Open your IDE and create an *int main(void)* function. Then, create an *int increment(int n)* function outside of *int main(void)*, which returns *n + 1*. Call *increment* on 3, 5, and 7 in *int main(void)* and print your answers, each on a separate line.

We will begin at 3:07pm!

```
int main(void){

        int a = increment(3);
        int b = increment(5);
        printf("%i\n", a)
}

int increment(int n)
{
        return n+1;
}
```

```
return type  name  args
{
    return [ ]
}
```

CS50 Section 2

- Compiling (again!)
- Debugging (and my personal favorite strategy, printing!)
- Memory
- Arrays
- Strings
- Command-line Arguments

Compiling

- Preprocessing – headers! What are headers again?

Compiling

- Preprocessing – headers! These are lines such as *#include <cs50.h>*, which tells **clang** to look for this header file first since it contains functions that we want to use in our program!

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

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

Compiling

- Preprocessing – headers! These are lines such as *#include <cs50.h>*, which tells **clang** to look for this header file first since it contains functions that we want to use in our program!

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

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

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

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

Compiling

- Preprocessing – headers! These are lines such as *#include <cs50.h>*, which tells **clang** to look for this header file first since it contains functions that we want to use in our program!

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

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```



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

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

- Compiling – converts source code to assembly code. What are some things we remember about assembly code?

popq addq mov_

source code → assembly code → machine code

functions like popq that operate on bits or registers

%ebx %eax

Compiling

- Preprocessing – headers! These are lines such as *#include <cs50.h>*, which tells **clang** to look for this header file first since it contains functions that we want to use in our program!

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

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

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

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

- Compiling – converts source code to assembly code. What are some things we remember about assembly code?
- Assembling – converts assembly code to machine code (binary)

Compiling

- Preprocessing – headers! These are lines such as *#include <cs50.h>*, which tells **clang** to look for this header file first since it contains functions that we want to use in our program!

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

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

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

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

- Compiling – converts source code to assembly code. What are some things we remember about assembly code?
- Assembling – converts assembly code to machine code (binary)
- Linking – contents of the previously compiled libraries that we want to link (ex. *cs50.c)* are combined with the binary in our program. Thus, in the program above, *hello.c, cs50.c, and printf.c* are all compiled into *hello*.

- Compiling (again!)
- Debugging (and my personal favorite strategy, printing!)
- Memory
- Arrays
- Strings
- Command-line Arguments

Error Messages

- Problem: ”No rule to make target _____”; Solution: in the terminal, cd into the correct directory before running “make _____”
- Runtime Errors: make — ✓ ⇒ no compile errors

  Are you dividing by zero somewhere?

  Are you indexing a negative index or an index > length for an array?

  Do you have an infinite loop?

  Are you accessing a piece of memory that you have yet to allocate?

- **Printing as a form of debugging!**

  If you don’t know where you’re getting an error, you can insert random print statements in the code to determine where the program stops running.

  If you’re printing an integer named “hi,” and it isn’t what you want, you can insert printf(“%i\n”, hi) around changes of the variable hi to see how hi is being changed in your code.

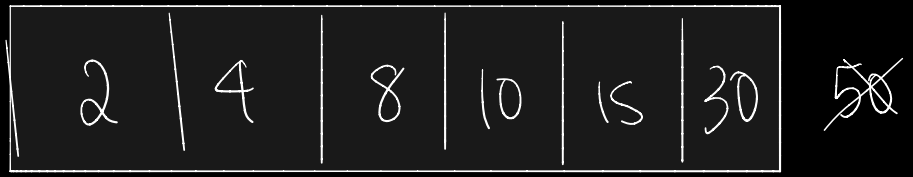
- Compiling (again!)
- Debugging (and my personal favorite strategy, printing!)
- Memory
- Arrays
- Strings
- Command-line Arguments

Memory and Arrays

When we store numbers in an array, each value is back to back, so we can use arithmetic to get an index. Then, we can instantly jump to that address. This gives us random access, which is constant time.

Suppose we want to store six values in memory. We then ask our operating system for just enough bytes for six numbers.

What if we wanted to add the number 50?



arrays : constant size

Memory and Arrays

When we store numbers in an array, each value is back to back, so we can use arithmetic to get an index. Then, we can instantly jump to that address. This gives us random access, which is constant time.

