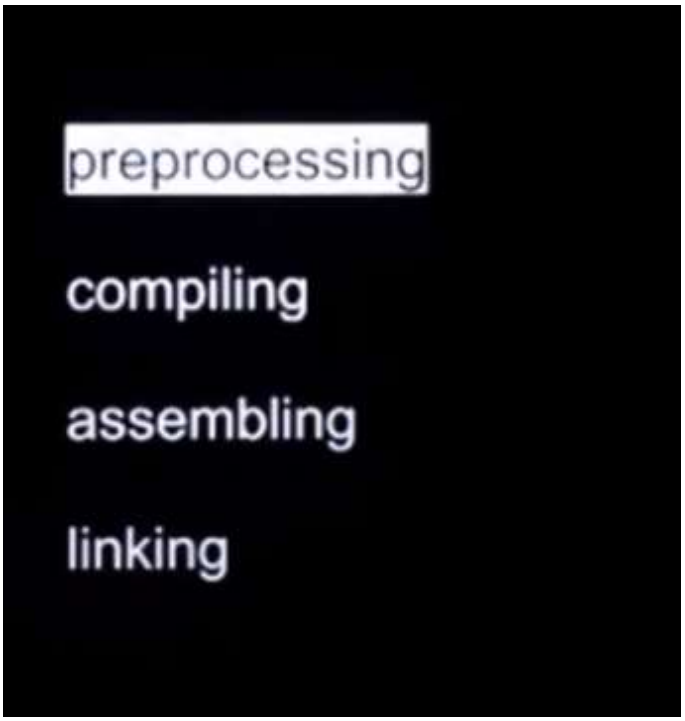
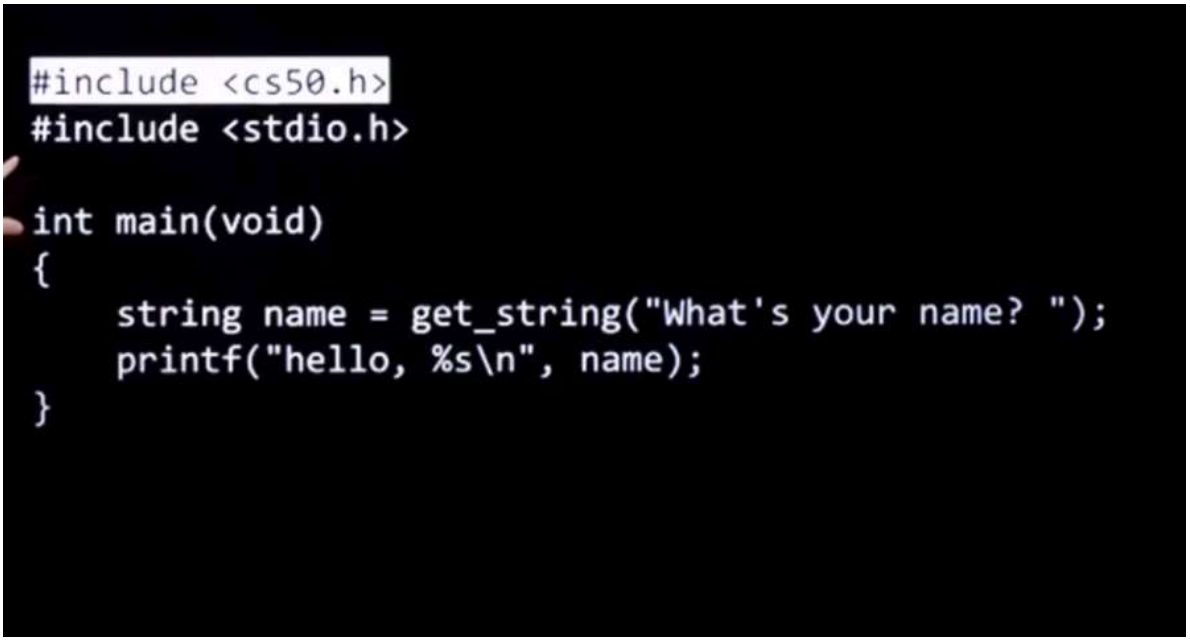
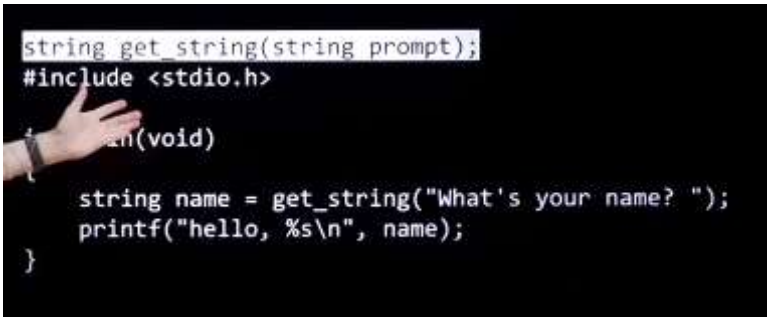
CS 50: Week-02-Arrays

