

# This is CS50

CS50's Introduction to Computer Science

OpenCourseWare

Donate ↗ (<https://cs50.harvard.edu/donate>)

David J. Malan (<https://cs.harvard.edu/malan/>)

malan@harvard.edu

 (<https://www.facebook.com/dmalan>)  (<https://github.com/dmalan>)   
 (<https://www.linkedin.com/in/malan/>)  (<https://www.reddit.com/user/davidjmalan>)  ([@davidjmalan](https://www.threads.net/@davidjmalan))  (<https://twitter.com/davidjmalan>)

## Lecture 2

---

- [Welcome!](#)
- [Reading Levels](#)
- [Compiling](#)
- [Debugging](#)
- [Arrays](#)
- [Strings](#)
- [String Length](#)
- [Command-Line Arguments](#)
- [Exit Status](#)
- [Cryptography](#)
- [Summing Up](#)

## Welcome!

---

- In our previous session, we learned about C, a text-based programming language.
- This week, we are going to take a deeper look at additional building-blocks that will

support our goals of learning more about programming from the bottom up.

- Fundamentally, in addition to the essentials of programming, this course is about problem-solving. Accordingly, we will also focus further on how to approach computer science problems.
- By the end of the course, you will learn how to use these aforementioned building blocks to solve a whole host of computer science problems.

## Reading Levels

---

- One of the real-world problems we will solve in this course is understanding reading levels.
- With the help of some of your peers, we presented readings at various reading levels.
- We will be quantifying reading levels this week as one of your many programming challenges.

## Compiling

---

- *Encryption* is the act of hiding plain text from prying eyes. *Decrypting*, then, is the act of taking an encrypted piece of text and returning it to a human-readable form.
- An encrypted piece of text may look like the following:

```
U I J T J T D T 5 0
```

- Recall that last week, you learned about a *compiler*, a specialized computer program that converts *source code* into *machine code* that can be understood by a computer.
- For example, you might have a computer program that looks like this:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

- A compiler will take the above code and turn it into the following machine code:

```
01111111 01000101 01001100 01000110 00000010 00000001 00000001 00000000  
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  
00000010 00000000 00111110 00000000 00000001 00000000 00000000 00000000  
10110000 00000101 01000000 00000000 00000000 00000000 00000000 00000000  
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  
11010000 00010011 00000000 00000000 00000000 00000000 00000000 00000000  
00000000 00000000 00000000 00000000 01000000 00000000 00111000 00000000  
00001001 00000000 01000000 00000000 00100100 00000000 00100001 00000000  
00000110 00000000 00000000 00000000 00000101 00000000 00000000 00000000  
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  
01000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000  
01000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000  
11111000 00000001 00000000 00000000 00000000 00000000 00000000 00000000  
11111000 00000001 00000000 00000000 00000000 00000000 00000000 00000000  
00001000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  
00000011 00000000 00000000 00000000 00000100 00000000 00000000 00000000  
00111000 00000010 00000000 00000000 00000000 00000000 00000000 00000000  
...
```

- VS Code, the programming environment provided to you as a CS50 student, utilizes a compiler called `clang` or *c language*.
- You can enter the following into the terminal window to compile your code: `clang -o hello hello.c`.
- *Command-line arguments* are provided at the command line to `clang` as `-o hello hello.c`.
- Running `./hello` in the terminal window, your program runs as intended.
- Consider the following code from last week:

```
#include <cs50.h>
#include <stdio.h>

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

- To compile this code, you can type `clang -fhello hello.c -lcs50`.
- If you were to type `make hello`, it runs a command that executes clang to create an output file that you can run as a user.
- VS Code has been pre-programmed such that `make` will run numerous command line arguments along with clang for your convenience as a user.
- While the above is offered as an illustration, such that you can understand more deeply the process and concept of compiling code, using `make` in CS50 is perfectly fine and the expectation!
- Compiling involves four major steps, including the following:

- First, *preprocessing* is where the header files in your code, designated by a `#` (such as `#include <cs50.h>`) are effectively copied and pasted into your file. During this step, the code from `cs50.h` is copied into your program. Similarly, just as your code contains `#include <stdio.h>`, code contained within `stdio.h` somewhere on your computer is copied to your program. This step can be visualized as follows:

```
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);
}
```

- Second, *compiling* is where your program is converted into assembly code. This step can be visualized as follows:

```
...
main:
    .cfi_startproc
# BB#0:
    pushq    %rbp
.Ltmp0:
    .cfi_def_cfa_offset 16
.Ltmp1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
.Ltmp2:
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    xorl    %eax, %eax
    movl    %eax, %edi
    movabsq   $.L.str, %rsi
    movb    $0, %al
    callq    get_string
    movabsq   $.L.str.1, %rdi
    movq    %rax, -8(%rbp)
    movq    -8(%rbp), %rsi
    movb    $0, %al
    callq    printf
    ...
```

- Third, *assembling* involves the compiler converting your assembly code into machine code. This step can be visualized as follows:

```
0111111010001010100110001000110
00000010000000010000000100000000
```

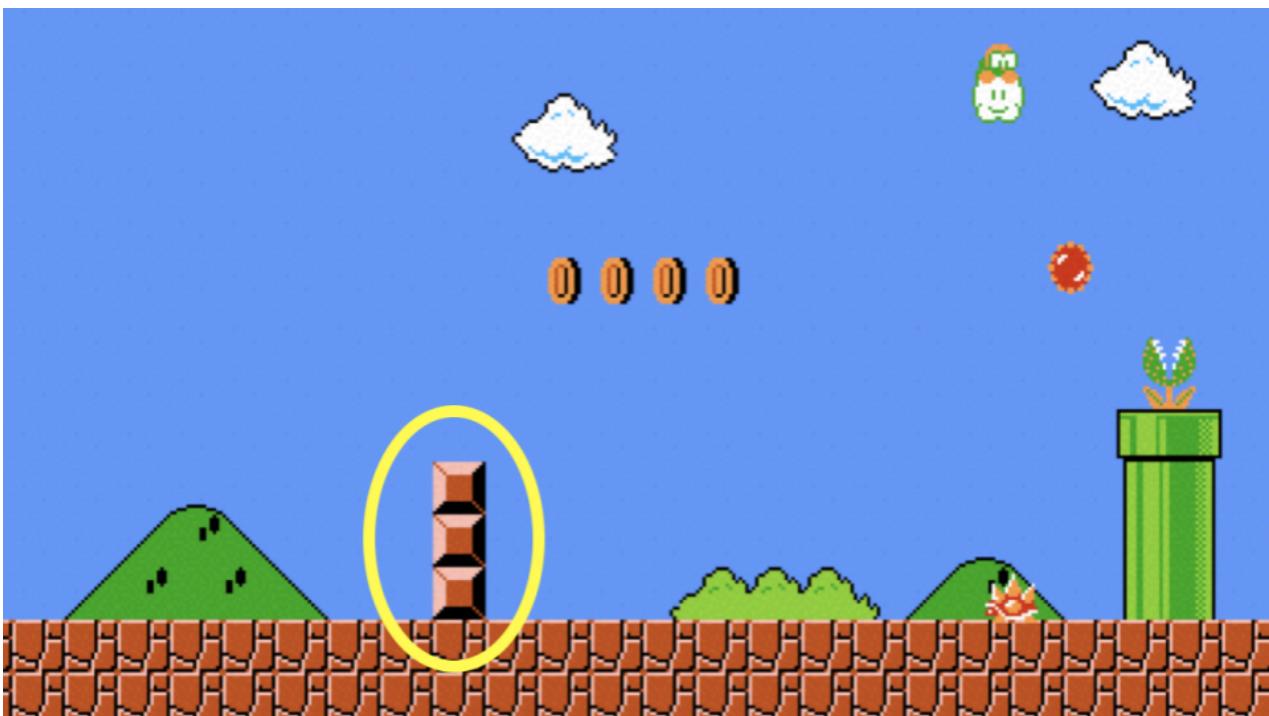
utputted.

01111111010001010100110001000110	01111111010001010100110001000110	00101111011011000110100101100010
000000100000001000000010000000	000000100000001000000010000000	0010001100101100111001101011111
00000000000000000000000000000000	00000000000000000000000000000000	00101110001101100010000001011111
00000000000000000000000000000000	00000000000000000000000000000000	01101010111001101110010001011111
00000000100000000001111100000000	000000110000000001111100000000	011011000110100101100010001011111
00000001000000000000000000000000	00000010000000000000000000000000	01111000001110000110110010111111
00000000000000000000000000000000	11000000000011110000000000000000	00110110001101000010110101101100
00000000000000000000000000000000	00000000000000000000000000000000	01101001011011000111010101111100
00000000000000000000000000000000	01000000000000000000000000000000	0010110101100111011011001110101
00000000000000000000000000000000	00000000000000000000000000000000	00101111011011000110100101100018
10100000000001000000000000000000	00101000001100100000000000000000	0110001101011110110111001101111
00000000000000000000000000000000	00000000000000000000000000000000	01101110011100110110110001100001
00000000000000000000000000000000	00000000000000000000000000000000	01100010011001011010000001011111
01000000000000000000000000000000	01000000000000000000000000000000	01100001001000000100000010000001
00000000000000000000000000000000	00000111000000000100000000000000	01010001101011110100111001000101
00001010000000000000000000000000	00011100000000000110010000000000	01000101010001000100010101000100
0101010101001000100100111100101	00000001000000000000000000000000	00100000001010000010000000101111
01001000100000111110110000010000	00000010100000000000000000000000	01101100011010010110001000101111
00110001110000001000100111000111	00000000000000000000000000000000	01111000001110000011011001011111
01001000101111100000000000000000	00000000000000000000000000000000	00110110001101000010110101101100
00000000000000000000000000000000	00000000000000000000000000000000	01010001011011000111010101111000
00000000000000000000000000000000	00000000000000000000000000000000	001011001010000000110110000000001
11101000000000000000000000000000	00000000000000000000000000000000	00101111011011000011001000010101
00000000000000000000000000000000	00000000000000000000000000000000	01010001010001000110100000011001
00000000000000000000000000000000	01011100001001010000000000000000	01111000001010101111000000111000
00000000000000000000000000000000	00000000000000000000000000000000	00101011000010101001011000000110100
...	...	...

# Debugging

- Everyone will make mistakes while coding.

- Debugging is the process of locating and removing bugs from your code.
- One of the debugging techniques you will use during this course to debug your code is called *rubber duck debugging*, where you can talk to an inanimate object (or yourself) to help think through your code and why it is not working as intended. When you are having challenges with your code, consider how speaking out loud to, quite literally, a rubber duck about the code problem. If you'd rather not talk to a small plastic duck, you are welcome to speak to a human near you!
- We have created the CS50 Duck and [CS50.ai \(https://cs50.ai\)](https://cs50.ai) as tools that can help you debug your code.
- Consider the following image from last week:



- Consider the following code that has a bug purposely inserted within it:

```
// Buggy example for printf

#include <stdio.h>

int main(void)
{
    for (int i = 0; i <= 3; i++)
    {
        printf("#\n");
    }
}
```

Notice that this code prints four blocks instead of three.

- Type `code buggy0.c` into the terminal window and write the above code.
- Running this code, four bricks appear instead of the intended three.
- `printf` is a very useful way of debugging your code. You could modify your code as

follows:

```
// Buggy example for printf

#include <stdio.h>

int main(void)
{
    for (int i = 0; i <= 3; i++)
    {
        printf("i is %i\n", i);
        printf("#\n");
    }
}
```

Notice how this code outputs the value of `i` during each iteration of the loop such that we can debug our code.

- Running this code, you will see numerous statements, including `i is 0`, `i is 1`, `i is 2`, and `i is 3`. Seeing this, you might realize that further code needs to be corrected as follows:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        printf("#\n");
    }
}
```

Notice the `<=` has been replaced with `<`.

- This code can be further improved as follows:

```
// Buggy example for debug50

#include <cs50.h>
#include <stdio.h>

void print_column(int height);

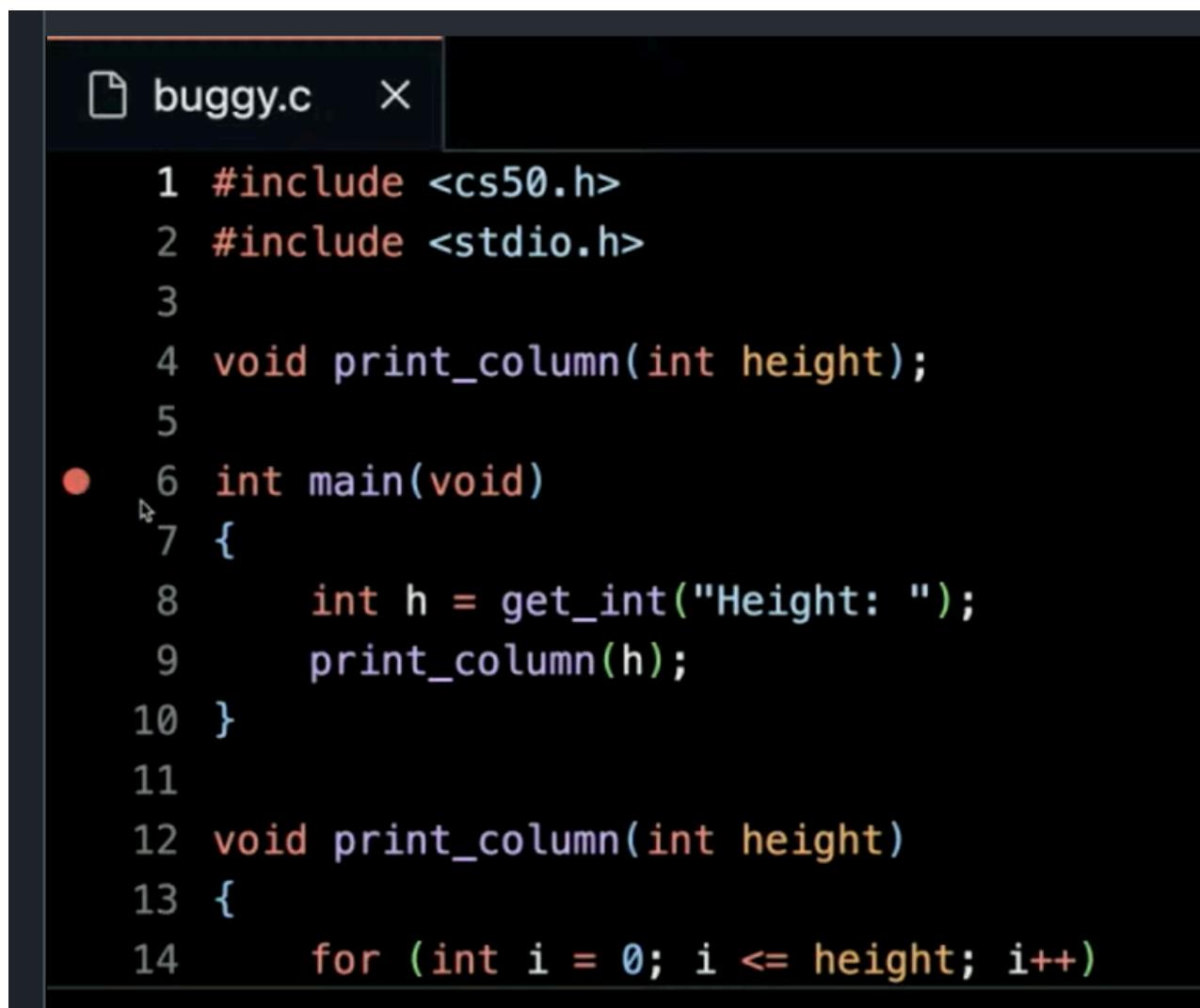
int main(void)
{
    int h = get_int("Height: ");
    print_column(h);
}

void print_column(int height)
{
    for (int i = 0; i <= height; i++)
    {
        printf("#\n");
    }
}
```

```
}
```

Notice that compiling and running this code still results in a bug.

