
UNSW
SYDNEY
Australia's
Global
University
COMP1911 - Computing 1A

Last week

- What is a Character
- ASCII Encoding
- Reading and writing Characters
- What is a String
- Manipulating Strings
- Command-line Arguments
- Linux I/O direction
- Accessing files using C
- ASCII file and binary file

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	“	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	‘	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

‘\0’ , ‘\n’ and ‘ ’

```
printf("%d is %c\n", '\0','\0');
printf("%d is %c\n", '\n','\n');
printf("%d is %c\n", ' ',' ');
```

‘\0’ value is 0

‘\n’ value is 10

‘ ‘ value is 32

fgets(line, MAXLINE, stdin);
COMP1911+enter

C	O	M	P	1	9	1	1	\n	\0

COMP1911+ctrl^D

C	O	M	P	1	9	1	1	\0	

Command-line Arguments

Command-line arguments are 0 or more strings specified when program is run.
For example, if you run this command in a terminal:

```
dcc count.c -o count
```

dcc will be given 3 command-line arguments: **"count.c" "-o“ "count"**
main needs a different prototype if you want to access command-line arguments

```
int main(int argc, char *argv[]) { ...
```

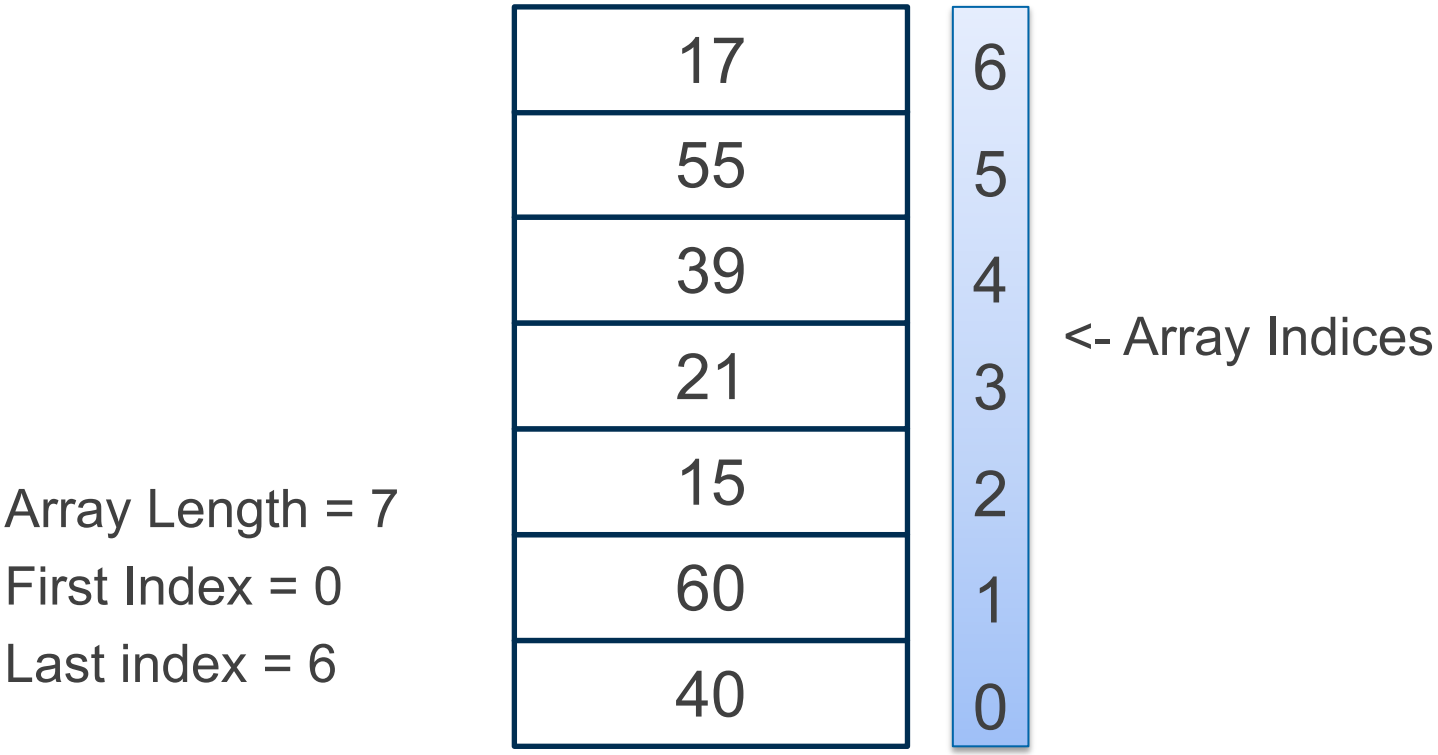
12. Dynamic Memory

In this lecture we will cover:

- Dynamic Memory

Memory allocation for an array

An array is collection of items stored at continuous memory locations.



Size of the array

- The size of the array is 7
- If there is an requirement to change this size:
  - ✓ If only 5 elements are needed, the remaining 2 indices are wasting memory
  - ✓ If all 7 elements are filled, but here is a need to enter 3 more element.