Suppose we want to store six values in memory. We then ask our operating system for just enough bytes for six numbers.

What if we wanted to add the number 50? Since we only asked our operating system for enough bytes for six numbers, the operating system might have already allocated the memory from 6 and beyond to some other aspect of our program.

For a temporary fix, we could've just asked the operating system for enough space for 7 or 8 or even 100 values. **In this case, we're asking for more memory than we actually need. Then, the computer has less space for other programs to store and run.**

Single quote vs double quote?

Single quote : chars

double quotes : strings

Single quote vs double quote?

Chars to ints? Ints to chars?

ASCII

'A' : 65

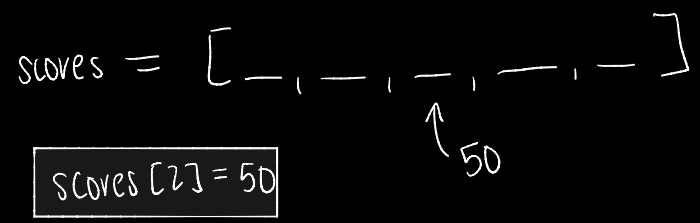
'a' : 97

' ' : 32

'0' : 48

type name[size]

int scores[5]



```
int scores[3];

for (int i = 0; i < 3; i++)
{
    scores[i] = i;
}
```

[- , - , -]

i = 0
scores[0] = 0

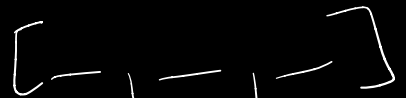
i = 1
scores[1] = 1

i = 2
scores[2] = 2

scores = [0, 1, 2];

```
int scores[3];

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



An array example

Create an array of size 5, where each element is two times the previous, and the first element is 1

```
int ex[5];
ex[0] = 1;

for(int i=1; i<=4; i++)
{
    ex[i] = ex[i-1] * 2;
}
```

ex[i] × 2

1	2	4	8	16
	↑			↑
	i=1			i=4

ex[1] = ex[0] × 2

1 × 2

= 2

Recall…

Strings are an array of chars

ASCII: 'a' = 97

count

'a'	97	-97 = 0
'b'	98	-97 = 1
'c'	99	-'a' = 2
'd'	100	-'a' = 3

[0 , 1 , 2 , ...]

a's b's c's

String s = get-string("givemestring!"); "cat"

print s[0], output 'c'

int strlen = 11;

"abcdefbcdea" = s ← string

0 1 2 3 4

how do I get 'e'? s[4]

How can I store the quantity of each character in an array of length 26?

[2, 2, 2, 2, 2, 1, ... 0]

↑ a ↑ b ↑ z

COUNT

int count[26];

count[s[0] - 'a']++;

1. s[0] = 'a' = 97
2. 97 - 97 = 0

evaluates to 0 ↓ count[0]++;

```
int count[26];
int strlen = 11;
for(int i=0; i<strlen; i++)
{
    count[s[i]-'a']++;
}
```

- Compiling (again!)
- Debugging (and my personal favorite strategy, printing!)
- Memory
- Arrays
- Strings
- Command Line Arguments

```
int count[26] = {0};
int strlen = 3;
for(int i=0; i<strlen; i++)
{
    count[s[i]-'a']++;
}
```

"cat"
0 1 2

count = [_ , _ , _ , _ , ... , _] length 26
strlen = 3
i = 0
count[s[i]-'a']++ → s[0] = 'c'
s[0] - 'a' : 2
count[2]++
[_ , _ , 1 , _ , ... _]

i = 1
count[s[i]-'a']++ → s[1] = 'a'
s[1] - 'a' = 0
count[0] = count[0] + 1 ⇔ count[0]++
[1 , _ , 1 , _ , ... _]

```
#include <string.h>
int l = strlen(s)
```

```
string s = "CS50";
```

```
string s = "CS50";
```

What data structure is this?

s[0]

s[1]

A string example

Print a string character by character, each on a new line

FOR LOOP

strlen(s)

```
string s = get_string("____");
int len = strlen(s)
for (int i=0 ; i<len ; i++)
{
    printf("%c\n",s[i]);
}
```

- Compiling (again!)
- Debugging (and my personal favorite strategy, printing!)
- Memory
- Arrays
- Strings
- Command-line Arguments

Command-Line Arguments

