Review quiz: answers

NWEN 241 · Weeks 9 to 12 · 40 questions

Questions 1 and 2 · Dynamic memory

1. Which line allocates space for 10 ints?

A. `int *p = malloc(10);`

B. `int *p = malloc(10 * sizeof(int));`

C. `int *p = malloc(sizeof(int *));`

D. `int p[10] = malloc(10 * sizeof(int));`

2. What is wrong with this function?

```
int *make(void)
{
    int x = 42;
    return &x;
}
```

A. Nothing is wrong.

B. It returns the address of a local variable, which no longer exists after the function returns.

C. It leaks memory.

D. It should return *x.

Answers 1 and 2 · Dynamic memory

1. Which line allocates space for 10 ints?

A. `int *p = malloc(10);`

B. `int *p = malloc(10 * sizeof(int));` ✓

C. `int *p = malloc(sizeof(int *));`

D. `int p[10] = malloc(10 * sizeof(int));`

Why: `malloc` takes a number of bytes, so 10 ints need `10 * sizeof(int)`.

2. What is wrong with this function?

```
int *make(void)
{
    int x = 42;
    return &x;
}
```

A. Nothing is wrong.

B. It returns the address of a local variable, which no longer exists after the function returns. ✓

C. It leaks memory.

D. It should return *x.

Why: `x` is a local variable. It's gone when `make` returns, so the pointer dangles.

Questions 3 and 4 · Dynamic memory

3. How many bytes does this code leak?

```
char *p = malloc(100);
p = malloc(50);
free(p);
```

A. 0
B. 50
C. 100
D. 150

4. What is wrong with this code?

```
int *a = malloc(5 * sizeof(int));
for (int i = 0; i <= 5; i++)
    a[i] = 0;
```

A. Nothing is wrong.
B. It writes one int past the end of the block.
C. It leaks the block.
D. `malloc` doesn't work with `int`.

Answers 3 and 4 · Dynamic memory

3. How many bytes does this code leak?

```
char *p = malloc(100);
p = malloc(50);
free(p);
```

A. 0
B. 50
C. 100 ✓
D. 150

Why: The only pointer to the 100-byte block is overwritten. Only the 50-byte block is freed.

4. What is wrong with this code?

```
int *a = malloc(5 * sizeof(int));
for (int i = 0; i <= 5; i++)
    a[i] = 0;
```

A. Nothing is wrong.
B. It writes one int past the end of the block. ✓
C. It leaks the block.
D. `malloc` doesn't work with `int`.

Why: `i <= 5` writes `a[5]`, one int past the 5 that were allocated.

Questions 5 and 6 · Dynamic memory

5. What happens at the `printf`?

```
int *p = malloc(sizeof(int));
*p = 7;
free(p);
printf("%d\n", *p);
```

A. It always prints 7.
B. It always prints 0.
C. Undefined behaviour: `p` points to memory that has been freed.
D. It doesn't compile.

6. `p` was returned by `malloc`. Which of these is it correct to pass to `free`?

A. `p`
B. `p + 1`
C. The address of a local array
D. A string literal, such as `"hello"`

Answers 5 and 6 · Dynamic memory

5. What happens at the `printf`?

```
int *p = malloc(sizeof(int));
*p = 7;
free(p);
printf("%d\n", *p);
```

A. It always prints 7.
B. It always prints 0.
C. Undefined behaviour: `p` points to memory that has been freed. ✓
D. It doesn't compile.

Why: Use after free. It might print 7 or anything else: undefined behaviour.

6. `p` was returned by `malloc`. Which of these is it correct to pass to `free`?

A. `p` ✓
B. `p + 1`
C. The address of a local array
D. A string literal, such as `"hello"`

Why: `free` takes exactly the pointer `malloc` returned, or NULL.

