To print higher-resolution math symbols, click the
Hi-Res Fonts for Printing button on the jsMath control panel.

Lab 4: Number Representation last modified by Aaron Bloomfield on September 23, 2013 3:08:23 PM EDT

Laboratory 4: Number Representation

Week of Monday, September 23, 2013

Objective

To become familiar with the underlying representation of various data types, and to learn how to examine these representations in the debugger.

Background

In class we discussed how various data types – integers, characters, and floating point numbers – were represented in computers. In this lab we will use the debugger to examine some of these representations.

Reading(s)

1. Any of the optional readings on the Readings page, in particular, those on arrays and unions, if you feel you need a bit more background.
2. If you are a bit confused about unions, you might want to read about them prior to the in-lab. Do a Google search for 'C++ unions'.

Procedure

Pre-lab

1. Any of the optional readings on the Readings page, in particular, those on arrays and unions, if you feel you need a bit more background.
2. Complete the Unix tutorial found in the Collab workspace (in the "03-04-more-unix" directory – the "01-unix-tutorial" directory is from lab 1): Tutorial 3: More UNIX, part 1. This tutorial is originally from the department of Electrical Engineering at the University of Surrey at http://www.ee.surrey.ac.uk/Teaching/Unix/. You went through sections 1-4 in the last tutorial; this lab has you completing sections 5-8.
3. Write the sizeOfTest() function (note the capitalization), as described in the pre-lab section.
4. Write the outputBinary() function, as described in the pre-lab section.
5. Write the overflow() function, as described in the pre-lab section.
6. Combine these functions into a prelab4.cpp file, as described in the pre-lab section.
7. Complete your floating point conversion, as described in the pre-lab section, into a Word file called floatingpoint.pdf; you can convert a file into a PDF via the directions on the How to convert a file to PDF page.
8. Files to download: none
9. Files to submit: prelab4.cpp, floatingpoint.pdf

In-lab

1. Work through the steps one a time. Be sure that you understand what is happening at each step.
2. The three parts of the in-lab have you editing inlab.doc or inlab4.cpp or both.
    1. Size of C++ data types
    2. Representations in memory
    3. Primitive arrays in C++
3. Files to download: inlab4.doc
4. Convert inlab4.doc to a PDF via the directions on the How to convert a file to PDF page
5. Files to submit: inlab4.pdf (the PDF version of inlab4.doc), inlab4.cpp

Post-lab

1. Write the recursive bit counter, in bitCounter,cpp, as described in the post-lab section.

2. Complete the radix worksheet – this worksheet is in the radixWorksheet.doc file in Collab. This will be converted to PDF and called radixWorksheet.pdf.
3. Files to download: radixWorksheet.doc
4. Files to submit: bitCounter.cpp, radixWorksheet.pdf

Pre-lab

One of the deliverables for the pre-lab is a PDF document named floatingpoint.pdf. It must be in PDF format! See How To Convert A File To PDF for details.

Reading

You should see the Readings on arrays – that will be needed in the in-lab. Also, the Readings on unions.

sizeOfTest()

The size of C++ data types is dependent on the underlying hardware on which you are running. A programmer may determine the size of various data types by using the sizeof() operator. Although it looks like a function, it's a language construct – like while() or if() – so it's technically an operator. sizeof() returns the size, in bytes, of a given variable or data type. Note that you can use sizeof() with types, variables, pointers, classes, and objects.

Write a small C++ function that demonstrates the use of sizeof() with the following types: int, unsigned int, float, double, char, bool, int*, char*, double*. Your function should print out all the types and their respective sizes. You will use the values outputted by your program to fill in the table in the in-lab section. The function should be called sizeOfTest() (note the capitalization), so as not to confuse C++ with the sizeof() operator. This function should not take in any parameters.

Binary number output

The second coding exercise for the pre-lab is a binary output program. The function to write is called outputBinary(), and it will take in one parameter, an unsigned int. It must be unsigned, or else your code may not work! You should then print out the binary representation of the passed parameters in **big Endian** format.

- If you do use for loops to do this, become familiar with the left shift operator (<<) and what it does to (unsigned) ints, and the binary and/or operators (& and |).
- For examples, see the course notes. You can also use Window's calculator to convert numbers to binary (select View->Scientific).

The Limits of Representation

What do you think will happen when you add 1 to a variable containing the maximum value of a type? Write a function called overflow() to answer the following questions:

- What happens when you add 1 to an unsigned int variable containing the maximum value of an unsigned int?
- Does the program halt?
- What answer do you get?
- Why does this happen?

