This exam has 5 questions, some with several parts. You have 50 minutes, so budget your time. Do all of your work on these pages and give the answer in the space provided. Assume the arm\lab/Linux/C/gcc217 environment unless otherwise stated. This is a closed-book, closed-note exam, and “cheat sheets” are not allowed. Please place all items that you will not need out of view in your bag or under your working space at this time. Electronic devices such as cell phones, PDAs, laptops, MP3 players, iPods, etc. may not be used during this exam.

Sample Solutions

Name: [Name]  NetID: [NetID]

Precept (circle one): 1: MW 1:30 X. Li  2: MW 3:30 X. Li  3: MW 7:30 A. Mizrahi  4: TTh 12:30 D. Gabai  5: TTh 1:30 D. Gabai  6: TTh 3:30 J. Zhang

This examination is administered under the Princeton University Honor Code. Students should sit one seat apart from each other and refrain from talking to other students during the exam. All suspected violations of the Honor Code must be reported to honor@princeton.edu.

Write out and sign the Honor Code pledge before turning in the test:

“I pledge my honor that I have not violated the Honor Code during this examination.”

Pledge: [Signature]

<table>
<thead>
<tr>
<th>Question # and Theme</th>
<th>Available points</th>
<th>Points earned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 You get the forms, I'll prepare them...</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1 What would you say you do here?</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>2 I can't describe it. It was just constant.</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3 Would you mind giving me some pointers here?</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4 This is a completely different dynamic.</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

Page 1 of 10
0. You get the forms, I'll prepare them... (1 point)
Fill out the identifying information on the front sheet correctly and in its entirety: name, NetID, section. Write out the honor pledge. When you have completed the exam, sign the honor pledge.

1. What would you say you do here? (28 points)
(1a – 5 points) What is printed to stdout by each of the following printf calls? You should assume that these are within the main function of a file that begins with #include <stdio.h>. Circle your final answer for each call.

```c
printf("%d", 0x21 + 7 + 0217);
183
0x21 is 2*16+1=33. 0217 is 2*64+1*8+7=143. 33+7+143=183.
printf("%d", ~-1);
0
-1: 32 1 bits. ~ flips all bits, so 32 0 bits. This is 0.
printf("%d", ~-1);
2
1: 31 0s then a 1. ~ flips all, 31 1s then a 0. That’s –2. –(–2) is 2.
printf("%d", ~(255 < 8));
-1
255 < 8 is false, so 0. 0 is 32 0 bits. So ~0 is 32 1 bits. That’s –1.
printf("%d", 2 ^ ~-2);
-4
2 is 30 0s then 10. –2 is 30 1s then 10. ^ them: 30 1s then 00. That’s –4.
```

(1b – 10 points) Compare and contrast the given literals by filling in the table below. For type give the appropriate C type (e.g. double). For value, if the literal’s value is 0, write ZERO; if the literal’s value is strictly between 0 and 128, write SMALL; if it is at least 128, write BIG.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>‘0’</th>
<th>‘\0’</th>
<th>“0”</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>int</td>
<td>char</td>
<td>char*</td>
<td>char*</td>
<td>int / void*</td>
</tr>
<tr>
<td>value</td>
<td>ZERO</td>
<td>SMALL</td>
<td>ZERO</td>
<td>BIG</td>
<td>ZERO</td>
</tr>
<tr>
<td></td>
<td>(ASCII 48)</td>
<td>(RODATA address)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(1c – 5 points) Consider the following program:

```c
#include <stdio.h>
enum mottoWords {
    IN, THE, SERVICE = 1746,
    OF = 1896,
    HUMANITY
};
int main(void) {
    printf("%d %d %d %d %d\n", IN, THE, SERVICE, OF, HUMANITY);
    return 0;
}
```

What values are printed to standard output? You can write GARBAGE to represent an uninitialized value that is printed. Answer in the space below the code box.

