1 1997 Paper 5 Question 5

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y1997p5q5.pdf

A slightly clumsy programmer had been lagging behind the company productivity targets and needed to write some C++ code in a hurry. Almost remembering an old optimising compilers question from student days, this programmer produced a file containing the test:

```
1  struct List {int head; struct list *tail};
2
3  struct List readlist(){
4          int i;
5          struct List *p, *q, *t;
6  L1:     p = NULL;
7  L2:     while (scanf("%d", i) = 1)
8          /* scanf reads an integer and returns 1
9                  if it finds one correctly /*
10         [
11 L3:             t = malloc(sizeof(List *));
12                 if (t == 0) printf("oops no memery\n");
13                 else t->hd = i;
14 L4:                       t->tl = 0;
15                 if (p == NULL)
16                         p = q = t;
17                 else
18                         q->tl = t, q = t;
19         }
20 L5:     return p;
21 }
```

Unfortunately the programmer had forgotten what this was supposed to achieve; you are asked to help re-create an explanation for the code and to identify problems (of either style or correctness) in it. You do not need to provide a correct version of the program: just draw attention to as many errors or oddities as you can.

The errors and oddities in this code are as follows:

- Firstly, the code is "written in C++" however uses none of the features of C++, employing only C. This defeats the purpose of writing in C++ and is completely against convention.
- On line 1:

  When declaring struct List, second field has type "struct list *" rather than "struct List *". This is not a valid type and so would cause a compile error.

  The fields of struct List are "head" and "tail". However, they are consistently referred to as "hd" and "tl" in the code. So either the fields should be called "hd" and "tl" or the rest of the code should be refactored to refer to the fields as "head" and "tail". These name errors are a compile time error. Additionally, both fields are declared on one line – which makes it harder to read.

  When declaring structs it is good practice to typedef to give the struct a better name and simplify later code. This is not done in this program and is an oddity.
- On line 3:

  The function "readlist()" has a return type of "struct List" – but the return type is actually of type "struct List *". This would cause a compile error.

  "readlist()" also has the wrong type signature. "()" means readlist may take any number of parameters of unknown types. It should instead be called "readlist(void)" to take no parameters. As declared, users could pass any number of arguments to

readlist and there would be no compile time error. Therefore declarations with "()" are strongly discouraged. If this code is compiled in C++ then this is not an issue – but there is some ambiguity about which language this code is actually compiled in and so this *could* be a problem.

- On lines 4 & 5:

  The variable names have no meaning – this is poor programming practice and makes the program harder to understand.

  It is better practice to only declare variables for the scope in which they are used. "t" is only used inside the while loop – it would therefore be better practice to declare it inside the while loop – it is an oddity to declare it outside. However, historically it was considered good practice to declare variables at the start of a function and was defined in the C standard at one point. I don't know whether this has *become* an oddity or whether it was initially intended as one.

- On line 6:

  The program uses labels throughout despite the fact that it never uses goto to jump to those labels. This just clutters up the program and makes it more difficult to read. I'm not suggesting the use of goto in this program – only stating that the only reason to use labels in C is if you are going to use goto. Labels are used on lines 6, 7, 11, 14 & 20. I will not repeat this oddity.

- On line 7:

  "scanf" is provided by the "stdio.h" header – which has not been included in the program. This will cause a compile error.

  "scanf" requires a pointer to the variable we wish to update – not just it's value – which is what is being passed. This means the call should be "scanf("%d", &i)". Passing an integer to scanf will cause a compile error.

  The programmer is using the assignment operator "=" rather than equality "==". So this attempts to assign the value 1 to a function call. This is a compile error.

  "scanf" will return 0 if it fails. It returns 1 if it successfully assigns an integer to i. It is more elegant therefore for the condition to be "while (scanf("%d", &i))" without an equality test. Having an equality test is an oddity.

  There is no message informing the user what they should be inputting. Whether or not this is necessary depends on the context. However, it felt worth mentioning.

  There is no obvious way to exit this while loop without error. Inputting "\" will gracefully escape the loop, however this is not conveyed to the user either when executing or in the source code. This is a usability failure.

- On line 8:

  This is likely personal, but it feels incorrect and certainly odd to have a comment between a "while" and the while block.

- On line 9:

  The multiline comment ends with "/*" – it should end with "*/" This means the comment is not ended – the rest of the program will be commented out. This will cause a compile error.

  The comment is ambiguous – does scanf return "1" if it finds an integer correctly or if the integer is "1"?

- On line 10:

  The block is opened with "[" rather than "{". This is a compile error.

- On line 11:

  "malloc" is implemented in the "stdlib.h" header – which has not been included in the program. This can be resolved with "#include <stdlib.h>". The lack of inclusion would cause a compile error.

