

Week 01a Problem Set

Elementary Data and Control Structures in C

[\[Show with no answers\]](#) [\[Show with all answers\]](#)

The purpose of the first problem set is mainly to familiarise yourself with programming in C.

For the remainder of the course it is vital that you understand and are able to solve all of the Exercises 1–7 below. In addition, the following site, for example, offers many small programming puzzles for you to solve:

C Puzzles

(*Hint*: Relevant are puzzles E, L, 1D and 2D. Pointers and random numbers will be introduced later in COMP9024.)

1. (Numbers)

There is a 5-digit number that satisfies $4 \cdot abcde = edcba$, that is, when multiplied by 4 yields the same number read backwards. Write a C-program to find this number.

Hint: Only use arithmetic operations; do not use any string operations.

[\[hide answer\]](#)

```
#include <stdio.h>

#define MIN 10000
#define MAX 24999    // solution has to be <25000

int main(void) {
    int a, b, c, d, e, n;

    for (n = MIN; n <= MAX; n++) {
        a = (n / 10000) % 10;
        b = (n / 1000) % 10;
        c = (n / 100) % 10;
        d = (n / 10) % 10;
        e = n % 10;
        if (4*n == 10000*e + 1000*d + 100*c + 10*b + a) {
            printf("%d\n", n);
        }
    }
    return 0;
}
```

Solution: 21978

2. (Characters)

Write a C-program that outputs, in alphabetical order, all strings that use each of the characters 'c', 'a', 't', 'd', 'o', 'g' exactly once.

How many strings does your program generate?

[\[hide answer\]](#)

There are $6! = 720$ permutations of "catdog".

A straightforward solution is to use six nested loops and a conditional statement to filter out all strings with duplicate characters. The following program includes a counter to check how many strings have been generated.

```
#include <stdio.h>

int main(void) {
    char catdog[] = { 'a', 'c', 'd', 'g', 'o', 't' };

    int count = 0;
    int i, j, k, l, m, n;
    for (i=0; i<6; i++)
        for (j=0; j<6; j++)
            for (k=0; k<6; k++)
                for (l=0; l<6; l++)
```

```

        for (m=0; m<6; m++)
            for (n=0; n<6; n++)
                if (i!=j && i!=k && i!=l && i!=m && i!=n &&
                    j!=k && j!=l && j!=m && j!=n &&
                    k!=l && k!=m && k!=n &&
                    l!=m && l!=n && m!=n) {
                    printf("%c%c%c%c%c%c\n", catdog[i], catdog[j],
                        catdog[k], catdog[l],
                        catdog[m], catdog[n]);

                    count++;
                }
    printf("%d\n", count);
    return 0;
}

```

3. (Elementary control structures)

a. Write a C-function that takes a positive integer n as argument and outputs a series of numbers according to the following process, until 1 is reached:

- if n is even, then $n \leftarrow n/2$
- if n is odd, then $n \leftarrow 3*n+1$

b. The Fibonacci numbers are defined as follows:

- $\text{Fib}(1) = 1$
- $\text{Fib}(2) = 1$
- $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$ for $n \geq 3$

Write a C program `fibonacci.c` that applies the process described in Part a. to the first 10 Fibonacci numbers.

The output of the program should begin with

```

Fib[1] = 1
1
Fib[2] = 1
1
Fib[3] = 2
2
1
Fib[4] = 3
3
10
5
16
8
4
2
1

```

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` program that corresponds to this exercise. It expects to find a program named `fibonacci.c` in the current directory. You can use `dryrun` as follows:

```
prompt$ 9024 dryrun fibonacci
```

Note: Please ensure that your output follows exactly the format shown above.

[\[hide answer\]](#)

```

#include <stdio.h>

#define MAX 10

void collatz(int n) { // named after the German mathematician who invented this problem
    printf("%d\n", n);
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3*n + 1;
        }
        printf("%d\n", n);
    }
}

```

```

}

int main(void) {
    int fib[MAX] = { 1, 1 };    // initialise the first two numbers
    int i;
    for (i = 2; i < MAX; i++) { // compute the first 10 Fibonacci numbers
        fib[i] = fib[i-1] + fib[i-2];
    }




    for (i = 0; i < MAX; i++) { // apply Collatz's process to each number
        printf("Fib[%d] = %d\n", i+1, fib[i]);
        collatz(fib[i]);
    }

    return 0;
}

```

4. (Elementary data structures)

Define a data structure to store all information of a single ride with the Opal card. Here are three sample records:

Transaction number	Date/time	Mode	Details	Journey number	Fare Applied	Fare	Discount	Amount
3505	Sun 28/07/2019 13:17		Kings Cross to Strathfield	9	Travel Reward	\$4.48	\$2.92	-\$1.56
3503	Fri 26/07/2019 18:00		Anzac Pde bf Addison St to Taylor Square	8		\$3.73	\$0.00	-\$3.73
3501	Fri 26/07/2019 10:20		Flinders St af Oxford St to UNSW	7		\$3.73	\$0.00	-\$3.73

You may assume that individual stops (such as "Anzac Pde bf Addison St") require no more than 31 characters.

