HW2.9. Common C Pitfalls

For each of the following code samples (all of which compile), determine if the code works as intended on a **32-bit system**, or if there is an error. If there is an error, choose an error type from the following list:

Probable Segfault: The code attempts to access memory not directly allocated by the program. For the purposes of this question, we also assume that accesses past the end of a stack- or heap-allocated array yield probable segfaults. (In practice, stack and heap allocations tend to be part of a larger block of allocated memory, so the system fails to catch the invalid memory access).

Incorrect free: The function free() is called on a pointer which cannot be freed.

Logic error: The code does not work as intended at least sometimes.

Memory Leak: The code runs to completion without any of the previous errors, but allocates memory which cannot be freed later.

Assume that all necessary libraries have been included, and that the system is 32-bit.

Q1.1:

```c
#define PASSWORD "correct horse battery staple"
char *check_permissions (char *user_guess) {
     if (user_guess = PASSWORD) {
          return "access granted";
     }
     return "access denied";
}
```

(a) Probable Segfault

(b) Incorrect Free

[✓ (c) Logic Error] ✓

(d) Memory Leak

(e) No Error

✓ 100%

Q1.2:

```c
#define PASSWORD "hunter2"
char *check_permissions (char *user_guess) {
     if (user_guess == PASSWORD) {
          return "access granted";
     }
     return "access denied";
}
```

(a) Probable Segfault

(b) Incorrect Free

[✓ (c) Logic Error] ✓

(d) Memory Leak

(e) No Error

✓ 100%

Q1.3:

```c
int main () {
     char *x = malloc(0xFFFFFFFF * sizeof(char));
     x[0] = '!';
}
```

[✓ (a) Probable Segfault] ✓

(b) Incorrect Free

(c) Logic Error

(d) Memory Leak

(e) No Error

✓ 100%

Q1.4:

```
int foo () {
     int *x = malloc(20);
     x[0] = x[1] = 1;
     x[2] = x[0] + x[1];
     x[3] = 99;
     return x[3];
}
```

(a) Probable Segfault

(b) Incorrect Free

(c) Logic Error

[✓ (d) Memory Leak] ✓

(e) No Error

✓ 100%

Q1.5:

```
int main () {
  char *x = "patrick";
  printf("%s", x);
  free(x); // tidy up
}
```

(a) Probable Segfault

[✓ (b) Incorrect Free] ✓

(c) Logic Error

(d) Memory Leak

(e) No Error

✓ 100%

Q1.6:

```
/*Copies up to maxlength characters from source into destination.
It is assumed that destination points to a buffer sufficiently long to hold
the copied string, and that source is a valid string.
You can assume that the maxlength input argument corresponds to the length of
the source string*/
void strncpy(char* destination, char* source, uint32_t maxlength) {
  int i = 0;
  while(i < maxlength && source[i]) {
    destination[i]=source[i];
    i++;
  }
  destination[strlen(destination)]='\0';
}
```

(a) Probable Segfault

(b) Incorrect Free

[✓ (c) Logic Error] ✓

(d) Memory Leak

(e) No Error

✓ 100%

Q1.7:

```
#DEFINE MAXSTRINGLENGTH 64 //Assume that the string input is at most 64
characters long
char* copystringtoheap(char* string) {
  char* c = malloc(sizeof(char)*(MAXSTRINGLENGTH+2));
  char* cclone = c;
  do {
   *(c++) = *string;
  } while (*(string++));
  free(c);
  return cclone;
}
```

(a) Probable Segfault

[✓ (b) Incorrect Free] ✓

(c) Logic Error

(d) Memory Leak

(e) No Error

✓ 100%

Try a new variant

Correct answer

Q1.1:

```
#define PASSWORD "correct horse battery staple"
char *check_permissions (char *user_guess) {
     if (user_guess = PASSWORD) {
          return "access granted";
     }
     return "access denied";
}
```

(c) Logic Error

Q1.2:

```
#define PASSWORD "hunter2"
char *check_permissions (char *user_guess) {
     if (user_guess == PASSWORD) {
          return "access granted";
     }
     return "access denied";
}
```

(c) Logic Error

Q1.3:

```
int main () {
     char *x = malloc(0xFFFFFFFF * sizeof(char));
     x[0] = '!';
}
```

(a) Probable Segfault

Q1.4:

```
int foo () {
     int *x = malloc(20);
     x[0] = x[1] = 1;
     x[2] = x[0] + x[1];
     x[3] = 99;
     return x[3];
}
```

(d) Memory Leak

Q1.5:

```
int main () {
  char *x = "patrick";
  printf("%s", x);
  free(x); // tidy up
}
```

