

Files and Arrays

- File access (Afzal Chapter 17)
- Arrays (Afzal Chapters 11-13)
- Arrays and pointers
- Arrays as function parameters
- Structures (Afzal Chapter 15)
- Coding style

File access

- Reading and writing to text files is similar to reading from keyboard and displaying to screen
- Use functions from `stdio` library
 - File pointer variable: `FILE *fp;`
 - Open file: `fp = fopen(name, mode);` with mode either "r" for reading, "w" for writing or "a" for appending
 - Read from file: `fscanf(FILE *fp, char *format, ...);`
 - Write to file: `fprintf(FILE *fp, char *format, ...);`
 - Close file: `fclose(fp);`

File example

```
1:  /* Create and write text file */
2:
3:  #include<stdio.h>
4:
5:  int main(void) {
6:
7:      FILE *fp;
8:      int i = 42;
9:
10:     fp = fopen("myfile.txt","w");
11:
12:     fprintf(fp, "%d albatros!\n", i);
13:
14:     fclose(fp);
15:
16:     return 0;
17: }
```

```
1:  /* Read number from text file */
2:
3:  #include<stdio.h>
4:
5:  int main(void) {
6:
7:      FILE *fp;
8:      int i;
9:
10:     fp = fopen("myfile.txt","r");
11:
12:     fscanf(fp, "%d", &i);
13:
14:     fclose(fp);
15:
16:     printf("%d albatros?\n", i);
17:
18:     return 0;
19: }
```

Arrays

- A group of variables of the same type (e.g. a vector)
- Stored sequentially in memory
- A single name applies to the whole array
- Individual members are accessed by an index number in the range $[0, n - 1]$ for a group of n variables
- Example:

```
int n[3]; /* Variable declaration */  
n[0] = 42;  
n[1] = 68;  
n[2] = n[0] + 2*n[1]; /* This statement is OK */  
n[3] = 1; /* Index number too big! */
```

Arrays and pointers

- Array names can be used as pointers
- The array name is actually a pointer to the first element of an array
- Since the array is stored in consecutive blocks of memory, subsequent elements can be accessed by doing **pointer arithmetic**

- Example:

```
int n[3];
```

```
*n = 42; /* Same as n[0] = 42; */
```

```
*(n+1) = 68; /* Same as n[1] = 68; */
```

- $(n+1) = n + 1 * \text{sizeof}(\text{int})$

Arrays as function parameters

- Array names are effectively pointers
- When passing a pointer as a parameter, it is **passed-by-reference**, i.e. the array name (the pointer to the array in memory) is **passed-by-value**, but having this pointer (or reference) means the function can actually change the values in the original array

Example

```
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:
19: void uppercase(char astring[]) {
20:     int i=0;
21:
22:     while (astring[i] !='\0') {
23:         astring[i] = astring[i]-32;
24:         i=i+1;
25:     }
26: }
```

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
{
    variable_declarations
};
```

- `struct` is a reserved word
- `structure_tag` is the user defined name of the structure type that is subsequently used to define structures
- `variable_declarations` are a set of data type declarations that are placed within braces and make up the structure

Structures Example

- 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 are accessed using the component name and dot operator

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

Structures: Operations

- We can determine the number of bytes used for a structure using the `sizeof` operator

```
printf("Sizeof employee = %d\n",sizeof(person1));
```

- Structures can also be initialised using

```
struct employee person1 = {"Jones",387,36000.00};
```

- We can define pointers to structures

```
struct employee person1, *pointer_to_employee;  
pointer_to_employee = &person1;
```

- C provides a special pointer operator `->` called *arrow operator* to access structure members

```
pointer_to_employee->name same as (* pointer_to_employee).name;
```

Arrays of Structures and Linked Lists

- Just like normal numbers we can have arrays of structures

```
struct employee person[30];
```

- We can also nest structures

```
struct FEIT {  
    char dean[40];  
    struct employee dcs_persons[30];  
    struct employee eng_persons[30];  
};
```

- More interestingly we can build linked lists

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

Coding style - why?

- If your code is easy to read, it will be easier to work with (debug or extend) later
- Some C code in Unix has been in use for 30 years! Even code in the Linux kernel is now more than 10 years old.
- Some code in Windows dates back to 1981 (MS-DOS)

Coding style - suggestions for C

- `/*` Use explanatory comments `*/`
- Recursively indent code within code-blocks (editors like *Emacs* can help with this)
- Break code into related chunks (logical blocks) separated by blank lines – white space costs nothing, so use it
- Use meaningful variable names (like `i`, `j` often used for loops)

Equivalent implementations

```
1: void uppercase(char string[]) {
2:
3:     /* Changes the characters in
4:        string[] from lower to upper
5:        case. Assumes all the
6:        characters are originally lower
7:        case letters. */
8:     int i=0;
9:
10:    while(string[i] != '\0') {
11:
12:        assert((string[i] >= 'a') &&
13:              (string[i] <= 'z'));
14:        string[i] = string[i]-32;
15:        i=i+1;
16:    }
17: }
```

```
1: void uppercase(char *q) {
2:     int i; for(i=0;*(q+i);*(q+i++)-=' ');
3: }
```