

CS50 - SECTION 2

9/25/18



WELCOME IF YOU'RE NEW TO THE SECTION!

wadesilvestro@college.harvard.edu

(863) 666-4003

POST-MORTEM ON PSET 1

REFLECTIONS:

PSET 1

*How did it go? What did you like?
Frustrations? Things to think about
moving forward?*

SOME REMINDERS

- There is no “right” way to program
 - There is only the *best* way given our context, constraints, etc.
- What are some tradeoffs we might have to make when coding?

SOME REMINDERS

- There is no “right” way to design a program
 - There is only the *best* way given our context, constraints, etc.
- What are some tradeoffs we might have to make when coding?
 - **Technical** - Memory, processing power, computational time available
 - **Knowledge** - What do we know? What solutions are available?
 - **Resources** - Developer quantity/time available, money, etc.
 - **Any many more...**
- In CS50, you’re graded on design, which tries to encapsulate a small cross-section of all the above tradeoffs

DESIGN TIP #1

Repetitive Code: What's the number one principle in programming?

DESIGN TIP #1

Repetitive Code: What's the number one principle in programming?

Don't Repeat Yourself

AKA:

DRY

DESIGN TIP #1

Repetitive Code: What's the number one principle in programming?

Don't Repeat Yourself

AKA:

DRY

DESIGN TIP #1

Repetitive Code: What's the number one principle in programming?

Don't Repeat Yourself

AKA:

DRY

DESIGN TIP #1

Repetitive Code: What's the number one principle in programming?

Don't Repeat Yourself

AKA:

DRY

DESIGN TIP #2

Unnecessarily and computationally difficult code

How can we improve the following snippet from `cash`?

```
// dollars is our input from the user
int cents = round(dollars * 100);
int coins = 0;
while (cents >= 25)
{
    coins++;
    cents -= 25;
}
```

DESIGN TIP #2

This is clearer and likely even faster because we've excluded a loop

```
// dollars is our input from the user
```

```
int cents = round(dollars * 100);
```

```
int coins = 0;
```

```
coins += cents / 25;
```

```
cents = cents % 25;
```

DESIGN TIP #3

Don't waste memory when you don't need to

What's wrong with the following code?

```
int x = 132 % 7;
```

```
printf("%i", x);
```

DESIGN TIP #3

Don't waste memory when you don't need to

What's wrong with the following code?

```
int x = 132 % 7;
```

```
printf("%i", x);
```

If we never intend to use `x` again, then we're wasting memory and add unnecessary code by declaring/initializing a variable! Do this instead:

```
printf("%i", 132 % 7);
```

STYLE TIPS

1. Use descriptive variable names (no **x**, **y**, **z**, unless it is used in a for loop and/or as a counter value)
2. Format your code cleanly—Keep your indentation, tabs, spaces, etc. clear
 - a. CS50 Style guide: <https://cs50.readthedocs.io/style/c/>
3. Use comments to explain non-obvious parts of your code
 - a. What do good commenting practices look like?
 - b. We care about quantity *AND* quality

CONCEPTS DEEP-DIVE

“SHORTS” FOR THE WEEK

A man in a maroon shirt stands against a light gray background. The word "FUNCTIONS" is overlaid in large white text.

FUNCTIONS

<https://www.youtube.com/watch?v=b7-0sb-DV84>

A man in a maroon shirt stands against a light gray background. The words "COMMAND LINE ARGS" are overlaid in large white text.

COMMAND LINE ARGS

<https://www.youtube.com/watch?v=thL7ILwRNMM>

A man in a maroon shirt stands against a light gray background. The word "ARRAYS" is overlaid in large white text.

ARRAYS

<https://www.youtube.com/watch?v=mISkNAfWl8k>

A man in a maroon shirt stands against a light gray background. The word "DEBUGGING" is overlaid in large white text.

