Midterm Exam

STUDENT NAME

Search students by name or email...

Q1 Instructions and Pledge

1 Point

This exam consists of 6 multi-part questions (plus the pledge), and you have 60 minutes — budget your time wisely.

This is a closed-book, closed-note exam, and "cheat sheets" are not allowed. During the exam you must not refer to the textbook, course materials, notes, or any information on the Internet. You may not compile or run any code on armlab or any other machine.

You are not allowed to communicate with any other person, whether inside or outside the class. You may not send the exam problems to anyone, nor receive them from anyone, nor communicate any information about the problems or their topics. *If you have technical issues or need to ask a clarifying question about the wording of some problem, please post a* ***private*** *message on Ed.*

You may use blank paper as scratch space, but you must enter your answer in the online system in order to receive credit.

This examination is administered under the Princeton University Honor Code, and by signing the pledge below you promise that you have adhered to the instructions above.

Please type out the Honor Code pledge exactly as follows,

including this exact spelling and punctuation:

> *I pledge my honor that I have not violated the Honor Code during this examination.*

Enter your answer here

Now type your name as a signature confirming that you have adhered to the Honor Code:

Enter your answer here

Save Answer

Q2 Yes, it's the usual binary arithmetic question...

6 Points

Consider the following binary addition problem on 4-bit quantities:

$$1011_{\mathrm{B}} + 1101_{\mathrm{B}} = ?$$

Q2.1

1 Point

What is the ***4-bit binary*** result of the addition?
(Write only the 4 bits, with no extra white space, punctuation, or text.)

Enter your answer here

Save Answer

Q2.2
1 Point

Interpreting the 4-bit result as an ***unsigned*** number, what is its *decimal* value?
(Write only the value, with no extra white space, punctuation, or text.)

Enter your answer here

Save Answer

Q2.3
1 Point

If the addition is interpreted as ***unsigned***, it overflows:

- ○ True
- ○ False

Save Answer

Q2.4
1 Point

Interpreting the result as a 4-bit two's complement ***signed*** number, what is its *decimal* value?
(Write only the value, with no extra white space, punctuation, or text.)

Enter your answer here

Save Answer

Q2.5

1 Point

If the addition is interpreted as ***signed***, it overflows:

- ○ True
- ○ False

Save Answer

Q2.6

1 Point

Overflow in an addition results in:

- ○ A compiler error
- ○ A compiler warning
- ○ A run-time crash
- ○ None of the above

Save Answer

Q3 Mismanaged memory

4 Points

Consider the following code:

```
1  int main()
2  {
3      int *numbers;
4      numbers = malloc(sizeof(int));
5      numbers = calloc(10, sizeof(int));
6      numbers = realloc(numbers, 20 * sizeof(int));
7      return 0;
8  }
```

As a reminder, the memory allocation functions have the following calling conventions:

```
void *malloc(size_t size);
void *calloc(size_t n_elements, size_t size_per_element);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Q3.1

2 Points

Assuming that all appropriate header files have been included, and that ***all calls to memory management functions succeed***, how many bytes of memory are leaked by the original code?

- ○ 0
- ○ 4
- ○ 40
- ○ 44
- ○ 80
- ○ 84
- ○ 124

Save Answer

Q3.2

2 Points

Which statement/statements need to be added to prevent ***all*** memory leaks?

- Add `free(numbers);` immediately after line 4
- Add `free(numbers);` immediately after line 5
- Add `free(numbers);` immediately after line 6
- Add `free(numbers);` immediately after line 4 and line 5
- Add `free(numbers);` immediately after line 4 and line 6
- Add `free(numbers);` immediately after line 5 and line 6

Save Answer

Q4 A hop, skip, and a jump

5 Points

Consider the following code:

```
1   struct Node {
2       int val;
3       struct Node *next;
4   };
5
6   struct List {
7       struct Node *first;
8   };
9
10  void List_insert(struct List *list, int new_val)
11  {
12      struct Node *new_node = malloc(sizeof(struct Node));
13      new_node->val = new_val;
14      new_node->next = list->first;
15      list->first = new_node;
16  }
17
18  void mystery(struct Node *node)
19  {
20      if (!node)
21          return;
22      printf("%d", node->val);
23      if (node->next)
24          mystery(node->next->next);
25      printf("%d", node->val);
```

