csc/cpe 357 C Quiz

Archive

Name: _____

Section: _____

Rules:

- Do all your own work. Nothing says your neighbor has any better idea what the answer is.
- Do not discuss this exam outside of class until after 3:00pm.

Suggestions(mostly the obvious):

- When in doubt, state any assumptions you make in solving a problem. **If you think there is a misprint, ask me.**
- **Read the questions carefully.** Be sure to answer all parts.
- Identify your answers *clearly.*
- Watch the time/point tradeoff: 60pts/50min works out to 50s/pt.
- Problems are not necessarily in order of difficulty.
- Be sure you have all pages. Pages other than this one are numbered "*n* of 46".

Encouragement:

- Good Luck!

Problem	Possible	Score
1	•	
2	•	
3	•	
4	•	
5	•	
6	•	
7	•	
8	•	
9	•	

Problem	Possible	Score
10	•	
11	•	
12	•	
13	•	
14	•	
15	•	
16	•	
17	•	
18	•	

Problem	Possible	Score
19	•	
20	•	
21	•	
22	•	
23	•	
24	•	
25	•	
26	•	
27	•	

Problem	Possible	Score
28	•	
29	•	
30	•	
31	•	
32	•	
33	•	
34	•	
35	•	
36	•	

Problem	Possible	Score
37	•	
38	•	
39	•	
40	•	
41	•	
42	•	
43	•	
44	•	
45	•	
Total:	**0**	

1. () (Warm up) Given the C variables defined below, for each of the following expressions, give its value.

```
int   A = 2;      char   *cp = "abcdefg";
int   B = -1;     int    grade[5] = { 4, 3, 2, 1, 0 };
```

If it is relevant, assume that `sizeof(int)` is 4, `sizeof(char)` is 1, and `sizeof()` any pointer type is 4.

Expression	Value
`3 / 6`	
`A + B`	
`+A++ + ++B`	
`grade[2]`	
`*(cp+4)`	
`&grade[4] - &grade[2]`	
`(int)&grade[4] - (int)&grade[2]`	
`(A!=A)?A:B`	
`strlen(cp)`	
`sizeof(grade)`	

Bonus: (requires explanation)

`strlen((char*)(&grade[0]))`	

2. () (Warm up) Given the C variables defined below, for each of the following expressions, give its value.

```
int  A = 2;    char  *cp = "abcdef";
int  B = 3;    int   grade[6] = { 4, 3, 2, 1, 0, -1 };
```

If it is relevant, assume that `sizeof(int)` is 4, `sizeof(char)` is 1, and `sizeof()` any pointer type is 4. Treat each expression as independent.

Expression	Value
`A + B`	
`A / B`	
`++A+B++`	
`*grade + 2`	
`cp[3]`	
`&grade[3] - &grade[2]`	
`(int)&grade[3] - (int)&grade[2]`	
`(A=0)?A:B`	
`strlen(cp)`	
`sizeof(grade)`	

Bonus: (requires explanation)

`strlen((char*)(&grade[3]))`	

3. () (Warm up) Given the C variables defined below, for each of the following expressions, give its value.

```
int  A = 3;    char  *cp = "acebdf";
int  B = 2;    int   grade[6] = { 4, 2, 1, 0, 1, -1 };
```

If it is relevant, assume that `sizeof(int)` is 4, `sizeof(char)` is 1, and `sizeof()` any pointer type is 4. Treat each expression as independent.

Expression	Value
`A - B`	
`B / A`	
`++A + ++B`	
`*(grade + 2)`	
`cp[3]`	
`&grade[4] - &grade[0]`	
`(int)&grade[4] - (int)&grade[0]`	
`(A=1)?B:A`	
`strlen(cp)`	
`sizeof(cp)`	

4. () (Warm up) Given the C variables defined below, for each of the following expressions, give its value.

```
int   A = 3;     char  letters[5] = { 'a', 'b', '\0', 'd', 'e' };
int   B = 5;     int   grades[5] = { 90, 80, 70, 60, 50 };
                 char  *cp = letters+5;
```

If it is relevant, assume that `sizeof(int)` is 4, `sizeof(char)` is 1, and `sizeof()` any pointer type is 4. Treat each expression as independent.