0 1 1746 1896 1897

(1d – 8 points) Consider the following program:

```c
#include <stdio.h>
int main(void) {
    char c;
    scanf("%c", &c);
    switch(c) {
    case 'a':
        printf("do ");
        break;
    case 'b':
        printf("re ");
        break;
    default:
        printf("mi ");
        break;
    case 'c':
        printf("fa ");
        break;
    case 'd':
        printf("so ");
        break;
    }
    printf("\n");
    return 0;
}
```

Recall that default is triggered if the switch expression matches none of the case options. What is printed to standard output for each of the five inputs below? Answer beside each corresponding input.

a  do
b  re mi fa
c  fa
d  so
e  mi fa
2. I can't describe it. It was just constant. (8 points)

Consider the following program, which has 8 lines for which gcc217 will emit a warning for discarding a const qualifier. Indicate any 4 of these questionable lines by circling the corresponding line number in the list to the right of the program box. (Do not circle more than 4.)

```c
int* ii(int* bar) {
    return bar;
}

int* ici(const int* bar) {
    return bar;
}

const int* cii(int* bar) {
    return bar;
}

const int* cici(const int* bar) {
    return bar;
}

int main(void) {
    int i;
    int* pi;
    const int* cpi;
    int* r;
    const int* c;
    i = 5;
    pi = &i;
    cpi = &i;
    r = ii(pi);
    r = ii(cpi);
    c = ii(pi);
    c = ii(cpi);
    r = ici(pi);
    r = ici(cpi);
    c = ici(pi);
    c = ici(cpi);
    r = cii(pi);
    r = cii(cpi);
    c = cii(pi);
    c = cii(cpi);
    r = cici(pi);
    r = cici(cpi);
    c = cici(pi);
    c = cici(cpi);
    return 0;
}
```
3. Would you mind giving me some pointers here? (15 points)

(3a – 5 points) Here is an excerpt of Assignment 3’s interface:

```c
/* SymTable_contains returns 1 (TRUE) if oSymTable contains a binding whose key is pcKey, and 0 (FALSE) otherwise. */
int SymTable_contains(SymTable_T oSymTable, const char *pcKey);

/* SymTable_get returns the value of the binding within oSymTable whose key is pcKey, or NULL if there is no binding with that key. */
void* SymTable_get(SymTable_T oSymTable, const char *pcKey);
```

Is it correct (setting aside the lack of assert validation) to implement `SymTable_contains` as:

```c
int SymTable_contains(SymTable_T oSymTable, const char *pcKey) {
    return SymTable_get(oSymTable, pcKey) != NULL;
}
```

If it is correct, then write “YES” and identify (using the terminology from our programming in the large lectures or in your own words) the software engineering principle this represents. If it is not correct, then write “NO” and identify a bug that could result from this implementation.

**NO** – this implementation of `SymTable_contains` would return FALSE for a key that is in the table but bound to the value NULL, for example if we were using a SymTable to implement a set of keys (such that all keys’ corresponding values would be NULL).

(3b – 5 points) Consider the following program, which results in a compilation error:

```c
#include <stdio.h>
int main(void) {
    int i = 'a';
    int* pi = &i;
    void* pv = &i;
    printf("%d\n", *pi);
    printf("%d\n", *pv);
    return 0;
}
```

Which line has the compilation error, and what is wrong with that line?

**Line 7 attempts to use the value obtained from dereferencing a void *, which is not allowed.**

Rewrite the line substantially unchanged except for a fix that will satisfy the compiler:

```c
printf("%d\n", (int*)pv);
```
A fellow COS 217 student has learned about pointers and now wants to use them everywhere! Your friend wrote the following program to read two numbers from stdin and print their sum to stdout:

```c
#include <stdio.h>
int* getInt()
{
    int x;
    scanf("%d", &x);
    return &x;
}
int main(void)
{
    int* p = getInt();
    int* q = getInt();
    printf("%d + %d = %d
", *p, *q, *p + *q);
    return 0;
}
```

Your friend tried two favorite numbers, 76 and 141, but found that the program didn’t work correctly. Worse, adding an `fprintf` that prints to stderr on line 12 resulted in the program printing a completely different (but still wrong) answer!

What was the most likely incorrect output from the program (with inputs 76 and 141) before the `fprintf` was added?

$$141 + 141 = 282$$

What is the bug in the program?
(Hint: think about why adding the call to `fprintf` could change the output from the program.)

`getInt()` returns the address of a local variable that is found in its own stackframe. When we return from a function, that function’s stackframe is no longer “active”, so dereferencing the pointer returned will have unpredictable results, because that memory may be overwritten if there are additional function calls.