(b) Incorrect Free

Q1.6:

```
/*Copies up to maxlength characters from source into destination.
It is assumed that destination points to a buffer sufficiently long to hold
the copied string, and that source is a valid string.
You can assume that the maxlength input argument corresponds to the length of
the source string*/
void strncpy(char* destination, char* source, uint32_t maxlength) {
  int i = 0;
  while(i < maxlength && source[i]) {
    destination[i]=source[i];
    i++;
  }
  destination[strlen(destination)]='\0';
}
```

(c) Logic Error

Q1.7:

```
#DEFINE MAXSTRINGLENGTH 64 //Assume that the string input is at most 64
characters long
char* copystringtoheap(char* string) {
  char* c = malloc(sizeof(char)*(MAXSTRINGLENGTH+2));
  char* cclone = c;
  do {
   *(c++) = *string;
  } while (*(string++));
  free(c);
  return cclone;
}
```

(b) Incorrect Free

Q1.1: We don't want to assign to user_guess (using a single =) inside the if conditional, we want to do some sort of comparison.

Q1.2: The if conditional is doing a pointer comparison, where we actually want a string comparison, probably using `strcmp`.

Q1.3: `malloc` returns NULL if it cannot return the requested amount of space, which is guaranteed since we are requesting 2^32-1 bytes on a 32-bit system; our code and stack exceed 1 byte long, so the heap cannot be large enough to contain a block of size 2^32-1. This code should add an explicit check: if (x != NULL) before referencing x. Dereferencing a null pointer results in a segfault.

Q1.4: This code doesn't explicitly call `free` on the space allocated with `malloc`.

Q1.5: `free` should only be used with heap-allocated space, i.e. space allocated with malloc, calloc, etc. x is statically allocated.

Q1.6: `strlen` uses the null terminator to determine the length of the string in the first place, so it's not useful to use `strlen` to try and place the null terminator. As such, the null terminator is never placed. This type of bug can be challenging to spot, since most memory is initially set to 0s before running. The best way to catch this is to use Valgrind; there is an option that lets you force `malloc` to return buffers filled with nonzero data, which would appear if you don't properly terminate your string.

Q1.7: `free` can only be called on pointers returned by `malloc`, not pointers within the buffer. There are several reasons for this, but we can imagine that the OS keeps track of blocks of malloced space, and only stores the starting pointer of each block. When `free` is called, the OS looks through the pointers it remembers, and frees the one that matches. Another way to see this is that `malloc` actually creates some metadata before and after the block you allocate, so trying to free pointer in the middle of a malloced block ends up interpreting random data as if it was heap metadata.

Submitted answer 3 **correct: 100%**

Submitted at 2022-09-03 09:48:38 (PDT)

hide

Q1.1:

```
#define PASSWORD "correct horse battery staple"
char *check_permissions (char *user_guess) {
     if (user_guess = PASSWORD) {
          return "access granted";
     }
     return "access denied";
}
```

(c) Logic Error ✓ 100%

Q1.2:

```
#define PASSWORD "hunter2"
char *check_permissions (char *user_guess) {
     if (user_guess == PASSWORD) {
          return "access granted";
     }
     return "access denied";
}
```

(c) Logic Error ✓ 100%

Q1.3:

```
int main () {
     char *x = malloc(0xFFFFFFFF * sizeof(char));
     x[0] = '!';
}
```

(a) Probable Segfault ✓ 100%

Q1.4:

```
int foo () {
     int *x = malloc(20);
     x[0] = x[1] = 1;
     x[2] = x[0] + x[1];
     x[3] = 99;
     return x[3];
}
```

(d) Memory Leak ✓ 100%

Q1.5:

```
int main () {
  char *x = "patrick";
  printf("%s", x);
  free(x); // tidy up
}
```

(b) Incorrect Free ✓ 100%

Q1.6:

```
/*Copies up to maxlength characters from source into destination.
It is assumed that destination points to a buffer sufficiently long to hold
the copied string, and that source is a valid string.
You can assume that the maxlength input argument corresponds to the length of
the source string*/
void strncpy(char* destination, char* source, uint32_t maxlength) {
  int i = 0;
  while(i < maxlength && source[i]) {
    destination[i]=source[i];
    i++;
  }
  destination[strlen(destination)]='\0';
}
```

(c) Logic Error ✓ 100%

Q1.7:

```
#DEFINE MAXSTRINGLENGTH 64 //Assume that the string input is at most 64
characters long
char* copystringtoheap(char* string) {
  char* c = malloc(sizeof(char)*(MAXSTRINGLENGTH+2));
  char* cclone = c;
  do {
```

Submitted answer 2 **partially correct: 57%**

Submitted at 2022-09-03 09:46:57 (PDT)

show

```
    [illegible]
}
```

Submitted answer 1 **partially correct: 28%**

Submitted at 2022-09-03 09:44:39 (PDT)

show