Lecture 6
C-Strings and Arrays

TIC1001 Introduction to Computing and Programming I

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
6	14	15	16	17 L	18	19 T	20
Recess	21	22	23	24	25	26 Makeup	27
7	28	29	30	1 MT	2	3 PE1	4
8	5	6	7	8 L	9	10 T	11
9	12	13	14	15 L	16	17 T	18
10	19	20	21	22 L	23	24 T	25
11	26	27	28	29 L	30	31 T	1
12	2	3	4	5 L	6	7 T	8
13	9	10	11	12 PE2	13	14	15
Reading	16	17	18	19	20	21	22
Exam	23	24	25	26	27	28	29
Exam	30	1 Final	2	3	4	5	6

Midterm Test – Thu 1 Oct 2020

Time: 7 pm to 8 pm

- Online: Zoom + Examplify

Scope: Lecture 1 to 5

- Everything you have learnt so far
- Open book
- No programmable calculators

Practical Exam 1 – Sat, 3 Oct 2020

Time: 10 am to 11 am

- Online: Zoom + Examplify

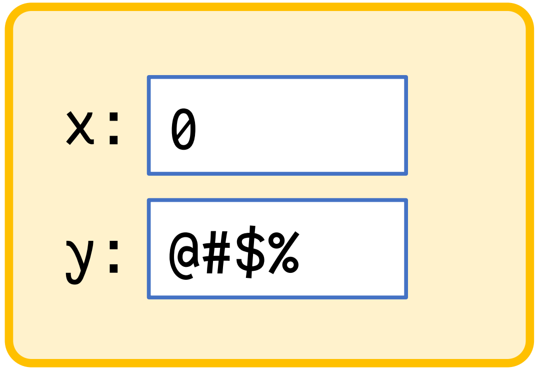
Scope: Lecture 1 to 5

- Everything you have learnt so far
- Open book
- Visual Studio Code on your own machine

Recap: Variables

Variables are aliases to memory spaces

```
int x = 0, y;
y = x;
x = 5;
printf("%d %d", x, y);
```

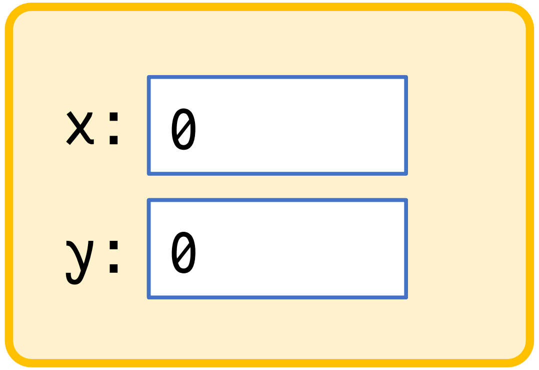


- y has no initial value given
- any content that happens to be in memory

Recap: Variables

Variables are aliases to memory spaces

```
int x = 0, y;
y = 0;
x = 5;
printf("%d %d", x, y);
```



= is the assignment operator

- Evaluate the RHS
- Assign the value to LHS

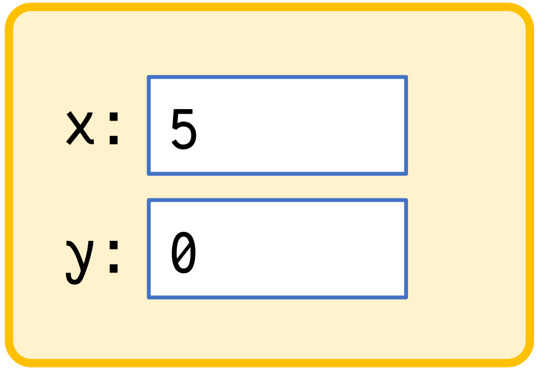
It does not “bind” x and y together.

- x evaluates to 0
- 0 is assigned to y

Recap: Variables

Variables are aliases to memory spaces

```c
int x = 0, y;
y = x;
x = 5;
printf("%d %d", x, y);
```



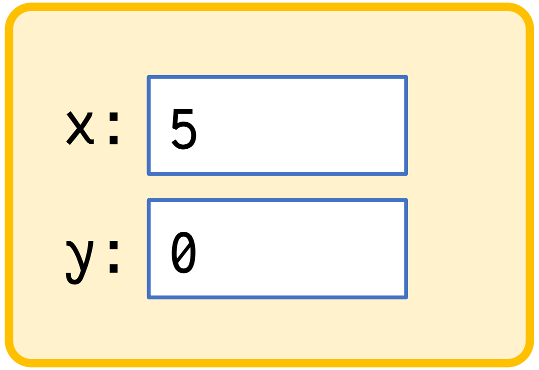
5 is assigned to x

– y is unchanged

Recap: Variables

Variables are aliases to memory spaces

```c
int x = 0, y;
y = x;
x = 5;
printf("%d %d", x, y);
```



output

```
5 0
```

Recap: Functions

Formal parameters, aka “parameters”

```
double foo(int x, int y) {
    return x/y;
}

int x = 5;
int y = 3;
printf("%f", foo(y, x));
```

Placeholder variables for input to function

Recap: Functions

Actual parameters, aka “arguments”

```
double foo(int x, int y) {
    return x/y;
}

int x = 5;
int y = 3;
printf("%f", foo(y, x));
```