Your function should create an unsigned int, give it the max value, and add 1 to that. By printing out the result, you will effectively answer the first 3 of the 4 questions. Answer the last question in a cout statement (NOT as a comment!). The function takes in no parameters.

prelab4.cpp

Your three functions, sizeOfTest(), outputBinary(), and overflow() should be combined into a prelab4.cpp file. This is the one C++ source code file that you will submit for the pre-lab. The input requirements for this program are fairly strict, so as to allow automated execution of your programs.

Your program should ask for a single integer value for input, which we will call x. The program will call the three functions in order: sizeOfTest(), outputBinary(), and then overflow(). Note that only outputBinary() takes in x as the parameter. The program should take in no further input.

Floating point conversion

For the last part of the pre-lab, you will need to convert a floating point number to binary representation, and another number from binary representation to a floating point number. You should do this by hand (i.e. not in a computer program), and have the worked-out solution (similar to the lecture notes) be in a floatingpoint.pdf file – you can use any editor you would like to generate the file, as long as what you submit is a PDF file.

First, you will need to determine what your floating point numbers are going to be – these numbers will be different for each student. To do so, visit the URL http://libra.cs.virginia.edu/getfloat and enter your UVa userid. Each floating point number is unique to the userid entered. Note that the hexadecimal number printed is in little Endian format.

The first number must be converted into (little-endian) format – you should leave your answer in hex. It will be easiest to represent this number as a hexadecimal number. The second number (the one in hexadecimal) needs to be converted to IEEE 754 single-precision (i.e. 32-bit) floating point number. To make your life easier, you can assume that all digits of the mantissa after the first 10 digits will be zeros.

Note that a '0x' before a number means that the number is in hexadecimal. Thus, 0x11 has decimal value 17, as that is what 11 is in hexadecimal. The '0x' part is not part of the value.

For example, if you entered 'asb2t', you would get:

```
Your magic (32 bit) floating point number is -35.125
This is the number that needs to be converted to (little endian) binary, and expressed in h

Your other magic floating point number is, in hex, 0x00401f41
This is the number that needs to be converted to a (32 bit) floating point number.
Note that the hexadecimal printed above is in little-endian format!
```

In this example, you would convert -35.125 to 0x800cc2 (or 0x00800cc2 – same thing), and 0x401f41 (or 0x00401f41 – again, the same thing) to 9.95312.

Note: during the conversion, you can assume that any bit in the mantissa after the 10th bit will be a zero. It could be that your floating point number needs the first 10 bits of the mantissa, or it could need less. But all bits after the first 10 should be set to zero.

You will be expected to be able to do this on a test – although in an exam situation, because no calculators are allowed, the math involved with determining the mantissa won't be very hard. Note that with the provided numbers for this pre-lab, any bit in the mantissa beyond the 10th bit will always be zero – so if you are getting 1's in the mantissa after the 10th bit, you are doing something wrong.

Your conversion should be in a PDF file called floatingpoint.pdf, which will be submitted with the pre-lab. The idea is to show the math behind the conversion (similar to how was done in class), not to write a program to do it.

In-Lab

The purpose of this in-lab is to complete the inlab4.doc worksheet, and turn that in (in PDF format - see How To Convert A File To PDF for details). The in-lab description, below, discusses filling in this worksheet. As part of this in-lab, you will have to write a few small programs, which you will combine into an inlab4.cpp file - the output of this program will give you some of the values to fill into the inlab4.doc worksheet. The sections below named 'Representation in memory' and 'Primitive Arrays in C++', describe what should be in this file. It should not take in any input, and should just print out the necessary values.

32-bit versus 64-bit

We realize that most of you will have 64 bit computers. However, the VirtualBox image is a 32-bit image. The information you provide for this in-lab can be either for a 32-bit computer (the VirtualBox image), or a 64-bit computer (what you probably have, as well as what is in the lab), as long as you are consistent - meaning you have to have **all** your answers be one or other; you can't mix-and-match.

If you are on a 64-bit machine, you can also compile your code with the "-m32" option, which will force the resulting program (a.out) to be a 32-bit program, regardless of whether it is running on a 64 bit machine or not.

When using GDB, you can use the 'x' (for 'eXamine') command to print out the pointee of an address. Consider the C++ program that has two variables defined, int i and int *p. To print out the int variable i, you would enter "x &i" (as you have to enter the address of the data). If p is a pointer to a value, you would enter "x p" to print out the pointee. This may print it using many more hexadecimal digits than you wanted, so you can add a parameter to the 'x' command to have it print only a certain amount:

- "x/xb p": this will print the one byte at the address that is pointed to by p
- "x/xh &i": this will print the two bytes (short) of int variable i
- "x/xw p": this will print the four bytes at the address that is pointed to by p
- "x/xg &i": this will print the eight bytes of int variable i

Note that this is only really useful when printing out a **smaller** size than really exists. If your variable is 4 bytes, and you print out 8 bytes, then the other 4 bytes will be whatever arbitrary values are adjacent in memory.

