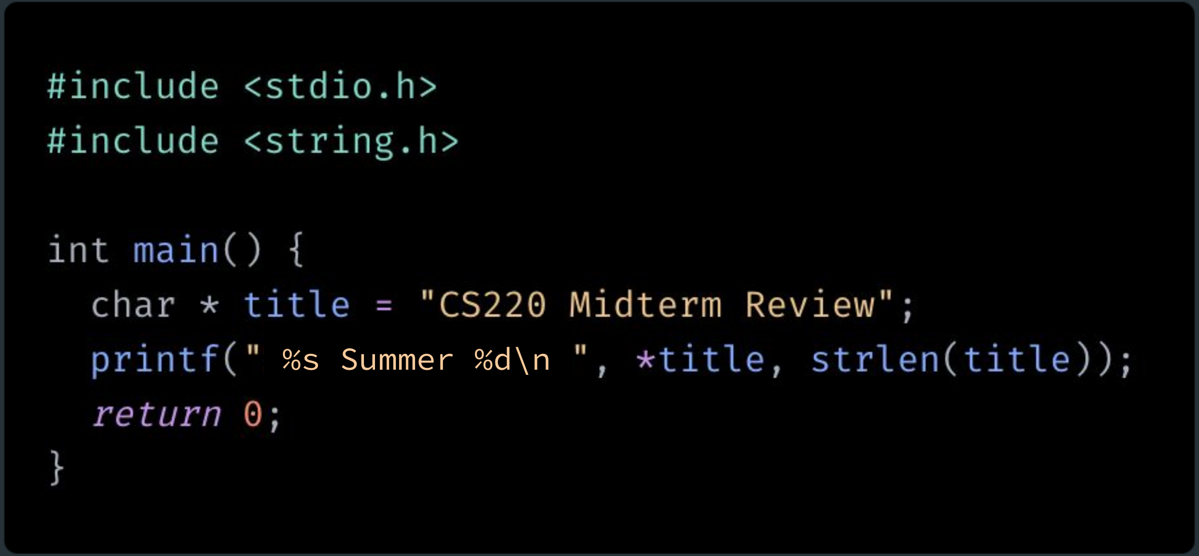


```c
#include <stdio.h>
#include <string.h>

int main() {
  char * title = "CS220 Midterm Review";
  printf(" %s Summer %d\n ", *title, strlen(title));
  return 0;
}
```

What’s wrong with this piece of code?

```c
#include <stdio.h>
#include <string.h>

int main() {
  char * title = "CS220 Midterm Review";
  printf(" %s Summer %d\n ", title, strlen(title));
  return 0;
}
```

Exam Overview

- Midterm on Friday (07/02)
- Held on Gradescope
- Open 7 am to 1 pm EDT
- 2 hours (or until 1 pm, whichever comes first) to finish the exam once you begin
- Technical help session on Zoom during exam window
- More details on Piazza
- 150 points (15%)

- Content: Everything about the C language that you have learned so far

- Format:
    - 5 true/false (7.5 points)
    - 5 multiple choices (15 points)
    - 5 multiple select (22.5 points)
    - 2 code tracing (15 points)
    - 2 code explanation (30 points)
    - 2 code writing questions (60 points)

Exam Preparation Tips

- Resources to use for midterm
  - Midterm practice questions
  - Class materials
  - In class exercises
  - Recap questions discussed in class
  - Review session slides
  - Your notes and code
  - Office hours and Piazza
  - repl.it

Review recommended time allocation on Piazza before exam!

Exam Preparation Tips

- Get *plenty* of sleep.
- Go through all the slides, and **understand** what is happening. It does not help to just *memorize* it.
- Code the problems and try out different scenarios (use repl.it!)
- Practice solving the problems on paper, and make sure you can trace through your own logic
    - If you can't follow your own logic, we probably can't either

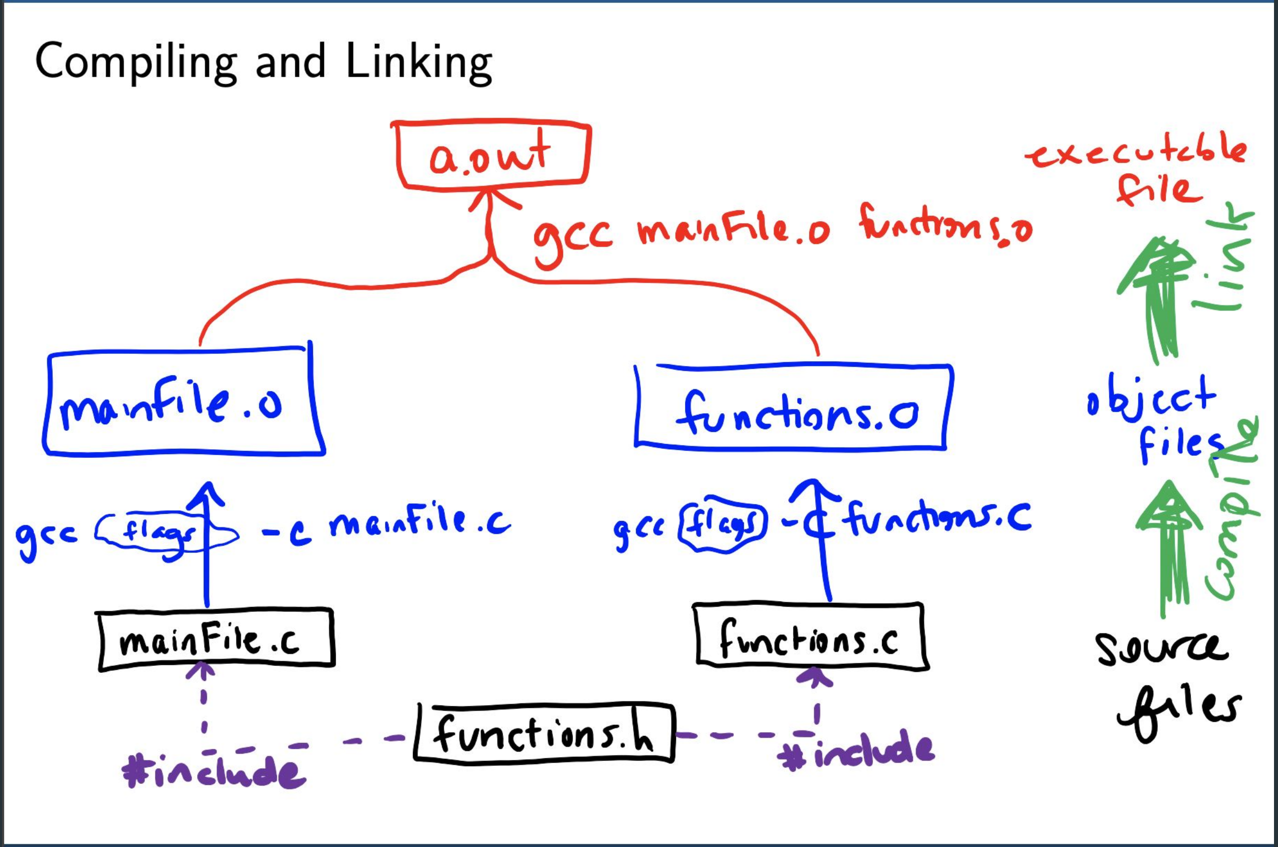
Exam Taking Tips

- Make sure you start the exam before 11 am EDT to have the full 2 hours to work on it!
- **Read the entire problem** and make sure you understand what it's asking for!
  - Many students lose points just by not fully understanding the problem
  - For example, print does not mean return, and vice versa
- Use reference materials wisely

Common Mistakes

- Spend too long on multiple choice questions (some can be time consuming)
- Not checking for syntax errors in code writing
- Not having a system for tracing variables
- Not understanding pointers and passing by value
- Confusion about linked lists
- Data type (sizes) and conversions (narrowing, promotion)