The actual values that are passed into the function

Recap: Function Call

Arguments must be "fully evaluated"

```c
double foo(int x, int y) {
    return x/y;
}
```

```c
int x = 5;
int y = 3;
printf("%f", foo(y, x));
```

```c
foo(3, 5)
```

```
x: 5
y: 3
```

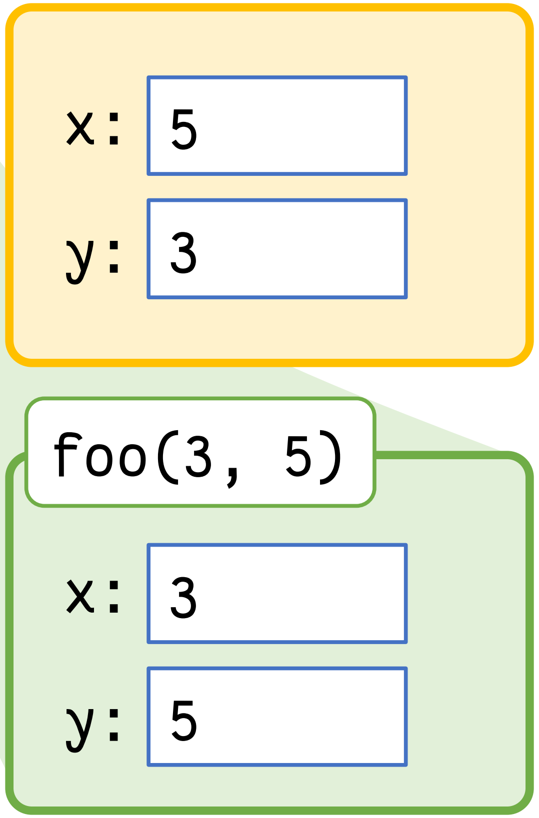
Execution is passed to the function (function call)

Recap: Function Call

Values of arguments are copied

```c
double foo(int x, int y) {
    return x/y;
}

int x = 5;
int y = 3;
printf("%f", foo(y, x));
```



Recap: Function Call

Return expression must be evaluated

```
double foo(int x, int y) {
    return 3/5; →0
}
```

x: 5

y: 3

foo(3, 5)

x: 3

y: 5

```
int x = 5;
int y = 3;
printf("%f", foo(y, x));
```

Recap: Function Call

Execution resumes from calling point

```c
double foo(int x, int y) {
    return x/y;
}
```

```c
int x = 5;
int y = 3;
printf("%f", foo(y, x));
```

x: 5

y: 3

output

```
0.000000
```

Another Example

Function calling function

```
int f(int a) {
    a = 7;
    return a;
}
int g(int a) {
    return f(a) + a;
}
int a = 3;
printf("%d", g(a+2) + f(a-1));
```

a: 3

Another Example

Evaluate the argument

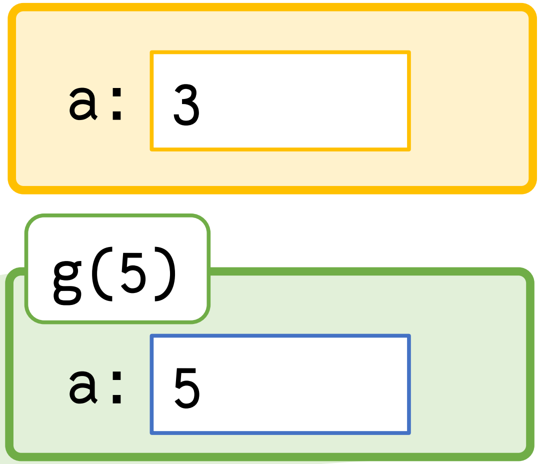
```c
int f(int a) {
    a = 7;
    return a;
}
int g(int a) {
    return f(a) + a;
}
int a = 3;
printf("%d", g( 5 ) + f(a-1));
```

```
a: 3
```

Another Example

Calling another function

```c
int f(int a) {
    a = 7;
    return a;
}
int g(int a) {
    return f(5) + 5;
}
int a = 3;
printf("%d", g(5) + f(a-1));
```



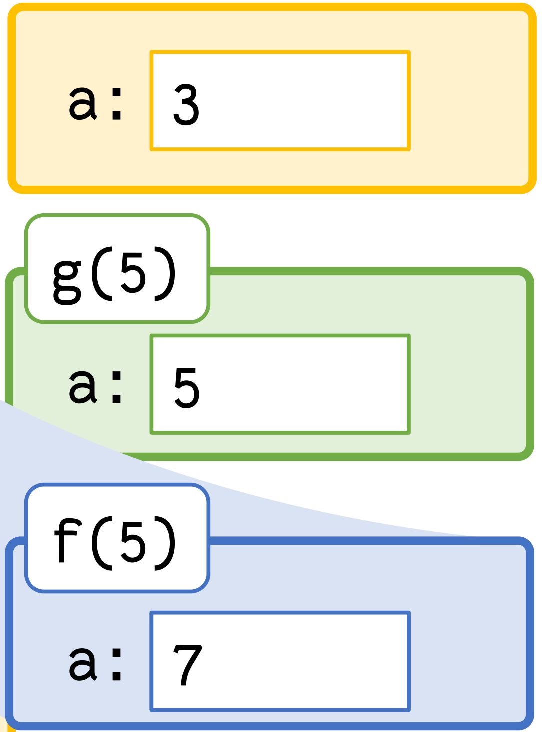
Another Example

Executes in new memory space

```c
int f(int a) {
    a = 7;
    return 7;
}
```

```c
int g(int a) {
    return f(5) + 5;
}
```

