

This handout assigns some of the written homework and the two projects for the first two weeks of the course.

The first project is to implement in either C++ or Java an application that evaluates Boolean expressions and prints truth tables. It must use two stacks that you implement yourself, and it must implement the “two-stacks algorithm” that is given in pseudo-code within this handout.

The second project is to write in C under Unix a very simple program so you can begin to learn the standard C input and output facilities, begin to get used to C programming, and observe what happens when binary data is interpreted different ways.

1 Homework Due NEXT CLASS

Hand in your solutions on paper at the beginning of lecture 2, Aug. 30. (Students who late register must hand give me this homework solved in exchange for an SKN number!)

1.1 HW1 Part 1, review of the class concept

Explain two examples of classes that might be useful to define inside (or import into) a program for certain applications. Such examples might be a “Date” class for business applications and a “Circle” class for a graphics system. Also, explain one example of a class from a library or package that is currently standard for C++ or Java. Examples could be a string class, some container class, or an input or output stream class. Each (of the 3 explanations) must include:

1. The name of the class.
2. The nature of the data in an instance of the class.
3. The names, parameters, actions and purposes of a few of the methods of the class.
4. How the class is useful.

If you don't remember or imagine well enough, please do refer to your previous or other textbooks, but try to write the explanations in your own words (except for the names) after putting the books aside.

1.2 HW1 Part 2, review of stacks

You will have to implement and use stacks within Project 1. Explain two implementations of a stack: (1) One implementation uses a “partially filled array,” which is an array together with an index variable that stores the index where the next pushed item should be copied, which is the number of array elements used (0 if the stack is empty). (2) The other implementation uses a singly-linked list.

In each explanation, explain how the pop, push and copy-the-top-value operations are implemented. Also explain how, in EACH implementation, the stack can be made to appear unlimited in size, up to system memory limitations.

Short, clear explanations accompanied by DIAGRAMS are preferred. In fact, a diagram for each is required for full credit.

1.3 HW1 Part 3, have fun learning Hex

Write out the hexadecimal digit to binary conversion table with 16 rows. Invent a strategy for memorizing it. Mine is to memorize A, C and F first. Then:

| | |
|------|------------------------|
| 31e2 | ???????????????? |
| b5ea | ???????????????? |
| f7ca | ???????????????? |
| 39da | ---\ ????????????????? |
| 9ddb | ---/ ????????????????? |
| cec0 | ???????????????? |
| ed5e | ???????????????? |
| 210e | ???????????????? |
| ffff | ???????????????? |

Translate the 9 16-bit words from hexadecimal to binary, and write the bits in a 9-row 8-column array as shown. (Computer graphics is more colorful today. You can create bitmaps yourself with the `bitmap` program on X-window Linux or ITSUNIX.)

1.4 HW1 Part 4, demonstrate an algorithm on paper

Project 1 requires that you implement the so-called 2-stacks algorithm for evaluating a boolean expression. To get started, the homework is to demonstrate that algorithm on paper. The algorithm is described in pseudo-code below.

The input will be a sequence consisting of characters that could be '(' or ')' for parenthesization, '|' for Boolean OR, '&' for Boolean AND, '~' for Boolean NOT, single capital letters (A-Z) for Boolean variables, or '0' or '1' for the Boolean constants FALSE or TRUE respectively.

The purpose of the algorithm is to compute the truth value of the input expression given the truth values of the variables. This truth value can be figured out manually by systematically figuring out the values of larger and larger sub-expressions, as demonstrated in the textbook.

But the 2-stacks *algorithm* is the method outlined below. It is not as convenient to use by hand, but it is much easier and faster for a computer to use this algorithm than to use an algorithm based on the manual method.

For homework due next class, make up a series of legal Boolean expressions for demonstrating the algorithm. They must include the "trivial" atomic expressions "1" and "0", expressions with parentheses like "(1|0)&(1|(~0))", expressions with no parentheses like "0|~0&0+1" Demonstrate the algorithm on each one. Your submission must show for each expression, a series of pictures of the two stacks. When an element is popped, you might show that by crossing out the popped element. However, when an element is pushed on top of a crossed out element, you MUST draw a new picture, to show clearly the position where the new element went.

The algorithm will be demonstrated in the first lecture.

