

- arrays
- arrays and pointers
- dynamically allocated arrays
- arrays as function parameters
- command line parameters
- structures
- coding style

Arrays

- a group of n variables of the same type (e.g. a vector), stored sequentially in memory
- a single name applies to the whole array; individual elements accessed using an index in the range $[0, n - 1]$, e.g.

```
int a[3]; /* declares array of length 3 */
int x = 0;
a[0] = 42;
a[1] = 68;
a[2] = a[0] + 2*a[1]; /* this statement is OK */
a[3] = 1;             /* index is too big! what will happen? */
```

- array names can be used as pointers

- the name *is* actually a pointer to the first element of the array
- since array elements are stored contiguously in memory, subsequent elements can be accessed by doing pointer arithmetic, e.g.

```
int a[3];
*a = 42;      /* same as a[0] = 42; */
*(a+1) = 68; /* same as a[1] = 68; */
```

- address of $(a+1) = a + 1 * \text{sizeof}(\text{int})$

- array names are effectively pointers
- thus a pointer variable set to an address in dynamically allocated memory can be used as an array
 - the `stdlib` library `void *malloc(size_t size)` can be used to allocate `size` contiguous bytes of such memory (n.b. `size_t = unsigned int` on 32-bit systems)
 - e.g. `allocarray.c`

```
int i, n, *A;
scanf("%d", &n);
A = (int *) malloc(n * sizeof(int)); // allocates n elements
for (i=0; i<n; i++)
    A[i] = i; // alt: *(A+i) = i;
```
- should deallocate when finished by `free(A)`; (avoid memory leaks!)
- pitfalls: (1) `A[-1] = 0`; `A[n] = ...`; , (2) `free(A+1)`; , (3) `free(A)`; `free(A)`; , (4) `free(A)`; ... `x = A[i]`;
- Q: what are the advantages over static allocation?

Array as Function Parameters: `uppercase.c`

- while an array parameter (pointer) is passed by value, the memory locations of the array elements are thus effectively passed by reference
- i.e. the function can change the array's elements issue: length of the array?

```
1: #include <stdio.h>
2: #include <string.h>
3:
4: void uppercase(char string []);
5:
6: int main(void) {
7:     char vowels[6];
8:
9:     strcpy(vowels, "aeiou");
10:    printf("%s\n", vowels);
11:
12:    uppercase(vowels);
13:
14:    printf("%s\n", vowels);
15:
16:    return 0;
17: }
18: // alt.: declare as char *string
19: void uppercase(char string []) {
20:     int i=0;
21:
22:     while (string[i] != '\0') {
23:         string[i] = string[i] - 'a' + 'A';
24:         i++;
25:     }
26: }
```

Command Line Parameters

- recall Unix processes are invoked by naming the corresponding executable program and supplying an optional number of string parameters (arguments)
 - e.g. `gcc -Wall -o x x.c`
 - convention: parameters preceded by '-' are called options (following parameters may be associated with this option)
 - a process can thus be thought of as a mathematical function (domain: arrays of strings, range: integer)
- by declaring `int main(int argc, char *argv[])`, can access these through `argv[0]`, `argv[argc-1]` e.g. `printarg.c`

```
#include <stdio.h>
int main(int argc, char *argv[]){
    int i;
    printf("number_of_arguments=%d\n", argc);
    for (i=0; i<argc; i++){
        printf("argument_%i_value:_%s\n", i, argv[i]);
    }
    return 0;
}
```

Structures: Operations

- the `sizeof` operator gives the number of bytes used for a structure:
`printf("size_of_struct_employee=_%d\n", sizeof(person1));`
- structures can be initialised upon declaration using:
`struct employee person1 = {"Jones", 387, 36000.00};;`
- we can define pointers to structures:
`struct employee person1, *pEmployee;`
`pEmployee = &person1;`
- C provides a special operator `->` called arrow operator to access structure members
`pEmployee->name` is same as `(*pEmployee).name;`
- cannot have function members (c.f. Java classes), but can have function pointer members...

User Defined Data Types: Structures

- a structure is a collection of logically related data items of different types grouped together as a single data type

```
struct structure_tag { // structure_tag is name of structure
    variable_declarations; // set of data type declarations making up the struct
};
```

- e.g. consider an employee record:

```
struct employee {
    char name[40]; /* an array member */
    int code; /* an int type member */
    float salary; /* a float type member */
};
```

- to declare one or more variables of this type:

```
struct employee person1, person2, person3;
```

- members (fields) are accessed using the dot operator and component name

```
strcpy(person1.name, "Jones");
person1.code = 387;
person1.salary = 36000.00;
```

- unlike arrays, members can be heterogeneous and are accessed by name

Arrays of Structures and Linked Lists

- we can have arrays of structures, e.g. `struct employee person[30];`
- we can also nest structures:

```
struct FEIT {
    char dean[40];
    struct employee dcsPerson[30];
    struct employee engPersons[30];
};
```

- more interestingly we can build linked lists:

```
struct employee {
    char name[40];
    float salary;
    struct employee *next; /* pointer to next employee */
};
```