Determine the memory requirements of your data structure, assuming that each integer and floating point number takes 4 bytes.

If you want to store millions of records, how would you improve your data structure?

[\[hide answer\]](#)

There are of course many possible ways in which this data can be structured; the following is just one example:

```

typedef struct {
    int day, month, year;
} DateT;

typedef struct {
    int hour, minute;
} TimeT;

typedef struct {
    int transaction;
    char weekday[4];           // 3 chars + terminating '\0'
    DateT date;
    TimeT time;
    char mode;                 // 'B', 'F' or 'T'
    char from[32], to[32];
    int journey;
    char faretext[12];
    float fare, discount, amount;
} JourneyT;

```

Memory requirement for one element of type JourneyT: $4 + 4 + 12 + 8 + 1 + (3 \text{ padding}) + 2 \cdot 32 + 4 + 12 + 3 \cdot 4 = 124$ bytes.

The data structure can be improved in various ways: encode both origin and destination (*from* and *to*) using Sydney Transport's unique stop IDs along with a lookup table that links e.g. 203311 to "UNSW"; use a single integer to encode the possible "Fare Applied" entries; avoid storing redundant information like the weekday, which can be derived from the date itself.

5. (Stack ADO)

- a. Modify the Stack ADO from the lecture ([Stack.h](#) and [Stack.c](#)) to implement an integer stack ADO ([IntStack.h](#) and [IntStack.c](#)).
- b. Complete the test program below ([StackTester.c](#)) and run it to test your integer stack ADO. The tester
 - initialises the stack
 - prompts the user to input a number n
 - checks that n is a positive number
 - prompts the user to input n numbers and push each number onto the stack
 - *uses the stack to output the n numbers in reverse order* (needs to be implemented)

StackTester.c

```
// Integer Stack ADO tester ... COMP9024 19T3
#include <stdio.h>
#include <stdlib.h>
#include "IntStack.h"

int main(void) {
    int i, n;
    char str[BUFSIZ];

    StackInit();

    printf("Enter a positive number: ");
    scanf("%s", str);
    if ((n = atoi(str)) > 0) {    // convert to int and test if positive
        for (i = 0; i < n; i++) {
            printf("Enter a number: ");
            scanf("%s", str);
            StackPush(atoi(str));
        }
    }

    /* NEEDS TO BE COMPLETED */

    return 0;
}
```

An example of the program executing could be

```
Enter a positive number: 3
Enter a number: 2019
Enter a number: 12
Enter a number: 25
25
12
2019
```

[\[hide answer\]](#)

a. IntStack.h

```
// Integer Stack ADO header file

#define MAXITEMS 10

void StackInit();    // set up empty stack
int  StackIsEmpty(); // check whether stack is empty
void StackPush(int); // insert int on top of stack
int  StackPop();     // remove int from top of stack
```

IntStack.c

```
// Integer Stack ADO implementation
#include "IntStack.h"
#include <assert.h>

typedef struct {
    int item[MAXITEMS];
    int top;
} stackRep;           // defines the Data Structure
```

```

static stackRep stackObject; // defines the Data Object

void StackInit() {           // set up empty stack
    stackObject.top = -1;
}

int StackIsEmpty() {         // check whether stack is empty
    return (stackObject.top < 0);
}

void StackPush(int n) {      // insert int on top of stack
    assert(stackObject.top < MAXITEMS-1);
    stackObject.top++;
    int i = stackObject.top;
    stackObject.item[i] = n;
}

int StackPop() {             // remove int from top of stack
    assert(stackObject.top > -1);
    int i = stackObject.top;
    int n = stackObject.item[i];
    stackObject.top--;
    return n;
}

```

b. StackTester.c

```

// Integer Stack ADO tester ... COMP9024 19T3
#include <stdio.h>
#include <stdlib.h>
#include "IntStack.h"

int main(void) {
    int i, n;
    char str[BUFSIZ];

    StackInit();

    printf("Enter a positive number: ");
    scanf("%s", str);
    if ((n = atoi(str)) > 0) { // convert to int and test if positive
        for (i = 0; i < n; i++) {
            printf("Enter a number: ");
            scanf("%s", str);
            StackPush(atoi(str));
        }
    }
    while (!StackIsEmpty()) {
        printf("%d\n", StackPop());
    }
    return 0;
}

