C Programming Quiz

(2022-09-02)

Problem 1)

Assume that during runtime the variable `p` points to the memory address `0x7fcb6ca200`. Determine the output of each `printf()` statement in `main()`, when the C code is compiled with GCC (c99) and executed on a 64-bit machine.

```
// Problem 1)

#include <stdio.h>
#include <string.h>

int main(void) {
	char data[] = {'A', 'B', 'c', '\0'};
	// p is a pointer variable, which points to the same address as data.
	char *p = data;
	printf( "p        = %p\n", p );
	printf( "data     = %p\n", data );
	printf( "&data[0] = %p\n", &data[0] );
	printf( "&data[1] = %p\n", &data[1] );
	printf( "&data[2] = %p\n", &data[2] );
	printf( "data[0]  = %c\n", *&data[0] );
	printf( "data[1]  = %c\n", *&data[1] );
	printf( "data[2]  = %c\n", *(data+2) );
	printf( "data[2]  = %c\n", (&p[1]+1)[0] );

	printf( "data     = %s\n", data );
	printf( "sizeof(data) = %lu bytes\n", sizeof(data) );
	printf( "strlen(data) = %lu bytes\n", strlen(data) );
	return 0;
}
```

Problem 2)

Implement the function `reverse_array()` that reverses the order of the elements in the array passed as the single argument to the function.

```
#include <stdio.h>

//------------------------------------------------------------
// Function arguments:
//  array        - an array of integers
//  num_elements - the number of elements (integers) in the array
//------------------------------------------------------------
void reverse_array( int *array, int num_elements ) {
      // Write your code below
      //------------------------------------------------

      //------------------------------------------------
}

void print_array( int *array, int num_elements ) {
      for ( int i=0; i < num_elements; i++ ) {
            printf( "%d%s", array[i],
                  (i==num_elements-1) ? "\n" : "," );
      }
}

int main(void) {
      int a[] = { 1,2,3,4,5 };
      int n = sizeof(a)/sizeof(int);
      reverse_array( a, n );
      print_array( a, n );
      reverse_array( &a[2], n-2 );
      print_array( a, n );
      return 0;
}
```

Problem 3)

Implement the function `count_zeros()` that counts the number of all '0's in a string. The string should contain only '0' or '1' (binary string). If it has any character other than '0' and '1', the function returns -1.

```
#include <stdio.h>
#include <string.h>

//-------------------------------------------------------------
// str is an array of chars passed as the argument to the function.
//-------------------------------------------------------------

int count_zeros( const char *str ) {
	// Write your code below.
	//-------------------------------------------------




	//-------------------------------------------------
}

int main(void) {
	char *arr[] = {
		"", "0","1","00","0110","01001110111011"
	};
	// compute the number of strings in the array 'arr'
	int n = sizeof( arr ) / sizeof( char * );
	printf( "n = %d\n", n );
	for ( int i=0; i < n; i++ ) {
		char *str = arr[i];
		int result = count_zeros( str );
		if ( result == -1 ) {
			printf( "Invalid binary string: %s\n", str);
		} else {
			int len = strlen( str  );
			printf( "#0 = %d, #1 = %d, s='%s'\n",
				result, len - result, str );
		}
	}
	return 0;
}
```

Sample output messages

```
n = 6
#0 = 0, #1 = 0, s=''
#0 = 1, #1 = 0, s='0'
#0 = 0, #1 = 1, s='1'
#0 = 2, #1 = 0, s='00'
#0 = 2, #1 = 2, s='0110'
#0 = 5, #1 = 9, s='01001110111011'
```

Problem 4)

Implement the function `detect_pattern()` that checks whether the (binary) string contains a substring that matches the regular pattern "10+1". If matched, return 1, otherwise return 0.

```
#include <stdio.h>

// str is an array of chars passed as the argument to the function.

int detect_pattern( const char *str ) {
	// Write your code below.
	//-------------------------------------------------

















	//-------------------------------------------------
}

int main(void) {
	char *test[]={ "","10","0100","011000","1010","0111001" };
	int n = sizeof(test)/sizeof(char *);
	for ( int i=0; i <  n; i++ ) {
		char *s = test[i];
		printf( "'%s': %d\n", s, detect_pattern(s) );
	}
	return 0;
}
```

Problem 5*)

Implement the function `create_random_hexstring()` that creates a text file that contains a hexstring with *m* hex digits per line and *n* lines.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// fout – the file descriptor for writing output text lines.
// note: open and read data from /dev/urandom to get pseudo-random numbers

void create_random_hexstring( FILE *fout, int n, int m ) {
	// Write your code below.
	//-------------------------------------------------