Lecture 2





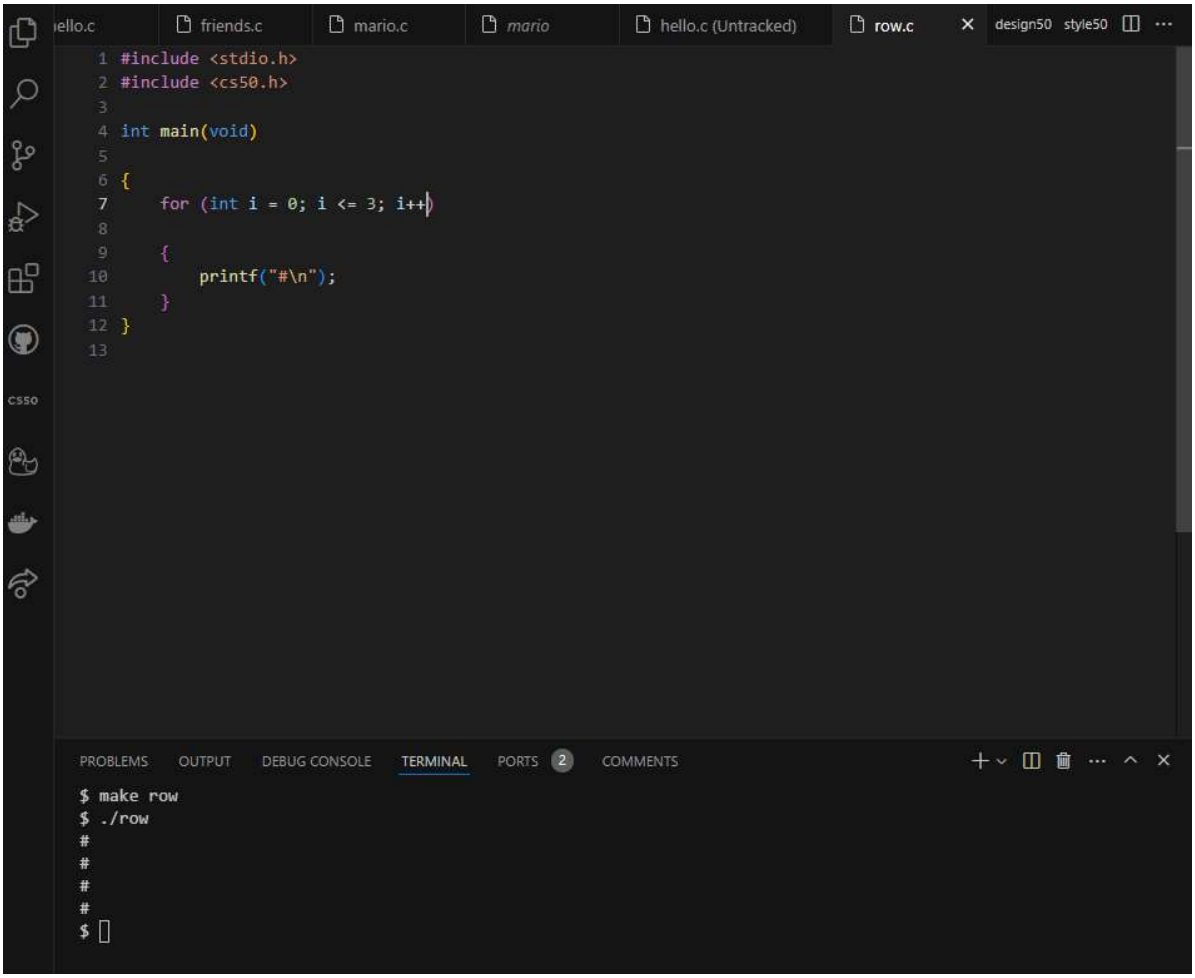


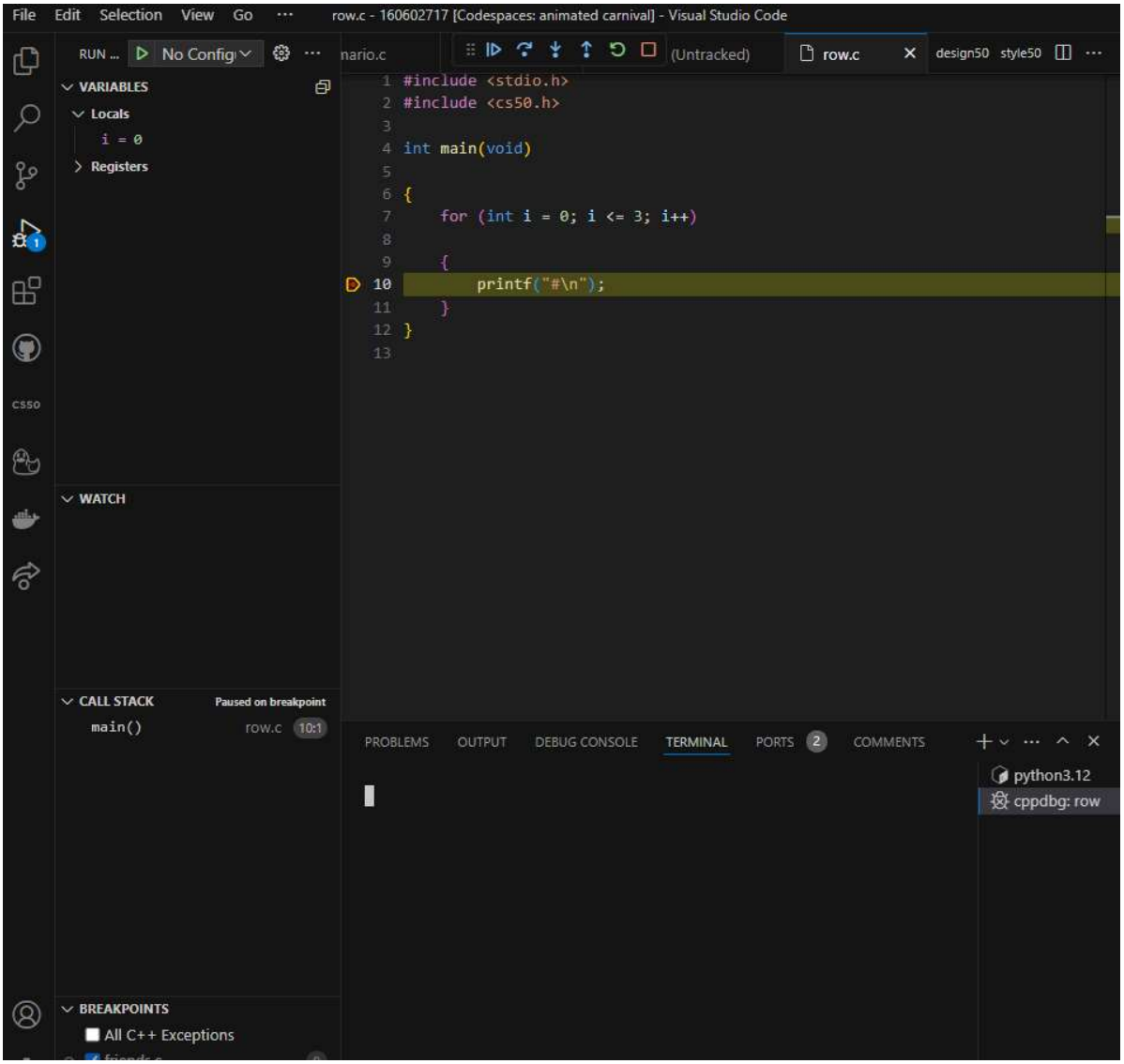

```c
string get_string(string prompt);
int printf(string format, ...);

int main(void)
{
    string name = get_string("What's your name? ");
    printf("hello, %s\n", name);
}
```