Size of C++ data types

The size of C++ data types is dependent on the underlying hardware on which you are running. A programmer may determine the size of various data types by using the sizeof() operator, discussed in the pre-lab. Note that, unlike a function, you can supply a type to the sizeof() operator (i.e., 'sizeof(int)') – you can't do this with a function.

Download the Word document called inlab4.doc from the Collab site – this contains (among other things) the table below, in which you will enter in your answers. Remember to include the standard identifying header at the top of the file (name, date, etc.). This file will be submitted electronically in PDF format.

Note: char, short, int, and long are all integral (i.e. integer) types. Integral types may be either signed or unsigned. Signed types have a different range of values than their unsigned counterpart. Unless specified as unsigned, integral types in C++ are signed.

To fill in this table, you can use the outputBinary() function that you write for the pre-lab for some of the types (specifically those of size 4). To do this, create a union (as per the lecture notes on number representation) – have it contain an unsigned int and a float (or whatever 4-byte type you are dealing with). Copy in a float, copy out an unsigned int, and print that to the screen using your outputBinary() method.

For the other types, you will need to use the debugger. Write a simple C++ program that creates the variables, and stores the appropriate value (zero or one) into them. Compile (remember the –g flag!), load the debugger, set a breakpoint, and start the program execution. To find out how the value is stored in hex, first find out the address of where the variable is in memory ('print &x', for example). Then, using that address, you can use the examine command: 'x/x 0xbf8cd9ac'. The first 'x' is telling gdb to eXamine a location in memory. The second 'x', after the forward slash, is telling GDB to print out that result in hex, and the address is the output from the previous print command. This will print the 4 bytes (32 bits) of memory at that location, in hex. If you want to print 8 bytes (64 bits), use 'x/xg' (the 'g' tells GDB to print 8 bytes instead of 4). You can also combine these commands by entering 'x/x &x'. The 'x/x' part works as above; the '&x' tells it to print the value in memory at the address ('&') of the 'x' variable.

For various optimization reasons, when you declare a variable in C++, it does not immediately initialize it. In fact, it will initialize it only when it is first used. Thus, if you set a breakpoint after you declare and initialize a variable, but before it is used, the variable will have a random value in it. You can solve this by printing out the variable via cout – this causes C++ to initialize the variable for the output statement. You can then set your breakpoint after this cout statement.

The 'max value' column can be determined by understanding how the type stores the values (see the lecture notes for details). Note that you can assign hex constants directly in your C++ program: rather than saying =17, you can say =0x11. You can also assign them directly in the debugger. In C++, a number beginning with 0 is in octal, e.g. 011 is 9 decimal. For chars, we are looking for the maximum integer value that can be stored therein (a char is just a byte-sized int); bools have only two possible values (true and false), so pick your max and min from those. For pointers, it's the highest memory address that can be addressed.

The following table does not render very will through the wiki, but the table in inlab4.doc renders much better. Note that 'zero' and 'one' should be interpreted appropriately for the given data type. So 'zero' would be 0 for an int, 0.0 for a float, false for a bool, etc. Likewise for 'one'.

As was mentioned in lecture, the #define'd value UNIT_MAX contains the maximum integer value, and you must #include <limits> in order to access it.

C++ Type	Size in bytes?	Max value? (base 10)	Zero is stored as (in hex)?	One is stored as (in hex)?
int				
unsigned int				
float				
double				
char				
bool				

C++ Type	Size in bytes?	Max value? (base 10)	NULL is stored as (in hex)?

int*
char*
double*

The results in the hex columns should be in big endian format, which is the same format that GDB displays, as well as the format that your outputBinary() program displays.

To convert binary into hex, split the bits into groups of 4. Convert each of the group of 4 to a decimal number, which will range from 0-15. To write it in hex, write 10 as a, 11 as b, 12 as c, 13 as d, 14 as e, and 15 as f. Thus, a 32-bit value will be split into 8 groups of 4, and each of those groups of 4 will be a single hexadecimal digit.

Representation in memory

This exercise will show you how to read the contents of a particular memory address. This will be useful for debugging code and for understanding the underlying data representation of abstract data types.