	//-------------------------------------------------
}

int main( int argc, char *argv[]) {
	char out_file_name[128];
	FILE* f;
	if (argc!=2) {
		printf( "Usage: %s <output filename>\n", argv[0] );
		exit(-1);
	}
	strncpy( out_file_name, argv[1], 128 );
	f = fopen( out_file_name, "w" );
	if ( f == NULL ) {
		printf( "Cannot open file for write!\n" );
		exit(-1);
	}
	create_random_hexstring( f, 10, 25 );
	fclose( f );
	return 0;
}
```

Problem 6*)

Implement the function `union_sets()` that creates a union of two sets of type `array_set_t`.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// array-based set
typedef struct {
	int size;   // represents the number of members in the set
	int *array; // stores the members (integer numbers)
} array_set_t;


// The function is used to create an array set from a given array of size n.
// It returns a pointer to the resulting array set.
array_set_t* create_array_set( const int *arr, int n ) {
	array_set_t *set = malloc( sizeof(array_set_t) );
	if ( set != NULL ) {
		if ( n > 0 ) {
			set->size = n;
			set->array = malloc( sizeof(int)*n );
			for ( int i=0; i < n; i++ ) {
				set->array[i] = arr[i];
			}
		}
		else {
			set->size  = 0;
			set->array = NULL;
		}
		return set;
	}
	return NULL;
}

array_set_t *union_sets( array_set_t *set_a, array_set_t *set_b ) {
	// Write your code below.
	//-------------------------------------------------









	//-------------------------------------------------
}
```

```
// This function prints the members of the array set.
void print_array_set( array_set_t *set ) {
	printf( "{ " );
	if ( set && set->size > 0 ) {
		int n = set->size;
		for ( int i=0; i < n; i++ ) {
			printf( "%d", set->array[i] );
			if ( i != (n-1) ) {
				printf(",");
			}
		}
	}
	printf( " }\n" );
}


int main(void) {
	int data[] = {1,2,3,4,5};
	// assume that the members of a set are sorted in ascending order.
	array_set_t *set_a = create_array_set( data, 0 );
	array_set_t *set_b = create_array_set( &data[0], 3 );
	array_set_t *set_c = create_array_set( &data[2], 3 );
	printf( "set A = " );
	print_array_set( set_a );
	printf( "set B = " );
	print_array_set( set_b );
	printf( "set C = " );
	print_array_set( set_c );

	array_set_t *set_d;
	set_d = union_sets( set_a, set_b );
	print_array_set( set_d );
	free( set_d );
	set_d = union_sets( set_c, set_b );
	print_array_set( set_d );
	free(set_d);

	// free allocated memory
	free(set_a);
	free(set_b);
	free(set_c);
	return 0;
}
```

Problem 7*)

Implement the function `append_node()` that adds a new node to the end of the linked list, and the function `append_node()` that deletes the tail node (last node) in the linked list. Both function return the pointer that points to the head node of the modified list.

```
#include <stdio.h>
#include <stdlib.h> // for malloc()

typedef struct node node_t;

// The data struture 'node_t' is used to implement a linked list.
struct node {
	int value;
	node_t *next;
};

// This function shows the value of each node in the linked list.
void print_nodes_in_sequence( node_t *head ) {
	node_t *p = head;
	while( p ) {
		printf( "node (value=%d)\n", p->value );
		p = p->next;
	}
}
// This function counts the number of nodes in the linked list
int count_nodes( node_t *head ) {
	node_t *p = head;
	int count = 0;
	while ( p ) {
		count++;
		p = p->next;
	}
	return count;
}

// head - points to the head node (before removal) in the linked list.
node_t *delete_head( node_t *head ) {
	if ( head == NULL ) return NULL;
	node_t *p = head;
	p = p->next;
	printf( "-node (value=%d)\n", head->value );
	free( head ); // free the allocated memory for this node
	return p;
}

// head - points to the head node (before removal) in the linked list.
node_t *delete_tail( node_t *head ) {
	// Write your code below.
	//-------------------------------------------------









	//-------------------------------------------------
}
```

```
// head  - points to the head node in the linked list.
// value - specifies the value used to create in a new node.
node_t *append_node( node_t *head, int value ) {
	// Write your code below.
	//-----------------------------------------------









	//-----------------------------------------------
}

int main(void) {
	int data[] = {3,1,4,2,0,5};
	node_t *head = NULL;
	// add nodes by using the numbers in the data array as values.
	for (int i=0; i < sizeof(data)/sizeof(int); i++ ) {
		head = append_node( head, data[i] );
	}
	printf( "Number of nodes: %d\n", count_nodes(head) );
	print_nodes_in_sequence( head );
#if 1
	// delete nodes (from tail to head)
	while (head) { head = delete_tail( head ); }
#else
	// delete nodes (from head to tail)
	while (head) { head = delete_head( head ); }
#endif
	printf( "Number of nodes: %d\n", count_nodes(head) );
	print_nodes_in_sequence( head );
	return 0;
}
```

****