Inside the compiler



linker
- collect object files to create an executable

compiler
- translate human-readable to object code

preprocessor
- process all #include/#define statements

Variable types

char
- 1 byte(8 bits) character; an integer type

int
- on ugrad, uses 4 bytes (32 bits)

unsigned
- same size as int, but >= 0

long
- greater capacity than int

float
- single-precision

double
- double-precision

bool
- #include <stdbool.h>

printf

```
printf("%[flags][width][.precision][length]specifier", ...)
```

Format Specifiers
d – decimal (integer type, ld for long int)
u – unsigned (integer type that disallows negatives, lu for long unsigned)
f – floating point (float, lf for double)
c – character
s – strings

Also **fprintf** (for files) – need to specify file pointer

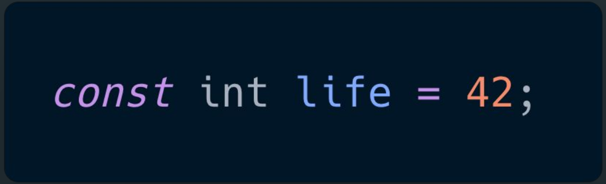
printf is just **fprintf** with the file pointer as **stdout**.

scanf

```
scanf("%d %d", &num1, &num2);
```

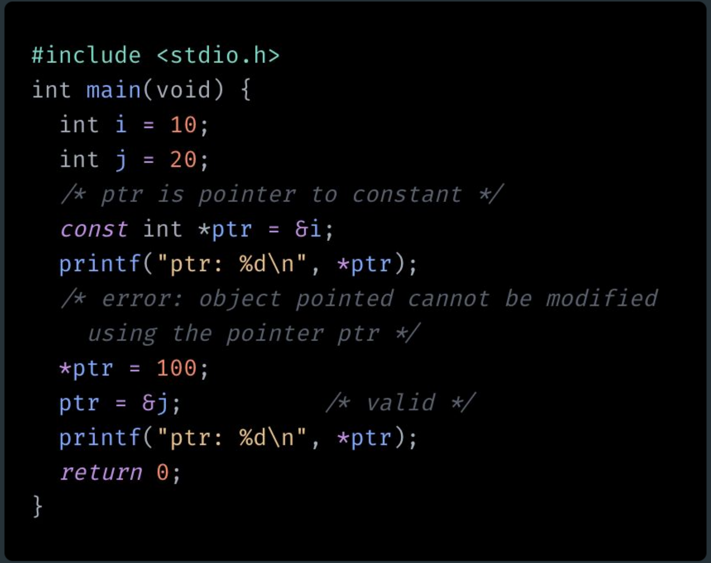
- Same type specifiers (%d, %f)
- Need to put the & for primitive data types (one without a memory access).
- fscanf (for files), sscanf (for strings)

const



```c
const int life = 42;
```

- Can’t adjust after declaration



```c
#include <stdio.h>
int main(void) {
  int i = 10;
  int j = 20;
  /* ptr is pointer to constant */
  const int *ptr = &i;
  printf("ptr: %d\n", *ptr);
  /* error: object pointed cannot be modified
    using the pointer ptr */
  *ptr = 100;
  ptr = &j;            /* valid */
  printf("ptr: %d\n", *ptr);
  return 0;
}
```

- const for pointers are different!
- const int *p -> int being pointed to is constant
- int * const p -> pointer itself is constant

file I/O

- To read to / write from the command line, we use the commands
  - `fprintf( output_file, format_str,…);`
  - `fscanf( input_file , format_str , … );`
- Getting User Input
  - `scanf` and `fscanf`
  - `getc` and `fgetc` // collect a single char at a time
  - `gets` and `fgets`
  - `fread` (reading binary files)
- Error Handling
  - `foef` and `ferror`
- `stdout` and `stdin` are instances of file-handles

What does the scanf function return?

- Returns number of elements it successfully read
- Need for reading until EOF

file handling (opening, accessing)

```
FILE* fopen(const char file_name[],
            const char mode[]);
```

- **Input:** file name, file open mode (a string)
- **Output:** A pointer to a file-handle, returns **NULL** if doesn't exist.

```
#include <stdio.h>

int main( void ) {
  FILE* fp = fopen( "foo.txt" , "w");
  if( !fp ) {
    fprintf( stderr , … );
    return 1;
  }
  fprintf( fp , "hello\n" );
  fclose( fp );
  return 0;
}
```

```
int fprintf(FILE* fp, const char
            format_str[], ... );
```

- Writes a formatted string to the specified file-handle
- Returns the number characters written (a negative value if the write failed)

```
#include <stdio.h>

int main( void ) {
  FILE* fp = fopen( "foo.txt" , "w" );
  if( !fp ) {
    …
  }
  fprintf( fp , "hello\n" );
  fclose( fp );
  return 0;
}
```

file handling (accessing)

Note: EOF is the end-of-file character (not the same as zero)

```
int fputc(int character, FILE * fp);
```

- Writes a single character to the specified file-handle
- **Returns the character (EOF if write failed)**

```c
#include <stdio.h>

int main( void ) {
  char str[] = "hello";
  FILE* fp = fopen( "foo.txt" , "w" );
  if( !fp ) {…}
  for( int i=0 ; str[i] ; i++ ) fputc( str[i] , fp );
  fclose( fp );
  return 0;
}
```

```
int fscanf( FILE* fp , const char
     format_str[] , ... );
```

- Reads a formatted string from the specified file-handle
- Returns the number of variables successfully set

```c
#include <stdio.h>

int main( void ) {
  char word[512];
  FILE* fp = fopen( "foo.txt" , "r" );
  if( !fp ) {…}
  while( fscanf( fp , "%s" , word )==1 )
    printf( "Read: %s\n" , word );
  fclose( fp );
  return 0;
}
```

file handling (accessing)

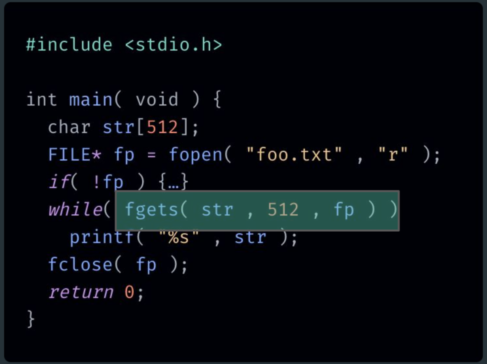
Note: EOF is the end-of-file character (not the same as zero)

```
char* fgets(char str[], int num,
            FILE* fp );
```

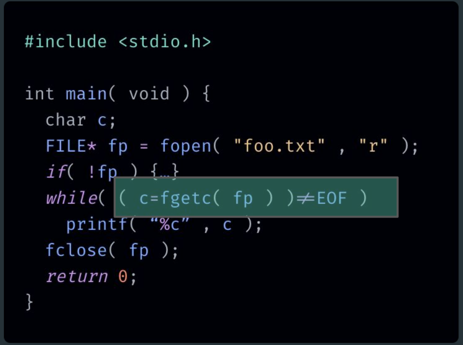
- Reads characters from a file-handle until either the string buffer is filled, a new line is reached, **EOF** is reached
- Returns **str** (**NULL** if the read failed)

```
int fgetc( FILE* fp );
```

- Reads a single character from the file-handle
- Returns the character written (EOF if the read failed)



```
#include <stdio.h>

int main( void ) {
  char str[512];
  FILE* fp = fopen( "foo.txt" , "r" );
  if( !fp ) {…}
  while( fgets( str , 512 , fp ) )
    printf( "%s" , str );
  fclose( fp );
  return 0;
}
```



```
#include <stdio.h>

int main( void ) {
  char c;
  FILE* fp = fopen( "foo.txt" , "r" );
  if( !fp ) {…}
  while( ( c=fgetc( fp ) )≠EOF )
    printf( "%c" , c );
  fclose( fp );
  return 0;
}
```

file handling (binary data accessing)

