Preecept 22
Week 12, Wed/Thu

1

Princeton University
COS 217: Introduction to Programming Systems
Writing Binary Data

Example 1

We wish to five 0 bytes (alias null characters, alias '\0' characters) to a file named `data`. That is, we wish to write these five bytes to the file:

```
00000000 00000000 00000000 00000000 00000000
```

Open the File
```
FILE *psFile;
psFile = fopen("data", "w");
```

Attempt 1 (Incorrect)
```
fprintf(psFile, "00000");  /* Writes 00110000 00110000 00110000 00110000 00110000 */
```

Attempt 2 (Incorrect)
```
for (i = 0; i < 5; i++)
   fprintf(psFile, "%c", '0');  /* Writes 00110000*/
```

Attempt 3 (Incorrect)
```
for (i = 0; i < 5; i++)
   putc('0', psFile);  /* Writes 00110000 */
```

Attempt 4 (Correct)
```
for (i = 0; i < 5; i++)
   fprintf(psFile, "%c", '\0');  /* Writes 00000000*/
```

Attempt 5 (Correct)
```
for (i = 0; i < 5; i++)
   fprintf(psFile, "%c", 0);  /* Writes 00000000*/
```

Attempt 6 (Correct)
```
for (i = 0; i < 5; i++)
   fprintf(psFile, "%c", 0x00);  /* Writes 00000000 */
```

Attempt 7 (Correct)
```
for (i = 0; i < 5; i++)
   putc('\0', psFile);  /* Writes 00000000 */
```

Attempt 8 (Correct)
```
for (i = 0; i < 5; i++)
   putc(0, psFile);  /* Writes 00000000 */
```

Attempt 9 (Correct)
```
for (i = 0; i < 5; i++)
   putc(0x00, psFile);  /* Writes 00000000 */
```

Close the File
```
fclose(psFile);
```

(2)

Example 2

We wish to write the unsigned long `0x0123456789abcdef` to a file named `data` as it would appear in memory as an eight-byte entity. As humans, we would express the unsigned long `0x0123456789abcdef` in binary like this:

```
00000001 00100011 01000101 01100111 10001001 10101011 11001101 11101111
most sig                                                         least sig
byte                                                             byte
```

But ARMv8 is a little-endian architecture. In the memory of a little-endian computer, the least significant byte of an integer is in the lowest memory location. So the unsigned long `0x0123456789abcdef` appears in memory like this:

```
11101111 11001101 10101011 10001001 01100111 01000101 00100011 00000001
least sig                                                        most sig
byte                                                             byte
```

Or, more precisely, like this:

```
pretend
address
1000      11101111   least sig byte
1001      11001101
1002      10101011
1003      10001001
1004      01100111
1005      01000101
1006      00100011
1007      00000001   most sig byte
```

```
Open the File
FILE *psFile;
psFile = fopen("data", "w");

Attempt 1 (Incorrect)
fprintf(psFile, "0123456789abcdef");
/* Writes 00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111
          00111000 00111001 01100001 01100010 01100011 01100100 01100101 01100110 */

Attempt 2 (Incorrect)
fprintf(psFile, "%x", 0x0123456789abcdef);
/* Writes 00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111
          00111000 00111001 01100001 01100010 01100011 01100100 01100101 01100110 */

Attempt 3 (Correct)
fprintf(psFile, "%c", 0xef);  /* Writes 11101111 */
fprintf(psFile, "%c", 0xcd);  /* Writes 11001101 */
fprintf(psFile, "%c", 0xab);  /* Writes 10101011 */
fprintf(psFile, "%c", 0x89);  /* Writes 10001001 */
fprintf(psFile, "%c", 0x67);  /* Writes 01100111 */
fprintf(psFile, "%c", 0x45);  /* Writes 01000101 */
fprintf(psFile, "%c", 0x23);  /* Writes 00100011 */
fprintf(psFile, "%c", 0x01);  /* Writes 00000001 */

Attempt 4 (Correct)
putc(0xef, psFile);  /* Writes 11101111 */
putc(0xcd, psFile);  /* Writes 11001101 */
putc(0xab, psFile);  /* Writes 10101011 */
putc(0x89, psFile);  /* Writes 10001001 */
putc(0x67, psFile);  /* Writes 01100111 */
putc(0x45, psFile);  /* Writes 01000101 */
putc(0x23, psFile);  /* Writes 00100011 */
putc(0x01, psFile);  /* Writes 00000001 */

Attempt 5 (Correct)             <--- the easiest approach
unsigned long ulData;
...
```

③

```
ulData = 0x0123456789abcdef;
fwrite(&ulData, sizeof(unsigned long), 1, psFile);
/* Writes 11101111 11001101 10101011 10001001 01100111 01000101 00100011 00000001 */
```

