



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. This examination is closed book and closed notes, aside from the permitted one-page fact sheet. Your fact sheet may contain definitions and examples, but it may not contain questions and/or answers taken from old tests or homework. You may include examples from the course notes.
- No calculators or other electronic 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 7 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*



xkcd.com

1. These questions relate to pointers, data representation, and data types in C.

- a) [4 points] The following C code fragment is intended to sum the values in an array, from high to low. There are no syntax errors. The array `A[]` has dimension `Sz`, and `Sz` actually equals 100. However, the loop does not perform as the programmer intended. Explain exactly what happens at runtime that prevents the loop from executing as intended.

```
int32_t Sum = 0;
int32_t* elem = &A[Sz-1];
while ( elem > 0 ) {
    Sum += *elem;
    elem--;
}
```

The pointer `elem` is initialized to the address of the last data element in the array, which is fine. But the loop test is checking whether `elem` is `NULL`, which makes no particular sense.

- b) [4 points] Complete the following C code, so that the condition in the comment is satisfied, no matter what value `A` has, and using only the following operations: `&` `|` `^` `~` `+` `!` `*`

```
uint32_t A = ???;
uint32_t Y = A + ( _____ ); // Y is set to 0
```

Since `A` is an unsigned integer, it's stored in pure base-2 form. Consider an example; suppose `A` is 10:

```
0000 0000 0000 0000 0000 0000 0000 1010
```

To get a result of 0, we would need to add:

```
1111 1111 1111 1111 1111 1111 1111 0110
```

which happens to be, precisely, the complement of `A` plus 1...

This was tricky. A common answer was `~B`; but `x + ~x` (`x` being a bit) always yields 1. However, if you noticed that, you might have realized that adding 1 to that will yield 0.

- c) [4 points] Complete the following C code, so that the condition in the comment is satisfied, no matter what value `B` has, and using only the following operations: `&` `|` `^` `~` `+` `!` `*`

```
int32_t B = ???;
int32_t B + ( _____ ); // Z is set to 0
```

Since `B` is a signed integer, we need to add the negation of `B`; since signed values are stored using 2's complement form, `-B` can be computed by flipping all the bits of `B` and then adding 1.

2. [15 points] Consider the following three C data types:

<pre> struct _RGB { uint8_t red; uint8_t green; uint8_t blue; }; typedef struct _RGB RGB; </pre>	<pre> struct _Pixel { int32_t x; int32_t y; RGB color; }; typedef struct _Pixel Pixel; </pre>	<pre> struct _Matches { uint32_t* indices; uint32_t nMatches; }; typedef struct _Matches Matches; </pre>
--	---	--

Implement the following C function, conforming to the header comment. **Be careful of syntax in your answer!**

```

/** Builds a list of indices of Pixels that match a given attribute.
 *
 * Pre:   Pixels is an array of nPixels Pixel objects
 * Returns: a pointer to a new Matches object, pointing to an array
 *          of nMatches indices corresponding to Pixel objects
 *          in Pixels[] that matches the given value of Red.
 */
Matches* findRed(uint8_t Red, const Pixel[] Pixels, uint32_t nPixels) {

    Matches* pM = malloc(sizeof(Matches));           // create Matches object

    pM->indices = malloc(nPixels * sizeof(uint32_t)); // create its array

    pM->nMatches = 0;                                // set counter for matches

    for (uint32_t pos = 0; pos < nPixels; pos++) {   // iterate through the Pixel objects

        if ( Pixels[pos].color.red == Red ) {       // see if current element is a match

            pM->indices[nMatches] = pos;             // if so, store its index, and
            nMatches++;                               // increment the counter

        }

    }

    return pM;                                       // return pointer to Matches object
}

```

The Matches object must be allocated dynamically so that it will continue to exist after the function returns. And the array indices the Matches object will point to must also be allocated dynamically, since there's no other valid way to create it.

Some observations:

- Simply making a declaration (`Matches* M`) does not create an object.
- Making a local automatic object (`Matches M`) does not allow you to return a pointer to a Matches object, unless you return `&M`, which is a pointer to a local automatic variable, which will cease to exist when the function returns.
- Incrementing the indices member of a Matches object (e.g. `M->indices++`) will remove your access to the beginning of the array.
- Repeatedly reallocating the array to get the exact size is wildly inefficient; the approach I used above is the correct way. However, this was not a requirement for this question.