Note: Requires `fopen("data.dat", "rb");`

```
int fread(where_to, size_of_el,
        num_els, fp);
```

- reads **`size_of_el`** * **`num_els`** bytes of memory
- from the file beginning at the file cursor location **`fp`**, and stores them starting at pointer location **`where_to`**
- returns the number of items successfully written

```
int fwrite(where_from, size_of_el,
        num_els, fp);
```

- Does the opposite, copying data from memory to the specified file
- Returns data read

file handling (closing)

```
int fclose( FILE * fp );
```

- **Input:** The file-handle
- **Output:** Returns 0 if the file was successfully closed (EOF if it wasn't)

```
#include <stdio.h>

int main( void ) {
  char c;
  FILE* fp = fopen( "foo.txt" , "r" );
  if( !fp ) {…}
  while( ( c=fgetc( fp ) )≠EOF )
    printf( “%c” , c );
  fclose( fp );
  return 0;
}
```

file handling (testing) + std output

```
int feof( FILE * fp );
```

- **Input:** The file-handle
- **Output:** Returns non-zero if we have read to the end of the file

```
int ferror( FILE * fp );
```

- **Input:** The file-handle
- **Output:** Returns non-zero if the file is in an error state

- **stdout** and **stderr** are both file-handles that allow writing to the command prompt
  - These are separate file-handles! (e.g. You can redirect them separately)

if else-if else && logical operators

```
int age = 23;

if (age >= 21) {
    prinf("at work\n");
} else if (age >= 18) {
    printf("at college")
} else if (age >= 5) {
    printf("at school");
} else {
    printf("at home");
}
```

&&	AND
||	OR
!	NOT

Recall
DeMorgan's Laws

Control Structures (Short-Circuiting)

- When C evaluates the composition of logical expression . . .

  ```
  if( (statement _1) || (statement_2) )
  while( (statement _1) && (statement_2) )
  ```

  . . . it short circuits as soon as answer is definitely true or definitely false.
- `if( a == 7 || b == 7 ):`
  - When `( a==7 )` is __true__, the entire expression is true so we don't need to test if `(b==7)` is true
- `while( a == 7 && b == 7 ):`
  - When `( a==7 )` is __false__, the entire expression is false so we don't need to test if `(b==7)` is true.

Loops (Summary)

```
while( boolean expression ) { statements }
```

- iterates 0 or more times, as long as boolean expression is true
- execute statements at each iteration

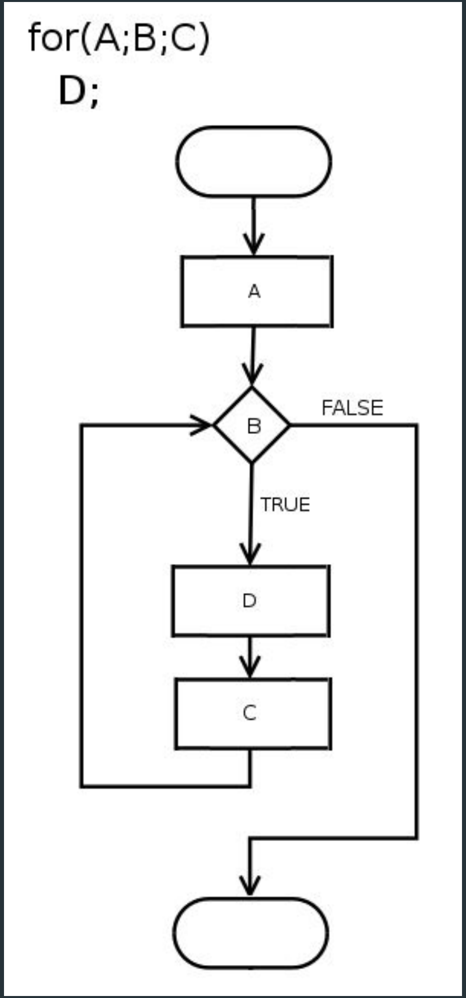
```
do { statements } while ( boolean expression )
```

- iterates 1 or more times, as long as boolean expression is true
- execute statements at each iteration

```
for( init ; boolean expression ; update ) { statements }
```

- init happens first; usually declares & assigns “index variable”
- iterates 0 or more times, as long as boolean expression is true
- execute statements at each iteration
- update is run after statements; often it increments the loop variable (i++)

For Loops (Flow Diagram)

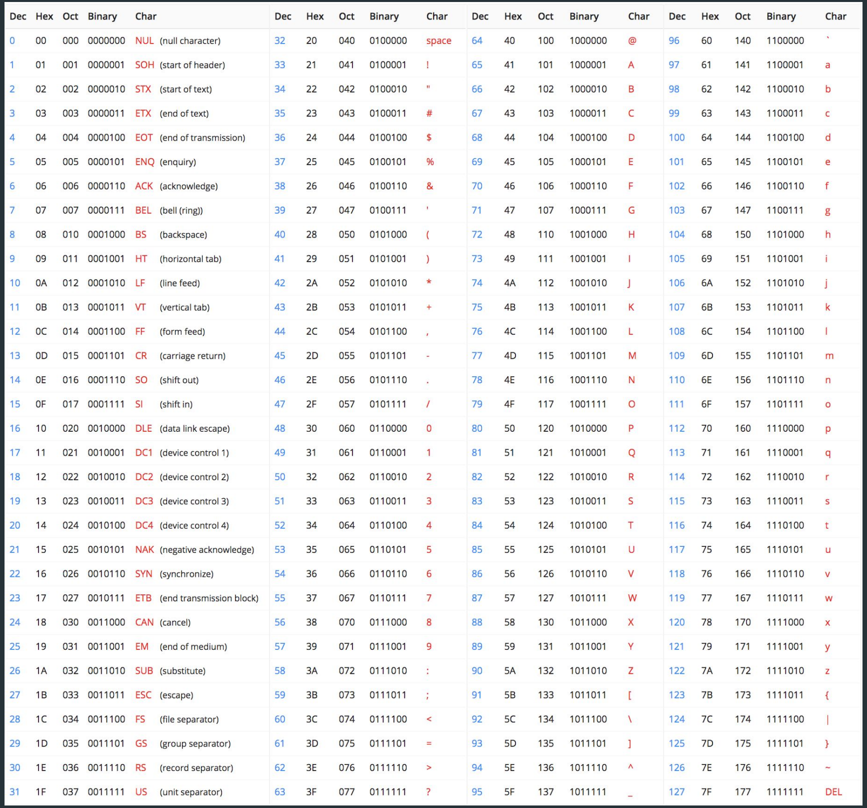


Arrays

- Arrays are laid out consecutively in memory
    - could be on stack or heap; depends on how created
- element access:
    - arr[5]

```
#include <stdio.h>

int main() {
    //declare array of size 15.
    int arr[15];
    // declare array of size 2 with values 1, 2.
    int arr[2] = {1, 2};
    // don't need to specify size either.
    int arr[] = {1, 2};
}
```