-

preprocessing
compiling
assembling
linking

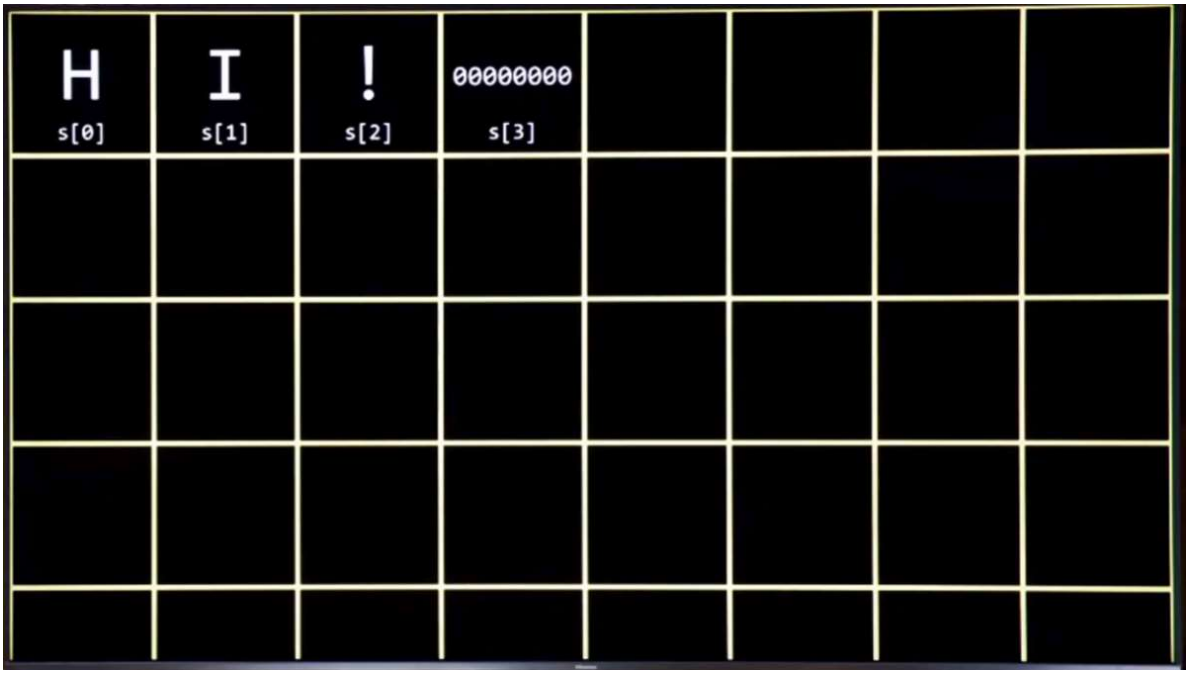

hello.c
cs50.c
stdio.c


ello.c
friends.c
mario.c
mario
hello.c (Untracked)
row.c
design50
style50
1 #include <stdio.h>
2 #include <cs50.h>
3
4 int main(void)
5
6 {
7 for (int i = 0; i <= 3; i++)
8
9 {
10 printf("#\n");
11 }
12 }
13
CSSO
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS 2
COMMENTS
$ make row
$ ./row
#
#
#
#
$

