McMaster University
Department of Computing and Software
Dr. W. Kahl

SFWR ENG 2S03
Exercise Sheet 6
Solution Hints

SFWR ENG 2S03 — Principles of Programming

18 October 2006

Exercise 6.1 — Calendar (22% of Final 2003)

For a calendar application, a year will be represented by **a single contiguous array** of days, called a **"year array"**.

For making access easier, a **"month start array"** will be calculated, containing for each month index *i* the index that the first day of month *i* has in year arrays.

Example: In a normal (i.e., non-leap) year, the first four elements (at indices 0, 1, 2, 3) of the month start array will be 0, 31, 59, 90.

Note: The items (a) and (b) are completely independent of each other.

(a) ≈10% Implement the C function

int * *startDays*(int *monthsNum* , const int *monthLen*[] , int * *yearLen*)

that

- returns a pointer to the beginning of a newly allocated month start array which should have *monthsNum* elements,
- initialises this new month start array according to the month lengths found in the *monthsNum*-element array *monthLen*, and
- writes the number of days the whole year has in this calendar into the reference parameter *yearLen*.

(b) ≈12% Implement the iterative C function

void *printDate*(int *monthsNum* , int *monthStart*[] , int *index*)

that, given a number of months and a month start array, uses **binary search** to find the month containing the day with index *index* in a year array; it should then print (to standard output) a message containing the day in that month and the number of the month as user-level day and month numbers.

Example: For index 0 it should print "`Day 1 month 1`", and for index 33 (using the standard calendar) it should print "`Day 3 month 2`".

Let the following enumeration type definition be given:

typedef enum {*SUN*, *MON*, *TUE*, *WED*, *THU*, *FRI*, *SAT* } *Weekday*;

(c) new Write a C function *weekday* that, given a month start array *monthStart*, the weekday *wd1* of the first day of the year (for 2003 this would be *WED*), and two int values *month* and a *day*, returns the weekday of the day indicated by *month* and a *day*, which are supplied as user-level numbers: For the 21st October, these arguments would be *month*=10 and *day*=21.

Solution Hints

```
#include <stdio.h>
#include <stdlib.h>
typedef int bool;
#define TRUE 1
#define FALSE 0
```

If memory allocation for the result array fails, *NULL* is returned, and we leave the decision to the caller whether or not to print a failure message.

However, the number of days of the year can still be calculated even if the memory allocation for the result array failed, so we do that.

```
int * startDays(const int monthsNum, const int monthLen[], int * yearLen) {
  int * result = malloc(monthsNum * sizeof(int));
  int i, s = 0;
  for ( i = 0; i < monthsNum; i++ ) {
    if ( result ≠ NULL ) { result[i] = s; }
    s += monthLen[i];
  }
  *yearLen = s;
  return result;     // pass the burden of error handling to caller
}
```

Standard binary search:

- initialise *lower* and *upper* to the extremes of the search range
- if the range has collapsed: *lower* (== *upper*) must be the index of the month we are looking for.
- if the range has not collapsed:
  - calculate the middle index *k* such that *k* is not equal to *lower*
  - Select the subrange to continue.

For printing the result, we have to take care to convert array indices (starting at 0) into natural-language ordinal numbers (starting at 1).

```
void printDate(int monthsNum, const int monthStart[], int index) {
  int lower = 0, upper = monthsNum – 1;
  int k;
  while ( upper > lower ) {
    k = (upper + lower + 1) / 2;
    if ( index ≥ monthStart[k])
       lower = k;
    else
       upper = k – 1;
  }
  printf("Day: %d, month: %d\n", index + 1– monthStart[lower], lower + 1);
}
```

calIndex(*monthStart*, *month*, *day*) considers *month* and *day* as natural-language ordinal numbers (starting at 1) and returns the calendar array index corresponding to the day indicated by *month* and *day* in calendar arrays governed by month start indices *monthStart*.

```
int calIndex(const int monthStart[], int month, int day) {
  return monthStart[month – 1] + day – 1;
}
```

```
typedef enum {SUN, MON, TUE, WED, THU, FRI, SAT } Weekday;
```

We now employ the fact that we know ***which*** integers the *Weekday* constants are, and that "%" returns non-negative integers less than its second argument.

```
Weekday weekday(const int monthStart[], Weekday wd1, int month, int day) {
  return (calIndex(monthStart, month, day) + wd1) % 7;
}
```

The *main* function here first prints the result of *startDays*, and then processes its argument list; the executable can be called in two ways:

```
./Calendar 294                # testing  printdate
./Calendar 21 10              # testing  weekday
```

```
int main(int argc, char * argv[]) {
  const char * weekdays[] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
  const int monthsNum = 12;
  const int monthLen[12] = // 2004 is a leap year!
          {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  int yearLen;
  Weekday w, wd1 = THU; // for 2004
  const int * monthStart = startDays(monthsNum, monthLen, &yearLen);
  int i, d, m;

  if (monthStart == NULL) {
    fprintf(stderr, "%s: Could not allocate memory for month start array!\n", argv[0]);
    return 1;         // exit status indicating error
  }
  for (i=0; i<monthsNum; i++) {
    w = weekday(monthStart, wd1, i+1, 1);  // weekday of first of month
    printf("Month %d has %d days and starts on a %s, day number %d\n",
          i+1, monthLen[i], weekdays[w], monthStart[i]);
  }
  printf("The year has %d days.\n", yearLen);

  if (argc == 2) {        // Argument read as "day index"
    d = atoi(argv[1]);
    printDate(monthsNum, monthStart, d);
```

```
  }
  if (argc > 2) {          // Arguments read as "day, month"
    d = atoi(argv[1]);
    m = atoi(argv[2]);
    w = weekday(monthStart, wd1, m, d);
    printf("Day %d of month %d is a %s.\n", d, m, weekdays[w]);
  }
  return 0;
}
```