- To address this bug, we will use a new tool.
- A second tool in debugging is called a *debugger*, a software tool created by programmers to help track down bugs in code.
- In VS Code, a preconfigured debugger has been provided to you.
- To utilize this debugger, first set a *breakpoint* by clicking to the left of a line of your code, just to the left of the line number. When you click there, you will see a red dot appearing. Imagine this as a stop sign, asking the compiler to pause so that you can consider what's happening in this part of your code.



The screenshot shows a code editor window for a file named "buggy.c". The code contains a main function that prompts the user for a height and prints a column of that height. A red circular breakpoint marker is placed at the start of the main function. The code is as follows:

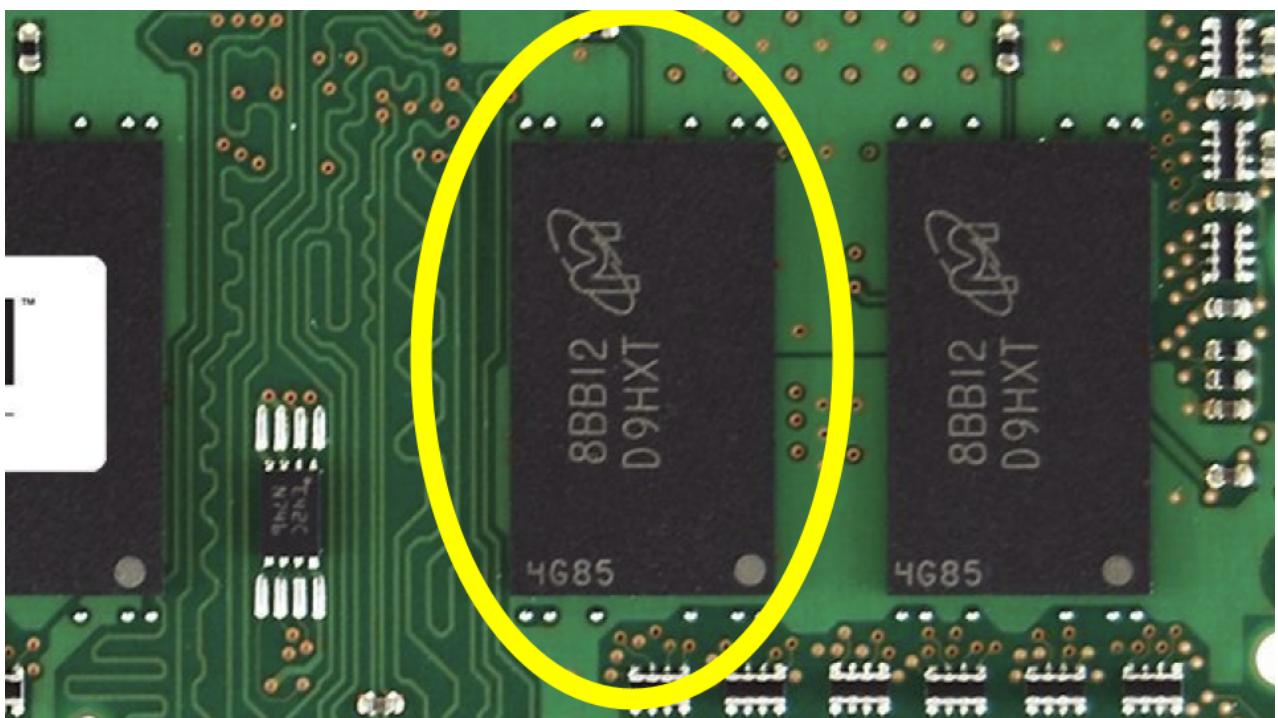
```
1 #include <cs50.h>
2 #include <stdio.h>
3
4 void print_column(int height);
5
6 int main(void)
7 {
8     int h = get_int("Height: ");
9     print_column(h);
10}
11
12 void print_column(int height)
13{
14    for (int i = 0; i <= height; i++)
```

- Second, run `debug50 ./buggy0`. You will notice that after the debugger comes to life and a line of your code will illuminate in a gold-like color. Quite literally, the code has *paused* at this line of code. Notice in the top left corner how all local variables are being displayed, including `h`, which currently does not have a value. At the top of your window, you can click the `step over` button, and it will keep moving through your code. Notice how the value of `h` increases.
- While this tool will not show you where your bug is, it will help you slow down and

