

C Notes for Professionals

Chapter 10: Arrays

Arrays are different data types, representing an ordered collection of values (variables) of the same type. They’re used to store multiple elements of any one type, and to represent static or constant memory allocated for an existing C library or module. Other static arrays are strings, and also arrays of pointers.

Using arrays is a widely used way to determine what’s been done. C99 and later support array types in C.

Section 10.1: Declaring and initializing an array

The general syntax for declaring a one-dimensional array is:

```
type identifier [size];
```

where `type` is a valid C type or user-defined C99 basic structures, and `identifier` is a valid C identifier.

Declaring an array can involve “`static`” variables. In this case, it’s `static` like:

```
static int myArr[10];
```

If you want to initialize values, to ensure it holds constant values, one declares:

```
int myArr[10] = {0};
```

Integers may have initializers that evaluate themselves in terms of “`static`” values 1, 2, or 3, all others will be zero.

```
int myArr[10] = {0, 1, 2, 3, 0, 0, 0, 0, 0, 0};
```

In the above method of declaration, the first value is the first to be initialized, and will be copied in the second element of the array, and so on. If there are more than 10 in the above example, the remaining members of the array will be zero. Initialization of 100 will result in each member of the array containing 100.

```
int myArr[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
```

In most cases, the compiler will deduce the length of the array for you. If it does not:

```
int myArr[] = {1, 2, 3, 4, 5}; // error: size of myArr is unknown
```

Don’t worry, an array of length is not needed.

Variables length array (%s) are standardly used in C99, and made available in C11. These are used to store strings in the memory, in the form of a C99 pointer, and not pointers. Only pointers to char can have static storage class.

C99, by the way.

Chapter 12: Enumerations

Section 12.1: Simple Enumeration

It’s an enumeration to use different data types consisting of legal constants and including a character giving a name. `MyEnumeration` is used to declare an enumerated data type.

If you use `enum` then the code will be longer, because you cannot combine everything and avoid separate padding in itself as constants, and you also can’t make it as legal as `switch`.

Example

```
enum MyEnumeration {
    RED,
    GREEN,
    BLUE,
    ORANGE
};
```

With function definitions

```
enum MyEnumeration {
    RED,
    GREEN,
    BLUE,
    ORANGE
};

void printColor(enum MyEnumeration color)
{
    switch (color) {
        case RED: printf("Red\n"); break;
        case GREEN: printf("Green\n"); break;
        case BLUE: printf("Blue\n"); break;
        case ORANGE: printf("Orange\n"); break;
    }
}
```

Example 2

```
(No or simple) is designated initialization which are standardized since C99.

enum MyEnum { ONE, TWO, THREE, FOUR, FIVE, SIX };

MyEnum myVar = ONE;
myVar = TWO; // error: assignment of incompatible type
myVar = ONE; // error: assignment of incompatible type
myVar = ONE; // error: assignment of incompatible type
myVar = ONE; // error: assignment of incompatible type
myVar = ONE; // error: assignment of incompatible type
```

Macro-like functions

```
#define myVar ONE
```

Variables length array (%s) are standardly used in C99, and made available in C11. These are used to store strings in the memory, in the form of a C99 pointer, and not pointers. Only pointers to char can have static storage class.

C99, by the way.

Chapter 20: Files and I/O streams

Section 20.1: Open and write to file

Parameter: `char *filename`: Using `freopen` changing mode of the file and opening it with `FILE *fp`. **Details**: Can be useful to not open the previous file if the file is still open. As for `freopen`, it’s better to use `FILE *fp` to open the new file.

```
FILE *fp = fopen("file.txt", "w");
if (fp == NULL) {
    perror("Error opening file");
    exit(1);
}

// write some content to file
fprintf(fp, "%s", "Hello world");

// close the file
fclose(fp);
```

This program opens the file with name given in the argument to main, displaying its content. If no argument is given, the file is not opened. If the file is already open, its contents are discarded and the file is then closed. If the file does not already exist, `freopen` will create it. If the file is not found, `freopen` will return `NULL`. If the file is found, `freopen` will return a non-`NULL` pointer. This means if the file is not found, it returns a non-`NULL` pointer. This pointer can then be used to reference the file until it is closed.

The file `file.txt` contains the string “Hello world”. It has two lines of text. The first line is “Hello world” and the second line is “Hello world”. If the file is not found, `freopen` will return `NULL`. If the file is found, `freopen` will return a non-`NULL` pointer. This pointer can then be used to reference the file until it is closed.