```

6. (ADO client)

A stack can be used to convert a positive decimal number n to a different numeral system with base k according to the following algorithm:

```

while  $n > 0$  do
    push  $n \% k$  onto the stack
     $n = n / k$ 
end while

```

The result can be displayed by printing the numbers as they are popped off the stack. Example ($k=2$):

```

n = 13      --> push 1 (= 13%2)
n = 6  (= 13/2) --> push 0 (= 6%2)
n = 3  (= 6/2)  --> push 1 (= 3%2)
n = 1  (= 3/2)  --> push 1 (= 1%2)
n = 0  (= 1/2)
Result: 1101

```

Using your stack ADO from Exercise 5, write a C-program that implements this algorithm to convert to base $k=2$ a number given on the command line.

Examples of the program executing could be

```
prompt$ ./binary
Enter a number: 13
1101
prompt$ ./binary
Enter a number: 128
10000000
prompt$ ./binary
Enter a number: 127
1111111
```

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` program that corresponds to this exercise. It expects to find three programs in the current directory:

- `IntStack.h` – your header file for the integer stack from Exercise 5
- `IntStack.c` – your implementation of the integer stack from Exercise 5
- `binary.c`

You can use `dryrun` as follows:

```
prompt$ 9024 dryrun binary
```

[\[hide answer\]](#)

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

int main(void) {
    int n;
    char str[BUFSIZ];

    printf("Enter a number: ");
    scanf("%s", str);
    n = atoi(str);
    StackInit();
    while (n > 0) {
        StackPush(n % 2);
        n = n / 2;
    }
    while (!StackIsEmpty()) {
        printf("%d", StackPop());
    }
    putchar('\n');
    return 0;
}
```

7. (Queue ADO)

Modify your integer stack ADO from Exercise 5 to an integer queue ADO.

Hint: A *queue* is a FIFO data structure (first in, first out). The principal operations are to *enqueue* and to *dequeue* elements. Elements are dequeued in the same order in which they have been enqueued. Below is a sample header file ([IntQueue.h](#)) to get you started.

`IntQueue.h`

```
// Integer Queue ADO header file ... COMP9024 19T3
#define MAXITEMS 10

void QueueInit();           // set up empty queue
int QueueIsEmpty();         // check whether queue is empty
void QueueEnqueue(int);     // insert int at end of queue
int QueueDequeue();         // remove int from front of queue
```

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` program that corresponds to this exercise. It expects to find two files named `IntQueue.c` and `IntQueue.h` in the current directory that provide an implementation of a queue ADO with the four queue functions shown above. You can use `dryrun` as follows:

prompt\$ **9024 dryrun IntQueue**[\[hide answer\]](#)**IntQueue.c**

```
// Integer Queue ADO implementation ... COMP9024 19T3
#include "IntQueue.h"
#include <assert.h>

static struct {
    int item[MAXITEMS];
    int top;
} queueObject; // defines the Data Object

void QueueInit() {           // set up empty queue
    queueObject.top = -1;
}

int QueueIsEmpty() {         // check whether queue is empty
    return (queueObject.top < 0);
}

void QueueEnqueue(int n) {   // insert int at end of queue
    assert(queueObject.top < MAXITEMS-1);
    queueObject.top++;
    int i;
    for (i = queueObject.top; i > 0; i--) {
        queueObject.item[i] = queueObject.item[i-1]; // move all elements up
    }
    queueObject.item[0] = n; // add element at end of queue
}

int QueueDequeue() {         // remove int from front of queue
    assert(queueObject.top > -1);
    int i = queueObject.top;
    int n = queueObject.item[i];
    queueObject.top--;
    return n;
}
```

8. Challenge Exercise

Write a C-function that takes 3 integers as arguments and returns the largest of them. The following restrictions apply:

- You are not permitted to use **if** statements.
- You are not permitted to use loops (e.g. **while**).
- You are not permitted to call any function.
- You are only permitted to use data and control structures introduced in Week 1's lecture.

[\[hide answer\]](#)

The following makes use of the fact that a true condition has value 1 and a false condition has value 0:

```
int max(int a, int b, int c) {
    int d = a * (a >= b) + b * (a < b); // d is max of a and b
    return c * (c >= d) + d * (c < d); // return max of c and d
}
```