Recall that all Intel 80x86 machines (i.e. all Pentium-class machines) are little-endian. Thus, 0xD97C34A2 is stored as: A2 34 7C D9, with the least significant byte listed first. However, when you examine the value in gdb (using the ‘x/x’ command), it will display it in big-endian format, as that is how humans think of numbers.

Write a C++ program, called inlab4.cpp, where you consecutively declare variables of these types: bool, char, int, double, int*, and assign a value to each of them. The last line(s) of the program should print out the values. Put a breakpoint near the end of the program, but before the last print statement(s). Once the breakpoint is hit, type expressions to examine the addresses of all of these variables (e.g. &i). Then for each of these variables, view the contents of their addresses (via the ‘x/x’ command from above).

Find one of your int variables in memory. Change its value via the ‘set variable <var> = <value>’ command. Examine the new variable’s contents in memory. Is it what you expected? Continue the program execution – did it properly print the changed value?

This program should take in no input.

After completing this section of the lab, you will be expected to understand how to use the debugger to:

- View variables in memory.
- Change an int value from positive to negative.
- Observe interesting/different features in memory (e.g. skipped memory) and be able to explain it.

Primitive Arrays in C++

For the pre-lab, you should read the Readings on arrays, if you feel you need a bit more background. Note how two (or higher) dimensional arrays are stored in row-major order in C++, as opposed to being stored as arrays of arrays in Java.

For this part, you will need to add a bit of code to your inlab4.cpp file. You program should show a clear separator where the previous section’s part of inlab4.cpp ends and where this section’s part of inlab4.cpp begins. The additional code should declare a one dimensional array of chars and a one dimensional array of ints:

```
int IntArray[10];
char CharArray[10];
```

In your program you should also declare a two dimensional array of chars and a two dimensional array of ints:

```
int IntArray2D[6][5];
char CharArray2D[6][5];
```

Assign different values into each element of all four arrays. As above, put a breakpoint in your program after the four arrays have been assigned values. Find the address of the first element of each array, and type that address into gdb (via the 'p' command).

This program should take in no input.

Examine where the elements of the four arrays are in memory. You will be expected to understand and be able to explain this

representation for the exams.

Assuming that memory is byte-addressable, write an expression that will tell you the address of the (i, j)th element of IntArray2D as declared above. The base address of the array is the identifier (IntArray2D), and the expression should be in terms of that, as well as *i*, *j*, and the size of the int type. You can assume that: ($0 \leq i < 6$), and ($0 \leq j < 5$). Note: & here means “the address of”, you may use & in your answer. This answer should also go in your inlab4.doc file.

&(IntArray2D[i][j]) = _____

Lastly, convert your inlab4.doc file to PDF format prior to submission.

Post-Lab

Binary bit counter

Write a recursive function that returns the number of 1’s in the binary representation of N. Use the fact that this is equal to the number of 1’s in the representation of N/2, plus 1, if N is odd. You may assume that N is a non-negative integer stored in two’s complement. However, N will be passed in the standard decimal (i.e. base-10) format. This should be a rather simple function that uses what you’ve learned about integer representation. If you find you need things like global variables or the pow() function to implement this then you are going too far.

This program, called bitCounter.cpp should take in a value as a command-line parameter (no input!). See below for how to handle command-line parameters. Note that if the program is run without any command-line parameters, your program should gracefully exit with an appropriate error message. Your program need not handle an invalid number for the command-line parameter. And any addition command-line parameters beyond the first can be (and should be) ignored.

This program should take in no input, only a command-line parameter.

So far, our main() method has had the following prototype:

```
int main()
```

We will now change that prototype to the following:

```
int main (int argc, char **argv)
```

These two parameters are providing you with the command-line parameters. The first parameter, argc, is the number of parameters plus one. The second parameter, argv, is an array of C-style strings (some people list the type as ‘char *argv[]’ to make this more clear – either way is fine).

Thus, if you supply the program with 3 command-line parameters, then argc would be set to 4, argv[0] would be the C-string that contains the program name (‘a.exe’, for example’), and argv[1], argv[2], and argv[3] are the 3 supplied command line parameters.

Your task is to implement the binary bit counter function that takes in a single command-line value (which is a standard base-10 integer) and prints out the number of bits contained therein.

Converting between number systems

Complete and submit the radixWorksheet.doc file (in Collab) as a PDF file called radixWorksheet.pdf. It must be in PDF format! See How To Convert A File To PDF for details.

Untitled note

Find a notebook

Add tag

Add comments

Version 6.0.6: c128369/1.0.1.250

Web Clipper tutorial

Options

Saving clip...

To:

Share

Save

Selection

PDF