```
int main(int argc, string argv[])
{



}
```

Command-Line Arguments

argc: number of command line arguments

argv: array of command line arguments

./caesar 2

argv[0] argv[1]

```c
#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;
}
```

./hello Phyllis

argv[0] argv[1]

Exercises

Create two files in your IDE: reverse.c and addition.c.

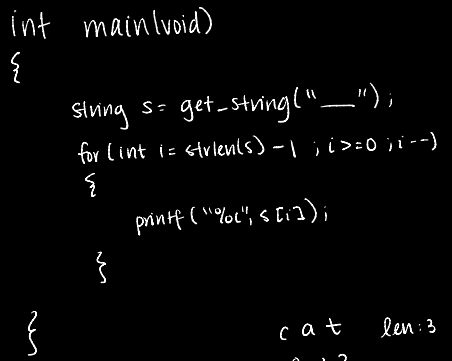
1. Write a program `reverse.c` that takes a string as input, and reverses it.

2. Write a program `addition.c` that adds two numbers provided as command-line arguments.

Exercises

Create two files in your IDE: reverse.c and addition.c.

1. Write a program `reverse.c` that takes a string as input, and reverses it.



- Sample Usage

```bash
$ ./reverse
Text: This is CS50.
Reverse: .05SC si sihT
```

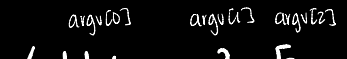
Exercises

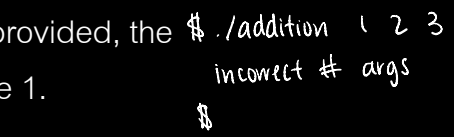
Create two files in your IDE: reverse.c and addition.c.

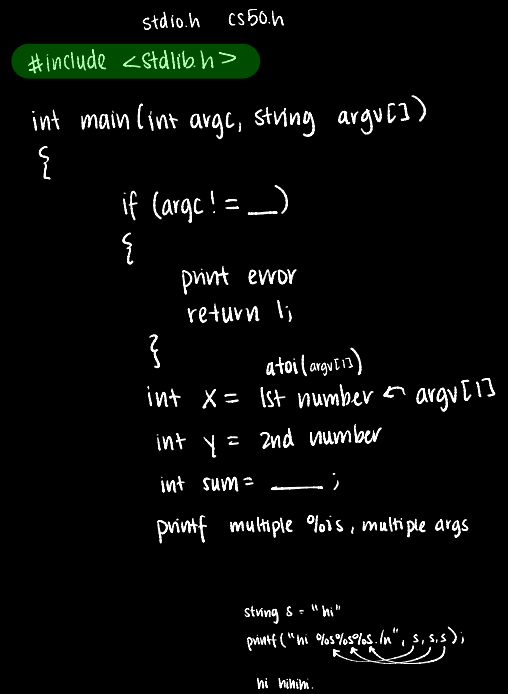
2. Write a program `addition.c` that adds two numbers provided as command-line arguments.

Details

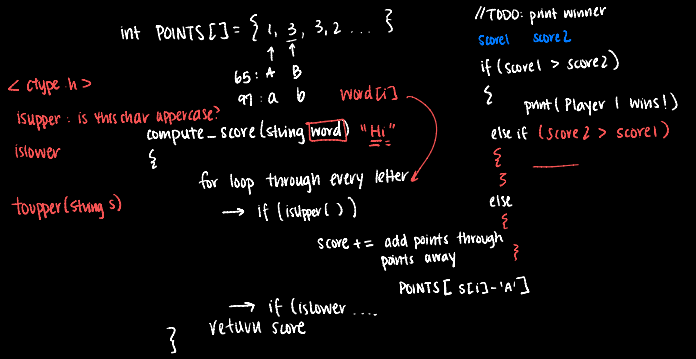
- The program should accept two integers as command-line arguments.
- The program should output both original numbers, and their sum. If the program is run as `./additional 2 8`, for example, the output should be `2 + 8 = 10`.
- If the incorrect number of command-line arguments is provided, the program should display an error and return with exit code 1.







Lab




```
//TODO: print winner
score1   score2
if (score1 > score2)
{
    print( Player 1 wins! )
else if (score2 > score1)
{
}
else
{
}
```

CS50 Section 2