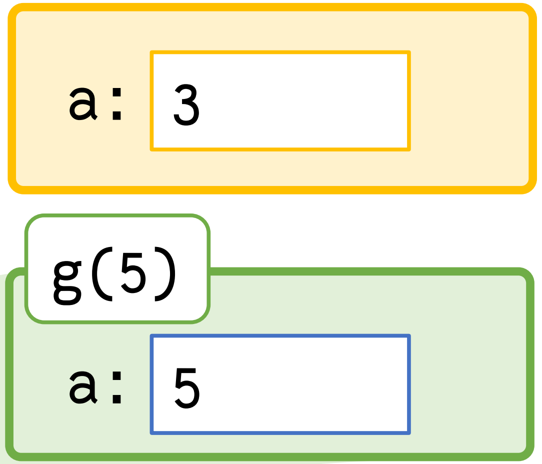
```c
int a = 3;
printf("%d", g( 5 ) + f(a-1));
```



Another Example

Value is returned

```c
int f(int a) {
    a = 7;
    return a;
}
int g(int a) {
    return 12;
}
int a = 3;
printf("%d", g(5) + f(a-1));
```



Another Example

Value is returned

```c
int f(int a) {
    a = 7;
    return a;
}
int g(int a) {
    return f(a) + a;
}
int a = 3;
printf("%d", 12 + f(a-1));
```

a: 3

Another Example

Evaluate argument

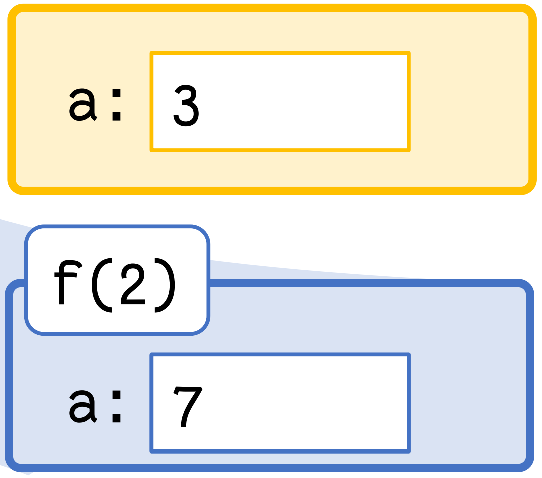
```
int f(int a) {
    a = 7;
    return a;
}
int g(int a) {
    return f(a) + a;
}
int a = 3;
printf("%d",  12  + f( 2 ));
```

```
a: 3
```

Another Example

Execute in new memory space

```
int f(int a) {
    a = 7;
    return 7;
}
int g(int a) {
    return f(a)5+ a;
}
int a = 3;
printf("%d",  12  + f( 2 ));
```



Another Example

Value is returned

```
int f(int a) {
    a = 7;
    return a;
}
int g(int a) {
    return f(a) + a;
}
int a = 3;
printf("%d", 12 + 7 );
```

a: 3

output

```
19
```

Recap: Scope

Variables are local to block

```c
int i = 1, j = 1;
if (i == 1) {
    int i = 2;
    j = 2;
}
printf("%d %d", i, j);
```

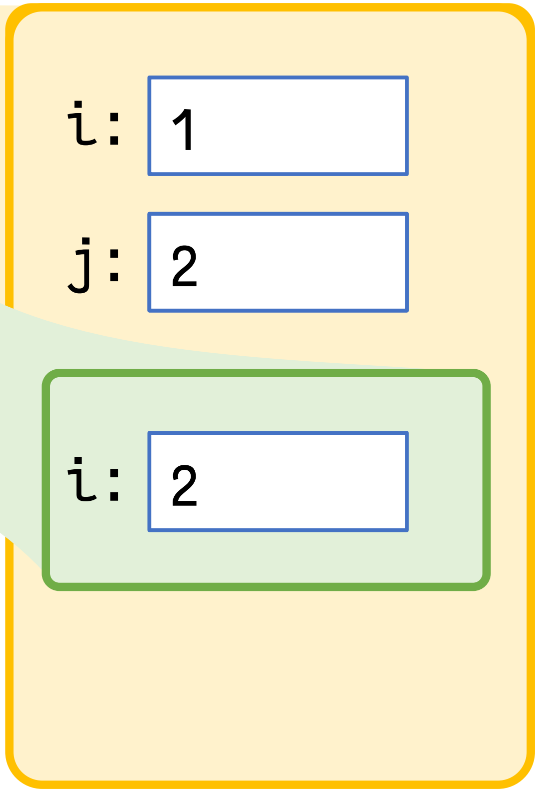
i: 1

j: 1

Recap: Scope

Variables are local to block

```c
int i = 1, j = 1;
if (i == 1) {
    int i = 2;
    j = 2;
}
printf("%d %d", i, j);
```