3. Suppose we have a memory region containing the following bytes:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	A4	2D	6A	C3	60	A3	12	81	C2	BD	10	94	5D	E5	04	08
00000010	88	F1	1E	3D	AE	3B	8E	19	28	84	12	07	0A	D1	04	53
00000020	A4	54	EE	AB	B5	14	7B	C6	E5	87	8F	C0	5B	DB	CA	03
00000030	10	6A	A0	D7	A7	8E	63	12	39	00	85	B8	F8	06	BD	EC

a) [6 points] Assume you have a pointer `p`, of type `uint8_t*`, and that `p` has been initialized to the first byte in the region shown above. Complete the following C statement so that it will copy 4 bytes of data, starting with the byte pointed to by `p`, into the given variable:

```
int32_t n = *(int32_t*) p;
```

We must cast `p` to create a pointer whose target type contains 4 bytes, then dereference. One error was to cast to `uint64_t*`; this is unwise since the values are specified as signed integers.

b) [5 points] What value (expressed in hex) will be assigned to the given variable by the following C statement?

```
uint8_t a = *(p + 14);
```

04

`p + 12` points to the byte 14 spots after the target of `p`; just count it out.

c) [4 points] Write the values of the bytes (in the memory region shown above) that would be replaced if the following C statement was executed:

```
*(uint16_t*)(p + 16) = 0x34;
```

88 F1

Due to the typecast, we will overwrite 2 bytes of memory, starting at the address `p + 16`. The constant shown will be "widened" with 0's for the assignment.

4. Suppose we have the following C code:

```
int32_t A[8] = {0, 1, 2, 3, 4, 5, 6, 7};
int32_t* p = &A[0];
```

Which element of the array `A` will be the target of each of the following pointer expressions? E.g., `A[3]`. If the pointer expression does not correspond to the address of an array element, say that instead.

a) [3 points] `p + 8` This is one element past the end of the array.

Pointer arithmetic; this advances by `8 * sizeof(int32_t)`.

b) [3 points] `p + 2` A[2]

Pointer arithmetic; this advances by `2 * sizeof(int32_t)`.

c) [3 points] `(int8_t*)p + 16` A[4]

Pointer arithmetic; this advances by `16 * sizeof(int8_t)`, or 16 bytes, which is the size of exactly 4 4-byte integers.

5. The short C program shown below compiles to the assembly code shown below:

<pre> int main() { int32_t N = 0xFFFF0000; // 1 N = N >> 4; // 2 uint32_t M = 0xFAFAFAFA; // 3 M = M >> 4; // 4 return 0; // 5 } </pre>	<pre> main: pushq %rbp movq %rsp, %rbp subq \$16, %rsp movl \$-65536, -4(%rbp) sarl \$4, -4(%rbp) movl \$-84215046, -8(%rbp) shrl \$4, -8(%rbp) movl \$0, %eax leave ret </pre>
---	--

a) [6 points] The shift instruction in line 2 of the C code is translated using a `sarl` assembly instruction, but the shift instruction in line 4 of the C code is translated using a `shrl` assembly instruction. Why does it make sense that the shift instruction in line 2 would be translated this way? Be specific.

Statement 2 is right-shifting a negative signed integer, so we need to shift in 1's for the 4 high bits; an arithmetic right shift will do that (a logical right shift would shift in 0's).

The biggest issue with this question was the lack of thorough explanations. It is not enough to say "We just use `shrl` for unsigned and `sarl` for signed values." You needed to explain why `sarl` is the appropriate instruction for a signed value.

b) [10 points] Complete the diagram of the stack frame for the function given above. Write the address for each stack element (relative to `%rbp`), and the value stored at each location (when the function completes line 4), if that value is known. Show values in hex; shade any "extra" rows in the table that are not needed for the stack frame.

The `0xFFFF0000` value is signed, which means that the sign bit (1) is preserved during the shift. Shifting right 4 positions adds 4 new 1s to the most significant bits, or `0xF`. The result of the shift is `0xFFFFF000`.

The `0xFAFAFAFA` value is unsigned, which means that the sign bit is not preserved; shifting right 4 positions adds 4 new 0s to the most significant bits. The result of the shift is `0x0FAFAFAF`.

The stack frame is allocated 16 bytes by the `subq` instruction in the x86 code. That means that only 4 blocks of the table to the right are needed. The bottom 2 are unnecessary and should be shaded.