(This explains the behaviors: `p` and `q` both point to the same location on the (inactive) stackframe. Before adding the debugging statement, this location (likely) has contents corresponding to the second call’s `x`. The debugging statement uses the same memory locations for `fprintf`’s stackframe, which is why `p` and `q` still point at the same memory location, but that location has a different value.)
4. This is a completely different dynamic. (18 points)

(4a – 6 points) Consider the following program to build an arbitrarily long string from repeated reads from standard input. You may assume that each string provided on stdin will fit in buf, memory allocation never returns NULL, and that size_t will always be sufficient to store the capacity of str.

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

enum { LIMIT = 100, K = 500 }; 

int main(void) {
    char buf[LIMIT];
    size_t length = 0;
    size_t capacity = 0;
    char* str = calloc(capacity+1,sizeof(char));

    while(scanf("%s", buf) == 1) {
        length += strlen(buf);
        if(length >= capacity) {
            capacity = length + (long) K;
            str = realloc(str, capacity+1);
        }
        strcat(str,buf);
    }

    printf("%lu: %s\n", length, str);
    free(str);
    return 0;
}
```

This program passes all our simple tests, passes all our boundary tests, passes all our statement tests, and even passes all our path tests. But it fails a stress test!

In the space below, identify the bug and describe the precise condition that will result in failure. Then, add a simple correction to the program printed above that will eliminate the bug so as to produce a working program under all conditions.

Bug: The pointer returned from realloc is not stored.

Manifests failure: when realloc cannot expand in-place.

Fix: (as above) use the return from realloc as the new pointer to the string.

Explanation: When we allocate memory from the heap, there may or may not be free memory immediately after that allocation. If there is not, when we realloc to a larger size, realloc has to copy our data to a new location and free the old spot. If we don't save a pointer to the new spot, then that memory is lost (memory leak), and str is a dangling pointer to the memory realloc freed but we keep dereferencing it, and when we free str eventually, that's a double free because realloc already freed it!
(4b – 12 points) In Assignment 2, you created two implementations of a string library that mirrored the string.h interface. As a reminder, here were the declarations from your str.h:

```c
size_t Str_getLength(const char* pcSrc);
char* Str_copy(char* pcDest, const char* pcSrc);
char* Str_concat(char* pcDest, const char* pcSrc);
int Str_compare(const char* pcFirst, const char* pcSecond);
char* Str_search(const char* pcHaystack, const char* pcNeedle);
```

The standard library also contains a function `strdup`, with the following signature:

```c
char* strdup(const char* pcSrc);
```

This function does the exact same thing as `strcpy`, except that instead of copying into a space pointed to by a parameter (for which the client must have allocated the space), `strdup` allocates memory dynamically to store the copy of the string.

Expand your string library by implementing `Str_duplicate` as the equivalent of `strdup`. Above your implementation, write a function comment that accords to the COS 217 standards for establishing a contract with clients who will call your function.

Your function should match the general requirements for your Assignment 2 implementation – you may call any of the other functions from your library, but nothing from string.h; your function must not be grossly inefficient: it must not traverse a (potentially long) string more times than necessary; etc. We have given the pointer version of `strdup`’s signature above, but you may choose to use either pointer or array syntax in your implementation.

```c
/*
   Str_duplicate copies the string src into a new string big enough to store the copy, then
   returns the new copy. The caller is then responsible for memory management of the
   returned copy. If insufficient memory is available to allocate the new string, Str_duplicate
   returns NULL.
*/
char* Str_duplicate(const char* src) {
    char* dst;
    assert( src != NULL);
    dst = malloc((Str_getLength(src) + 1) * sizeof(char));
    if(dst == NULL)
        return NULL;
    return Str_copy(dst, src);
}
```

1 pt each: signature; assert; find src’s len; len is size_t; allocate len+1; check malloc’s return; copy src to new string; return new string; comment states: arg by name; behavior; return values; and that client owns new string.
(This page intentionally left blank. You may use it for scratch work or to complete a problem for which you found insufficient space earlier in the exam. In the latter case, please clearly label which problem you are continuing.)
(This page intentionally left blank. You may use it for scratch work or to complete a problem for which you found insufficient space earlier in the exam. In the latter case, please clearly label which problem you are continuing.)

(These are the source movies of the quotations used as problem themes.)
Because some of you will have been thinking about this:
Shawshank Redemption, Office Space, Interstellar, There’s Something About Mary, Gone Girl.