2 The Two-Stacks Algorithm

The algorithm uses the precedence relation that specifies that prefix operator ~ (Boolean NOT) has highest precedence, infix operator & (Boolean AND) has precedence between that of ~ and |, and | (Boolean OR) has lowest precedence.

The algorithm processes the input characters one by one, ignoring any blanks and interpreting the characters ()|~&01 and A—Z as input tokens. If letters appear, the algorithm would have access to their Boolean values (0 or 1 for false or true).

The algorithm uses two stacks named `ValStack` and `OpStack`. Both stacks are initially empty. The algorithm sometimes invokes the `reduce()` operation. This operation is the following steps:

1. Copy and pop the top operator from `OpStack`,
2. Copy and pop the top one or two values from `ValStack`, depending on whether the copied operator applied to one or to two operands.
3. Calculate the result of applying the copied operator to the one or two copied values.
4. Push that result value on `ValStack`

The main idea of the two-stacks algorithm is that when we scan the input from left to right, we have to choose *between* pushing the current input symbol or value into one of the stacks for later use, *or* performing the `reduce()` operation. Here is the algorithm in detail:

```
while (there are more input characters to process) {
  Obtain the next input token or end-of-input character c.
  switch(c) {
    constant or variable: Push the value of c on ValStack;
                        break;
    unary operator ~:
                        push c ( which is '~' ) on the OpStack;
                        break;
    binary operator | or &:
                        while( ( !OpStack.empty() ) &&
                              ( the operator OpStack.top() has equal or higher
                                precedence than c ) )
                          reduce( ); // Remember, reduce( ) removes an operator from OpStack
                        };
                        OpStack.push( c );
                        break; // End of binary, i.e., two-operand operator case
    '(' : OpStack.push( '(' );
          break; // End of '(' case.
    ')' : while( OpStack.top() != '(' ) {
          reduce( );
          }
          pop the '(' from OpStack;
          break; // End of ')' case. We match the current ')' with a previous '('.
  end of input:
          while( ! OpStack.empty() ) {
            reduce( );
          }
  } // End of switch.
} //End of while.
```

The `OpStack` should be empty and `ValStack` should have exactly one value in it. If so, it is the expression's value. If not, the input was syntactically wrong.

(An operator like NOT that applies to one operand is a **unary** operator. An operator like AND, OR and addition or subtraction of integers that applies to two operands is a **binary** operator. Do not confuse the idea of a binary *operator* with the idea of a binary *number representation*!)

3 Project 1

The program must be structured as a class named “BoolExpression”. You may design and use additional classes, or put inner classes in `class BoolExpression`, or just use data members and helper methods.

When the program runs, it should read from the terminal a Boolean formula possibly with variables drawn from A-Z as above. The formula will be on one line and will be 100 or fewer characters long. The program should prompt for input.

If the formula only has constants, the program should print its truth value, “0” or “1” on one line, underneath a copy of the formula. For example, with input “`1&(0|~(1&0))`” the output should be

```
1&(0|~(1&0))
1
```

You must (1) implement the two stacks yourself, either using arrays or linked lists; and (2) use the two-stacks algorithm outlined above.

The functionality of evaluating constant Boolean expressions using your own stacks and the two-stacks algorithm is the minimum for getting a C grade on this project (60%). The full 60% also requires that an error message be printed instead of truth values if the stacks underflow or if, at the end of the input, `OpStack` is not empty or `ValStack` doesn’t have exactly one element.

If the formula has Boolean variables, the program should print the truth table for the Boolean function given by the expression. The variables, which are capital letters, should be listed in alphabetical order (see below.)

The remaining 40% will be awarded for generating the truth table for Boolean expressions with variables. For this, you must create a “symbol table” to hold just the letters that appear plus a current truth value for each. You must code algorithms to insert the letters, generate and record the truth value assignments one by one, and enable the 2-stacks algorithm to get the current Boolean value for a letter whenever it needs that.

Example: The the user could run the program by typing to the shell

```
BoolExpression
```

```
if it is a C++ application, or
```

```
java BoolExpression
```

```
if it is a Java application. The program should type:
```

```
Input a Boolean Expression:
```

```
If we then type after the colon, for example,
```

```
Input a Boolean Expression:(W|~B)&1
```

the program should, for example, print the following and exit:

```
BW (W|~B)&1
00 1
01 1
10 0
11 1
```

A bonus of 10% will be added if your program *always* detects a syntactically incorrect expression and prints an error message instead of a truth table. This is most easily accomplished if you use the concept of state in your algorithm design. It requires somewhat more logic than just testing for stack underflow and checking the proper stack contents at the end.

