**Instructions:**

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet. No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 8 questions, some with multiple parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- If you brought a fact sheet to the test, write your name on it and turn it in with the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

**Do not start the test until instructed to do so!**

**Name** Solution

printed

**Pledge:** On my honor, I have neither given nor received unauthorized aid on this examination.

signed

**1.** [8 points] Using a k-bit representation, signed integers cover a range from MIN to MAX, where MIN and MAX depend on k, and the exact values of MIN and MAX are irrelevant.

If we double the number of bits, to 2k, what happens to the range of signed integers? Be as precise as you can.

Doubling the number of bits essentially doubles the maximum power of 2 that can appear in the expansion of an integer value.

Doubling the exponent squares the value.

So, the new range would be (more or less) from $-MIN^2$ to $MAX^2$.

More precisely, the range for signed ints (2's complement representation is universal) using k bits would be from $-2^{k-1}$ to $2^{k-1} - 1$.

Using 2k bits, the range would be from $-2^{2k-1}$ to $2^{2k-1} - 1$, which agrees with the rough estimate above.

---

**2.** [12 points] Consider the following x86 assembly code fragment:

```
   . . .
   movl   %eax, -8(%ebp)
   addl   12(%ebp), %eax
   movl   %eax, 4(%~~$~~esp)     // typo!
   . . .
```

List all (and only those) parameters to x86 instructions that refer to values that are stored on the stack.

In order to refer to values stored on the stack, the parameter must indicate it is specifying an address, so that would be the following parameters:

-8(%ebp)      12(%ebp)      4(%esp)

The only other parameter, %eax, refers to a value stored in a register, hence in the processor.

Note: "stack" is not synonymous with "stack frame". The stack is a collection of stack frames.

**3.** [12 points] Circle the strings below that would match the following regular expression: `0?[1-9]+[abc]*`

[✓ `0712b`] `0b` [✓ `9ab`] `0420c` [✓ `712`] [✓ `011aa`]

The regular expression implies:

- The string may, optionally, start with a single 0.
- The next (or first) part of the string must be a sequence of one or more of the digits 1-9.
- The string may, optionally, continue with a sequence of an arbitrary number of occurrences of the characters a, b and/or c.

---

**4.** [12 points] Formulate a regular expression that would match all, and only, those strings that satisfy the following conditions:

- The string is at the beginning of a line.
- The string starts with two or more occurrences of lower-case letters.
- Those are optionally followed by `ROFL`.
- The next part of the string consists of zero or more occurrences of `B`, or zero or more occurrences of `M`, or zero or more occurrences of `Z`.
- The string is followed by a space character. (What follows that is irrelevant.)

`^[a-z]{2,}(ROFL)?(B*|M*|Z*)[ ]`

| | |
|---|---|
| ^ | specifies match starts at beginning of a line |
| [a-z]{2,} | specifies two or more occurrences of characters from the range a-z |
| (ROLF)? | specifies zero or one occurrence of the group "ROFL"; the parentheses are necessary |
| B*\|M*\|Z* | specifies zero or more B's OR zero or more M's OR zero or more Z's This is NOT the same as [BMZ]*, which would include BMMZMB. |
| [ ] | specifies a space character; there are several alternatives |

**5.** [20 points] Write the implementation for a small C function that will determine whether or not there are two values in an array of `int`s whose sum equals a given `int` target value.

For example, given the arrays and targets below, the function should respond as indicated:

```
{1, 3, 5, 4, 2} and 7 → yes
{2, 1, 3, 5, 4} and 2 → no
{1, 2, 3, 4, 5} and 10 → no
```

The function can have any interface you think is appropriate, so long as its return type and parameters are appropriate and calling it yields the desired effect. You should, of course, give careful thought to the interface.

Write your implementation of the function, including the function interface, here:

```
bool sum2target(int A[], int A_SZ, int Target) {

   for (int left = 0; left < A_SZ - 1; left++) {

      for (int right = left + 1; right < A_SZ; right++) {

         if ( Target == right + left ) {
            return true;
         }
      }
   }

   return false;
}
```

If you don't write the inner loop carefully, you'll look at the same pair of values twice. That's inefficient, and should be avoided.

But… you may also manage to consider using the same value from the array twice (A[i] + A[i]), which isn't allowed (see the second example), so you'd have to ensure the indices are different.

The best approach is to simply avoid the issue as shown.

**6.** Consider the following short C function:

```
// FindFirstZero() returns the lowest index at which the value zero
// occurs in the array A[].
//
// Pre:
//    A[] has been initialized with Size int values
// Post:
//    A[] is unmodified
// Returns:
//    Smallest value of k for which A[k] == 0.
//
int FindFirstZero(int A[], int Size) {

   int idx = 0;

   while ( A[idx] != 0 ) {

      ++idx;
   }

   return idx;
}
```

a) [6 points] The implementer of the C function shown above has assumed something that is not guaranteed by the preconditions. What is that?

The implementer has assumed the loop will be terminated by finding an occurrence of zero within the array; i.e, he's assumed there's a zero in the array. In addition, you could argue he's assumed that Size > 0.

b) [6 points] And, assuming all preconditions are satisfied, when will this error affect the results, and how?

The preconditions do not specify that the array must contain a zero, so it is consistent with them to pass an array that does not.

In that event, the while loop condition will never be true for any valid index, from 0 up to Size - 1.

Therefore, the function will eventually attempt to access a location beyond the end of the array.