Close the File

```
fclose(psFile);
```

Example 3

We wish to write the unsigned int `0x01234567` to a file named `data` as it would appear in memory as a four-byte entity. As humans, we would express the unsigned int `0x01234567` in binary like this:

```
00000001 00100011 01000101 01100111
most sig                   least sig
byte                       byte
```

But ARMv8 is a little-endian architecture. In the memory of a little-endian computer, the least significant byte of an integer is in the lowest memory location. So the unsigned int `0x01234567` appears in memory like this:

```
01100111 01000101 00100011 00000001
least sig                  most sig
byte                       byte
```

Or, more precisely, like this:

```
pretend
address
1000     01100111  least sig byte
1001     01000101
1002     00100011
1003     00000001  most sig byte
```

Open the File

```
FILE *psFile;
psFile = fopen("data", "w");
```

Attempt 1 (Incorrect)

```
fprintf(psFile, "01234567");
/* Writes 00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111 */
```

Attempt 2 (Incorrect)

```
fprintf(psFile, "%x", 0x01234567);
/* Writes 00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111 */
```

Attempt 3 (Correct)

```
fprintf(psFile, "%c", 0x67);  /* Writes 01100111 */
fprintf(psFile, "%c", 0x45);  /* Writes 01000101 */
fprintf(psFile, "%c", 0x23);  /* Writes 00100011 */
fprintf(psFile, "%c", 0x01);  /* Writes 00000001 */
```

Attempt 4 (Correct)

```
putc(0x67, psFile);  /* Writes 01100111 */
putc(0x45, psFile);  /* Writes 01000101 */
putc(0x23, psFile);  /* Writes 00100011 */
putc(0x01, psFile);  /* Writes 00000001 */
```

Attempt 5 (Correct) <--- **the easiest approach**

```
unsigned int uiData;
...
uiData = 0x01234567;
fwrite(&uiData, sizeof(unsigned int), 1, psFile);
/* Writes 01100111 01000101 00100011 00000001 */
```

Close the File

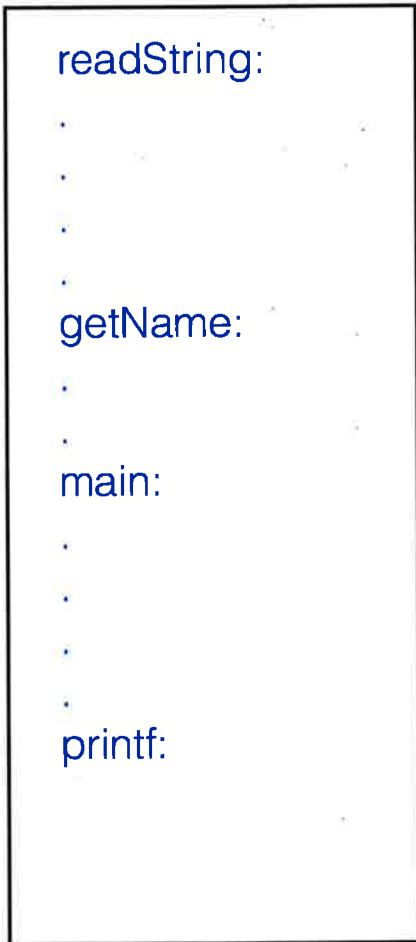
```
fclose(psFile);
```

Copyright © 2019 by Robert M. Dondero, Jr.

grader.c (Page 1 of 1)

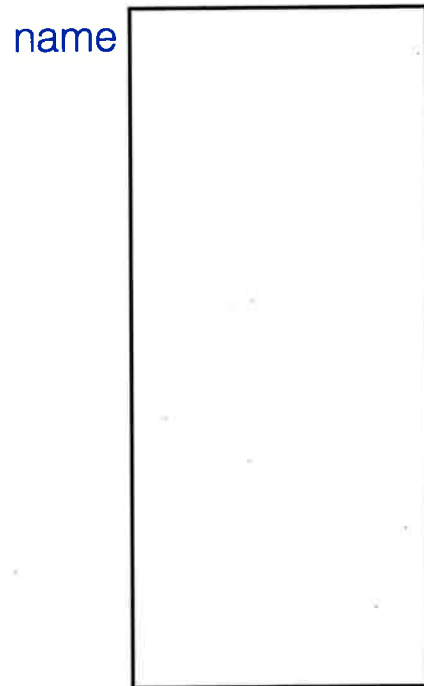
```
 1: #include <stdio.h>
 2: #include <string.h>
 3: #include <stdlib.h>
 4: #include <sys/mman.h>
 5:
 6: enum {BUFSIZE = 48};
 7:
 8: char grade = 'D';
 9: char name[BUFSIZE];
10:
11: void readString(void)
12: {
13:    char buf[BUFSIZE];
14:    int i = 0;
15:    int c;
16:
17:    for (;;)
18:    {
19:       c = fgetc(stdin);
20:       if ((c == EOF) || (c == '\n'))
21:          break;
22:       buf[i] = c;
23:       i++;
24:    }
25:    buf[i] = '\0';
26:
27:    for (i = 0; i < BUFSIZE; i++)
28:       name[i] = buf[i];
29: }
30:
31: void getName(void)
32: {
33:    printf("What is your name?\n");
34:    readString();
35: }
36:
37: int main(void)
38: {
39:    mprotect((void*)((unsigned long)name & 0xfffffffffffff000), 1,
40:             PROT_READ | PROT_WRITE | PROT_EXEC);
41:
42:    getName();
43:
44:    if (strcmp(name, "Andrew Appel") == 0)
45:       grade = 'B';
46:
47:    printf("%c is your grade.\n", grade);
48:    printf("Thank you, %s.\n", name);
49:
50:    return 0;
51: }
```