Recap: Scope

Variables are local to block

```c
int i = 1, j = 1;
if (i == 1) {
    int i = 2;
    j = 2;
}
printf("%d %d", i, j);
```

i: 1

j: 2

output

```
1 2
```

Recap: Pass-by-Pointer

```c
int f(int *x) {
    if (*x > 0)
        *x = -*x;
    else
        *x += 2;
    return *x;
}
int x = 0;
x = f(&x) + f(&x);
printf("%d", x);
```

x: 0 0x01

Recap: Pass-by-Pointer

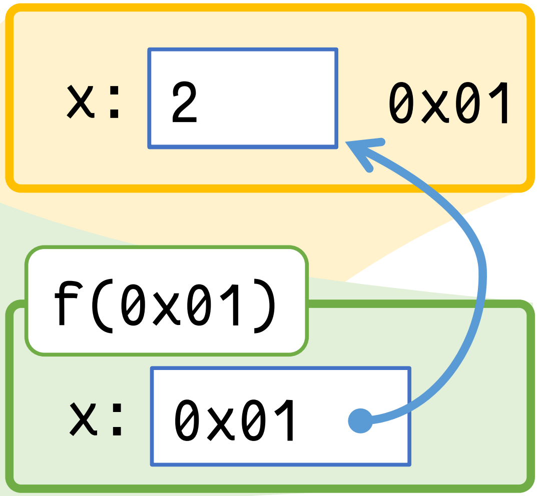
```
int f(int *x) {
    if (*x > 0)
        *x = -*x;
    else
        *x += 2;
    return *x;
}
int x = 0;
x = f( 0x01 ) + f(&x);
printf("%d", x);
```

x: 0 0x01

Recap: Pass-by-Pointer

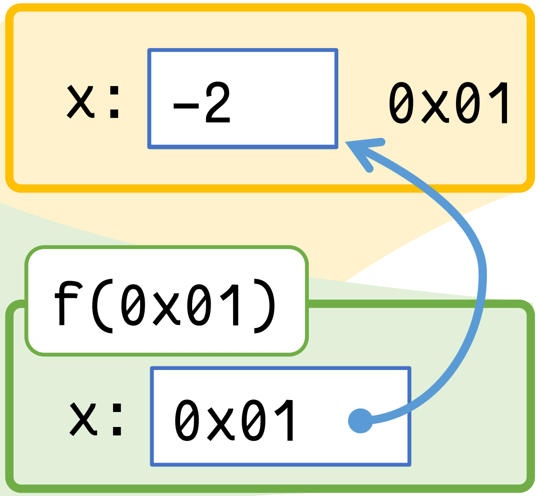
```c
int f(int *x) {
    if (*x > 0)
        *x = -*x;
    else
        *x += 2;
    return 2 ;
}
```

```c
int x = 0;
x = f( 0x01 ) + f(&x);
printf("%d", x);
```



Recap: Pass-by-Pointer

```c
int f(int *x) {
    if (*x > 0)
        *x = -2;
    else
        *x += 2;
    return -2;
}
int x = 0;
x = 2 + f(0x01);
printf("%d", x);
```



Recap: Pass-by-Pointer

```c
int f(int *x) {
    if (*x > 0)
        *x = -*x;
    else
        *x += 2;
    return *x;
}
int x = 0;
x =       0       ;
printf("%d", x);
```

x: 0   0x01

output

```
0
```

C exposes actual memory storage to programmers

Understanding how memory is organized is important

Arrays

What are arrays?

Abstractly

- Compound data
- A sequence of identical types
- e.g. a list of `int`

Concretely

- A sequential group of memory locations
- Number of elements must be pre-specified

Declaring Arrays

Size (length) of array in square brackets

```
int x[8];
```

Compiler will set aside contiguous memory

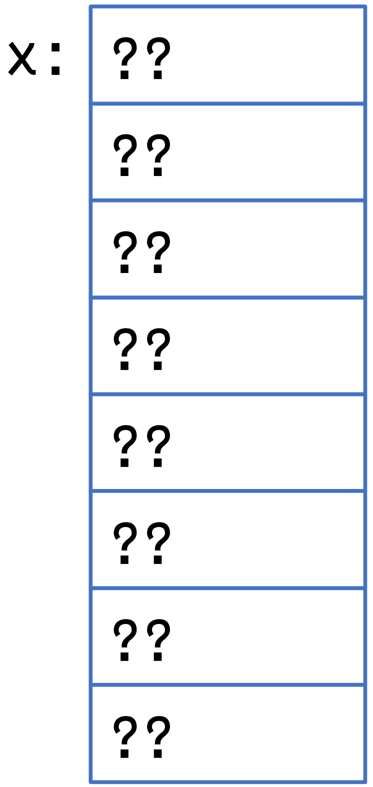
– to accommodate all elements

```
cout << sizeof(x) << endl;
```

output

```
32
```

– `int` is 4 bytes, so $8 \times 4 = 32$ bytes



Initializing Arrays

Using an initializer list

```
int x[8] = {16, 12, 8, 6, 0, 3, 6, 3};
```

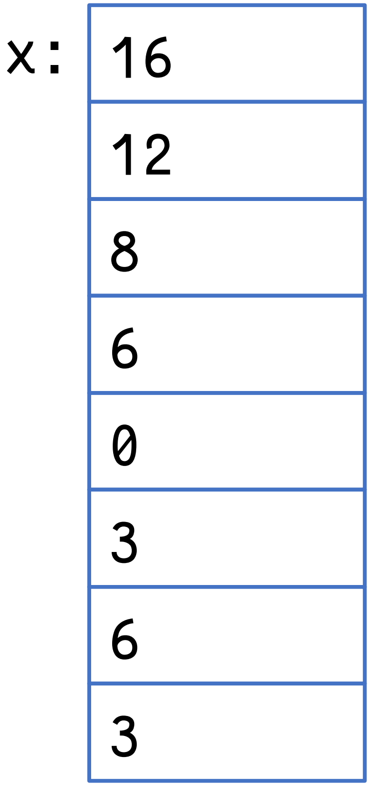
- specifies the initial values
- during declaration only
- if shorter than array size, padded with zeros

both declarations below are equal

```
int x[8] = {16,0,0,0,0,0,0,0};
int x[8] = {16};
```

to zero the array

```
int x[8] = {0};
```



Array Subscripts

Elements are accessed individually using [ ]

- Starts from 0

What happens if subscript is beyond the size of array?