Questions 7 and 8 · Dynamic memory

7. What is wrong with this code?

```
char *a = malloc(10);
char *b = a;
free(a);
free(b);
```

A. Nothing: `a` and `b` are two blocks.

B. The same block is freed twice.

C. It leaks 10 bytes.

D. `free(a)` sets `b` to NULL, so `free(b)` does nothing.

8. `t` points to the string `"hello"`. What is wrong with this copy?

```
char *s = malloc(strlen(t));
strcpy(s, t);
```

A. Nothing is wrong.

B. There is no room for the `'\0'`: it needs `strlen(t) + 1` bytes.

C. `strcpy` copies the other way round.

D. `strlen` counts the `'\0'`, so the block is one byte too big.

Answers 7 and 8 · Dynamic memory

7. What is wrong with this code?

```
char *a = malloc(10);
char *b = a;
free(a);
free(b);
```

A. Nothing: a and b are two blocks.

B. The same block is freed twice. ✓

C. It leaks 10 bytes.

D. `free(a)` sets b to NULL, so `free(b)` does nothing.

Why: a and b point to the same block, so it is freed twice.

8. `t` points to the string `"hello"`. What is wrong with this copy?

```
char *s = malloc(strlen(t));
strcpy(s, t);
```

A. Nothing is wrong.

B. There is no room for the `'\0'`: it needs `strlen(t) + 1` bytes. ✓

C. `strcpy` copies the other way round.

D. `strlen` counts the `'\0'`, so the block is one byte too big.

Why: `strlen` doesn't count the `'\0'`, so `strcpy` writes 6 bytes into 5.

Questions 9 and 10 · Dynamic memory

9. What does this print?

```
int *p = malloc(10 * sizeof(int));
printf("%zu\n", sizeof(p));
```

A. 4 B. 8 C. 10 D. 40

10. Why is this line risky?

```
p = realloc(p, 200);
```

A. `realloc` always frees the old block.

B. `realloc` can only make a block smaller.

C. If `realloc` fails, it returns NULL, and the only pointer to the old block is lost.

D. It doesn't compile.

Answers 9 and 10 · Dynamic memory

9. What does this print?

```
int *p = malloc(10 * sizeof(int));
printf("%zu\n", sizeof(p));
```

A. 4 **B. 8 ✓** C. 10 D. 40

Why: A pointer is 8 bytes on a 64-bit machine, whatever it points to.

10. Why is this line risky?

```
p = realloc(p, 200);
```

A. `realloc` always frees the old block.

B. `realloc` can only make a block smaller.

C. If `realloc` fails, it returns NULL, and the only pointer to the old block is lost. ✓

D. It doesn't compile.

Why: On failure `p` becomes NULL and the old block leaks. Use a temporary pointer.

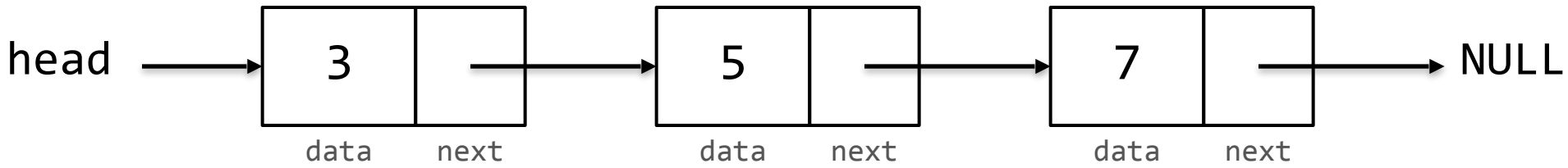
Linked lists: the setup for 11 to 18

Questions 11 to 18 use this node type:

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

Where a question shows a list, head points to the first node:

head → 3 → 5 → 7 → NULL



Questions 11 and 12 · Linked lists

11. Compared with an array, which is true of a linked list?

A. Its elements sit next to each other in memory.
B. You can reach element i in one step.
C. Its size must be fixed when you compile.
D. It can grow one node at a time, and its nodes can be anywhere in memory.

12. The list is head → 3 → 5 → 7 → NULL. What does this print?

```
struct node *p = head;
while (p->next != NULL) {
    printf("%d ", p->data);
    p = p->next;
}
```

A. 3 5
B. 3 5 7
C. 5 7
D. It crashes.

Answers 11 and 12 · Linked lists

11. Compared with an array, which is true of a linked list?

A. Its elements sit next to each other in memory.
B. You can reach element i in one step.
C. Its size must be fixed when you compile.
D. It can grow one node at a time, and its nodes can be anywhere in memory. ✓

Why: Nodes are allocated one at a time, anywhere on the heap.

