Mathematical Software Programming (02635)

Module 3 — Fall 2016

Instructor: Martin S. Andersen

Practical information

Assignment 1

- will be posted no later than Sept. 30
- due Wednesday Oct. 26
- approximately 10% of final grade

Assignment 2

- will be posted no later than Oct. 28
- due Wednesday Nov. 23
- approximately 10% of final grade

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
- Pitfalls with integer and floating point arithmetic

This week

Topics

- Arrays
- Pointers
- Multidimensional arrays
- Memory

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.

Arrays

Compile-time array allocation

```c
double data[5] = {-1.0,2.0,4.0,1e3,0.1};
```

Run-time array allocation

```c
size_t n = 0;
scanf("%zu",&n);  // Windows: %Iu
double data[n];
```

- also known as *variable-length arrays* (VLA)
- defined in C99, but optional in C11
- we will talk about variable scope and memory next week

Pointers

```c
int val = 1;          // val has type int
int * pval;           // pval has type (int *)

pval = &val;          // make pval point to val
*pval = 2;            // set val = 2 (pval is unchanged)
```

- a pointer stores an address in memory (it “points” to something)
  - declaring a pointer: `<type> * <name>`
- `*` is **dereferencing** operator
  - `*pval` dereferences a pointer `pval`
  - yields **content** of memory pointed to by `pval`
- `&` is **address of** operator
  - `&val` yields address of variable `val`
  - location in memory where `val` is stored
- use format specifier %p to print pointer using `printf`

Example: pointers and arrays

```
/* Declare double array and double pointer */
double data[4] = {1.0}; // double array of length 4
double * pdata;         // pointer to double

/* Initialize pdata with address of 2nd element of array */
pdata = &data[1];       // same as pdata = data+1;

/* Update values of array via pointer */
pdata[0] = 2.0;         // sets data[1] = 2.0
pdata++;                // increments pointer
*pdata = 3.0;           // sets data[2] = 3.0
*(++pdata) = 4.0;       // sets data[3] = 4.0
*(pdata-3) = 0.5;       // sets data[0] = 0.5
```

Why use pointers? Is this code easy to read/understand?

Multidimensional arrays

A two-dimensional example

```
double mat[3][4];     // uninitialized array of size 3-by-4

// Set all elements of mat to 1.0
for (size_t i=0;i<3;i++) {
    for (size_t j=0;j<4;j++) {
        mat[i][j] = 1.0;
    }
}
```

- a two-dimensional array is "an array of arrays"
- an array "behaves" like a pointer in many ways
- `mat[i]` is an array — corresponds to *i*th row
    - `mat[i]` (or `&mat[i][0]`) is a pointer to first element of *i*th row
    - `mat` (or `&mat[0]`) is pointer to array of pointers (`double **`)

Example 1

```
double mat[3][4];     // uninitialized array of size 3-by-4
double * pi;

// Set all elements of mat to 1.0
for (size_t i=0;i<3;i++) {
    pi = mat[i];                // pointer to i'th array
    for (size_t j=0;j<4;j++) {
        pi[j] = 1.0;            // same as mat[i][j] = 1.0
    }
}
```

- `pi[0]` is first element of *i*th array
- `pi[3]` is fourth element of *i*th array
- What happens if we try to access `pi[4]` or `pi[-1]`?

Example 2

```
double mat[3*4];    // uninitialized array of length 12
double * pd;        // pointer to double

// Treat mat as a 3-by-4 matrix with row-wise storage
for (size_t i=0;i<3;i++) {       // loop over rows
    pd = mat+i*4;
    for (size_t j=0;j<4;j++) {  // loop over cols.
        pd[j] = i*4.0 + j;
    }
}
```

Alternatively, loop can be expressed as:

```
for (size_t i=0;i<3;i++) {       // loop over rows
    for (size_t j=0;j<4;j++) {  // loop over cols.
        mat[i*4+j] = i*4.0 + j;
    }
}
```

Example 3

```
double mat[3*4];    // uninitialized array of length 12
double * pd;        // pointer to double

// Treat mat as a 3-by-4 matrix with col.-wise storage
for (size_t j=0;j<4;j++) {       // loop over cols.
    pd = mat+j*3;
    for (size_t i=0;i<3;i++) {  // loop over rows
        pd[i] = j*3.0 + i;
    }
}
```

Alternatively, loop can be expressed as:

```
for (size_t j=0;j<4;j++) {       // loop over cols.
    for (size_t i=0;i<3;i++) {  // loop over rows
        mat[i+j*3] = j*3.0 + i;
    }
}
```

Exercises

Linear interpolation of $f(x_1)$ and $f(x_2)$ ($x_1 \neq x_2$)

$$f(x) \approx \alpha f(x_1) + (1-\alpha) f(x_2)$$

with $x_1 \leq x \leq x_2$ and $\alpha = \frac{x_2 - x}{x_2 - x_1} \in [0, 1]$

Bilinear interpolation in $\mathbb{R}^2$

Linear interpolation of $f(x, y)$ in x direction

$$g(y) = \alpha f(x_1, y) + (1-\alpha) f(x_2, y), \quad \alpha = \frac{x_2 - x}{x_2 - x_1}$$

followed by linear interpolation in y direction

$$f(x, y) \approx \beta g(y_1) + (1-\beta) g(y_2), \quad \beta = \frac{y_2 - y}{y_2 - y_1}.$$