Exercise 6.2 — Calendar (modified 23% of Final 2003)

For the calendar application of Exercise 6.1:

(a) Write and document **appropriate** type definitions for the calendar data — of type *Day* — to be stored in year arrays.

For each day, there should be the times of sunrise and sunset, and up to 10 appointments.

An appointment — of type *Appointment* — has begin and end times, a title string, and a comment string.

(b) **Design and implement** a C function *find* that accepts the following parameters:

- the number of months and a month start array,
- the number of days in the year and a year array containing *Day* elements,
- a **function** *check* that takes an *Appointment* — see (c) — as argument and returns either *NULL* to signal that the argument *Appointment* is irrelevant, or a pointer to a string containing a message to be printed.

The function *find* should apply *check* to all appointments in the year array, and for each appointment for which a message is returned, it should print the message and use *printDate* from (b) above to print the date at which the appointment was found.

(c) [new] Implement argument functions for *find* from (b), e.g.:

- *checkWhite* finds appointments where the comment string contains only white-space characters, and returns a message transscribing the comment into a C string literal.

  So if the comment consisted of an empty line, and a line containing a space and a tab character, the returned message, when printed to the screen, would contain the nine-character string "\n \t\n".

  (For manually generating this, you would write: "\"\\n \\t\\n\"".)
- *checkBirthday* finds birthdays: If the comment of an appointment does not contain (case insensitive) the sub-string "birthday", it returns *NULL*. If a birthday comment starts with "Birthday: ", then *checkBirthday* only returns the suffix after that prefix, otherwise the whole comment.

(d) [new] Write a *main* program to test everything!

Solution Hints

Different ways to implement "up to ten" appointments are possible — here we choose a solution that does not involve and *malloc*/*free* for adding and deleting appointments, and uses an explicit couter rather than some “invalid begin time” sentinel value to indicate which array entries are valid appointments.

```
typedef struct { int hour, minutes; } MyTime;

typedef struct {
   MyTime begin, end;
   char * title;        // allocated via malloc
   char * comment;      // allocated via malloc
 } Appointment;
```

It is essential that allocation assumptions are documented!

```
#define MAXAPPOINTMENTS 10
typedef struct {
   MyTime sunrise, sunset;
   Appointment[MAXAPPOINTMENTS] appointments;
   int numberOfAppointments;
 } Day;
```

Linked lists have not yet been presented, and are therefore not expected here.

There are of course different ways to handle “up to ten appointments”: They could be *malloc*ed and the array would then contain pointers; with that option, one could also make it a *NULL*-terminated 11-element array.

Even with *Appointment*s in the array (and not pointers), one could still use some kind of sentinel values for termination, for example *NULL* titles or negative times.

```
void find(int monthsNum, int monthStart[], int yearLen, Day cal[], char * (*check)(Appointment a)) {
  int i,j;
  char * message;
  Appointment * l;
  for ( i=0; i<yearLen; i++ ) {
   l = cal[i].appointments;
   for ( j=0; j<MAXAPPOINTMENTS; j++ ) {
     if ( (message = check(l[j])) ) {
       printf("%s  ", message);
       printDate(monthsNum, monthStart, i);
     }
     l = l→next;
   }
  }
}

char * checkWhite(Appointment a) {
  char * s = a.comment;
  bool allSpace = True;
  while (allSpace && *s) { allSpace &&= isSpace(*s); }
  if ( allSpace ) {
   char msg [strlen(a.comment) + 30] = "All white!";
```

```
    return msg;
  }
  else return NULL;
}
```

(c) and (d) not yet covered.

Exercise 6.3 — Typing (22% of Midterm 2, 2005)

Give variable declarations (and only variable **declarations**) to preceed the following statements so that the resulting code is valid ANSI C. In each case, you must provide **the most appropriate type**.

(a) `d = 0.5;`

Solution Hints

`double d;`

(b) `*p = q + 0.5;`

Solution Hints

`double q, p[1];`

(c) `p = q + *q;`

Solution Hints

The following is not really "only a declaration":

`int *p, q[1] = {2};`

(d) `array[3] = 3.14;`

Solution Hints

`double array[N];`

for some $N > 3$

(e) `*answer = 42;`

Solution Hints

`int answer[1];`

Declaring as pointer "`int * answer`" without initialisation is "dynamically invalid".

(f)
```
array = malloc( 10 * sizeof (double) );
array[6] = 2.73e5;
```

Solution Hints

`double *array;`

(g)
```
matrix = malloc( 5 * sizeof (double *) );
matrix[2] = malloc( 8 * sizeof (double) );
matrix[2][4] = 0.0;
```

Solution Hints

`double **matrix;`