Dynamic Memory Allocation

Current Problems

- Want to create arrays within functions and return them
- Want to create a really really big arrays
- Want to create an array whose size is not know until runtime

Future Problem

- We don't want to have to guess how much memory to use
- Want to use as much memory as needed, no more, no less.
- We want to be able to ask for more as needed and give it back when we have finished
- We will look into ways to do this in the bridging courses and future cse courses

Dynamic Memory Allocation

Solution

- Dynamic memory allocation: Allocating and deallocating memory at runtime as we need it

malloc and free

malloc() or “memory allocation” is used to dynamically allocate a single large block of memory with the specified size.
It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.

Syntax:

```
ptr = (castType*) malloc(size);
```

malloc() doesn't get freed on their own. You must explicitly use **free()** to release the space.

Syntax:

```
free(ptr);
```

malloc and free

For example, let's assume we need a block of memory to hold 100,000,000 ints.

```
int *p;
p = malloc(100000000 * sizeof (int));
if (p == NULL) {
    printf("Error: array could not be allocated.\n");
    exit(EXIT_FAILURE);
}
            // we can now use the pointer
            // ... lots of things to do

free(p);  // free up the memory that was used
```

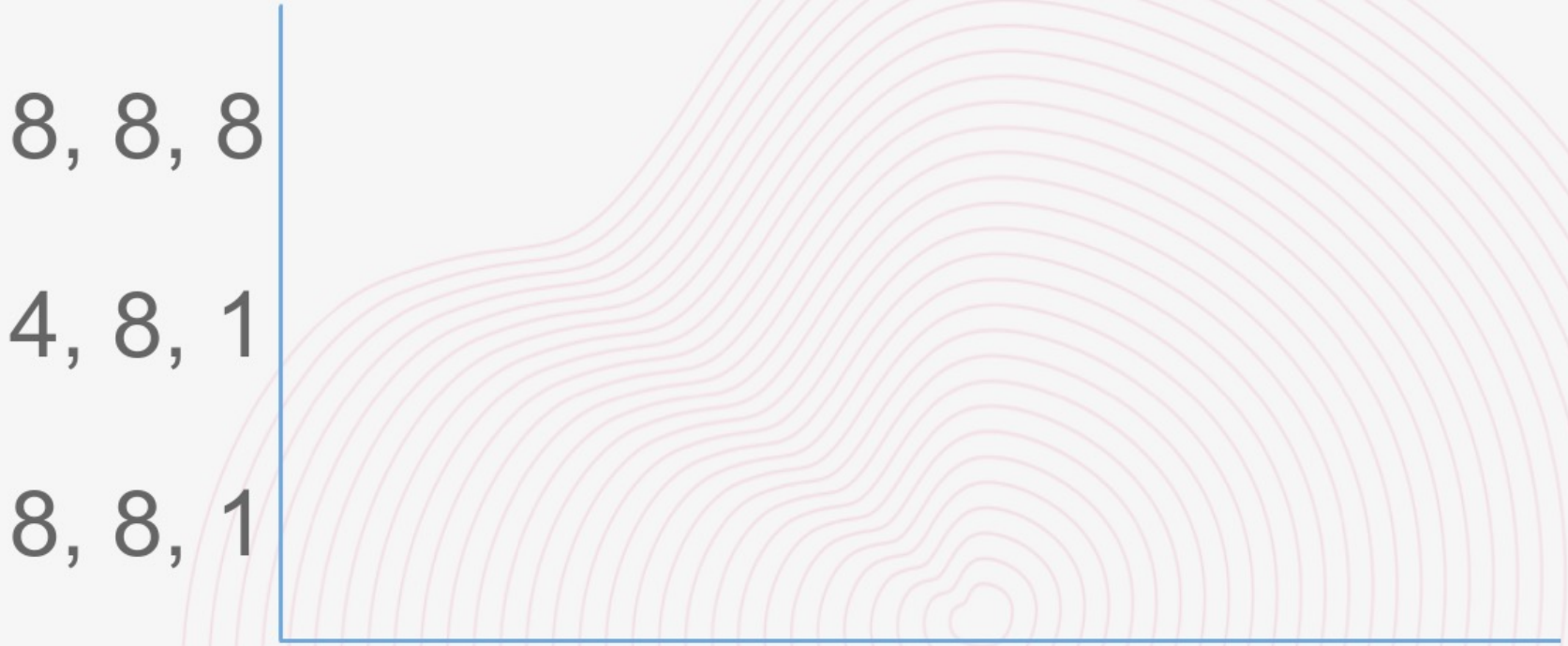
sizeof

- **sizeof** - C operator yields bytes needed for type or variable
- **sizeof (type)** or **sizeof** variable
- note unusual (badly designed) syntax - brackets indicate argument is a type
- use sizeof for every malloc call

```
printf("%d", sizeof (char));
printf("%d", sizeof (int));
printf("%d", sizeof (double));
printf("%d", sizeof (int[10]));
printf("%d", sizeof (int *));

printf("%d", sizeof "hello");
```

sizeof(int *) sizeof(double) sizeof(char *)



Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

sizeof

- **sizeof** - C operator yields bytes needed for type or variable
- **sizeof (type)** or **sizeof** variable
- note unusual (badly designed) syntax - brackets indicate argument is a type
- use sizeof for every malloc call

```
printf("%d", sizeof (char));     // 1
printf("%d", sizeof (int));      // 4 commonly
printf("%d", sizeof (double));   // 8 commonly
printf("%d", sizeof (int[10])); // 40 commonly
printf("%d", sizeof (int *));    // 4 on cse machines
                                  // or often 8
printf("%d", sizeof "hello");    // 6
```

malloc and sizeof

- **sizeof** - C operator yields bytes needed for type or variable
- note unusual syntax **sizeof (type)** or **sizeof variable**
- use sizeof for every malloc call
- malloc() returns pointer to block of memory
- malloc() returns a (void *) pointer - can be assigned to any pointer type
- malloc() returns NULL if insufficient memory available - check for this

free

- free() indicates you've finished using the block of memory
- Continuing to use memory after free() results in very nasty bugs.
- free() memory block twice also cause bad bugs.
- if program keeps calling malloc() without corresponding free() calls program's memory will grow steadily larger called a **memory leak**.
- Memory leaks major issue for long running programs.
- Operating system recovers memory when program exists.
- To find memory leaks using dcc, compile with the following flag

  dcc --leak-check

calloc() and realloc()

calloc() means contiguous allocation.
malloc() allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes all bits to zero.

Syntax:

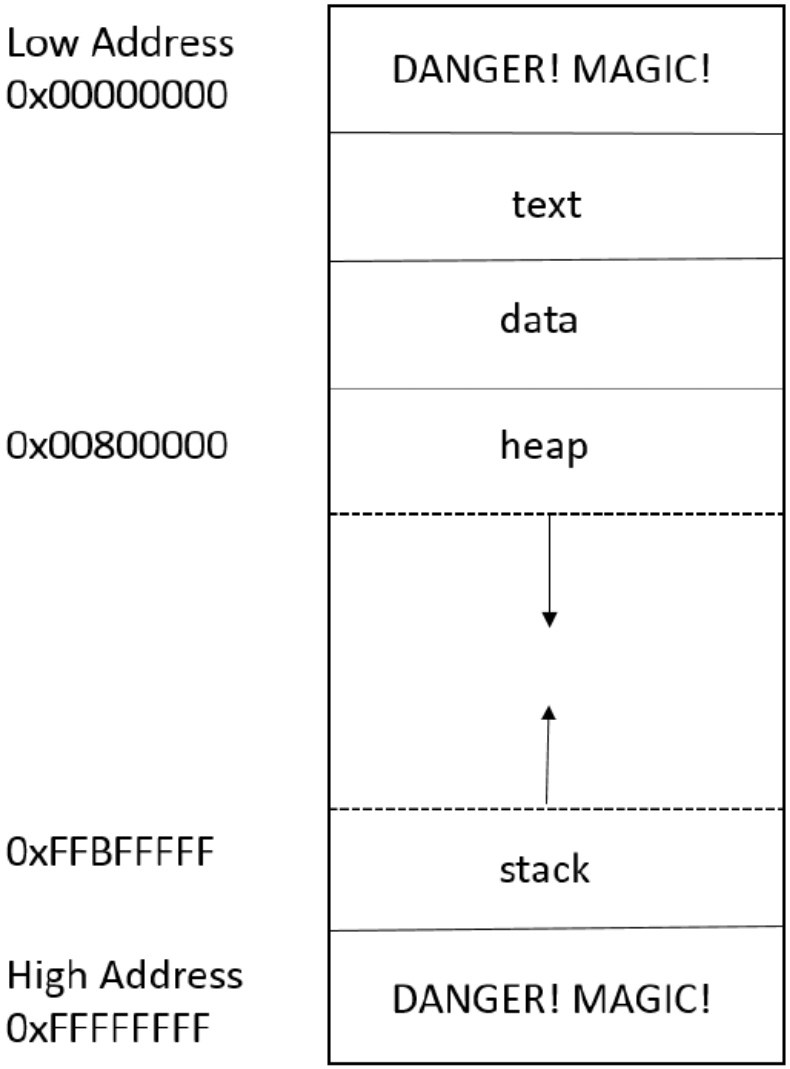
```
ptr = (castType*) calloc(n, size);
```

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the **realloc**() function.

Syntax:

```
ptr = realloc(ptr, x);
```

Memory Map



Segments in Memory:

- text: code executable/binary
- data: global variables, static variables, constant data
- heap: malloced memory
- stack: local variables defined in functions

example

```
int a = 0; //data segment
char *p1; //data segment
main()
{
        int b; //stack
        char s[] = “abc”; //stack
        char *p2; //stack
        char *p3 = “123456”; //123456 at data segment,  p3 at stack
        static int c =0 ;  //data segment
        p1 = (char *)malloc(10); //heap
        p2 = (char *)malloc(20); //heap
        strcpy(p1, "123456"); //123456 at data segment
}
```

Questions