DEBUGGING

<https://www.youtube.com/watch?v=w4TAY2HPLEq>

EXTRA “SHORTS” IF YOU HAVE TIME

A man in a maroon shirt stands in front of a light gray background. The text "SELECTION SORT" is overlaid in large white letters.

SELECTION SORT

<https://www.youtube.com/watch?v=3hH8kTHFW2A>

A man in a maroon shirt stands in front of a light gray background. The text "BUBBLE SORT" is overlaid in large white letters.

BUBBLE SORT

<https://www.youtube.com/watch?v=RT-hUXUWQ2I>

A man in a maroon shirt stands in front of a light gray background. The text "INSERTION SORT" is overlaid in large white letters.

INSERTION SORT

<https://www.youtube.com/watch?v=O0VbBkUvriI>

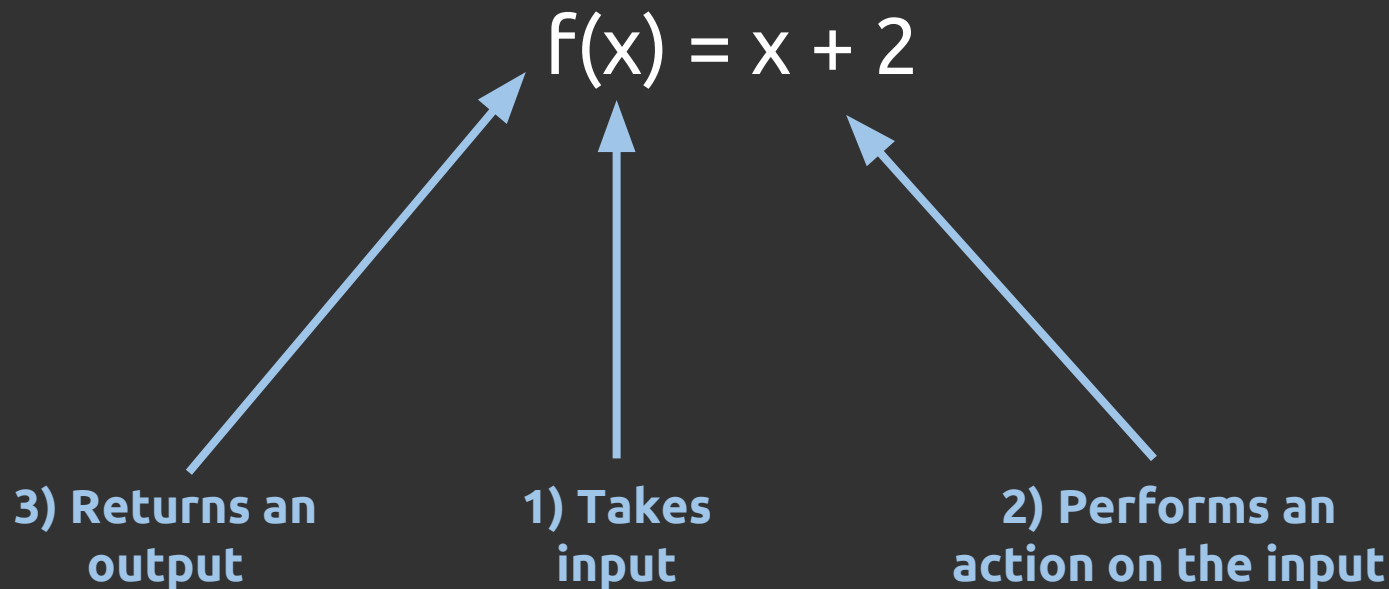
A man in a maroon shirt stands in front of a light gray background. The text "MERGE SHORT" is overlaid in large white letters.

MERGE SHORT

<https://www.youtube.com/watch?v=Ns7tGNbtvV4>

FUNCTIONS

What is a function?