300+ pages
of professional hints and tricks

Contents

About	1
Chapter 1: Getting started with C Language	2
Section 1.1: Hello World	2
Section 1.2: Original "Hello, World!" in K&R C	4
Chapter 2: Comments	6
Section 2.1: Commenting using the preprocessor	6
Section 2.2: /* */ delimited comments	6
Section 2.3: // delimited comments	7
Section 2.4: Possible pitfall due to trigraphs	7
Chapter 3: Data Types	9
Section 3.1: Interpreting Declarations	9
Section 3.2: Fixed Width Integer Types (since C99)	11
Section 3.3: Integer types and constants	11
Section 3.4: Floating Point Constants	12
Section 3.5: String Literals	13
Chapter 4: Operators	14
Section 4.1: Relational Operators	14
Section 4.2: Conditional Operator/Ternary Operator	15
Section 4.3: Bitwise Operators	16
Section 4.4: Short circuit behavior of logical operators	18
Section 4.5: Comma Operator	19
Section 4.6: Arithmetic Operators	19
Section 4.7: Access Operators	22
Section 4.8: sizeof Operator	24
Section 4.9: Cast Operator	24
Section 4.10: Function Call Operator	24
Section 4.11: Increment / Decrement	25
Section 4.12: Assignment Operators	25
Section 4.13: Logical Operators	26
Section 4.14: Pointer Arithmetic	27
Section 4.15: _Alignof	28
Chapter 5: Boolean	30
Section 5.1: Using stdbool.h	30
Section 5.2: Using #define	30
Section 5.3: Using the Intrinsic (built-in) Type _Bool	31
Section 5.4: Integers and pointers in Boolean expressions	31
Section 5.5: Defining a bool type using typedef	32
Chapter 6: Strings	33
Section 6.1: Tokenisation: strtok(), strtok_r() and strtok_s()	33
Section 6.2: String literals	35
Section 6.3: Calculate the Length: strlen()	36
Section 6.4: Basic introduction to strings	37
Section 6.5: Copying strings	37
Section 6.6: Iterating Over the Characters in a String	40
Section 6.7: Creating Arrays of Strings	41
Section 6.8: Convert Strings to Number: atoi(), atof() (dangerous, don't use them)	41
Section 6.9: string formatted data read/write	42

Section 6.10: Find first/last occurrence of a specific character: strchr(), strrchr()	43
Section 6.11: Copy and Concatenation: strcpy(), strcat()	44
Section 6.12: Comparison: strcmp(), strncmp(), strcasecmp(), strncasecmp()	45
Section 6.13: Safely convert Strings to Number: strtod functions	47
Section 6.14: strspn and strcspn	48
Chapter 7: Literals for numbers, characters and strings	50
Section 7.1: Floating point literals	50
Section 7.2: String literals	50
Section 7.3: Character literals	50
Section 7.4: Integer literals	51
Chapter 8: Compound Literals	53
Section 8.1: Definition/Initialisation of Compound Literals	53
Chapter 9: Bit-fields	55
Section 9.1: Bit-fields	55
Section 9.2: Using bit-fields as small integers	56
Section 9.3: Bit-field alignment	56
Section 9.4: Don'ts for bit-fields	57
Section 9.5: When are bit-fields useful?	58
Chapter 10: Arrays	60
Section 10.1: Declaring and initializing an array	60
Section 10.2: Iterating through an array efficiently and row-major order	61
Section 10.3: Array length	62
Section 10.4: Passing multidimensional arrays to a function	63
Section 10.5: Multi-dimensional arrays	64
Section 10.6: Define array and access array element	67
Section 10.7: Clearing array contents (zeroing)	67
Section 10.8: Setting values in arrays	68
Section 10.9: Allocate and zero-initialize an array with user defined size	68
Section 10.10: Iterating through an array using pointers	69
Chapter 11: Linked lists	71
Section 11.1: A doubly linked list	71
Section 11.2: Reversing a linked list	73
Section 11.3: Inserting a node at the nth position	75
Section 11.4: Inserting a node at the beginning of a singly linked list	76
Chapter 12: Enumerations	79
Section 12.1: Simple Enumeration	79
Section 12.2: enumeration constant without typename	80
Section 12.3: Enumeration with duplicate value	80
Section 12.4: Typedef enum	81
Chapter 13: Structs	83
Section 13.1: Flexible Array Members	83
Section 13.2: Typedef Structs	85
Section 13.3: Pointers to structs	86
Section 13.4: Passing structs to functions	88
Section 13.5: Object-based programming using structs	89
Section 13.6: Simple data structures	91
Chapter 14: Standard Math	93
Section 14.1: Power functions - pow(), powf(), powl()	93
Section 14.2: Double precision floating-point remainder: fmod()	94

