Arrays and Structures

Davood Rafiei

Arrays

- Declared as
  type arrayname[SIZE];
  e.g. *int a[6];*
- Accessed as
  a[0], a[1], ..., a[SIZE-1]
- Memory allocation
  - Sequential (back-to-back)
  - Allows easy address calculation

Element Indexing

a	1000	10	a[0]
a+1	1004	12	a[1]
a+2	1008	1	a[2]
a+3	1012	4	a[3]
a+4	1016	24	a[4]
a+5	1020	55	a[5]

Array Elements and Subscripts

- Array elements are like normal variables

```
a[0] =  3;
```

- The first element in an array **a** is **a[0].**

- Expressions as array subscripts
  - Suppose **x == 3** then **a[5 - 2]**, **a[3]**, and **a[x]** refers to the same element.

Array Size

- The size of an array must be a constant
- Allowed

```
#define SIZE 10
#define DSIZE SIZE*2
const int TSIZE = SIZE*3;

char w[10];
char word[SIZE];
char dword[DSIZE];
Char tword[TSIZE];
```

Initialization of Arrays

- Initializers

```
int n[5] = { 1, 2, 3, 4, 5 };
```

- If not enough initializers, remaining elements become 0.

```
int n[5] = { 0 };
```

  sets all the elements to 0.

- What is the size of n?

```
int n[] = { 1, 2, 3, 4, 5 };
```

Example: Standard Deviation

- Formula

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}}$$

- Algorithm
  - read a sequence of numbers into an array
  - compute the average
  - compute the standard deviation

std-dev.c

```
#include <stdio.h>
#include <math.h>
#define MAXLENGTH 100

int main()  {
  int numbers[MAXLENGTH];     // the input sequence of numbers
  double sum = 0;             // the sum of the input numbers
  double stddev = 0;          // the standard deviation
  double avg;
  int cnt = 0;

  // read the input numbers until the end of file is reached.
  while (cnt<MAXLENGTH && scanf("%d", numbers+cnt)>0) {
    sum += numbers[cnt++];
  }
```

```
  if (cnt==0) {
    fprintf(stderr,"No input numbers\n");
    return 1;
  }
  avg = sum/cnt;
  printf("Sum: %f   avg: %f\n", sum, avg);
  for (int i = 0; i<cnt; i++) {
    stddev += (numbers[i]-avg)*(numbers[i]-avg);
  }
  stddev = sqrt(stddev/cnt);
  printf("The standard deviation is %f\n", stddev);
  return 0;
}
```

```
drafiei@ug20:~/201>gcc -Wall -std=c99 -lm std-dev.c
drafiei@ug20:~/201>./a.out
1
2
3
4
Sum: 10.000000   avg: 2.500000
The standard deviation is 1.118034
```

Major Array Pitfall

- Array indexes always start with zero!
- Zero is ‘first’ number to computer scientists.
- C will ‘let’ you go beyond range
  - Unpredictable results
  - Compiler will **NOT** detect these errors!
- Up to programmer to ‘stay in range’

Major Array Pitfall Example

- Indexes range from 0 to (array_size – 1)
  - Example:
    double temperature[24]; // 24 is array size

- Common mistake:
  temperature[24] = 5;
    - Index 24 is ‘out of range’!
    - No warning, possibly disastrous results

```c
/* print the histogram of character counts */
#include <stdio.h>
#define MAXCHARS 256

int main()
{
  int counts[MAXCHARS]={0};
  int ch;

  // count the characters in the input
  while ((ch=getchar()) != EOF)
    counts[ch]++;

  // print the histogram
  for (ch = 0; ch<MAXCHARS; ch++)
    if (counts[ch] > 0)
      printf("'%c'\t%d\n", (unsigned char) ch, counts[ch]);
  return 0;
}
```