FUNCTIONS IN C

Here is a function that calculates the balance in a bank account after 1 year of simple interest has accumulated

```
double accumulate_interest(double balance, double rate)
{
    double updated_balance = balance + rate * balance;
    return updated_balance;
}
```

FUNCTIONS IN C

Here is a function that calculates the balance in a bank account after 1 year of simple interest has accumulated

```
double accumulate_interest(double balance, double rate)
{
    double updated_balance = balance + rate * balance;
    return updated_balance;
}
```

1) Takes
input

2) Performs an
action on the input

3) Returns an
output

FUNCTIONS IN C

A program that uses it might look like this:

```
#include <cs50.h>
#include <stdio.h>
double accumulate_interest(double balance, double rate);
int main(void)
{
    printf("Starting balance: ");
    double start = get_double();
    printf("Annual interest rate: ");
    double interest = get_double();
    double updated = accumulate_interest(start, interest);
    printf("Updated balance: %.2f\n", updated);
}
double accumulate_interest(double balance, double rate)
{
    double updated_balance = balance + rate * balance;
    return updated_balance;
}
```

FUNCTIONS IN C

A program that uses it might look like this:

```
#include <cs50.h>
#include <stdio.h>
double accumulate_interest(double balance, double rate);
int main(void)
{
    printf("Starting balance: ");
    double start = get_double();
    printf("Annual interest rate: ");
    double interest = get_double();
    double updated = accumulate_interest(start, interest);
    printf("Updated balance: %.2f\n", updated);
}
double accumulate_interest(double balance, double rate)
{
    double updated_balance = balance + rate * balance;
    return updated_balance;
}
```

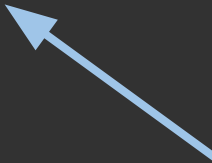


What is this?

FUNCTIONS IN C

A program that uses it might look like this:

```
#include <cs50.h>
#include <stdio.h>
double accumulate_interest(double balance, double rate);
int main(void)
{
    printf("Starting balance: ");
    double start = get_double();
    printf("Annual interest rate: ");
    double interest = get_double();
    double updated = accumulate_interest(start, interest);
    printf("Updated balance: %.2f\n", updated);
}
double accumulate_interest(double balance, double rate)
{
    double updated_balance = balance + rate * balance;
    return updated_balance;
}
```



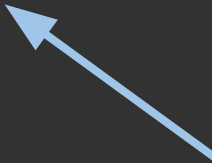
What is this?
**A function
prototype**

FUNCTIONS IN C

A program that uses it might look like this:

```
#include <cs50.h>
#include <stdio.h>
double accumulate_interest(double balance, double rate);
int main(void)
{
    printf("Starting balance: ");
    double start = get_double();
    printf("Annual interest rate: ");
    double interest = get_double();
    double updated = accumulate_interest(start, interest);
    printf("Updated balance: %.2f\n", updated);
}
double accumulate_interest(double balance, double rate)
{
    double updated_balance = balance + rate * balance;
    return updated_balance;
}
```

What is this?
**A function
prototype**



**C reads top to
bottom like we do,
so if you “call” a
function before
you define it, an
error will be
thrown**

FUNCTIONS IN C

Alternatively, define your function first to exclude the prototype:

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

double accumulate_interest(double balance, double rate)
{
    double updated_balance = balance + rate * balance;
    return updated_balance;
}