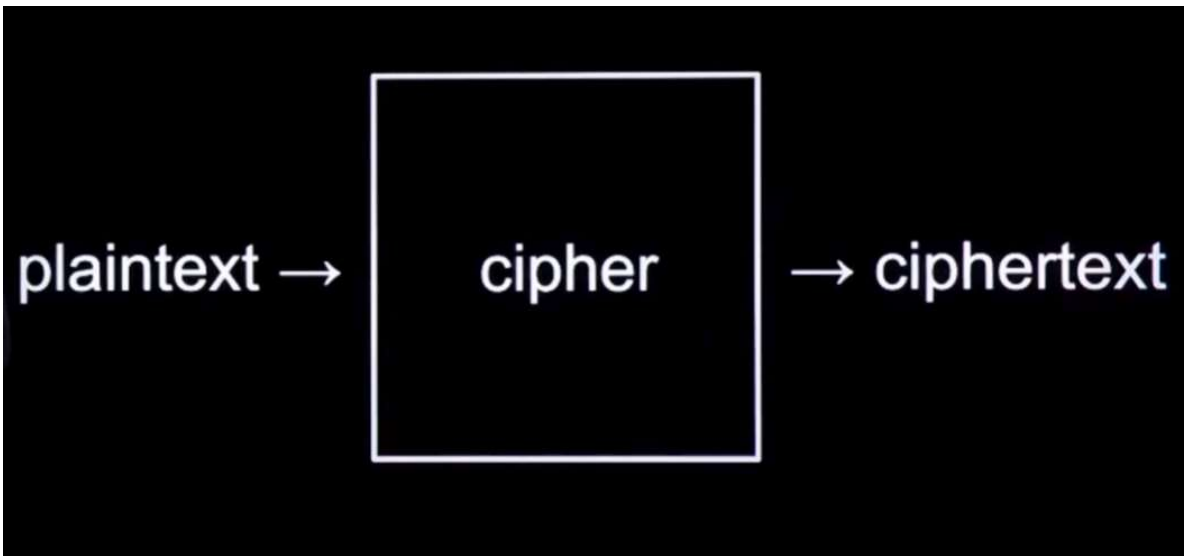

File Edit Selection View Go ··· row.c - 160602717 [Codespaces: animated carnival] - Visual Studio Code
RUN ... No Config
VARIABLES
Locals
i = 0
Registers
WATCH
CALL STACK
Paused on breakpoint
main()
row.c 10:1
BREAKPOINTS
All C++ Exceptions
nario.c
(Untracked)
row.c
design50 style50
1 #include <stdio.h>
2 #include <cs50.h>
3
4 int main(void)
5
6 {
7 for (int i = 0; i <= 3; i++)
8
9 {
10 printf("#\n");
11 }
12 }
13
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 2 COMMENTS
python3.12
cppdbg: row

`bool`	1 byte
`int`	4 bytes
`long`	8 bytes
`float`	4 bytes
`double`	8 bytes
`char`	1 byte
`string`	? bytes

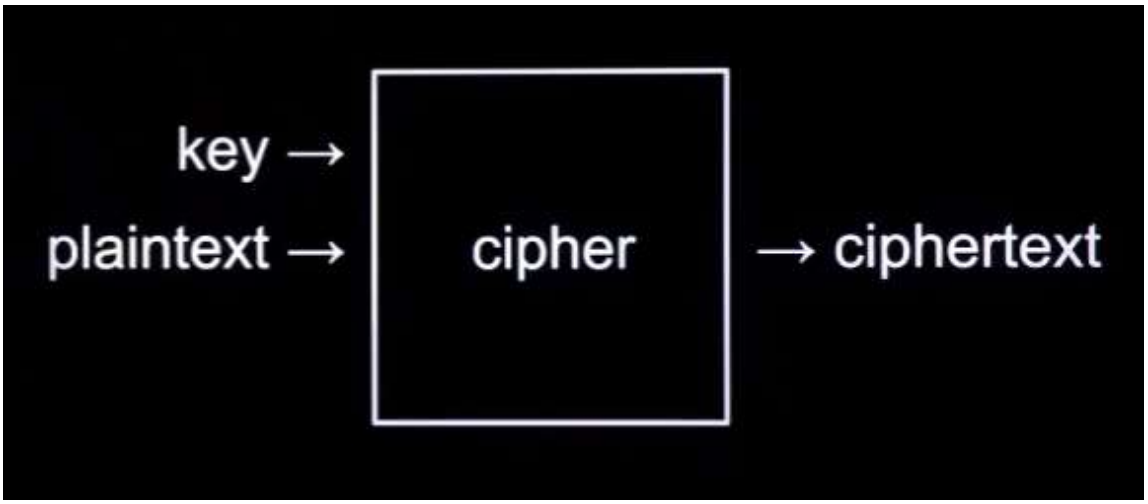
```
int scores[3];

scores[0] = 72;

scores[1] = 73;

scores[2] = 33;
```

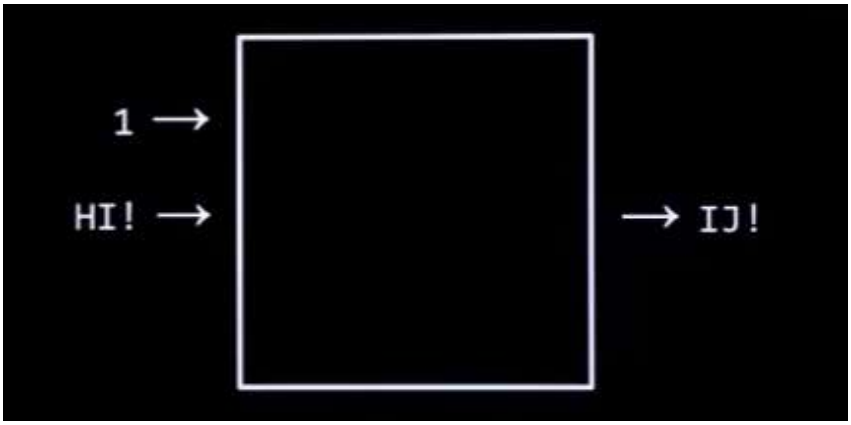


0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102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

