

# Mathematical Software Programming (02635)

Module 5 — Fall 2016

Instructor: Martin S. Andersen

# Assignment 1

- ▶ Posted on CampusNet
- ▶ Due on October 26, 2016 (hand-in through CampusNet)
- ▶ Please post your questions on [Piazza](#)

# Checklist — what you should know by now

- ▶ How to write a simple program in C (`int main(void) {}`)
- ▶ Basic data types (`int`, `long`, `float`, `double`, ...)
- ▶ Basic input/output (`printf`, `scanf`)
- ▶ Implicit/explicit typecasting
- ▶ How to compile and run a program from terminal / command prompt
- ▶ Control structures and loops (`if`, `else if`, `switch`, `for`, `do`, `while`)
- ▶ Pitfalls with integer and floating point arithmetic
- ▶ Arrays and multidimensional arrays
- ▶ Pointers: “*dereferencing*” and “*address of*” operators
- ▶ Use of functions to structure programs
- ▶ Dynamic memory allocation (`malloc`, `calloc`, `realloc`, `free`)
- ▶ Basic error checking (check return values, etc.)

# This week

## Topics

- ▶ Data structures

## Learning objectives

- ▶ Describe and use **data structures** such as arrays, linked lists, stacks, and queues.
- ▶ Choose appropriate data types and **data structures** for a given problem.
- ▶ Design, implement, and document a program that solves a mathematical problem.

# Structures in C

A struct is a type declaration that *groups* a set of variables

## Example: a matrix

A matrix can be represented as a two-dimensional array and its dimensions

```
struct matrix {  
    size_t m;    /* number of rows          */  
    size_t n;    /* number of columns        */  
    double **A;  /* pointer to two-dim. array */  
} mat1, mat2;
```

- ▶ `mat1` and `mat2` are variables of type `struct matrix`
- ▶ `m`, `n`, and `A` are so-called *members* of the struct
- ▶ period (`.`) is the *member access operator* (e.g., `mat1.n = 5;`)

## Example: declaring a struct

```
struct matrix {  
    size_t m;      /* number of rows           */  
    size_t n;      /* number of columns        */  
    double **A;    /* pointer to two-dim. array */  
};  
  
int main(void) {  
    /* Automatic allocation of "struct matrix" */  
    struct matrix mat;  
  
    mat.A = malloc_array2d(m,n);  
    mat.m = 5;      /* set number of rows       */  
    mat.n = 4;      /* set number of columns    */  
    ...  
  
    return 0;  
}
```

## Example: pointer to struct

- ▶ Pointer to a struct is useful as function input/output
- ▶ Use `p->member` to access a member of a struct via a pointer `p`

```
struct matrix * malloc_matrix(size_t m, size_t n);
void free_matrix(struct matrix * A);

int main(void) {
    struct matrix * A = malloc_matrix(4,5);
    if ( A == NULL ) return EXIT_FAILURE;

    printf("m = %zu\n",A->m);  /* equivalently, use (*A).m */
    printf("n = %zu\n",A->n);  /* equivalently, use (*A).n */
    ...

    free_matrix(A);
    return EXIT_SUCCESS;
}
```

# Example: Array of structs

## Automatic allocation

```
struct point {  
    double x;  
    double y;  
};  
struct point pts[10];  
pts[0].x = 2;  
pts[0].y = 3;
```

## Dynamic allocation

```
struct point *pts;  
pts = malloc(10*sizeof(*pts));  
if ( pts == NULL ) return EXIT_FAILURE;  
pts[0].x = 2;  
pts[0].y = 3;
```



# Type definitions

Assign alternative names to existing types

```
typedef <type> <new type>;
```

## Examples

```
typedef unsigned long size_t;
```

```
typedef struct matrix {  
    size_t m;  
    size_t n;  
    double **A;  
} matrix_t;
```

```
/* Allocate new matrix_t */  
matrix_t *A = malloc(sizeof(*A));
```

# Abstract data types

## List

A *list* is an ordered set of elements with the following properties

- ▶ an element can be accessed, inserted, or deleted at any position
- ▶ a list can be split into sublist
- ▶ two lists can be concatenated

## Stack

A *stack* is a special list where elements are inserted and deleted at one end only (*first-in-last-out*, or FILO)

## Queue

A *queue* is a special list where elements are inserted at one end and deleted at the other (*first-in-first-out*, or FIFO)

# Implementing a list

## Array-based implementation

- ▶ cost of finding/accessing an element does not depend on list length  $n$
- ▶ average cost of inserting/deleting an element is proportional to  $n$

## Linked list

- ▶ average cost of finding/accessing an element is proportional to  $n$
- ▶ cost of inserting/deleting an element does not depend on  $n$

```
/* Struct-based implementation of a linked-list element */  
struct element {  
    <type> variable;  
    struct element * pnext;  
};
```

# Week 6 and week 7

Instructor: Bernd Dammann

- ▶ Timing and profiling
- ▶ Memory model
- ▶ Parallel computations

# Midterm evaluation

- ▶ Please complete midterm evaluation on CampusNet
- ▶ Five quick questions — it only takes a few minutes
- ▶ Open from September 29 to October 14