Mathematical Software Programming (02635)

Lecture 3 — September 18, 2025

Instructor: Martin S. Andersen

Fall 2025

This week

Topics

- Loops, arrays, and functions
- Makefiles
- Introduction to HPC system (Bernd Dammann)

Learning objectives

- Evaluate discrete and continuous mathematical expressions
- Describe and use data structures such as **arrays**, linked lists, stacks, and queues
- Choose appropriate data types and data structures for a given problem

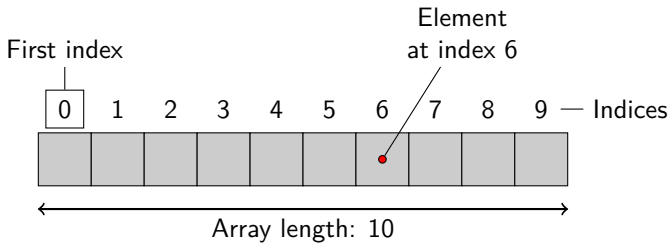
Priorities

- Correctness: design with debugging and testing in mind
- Numerical stability: finite-precision arithmetic, error analysis
- Accuracy: model, discretization, approximation
- Flexibility: abstraction level, modularity
- Efficiency: time and space (memory)

There is often a trade-off between efficiency and stability/accuracy/flexibility.

Arrays

Contiguous (adjacent in memory) collection of *homogeneous* (same type) elements

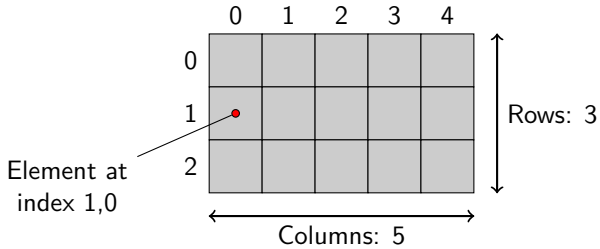


Example

```
int data[10] = {2,5,3,4,8,-1,3,5,4,-3};
for (int i=0;i<10;i++)
    printf("%d\n", data[i]);
```

Two-dimensional arrays

Indexing requires two indices

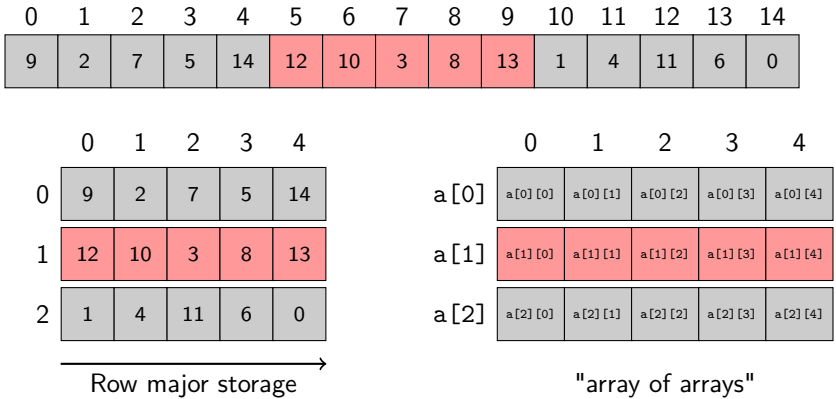


Example

```
double a[3][5] = {{  9,  2,  7,  5, 14},
                  { 12, 10,  3,  8, 13},
                  {  1,  4, 11,  6,  0}};
```

Row-major storage order

In C/C++/Objective-C, multidimensional arrays are stored in row-major storage order.



Column-major storage order

In FORTRAN/MATLAB/Julia, multidimensional arrays are stored in column-major storage order.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
9	12	1	2	10	4	7	3	11	5	8	6	14	13	0

	1	2	3	4	5
1	9	2	7	5	14
2	12	10	3	8	13
3	1	4	11	6	0

Column major storage

FORTRAN/MATLAB/Julia use 1-based indexing: `a(1,1)`, `a(2,1)`, `a(3,1)`, ..., `a(3,5)`

Automatic array allocation/deallocation

Compile-time array allocation

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

Run-time array allocation

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

- also known as *variable-length arrays* (VLA)
- defined in C99, but optional in C11 and later standards
- we will talk about variable scope and dynamic memory allocation next week
- use with caution (add `-Wvla` or `-Werror=vla` to `CFLAGS` in `Makefile` to detect)

Functions

```
return-type function-name ( arguments )
{
  // function "body" with declarations and statements
}
```

- header line `return-type function-name(arguments);` is called a *prototype*
- at most one return value (cannot be an array)
- `void` is used to indicate no return value and/or no arguments
- arguments (aka parameters) are *passed by value*
- variables are *automatic* — scope is code block enclosed between `{ }`

```
double average(double a, double b) {
    double avg = (a+b)/2;
    return avg;
}
```

Array arguments

```c
int sum_array(int a[], int n) {
    int i, sum = 0;
    for (i=0;i<n;i++)
        sum += a[i];
    return sum;
}
```