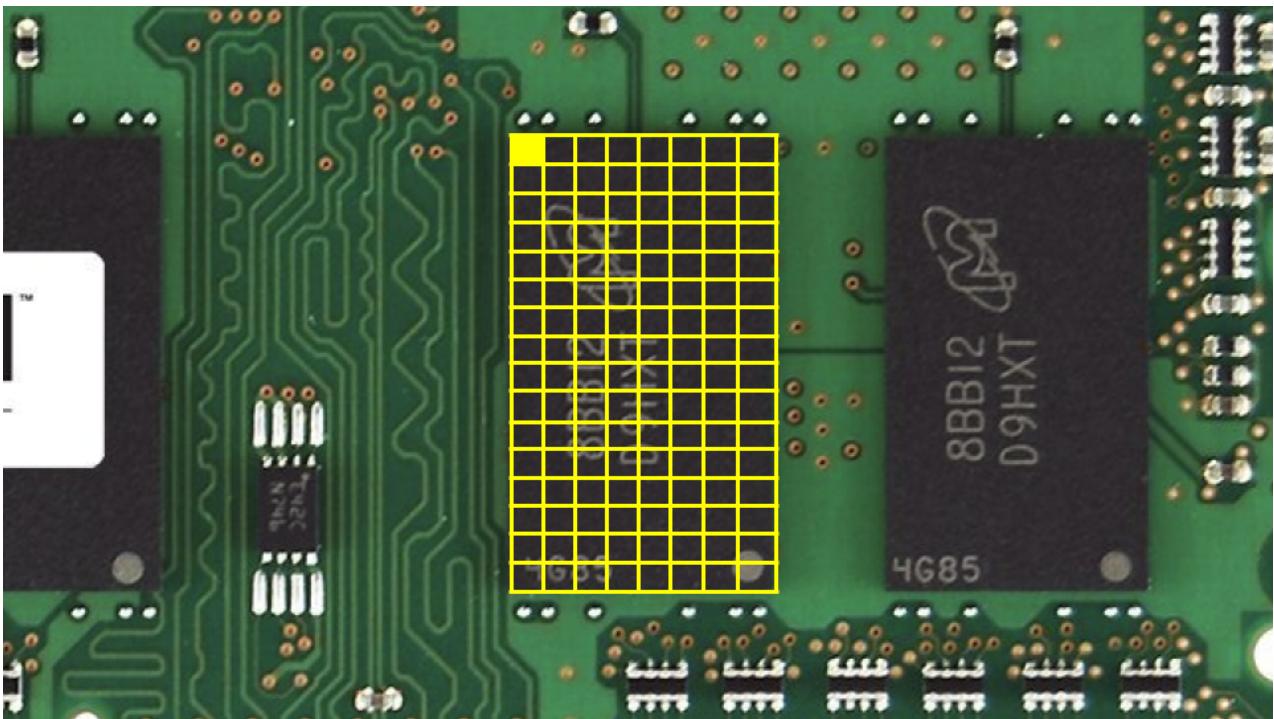
see how your code is running step by step. You can use `step into` as a way to look further into the details of your buggy code.

## Arrays

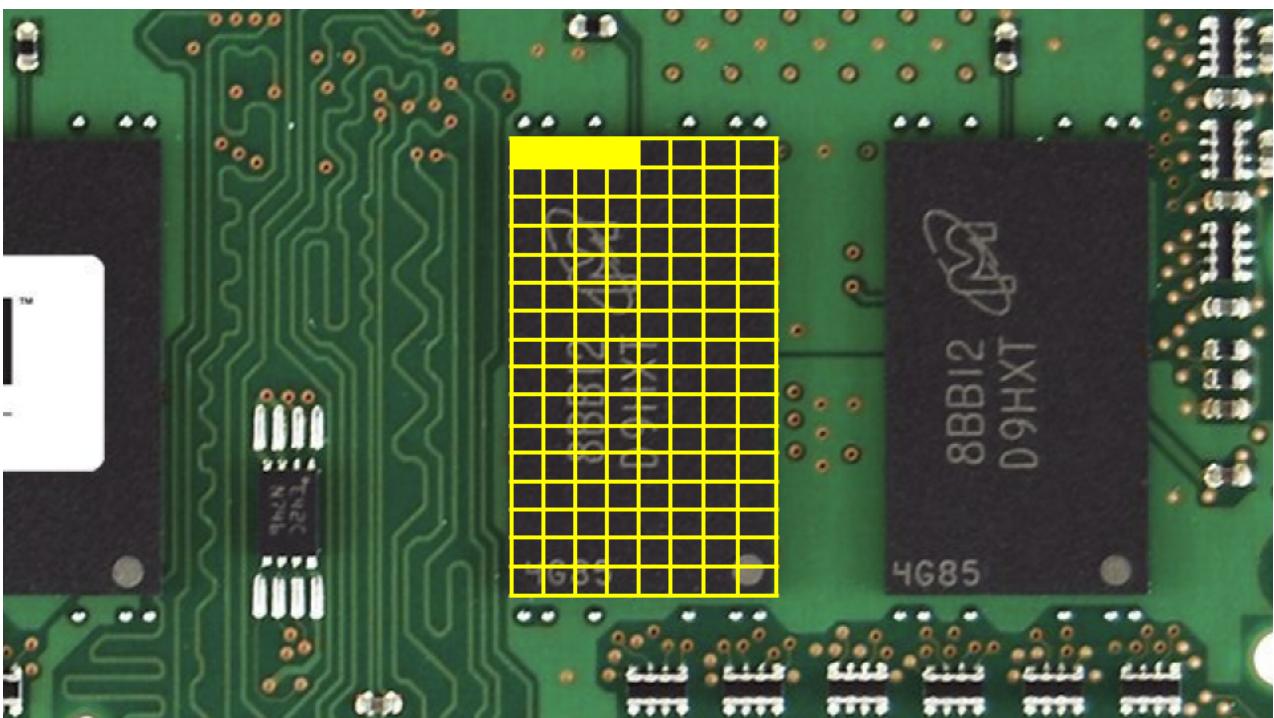
- In Week 0, we talked about *data types* such as `bool`, `int`, `char`, `string`, etc.
- Each data type requires a certain amount of system resources:
  - `bool` 1 byte
  - `int` 4 bytes
  - `long` 8 bytes
  - `float` 4 bytes
  - `double` 8 bytes
  - `char` 1 byte
  - `string` ? bytes
- Inside of your computer, you have a finite amount of memory available.



