02635 Fall 2016 — Module 4 (solutions)

Exercises

Exercise 7-1

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

int main(void) {

  int n = 0, N = 4;
  double *data = NULL, *tmp = NULL, val = 0.0, avg = 0.0;

  // Allocate storage for N doubles
  data = (double *) malloc(N*sizeof(double));
  if (!data) {
    printf("Memory allocation failed\n");
    return EXIT_FAILURE;
  }

  // Prompt user to enter data
  printf("Please enter your data:\n>");
  while (scanf("%lf", &val) == 1) {
    if (n == N) {
      // Ask for more memory: allocate 4 more doubles
      N = N + 4;
      tmp = (double *)realloc(data, N * sizeof(double));
      if (!tmp) {
        // Reallocation failed
        printf("Memory reallocation failed\n");
        free(data);
        return EXIT_FAILURE;
      }
      data = tmp;
    }
    data[n++] = val;
    printf(">");
  }
```

```
  // Compute and print average
  printf("You entered %d values.\n",n);
  for (size_t i=0;i<n;i++) {
    avg += data[i];
  }
  avg /= n;
  printf("Average value: %g\n",avg);

  // Free allocated memory and exit
  free(data);
  return EXIT_SUCCESS;
}
```

Exercise 7-4

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

#define REC_PER_DAY 6

int main(void) {

  int ndays = 0, N = 4;
  double *data = NULL, *tmp = NULL, *avg = NULL;
  char choice = 'n';

  // Allocate memory for N days
  data = (double *)malloc(N*REC_PER_DAY*sizeof(double));
  if (!data) {
    printf("Memory allocation failed.\n");
    exit(-1);
  }

  // Prompt user to input data
  do {
    if (ndays == N) {
      // Ask for more memory: N+4 days
      N += 4;
      tmp = realloc(data, N*REC_PER_DAY*sizeof(double));
      if (!tmp) {
         printf("Reallocation failed.\n");
         free(data);
         return EXIT_FAILURE;
```

```
      }
      data = tmp;
    }
    printf("Input data for day %d:\n",ndays+1);
    for (size_t i=0;i<REC_PER_DAY;i++) {
      printf("  Measurement #%d: ",i+1);
      scanf("%lf", data + ndays*REC_PER_DAY + i);
    }
    ndays++;
    printf("Continue? [y/n] ");
    scanf(" %c",&choice);
  } while (choice == 'y');

  // Allocate memory for average temperature
  avg = (double *)malloc(ndays*sizeof(double));
  if (!avg) {
    printf("Allocating memory for avg. temperature failed.\n");
    free(data);
    return EXIT_FAILURE;
  }

  // Compute and print averages
  for (int i=0;i<ndays;i++) {
    avg[i] = 0.0;
    for (int j=0;j<REC_PER_DAY;j++)
      avg[i] += data[i*REC_PER_DAY+j];
    avg[i] /= REC_PER_DAY;
    printf("Day %d average: %g\n",i+1,avg[i]);
  }

  // Free memory and exit
  free(avg);
  free(data);
  return EXIT_SUCCESS;
}
```

Exercise 8-1

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

double average(double *data, size_t n);

int main(void) {
```

```c
  int n = 0, N = 4;
  double *data = NULL, *tmp = NULL, val = 0.0;

  // Allocate storage for N doubles
  data = (double *) malloc(N*sizeof(double));
  if (!data) {
    printf("Memory allocation failed\n");
    return EXIT_FAILURE;
  }

  // Prompt user to enter data
  printf("Please enter your data:\n>");
  while (scanf("%lf", &val) == 1) {
    if (n == N) {
      // Ask for more memory: allocate 4 more doubles
      N = N + 4;
      tmp = (double *)realloc(data, N * sizeof(double));
      if (!tmp) {
        // Reallocation failed
        printf("Memory reallocation failed\n");
        free(data);
        return EXIT_FAILURE;
      }
      data = tmp;
    }
    data[n++] = val;
    printf(">");
  }