ASCII Table



Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char
0	00	000	0000000	NUL (null character)	32	20	040	0100000	space	64	40	100	1000000	@	96	60	140	1100000	`
1	01	001	0000001	SOH (start of header)	33	21	041	0100001	!	65	41	101	1000001	A	97	61	141	1100001	a
2	02	002	0000010	STX (start of text)	34	22	042	0100010	"	66	42	102	1000010	B	98	62	142	1100010	b
3	03	003	0000011	ETX (end of text)	35	23	043	0100011	#	67	43	103	1000011	C	99	63	143	1100011	c
4	04	004	0000100	EOT (end of transmission)	36	24	044	0100100	$	68	44	104	1000100	D	100	64	144	1100100	d
5	05	005	0000101	ENQ (enquiry)	37	25	045	0100101	%	69	45	105	1000101	E	101	65	145	1100101	e
6	06	006	0000110	ACK (acknowledge)	38	26	046	0100110	&	70	46	106	1000110	F	102	66	146	1100110	f
7	07	007	0000111	BEL (bell (ring))	39	27	047	0100111	'	71	47	107	1000111	G	103	67	147	1100111	g
8	08	010	0001000	BS (backspace)	40	28	050	0101000	(	72	48	110	1001000	H	104	68	150	1101000	h
9	09	011	0001001	HT (horizontal tab)	41	29	051	0101001	)	73	49	111	1001001	I	105	69	151	1101001	i
10	0A	012	0001010	LF (line feed)	42	2A	052	0101010	*	74	4A	112	1001010	J	106	6A	152	1101010	j
11	0B	013	0001011	VT (vertical tab)	43	2B	053	0101011	+	75	4B	113	1001011	K	107	6B	153	1101011	k
12	0C	014	0001100	FF (form feed)	44	2C	054	0101100	,	76	4C	114	1001100	L	108	6C	154	1101100	l
13	0D	015	0001101	CR (carriage return)	45	2D	055	0101101	-	77	4D	115	1001101	M	109	6D	155	1101101	m
14	0E	016	0001110	SO (shift out)	46	2E	056	0101110	.	78	4E	116	1001110	N	110	6E	156	1101110	n
15	0F	017	0001111	SI (shift in)	47	2F	057	0101111	/	79	4F	117	1001111	O	111	6F	157	1101111	o
16	10	020	0010000	DLE (data link escape)	48	30	060	0110000	0	80	50	120	1010000	P	112	70	160	1110000	p
17	11	021	0010001	DC1 (device control 1)	49	31	061	0110001	1	81	51	121	1010001	Q	113	71	161	1110001	q
18	12	022	0010010	DC2 (device control 2)	50	32	062	0110010	2	82	52	122	1010010	R	114	72	162	1110010	r
19	13	023	0010011	DC3 (device control 3)	51	33	063	0110011	3	83	53	123	1010011	S	115	73	163	1110011	s
20	14	024	0010100	DC4 (device control 4)	52	34	064	0110100	4	84	54	124	1010100	T	116	74	164	1110100	t
21	15	025	0010101	NAK (negative acknowledge)	53	35	065	0110101	5	85	55	125	1010101	U	117	75	165	1110101	u
22	16	026	0010110	SYN (synchronize)	54	36	066	0110110	6	86	56	126	1010110	V	118	76	166	1110110	v
23	17	027	0010111	ETB (end transmission block)	55	37	067	0110111	7	87	57	127	1010111	W	119	77	167	1110111	w
24	18	030	0011000	CAN (cancel)	56	38	070	0111000	8	88	58	130	1011000	X	120	78	170	1111000	x
25	19	031	0011001	EM (end of medium)	57	39	071	0111001	9	89	59	131	1011001	Y	121	79	171	1111001	y
26	1A	032	0011010	SUB (substitute)	58	3A	072	0111010	:	90	5A	132	1011010	Z	122	7A	172	1111010	z
27	1B	033	0011011	ESC (escape)	59	3B	073	0111011	;	91	5B	133	1011011	[	123	7B	173	1111011	{
28	1C	034	0011100	FS (file separator)	60	3C	074	0111100	<	92	5C	134	1011100	\	124	7C	174	1111100	|
29	1D	035	0011101	GS (group separator)	61	3D	075	0111101	=	93	5D	135	1011101	]	125	7D	175	1111101	}
30	1E	036	0011110	RS (record separator)	62	3E	076	0111110	>	94	5E	136	1011110	^	126	7E	176	1111110	~
31	1F	037	0011111	US (unit separator)	63	3F	077	0111111	?	95	5F	137	1011111	_	127	7F	177	1111111	DEL

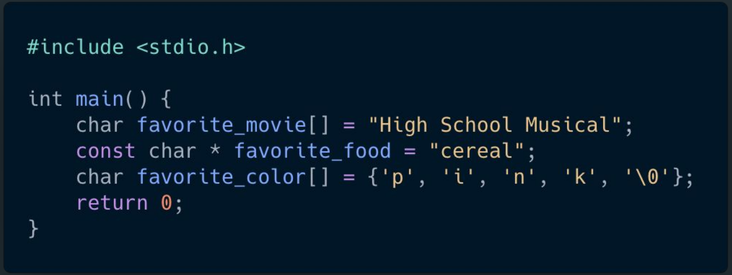
Key Values

0 – null character
10 – ‘\n’
32 – space
65 – ‘A’ (till 90 – ‘Z’)
97 – ‘a’ (till 122 – ‘z’)

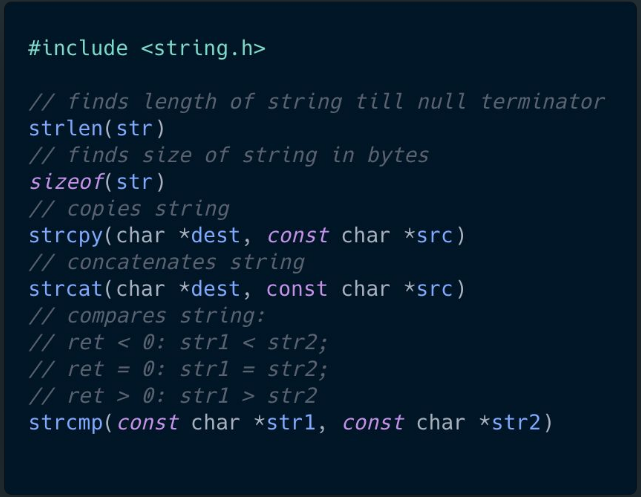
- behind the scenes, char is like an int (just takes up fewer bytes in memory – smaller range)
  - char digit = ‘4’ – 1
  - can be printed using %d

C-Strings

- Strings are an array of chars, with a null terminator (‘\0’)
- DON’T FORGET ABOUT THE NULL TERMINATOR



```c
#include <stdio.h>

int main() {
    char favorite_movie[] = "High School Musical";
    const char * favorite_food = "cereal";
    char favorite_color[] = {'p', 'i', 'n', 'k', '\0'};
    return 0;
}
```



```c
#include <string.h>

// finds length of string till null terminator
strlen(str)
// finds size of string in bytes
sizeof(str)
// copies string
strcpy(char *dest, const char *src)
// concatenates string
strcat(char *dest, const char *src)
// compares string:
// ret < 0: str1 < str2;
// ret = 0: str1 = str2;
// ret > 0: str1 > str2
strcmp(const char *str1, const char *str2)
```

- DON’Ts:
  - Try to copy a string: char * str_2 = str_1; (please don’t)
  - What to do instead? Iterate through each character, or use strcpy.

2D Arrays

```
int grid[10][10];
```

- **IMPORTANT:**
  - when passing a 2D array as an argument need second and following values:
    - `void foo(int grid[][10]);`

Iterating through a 2D array.

```
for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
    // DO SOMETHING grid[i][j]
  }
}
```

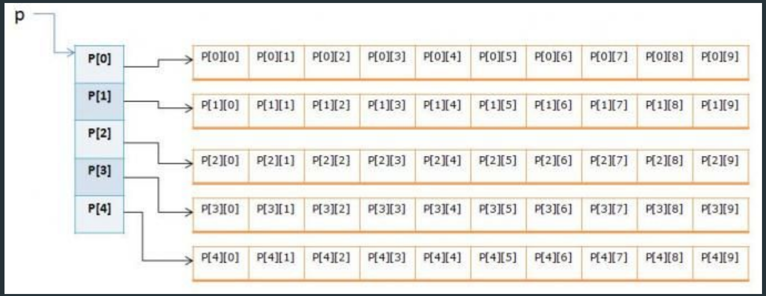
```
#include <stdio.h>

int main( void ) {
  int c , r;
  //do something
  int** grid = (int**)malloc( sizeof(int*)*r );
  for( int j=0 ; j<r ; j++ ) {
    grid[j] = (int*)malloc( sizeof(int)*c );
  }
  // do something
  for( int j=0 ; j<r ; j++ )
    free( grid[j] );
  free( grid );
  return 0;
}
```

```
grid[i][j] = (*(grid+i))[j] = *((*(grid+i))+j)
```