- e.g. `x[8]`
- This is a common mistake
- No error or warnings!
- Program may crash at runtime
- Logic error may occur

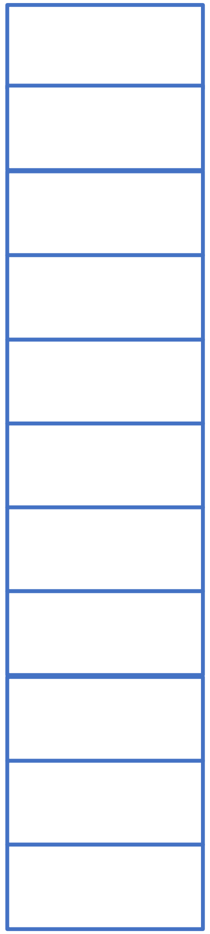
`x[0]:`	16
`x[1]:`	12
`x[2]:`	8
`x[3]:`	6
`x[4]:`	0
`x[5]:`	3
`x[6]:`	6
`x[7]:`	3

Example: Accessing Array

Fill an array with squared values $0^2, 1^2, 2^2, \ldots, 10^2$

```
#define SIZE 11
int main(void) {
    int i, sq[SIZE];
    for (i = 0; i < SIZE; i++) {
        sq[i] = i * i;
    }
    return 0;
}
```

sq:



Looping condition must ensure final iteration with `i = 10`, since array elements are `sq[0]` through `sq[10]`.

Array Assignment

Arrays cannot be assigned

```
int main() {
  int i[10] = {0};
  int j[10];
  j = i;
  return 0;
}
```

Invalid array assignment

Copy an Array

Have to copy an array

```
void copy(int dst[], int src[], int size) {
  for (int i = 0; i < size; i++) {
    dst[i] = src[i];
  }
}

int main() {
  int i[10] = {1, 2, 3, 4, 5};
  int j[10];
  copy(j, i, 10);
  return 0;
}
```

Passing Array into Functions

You can declare the formal parameters as

1. an unsized array

```
void my_function(int param[])
```

2. a sized array

```
void my_function(int param[10])
```

3. a pointer

```
void my_function(int *param)
```

The result of all three ways is fundamentally identical

- It decays into a pointer

Passing size of the Array

You might often need to specify the size of the array

Example: display our array

```
void print_arr(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d ", arr[i]);
    }
}
```

Arrays are Passed-by-Pointer

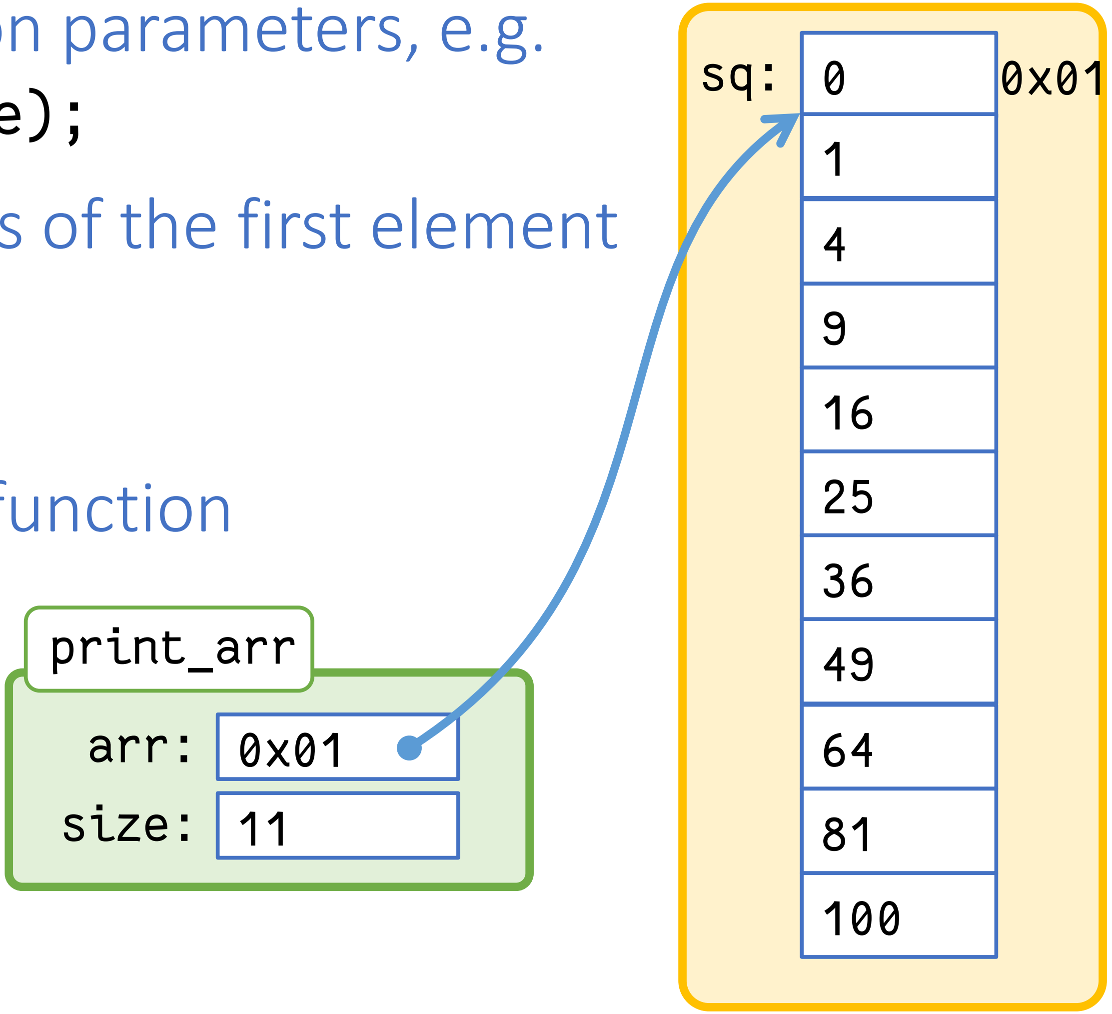
Array must be declared in the function parameters, e.g.

```
void print_arr(int arr[], int size);
```