12. The list is head → 3 → 5 → 7 → NULL. What does this print?

```
struct node *p = head;
while (p->next != NULL) {
    printf("%d ", p->data);
    p = p->next;
}
```

A. 3 5 ✓
B. 3 5 7
C. 5 7
D. It crashes.

Why: The loop stops when p reaches the last node, so 7 is never printed.

Questions 13 and 14 · Linked lists

13. This code adds a new node `n` to the front of the list. Which two lines go at the comment, in which order?

```
struct node *n =
    malloc(sizeof(struct node));
n->data = 1;
/* two lines here */
```

A. `head = n;` then `n->next = head;`

B. `head->next = n;` then `n->next = NULL;`

C. Either order of A's two lines works.

D. `n->next = head;` then `head = n;`

14. Someone writes the lines from 13 as `head = n;` then `n->next = head;`. What happens?

A. `n->next` points to `n` itself, and the rest of the list is lost.

B. It works the same as the other order.

C. It doesn't compile.

D. The new node goes at the end of the list.

Answers 13 and 14 · Linked lists

13. This code adds a new node `n` to the front of the list. Which two lines go at the comment, in which order?

```
struct node *n =
    malloc(sizeof(struct node));
n->data = 1;
/* two lines here */
```

A. `head = n;` then `n->next = head;`

B. `head->next = n;` then `n->next = NULL;`

C. Either order of A's two lines works.

D. `n->next = head;` then `head = n;` ✓

Why: Link the new node to the old head first, then move `head`.

14. Someone writes the lines from 13 as `head = n;` then `n->next = head;`. What happens?

A. `n->next` points to `n` itself, and the rest of the list is lost. ✓

B. It works the same as the other order.

C. It doesn't compile.

D. The new node goes at the end of the list.

Why: After `head = n`, `n->next = head` makes `n` point to itself. The old nodes can't be reached.

Questions 15 and 16 · Linked lists

15. This code deletes the first node. What happens if the list is empty (`head` is NULL)?

```
struct node *tmp = head;
head = head->next;
free(tmp);
```

A. Nothing happens.
B. `head` stays NULL, and `free` does nothing.
C. It doesn't compile.
D. `head->next` follows a NULL pointer: undefined behaviour, usually a crash.

16. This code removes the node after `prev`. What is wrong with it?

```
struct node *victim = prev->next;
free(victim);
prev->next = victim->next;
```

A. It reads `victim->next` after `victim` has been freed.
B. Nothing is wrong.
C. It should free `prev` instead.
D. It leaks `victim`.

Answers 15 and 16 · Linked lists

15. This code deletes the first node. What happens if the list is empty (`head` is NULL)?

```
struct node *tmp = head;
head = head->next;
free(tmp);
```

A. Nothing happens.
B. `head` stays NULL, and `free` does nothing.
C. It doesn't compile.
D. `head->next` follows a NULL pointer: undefined behaviour, usually a crash. ✓

Why: `head->next` follows NULL. Check `head != NULL` first.

16. This code removes the node after `prev`. What is wrong with it?

```
struct node *victim = prev->next;
free(victim);
prev->next = victim->next;
```

A. It reads `victim->next` after `victim` has been freed. ✓
B. Nothing is wrong.
C. It should free `prev` instead.
D. It leaks `victim`.

Why: Relink first (`prev->next = victim->next`), then free.

Questions 17 and 18 · Linked lists

17. The list is head → 3 → 5 → 7 → NULL. After `push(head, 9);` in `main`, what is `head->data` in `main`?

```
void push(struct node *head, int x)
{
    struct node *n =
        malloc(sizeof(struct node));
    n->data = x;
    n->next = head;
    head = n;
}
```

A. 9 B. 0 C. It crashes. D. 3

18. A program only ever inserts at the head of a list and deletes from the head. Which structure is that?

A. A queue
B. A sorted list
C. A stack
D. An array

Answers 17 and 18 · Linked lists

17. The list is head → 3 → 5 → 7 → NULL. After `push(head, 9);` in `main`, what is `head->data` in `main`?

```
void push(struct node *head, int x)
{
    struct node *n =
        malloc(sizeof(struct node));
    n->data = x;
    n->next = head;
    head = n;
}
```

A. 9 B. 0 C. It crashes. **D. 3 ✓**

Why: `push` changes its own copy of `head`. `main`'s `head` still points to 3.