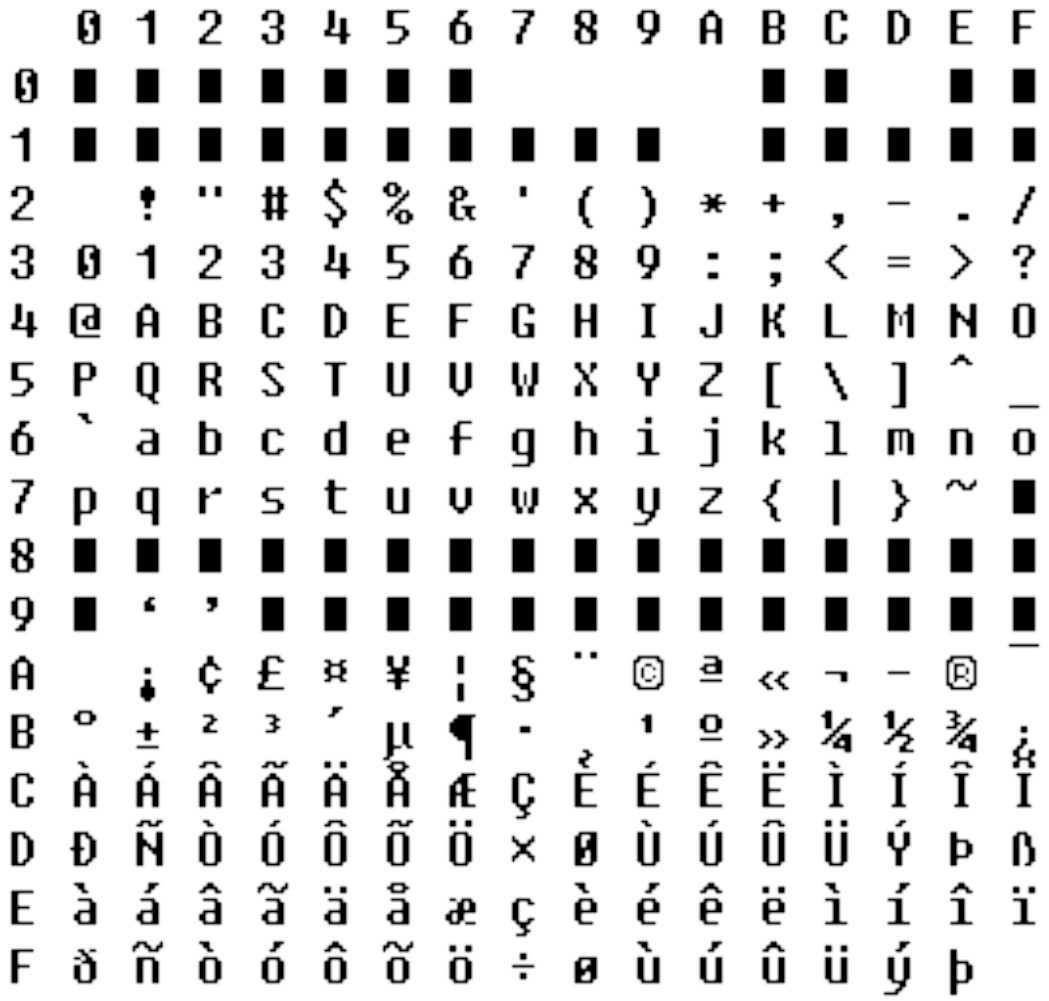
Relationship between char and int

- The char type is really an 8-bit integer.
  - Value range -128 to 127
- Unsigned char
  - Value range 0 to 255
- How to print the ASCII code for ‘D’?

ASCII Table

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	DEL

Extended ASCII Table (Win)



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	■	■	■	■	■	■	■					■	■		■	■
1	■	■	■	■	■	■	■	■	■	■		■	■	■	■	■
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	■
8	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
9	■	‘	’	■	■	■	■	■	■	■	■	■	■	■	■	■
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	

Arrays as Arguments

- A parameter can be an entire array
  - Argument then passed in function call is array name
- Send size of array as well
  - Typically done as second parameter
  - Simple int type formal parameter