  The call allocates a pointer to a "List *" which will be allocated on the heap. "List *" is not a valid type – this is a compile error – the programmer means to allocate a struct "List *". Secondly, allocating a pointer to a "struct List *" is a logic error. We should instead allocate a "struct List".

- On line 12:

  "t == 0" compares a pointer to an integer. This will raise a warning. It would be better to compare to "NULL" – however checking for "t == NULL" is odd. It would be more elegant for the comparison to be "if (t)" and to swap the if and else clauses.

  The error message given is uninformative and almost joking. This is an unhelpful message. Additionally there is a typo in the message.

  After the out of memory exception is noticed, it is never addressed. This would be a suitable time for the function to exit or an error handling routine to start.

  Additionally, whether printf allocates memory is unspecified. So in some implementations, printf may need to allocate heap memory – however, we've just run out of heap space. Meaning that the call to printf may fail. There is no obvious way to resolve an out of memory exception and the best approach is case specific. For example Java allocates the OutOfMemoryException object on startup and raises it on failure. I think the best approach in this situation would be to return or free the last few nodes of the list.

- On line 13:

  The programmer has not put a braces around the body of the else clause. This means that the else clause only contains only "t->hd = i". The "t->tl = 0" is outside the clause. Since t could be NULL, this may lead to dereferencing a null pointer.

- On line 14:

  The tl attribute of struct List is of type "struct List *". However we assign an integer to it. This is a warning and instead we should assign NULL .

- On line 15:

  if "p == NULL" is unnecessarily verbose. It would be better practice to swap the if and else clauses and have the condition as "if (p)".

- On line 18:

  An oddity is that the else clause has two different expressions separated by a ",". However, the return type is not used. It is therefore better practice to separate them by ";" – it's clearer and does not change the meaning.

This code is meant to read in a stream of integers and build a linked list containing those integers.

A correct implementation in C is below:

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

typedef struct List* list;

struct List {
```

```
        int value;
        list next;
};

list readlist(void){
        int i;
        list head, current, tail = NULL;
        printf("Input␣the␣elements␣of␣the␣integer␣linked␣list\n")
        while (scanf("%d", &i)){
                t = (list) malloc(sizeof(struct List));
                if (t){
                        t->value = i;
                        t->next = NULL;
                }
                else{
                        /*
                        error routine for out of memory exception
                        I would need more information to know what
                        would be most appropriate.
                        possible routines would include:
                        return current list,
                        write current list to a file,
                        deallocate head of the list to make space,
                        deallocate the whole list
                        ...
                        */
                }
                if (head){
                        current->value = tail;
                        current = tail;
                }
                else{
                        head = current = tail;
                }
        }
}
```

Suggest two ways in which a move from C to C++ might allow the structure of the code to be improved.

C++ has classes. This would allow all methods related to the Linked List to be encapsulated in a single class. This would remove the need for a separate "struct" and allow us to neatly organise and contain all code relating to the Linked List in a sigle file. This would also allow a level of abstraction – the end-user would not need to know how the linked list works to use it. Implementing the Linked List in this way would also allow us to manage it better, preventing it becoming cyclic or leaking.

C++ also provides exceptions, this would allow better management of the OutOfMemoryException that could be caused by repeatedly allocating onto the heap.

C++ also has more intuitive memory management. Allocating objects on the heap only requires the keyword "new" and removing it uses only the keyword delete. The programmer will no longer have to use sizeof operator which will make the code neater.

2 2007 Paper 3 Question 4

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2007p3q4.pdf

A C programmer is working with a little-endian machine with 8 bits in a byte and 4 bytes in a word. The compiler supports unaligned access and uses 1, 2 and 4 bytes to store char, short and int respectively. The programmer writes the following definitions (below right) to access values in main memory (below left):

Address	Byte offset			
	0	1	2	3
0x04	10	00	00	00
0x08	61	72	62	33
0x0c	33	00	00	00
0x10	78	0c	00	00
0x14	08	00	00	00
0x18	01	00	4c	03
0x1c	18	00	00	00

```
int **i=(int **)0x04;

short **pps=(short **)0x1c;