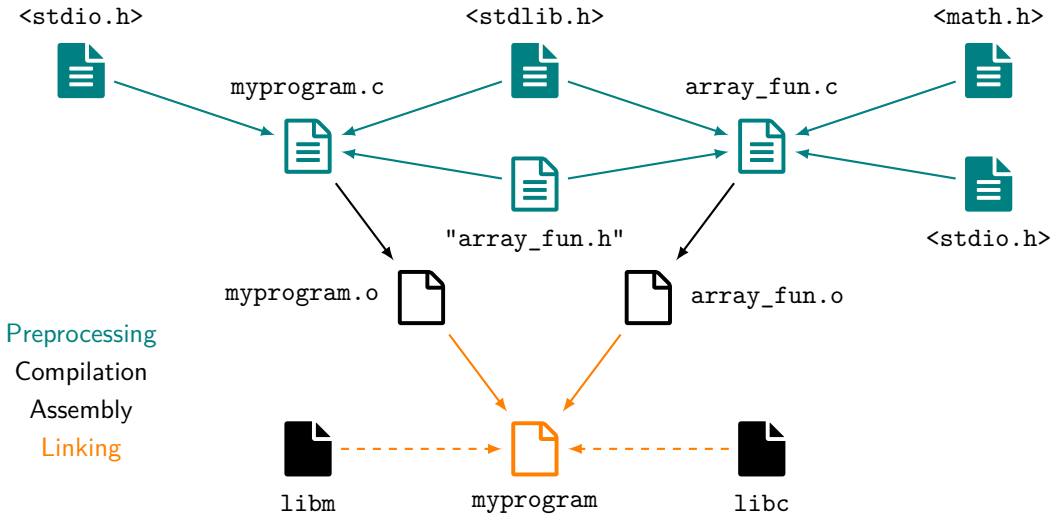
- length of array is normally left unspecified (requires additional parameter)
- undefined behavior if length of the array `a` is less than `n`
- arrays are (essentially) *passed by reference*
- compound literals (C99): `sum_array((int []){3,0,3,4,1}, 5);`

Program termination

```c
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    int n;
    printf("Enter an integer: ");
    if (scanf("%d", &n)!=1)
        return EXIT_FAILURE;
    else
        printf("You entered the number %d\n");
    return EXIT_SUCCESS;
}
```

- `EXIT_SUCCESS` and `EXIT_FAILURE` defined in header `stdlib.h`
- program can be terminated anywhere with `exit(EXIT_SUCCESS)` or `exit(EXIT_FAILURE)`

Multiple-file projects



Source files: array_fun.h and myprogram.c

```c
#ifndef ARRAY_FUN_H
#define ARRAY_FUN_H
double array_sum(const double a[], int n);  // Computes sum
void array_cumsum(double a[], int n);       // Computes cumulative sum
void array_print(const double a[], int n);  // Prints array
#endif
```

```c
#include <stdlib.h>
#include <stdio.h>
#include "array_fun.h"
int main(void) {
  double a[] = {1.0,2.0,3.0,4.0};
  array_cumsum(a,sizeof(a)/sizeof(double));
  array_print(a,sizeof(a)/sizeof(double));
  return (EXIT_SUCCESS);
}
```

Building multiple-file projects

Manual compilation/assembly and linking

```
$ gcc -Wall -std=c11 -c array_fun.c
$ gcc -Wall -std=c11 -c myprogram.c
$ gcc myprogram.o array_fun.o -lm -o myprogram
```

Building with make

- Create a makefile with source and library dependencies

```
$ make myprogram
gcc -Wall -std=c11   -c -o myprogram.o myprogram.c
gcc -Wall -std=c11   -c -o array_fun.o array_fun.c
gcc   myprogram.o array_fun.o  -lm -o myprogram
```

Makefiles revisited

```
variable = value
target : dependencies
    command
```

- Make has many implicit rules (`make -p -f/dev/null`), e.g.,

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
$(COMPILE.c) -o $@ $<
LINK.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
$(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

- implicit dependencies: `target` depends on `target.o`, `target.o` depends on `target.c`
- explicit dependencies
  - object files may depend on header file(s)
  - executable `target` may depend on multiple object files

Makefile for project with two source files

```
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c11
LDFLAGS=
LDLIBS=-lm

myprogram: myprogram.o array_fun.o
myprogram.o: array_fun.h
array_fun.o: array_fun.h

.PHONY: clean           # "clean" does not create file with target name
clean:                  # Removes myprogram and all object files
    -$(RM) myprogram *.o
```

Automatically generate dependencies with GCC/Clang

```
$ gcc -MM *.c
```

Generic makefile for multiple-file projects

```
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c11
LDFLAGS=
LDLIBS=-lm
obj=$(patsubst %.c,%.o,$(wildcard *.c))

myprogram: $(obj)

.PHONY: clean
clean:
    -$(RM) myprogram $(obj)
```

Exercises

- Quizzes: loops, functions, and arrays
- Autolab: function evaluation (avoid over-/underflow and catastrophic cancellation)
  - evaluate Poisson pmf: $f(k;\lambda) = \frac{\lambda^k}{k!}\exp(-\lambda)$, $k \in \mathbb{N}_0$, $\lambda > 0$
  - use `lgamma()`: $\log\Gamma(k+1) = \log(k!)$ for $k \in \mathbb{N}_0$
  - Stirling's formula: $k! = \sqrt{2\pi k}(k/e)^k(1 + O(1/k))$