18. A program only ever inserts at the head of a list and deletes from the head. Which structure is that?

A. A queue
B. A sorted list
C. A stack ✓
D. An array

Why: Last in, first out.

Questions 19 and 20 · Files and streams

19. You want to add lines to the end of `log.txt` without erasing what is already there. Which mode do you pass to `fopen`?

A. `"r"`

B. `"w"`

C. `"rb"`

D. `"a"`

20. `data.txt` holds 100 bytes. What does `fopen("data.txt", "w")` do to it?

A. Adds new writes after the 100 bytes.

B. Fails and returns NULL, because the file exists.

C. Empties it: the file now holds 0 bytes.

D. Opens it for reading only.

Answers 19 and 20 · Files and streams

19. You want to add lines to the end of `log.txt` without erasing what is already there. Which mode do you pass to `fopen`?

A. `"r"`

B. `"w"`

C. `"rb"`

D. `"a"` ✓

Why: `"a"` appends. `"w"` would empty the file.

20. `data.txt` holds 100 bytes. What does `fopen("data.txt", "w")` do to it?

A. Adds new writes after the 100 bytes.

B. Fails and returns NULL, because the file exists.

C. Empties it: the file now holds 0 bytes. ✓

D. Opens it for reading only.

Why: `"w"` truncates an existing file to 0 bytes.

Questions 21 and 22 · Files and streams

21. `missing.txt` does not exist. What happens?

```
FILE *fp = fopen("missing.txt", "r");
int c = fgetc(fp);
```

A. `fopen` returns NULL, and passing NULL to `fgetc` is undefined behaviour, usually a crash.
B. `fgetc` returns EOF.
C. `fopen` creates an empty `missing.txt`.
D. `fgetc` returns 0.

22. Why is `c` an `int` and not a `char`?

```
int c;
while ((c = fgetc(fp)) != EOF)
    putchar(c);
```

A. An `int` is faster to read.
B. `putchar` only accepts an `int`.
C. EOF is not a character: an `int` can hold every character value and EOF as well.
D. A `char` can't be compared with `!=`.

Answers 21 and 22 · Files and streams

21. `missing.txt` does not exist. What happens?

```
FILE *fp = fopen("missing.txt", "r");
int c = fgetc(fp);
```

A. `fopen` returns NULL, and passing NULL to `fgetc` is undefined behaviour, usually a crash. ✓

B. `fgetc` returns EOF.

C. `fopen` creates an empty `missing.txt`.

D. `fgetc` returns 0.

Why: `fopen` returns NULL when the file can't be opened. Always check before using `fp`.

22. Why is `c` an `int` and not a `char`?

```
int c;
while ((c = fgetc(fp)) != EOF)
    putchar(c);
```

A. An `int` is faster to read.

B. `putchar` only accepts an `int`.

C. EOF is not a character: an `int` can hold every character value and EOF as well. ✓

D. A `char` can't be compared with `!=`.

Why: `fgetc` returns every possible byte value plus EOF, so it needs more values than a `char` holds.

Questions 23 and 24 · Files and streams

23. The user types `5 x` and presses return. What does this `scanf` return?

```
int a, b;
int r = scanf("%d %d", &a, &b);
```

A. 0 B. 1 C. 2 D. EOF

24. This program crashes on its third line. Why might `hello` never appear on the screen?

```
printf("hello");
int *p = NULL;
*p = 1;
```

A. `hello` was still waiting in the output buffer when the crash killed the process.

B. `printf` only prints after `main` returns.

C. Without `\n`, the line doesn't compile.

D. The terminal ignores text that has no newline.

Answers 23 and 24 · Files and streams

23. The user types `5 x` and presses return. What does this `scanf` return?

```
int a, b;
int r = scanf("%d %d", &a, &b);
```

A. 0 **B. 1 ✓** C. 2 D. EOF

Why: `5` fills `a`, then `x` doesn't match `%d`, so 1 item is assigned.

24. This program crashes on its third line. Why might `hello` never appear on the screen?

```
printf("hello");
int *p = NULL;
*p = 1;
```

A. `hello` was still waiting in the output buffer when the crash killed the process. ✓

B. `printf` only prints after `main` returns.

C. Without `\n`, the line doesn't compile.

D. The terminal ignores text that has no newline.

Why: Output to a terminal is line-buffered. With no `\n`, `hello` is still in the buffer when the crash happens.