Expression	Value
`B / A`	
`--B-A--`	
`*grades`	
`cp[-1]`	
`grades[3] - grades[1]`	
`&grades[3] - &grades[1]`	
`(int)&grades[3] - (int)&grades[1]`	
`(A-3)?'y':'n'`	
`strlen(letters)`	
`sizeof(letters)`	

5. () (Warm up) Given the C variables defined below, for each of the following expressions, give its value.

```
int  A = 2;    char  letters[6] = { 'a', 'b', 'c', '\0', 'd', 'e' };
int  B = 4;    int   sizes[6] = { 4, 5, 6, 7, 8, 9 };
               char  *str = "string";
```

If it is relevant, assume that `sizeof(int)` is 4, `sizeof(char)` is 1, and `sizeof()` any pointer type is 4. Treat each expression as independent.

Expression	Value
`A % B`	
`letters[4]`	
`*sizes+2`	
`strlen(str+2)`	
`sizes[5] - sizes[3]`	
`&sizes[5] - &sizes[3]`	
`(int)&sizes[3] - (int)&sizes[1]`	
`strlen(str)+2`	
`strlen(letters)`	
`sizeof(sizes)`	

6. () (Warm up) Given the C variables defined below, for each of the following expressions, give its value.

```
int  A = 0;     char  letters[6] = { 'c', 'p', '\0', 's', 'l', 'o' };
int  B = 8;     int   numbers[5] = { 9, 3, 4, 0 , 7 };
                char  *str = "c-quiz";
```

If it is relevant, assume that `sizeof(int)` is 4, `sizeof(char)` is 1, and `sizeof()` any pointer type is 4. Treat each expression as independent.

Expression	Value
`A % B`	
`letters[4]`	
`*numbers + 4`	
`sizeof(str)`	
`numbers[3] - numbers[0]`	
`&numbers[3] - &numbers[0]`	
`(int)&numbers[3] - (int)&numbers[0]`	
`strlen(str+2)`	
`strlen(letters)`	
`sizeof(numbers)`	

7. () (Warm up) Given the C variables defined below, for each of the following expressions, give its value.

```
int  A = 3;    char  letters[7] = { 'c', 'p', 'e', '\0', '3', '5', '7' };
int  B = 4;    int   scores[] = { 32, 34, 50, 49, 22, 14 };
               char  *name = "Fall 2009";
```

If it is relevant, assume that `sizeof(int)` and `sizeof(long)` are 4, `sizeof(char)` is 1, and `sizeof()` any pointer type is 4. Treat each expression as independent.

Expression	Value
`A = B`	
`scores[4] + 1`	
`*scores+3`	
`sizeof(*(name+2))`	
`scores[5] - scores[2]`	
`&scores[5] - &scores[2]`	
`(long)&scores[3] - (long)&scores[0]`	
`strlen(name+2)`	
`sizeof(name)`	
`sizeof(scores + 2)`	

8. () People learning the C language often say that "arrays and pointers are the same thing." This is a common misconception, but like many misconceptions, there is an element of truth to it.

   (a) State one common property of arrays and pointers.

   (b) State one significant difference between arrays and pointers.