- Physically, on the memory of your computer, you can imagine how specific types of data are stored on your computer. You might imagine that a `char`, which only requires 1 byte of memory, may look as follows:



- Similarly, an `int`, which requires 4 bytes, might look as follows:



- We can create a program that explores these concepts. Inside your terminal, type `code scores.c` and write code as follows:

```
// Averages three (hardcoded) numbers

#include <stdio.h>

int main(void)
{
    // Scores
    int score1 = 72;
    int score2 = 73;
    int score3 = 33;
```

```

    // Print average
    printf("Average: %f\n", (score1 + score2 + score3) / 3.0);
}

```

Notice that the number on the right is a floating point value of `3.0`, so that the calculation is rendered as a floating point value in the end.

- Running `make scores`, the program runs.
- You can imagine how these variables are stored in memory:



- Arrays are a sequence of values that are stored back to back in memory.
- `int scores[3]` is a way of telling the compiler to provide you three back-to-back places in memory of size `int` to store three `scores`. Considering our program, you can revise your code as follows:

```

// Averages three (hardcoded) numbers using an array

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Scores
    int scores[3];
    scores[0] = 72;
    scores[1] = 73;
    scores[2] = 33;

    // Print average
    printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0
}

```

Notice that `score[0]` examines the value at this location of memory by `indexing`

into the array called `scores` at location `0` to see what value is stored there.

- You can see how, while the above code works, there is still an opportunity for improving our code. Revise your code as follows:

```
// Averages three numbers using an array and a loop

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Get scores
    int scores[3];
    for (int i = 0; i < 3; i++)
    {
        scores[i] = get_int("Score: ");
    }

    // Print average
    printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0
}
```

Notice how we index into `scores` by using `scores[i]` where `i` is supplied by the `for` loop.

- We can simplify or *abstract away* the calculation of the average. Modify your code as follows:

```
// Averages three numbers using an array, a constant, and a helper fu

#include <cs50.h>
#include <stdio.h>

// Constant
const int N = 3;

// Prototype
float average(int length, int array[]);

int main(void)
{
    // Get scores
    int scores[N];
    for (int i = 0; i < N; i++)
    {
        scores[i] = get_int("Score: ");
    }

    // Print average
    printf("Average: %f\n", average(N, scores));
}

float average(int length, int array[])
{
```

```

// Calculate average
int sum = 0;
for (int i = 0; i < length; i++)
{
    sum += array[i];
}
return sum / (float) length;
}

```

Notice that a new function called `average` is declared. Further, notice that a `const` or constant value of `N` is declared. Most importantly, notice how the `average` function takes `int array[]`, which means that the compiler passes an array to this function.

- Not only can arrays be containers: They can be passed between functions.

## Strings

- A `string` is simply an array of variables of type `char`: an array of characters.
- To explore `char` and `string`, type `code hi.c` in the terminal window and write code as follows:

```

// Prints chars

#include <stdio.h>

int main(void)
{
    char c1 = 'H';
    char c2 = 'I';
    char c3 = '!';

    printf("%c%c%c\n", c1, c2, c3);
}

```

Notice that this will output a string of characters.

- Similarly, make the following modification to your code:

```

#include <stdio.h>

int main(void)
{
    char c1 = 'H';
    char c2 = 'I';
    char c3 = '!';

    printf("%i %i %i\n", c1, c2, c3);
}

```

Notice that that ASCII codes are printed by replacing `%c` with `%i`.

- Considering the following image, you can see how a string is an array of characters that begins with the first character and ends with a special character called a **NUL character**:

H	I	!	\0				
s[0]	s[1]	s[2]	s[3]				

- Imagining this in decimal, your array would look like the following:

72	73	33	0				
s[0]	s[1]	s[2]	s[3]				

- To further understand how a **string** works, revise your code as follows:

```
// Treats string as array

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
```

```
    printf("%c%c%c\n", s[0], s[1], s[2]);  
}
```

Notice how the `printf` statement presents three values from our array called `s`.

- As before, we can replace `%c` with `%i` as follows:

```
// Prints string's ASCII codes, including NUL  
  
#include <cs50.h>  
#include <stdio.h>  
  
int main(void)  
{  
    string s = "HI!";  
    printf("%i %i %i %i\n", s[0], s[1], s[2], s[3]);  
}
```

Notice that this prints the string's ASCII codes, including NUL.