4 Programming Project 2

You will write a C program that uses the standard C formatted input and output functions `printf` and `scanf`, the unformatted block I/O functions `fread` and `fwrite`, and a byte-printing function from Bryant and O'Hallaron's book.

As needed, read about the basics of C in Patt and Patel's book, the handout from Bryant and O'Hallaron, the optional review book, etc., and send direct, factual questions and mysterious compilation errors to me at `sdc@cs.albany.edu`. For this project, the elements of C you need are the same as in C++, with the following exceptions:

- In C, you must put all declarations at the beginning of the block, not simply before the declared variable is used, as in C++ and Java. Similarly, you cannot code `for(int i = 0;` You must declare `int i;` on top of the block containing that `for(...)` and then code `for(i = 0;` You will get VERY MYSTERIOUS ERROR MESSAGES from `gcc` if you overlook this rule.
- You must use the `gcc` command to compile the program, not `g++`
- You cannot use classes and member functions. You can use `structs`, with data members only. (This project doesn't need `structs`.)
- C does not have reference parameters. (You cannot write `void setInt(int &X) { X = 0;}` so the following code returns 0 through argument II: `int II; setInt(II);`.) Every argument is passed by value. The `scanf()` input function expects a POINTER to the variable to which the input value is delivered. So, the translation of the C++ code "int Var; cin >> Var; cout << Var << endl;" is

```
#include <stdio.h>
...
int Var;
/* Read one integer of data */
scanf("%d", &Var ); /* Pass THE ADDRESS OF Var to scanf */
/* Print the integer that was read in, and follow it by a newline */
printf("%d\n", Var ); /* Pass the value stored in Var to printf */
```

In fact, the above code is expresses the main idea of this project!

- Tip: To read one character of data, such as the characters used for menu choices below, just code

```
char menuChoice;  
...  
scanf("%c", &menuChoice );
```

4.1 Project Goals

You will begin with a very simple application to get used to programming in C in the command line UNIX environment. The program must be named `datatool32`

The overall purpose is to expose some data representation issues that arise at the interface between data types of high level languages and the bitwise data storage features of computer hardware.

Quizzes and examinations will include questions to perform small examples of the operations in the program and to answer theoretical questions about them.

The program will repeatedly:

- Offer the user a choice of data input operation.
- Input data according to the chosen method and in each case store it in a 32 bit C variable.
- Display what's stored in hexadecimal AND binary.
- Offer the user a choice of output operation.
- Display the result of the chosen output operation.

All input will be from standard input and all output will be to standard output. Therefore, the user can observe the display on a text terminal and provide input from the keyboard; but one can also use file input or output by shell redirection when one runs the program.

4.2 Input

(S1) Here is the model input menu, interleaved with remarks which comprise numbered specifications: (Sample input, output, code or ASCII strings will be printed in `typewriter` font. Explanations and specifications will be printed in Roman font.)

Since this is a programming and not a typing course, a text file with all the menu text is available from the course web site. Copy and paste each line into your program. Tip: To print a % from inside a printf format string, escape it with `%%`.

- (0) Quit.
- (1) Previous input.

(S2) Display “(1) Previous input” only if previous input is available.

- (2) Decimal input of signed integers using `stdio %d` format.
 - (3) Input of signed integers with self indicated base, `%i` format.
 - (4) Hexadecimal input of integers, `%x` format.
 - (5) Decimal single precision input, `%f` format.
 - (6) C string up to length 3, `%3s` format.
 - (7) Unformatted block input of 4 bytes with `fread(&dest, 1, 4, STDIN)`
- Enter input choice:

- (S3) Do not display the choices your program doesn't implement.
- (S4) If an invalid choice is entered, display a useful error message, redisplay the entire menu and prompt, and let the user try again. If 0 is entered, exit immediately.
- (S5) The terminal cursor should be after the `:` so the choice number will be echoed on the "Enter.." line when the user types it. All prompted input should follow a similar style.