9. () Given the following defintions:

```
#define one(x) x + x
#define two(a) (a * x)
int  x = 6;
int  y = 7;
```

what is the value of each of the following expressions:

Expression	Value	Explanation
`one(2)`		
`two(y++)`		
`one(2) + one(2)`		
`two(2 + 2)`		

10. () Given the following defintions:

```
#define a(x) (x) + (x)
#define b(c,d) (d)
int  x = 2;
int  y = 3;
```

what is the value of each of the following expressions:

Expression	Value	Explanation
`a(1)`		
`b((y*y),y)`		
`((6) * a(2))`		
`a(b(1,2))`		

11. () Given the following defintions:

```
#define one(x,y) (x + y)
#define two(a) (x * x)
int  x = 3;
int  y = 4;
```

what is the value of each of the following expressions:

Expression	Value	Explanation
`one(2,3)`		
`one(2*3,3*2)`		
`two(two(2))`		
`two(one(++x,0))`		

12. () Given the following defintions:

```
int three(int x) {
  return x+x+x;
}
#define one(x) (x * x)
#define two(y,x) ((x)?(x):(y))
int  x = 3;
int  y = 4;
```

what is the value of each of the following expressions:

Expression	Value	Explanation
`one(2+3)`		
`two(x,y)`		
`three(++x)`		
`one(two(0,1))`		

13. () Given the following defintions:

```
#define fun(x) (x * x)
#define two(y,x) (y)
int  x = 3;
int  y = 4;
```

```
int one(int x) {
  return x+x;
}
```

what is the value of each of the following expressions:

Expression	Value	Explanation
`one(2+3)`		
`two(x,y)`		
`fun(x)`		
`one(two(0,1))`		

14. () Given the following defintions:

```
#define nonce(x) x - x
#define div(x,y) (x / y)
int  x = 2;
int  y = 3;
```

```
int twice(int x) {
  return x+x;
}
```

what is the value of each of the following expressions:

Expression	Value	Explanation
`nonce(x+y)`		
`twice(x+y)`		
`div(1, 2.0)`		
`div(twice(nonce(y)), nonce(twice(x)))`		

If you need to, use the back of the previous page for notes.

15. () Given the following defintions:

```
#define twice(x) (x) + (x)
#define nonce(x) (3)
int  x = 6;
int  y = 7;
```

what is the value of each of the following expressions:

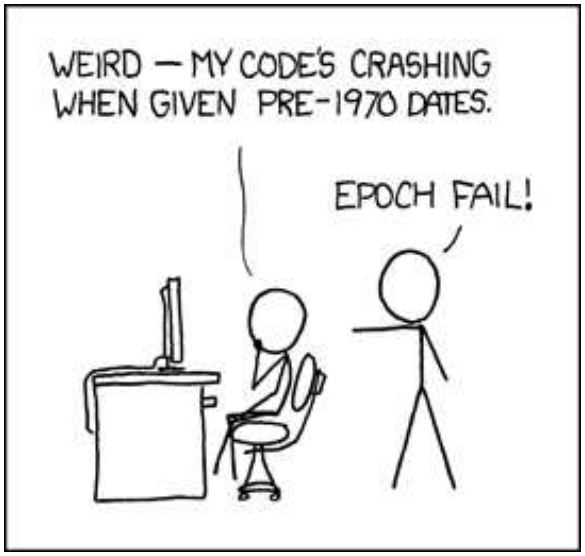
Expression	Value	Explanation
`twice(2)`		
`twice(y++)`		
`twice(2) * twice(2)`		
`twice(nonce(x))`		

16. () While we're on the subject of macros, the C stdio library contains the following definitions:

```
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
```

Each one writes a character to the given stream. The *only* difference between the two is that `fputc()` is guaranteed to be a function while `putc()` may be implemented as a macro. What is the purpose of having both?

17. () **What's wrong with this picture?** The comic below is amusing, but, unfortunately, incorrect. Why?



(The universe started in 1970. Anyone claiming to be over 38 is lying about their age. (`http://xkcd.com/376/`))

18. () The stdio function `getchar()` reads a character from stdin and, on success, returns it. On failure, `getchar()` returns `EOF` which is defined to be -1. In spite of the fact that it's reading chars, `getchar()` returns an int. Why? (and explain)

19. () What is the meaning of an *extern* declaration in C?

20. () Why is it important always to use the `sizeof()` operator when allocating space for a structure?