An array actually refers to the address of the first element

- array is passed as pointer
- It is “shared” between the functions

Size must be explicitly passed to the function

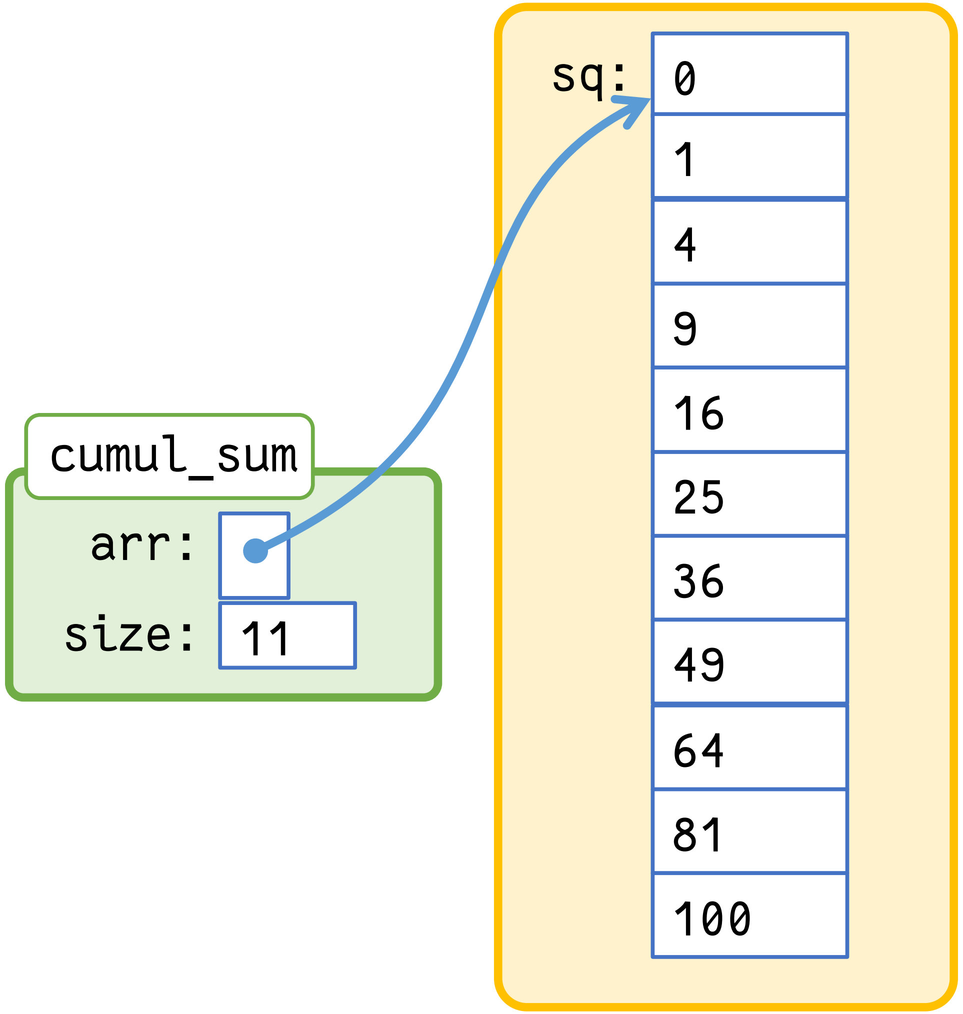


Example: Cumulative Sum

Cumulative sum of a sequence $[a, b, c, \ldots]$ is given by $[a, a+b, a+b+c, \ldots]$

```
void cumul_sum(int arr[], int size) {
    for (int i = 1; i < size; ++i) {
        arr[i] = arr[i] + arr[i-1];
    }
}
// sq from earlier
cumul_sum(sq, 10);
```