(S6) Upon a valid choice, display prompt:

Enter input value:

4.3 Universal Output

(S7) Display the 32 bit binary data value in hexadecimal by calling `printf("%#018x\n", value);` (In this format string, `#` signifies use the variation of hexadecimal that begins with `0x`, `0` [zero] signifies print leading zeros, `18` means use 18 characters: 16 hexadecimal digits plus the two in "0x", and `x` signifies use hexadecimal format. The `\n` denotes the newline character. So a newline is output after the number.

(S8, 10 points) On the next line, display the value as a sequence of bits expressed by the characters "0" and "1", with successive groups of 4 bits separated by single spaces. For example, when input choice (2) was made and data 62 is entered, the terminal should show:

```
Enter input value:62
0x0000001E
0000 0000 0000 0000 0000 0000 0000 0001 1110
```

To accomplish this, design and code an algorithm that repeatedly uses C/C++/Java shift operators (`<<` or `>>`) to get the bit to be printed into a particular position, and then use bitwise AND (`&`) with a "mask constant" to obtain a value that is zero if the particular bit is 0 and is non-zero otherwise. This value can be used to select whether to print 0 or 1. (S9, 5 points) The algorithm must also keep track of when to print the nibble separating space.

4.4 Selected Output

(S20) After printing the line of binary, display the output menu below.

- (1) Integer signed decimal, `%d` format.
- (2) Integer unsigned decimal, `%u` format.
- (3) Floating point interpretation, decimal display, `%f` format.

- (4) C string with Max. length 3, %3s format.
 - (5) Unformatted byte output with `fwrite(&variable, 1, 4, STDOUT)`
 - (6) Bryant and O'Hallaron's `show_bytes(&variable, 4)`
 - (7) Unsigned integer in English.
 - (8) Signed integer in English.
 - (9) LC-3 3-register instruction 4-3-3-1-2-3 interpretation of the lower 16 bits, decimal.
 - (A) LC-3 register-signed PC offset 4-3-9 interpretation of the lower 16 bits, decimal.
- Enter output choice:

Warning: A C-string is represented by the address of the first byte of the string. That address is also called a pointer (value) to the first byte. Therefore, the call to `printf(...)` run in case (4) must pass the address of the variable holding the data to be printed. `fwrite` and BOH's `show_bytes()` functions also expect pointers to the data to print, not the data itself.

The 9-bit offset value in case (A) should be sign-extended and displayed as a signed decimal number.

After the output choice is read and validated, the program should run code to **interpret the stored 32-bit data according to the output choice and print the interpretation as indicated.**

Fortunately (and sometimes unfortunately) the standard C library function `printf` does not check or automatically convert the types of the expressions or variables that provide the data to be printed.

The first 4 output cases should be handled with `printf`. The data argument to `printf` is always the value to be printed. Different coding is needed for case 4 since the value that represents a C string is the address of the first character in the string.

Surprise: It is normal and correct that when this program does integer input and floating point output, or vice versa, the input and output numbers will be different!

The C unformatted `fread` and `fwrite` functions both require the pointer (address) of the storage into which the data is read and from which it is copied out respectively.

“English” output is what is often printed on checks: 11998_{ten} is “eleven thousand nine hundred ninety eight”. Negative numbers should be printed beginning with the word “negative”.

The LC-3 computer hardware decomposes the 16 bits of an LC-3 word it uses as instructions in several ways, for further interpretation. Learning to use C/C++ shift and bitwise logic operations to program such interpretations is part of the course¹. Here's example of cases 9 and A for the same input:

Hex (input): 0x04432146

Binary grouped by nibble: 0000 0100 0100 0011 0010 0001 0100 0110

Binary grouped by LC-3 4-3-3-1-2-3: 0010 000 101 0 00 110

Decimal grouped by this format : 2-0-5-0-0-6 (case 9 output)

Binary grouped by LC-3 4-3-9: 0010 000 101000110

Decimal grouped by this format : 2--186 (case A output)

¹It will be used in the some projects.

4.5 English conversion

The result is composed of words separated by spaces. The words and spaces can be generated and outputted in left-to-right order. After the last word, please output a newline with `printf("\n");` to flush the output buffer.