struct i2c {
  int i;
  char *c;
}*p=(struct i2c*)0x10;
```

(a) Write down the values for the following C expressions:

$$**i \qquad p \to c[2] \qquad \&(*pps)[1] \qquad ++p \to i$$

- $**i$ has value 3192 of type int.

  i is a pointer to $0x04$. So $*i$ returns the value stored at $0x04$. Which is the address $0x10$. $**i$ returns the value stored at $0x10$ – which is $12 \cdot 16^2 + 7 \cdot 16 + 8 = 3192$.

- $p \to c[2]$ has value '\0' of type char.

  $p \to c[2]$ is the dereference of the address of the c attribute of p plus two bytes (as a char in C is 1 byte). p is the struct i2c with memory starting at 0x10. The i attribute is 4 bytes. So the c attribute is a pointer to $0x14$. This can be viewed as an array starting at $0x14$. $p \to c[2]$ is the second element of this array. Since chars have size 1 byte, the second element of this array is at address $0x16$ – which has value 0 – or '\0' in char.

- $\&(*pps)[1]$ is a pointer of type $short*$ to $0x1a$.

  pps first dereferences pps. This is a pointer to $0x18$. $*pps$ has type $*short$ – which can be viewed as an array of shorts. [1] finds the second element in this array. This can be done by adding the size of a short to the address $*pps$. Shorts in C are two bytes. Therefore the address of $(*pps)[1]$ is $0x1a$. So $\&(*pps)[1]$ is a pointer to $0x1a$ and is of type short.

- $++p \to i$ has value 3193 and is of type int.

  This accesses the i element of p, increments it and returns the incremented value. The i attribute of p is stored at address 0x10. This has value 3192. So we increment this by one and return the incremented value. Therefore this expression has value 3193 and type int.

(b) Explain why the code shown below, when executed will print the value 420.

```
1 #include <stdio.h>
2
3 #define init_employee(X,Y) {(X),(Y), wage_emp}
4 typedef struct Employee Em
5 struct Employee {int hours, salary; int (*wage) (Em*);};
6 int wage_emp(Em *ths) {return ths->hours*ths->salary;}
7
```

```
 8 #define init_manager(X,Y,Z) {(X),(Y),wage_man,(Z)}
 9 typedef struct Manager Mn;
10 struct Manager {int hours,salary;int (*wage)(Mn*);int bonus;};
11 int wage_man(Mn *ths){return ths->hours*ths->salary+ths->bonus;}
12
13 int main(void){
14         Mn m = init_manager(40,10,20);
15         Em *e= (Em *) &m;
16          printf("%d\n",e->wage(e));
17          return 0;
18 }
```

The datatype Em has fields hours, salary and a function pointer (the function pointer is called wage – the function is not necessarily called wage) pointing to a function that takes an argument of type Em* and returns an integer. Mn is the same but with two differences: the function takes argument of type Mn and there is an additional field "bonus".

#define is used to define macros in the C preprocessor. The first argument will be replaced by the second argument.

The two macros on lines 3 & 8 tell the C preprocessor to replace all occurrences of init_employee and init_manager in the rest of the program with initialisers for the respective structs. This means init_employee and init_manager now function as initialisers.

Therefore "Mn m = init_manager(40, 10, 20);" (line 14) will initialise a struct of type manager on the stack who works 40 hours at salary 10 with a bonus of 20.

On line 15, we create a pointer e of type Em * to this object – this requires an unsafe cast. C is not strongly typed and so this is allowed.

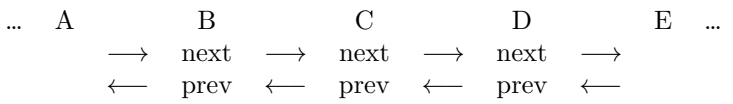
In C, attributes of structs are accessed only by offset. So the attempt to access the wage function pointer of e will access the bits at offset 2 * sizeof(int). Since the first fields of Mn are the same as the first fields of Em; the offset for the function pointer of Mn is the same as the offset for the function pointer of Em. Therefore e->wage will access the function pointer of m. This is a pointer to the wage_man function. This is then passed e. e is interpreted by the function wage_man as being of type Mn. This is the same as passing a pointer to m to wage_man. Therefore the return value of the function call is $40 * 10 + 20 = 420$. This is then passed to printf – therefore the program outputs 420 as required.

3 2010 Paper 3 Question 6

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2010p3q6.pdf

(a) Popular programming journal *Obscure C Techniques for Experts* has published a novel way to save space for a doubly-linked list program. Instead of storing two pointers (one next and one previous), this new technique stores a single value: the XOR of *previous* and *next* pointers.

A traditional two-pointer linked list might be illustrated as:



In contrast, the proposed new technique stores a bit-wise XOR of the *previous* and *next* pointers within a single field.

You have been engaged to provide code examples of this approach for publication.

Ensure your code illustrates the creation and initialisation of such a list as well as the insertion, and deletion of elements from such a list. Additionally, you must provide examples of a forward or backward traversal of the list permitting combination of each element in turn.