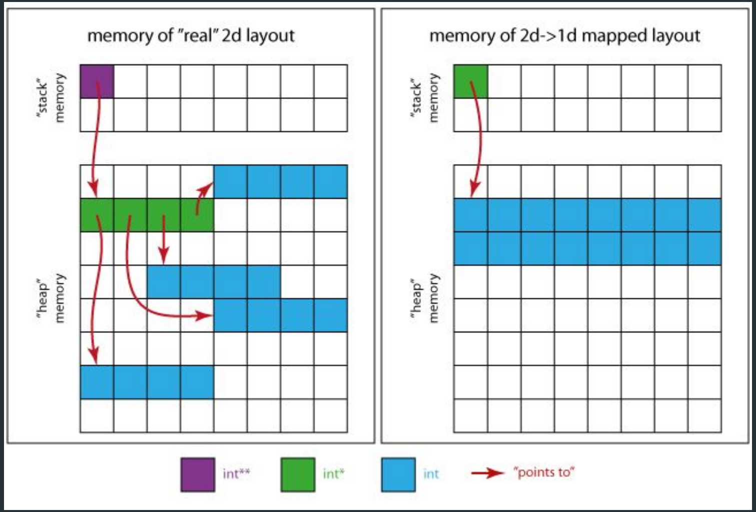
2D Arrays

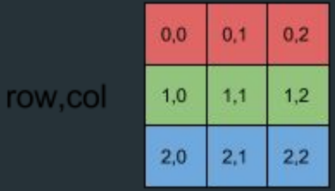
Two ways to initialize 2D arrays
- A 1D array that's indexed as a 2D array
- A twice dynamically allocated 2D array

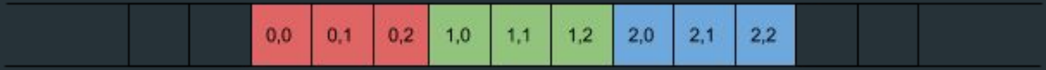
How will they be initialized?



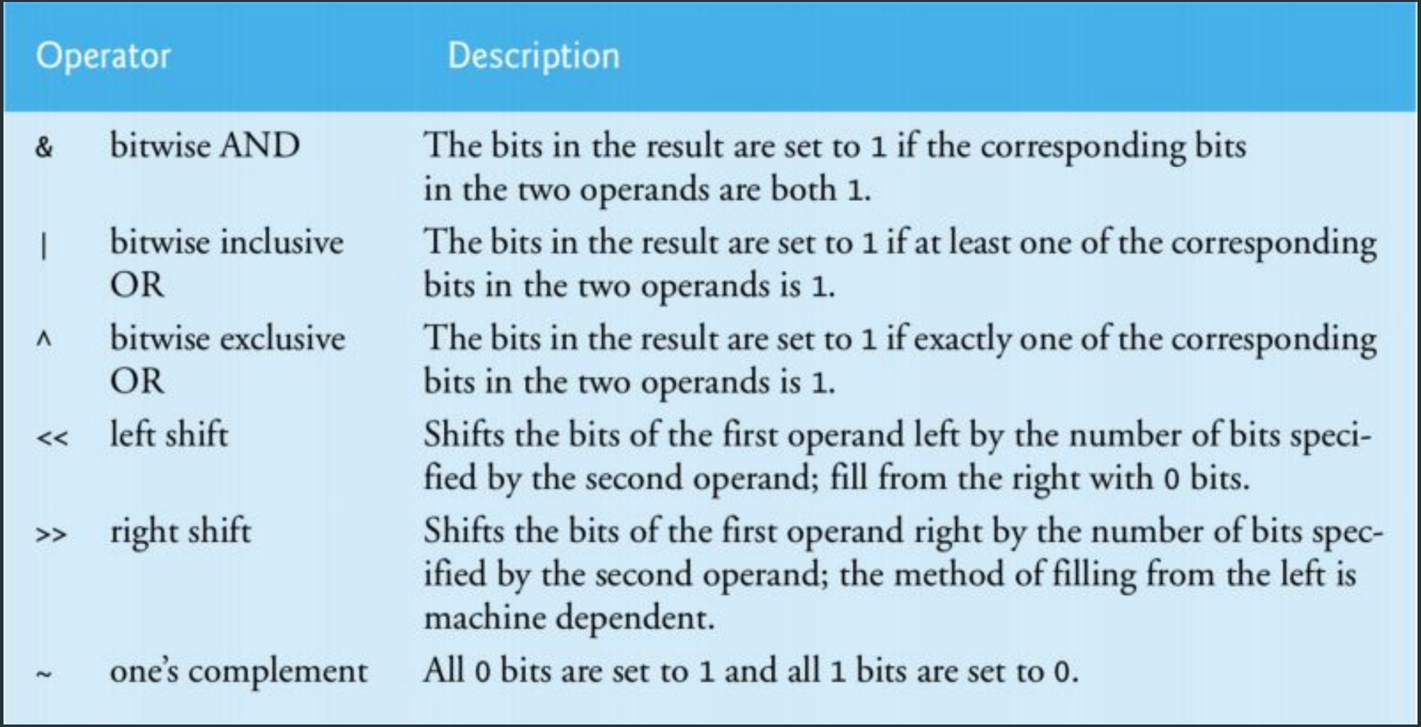
p
P[0]
P[1]
P[2]
P[3]
P[4]
P[0][0] P[0][1] P[0][2] P[0][3] P[0][4] P[0][5] P[0][6] P[0][7] P[0][8] P[0][9]
P[1][0] P[1][1] P[1][2] P[1][3] P[1][4] P[1][5] P[1][6] P[1][7] P[1][8] P[1][9]
P[2][0] P[2][1] P[2][2] P[2][3] P[2][4] P[2][5] P[2][6] P[2][7] P[2][8] P[2][9]
P[3][0] P[3][1] P[3][2] P[3][3] P[3][4] P[3][5] P[3][6] P[3][7] P[3][8] P[3][9]
P[4][0] P[4][1] P[4][2] P[4][3] P[4][4] P[4][5] P[4][6] P[4][7] P[4][8] P[4][9]





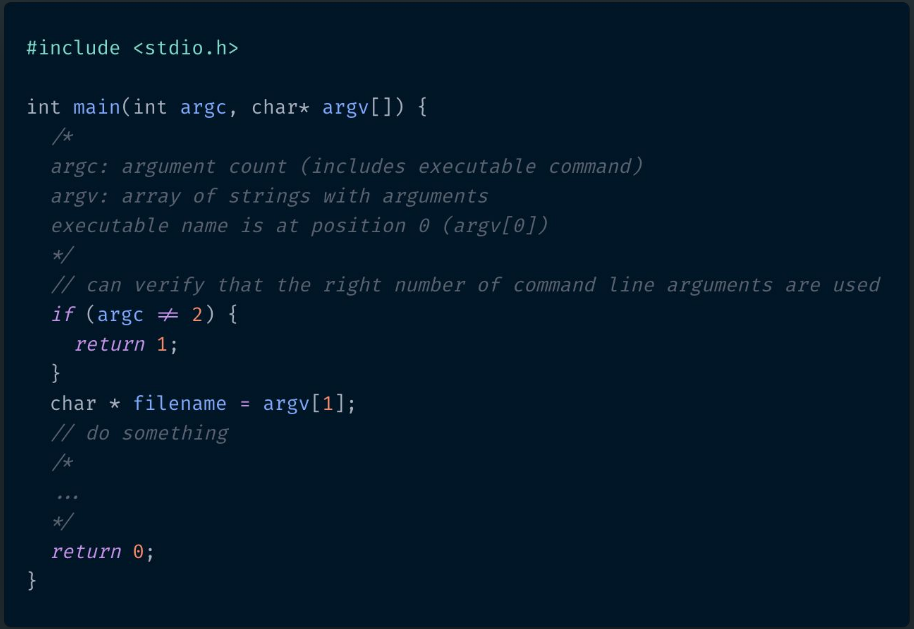


Bit Operators



Operator		Description
&	bitwise AND	The bits in the result are set to 1 if the corresponding bits in the two operands are both 1.
|	bitwise inclusive OR	The bits in the result are set to 1 if at least one of the corresponding bits in the two operands is 1.
^	bitwise exclusive OR	The bits in the result are set to 1 if exactly one of the corresponding bits in the two operands is 1.
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from the right with 0 bits.
>>	right shift	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent.
~	one's complement	All 0 bits are set to 1 and all 1 bits are set to 0.

