CSCI-UA.0201

Computer Systems Organization

C Programming – Pointers, Structs, Arrays

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies

Pointers:

Very powerful but also dangerous concept!

Can a function modify its arguments?

What if we wanted to implement a function pow_assign() that *modified* its argument, so that these are equivalent:

```
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

⟺

```
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* p is 32.0 here */
```

Would this work?

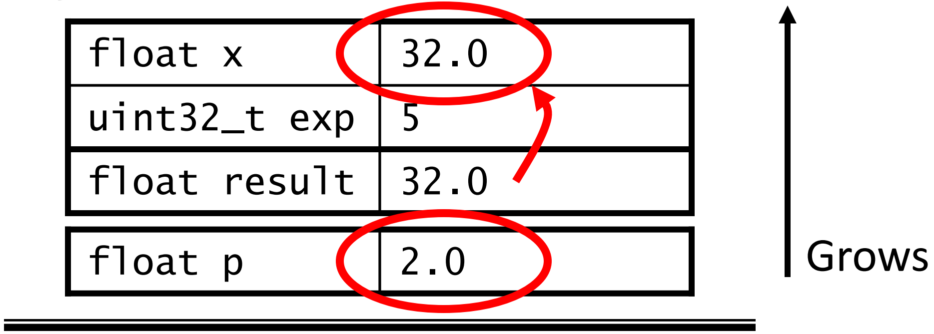
```
void pow_assign(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  x = result;
}
```

NO!

Remember the stack!

```
void pow_assign(float x, unsigned int exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  x = result;
}

main()
{
  float p=2.0;
  pow_assign(p, 5);
}
```



float x	32.0
uint32_t exp	5
float result	32.0
float p	2.0

In C, all arguments are passed *by value*

But, what if the argument is the *address* of a variable?

Passing Addresses

Symbol	Addr	Value
	0	
	1	
	2	
	3	
char x	4	‘H’ (72)
char y	5	‘e’ (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

address of x: 4

memory content at address 4: 72

“Pointers”

This is exactly how “pointers” work.

A “pointer type”: pointer to char

- **address of `x`: &x**
- **if y is an address, the content of the memory at that address: *y**

```
void f(char * p)
{
  *p = *p - 32;
}

char y = 101;       /* y is 101 */
f(&y);              /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

Pointers are used in C for many other purposes:

- Passing large objects without copying them
- Accessing dynamically allocated memory
- Passing functions to other functions
- Implement functions with multiple return values

Pointer Validity

A valid pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior, and will often cause your OS to kill your process (SEGV or Segmentation Fault).

There are two general causes for these errors:
- Program errors that set the pointer value to an invalid address
- Use of a pointer that was at one time valid, but later became invalid

Will **`ptr`** be valid or invalid?

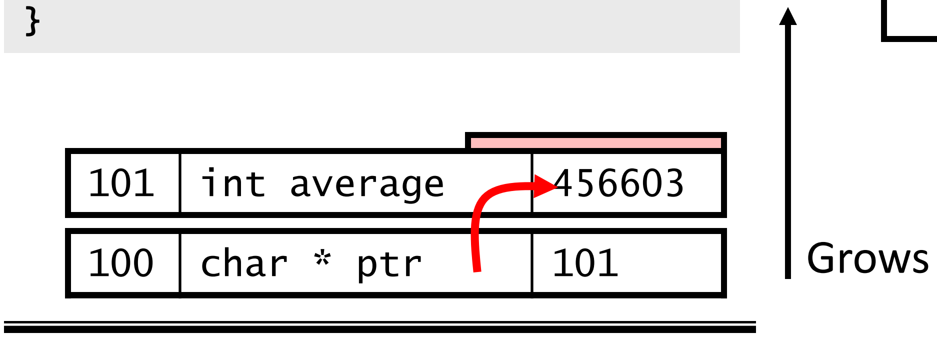
```
char * get_pointer() {
  char x=0;
  return &x;
}

void foo() {
  char * ptr = get_pointer();
  *ptr = 12;  /* valid? */
}
```

Answer: Invalid!