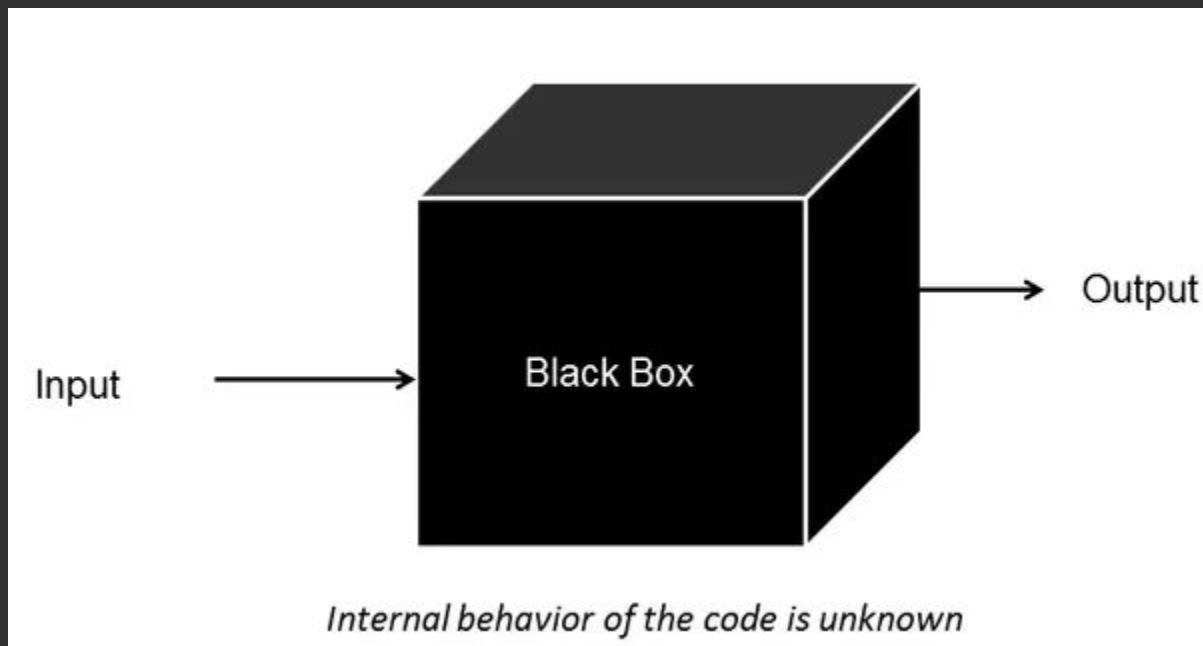
int main(void)
{
    printf("Starting balance: ");
    double start = get_double();
    printf("Annual interest rate: ");
    double interest = get_double();
    double updated = accumulate_interest(start, interest);
    printf("Updated balance: %.2f\n", updated);
}
```

WHY DO WE USE FUNCTIONS?

1) They help us better organize our code.

WHY DO WE USE FUNCTIONS?

2) They offer us a powerful tool of abstraction.



WHY DO WE USE FUNCTIONS?

2) They offer us a powerful tool of abstraction.

You have called a variety of functions from the CS50 library already:

- `get_int()`
- `get_float()`
- `get_string()`

You have no clue how these functions are actually *implemented*, but it doesn't matter. *You know what inputs to provide and what outputs to expect, so you can use them reliably!*

WHY DO WE USE FUNCTIONS?

2) They offer us a powerful tool of abstraction.

Imagine you needed to change our interest function from before to use compound interest. How might you do that?

WHY DO WE USE FUNCTIONS?

2) They offer us a powerful tool of abstraction.

Imagine you needed to change our interest function from before to use compound interest. How might you do that?

Simply change the formula in the function! This means anywhere you call that function, it's using the new formula. You change your code in one place and not a dozen...

WHY DO WE USE FUNCTIONS?

3) They simplify development of our program.

You can worry about testing/debugging a single function rather than everything at once.

They also help us think about breaking our problem into smaller parts. This is a fundamental CS principle.

WHY DO WE USE FUNCTIONS?

4) They offer reusability.

You can call a function over and over again instead of rewriting that code.

Look for ways to **refactor** your code to extract out “common parts” and put them in a function.

WAIT, WHAT'S ANOTHER FUNCTION WE'VE ALREADY SEEN?

What is one function you've already used in every C program you've written?

WAIT, WHAT'S ANOTHER FUNCTION WE'VE ALREADY SEEN?

What is one function you've already used in every C program you've written?

The main function!

WAIT, WHAT'S ANOTHER FUNCTION WE'VE ALREADY SEEN?

```
int main(void)
```

```
{
```

```
}
```

1) Takes
input

2) Performs an
action on the input

3) Returns an
output

WAIT, WHAT'S ANOTHER FUNCTION WE'VE ALREADY SEEN?