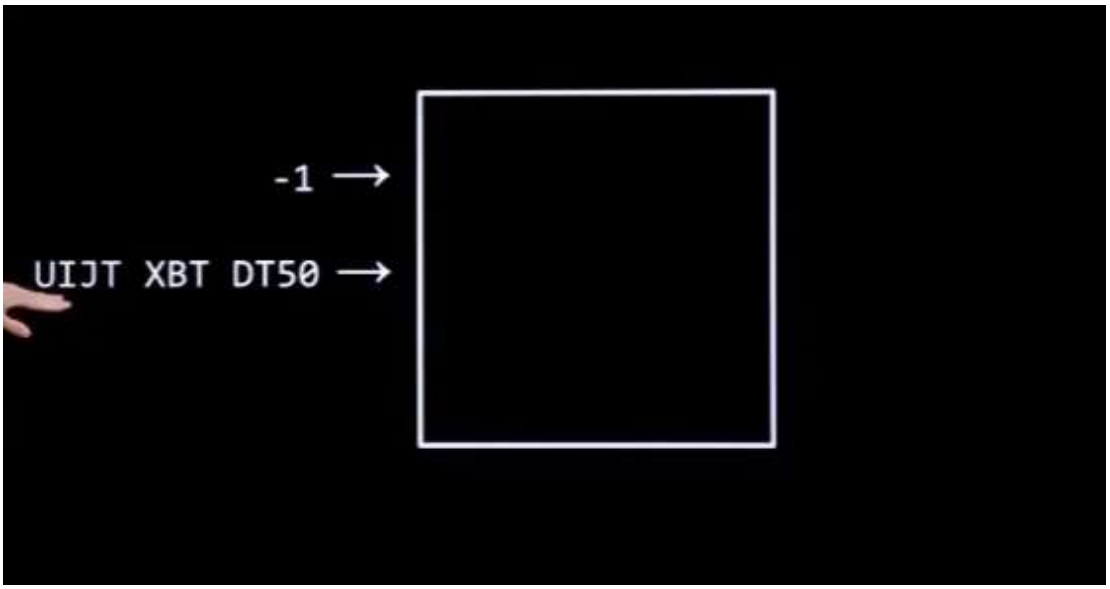


Key - second Input for decrepit



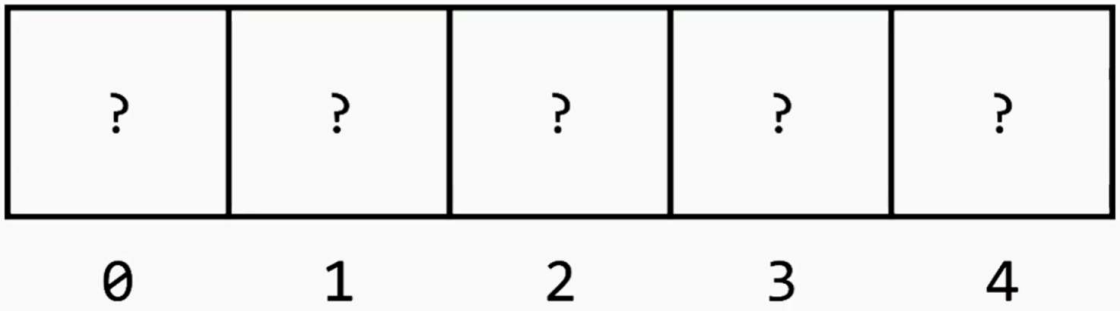


Decrypt




```
int hours[5];
```

hours



What are some examples of programs we've seen that take command-line arguments?

```
$ make mario
```

`int argc` ⟶ ARGument Count

`string argv[]` ⟶ ARGument Vector

```
$ ./caesar 13
```

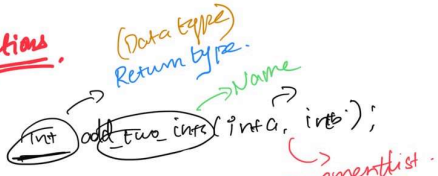
argv[0] argv[1]

`int` argc ⟶ ARGument `Count`

`string` argv[] ⟶ ARGument `Vector`

Functions

Functions.



Return type:
↳ Result of the function is return type!

Name
↳ To address function.

Argument list
↳ input Values / Parameters.