Section 14.3: Single precision and long double precision floating-point remainder: fmodf(), fmodl()	94
Chapter 15: Iteration Statements/Loops: for, while, do-while	96
Section 15.1: For loop	96
Section 15.2: Loop Unrolling and Duff's Device	96
Section 15.3: While loop	97
Section 15.4: Do-While loop	97
Section 15.5: Structure and flow of control in a for loop	98
Section 15.6: Infinite Loops	99
Chapter 16: Selection Statements	100
Section 16.1: if () Statements	100
Section 16.2: Nested if()...else VS if()..else Ladder	100
Section 16.3: switch () Statements	102
Section 16.4: if () ... else statements and syntax	104
Section 16.5: if()...else Ladder Chaining two or more if () ... else statements	104
Chapter 17: Initialization	105
Section 17.1: Initialization of Variables in C	105
Section 17.2: Using designated initializers	106
Section 17.3: Initializing structures and arrays of structures	108
Chapter 18: Declaration vs Definition	110
Section 18.1: Understanding Declaration and Definition	110
Chapter 19: Command-line arguments	111
Section 19.1: Print the arguments to a program and convert to integer values	111
Section 19.2: Printing the command line arguments	111
Section 19.3: Using GNU getopt tools	112
Chapter 20: Files and I/O streams	115
Section 20.1: Open and write to file	115
Section 20.2: Run process	116
Section 20.3: fprintf	116
Section 20.4: Get lines from a file using getline()	116
Section 20.5: fscanf()	120
Section 20.6: Read lines from a file	121
Section 20.7: Open and write to a binary file	122
Chapter 21: Formatted Input/Output	124
Section 21.1: Conversion Specifiers for printing	124
Section 21.2: The printf() Function	125
Section 21.3: Printing format flags	125
Section 21.4: Printing the Value of a Pointer to an Object	126
Section 21.5: Printing the Difference of the Values of two Pointers to an Object	127
Section 21.6: Length modifiers	128
Chapter 22: Pointers	129
Section 22.1: Introduction	129
Section 22.2: Common errors	131
Section 22.3: Dereferencing a Pointer	134
Section 22.4: Dereferencing a Pointer to a struct	134
Section 22.5: Const Pointers	135
Section 22.6: Function pointers	138
Section 22.7: Polymorphic behaviour with void pointers	139
Section 22.8: Address-of Operator (&)	140
Section 22.9: Initializing Pointers	140

Section 22.10: Pointer to Pointer	141
Section 22.11: void* pointers as arguments and return values to standard functions	141
Section 22.12: Same Asterisk, Different Meanings	142
Chapter 23: Sequence points	144
Section 23.1: Unsequenced expressions	144
Section 23.2: Sequenced expressions	144
Section 23.3: Indeterminately sequenced expressions	145
Chapter 24: Function Pointers	146
Section 24.1: Introduction	146
Section 24.2: Returning Function Pointers from a Function	146
Section 24.3: Best Practices	147
Section 24.4: Assigning a Function Pointer	149
Section 24.5: Mnemonic for writing function pointers	149
Section 24.6: Basics	150
Chapter 25: Function Parameters	152
Section 25.1: Parameters are passed by value	152
Section 25.2: Passing in Arrays to Functions	152
Section 25.3: Order of function parameter execution	153
Section 25.4: Using pointer parameters to return multiple values	153
Section 25.5: Example of function returning struct containing values with error codes	154
Chapter 26: Pass 2D-arrays to functions	156
Section 26.1: Pass a 2D-array to a function	156
Section 26.2: Using flat arrays as 2D arrays	162
Chapter 27: Error handling	163
Section 27.1: errno	163
Section 27.2: strerror	163
Section 27.3: perror	163
Chapter 28: Undefined behavior	165
Section 28.1: Dereferencing a pointer to variable beyond its lifetime	165
Section 28.2: Copying overlapping memory	165
Section 28.3: Signed integer overflow	166
Section 28.4: Use of an uninitialized variable	167
Section 28.5: Data race	168
Section 28.6: Read value of pointer that was freed	169
Section 28.7: Using incorrect format specifier in printf	170
Section 28.8: Modify string literal	170
Section 28.9: Passing a null pointer to printf %s conversion	170
Section 28.10: Modifying any object more than once between two sequence points	171
Section 28.11: Freeing memory twice	172
Section 28.12: Bit shifting using negative counts or beyond the width of the type	172
Section 28.13: Returning from a function that's declared with `__Noreturn` or `noreturn` function specifier	173
Section 28.14: Accessing memory beyond allocated chunk	174
Section 28.15: Modifying a const variable using a pointer	174
Section 28.16: Reading an uninitialized object that is not backed by memory	175
Section 28.17: Addition or subtraction of pointer not properly bounded	175
Section 28.18: Dereferencing a null pointer	175
Section 28.19: Using fflush on an input stream	176
Section 28.20: Inconsistent linkage of identifiers	176
Section 28.21: Missing return statement in value returning function	177

[Click here to download full PDF material](#)