```
int main(void)
{
}

```

3) Returns an output

2) Performs an action on the input

1) Takes input -
Actually, another way you can take input is void. This means you don't take any input at all!

WAIT, WHAT'S ANOTHER FUNCTION WE'VE ALREADY SEEN?

```
int main(void)
```

```
{
```

```
}
```

3) Returns an output -
We return an “exit code”
which is an integer.

2) Performs an
action on the input

1) Takes input -
Actually, another
way you can take
input is void. This
means you don't take
any input at all!

READING DOCUMENTATION/REFERENCE

CS50 provides reference/documentation at:
<https://reference.cs50.net/>

SYNOPSIS

```
#include <cs50.h>
float get_float(string format, ...);
```


READING DOCUMENTATION/REFERENCE

CS50 provides reference/documentation at:
<https://reference.cs50.net/>

SYNOPSIS

```
#include <cs50.h>
float get_float(string format, ...);
```

Parameters (input)



Return type



READING DOCUMENTATION/REFERENCE

CS50 provides reference/documentation at:

<https://reference.cs50.net/>

RETURN VALUE

Returns the float equivalent to the line read from stdin in [FLT_MIN, FLT_MAX), as precisely as possible. If line can't be read, returns FLT_MAX.



Return value to expect - Notice that implementation details are excluded

VARIABLE SCOPES

In C, we have two types of scope:

- **Global** - Defined outside of a function (including main) and available to the entire program
 - That is, unless you override them with precedence...
- **Local** - Defined inside of a function and available only within that “block”

VARIABLE SCOPES

Where are the local and global variables in this code?

```
#include <stdio.h>

int g;

int main () {
    int a, b;
    a = 10;
    b = 20;
    g = a + b;

    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
}
```

VARIABLE SCOPES

Where are the local and global variables in this code?

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

```
int g;
```

```
int main () {
```

```
    int a, b;
```

```
    a = 10;
```

```
    b = 20;
```

```
    g = a + b;
```

```
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
```

```
}
```

Global Variable



Local Variables



VARIABLE SCOPES

You can use `{ }` to create a new block and specify a local scope:

```
#include <stdio.h>

int main () {
    int a = 7;

    {
        int a = 4;
        printf("%i\n", a);
    }

    printf("%i\n", a);
}
```

VARIABLE SCOPES

You can use `{ }` to create a new block and specify a local scope:

```
#include <stdio.h>

int main () {
    int a = 7;

    {
        int a = 4;
        printf("%i\n", a);
    }

    printf("%i\n", a);
}
```

This will print 4 and then 7.

Note: If you compile this with `make`, it will throw an error because of how it's configured, but `clang` will let it through. Technically, this points out that it's bad practice to name variables the same thing, even if their scope is different.

HANDS ON PRACTICE - TASK #1 (`temp1.c`)

<http://bit.ly/2DvR5nb>

You'll use this sandbox for all the practice in this section.

HANDS ON PRACTICE - SOLUTION #1 (temp1.c)

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

float ctof(float temp);
float ftoc(float temp);

int main(void) {
    double temp1 = 45.3;
    double temp2 = 23.8;

    printf("%f\n", ftoc(temp1));
    printf("%f\n", ctof(temp2));
}

float ctof(float temp) {
    return (temp * (9.0 / 5.0)) + 32.0;
}

float ftoc(float temp) {
    return (temp - 32.0) * (5.0 / 9.0);
}
```

ARRAYS

What is an array?

ARRAYS

What is an array?

A data type that allows us to store multiple values of the same type in memory

ARRAYS - HOW DO WE DECLARE THEM?