- Note that `i` loops from 1 to n-1
- Ensure that array access stays within bounds

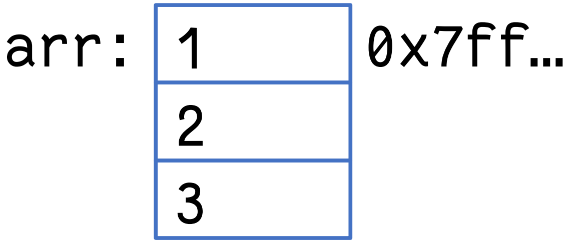


Arrays are pointers

The value of an array

– is the address of the start of the contiguous memory allocated

```c
int arr[3] = {1, 2, 3};
printf("arr    : %p\n", arr);
printf("&arr   : %p\n", &arr);
printf("&arr[0]: %p\n", &arr[0]);
printf("&arr[1]: %p\n", &arr[1]);
```



output

```
arr    : 0x7fffe8ce3b60
&arr   : 0x7fffe8ce3b60
&arr[0]: 0x7fffe8ce3b60
&arr[1]: 0x7fffe8ce3b64
```

Out of Bounds

C/C++ does not check out of bounds access

```
int arr[3] = {1, 2, 3};
printf("%d %d %d\n", arr[0], arr[1], arr[2]);
printf("%d\n", arr[4]);

arr[4] = 42;
printf("%d\n", arr[4]);
```

output

```
1 2 3
0
42
```

arr:	1	0x7ff…
	2	
	3	
x:	42	

- No error or warnings
- Either segmentation fault (crash), or corrupted data

Summary: Arrays

A sequence of data

- homogenous; all elements the same type
- contiguous in memory

Initialized using

- initializer lists

```
int x[8] = {0};
```

- only at declaration

Elements can be accessed by subscript

- Accessing outside index bounds can lead to errors

Passed into functions as pointers

- Size must be explicitly passed to stay within bounds

Strings and Characters

Character Type

- Literal is enclosed in single quotes

  `Examples 'A', 'b', '3', '\n'`

- 8 bits (1 byte) of memory
  - Internally represented as an integer
  - Mapped to a value in the ASCII table

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Character Type

Literal is enclosed in single quotes

`examples 'A', 'b', '3', '\n'`

8 bits (1 byte) of memory

– Internally represented as an integer
– Mapped to ASCII table

Character arithmetic

`'A'+1` → `'B'`

`'d'-32` → `'D'`

Character relations

`'\0' < '0' < '9' < 'A' < 'Z' < 'a' < 'z'`

Using Characters

Declaring characters

```c
char c;
char c = 'd';
```

Printing

```c
printf("The character is %c\n", c);
```

Array of Characters

Stringing characters together

```
char vowel[5] = {'a','e','i','o','u’}
```

Strings are just array of characters

```
char code[8] = "tic1001";
```

- Known as C-strings
- Literal enclosed in double quotes
- Ends will a null character

Character String

Character array ending with a null

`'\0'` or `0` (Both are equivalent)

Initialization during declaration only

```
char code[8] = "tic1001";
```

To store a string of n characters

- ensure array size of at least n+1
- to accommodate the null terminator

Print using

```
printf("Module code: %s\n", code);
```

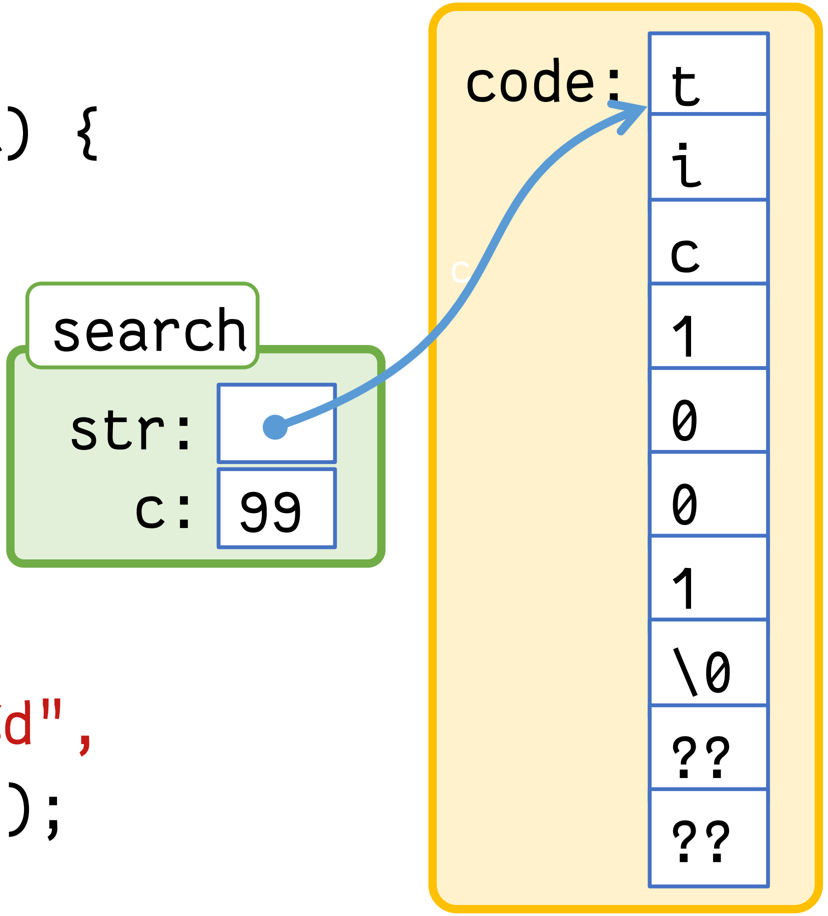
- `%s` will print until first occurrence of `'\0'`

