begin{array}{r} 0 \\ 1 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ 0 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ 1 \\ \hline 1 \ 0 = 2 \end{array}$$

$$\begin{array}{r} 0 \\ 0 \\ 0 \\ \hline 0 \end{array} \quad \begin{array}{r} 0 \\ 0 \\ 1 \\ \hline 1 \end{array} \quad \begin{array}{r} 0 \\ 1 \\ 0 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ 0 \\ 0 \\ \hline 1 \end{array} \quad \begin{array}{r} 0 \\ 1 \\ 1 \\ \hline 1 \ 0 \end{array} \quad \begin{array}{r} 1 \\ 0 \\ 1 \\ \hline 1 \ 0 \end{array} \quad \begin{array}{r} 1 \\ 1 \\ 0 \\ \hline 1 \ 0 \end{array}$$

$$\begin{array}{r} 1 \\ 1 \\ 1 \\ \hline 1 \ 1 = 3 \end{array}$$

way!

Mathematics requires a binary computer to work this

Express these rules with LOGIC

0	0	1	1	A
0	1	0	1	B
<hr/>				<hr/>
0 0	0 1	0 1	1 0	Carry Sum

Explain logically how A and B determine Sum and $Carry$?

Express these rules with LOGIC

0	0	1	1	<i>A</i>
0	1	0	1	<i>B</i>
<hr/>				<hr/>
0 0	0 1	0 1	1 0	<i>Carry</i> <i>Sum</i>

Explain logically how *A* and *B* determine *Sum* and *Carry*?

Sum = 1 exactly when (*A* = 1 and *B* = 0) or (*A* = 0 and *B* = 1).

Express these rules with LOGIC

0	0	1	1	<i>A</i>
0	1	0	1	<i>B</i>
<hr/>				<hr/>
0 0	0 1	0 1	1 0	<i>Carry Sum</i>

Explain logically how *A* and *B* determine *Sum* and *Carry*?

Sum = 1 exactly when (*A* = 1 and *B* = 0) or (*A* = 0 and *B* = 1).

Carry = 1 exactly when (*A* = 1 and *B* = 1).

Express these rules with LOGIC

0	0	1	1	<i>A</i>
0	1	0	1	<i>B</i>
<hr/>				<i>Carry</i> <i>Sum</i>
0 0	0 1	0 1	1 0	

Explain logically how *A* and *B* determine *Sum* and *Carry*?

Sum = 1 exactly when (*A* = 1 and *B* = 0) or (*A* = 0 and *B* = 1).

Carry = 1 exactly when (*A* = 1 and *B* = 1).

The electronic *devices* called *logic gates* determine output signals like *Carry* from inputs like *A* and *B*. Typical computation time:

0.1 nanosecond = 10^{-10} second.

3 Basic Gates: AND, OR, and NOT

Truth Table for the AND gate (and AND Boolean Operation):

A	B	$A \text{ AND } B = A \& B$
0	0	0
0	1	0
1	0	0
1	1	1

Other useful gates and Boolean Operations:

X	Y	$X \text{ OR } Y = X Y$
0	0	0
0	1	1
1	0	1
1	1	1

(our OR is *Inclusive*)

W	$\text{NOT } W = \sim W$
0	1
1	0

Some Boolean Expressions

A	B	$\sim B$	$(A \& \sim B)$	$\sim A$	$(\sim A \& B)$	$(A \& \sim B) (\sim A \& B)$
0	0	1	0	1	0	0
0	1	0	0	1	1	1
1	0	1	1	0	0	1
1	1	0	0	0	0	0
Two independent Boolean Variables		Intermediate Values				This is <i>Sum</i> !
						Desired Result

```

#include <stdio.h>
unsigned char N = 0; // 8 bits will be used
for( int i=0; i<1000; i++ )
{
    printf("%d ", N );//Print N as decimal integer.
    N = N + 1;
}

```

overflows or wraps around when N= 255.

```

0 1 ... 253 254 255 0 1 2 3 ... 255 0 ...
252 253 254 255 0 1 2 3 ... 229 230 231

```

Something like this **absolutely must happen** since 8 bits can only distinguish $2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 256$ different data values.

```

#include <stdio.h>
signed char N = 0; // 8 bits will be used
for( int i=0; i<1000; i++ )
{
    printf("%d ", N );//Print N as decimal integer.
    N = N + 1;
}

```

N declared signed this time

```

0 1 2 3 ... 126 127 -128 -127 -126 -125 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
... -29 -28 -27 -26 -25

```

The reason for the difference is subtle:

- ▶ Standard C `printf("%d", integer value);`
`%d` format ALWAYS uses SIGNED interpretation.
- ▶ C uses “usual argument conversions” on (8 bit) char types to (16 or 32) bit int since the `printf` function doesn't have declared argument types. `printf` is not type-safe!
- ▶ Bits of unsigned char variables are interpreted in unsigned binary for conversion to int.
- ▶ Bits of signed char variables are interpreted in signed binary for conversion to int.

Method calling

When a (non-static) method is CALLED, it is always called FOR (or ON or WITH) a particular OBJECT.
(That object has type given by the CLASS of the method.)

```
#include "Pile.h"
main()
{
    Pile myPile;
    Pile yourPile;

    myPile.printMe();
    yourPile.makeFullDeck();
    yourPile.printMe();
}
```

```

class Pile
{ ...
  string items[3000];
  int size;
  ...
  void printMe();
  ...
};
void myPile::printMe()
{
  for( int i = 0; i < size; i++ )
  {
    cout << items[i] << endl;
  }
}

```

When the debugger “steps into” a method call, you can view the value of the `this` pointer.

You can view the data in the *object for which the method was called* by DEREFERENCING the `this` pointer; double-click to do this in DDD.

Variables in C/C++/Java

```
int X, Y, Z;  
Z = X + Y*Z;  
if( Z < 0 )  
{  
    Z = 0;  
}
```

X

Y

Z



store values.

Java always initializes its variables, whether or not the programmer codes an initializer.

C/C++ NEVER initialize variables automatically, unless the C++ class has default initializer coded.