  // Print out data average
  printf("You entered %d values.\n",n);
  printf("Average value: %g\n", average(data,n));

  // Free allocated memory and exit
  free(data);
  return EXIT_SUCCESS;
}

// Auxilliary function: computes average of double array of length n
double average(double *data, size_t n) {
  double avg = 0.0;
  for (size_t i=0; i<n; i++)  {
    avg += data[i];
  }
```

```
  avg /= n;
  return avg;
}
```

Optional exercise

Rewrite your code from Part II of the module 3 exercises using what you have learned from chapter 8 in "Beginning C". Your code should

- *be modular and consist components/functions that are easy to test;*
- *use dynamic memory allocation for arrays.*

`main.c`

```c
#include <stdlib.h>
#include <stdio.h>
#include "numint.h"

int main(void) {

  int n=0;                  // number of subintervals
  size_t M=0, N=0;          // vertical/horizontal grid points
  double xl=0.0,yl=0.0;  // upper-right corner of grid
  double xu=1.0,yu=1.0;  // lower-left corner of grid
  double x1=0.0,y1=0.0;  // point 1 on line
  double x2=0.0,y2=0.0;  // point 2 on line
  double **grid = NULL;
  double val = 0.0;

  // Prompt user to enter data
  printf("Enter number of subintervals: ");
  scanf("%d",&n);

  printf("Enter grid dimensions M,N: ");
  scanf("%zd,%zd",&M,&N);
  grid = create_grid(M,N,xl,yl,xu,yu);
  if ( grid == NULL ) {
    printf("Creating grid array failed.\n");
    return EXIT_FAILURE;
  }

  printf("Enter x1,y1: ");
  scanf("%lf,%lf",&x1,&y1);
  printf("Enter x2,y2: ");
  scanf("%lf,%lf",&x2,&y2);

  printf("Computing integral over line segment between (%.2f,%.2f)"
         " and (%.2f,%.2f):\n",x1,y1,x2,y2);
  val = line_int(M, N, grid, xl, yl, xu, yu, x1, y1, x2, y2, n);
  printf("Value: %.4e\n",val);

  free_grid(grid);
  return 0;
}
```

`numint.h`

```c
#ifndef NUMINT_H

#define NUMINT_H
#include <stdlib.h>
#include <math.h>

// Function prototypes
double ** create_grid(size_t M, size_t N,
    double xl, double yl, double xu, double yu);
void free_grid(double **grid);
double line_int(size_t M, size_t N, double **grid,
    double xl, double yl, double xu, double yu,
    double x1, double y1, double x2, double y2, size_t n);

#endif
```

`numint.c`

```c
#include "numint.h"

/*
  create_grid

  Purpose:
    Allocates 2D array of size MxN with function values at MN grid points.

  Parameters:
    M         number of grid points in vertical direction
    N         number of grid points in horizontal direction
    xl        x-coordinate of lower-left corner of grid
    yl        y-coordinate of lower-left corner of grid
    xu        x-coordinate of upper-right corner of grid
    yu        y-coordinate of upper-right corner of grid

  Return value:
    Returns a pointer to the grid array (NULL if memory allocation fails).
*/
double ** create_grid(size_t M, size_t N, double xl, double yl,
    double xu, double yu) {
```

```c
  // Allocate two-dimensional array of size MxN (see section 8.3 in WSS)
  double **grid = (double **)malloc(M*sizeof(double *));
  if ( grid == NULL ) return NULL;
  grid[0] = (double *)malloc(M*N*sizeof(double));
  if ( grid[0] == NULL ) { free(grid); return NULL; }
  for (size_t i=1; i<M; i++) grid[i] = grid[0] + i*N;

  // Compute grid[i][j]
  double x,y;
  for (size_t i=0;i<M;i++) {
    y = yl + i*(yu-yl)/(M-1);
    for (size_t j=0;j<N;j++) {
      x = xl + j*(xu-xl)/(N-1);
      grid[i][j] = cos(1.0-x*y);
    }
  }
  return grid;
}

/*
  free_grid

  Purpose:
    Free memory allocated by create_grid routine.