- Let's imagine we want to say both `HI!` and `BYE!`. Modify your code as follows:

```
// Multiple strings  
  
#include <cs50.h>  
#include <stdio.h>  
  
int main(void)  
{  
    string s = "HI!";  
    string t = "BYE!";  
  
    printf("%s\n", s);  
    printf("%s\n", t);  
}
```

Notice that two strings are declared and used in this example.

- You can visualize this as follows:

H s[0]	I s[1]	! s[2]	\0 s[3]	B t[0]	Y t[1]	E t[2]	! t[3]
\0 t[4]							

- We can further improve this code. Modify your code as follows:

```
// Array of strings

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string words[2];

    words[0] = "HI!";
    words[1] = "BYE!";

    printf("%s\n", words[0]);
    printf("%s\n", words[1]);
}
```

Notice that both strings are stored within a single array of type `string`.

- We can consolidate our two strings into an array of strings.

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string words[2];

    words[0] = "HI!";
    words[1] = "BYE!";

    printf("%c%c%c\n", words[0][0], words[0][1], words[0][2]);
    printf("%c%c%c%c\n", words[1][0], words[1][1], words[1][2], words
}
```

Notice that an array of `words` is created. It is an array of strings. Each word is stored in `words`.

## String Length

- A common problem within programming, and perhaps C more specifically, is to discover the length of an array. How could we implement this in code? Type `code length.c` in the terminal window and code as follows:

```
// Determines the length of a string

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // Prompt for user's name
    string name = get_string("Name: ");

    // Count number of characters up until '\0' (aka NUL)
    int n = 0;
    while (name[n] != '\0')
    {
        n++;
    }
    printf("%i\n", n);
}
```

Notice that this code loops until the NUL character is found.

- This code can be improved by abstracting away the counting into a function as follows:

```
// Determines the length of a string using a function

#include <cs50.h>
#include <stdio.h>

int string_length(string s);

int main(void)
{
    // Prompt for user's name
    string name = get_string("Name: ");
    int length = string_length(name);
    printf("%i\n", length);
}

int string_length(string s)
{
    // Count number of characters up until '\0' (aka NUL)
    int n = 0;
    while (s[n] != '\0')
```

```
    {
        n++;
    }
    return n;
}
```

Notice that a new function called `string_length` counts characters until NUL is located.

- Since this is such a common problem within programming, other programmers have created code in the `string.h` library to find the length of a string. You can find the length of a string by modifying your code as follows:

```
// Determines the length of a string using a function

#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Prompt for user's name
    string name = get_string("Name: ");
    int length = strlen(name);
    printf("%i\n", length);
}
```

Notice that this code uses the `string.h` library, declared at the top of the file. Further, it uses a function from that library called `strlen`, which calculates the length of the string passed to it.

- Our code can stand on the shoulders of programmers who came before and use libraries they created.
- `ctype.h` is another library that is quite useful. Imagine we wanted to create a program that converted all lowercase characters to uppercase ones. In the terminal window, type `code uppercase.c` and write code as follows:

```
// Uppercases a string

#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string s = get_string("Before: ");
    printf("After: ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
        {
            printf("%c", s[i] - 32);
        }
    }
}
```

```

    else
    {
        printf("%c", s[i]);
    }
    printf("\n");
}

```

Notice that this code *iterates* through each value in the string. The program looks at each character. If the character is lowercase, it subtracts the value 32 from it to convert it to uppercase.

- Recalling our previous work from last week, you might remember this ASCII values chart:

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

- When a lowercase character has  subtracted from it, it results in an uppercase

version of that same character.

- While the program does what we want, there is an easier way using the `ctype.h` library. Modify your program as follows:

```
// Uppercases string using ctype library (and an unnecessary condition)

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string s = get_string("Before: ");
    printf("After: ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        if (islower(s[i]))
        {
            printf("%c", toupper(s[i]));
        }
        else
        {
            printf("%c", s[i]);
        }
    }
    printf("\n");
}
```

Notice that the program iterates through each character of the string. The `toupper` function is passed `s[i]`. Each character (if lowercase) is converted to uppercase.

- It's worth mentioning that `toupper` automatically knows to uppercase only lowercase characters. Hence, your code can be simplified as follows:

```
// Uppercases string using ctype library

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string s = get_string("Before: ");
    printf("After: ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        printf("%c", toupper(s[i]));
    }
    printf("\n");
}
```

Notice that this code uppercases a string using the `ctype` library.

- You can read about all the capabilities of the `ctype` library on the [Manual Pages](https://manual.cs50.io/#ctype.h) (<https://manual.cs50.io/#ctype.h>).

## Command-Line Arguments

- `Command-line arguments` are those arguments that are passed to your program at the command line. For example, all those statements you typed after `clang` are considered command line arguments. You can use these arguments in your own programs!
- In your terminal window, type `code greet.c` and write code as follows:

```
// Uses get_string

#include <cs50.h>
#include <stdio.h>

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

Notice that this says `hello` to the user.