- The biggest issues with this question were
- Shifting by 4 nybbles instead of by 4 bits
 - Issues with the `sarl/shrl` shift behavior
 - Not shifting at all

<code>rbp</code>	old value of <code>rbp</code>
<code>rbp - 4</code>	<code>0FFFF000</code>
<code>rbp - 8</code>	<code>FFAFAFAF</code>
<code>rbp - 12</code>	
<code>rbp - 16</code>	

6. [14 points] Implement the following C function, obeying all restrictions. Violations of the restrictions will earn a score of zero.

```

/** Returns 1 if every odd-indexed bit is 0, and 0 otherwise.
 *
 * Allowed operations: ! ~ & | ^ << >>
 * No more than 20 uses of operators.
 * No use of loops, if, if-else, switches.
 * No declarations of constants wider than one byte.
 *
 * Examples:
 * 0000 0000 0000 0000 0000 0000 0000 0000 : 1
 * 0000 0000 0000 0000 0000 0000 1000 0010 : 0
 * 0000 0000 0000 0000 0000 0000 0000 0100 : 1
 * 0101 0100 1000 0000 0000 0000 0000 0000 : 0
 */
int evensAreZero(int N) {

    int mask = 0xAA; // 1010 1010

    return ( !( (N & mask) |
                (N & (mask << 8)) |
                (N & (mask << 16)) |
                (N & (mask << 24)) ) );
}

```

The biggest issues with this question were

- Violating the restrictions, particularly with the use of disallowed operations (+, ==) and with the use of 32-bit constants (0xAAAAAAAA).
- Some students provided solutions that were effectively correct, but returned values such as 2 or 10 instead of 1. This issue was resolvable with the use of the opposite mask (using 0xAA instead of 0x55) and applying the ! operation at the very last step.
- If you shift a 32-bit value by 32 bits, you have removed all data from that memory segment.

7. For this question, you are going to complete a short C program:

```

struct _Segment {
    int length;
    char token;
};
typedef struct _Segment Segment;

char** builder(const Segment* const segments, int nSegments);
void writestuff(char** stuff);

int main() {

    Segment segs[5] = {{3, 'a'}, {8, 'x'}, {2, 'c'}, {12, 'd'}, {6, 'm'}};

    char** result = builder( segs, 5 );

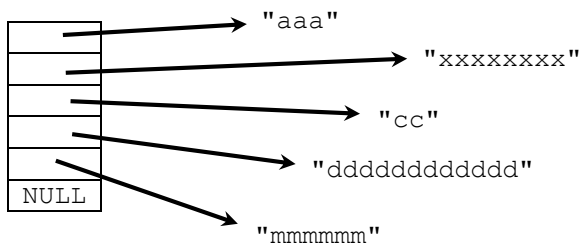
    writestuff( result );

    delete( result );

    return 0;
}

```

The function `builder()` is intended to create an array of C-strings from an array of `Segment` objects. In the code shown above, `builder()` should return a pointer to an array-based structure like this:



Each array cell holds a pointer to a C-string, as described by one of the `Segment` objects; the last array cell stores `NULL`. The C-strings are stored in arrays that are exactly the right size for the specified contents, and the array of pointers is also exactly the right size.

a) [7 points] Implement the following function:

```

/** Deallocates an array of strings, as described in the problem statement.
 * Pre: p points to a dynamically-allocated array of char*, each
 *       of which points to a C-string as described
 * Post: all dynamically-allocated memory has been deallocated
 */
void delete(char** p) {

    char** current = p;           // get handle on 0th array cell
    while ( *current != NULL ) { // walk until find NULL at end of array
        free( *current );       // deallocate the current string
        current++;               // step to the next array cell
    }
    free( p );                   // free the array of char*
}

```

Two major issues with this question. (1) Must deallocate ALL memory - the individual strings and the `char*` array. (2) Pointers - difference between `p`, `*p`, and `**p`.

- b) [12 points] Implement the following function. You must use pointer arithmetic, not array bracket notation when making any accesses to array elements. Violating that restriction will result in a deduction of 6 points.

```

/** Creates an array of C-strings, as described in the problem.
 * Pre: segments points to an array of nSegments Segment objects
 * Post: an array structure, as described, has been created
 * Returns: a pointer to that array structure
 */
char** builder(const Segment* const segments, int nSegments) {

    char** strings = calloc(nSegments + 1, sizeof(char*));

    for (int pos = 0; pos < nSegments; pos++) {

        char* current = calloc(segments[pos].length + 1, sizeof(char));

        for (int idx = 0; idx < segments[pos].length; idx++) {

            current[idx] = segments[pos].token;
        }

        strings[pos] = current;
    }

    return strings;
}

```

The biggest issues with this question were

- Just like you had to deallocate memory for both the individual strings and the char* pointer array in Q7a, you also have to allocate memory for both: space for the char* pointer array outside the loop, and space for each individual string inside the loop.
- sizeof() exists to allow you to allocate precisely the amount of memory you need for a certain datatype. If you're creating an array of char*s, you should be using sizeof(char*). If you're creating an array of chars, you should be using sizeof(char).