Command line Arguments



```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  /*
  argc: argument count (includes executable command)
  argv: array of strings with arguments
  executable name is at position 0 (argv[0])
  */
  // can verify that the right number of command line arguments are used
  if (argc ≠ 2) {
    return 1;
  }
  char * filename = argv[1];
  // do something
  /*
  ...
  */
  return 0;
}
```

- ./hw3 text.txt
  - argc = 2
  - argv[0] -> ./hw3
  - argv[1] -> text.txt

Swap Function

- A pointer is a variable that stores a memory address/location
- Every pointer points to a specific data type (int * , char *, float *, ...)

- address-of operator &: returns address
- dereferencing operator *: returns value being pointed to

```c
#include <stdio.h>

void swap( int *px , int *py ) {
  int temp = *px;
  *px = *py;
  *py = temp;
}

int main( void ) {
  int a = 1 , b =2;
  swap( &a , &b );
  printf( "%d %d\n" , a , b );
  return 0;
}
```

Pass-by-value (very important concept)

- EVERY TYPE (int, float, char, pointer, …) passed as an argument to a function uses pass-by-value
  - Special case: array arguments
- What does this mean?
  - A copy of argument item is made, so changes made to that copy inside the function won't be noticeable outside of it.
  - For pointers: if you pass a pointer argument p which points to some item x, you receive a copy of p. But a copy of a pointer, when dereferenced, gets you to the same item x.
    - So changes made via the pointer to x will be noticeable outside the function!

Pass-by-value (very important concept)

- EVERY TYPE (int, float, char, pointer, ...) passed as an argument to a function uses pass-by-value
  - Special case: array arguments
- What does this mean?
  - An array argument has its <u>address</u> passed (copied), not its contents. (It's as if you passed in a pointer to the array.) So changing contents of array elements within that function <u>will</u> modify the original, and edits are noticeable outside!

Pointers

- A pointer is a variable that stores a memory address/location
- Every pointer points to a specific data type (int * , char *, float *, ...)

- address-of operator &: returns address
- dereferencing operator *: returns value being pointed to

Example

```
int i = 1;
int * p = &i;
```

`*p`	value stored at address p
`&i`	Address of i
`p`	Pointer to i

Lifetime/Scope

- **Local variables** live in a region of memory known as the stack
    - Stack frames are added/removed as functions get called and then return

- Both **static and global variables** live in a region of memory known as the data segment
    - The data segment is allocated when program begins, freed when program exits

- **Dynamically-allocated memory** lives in a third region of memory, called the heap
    - User is responsible for allocating and freeing memory in the heap

Dynamic Memory Allocation

- **malloc:** make dynamically-allocated memory lives "on the heap"
  - lives as long as we want
  - we are responsible for deallocating it using free

```
int * arr = (int *) malloc(sizeof(int) * n);
free(arr);
```

- **calloc:** allocate, then initialize elements to zero

```
int * arr = (int *) calloc(n, sizeof(int));
```

- **realloc:** adjusts dynamically-allocated memory's size
  - if needed: copies data from previously allocated mem and frees "old" memory
  - returns a pointer to the newly-resized memory

Static Variables (!!)

- automatically initialized to zero once
- not destroyed at the end of block of code
- its value in next call will be the same as when block (usually a function) was executed last

Pointer Arithmetic

```
int * arr = malloc(n * sizeof(int));

*(arr + i) = arr[i]
```

- arr points to the first element of the array
  - `arr + 1` : points to the second one.
  - `arr++` : moves the pointer to the next element.

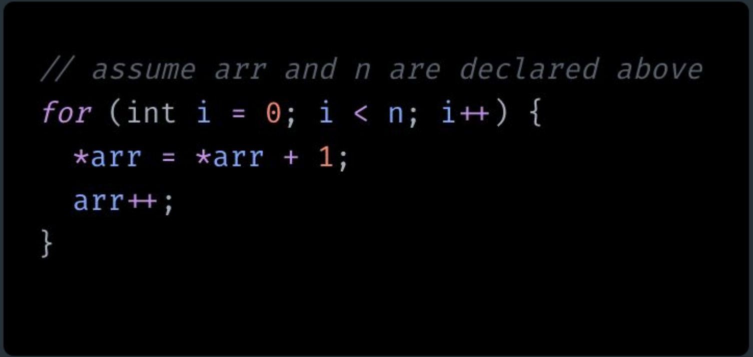
- This works regardless of the size of the data element in the array

```
#include <stdio.h>

int main( void ) {
  int a[] = { 2 , 4 , 6 , 8 };
  int* b = a+2;
  printf( "%d %d\n" , *a , *b );
  return 0;
}
```

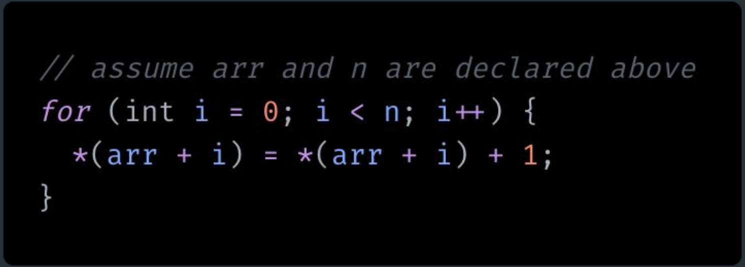
Ans: 2 6

Pointer Arithmetic

What is the downside of doing this?



```
// assume arr and n are declared above
for (int i = 0; i < n; i++) {
  *arr = *arr + 1;
  arr++;
}
```



```
// assume arr and n are declared above
for (int i = 0; i < n; i++) {
  *(arr + i) = *(arr + i) + 1;
}
```

Pointer Arithmetic

What is the downside of doing this?

```
// assume arr and n are declared above
for (int i = 0; i < n; i++) {
  *arr = *arr + 1;
  arr++;
}
```

```
// assume arr and n are declared above
for (int i = 0; i < n; i++) {
  *(arr + i) = *(arr + i) + 1;
}
```

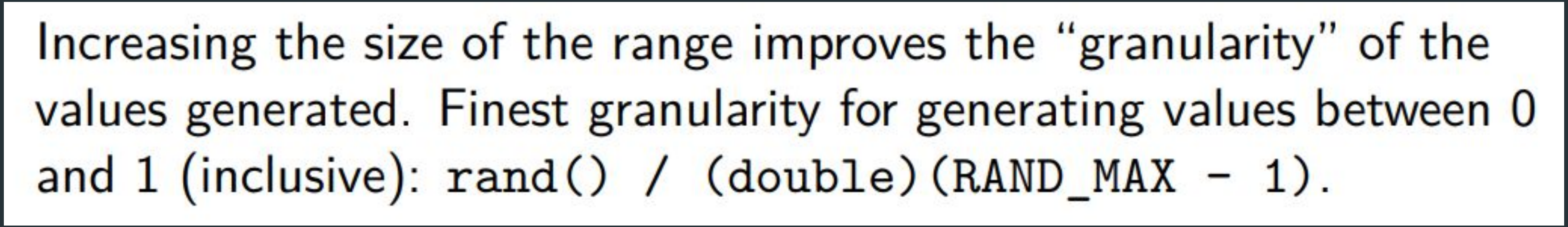
You lose access to the first arr pointer!