Array as Function Arguments

```
#include <stdio.h>
#include <stdlib.h>  // need it for rand()
#define NUMSCORES 5

void fillup(int array[], int length)
{
  for (int i = 0; i<length; i++)
    array[i] = rand()%100;
}

int main()
{
  int score[5];
  fillup(score, NUMSCORES);
  for (int i = 0; i<NUMSCORES; i++)
    pritnf("%d ", score[i]);
  printf("\n");
return 0;
}
```

The const Parameter Modifier

- Protect array contents from modification
  - Use ‘const’ modifier before array parameter
    - Tells compiler to ‘not allow’ modifications

```
void print(const int array[], int length)
{
  for (int i = 0; i<length; i++)
    printf("%d\n", array[i]);
}
```

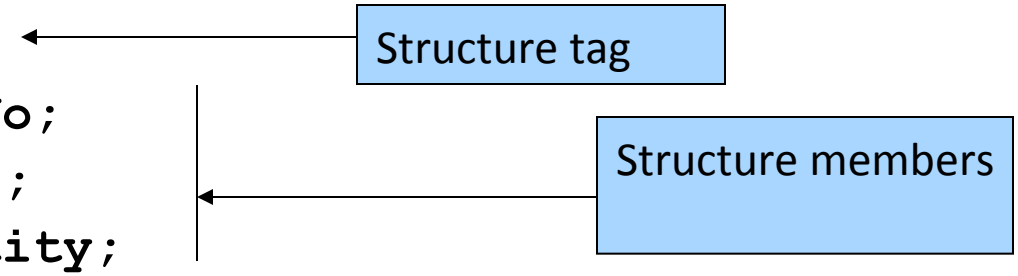
Structures

- A structure is a collection of logically related data items
- Unlike arrays, the elements of a structure are:
  - possibly heterogeneous
  - accessed by name, not by subscript

Structure Example

- Example:

```
struct item {
  int    itemNo;
  double price;
  double quantity;
};
```



- Or

```
typedef struct {
  int    itemNo;
  double price;
  double quantity;
} Item;
```

Creating Instances

- The struct statement defines a data type.
  - a programmer/user defined type (same as system type such as int, double, …).
  - No memory is allocated

- Declare instances of struct

```
struct item ita;
Item itb;
```

Accessing Members of Structures

- Member access operators:
  - Dot operator (**.**) for structures and objects
  - Arrow operator (**->**) for pointers
  - E.g. member **price** of **latestPurchase**:

```
latestPurchase.price;
```

OR

```
itemPtr = &latestPurchase;
itemPtr->price;
```

  - **itemPtr->price** is the same as **(*itemPtr).price**
    - Parentheses is required: ***** has lower precedence than **.**

Structure Initialization and Assignment

- Initialization
  - `Item item1 = {12345, 19.99, 5};`
- Assignment
  - A structure variable may be assigned to another structure variable of the same type.
  - By default, assignment means member-wise copy, although this behavior may be changed.
    - `Item item2;`
    - `item2 = item1;`
    - **`item2`** is a copy of **`item1`**.

Structures within Structures

```
struct Payroll {
  double salary;
  int numPayDays;
};

struct Employee {
  struct Payroll compensation;
  char firstName[10];
  char lastName[10];
  int age;
};

struct Employee record = {{34567, 200},"Bob","Smith",29};
```

Self-referential Structures

- A structure cannot have a data member of the same type as the structure.

```
struct NoGood {
  struct NoGood member;
};
```

- Contains a member that is a pointer to the same structure type
  - Used for linked lists, queues, stacks and trees

```
struct Link {
  struct Link* pre;
  struct Link* suc;
};
```

Passing Structure to Functions

- Passing Structures to functions is no different from other data types
  - Call by value:

```
void displaySalary(struct Employee record) {
  printf("%f\n", record.compensation.salary);
}
```

  - Call by reference:

```
void payRaise(struct Employee *record, double rate) {
  record->compensation.salary *= (1+rate);
}
```

```
struct Employee e1, *e2;
*e2 = e1; // BAD
e2 = malloc(sizeof(struct Employee));
*e2 = e1; // OK
payRaise(&e1, .25);
payRaise(e2, .25);
```