What actually happens at runtime will depend on whether the function stumbles across a value zero in memory, past the end of the array, in which case it will return an index that's outside the valid range (0 to Size-1), or reaches an address the program is not allowed to access, in which case there will be segmentation fault error.

**Note:**

**There is no such thing as an exception in C; nor is there any automatic checking of array bounds like you have in Java.**

**The two possibilities mentioned above are the ONLY possible outcomes if the array has size 0, or does not contain a 0.**

**7.** The C code shown below compiles to the x86 assembly code shown below. (Some irrelevant assembly code has been omitted.)

```
int main() {         // 1

   int a = 10,       // 2
       b = 20;       // 3
   int c;            // 4

   c = 2 * a + b;    // 5
   b = c - a;        // 6

   return 0;         // 7
}
```

```
      .file  "q7.c"            #  1
      .text                    #  2
.globl  main                   #  3
      .type  main, @function   #  4
main:                          #  5
      . . .
      movl   $10, -8(%ebp)     #  9
      movl   $20, -12(%ebp)    # 10
      movl   -8(%ebp), %eax    # 11
      addl   %eax, %eax        # 12
      addl   -12(%ebp), %eax   # 13
      movl   %eax, -4(%ebp)    # 14
      movl   -8(%ebp), %eax    # 15
      movl   -4(%ebp), %edx    # 16
      movl   %edx, %ecx        # 17
      subl   %eax, %ecx        # 18
      movl   %ecx, %eax        # 19
      movl   %eax, -12(%ebp)   # 20
      movl   $0, %eax          # 21
      . . .
```

a) [6 points] At what addresses (if anywhere) are the C variables a, b, and c stored on the stack? (You cannot give exact, numeric addresses but you can give exact symbolic ones.)

Compare the initializations of a and b in lines 2 and 3 of the C code to lines 9 and 10 of the assembly code, and it's obvious that:

a is at ebp - 8
b is at ebp - 12

As for c, it's assigned a value computed from the values of a and b, and we see that happens in line 14 of the assembly code, from which we see that:

c is at ebp - 4

b) [10 points] Which line(s) of assembly code correspond to statement 5 in the C code? Explain your logic.

```
movl   -8(%ebp), %eax     # 11:  eax = a
addl   %eax, %eax         # 12:  eax = 2 * a      (by addition)
addl   -12(%ebp), %eax    # 13:  eax = 2 * a + b
movl   %eax, -4(%ebp)     # 14:  c = eax
```

**8.** Consider the following C function (irrelevant parts are not shown):

```
void G(int A) {

   static int X = 10;
   int Y = 20;
   . . .
}
```

a) [4 points] Is the value of the variable X stored in the stack frame for G()? Briefly justify your answer.

No. Values stored in the stack frame logically disappear whenever the function returns. But, a static variable must retain its value from one call to the next; i.e., X has static storage duration. Hence, the values of static variables must be stored somewhere else than the stack.

Notes:

- "static" in C does not mean that a constant is being declared; X is a variable and it's value can be changed at will.
- What "static" does mean depends on where it's used.
- X is declared within a function body, so it has local (block) scope, and no linkage. X cannot be referred to from outside G(). Neither of those facts is relevant.

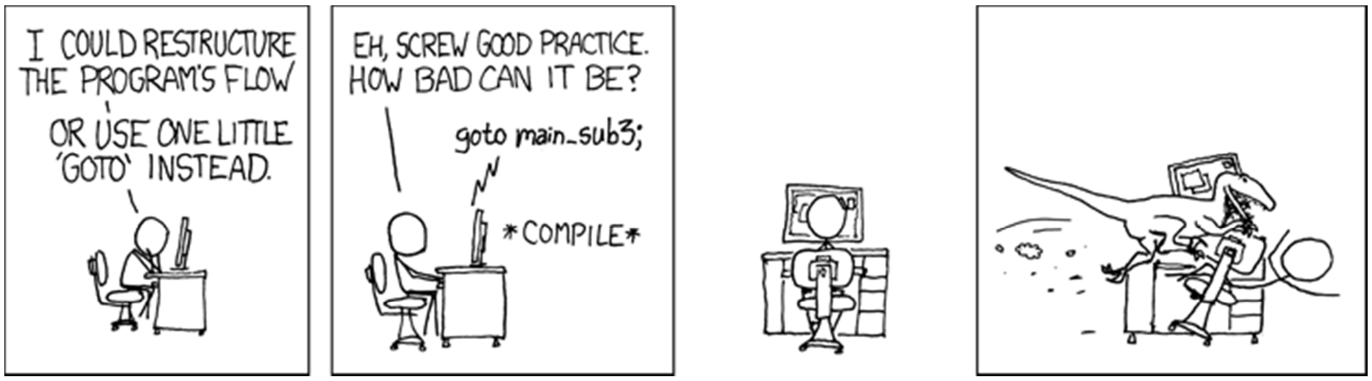
b) [4 points] Is the value of the variable Y stored in the stack frame for G()? Briefly justify your answer.

Yes. Y is an automatic local variable, so it is created (i.e., space to hold its value is allocated) when the enclosing block is entered, and it is destroyed (i.e., that space is deallocated) whenever the enclosing block is left.

This is one of the important advantages of using the stack model for function execution.

Notes:

- Like X, Y has local scope and no linkage; again, neither of those facts is relevant.


I COULD RESTRUCTURE THE PROGRAM'S FLOW OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
*COMPILE*