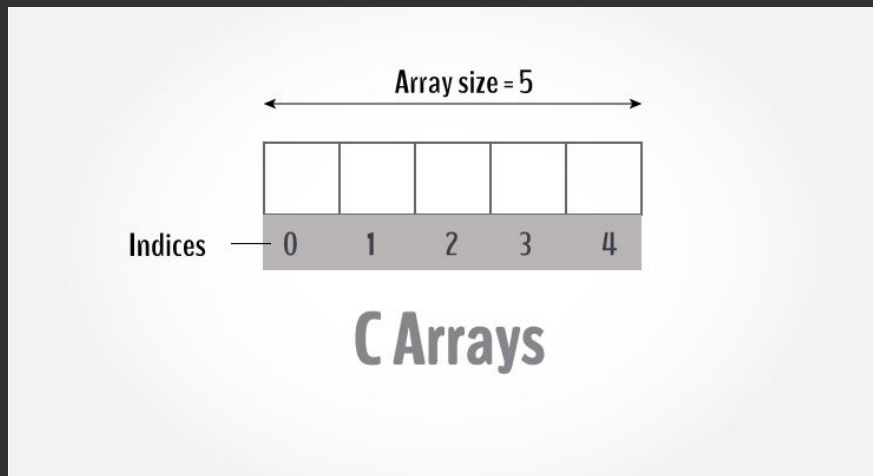
```
type name[size];
```

- A note on types: Arrays must contain values of the same type—Why do we think that might be?

ARRAYS - HOW DO WE DECLARE THEM?

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

- A note on types: Arrays must contain values of the same type—Why do we think that might be?
 - **C has only partitioned enough memory for that particular type and size of array!**



ARRAYS - HOW DO WE INITIALIZE THEM?

- You can either initialize with declaration (instantiation):

```
type name[size] = {<values>;
```

- Or you can initialize separately:

```
name[i] = {<value>;
```

ARRAYS - HOW DO WE INITIALIZE THEM?

- Not providing a size for the declaration will set the array equal to whatever you initialize with:

```
int arr[] = {1, 2, 3};
```

This causes C to create an array of size 3 implicitly

ARRAYS - HOW DO WE ACCESS VALUES?

`name[i] ;`

- `i` is our counter value
- Note that arrays are indexed from zero—If want to access the n th element, we use `name[n-1]`

ARRAYS - OUT OF BOUNDS

```
int a[10];
```

```
printf("%i", a[143]);
```

This code will generate an error due to the configuration of `make` and `clang` in CS50 Labs/Sandbox, but being notified of this error is not always the case!

ARRAYS - OUT OF BOUNDS

```
int a[3] = {1, 2, 3};
```

```
printf("%i", a[143]);
```

Without error reporting, C will often try to access that memory location and give you back a random value. Where do we think that value is coming from?

ARRAYS - OUT OF BOUNDS

```
int a[3] = {1, 2, 3};
```

```
printf("%i", a[143]);
```

C will often give you back a random value. Where do we think that value is coming from? - **It's coming from whatever is located at that memory location adjacent to the array**

ARRAYS - OUT OF BOUNDS

```
int a[3] = {1, 2, 3};
```

```
printf("%i", a[143]);
```