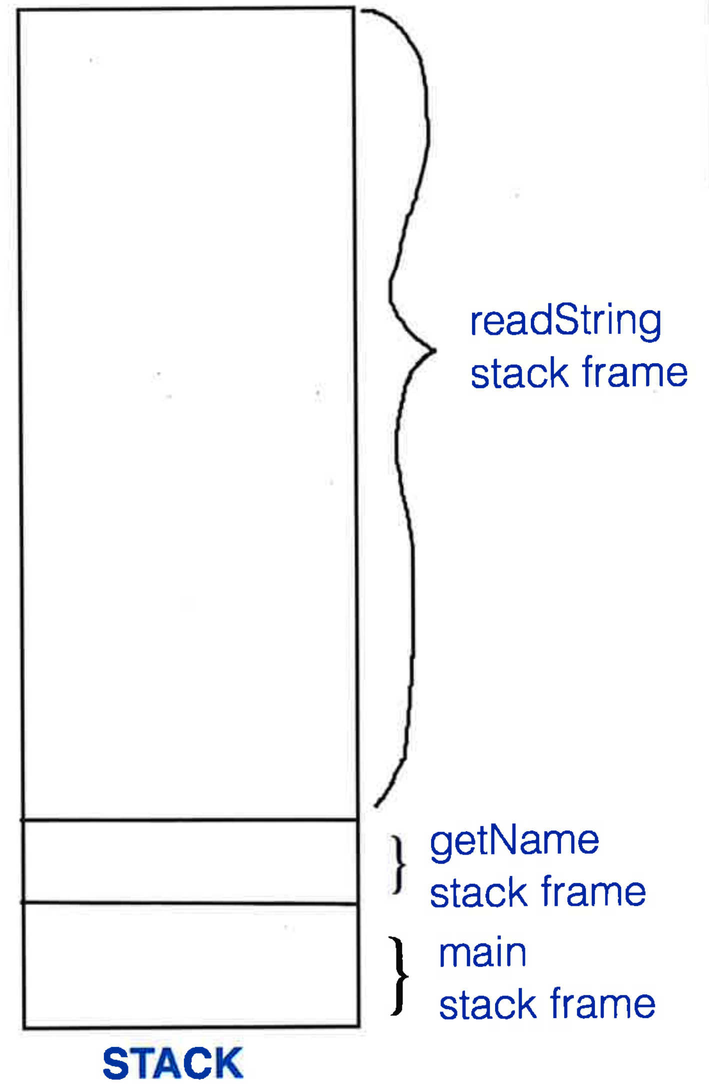
TEXT



DATA

BSS





STACK

Precept Activity Instructions:

Question Q1: What is the memory address and assembly code instruction that is executed after the `getName()` function returns to the `main()` function?

- `cp /u/cos217/PreceptDemos/grader .`
- `./grader`
- `gdb grader`
- `x/71i readString`
- Scroll back and forth to examine the displayed code to answer Question 1.
- `display/i $pc` (not useful until you run)
- Place a breakpoint at `main` and then `run`
- Use `nexti` to get to the `getName` call
- Use `stepi` to step into the `getName` function
- `print/x $x30` (Print what is in register x30?)
- `x/i $x30` (Print the instruction that x30 points to)
- **Question Q2**: What gdb instruction can you use to examine that instruction directly using its hex address

Precept Activity Answers:

Question Q1: What is the memory address and assembly code instruction that is executed after the `getName()` function returns to the `main()` function?

- `cp /u/cos217/PreceptDemos/grader . .`
- `./grader`
- `gdb grader`
- `x/71i readString`
- Scroll back and forth to examine the displayed code to answer Question 1.
- `display/i $pc`
- Place a breakpoint at main
- Use `nexti` to get to the getName call
- Use `stepi` to step into the getName function
- `print/x $x30` (What is in register 30?)
- `x/i $x30` (Dereference x30 to examine the instruction that register 30 points to)

- Question Q2: What gdb instruction can you use to examine that instruction directly using its hex address

Q1 Answer: 0x400844 <main+36>: mov x0, x19

Q2 Answer: x/i 0x400844