Example

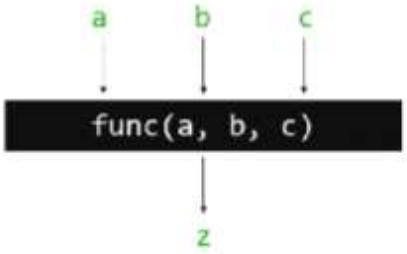
```
int add_two_ints(int a, int b).
int add_two_ints(int a, int b).
{
    int sum;          // declaration
    sum = a+b;        // calculate the sum
    return sum;       // give result back.
}
```

If functions do not have input
⇓
[Argument list = Void]

If functions do not have output
⇓
[Datatype = Void type]

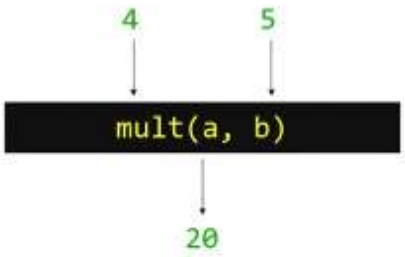
Functions

- What is a function?
  - A *black box* with a set of 0+ inputs and 1 output.



Functions

- What is a function?
  - A *black box* with a set of 0+ inputs and 1 output.



Functions

- Why call it a *black box*?
  - If we aren't writing the functions ourselves, we don't need to know the underlying implementation.

```
mult(a, b):
    output a * b
```

Functions

- Why call it a *black box*?
  - If we aren't writing the functions ourselves, we don't need to know the underlying implementation.

```
mult(a, b):
    output a * b
```

Functions

- Why call it a *black box*?
  - If we aren't writing the functions ourselves, we don't need to know the underlying implementation.

```
mult(a, b):
    set counter to 0
    repeat b times
        add a to counter
    output counter
```

Functions

- Why call it a *black box*?
  - If we aren't writing the functions ourselves, we don't need to know the underlying implementation.
  - That's part of the contract of using functions. The behavior is typically predictable based on that name. That's why most functions have clear, obvious(ish) names, and are well-documented.

Functions

- Why use functions?
  - Organization
    - Functions help break up a complicated problem into more manageable subparts.
  - Simplification
    - Smaller components tend to be easier to design, implement, and debug.
  - Reusability
    - Functions can be recycled; you only need to write them once, but can use them as often as you need!

Functions

- Function Declarations
  - The first step to creating a function is to declare it. This gives the compiler a heads-up that a user-written function appears in the code.
  - Function declarations should always go atop your code, before you begin writing `main()`.
  - There is a standard form that every function declaration follows.

Functions

- Function Declarations

```
return-type name(argument-list);
```

  - The `return-type` is what kind of variable the function will output.
  - The `name` is what you want to call your function.
  - The `argument-list` is the comma-separated set of inputs to your function, each of which has a type and a name.

Functions

- A function to add two integers.

```
int add_two_ints(int a, int b);
```

  - The sum of two integers is going to be an integer as well.
  - Given what this function does, make sure to give it an appropriate name.
  - There are two inputs to this function, and we need to give a name to each of them for purposes of the function. There's nothing important about these inputs as far as we know, so giving them simple names is okay.

Functions

- Function Definitions
  - The second step to creating a function is to define it. This allows for predictable behavior when the function is called with inputs.
  - Let's try to define `mult_two_reals()`, from a moment ago.

Functions

- A function definition looks **almost** identical to a function declaration, with a small change.

```
float mult_two_reals(float x, float y);
```

```
float mult_two_reals(float x, float y)
{
   float product = x * y;
   return product;
}
```

- How would you fill in this black box?

Functions

- A function definition looks **almost** identical to a function declaration, with a small change.

```
float mult_two_reals(float x, float y);


float mult_two_reals(float x, float y)
{
   return x * y;
}
```

Functions

- Now, take a moment and try to define `add_two_ints()`, from a moment ago.

```
int add_two_ints(int a, int b);

int add_two_ints(int a, int b)
{
    int sum;       // declare variable
    sum = a + b;   // calculate the sum
    return sum;    // give result back

}
```

Functions

- Now, take a moment and try to define `add_two_ints()`, from a moment ago.

```
int add_two_ints(int a, int b);

int add_two_ints(int a, int b)
{
    int sum = 0;
    if(a > 0)
       for(int i = 0; i < a; sum++, i++);
    else
       for(int i = a; i < 0; sum--, i++);
    if(b > 0)
       for(int i = 0; i < b; sum++, i++);
    else
       for(int i = b; i < 0; sum--, i++);
    return sum;
}
```

Functions

- Function Calls
  - Now that you've created a function, time to use it!
  - To call a function, simply pass it appropriate arguments and assign its return value to something of the correct type.
  - To illustrate this, let's have a look at `adder-1.c`

Variables and Scope

VARIABLE

Variable Scope .