Strings as Arguments

- Strings are just arrays of character
  - a.k.a char array
- Pass into function just like arrays
  - Since strings are terminated by '\0'
  - No need to pass in the size of string

Example: Passing Strings

```c
int search(char str[], char c) {
    for (i = 0; str[i] != '\0'; ++i) {
        if (str[i] == c)
            return i;
    }
    return -1;
}
char code[10] = "tic1001";
printf("Finding %c in %s at index %d",
       'c', code, search(code, 'c'));
```



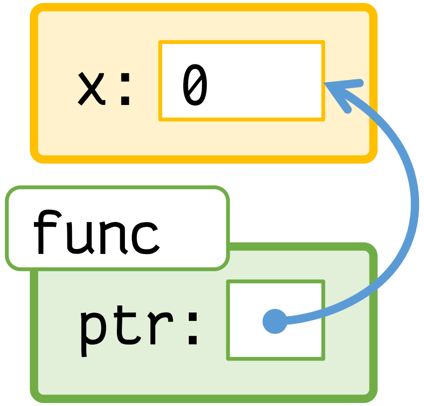
output

```
Finding c in tic1001 at index 2
```

Strings/Arrays as Pointers

Recall pass-by-pointers

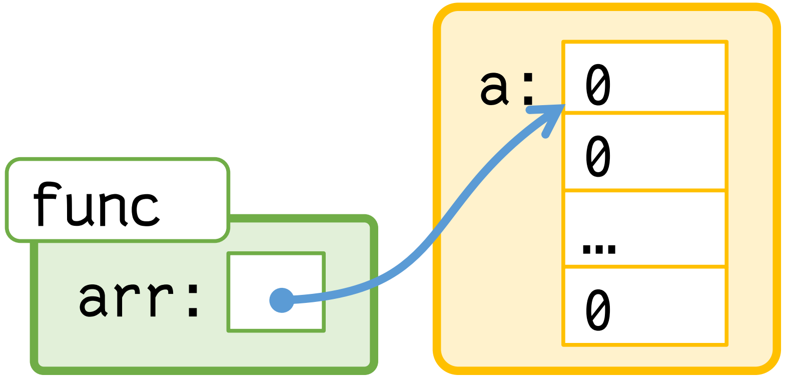
```
void func(int *ptr);
int x = 0;
func(&x);
```



Since strings/arrays are passed by pointers

– they can be written as pointers in the parameters

```
void func(char *arr);
char a[10] = {0};
func(a);
```



String Functions

- The Standard C library contains a number of string functions
  - #include <string.h>
- Four main useful functions
  - strlen
  - strcmp
  - strcpy
  - strcat

String Function - strlen

```
unsigned int strlen(const char s[]) {
    unsigned int i = 0;
    for (i = 0; s[i] != '\0'; ++i) ;
    return i;
}
char code[10] = "tic1001";
printf("%i", (int)strlen(code));
```

Returns the length of the string

- i.e. number of characters before the '\0' terminal
- const keyword prevents string s from being modified in the function

String Function - `strcmp`

```c
int strcmp(const char s[], const char t[]) {
    int i;
    for (i=0; s[i] != '\0' && s[i] == t[i]; ++i);
    return s[i] - t[i];
}
```

Compares two strings and returns

- negative if s < t
- zero if both s = t
- positive if s > t

More precisely,

- the difference between the first unequal characters

String Function - strcpy

```
char *strcpy(char dest[], const char src[]) {
    for (int i=0; dest[i] = src[i]; ++i) ;
    return dest;
}
```

How the heck this works?

Copies string src to a string dest

– Must ensure that dest has sufficient space to accommodate src

```
char code[10] = "tic1001";
char mod[8];
strcpy(mod, code);
printf("%s %s\n", code, mod);
```

output

```
tic1001 tic1001
```

String Function - strcat

```
char *strcat(char dest[], const char src[]) {
    int i = 0;
    for (; dest[i]; i++) ;
    for (; dest[i] = src[i]; i++) ;
    return dest;
}
```

Concatenates (join) string src to end of string dest

– Again, ensure dest has sufficient space to accommodate src

```
char s1[10] = "tic", s2[10] = "1001";
strcat(s1, s2);
printf("%s %s\n", s1, s2);
```

output

```
tic1001 1001
```

strcpy and strcat

C

Both functions return pointer to the modified string

– this allows string functions to be composed

```
char s1[10] = "tic";
char s2[10] = "1001";
char out[10];
strcat(strcpy(out, s1), s2);
              out
```

s1:	t	i	c	\0						
s2:	1	0	0	1	\0					
out:	t	i	c	1	0	0	1	\0	#	#

Summary

Arrays

- Must be declared/initialized with a predetermined size
- Use of subscripting/indexing to access individual elements
- Passed into functions by reference

Characters

- Single byte, unsigned integer

C-Strings

- Array of characters terminating with '\0'
- Operation of string functions depend on the position of '\0'