21. () Given an implementation of `fw` implemented (at the last minute, of course) using an ordinary binary tree and the two invocations:

    a) `% fw /usr/dict/words /usr/man/*/*`
    b) `% fw /usr/man/*/* /usr/dict/words`

    Which would you expect to complete its execution more quickly and why?

22. () What is the meaning of a *static* declaration in C?

23. () It is said, "Never include anything in a header file that either allocates memory or defines (rather than declares) a function." Why would this be a problem?

24. () Implement a C function `Amin()` that takes an array of integers `A` and its length `len` and returns the smallest integer in `A`. You may assume that you will receive proper arguments and that `A` has at least one element.

```
int Amin(int A[], int len){

}
```

25. () Implement a C function `dot()` that takes two arrays of doubles, `A` and `B`, as well as their length, `len`, and returns their dot product. Recall that the dot product of two vectors is defined as

$$A \cdot B = \sum_{i=0}^{len-1} a_i b_i$$

You may assume that you will receive proper arguments and that `A` and `B` have at least one element.

```
double dot(double A[], double B[], int len){

}
```

26. () Why is it important always to use the `sizeof()` operator when allocating space for a structure?

27. () UNIX system calls (and many other C functions) return zero to indicate success and a non-zero value to indicate failure. Given that zero in C is interpreted as false and non-zero is interpreted as true, this convention seems misguided. Give one good reason why it is a good idea to do it this way.

28. () The C library function `gets()`:

    ```
    char *gets(char *s)
    ```

    reads a line from stdin into the buffer pointed to by `s` until either a terminating newline or EOF, which it replaces with '\0'.

    Even the man page says "Never use `gets()`." Why? What is the danger of using this function?

29. () Implement a C function `sum()` that takes two integers `x` and `y` such that $x \leq y$ and returns the sum of all integers from x to y, inclusive. You may assume that you will receive proper arguments and do not need to be concerned about overflow.

```
int sum(int x, int y){

}
```

30. () Implement the C library function `strpbrk()`:

Name:

```
char *strpbrk(const char *s, const char *accept);
```

Description:

```
The strpbrk() function locates the first occurrence in the string s of any
of the characters in the string accept.
```

Return Value:

```
The strpbrk() function returns a pointer to the character in s that matches
one of the characters in accept, or NULL if no such character is found.
```

Write robust code (even though the library version is fragile). That is, return `NULL` on failure, but do not crash. Think before you write anything.

```
char *strpbrk(const char *s, const char *accept){
```

Optional extra room for problem 39

```
}
```

31. () Implement a robust version of the C library function `strspn()`:

    Name:

    `size_t strspn(const char *s, const char *accept);`

    Description:

    `The strspn() function calculates the length of the initial segment of s which consists entirely of characters in accept.`

    Return Value:

    `The strspn() function returns the number of characters in the initial segment of s which consist only of characters from accept, or −1 if it is unable to complete its task.`

    Write robust code (even though the library version is fragile). That is, return -1 on failure, but do not crash. Think before you write anything.

```
size_t strspn(const char *s, const char *accept){
```

Optional extra room for problem 39

32. () A novice programmer wrote the following fragment of code for a new game destined to be a mega-hit. At the next code review, the project manager looked at the program, announced, "I will not be responsible for this kind of work!" and quit on the spot (in today's economy, no less!)

```
[...]
void ask_the_question(NODE node) {
  char *question;
  char c;
  /* build the query */
  question=strcat("Is it ",node->str);
  /* print the query */
  printf("%s?",question);
  c = get_answer();
  [...]
}
```

What is wrong with the code fragment? (Hint: It compiles fine, and you don't have to make any assumptions about any other code.)

33. () In ANSI C, the `const` qualifier can be applied to a variable declaration to indicate that its value will not be changed. Consider the following two function prototypes for error-handling routines:

`char *strerr(int errnum)`	return string describing error code
`void perror(const char *s);`	print a system error message

In these two prototypes, `perror()`'s parameter has the `const` attribute, while `strerr()`'s does not. Why?