- Still, would it not be nice to be able to take arguments before the program even runs? Modify your code as follows:

```
// Prints a command-line argument

#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    if (argc == 2)
    {
        printf("hello, %s\n", argv[1]);
    }
    else
    {
        printf("hello, world\n");
    }
}
```

Notice that this program knows both `argc`, the number of command line arguments, and `argv`, which is an array of the characters passed as arguments at the command line.

- Therefore, using the syntax of this program, executing `./greet David` would result

in the program saying `hello, David`.

- You can print each of the command-line arguments with the following:

```
// Prints command-line arguments

#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    for (int i = 0; i < argc; i++)
    {
        printf("%s\n", argv[i]);
    }
}
```

## Exit Status

- When a program ends, a special exit code is provided to the computer.
- When a program exits without error, a status code of `0` is provided to the computer. Often, when an error occurs that results in the program ending, a status of `1` is provided by the computer.
- You could write a program as follows that illustrates this by typing `code status.c` and writing code as follows:

```
// Returns explicit value from main

#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    if (argc != 2)
    {
        printf("Missing command-line argument\n");
        return 1;
    }
    printf("hello, %s\n", argv[1]);
    return 0;
}
```

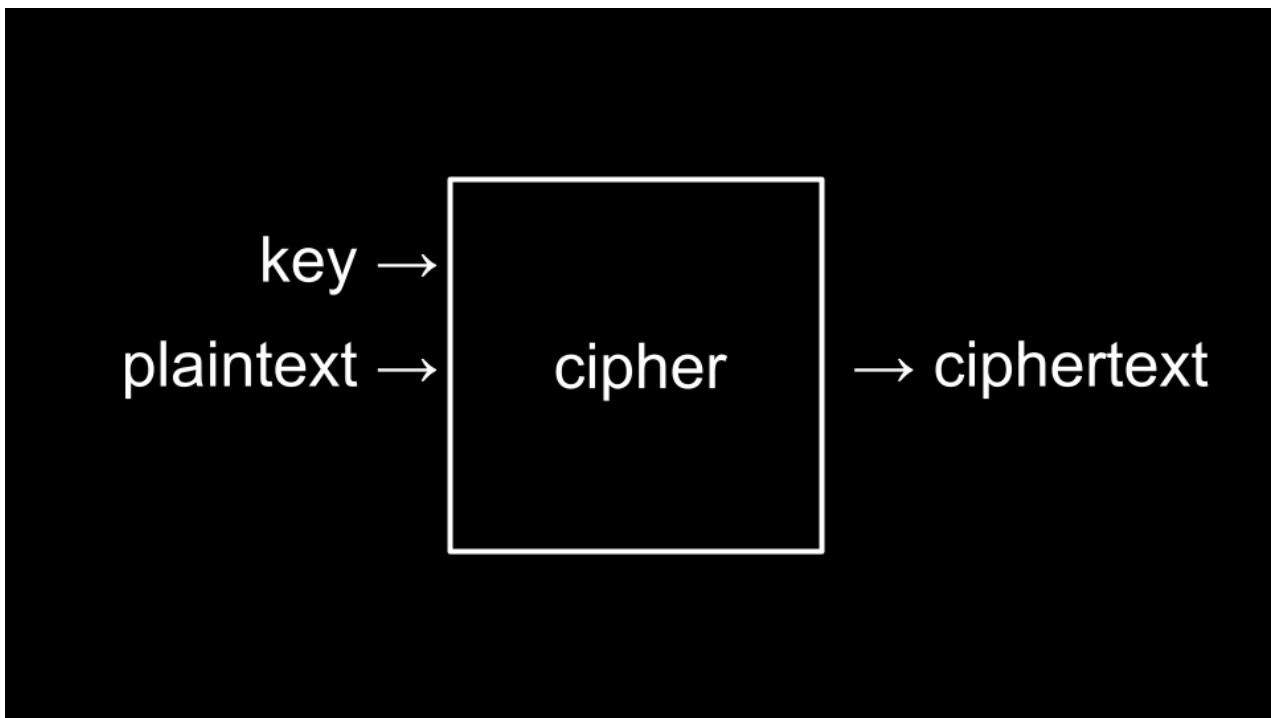
Notice that if you fail to provide `./status David`, you will get an exit status of `1`. However, if you do provide `./status David`, you will get an exit status of `0`.

- You can type `echo $?` in the terminal to see the exit status of the last run command.
- You can imagine how you might use portions of the above program to check if a user provided the correct number of command-line arguments.

# Cryptography

---

- Cryptography is the art of ciphering and deciphering a message.
- Now, with the building block of arrays, chars, and strings, you can cipher and decipher a message.
- `plaintext` and a `key` are provided to a `cipher`, resulting in ciphered text.



- The key is a special argument passed to the cipher along with the plaintext. The cipher uses the key to make decisions about how to implement its cipher algorithm.
- This week, you will undertake programming challenges similar to the above.

## Summing Up

---

In this lesson, you learned more details about compiling and how data is stored within a computer. Specifically, you learned...

- Generally, how a compiler works.
- How to debug your code using four methods.
- How to utilize arrays within your code.
- How arrays store data in back to back portions of memory.
- How strings are simply arrays of characters.
- How to interact with arrays in your code.
- How command-line arguments can be passed to your programs.
- The basic building-blocks of cryptography.

See you next time!