You might code useful arrays of constant C strings. For example,

```
const char *zeroToNineteen[20] =
{ "zero", "one ", "two ", "three ", "four ",
  "five ", "six ", "seven ", "eight ", "nine ",
  "ten ", "eleven ", "twelve ", "thirteen ", "fourteen ",
  "fifteen ", "sixteen ", "seventeen ", "eighteen ", "nineteen " };
```

Another useful array can contain "twenty ", "thirty ", etc.

The codes `printf("%s", zeroToNineteen[1]);` and `printf("%s", "one ");` will both print "one ".

Here are some facts and observations to help you:

- The maximum absolute value of an integer with a 32 bit unsigned binary or 2-s complement binary representation is under five billion.
- English expresses such numbers in terms of how many billions, millions, thousands, and whatever is left over under 1,000 are added up to form the number.
- Only 0 uses the word "zero". (1,020 is NOT "one thousand zero hundred twenty zero").
- The C/C++ operator / on positive integers divides with throwing away the remainder.
- The names of numbers from 1 to 19 are rather arbitrary. The names of numbers from 20 to 99 follow a regular pattern. There is another pattern for 100 to 999.
- DO NOT use more than 9 hundreds. For example, 1200 should be expressed "one thousand two hundred", not "twelve hundred".
- Use the American English billion, which is 1000 millions, NOT the British form where 10,000,000,000 is "ten thousand million" and a (British) billion is 10^{12} . The numbers of billions, millions and thousands are therefore between 0 and 999.

5 Grading, etc.

- My evaluation of your work in Projects 1 and 2 will be used to determine whether I will waive the prerequisite of C or better in a Data Structures course. If you don't have this listed prerequisite AND you do worse than C (60%) either Project, you will be deregistered from the course for not meeting its prerequisites. (Lateness will be ignored for this purpose.)

- Your work for each project must be submitted through WebCT to the appropriate assignment. If, after the due time, you improved your work after a submission, you may submit the late but improved version to the “-extra” version of the project. The additional points of your improved but late version will be penalized by the late penalty rate and then added to your regular, earlier submission’s score.
- Each submission must be comprised of a complete set of Java, C++, C, assembler source, or ascii files, including any C or C++ header files, needed to build the executable program.

After you submit your first file to WebCT, you can continue to add additional files.

Executable UNIX files, Java `.class` or executable `.jar` files, and Unix object (`.o`) files will be deleted before we attempt to build your program from sources. 5 points are automatically deducted for any such files received.

Each submission MUST also contain a file named `build.sh` to build the executable program from the sources. It must contain in a shell script the UNIX commands to build the program when all your submitted files (including `build.sh`) are put in an empty UNIX directory and the script is run. The first line of `build.sh` must be `#!/bin/sh -v` An example of a `build.sh` script for a Java application is

```
#!/bin/sh -v
javac *.java
```

The Project 1 script must build a UNIX application named “BoolExpression” or a class file named “BoolExpression.class”. Project 2 requires a C program; the script must create an UNIX executable named “datatool32”. Later in the course `make` will be taught and “Makefiles” must be submitted to orchestrate the build process. A Makefile can contain a “clean” target to help remove `.o`, executable and `.class` files safely. Such future Makefiles must meet standards particular to this course to help teach the need and operation of “`make`”. For Projects 1 and 2, 10 points will go for whether the build script works.

(Additional requirements will be imposed on future projects.)

- If it doesn’t compile it gets ZERO points! You can and MUST run your code through the compiler to find and then fix ALL syntax and other compilation and linkage errors.
- The due time both Projects 1 and 2 is Monday, September 10, 12:00 noon. Late submissions received until Wednesday, Sept. 12, 12:00 noon (exactly 2 days equals 48 hours later) will be subject to a lateness penalty factor equal to $(4.0 - ND)/4.0$. In this formula, ND is the amount of time late, measured in days. The penalty factor, computed using floating point arithmetic, will be multiplied into your score. This means the amount deducted for lateness will begin at 0% and rise (almost) continuously to 50% during the 48 hour late period. For Project 1 and 2 only, very late submissions received after that but by 12:00 noon, Sept. 21 be penalized by 50%.
- Give your build script and the resulting executable a final test before you submit your work; do NOT open any of the files with an editor AT ALL after that final test. Carefully delete the objects, `.class` files and executables and then submit it. You will be instructed later on how to put a “clean” target in a Makefile to automate this cleaning process.