```c
#include <stdlib.h>

typedef struct List * list;

struct List {
        int value;
        list xorpntr;
};

list create_list(int head_value){
        list lst = malloc(sizeof(struct List));
        lst->value = head_value;
        // NULL ^ NULL = NULL
        lst->xorpntr = NULL;
        return lst;
}

void free_list(*list lst){
        // check for NULL
        if (!lst) return;
        /*
        The while condition is true until both pointers are equal
        The linked list is not cyclic so the pointers should only
        be equal when both are NULL -- when we're on the last node
        */
        while (lst->xorpntr){
                /*
                if one pointer is NULL, xorpntr is the
                value of the other pointer.
                This updates the next nodes xorpntr
                */
                lst->xorpntr->xorpntr ^= lst;
                list prev = lst;
                lst = lst->xorpntr;
                free(prev);
        }
        free(lst);
}

void addfirst(list *lst, int val){
        list fst = malloc(sizeof(struct List));
        fst->value = val;
        fst->xorpntr = *lst;
        (*lst)->xorpntr ^= fst;
        *lst = fst;
}
```

```
void addlast(list lst, int val){
        list current = lst;
        list next = current->xorpntr;
        // if there is only one element in the list
        if (!next){
                list last = malloc(sizeof(struct List));
                current->xorpntr = last;
                last->xorpntr = NULL;
                last->value = val;
                return;
        }
        // while ends when next is the last element in the list
        while (next->xorpntr != current){
                list tmp = next;
                next = next->xorpntr^current;
                current = tmp;
        }
        list last = malloc(sizeof(struct List));
        last->value = val;
        last->xorpntr = next;
        next->xorpntr = current^last;
}

int popFirst(list *lst){
        // check for NULL
        if (!lst) return 0;
        // this works for list of length 1+
        int i = (*lst)->value;
        list tmp = *lst;
        *lst = (*lst)->xorpntr;
        free(*lst);
        return i;
}

int popLast(list *lst){
        // check for NULL
        if (!(*lst)) return 0;
        // check for list of length 1
        if (!(*lst)->xorpntr){
                int i = (*lst)->value;
                free(*lst);
                *lst = NULL;
                return i;
        }
        int current = *lst;
        int next = current->xorpntr;
        /*
        Terminates when next is the last node and
        current is the penultimate node
        */
        while (next->xorpntr != current){
                current = next;
                next = next->xorpntr;
        }
        int i = next->value;
```

```
        // update penultimate pointer
        current->xorpntr ^= next;
        // free last node
        free(next);
        return i;
}
```

(b) Comment on this form of linked list. Consider the comparative speed, memory overheads and maintenance and other advantages or disadvantages of the XOR doubly-linked list approach when compared with an approach that stores both *previous* and *next* pointers.

The primary advantage of the XOR list is that it requires less memory. Each node now only requires one pointer (size) rather than two. This will reduce the size of the linked list structure significantly – especially in C as structs have no additional information. In a memory-critical system this could be a significant advantage.

However, iterating through the XOR list is more complicated and slower. Rather than just looking up a value (which will likely already be in cache), we've got to perform an XOR operation which will double the amount of machine instructions. I'd like to note that iterating through a linked list is dominated by memory accesses rather than cpu time – so this will not double the time; just increase it. Additionally, the programmer has to do more work, will write code that is more difficult to understand and less intuitive. This increases the probability of a bug which could lead to accessing unallocated memory or dereferencing a NULL pointer.

Furthermore, it's much harder implement cyclical linked lists in the XOR list or incorporate it into larger data structures. If we pass a reference to a node in the linked list, we cannot use it. We need two nodes to iterate in one direction – in the base case we know one pointer is NULL and so can iterate that in the other direction. Therefore to use the XOR list in a cyclical linked list, we need to pass two nodes which adds extra work to external methods and breaks encapsulation. If we want to use the XOR list in another structure such as a Fibonacci Heap, then we will have to hold the node and an adjacent node – which doubles the space requirement for the linked list in the fibonacci heap – which cancels out the memory gain in the XOR list itself. Therefore there is no reason to use the XOR list in another more complicated data structure.

4 2014 Paper 3 Question 3

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2014p3q3.pdf

(a) Write a C function revbits() which takes a single 8-bit char parameter and returns a char result by reversing the order of the bits in the char.

```
char revbits(char c){
        char x = 0;
        int j = 128;
        for (int i = 1; i <= 128; i *= 2, j /= 2){
                if (c & i) {
                        x |= j;
                }
        }
        return x;
}
```

(b) Write a C function revbytes() taking two parameters and returning no result. The first parameter is a pointer to memory containing n contiguous bytes (each of type char), and the second is the number of bytes. The function should have the side effect of

reversing the order of the bits in the n contiguous bytes, seen as a bitstring of length $8n$. For example the first bit of the first char should be swapped with the last bit of the last char.

```
void revbytes(char *mem, int n){
        int lo = 0;
        int hi = n - 1;
        while (lo < hi){
                char tmp = revbits(mem[lo]);
                mem[lo++] = revbits(mem[hi]);
                mem[hi--] = tmp;
        }
        if (n % 2) {
                mem[n / 2] = revbits(mem[n / 2];
        }
}
```