A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```
char * get_pointer()
{
  char x=0;
  return &x;
}
void foo()
{
  char * ptr = get_pointer();
  *ptr = 12;  /* valid? */
  other_function();
}
```



101	`int average`	456603
100	`char * ptr`	101

Grows

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!

Now that we know about pointers, let’s go back to types.

More on Types

We’ve seen a few types at this point: char, int, float, char *

Types are important because:

- They allow your program to impose logical structure on memory
- They help the compiler tell when you’re making a mistake

In the next slides we will discuss:

- How to create logical layouts of different types (structs)
- How to use arrays
- How to parse C type names (there is a logic to it!)
- How to create new types using typedef

Structures

- a collection of related data items
- possibly of different types
- defined using the keyword struct
- The members of a struct type variable are accessed with the dot (.) operator:

  `<struct-variable>.<member_name>`

struct basics

- Definition of a structure:

```
struct <struct-name> {
    <type> <identifier_list>;
    <type> <identifier_list>;
    ...
} ;
```

Each identifier defines a member of the structure.

struct basics

- Example:

```
struct Address {
    int zip;
    char street[50];
    char city[20];
} ;
```

Example

```
main()
{
  struct Address  addrs;
   …
  addrs.zip = 10012;
}
```

Example of initializing a structure

```
struct Address addrs = {10012, “Mercer”, “New York”};
```

Arrays

Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

```
/* define an array of 5 chars */
char x[5] = {‘t’,’e’,’s’,’t’,’\0’};
```

Brackets specify the count of elements. Initial values optionally set in braces.

```
/* accessing element 0 */
x[0] = ‘T’;
```

Arrays in C are 0-indexed (here, 0..4)

```
/* pointer arithmetic to get elt 3 */
char elt3 = *(x+3);  /* x[3] */
```

x[3] == *(x+3) == ‘t’   (*NOT* ‘s’!)

```
/* x[0] evaluates to the first element;
 * x evaluates to the address of the
 * first element, or &(x[0]) */

/* 0-indexed for loop idiom */
#define COUNT 10
char y[COUNT];
int i;
for (i=0; i<COUNT; i++) {
  /* process y[i] */
  printf(“%c\n”, y[i]);
}
```

For loop that iterates from 0 to COUNT-1.
Memorize it!

Symbol	Addr	Value
char x [0]	100	‘t’
char x [1]	101	‘e’
char x [2]	102	‘s’
char x [3]	103	‘t’
char x [4]	104	‘\0’

Pointers and Arrays in C

- An array name by itself is an address, or pointer in C.
- When an array is declared, the compiler allocates sufficient space beginning with some base address to accommodate every element in the array.
- The base address of the array is the address of the first element in the array (index position 0).
  - Example: `int num[10];`

    `&num[0]` is the same as `num`

Pointers and Arrays in C

- Suppose we define the following array and pointer:

  `int a[100];   int* ptr;`

  Assume that the system allocates memory at addresses 400, 404, 408, ..., 796 to the array. `int` values are allocated 32 bits = 4 bytes.

  - The two statements: ptr = a; and ptr = &a[0]; are equivalent and would assign the value of 400 to ptr.

- Pointer arithmetic provides an alternative to array indexing in C.

  - The two statements: ptr = a + 1; and ptr = &a[1]; are equivalent and would assign the value of 404 to ptr.

Pointers and Arrays in C

- Assuming the elements of the array of integers have been assigned values, the following code would sum the elements of the array:

```
int sum = 0;
for (ptr = a; ptr < &a[100]; ++ptr)
  sum += *ptr;
```

- Here is another way to sum the array:

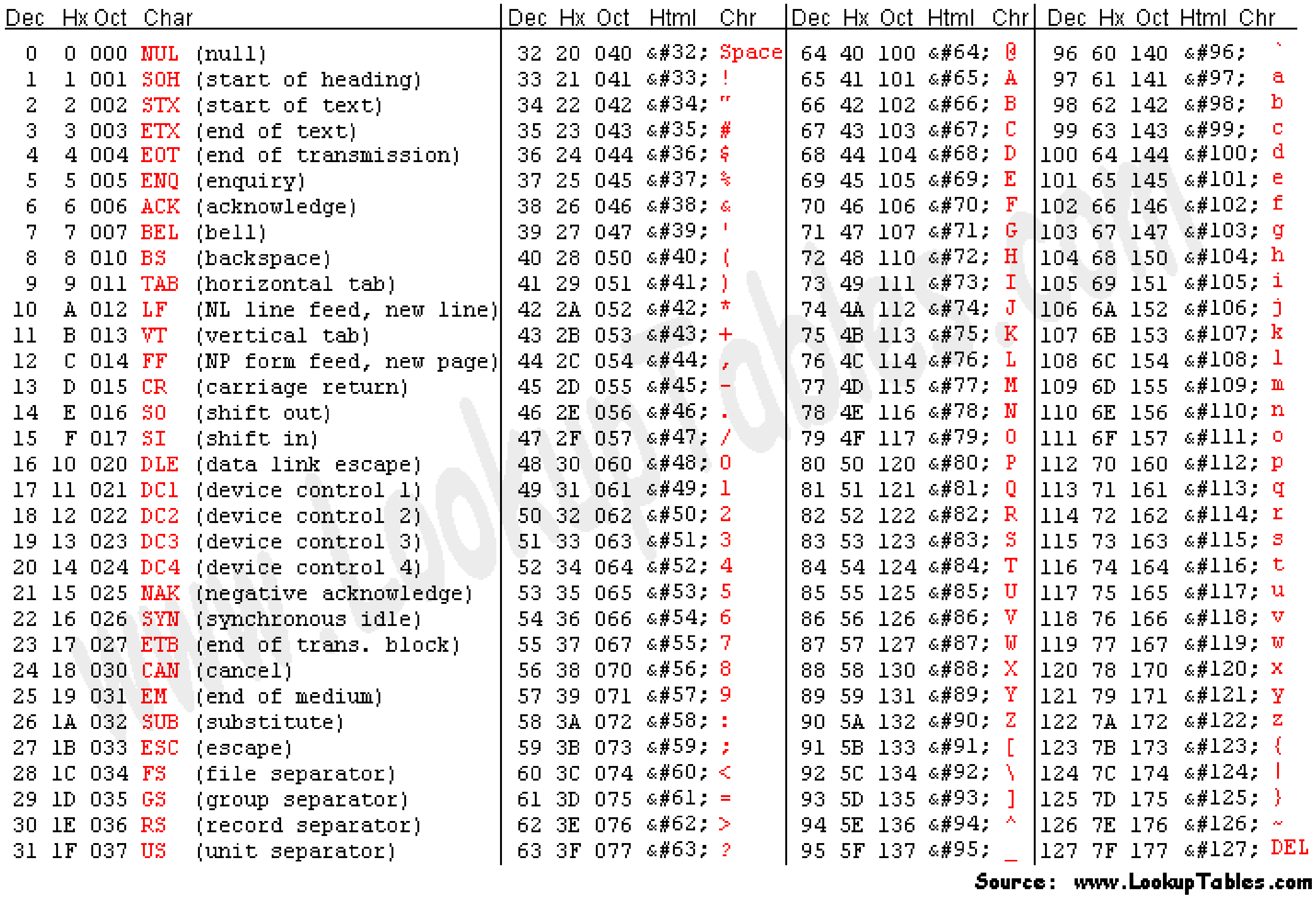
```
int sum = 0;
for (i = 0; i < 100; ++i)
  sum += *(a + i);
```

`a[b]` is just syntactic sugar for `*(a + b)`

Strings

- Series of characters treated as a single unit
- Can include letters, digits, and certain special characters (*, /, $)
- String literal (string constant) - written in double quotes
  - `"Hello"`
- Strings are arrays of characters (type `char[]`)
- String literals are implicitly terminated by a `'\0'`.
- Each character is represented in numerical code called ASCII code.
- Example:
  - `char greeting[] = “Hello”;`
  - size of `greeting` is 6 (length of `“Hello” + 1 for '\0').`
  - address of the above string can be expressed in two ways:
    - `&greeting[0]`
    - `greeting`

ASCII code



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Strings

- String declarations
  - Declare as a character array or a variable of type **`char *`**

    ```
    char color[] = "blue";
    char *colorPtr = "blue";
    ```

  - Remember that strings represented as character arrays end with `'\0'`
    - **`color`** has 5 elements
- Inputting strings
  - Use **`scanf`**

    ```
    scanf("%s", word);
    ```

    - Copies input into **`word[]`**, which does not need **`&`** (because a string is a pointer)
  - Remember to leave space for `'\0'`

Character Handling Library

- In `<ctype.h>`

Prototype	Description
`int isdigit( int c )`	Returns `true` if `c` is a digit and `false` otherwise.
`int isalpha( int c )`	Returns `true` if `c` is a letter and `false` otherwise.
`int isalnum( int c )`	Returns `true` if `c` is a digit or a letter and `false` otherwise.
`int isxdigit( int c )`	Returns `true` if `c` is a hexadecimal digit character and `false` otherwise.
`int islower( int c )`	Returns `true` if `c` is a lowercase letter and `false` otherwise.
`int isupper( int c )`	Returns `true` if `c` is an uppercase letter; `false` otherwise.
`int tolower( int c )`	If `c` is an uppercase letter, `tolower` returns `c` as a lowercase letter. Otherwise, `tolower` returns the argument unchanged.
`int toupper( int c )`	If `c` is a lowercase letter, `toupper` returns `c` as an uppercase letter. Otherwise, `toupper` returns the argument unchanged.
`int isspace( int c )`	Returns `true` if `c` is a white-space character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), or vertical tab (`'\v'`)—and `false` otherwise