Question 25 · Files and streams

25. `nums.txt` holds three lines: `1`, `2` and `3`. What does this print?

```
int n;
while (!feof(fp)) {
    fscanf(fp, "%d", &n);
    printf("%d\n", n);
}
```

A. 1 2 3
B. 1 2
C. 1 2 3 3
D. Nothing

Answer 25 · Files and streams

25. `nums.txt` holds three lines: `1`, `2` and `3`. What does this print?

```
int n;
while (!feof(fp)) {
    fscanf(fp, "%d", &n);
    printf("%d\n", n);
}
```

A. 1 2 3

B. 1 2

C. 1 2 3 3 ✓

D. Nothing

Why: `feof` only turns true after a read fails. The 4th `fscanf` fails, `n` is still 3, and it's printed again.

Questions 26 and 27 · Files and streams

26. `int x = 7000000;`. How many bytes does `fprintf(fp, "%d", x)` write, and how many does `fwrite(&x, sizeof x, 1, fp)` write?

A. 4 and 4
B. 7 and 7
C. 4 and 7
D. 7 and 4

27. A file is 27 bytes long. After `fseek(fp, 0, SEEK_END)`, what does `ftell(fp)` return?

A. 26 B. 0 C. -1 D. 27

Answers 26 and 27 · Files and streams

26. `int x = 7000000;`. How many bytes does `fprintf(fp, "%d", x)` write, and how many does `fwrite(&x, sizeof x, 1, fp)` write?

A. 4 and 4
B. 7 and 7
C. 4 and 7
D. 7 and 4 ✓

Why: `fprintf` writes the text `7000000` (7 bytes). `fwrite` writes the int's 4 bytes.

27. A file is 27 bytes long. After `fseek(fp, 0, SEEK_END)`, what does `ftell(fp)` return?

A. 26 B. 0 C. -1 **D. 27 ✓**

Why: The end of the file is position 27: the file's size.

Questions 28 and 29 · Arguments and arrays

28. You run `./prog one two three`. What is `argc`?

A. 4 B. 3 C. 2 D. 5

29. You run `./prog "hello world" 5`. What is `argv[2]`?

A. `"world"`

B. The number 5, as an `int`

C. NULL

D. The string `"5"`

Answers 28 and 29 · Arguments and arrays

28. You run `./prog one two three`. What is `argc`?

A. 4 ✓ B. 3 C. 2 D. 5

Why: The program's name counts, as `argv[0]`.

29. You run `./prog "hello world" 5`. What is `argv[2]`?

A. `"world"`

B. The number 5, as an `int`

C. NULL

D. The string `"5"` ✓

Why: Every argument arrives as a string. `atoi` turns it into a number.

Questions 30 and 31 · Arguments and arrays

30. You run `./prog abc`. What is `argv[1][2]`?

A. `'a'`

B. `'b'`

C. `'c'`

D. `'\0'`

31. You run `./prog` with no arguments. What is `argv[1]`?

A. An empty string, `""`

B. NULL

C. `"./prog"`

D. Whatever happens to be in memory

Answers 30 and 31 · Arguments and arrays

30. You run `./prog abc`. What is `argv[1][2]`?

A. `'a'`

B. `'b'`

C. `'c'` ✓

D. `'\0'`

Why: `argv[1]` is `"abc"`, and index 2 is `'c'`.

31. You run `./prog` with no arguments. What is `argv[1]`?

A. An empty string, `""`

B. NULL ✓

C. `"./prog"`

D. Whatever happens to be in memory

Why: `argc` is 1, and `argv[argc]` is always NULL.

Question 32 · Arguments and arrays

32. `main` declares `int arr[10];` and calls `f(arr);`. What does `f` print?

```
void f(int a[])
{
    printf("%zu\n", sizeof a);
}
```

A. 8 B. 10 C. 40 D. 4

Answer 32 · Arguments and arrays

32. `main` declares `int arr[10];` and calls `f(arr);`. What does `f` print?

```
void f(int a[])
{
    printf("%zu\n", sizeof a);
}
```

A. 8 ✓ B. 10 C. 40 D. 4

Why: An array parameter is really a pointer, so `sizeof` gives 8. The compiler even warns about it.