Local Variable

```
int main(void)
{
    int return = triple(5);
}
int triple (int x)
{
    return x * 3;
}
```

Here X is **local** to the function triple( ). no other function can refer to that variable, not even main
result is local to the main().

Global Variable .

```
# include <stdio.h>
float global = 0.5050;
int main(void);
{   tripple( );
    printf("%f", global );
}   void tripel (void)
{   global *= 3;
}
```

Global variables exist to if a variable is declared outside of all functions. **Any** functions may refer to it.

Variable Scope

- Why does this distinction matter? For the most part, local variables in C are **passed by value** in function calls.
- When a variable is passed by value, the **callee** receives a copy of the passed variable, not the variable itself.
- That means that the variable in the **caller** is unchanged unless overwritten.

Arrays

Arrays

Arrays	Post Office Boxes
An **array** is a block of contiguous space in memory...	A **mail bank** is a large space on the wall of the post office...
...which has been partitioned into small, identically-sized blocks of space called **elements**...	...which has been partitioned into small, identically-sized blocks of space called **post office boxes**...
...each of which can store a certain amount of **data**...	...each of which can hold a certain amount of **mail**...
...all of the same data type such as **int** or **char**...	...all of a similar type such as **letters** or **small packages**...
...and which can be accessed directly by an **index**.	...and which can be accessed directly by a **mailbox number**.

Arrays

- Array declarations

```
type name[size];
```

  - The `type` is what kind of variable each element of the array will be.
  - The `name` is what you want to call your array.
  - The `size` is how many elements you would like your array to contain.

Arrays

- If you think of a single element of an array of type `data-type` the same as you would any other variable of type `data-type` (which, effectively, it is) then all the familiar operations make sense.

```
bool truthtable[10];

truthtable[2] = false;
if(truthtable[7] == true)
{
   printf("TRUE!\n");
}
truthtable[10] = true;
```

Arrays

- When declaring and initializing an array simultaneously, there is a special syntax that may be used to fill up the array with its starting values.

```
// instantiation syntax
bool truthtable[3] = { false, true, true };

// individual element syntax
bool truthtable[3];
truthtable[0] = false;
truthtable[1] = true;
truthtable[2] = true;
```

Arrays

- Arrays can consist of more than a single dimension. You can have as many size specifiers as you wish.

```
bool battleship[10][10];
```

- You can choose to think of this as either a 10x10 grid of cells.
  - In memory though, it's really just a 100-element one-dimensional array.
  - Multi-dimensional arrays are great **abstractions** to help visualize game boards or other complex representations.

Arrays

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

for(int j = 0; j < 5; j++)
{
    bar[j] = foo[j];
}
```

Arrays

- Recall that most variables in C are **passed by value** in function calls.
- Arrays do not follow this rule. Rather, they are **passed by reference**. The callee receives the actual array, not a copy of it.
  - What does that mean when the callee manipulates elements of the array?
- For now, we'll gloss over why arrays have this special property, but we'll return to it soon enough!

Command Line Arguments

Command-Line Arguments

- So far, all of your programs have begun pretty much the same way.

```
int main(void)
{
```

- Since we've been collecting user input through in-program prompts, we haven't needed to modify this declaration of `main()`.

- If we want the user to provide data to our program before the program starts running, we need a new form.

Command-Line Arguments

- `argc` (argument count)
  - This integer-type variable will store the **number** of command-line arguments the user typed when the program was executed.

command	argc
`./greedy`	1
`./greedy 1024 cs50`	3

Command-Line Arguments

- argv (argument vector)
  - This array of strings stores, one string per element, the actual text the user typed at the command-line when the program was executed.
  - The first element of argv is always found at argv[0]. The last element of argv is always found at argv[argc-1].
    - Do you see why?

Command-Line Arguments

- argv (argument vector)
  - Let's assume the user executes the greedy program as follows

```
./greedy 1024 cs50
```

argv indices	argv contents
argv[0]	"./greedy"
argv[1]	"1024"
argv[2]	"cs50"
argv[3]	???