34. () Given the following structure definition:

```
struct node_st {
    int data;
    struct node_st *next;
};
```

Write a C function, `sorted_insert_node()`, that takes a pointer to a NULL-terminated sorted linked list (possibly empty) made up of these structures and an integer, `num`. `Sorted_insert_node()` creates a new node with data field of `num` and inserts it into the list in sorted order and returns a pointer to the head of the modified list. The list is sorted in ascending numerical order of the nodes' `data` fields.

```
struct node_st *sorted_insert_node(struct node_st *l, int num) {
```

Optional extra room for problem 37

}

35. () Implement the Linux C library function `strfry()` that takes a string `s` (possibly NULL) and randomizes its contents by randomly swapping characters. The result is an anagram of the given string. The function returns a pointer to the head of the processed string.

    You may (will) find the library function `rand()` helpful. `Rand()` returns a randomly selected integer in the range 0–`RAND_MAX` (defined in `stdlib.h`).

    Write robust code. That is, return `NULL` on failure, but do not crash. Think before you write anything.

    ```
    char *strfry(char *s) {
    ```

Optional extra room for problem 39

```
}
```

36. () Implement the C library function `strstr()` that takes two strings `needle` and `haystack` (possibly NULL) and returns a pointer to the first occurance of `needle` in `haystack` or NULL if the substring is not found.

    Write robust code. That is, return `NULL` on failure, but do not crash. Think before you write anything.

    ```
    char *strstr(char *haystack, char *needle){
    ```

Optional extra room for problem 39

```
}
```

37. () Given the following structure definition and typedef:

```
typedef struct node_st node;
struct node_st {
    int data;
    node *next;
};
```

Write a C function, `remove_item()`, that takes an integer argument, `num`, and a linked list, `list`, (possibly NULL) of these structures. `remove_item()` removes the first node of the list with the given value, if any, and returns a pointer to the head of the modified list. If a node is removed, it should be free()d.

Write robust code.

```
node *remove_item(int num, node *list){
```

Optional extra room for problem 46

}

38. () Given the following structure definition and typedef for a linked list of strings:

```
typedef struct node_st node;
struct node_st {
    char *word;      /* a valid string pointer or NULL */
    node *next;      /* next node in the list or NULL */
};
```

Write a C function, `free_list()`, that takes as an argument one of these lists, possibly NULL, and frees all the strings as well as the list itself.

Write robust code.

```
void free_list(node *list){
```

Optional extra room for problem 46

}

39. () Write a C function, `duplicate_string()` that takes a string as its parameter and returns a pointer to a newly allocated region of memory containing a copy of the string. Do not use any of the C library's string functions.

```
char *duplicate_string(char *s) {
```

}

40. () Given the following structure definition:

```
struct node_st {
    int data;
    struct node_st *next;
};
```

Write a C function, `reverse_list()`, that takes a pointer to a NULL-terminated linked list made up of these structures, reverses the list, and returns a pointer to the new head (the former tail). If the given list is `NULL`, `reverse_list()` should return `NULL`.

```
struct node_st *reverse_list(struct node_st *l) {




}
```

41. () Write a C function, `copy_append_string()` that takes a two strings, `s` and `t`, possibly `NULL`, as its parameters and returns a pointer to a newly allocated region of memory containing a string that is the concatenation of the two. (That is,

    ```
    copy_append_string("Hi ","There")
    ```

    would return a pointer to `"Hi There"`).

    Write robust code. That is, return `NULL` on failure, but do not crash. Think before you write anything.

    ```
    char *copy_append_string(char *s, char *t) {
    ```

Optional extra room for problem 44

}

42. () Given the following structure definition:

```
struct node_st {
    char *string;
    struct node_st *next;
};
```

Write a C function, `stir_list()`, that takes a pointer to a NULL-terminated linked list (possibly empty) made up of these structures containing strings. It makes a new list of anagrams of the given strings. (The anagrams can be created using `strfry()`, defined in problem 39.) `Stir_list()` returns a pointer to the head of the new list. The original list of strings should be left unchanged.