Random Number Generation

- `rand()` generates (pseudo) random integers between `0` and `RAND_MAX`
- Distribution is uniform: each value in range is equally likely to be generated
- The pseudo random sequence of integers is based on a seed
- `srand(unsigned int)` sets the seed value
- The modulus (%) operator is useful for constraining the range of values generated by `rand()`

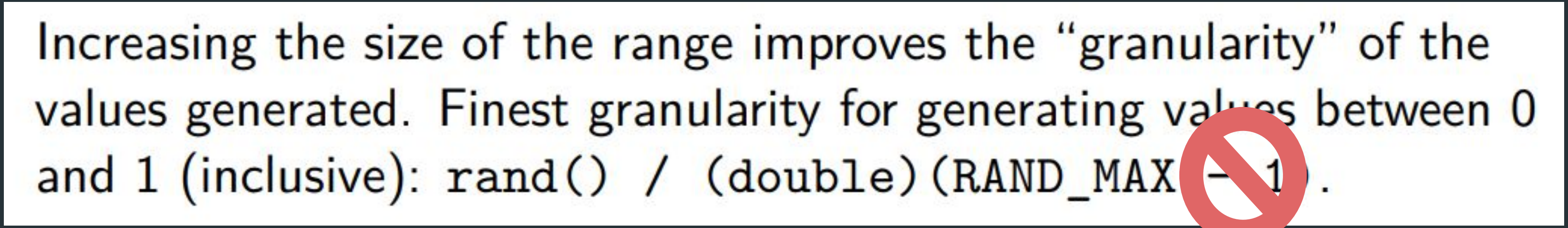
Random Number Generation

- Generating pseudo-random integers in a specific range
  - `0 to 100 (inclusive) → rand() % 101`
- Generating pseudo-random floating point values
  - `0.0 to 1.0 (inclusive) → ((rand() % 100001) / 100000.0)`
  - `0.0 to 1.0 (inclusive) → rand() / (double)(RAND_MAX - 1)`


Increasing the size of the range improves the "granularity" of the values generated. Finest granularity for generating values between 0 and 1 (inclusive): `rand() / (double)(RAND_MAX - 1)`.

Random Number Generation

- Generating pseudo-random integers in a specific range
  - `0 to 100 (inclusive) → rand() % 101`
- Generating pseudo-random floating point values
  - `0.0 to 1.0 (inclusive) → ((rand() % 100001) / 100000.0)`
  - `0.0 to 1.0 (inclusive) → rand() / (double)(RAND_MAX)` ✓


Increasing the size of the range improves the "granularity" of the values generated. Finest granularity for generating values between 0 and 1 (inclusive): `rand() / (double)(RAND_MAX - 1)`.

Structs

```c
struct person {
    char * name;
    int age;
    bool isAwesome;
};
```

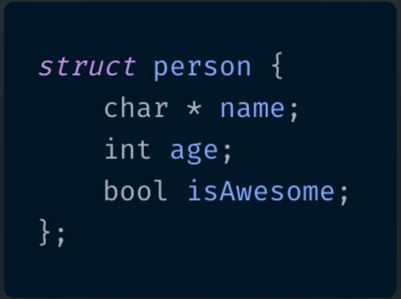
```c
typedef struct _person {
    char * name;
    int age;
    bool isAwesome;
} Person;
```

Declaration

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    struct person ca;
    ca.name = “Ryan”;
    ca.age = 21;
    ca.isAwesome = true;
    return 0;
}
```

```c
#include <stdio.h>
#include <stdbool.h>

int main(void) {
  Person ca;
  ca.name = "Ryan";
  ca.age = 21;
  ca.isAwesome = true;
  return 0;
}
```

Structs sizes



```c
struct person {
    char * name;
    int age;
    bool isAwesome;
};
```

- pointers = 8 bytes
- int = 4 bytes
- bool = 1 byte

Q: What's the size of this struct?

Accessing elements of struct *

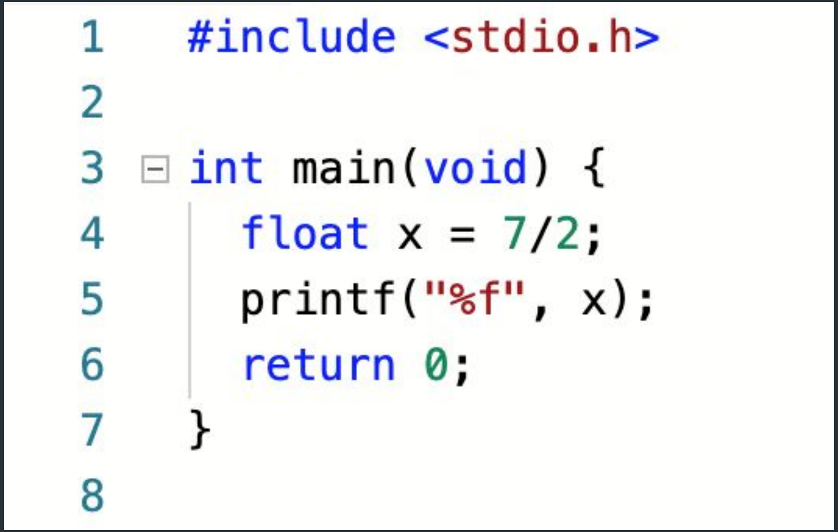
```c
struct person {
    char * name;
    int age;
    bool isAwesome;
};
```

```c
#include <stdio.h>
#include <stdbool.h>

int main(void) {
  //allocation
  struct person * p = malloc(sizeof(struct person));

  // either works
  p->age = 25;
  (*p).age = 25;

  p->name = malloc(sizeof(char) * MAX_NAME_SIZE));

  // freeing dynamically allocated memory
  free(p->name);
  free(p);

  // # of malloc = # of frees
}
```

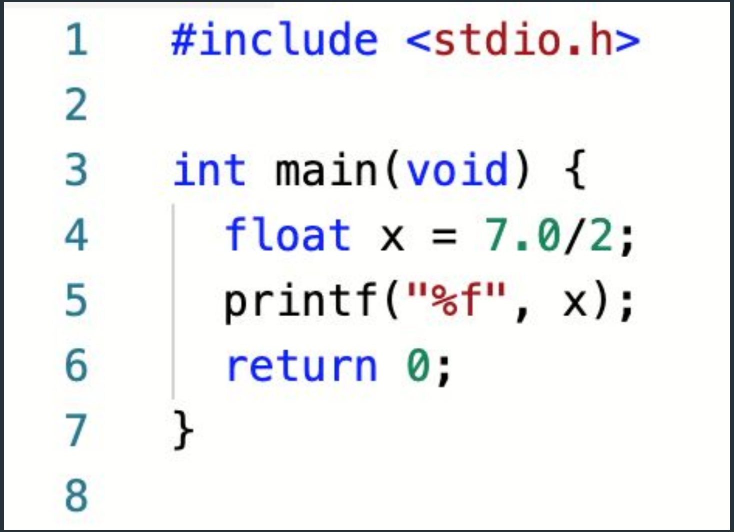
Promotion | Narrowing | Casting

- **PROMOTION:** smaller type is promoted to larger
  - `char < int < unsigned < long < float < double`
- **NARROWING:** from larger to smaller types
- **CASTING:** gives programmer control over promotion and narrowing

Promotion | Narrowing | Casting



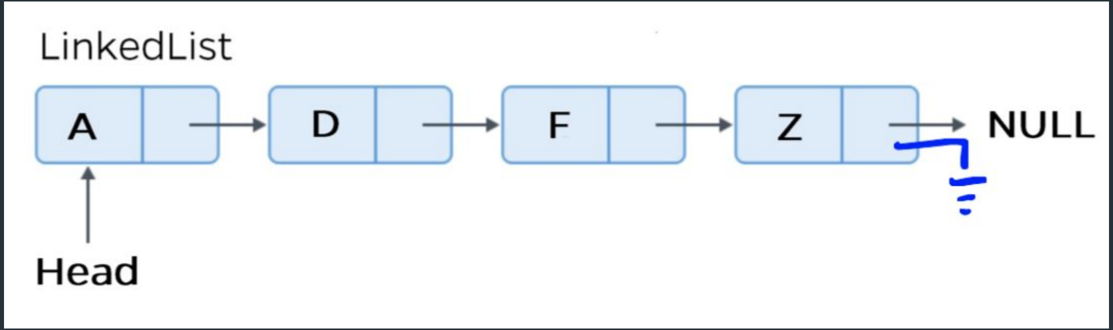
```c
#include <stdio.h>

int main(void) {
   float x = 7/2;
   printf("%f", x);
   return 0;
}

```



```c
#include <stdio.h>

int main(void) {
   float x = 7.0/2;
   printf("%f", x);
   return 0;
}

```

Linked Lists (IMPORTANT)