Questions 33 and 34 · Processes and pipes

33. Which of these is a system call?

A. `printf`

B. `strlen`

C. `write`

D. `fopen`

34. How many lines does this print?

```
printf("hi\n");
fork();
printf("bye\n");
```

A. 2 B. 4 C. 1 D. 3

Answers 33 and 34 · Processes and pipes

33. Which of these is a system call?

A. `printf`

B. `strlen`

C. `write` ✓

D. `fopen`

Why: `write` is the system call. `printf`, `strlen` and `fopen` are library functions.

34. How many lines does this print?

```
printf("hi\n");
fork();
printf("bye\n");
```

A. 2 B. 4 C. 1 **D. 3 ✓**

Why: `hi` once, before the fork. `bye` twice, once from each process.

Questions 35 and 36 · Processes and pipes

35. How many times is x printed?

```
fork();
fork();
printf("x\n");
```

A. 4 B. 2 C. 3 D. 8

36. What does the parent print?

```
int x = 1;
if (fork() == 0) {
    x = x + 1;
    exit(0);
}
wait(NULL);
printf("%d\n", x);
```

A. 2
B. 1
C. 3
D. It depends on which process runs first.

Answers 35 and 36 · Processes and pipes

35. How many times is x printed?

```
fork();
fork();
printf("x\n");
```

A. 4 ✓ B. 2 C. 3 D. 8

Why: 1 process becomes 2, then 4. Each prints once.

36. What does the parent print?

```
int x = 1;
if (fork() == 0) {
    x = x + 1;
    exit(0);
}
wait(NULL);
printf("%d\n", x);
```

A. 2

B. 1 ✓

C. 3

D. It depends on which process runs first.

Why: The child changes its own copy of x. The parent's x is still 1.

Questions 37 and 38 · Processes and pipes

37. Which lines appear?

```
printf("start\n");
execlp("ls", "ls", NULL);
printf("end\n");
```

A. `start`, then the `ls` listing, then `end`

B. The `ls` listing, then `start`

C. `start`, then the `ls` listing. `end` is never printed.

D. Only `end`

38. What does `fork()` return in the child process?

A. The child's process ID

B. -1

C. The parent's process ID

D. 0

Answers 37 and 38 · Processes and pipes

37. Which lines appear?

```
printf("start\n");
execlp("ls", "ls", NULL);
printf("end\n");
```

A. `start`, then the `ls` listing, then `end`

B. The `ls` listing, then `start`

C. `start`, then the `ls` listing. `end` is never printed. ✓

D. Only `end`

Why: `exec` replaces the program. `end` only prints if `exec` fails.

38. What does `fork()` return in the child process?

A. The child's process ID

B. -1

C. The parent's process ID

D. 0 ✓

Why: 0 in the child, the child's PID in the parent, -1 if `fork` fails.

Questions 39 and 40 · Processes and pipes

39. After `int fd[2]; pipe(fd);`, which end do you read from?

A. `fd[0]`
B. `fd[1]`
C. Either end
D. `fd[2]`

40. A process is waiting for the user to type a line. Which state is it in?

A. Running
B. Ready
C. Terminated
D. Waiting

Answers 39 and 40 · Processes and pipes

39. After `int fd[2]; pipe(fd);`, which end do you read from?

A. `fd[0]` ✓

B. `fd[1]`

C. Either end

D. `fd[2]`

Why: `fd[0]` is the read end, `fd[1]` the write end.

40. A process is waiting for the user to type a line. Which state is it in?

A. Running

B. Ready

C. Terminated

D. Waiting ✓

Why: It is waiting (blocked) until the input arrives.

All 40 answers

1	2	3	4	5	6	7	8	9	10
B	**B**	**C**	**B**	**C**	**A**	**B**	**B**	**B**	**C**
11	12	13	14	15	16	17	18	19	20
D	**A**	**D**	**A**	**D**	**A**	**D**	**C**	**D**	**C**
21	22	23	24	25	26	27	28	29	30
A	**C**	**B**	**A**	**C**	**D**	**D**	**A**	**D**	**C**
31	32	33	34	35	36	37	38	39	40
B	**A**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**D**