Write robust code.

```
struct node_st *stir_list(struct node_st *list) {
```

Optional extra room for problem 45

}

43. () Given the following structure definition:

```
struct node_st {
    int data;
    struct node_st *next;
};
```

Write a C function, `sorted_insert_list()`, that takes an integer argument, `num`, and a linked list, `list`, (possibly NULL) of these structures sorted in ascending order. `sorted_insert_list()` creates a new node with the given value and inserts it in the list so that all the data values in the list remain sorted. `sorted_insert_list()` returns a pointer to the head of the new list.

Write robust code.

```
struct node_st *sorted_insert_list(int num, struct node_st *list){
```

Optional extra room for problem 46

}

44. () Given the following structure definition:

```
struct node_st {
    int data;
    struct node_st *next;
};
```

Write a C function, `print_list_backwards()`, that takes a pointer to a NULL-terminated linked list made up of these structures and prints out the values in reverse order, one per line. (If you are unable to print the list backwards, do it forwards for **half credit**.)

```
void print_list_backwards(struct node_st *l) {
```

}

45. () Write a C function, `extract_substring()` that takes a string, `s`, and two integers, `i` and `j` as its parameters and returns a pointer to a newly allocated region of memory containing a copy of the substring of `s` from `s[i]` to `s[j]`, inclusive. If either `i` or `j` lie outside of `s`, the end of the substring is the appropriate end of `s`. If `i` is greater than `j`, the substring will be reversed.

    Write robust code. That is, return `NULL` on failure, but do not crash under any circumstances. Think before you write anything; it's easier than it sounds.

    ```
    char *extract_substring(char *s, int i, int j) {
    ```

Optional extra room for problem 48

```
}
```

Useful Information

Selected Useful Prototypes

```
void *   calloc(size_t nmemb, size_t size);
int      fclose(FILE *stream);
FILE *   fdopen(int fildes, const char *mode);
int      feof( FILE *stream);
int      fgetc(FILE *stream);
char *   fgets(char *s, int size, FILE *stream);
FILE *   fopen(const char *path, const char *mode);
int      fprintf(FILE *stream, const char *format, ...);
int      fputc(int c, FILE *stream);
int      fputs(const char *s, FILE *stream);
void     free(void *ptr);
FILE *   freopen(const char *path, const char *mode, FILE *stream);
int      getc(FILE *stream);
int      getchar(void);
char *   gets(char *s);
char *   index(const char *s, int c);
int      isalnum(int c);
int      isalpha(int c);
int      isascii(int c);
int      isblank(int c);
int      iscntrl(int c);
int      isdigit(int c);
int      isgraph(int c);
int      islower(int c);
int      isprint(int c);
int      ispunct(int c);
int      isspace(int c);
int      isupper(int c);
int      isxdigit(int c);
void *   malloc(size_t size);
void     perror(const char *s);
int      printf(const char *format, ...);
int      putc(int c, FILE *stream);
int      putchar(int c);
int      puts(const char *s);
void *   realloc(void *ptr, size_t size);
int      rand(void);
char *   rindex(const char *s, int c);
int      snprintf(char *str, size_t size, const char *format, ...);
int      sprintf(char *str, const char *format, ...);
char *   strcat(char *dest, const char *src);
char *   strchr(const char *s, int c);
int      strcmp(const char *s1, const char *s2);
char *   strcpy(char *dest, const char *src);
int      strlen(const char *s);
char *   strerror(int errnum);
char *   strncat(char *dest, const char *src, size_t n);
int      strncmp(const char *s1, const char *s2, size_t n);
char *   strncpy(char *dest, const char *src, size_t n);
char *   strrchr(const char *s, int c);
char *   strstr(const char *haystack, const char *needle);
int      ungetc(int c, FILE *stream);
int      tolower(int c);
int      toupper(int c);
```