```
typedef struct _node {
  char data;
  struct _node *next;
} Node;
```



Be ready to write functions for linked lists, such as…

- `create_node`
- `length`
- `print (iterative and recursive), reverse_print`
- `add_front`
- `add_after`
- `delete_front`
- `delete_after`
- `delete_at // argument is int position number`
- `clear_list`
- `copy_list`

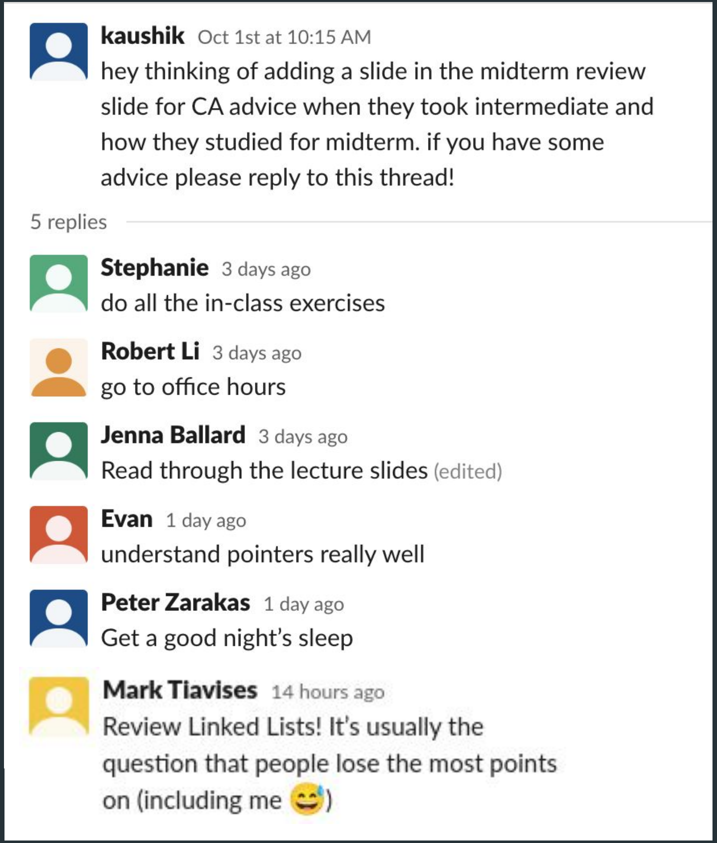
Must-Review

- Linked lists
- Pointers
- Dynamic memory allocation

Also review topics we didn’t cover!

Number representation, passing arrays to functions, control flow and more!

Tips from CAs on Midterm



kaushik Oct 1st at 10:15 AM
hey thinking of adding a slide in the midterm review slide for CA advice when they took intermediate and how they studied for midterm. if you have some advice please reply to this thread!

5 replies

Stephanie 3 days ago
do all the in-class exercises

Robert Li 3 days ago
go to office hours

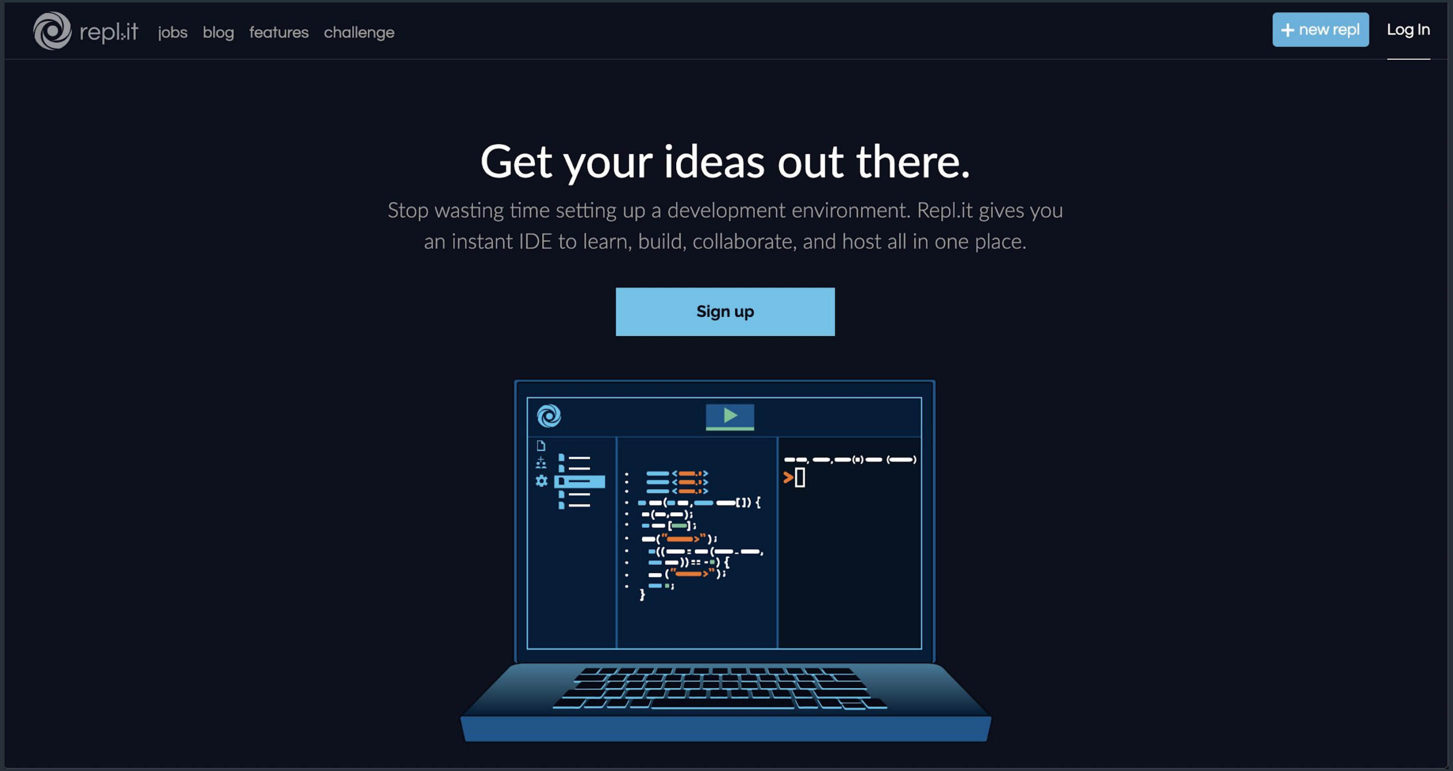
Jenna Ballard 3 days ago
Read through the lecture slides (edited)

Evan 1 day ago
understand pointers really well

Peter Zarakas 1 day ago
Get a good night's sleep

Mark Tiavises 14 hours ago
Review Linked Lists! It's usually the question that people lose the most points on (including me 😅)

repl.it



Get your ideas out there.

Stop wasting time setting up a development environment. Repl.it gives you an instant IDE to learn, build, collaborate, and host all in one place.

Questions?