```
26  }
27
28  int main()
29  {
30      struct List l = { NULL };
31      List_insert(&l, 1);
32      List_insert(&l, 2);
33      mystery(l.first);
34      List_insert(&l, 3);
35      mystery(l.first);
36      return 0;
37  }
```

Q4.1

1 Point

The implementation of `List_insert` runs in:

- ○ Constant time
- ○ Linear time (i.e., proportional to the number of items in the list)
- ○ Quadratic time

Save Answer

Q4.2

1 Point

The implementation of `List_insert` produces a list containing values in the *reverse* of the order in which they were inserted:

- ○ True
- ○ False

Save Answer

Q4.3

1 Point

Without changing the definitions of `struct Node` and `struct List`, it is possible to change `List_insert` such that it runs in constant time *and* produces a list containing values in the *same* order in which they were inserted:

- ○ True
- ○ False

Save Answer

Q4.4
1 Point

What is printed by the call to `mystery` on line 33?
(Write only the digits printed, with no extra white space, punctuation, or text. If the code crashes or enters an infinite loop, write "crash".)

Enter your answer here

Save Answer

Q4.5
1 Point

What is printed by the call to `mystery` on line 35?
(Write only the digits printed, with no extra white space, punctuation, or text. If the code crashes or enters an infinite loop, write "crash".)

Enter your answer here

Save Answer

Q5 Big cat

6 Points

For this problem, you will be using some `<string.h>` functions, which have the following signatures:

```
char *strcat(char *dest, const char *src);
char *strcpy(char *dest, const char *src);
int strcmp(const char *s1, const char *s2);
size_t strlen(const char *s);
char *strstr(const char *haystack, const char *needle);
```

Consider writing a function that concatenates *two* source strings, in order, onto a destination string. We shall call it `strtiger` (because it is bigger than just `strcat` (*groan*)).

Here is one implementation, to show the desired behavior in code:

```
char *strtiger1(char *dest, const char *src1, const char *src2)
{
    return strcat(strcat(dest, src1), src2);
}
```

Q5.1

1 Point

Now you want to write a version of `strtiger` that doesn't rely on `strcat`. Fill in the blanks with the appropriate expressions.

```
char *strtiger2(char *dest, const char *src1, const char *src2)
{
    char *p = /* __blank 1__ */;
    while (*p != '\0')
        p++;
    /* __blank 2__ */(p, src1);
    p += /* __blank 3__ */;
    strcpy(p, src2);
    return dest;
}
```

What should go in `__blank 1__`?

- `src1`
- `dest`
- `*src1`
- `*dest`
- `&src1`
- `&dest`

Save Answer

Q5.2

1 Point

What should go in `__blank 2__`?

- `strcpy`
- `strcmp`
- `strlen`
- `strstr`

Save Answer

Q5.3

1 Point

What should go in `__blank 3__`?

- `strlen(src1) - 1`
- `strlen(src1)`
- `strlen(src1) + 1`
- `strlen(src2) - 1`
- `strlen(src2)`
- `strlen(src2) + 1`

Save Answer

Q5.4

3 Points

We now have a skeleton of a third version of `strtiger`:

```
char *strtiger3(char *dest, const char *src1, const char *src2)
{
    char *p = dest;
    /* Insert code here */
    return dest;
}
```

but the lines that go in the middle have gotten scrambled:

```
1  while (*p++ = *src1++) ;
2  while (*p++ = *src2++) ;
3  while (*p)
4  p--;
5  p++;
```

Write the correct permutation of these 5 lines that would make the code work.
(Write only the 5 numbers representing the order of lines — e.g., 12345 or 54321 — with no extra white space, punctuation, or text.)

Enter your answer here

Save Answer

Q6 Perplexing pointers and silly strings

6 Points

Consider the following code:

```
char princeton[] = "Tigers!";
const char *cos = "217!";
char *exam[9];
*exam = malloc(9);
strcpy(*exam, "Midterm!");
*(exam + 1) = (*exam) + 1;
```

Assuming that all appropriate header files have been included, and that the call to `malloc` succeeds, answer the following questions. ***Hint: definitely grab a sheet of scratch paper and draw out the variables and pointers! Also, read the definition of `exam` carefully!***

Q6.1

1 Point

What section of memory contains the characters `217!`?

- heap
- rodata
- stack
- text
- more than one of these

Save Answer

Q6.2

1 Point

What section of memory contains the characters `Midterm!`?

- ○ heap
- ○ rodata
- ○ stack
- ○ text
- ○ more than one of these

Save Answer

Q6.3

1 Point

For each of the following expressions, indicate whether it results in a compiler error or warning, or what it evaluates to otherwise. ***Hint: the compiler will warn about comparisons between different pointer types.***

`princeton[6] == cos[3]`

- ○ Results in a compiler error or warning
- ○ Evaluates to `0` (FALSE)
- ○ Evaluates to `1` (TRUE)

Save Answer

Q6.4

1 Point

`exam[7] == cos[3]`

- ○ Results in a compiler error or warning
- ○ Evaluates to `0` (FALSE)
- ○ Evaluates to `1` (TRUE)

Save Answer

Q6.5
1 Point

`*(exam + 1) == (princeton + 1)`

- Results in a compiler error or warning
- Evaluates to `0` (FALSE)
- Evaluates to `1` (TRUE)

Save Answer

Q6.6
1 Point

`**(exam + 1) == *(princeton + 1)`

- Results in a compiler error or warning
- Evaluates to `0` (FALSE)
- Evaluates to `1` (TRUE)

Save Answer

Q7 !sgub eht dniF
5 Points

The following functions are each intended to print a C string ***backwards***. They all use the `putchar` standard library function to print characters to `stdout` — it has the following signature:

```
int putchar(int c);
```

Assume that `pc` is not `NULL`, and points to a correctly null-

terminated string whose length fits into an `int`. For each function, indicate whether it succeeds, or how it fails.

Q7.1
1 Point

```
void fun1(const char *pc)
{
    int i;
    for (i = strlen(pc) - 1; i >= 0; i--)
        putchar(pc[i]);
}
```

- ○ Results in a compiler error.
- ○ Compiles and runs, but accesses memory it shouldn't and may crash at run-time.
- ○ Runs and terminates, but produces incorrect output.
- ○ Produces correct output, but is grossly inefficient (by more than a constant factor).
- ○ Runs correctly and efficiently (within a constant factor of optimal).

Save Answer

Q7.2
1 Point

```
void fun2(const char *pc)
{
    int i;
    int n = strlen(pc);
    for (i = n; i >= 0; i--)
        putchar(pc[i]);
}
```

- Results in a compiler error.
- Compiles and runs, but accesses memory it shouldn't and may crash at run-time.
- Runs and terminates, but produces incorrect output.
- Produces correct output, but is grossly inefficient (by more than a constant factor).
- Runs correctly and efficiently (within a constant factor of optimal).

Save Answer

Q7.3

1 Point

```
void fun3(const char *pc)
{
    int i;
    int n = strlen(pc - 1);
    for (i = n; i > 0; i--)
        putchar(pc[i]);
}
```

- Results in a compiler error.
- Compiles and runs, but accesses memory it shouldn't and may crash at run-time.
- Runs and terminates, but produces incorrect output.
- Produces correct output, but is grossly inefficient (by more than a constant factor).
- Runs correctly and efficiently (within a constant factor of optimal).

Save Answer

Q7.4

1 Point

```
void fun5(const char *pc)
{
    int i;
    int n = strlen(pc) - 1;
    for (i = n; i >= 0; i--)
        putchar((*pc) + i);
}
```

- Results in a compiler error.
- Compiles and runs, but accesses memory it shouldn't and may crash at run-time.
- Runs and terminates, but produces incorrect output.
- Produces correct output, but is grossly inefficient (by more than a constant factor).
- Runs correctly and efficiently (within a constant factor of optimal).

Save Answer

Q7.5

1 Point

```
void fun6(const char *pc)
{
    int i;
    for (i = 0; i < strlen(pc); i++)
        putchar(pc[strlen(pc) - i - 1]);
}
```

- ○ Results in a compiler error.
- ○ Compiles and runs, but accesses memory it shouldn't and may crash at run-time.
- ○ Runs and terminates, but produces incorrect output.
- ○ Produces correct output, but is grossly inefficient (by more than a constant factor).
- ○ Runs correctly and efficiently (within a constant factor of optimal).

Save Answer

Save All Answers

Submit & View Submission ›