`int iscntrl( int c )`	Returns `true` if `c` is a control character and `false` otherwise.
`int ispunct( int c )`	Returns `true` if `c` is a printing character other than a space, a digit, or a letter and `false` otherwise.
`int isprint( int c )`	Returns `true` value if `c` is a printing character including space (`' '`) and `false` otherwise.
`int isgraph( int c )`	Returns `true` if `c` is a printing character other than space (`' '`) and `false` otherwise.

Each function receives a character (an `int`) or `EOF` as an argument

String Conversion Functions

- Conversion functions
  - In `<stdlib.h>` (general utilities library)
  - Convert strings of digits to integer and floating-point values

Prototype	Description
`double atof( const char *nPtr )`	Converts the string `nPtr` to `double`.
`int atoi( const char *nPtr )`	Converts the string `nPtr` to `int`.
`long atol( const char *nPtr )`	Converts the string `nPtr` to long `int`.
`double strtod( const char *nPtr, char **endPtr )`	Converts the string `nPtr` to `double`.
`long strtol( const char *nPtr, char **endPtr, int base )`	Converts the string `nPtr` to `long`.
`unsigned long strtoul( const char *nPtr, char **endPtr, int base )`	Converts the string `nPtr` to `unsigned long`.

String Manipulation Functions

- In **<string.h>**
- String handling library has functions to
  - Manipulate string data
  - Search strings
  - Determine string length

Function prototype	Function description
`char *strcpy( char *s1, const char *s2 )`	Copies string `s2` into array `s1`. The value of `s1` is returned.
`char *strncpy( char *s1, const char *s2, size_t n )`	Copies at most `n` characters of string `s2` into array `s1`. The value of `s1` is returned.
`char *strcat( char *s1, const char *s2 )`	Appends string `s2` to array `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`char *strncat( char *s1, const char *s2, size_t n )`	Appends at most `n` characters of string `s2` to array `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.

String Manipulation Functions

```
int strcmp ( const char * str1,
                   const char * str2 )
```

return value	**indicates**
<0	the first character that does not match has a lower value in *ptr1* than in *ptr2*
0	the contents of both strings are equal
>0	the first character that does not match has a greater value in *ptr1* than in *ptr2*