  Parameters:
    grid      pointer to array
*/
void free_grid(double **grid) {
  free(grid[0]);
  free(grid);
  grid = NULL;
  return;
}

/*
  line_int

  Purpose:
    Approximates line integral over line segment using bilinear interpolation.

  Parameters:
    M         number of grid points in vertical direction
    N         number of grid points in horizontal direction
```

```
    grid      pointer to 2D array with function values at grid points
    xl        x-coordinate of lower-left corner of grid
    yl        y-coordinate of lower-left corner of grid
    xu        x-coordinate of upper-right corner of grid
    yu        y-coordinate of upper-right corner of grid
    x1        x-coordinate of start of line segment
    y1        y-coordinate of start of line segment
    x2        x-coordinate of end of line segment
    y2        y-coordinate of end of line segment
    n         number of subintervals used for numerical integration

  Return value:
    Double precision floating point number with approximation
*/
double line_int( size_t M, size_t N, double **grid,
    double xl, double yl, double xu, double yu,
    double x1, double y1, double x2, double y2, size_t n) {

  double x,y;             // a point (x,y)
  double val=0.0;         // result
  double dist=0.0;        // length of line segment

  // Compute distance between the two points
  dist = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));

  // Apply repeated trapezoidal rule
  int ii,jj;
  double idbl,jdbl,c,s;
  double x_ll,y_ll,x_ur,y_ur;  // lower-left and upper-right grid point
  for (size_t i=0;i<=n;i++) {

    // Point on line segment
    x = x1+(x2-x1)*i/n;
    y = y1+(y2-y1)*i/n;

    // Compute index of grid point left and below (x,y)
    jdbl = floor((x-xl)/(xu-xl)*(N-1));
    idbl = floor((y-yl)/(yu-yl)*(M-1));
    if (idbl<0 || idbl>M-1 || jdbl<0 || jdbl>N-1) {
      // Point is outside grid
      continue;
    }
    else {
      ii = (int)idbl;
      jj = (int)jdbl;
```

```
    }

    // Grid point to the left and below (x,y) ["lower-left"]
    x_ll = xl+(xu-xl)*jj/(N-1);
    y_ll = yl+(xu-xl)*ii/(M-1);

    // Grid point to the right and above (x,y) ["upper-right"]
    x_ur = x_ll + (xu-xl)/(N-1);
    y_ur = y_ll + (yu-yl)/(M-1);

    // Compute scalar constant
    s = 1.0/((x_ur-x_ll)*(y_ur-y_ll));
    if (i>0 && i<n) c = s;  // scale by s
    else c = 0.5*s;         // scale by 0.5*s if i = 0 or i = n

    // Approximate f(x,y) using bilinear interpolation
    val += c*grid[ii][jj]*(x_ur-x)*(y_ur-y);
    if (jj<N-1) val += c*grid[ii][jj+1]*(x-x_ll)*(y_ur-y);
    if (ii<M-1) val += c*grid[ii+1][jj]*(x_ur-x)*(y-y_ll);
    if (jj<N-1 && ii<M-1) val += c*grid[ii+1][jj+1]*(x-x_ll)*(y-y_ll);
  }
  val *= dist/n;
  return val;
}
```

`Makefile`

```
CC=cc
SRCS=main.c numint.c
EXECUTABLE=numint
CFLAGS=-g -Wall -Wextra --std=c99
LFLAGS=
LIBS=-lm
INCLUDES=
OBJS=$(SRCS:.c=.o)

all: $(SRCS) $(EXECUTABLE)

$(EXECUTABLE): $(OBJS)
	$(CC) $(OBJS) $(LFLAGS) $(LIBS) -o $@

%.o: %.c
	$(CC) -c $(CFLAGS) $(INCLUDES) $< -o $@

clean:
	rm *.o $(EXECUTABLE)

run: $(EXECUTABLE)
	./$<
```