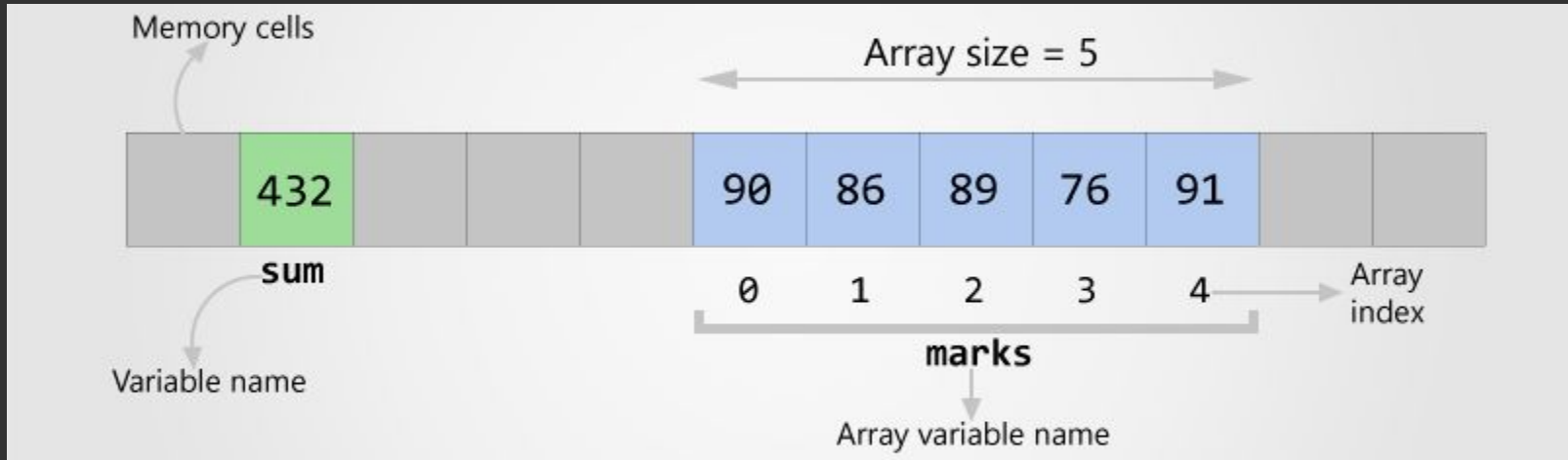
Other times you'll get a scary error:

Segmentation fault

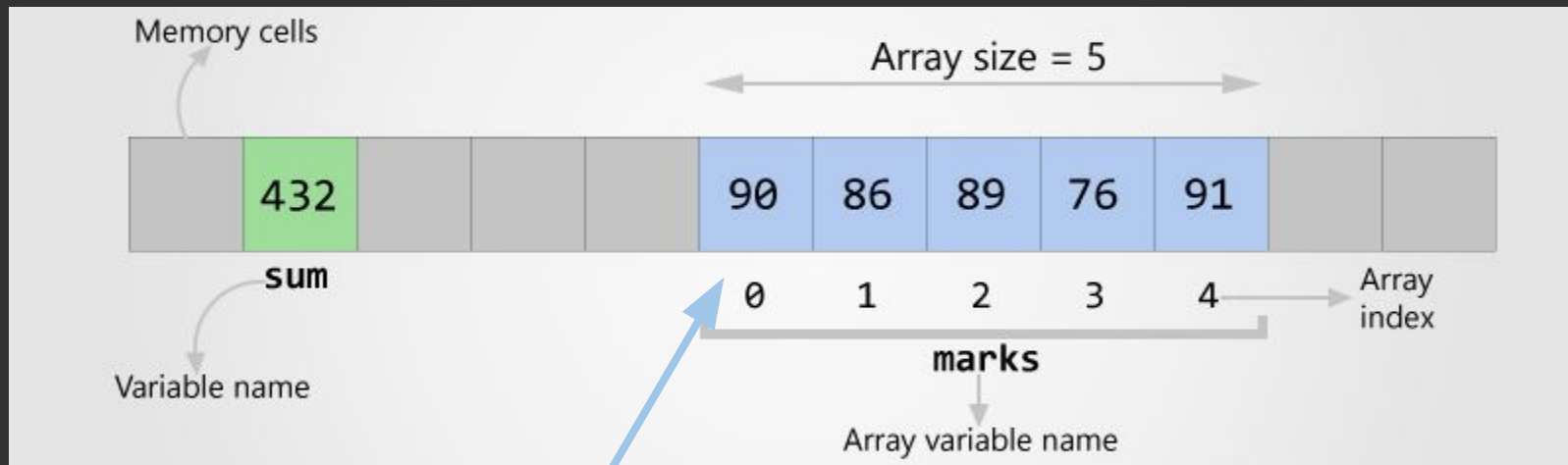
This means you're trying to read/write to an illegal memory location

ARRAYS - UNDER THE HOOD

We can understand why we get the segmentation fault error better if we look at how arrays are stored by C under the hood:

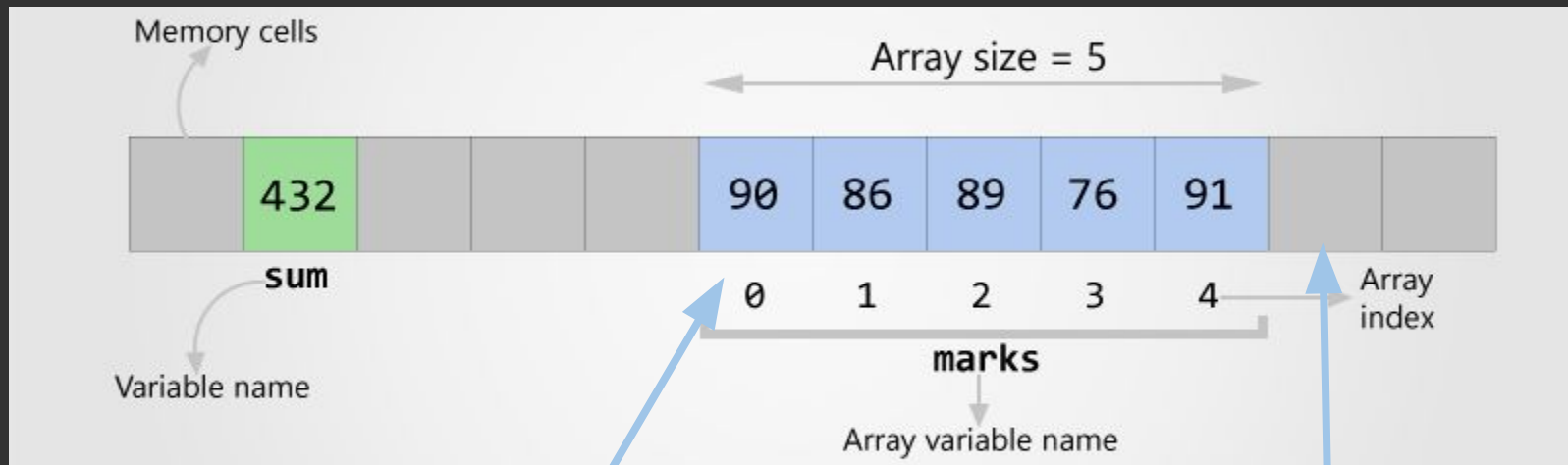


ARRAYS - UNDER THE HOOD



Notice that arrays are stored
in *contiguous memory*

ARRAYS - UNDER THE HOOD



Notice that arrays are stored in *contiguous memory*

If we try to access a value outside of the array bounds, we hit other values in memory

ARRAYS - IMPORTANT NOTES

1. Because C reserves a set amount of memory upon the declaration of an array, **you cannot change the size of your array after creating it.**

How might we accomplish changing the size of our array given this constraint?

ARRAYS - IMPORTANT NOTES

1. Because C reserves a set amount of memory upon the declaration of an array, **you cannot change the size of your array after creating it.**

How might we accomplish changing the size of our array given this constraint?

You would have to create an entirely new array of a larger size and then copy the values from the old array into the new one.

ARRAYS - IMPORTANT NOTES

2. C does not treat arrays as a single entity. Meaning you can't copy the values of one array to another by just setting them equal to one another.

```
int arr1[3] = {1, 2, 3};  
int arr2[3];  
arr2 = arr1;
```

The above is not permitted by C!

ARRAYS - IMPORTANT NOTES

2. C does not treat arrays as a single entity. Meaning you can't copy the values of one array to another by just setting them equal to one another.

If you want to copy values, you have to do it one-by-one:

```
int arr1[3] = {1, 2, 3};
```

```
int arr2[3];
```

```
for(int i=0; i<3; i++) {
```

```
    arr2[i] = arr1[i];
```

```
}
```

ARRAYS - IMPORTANT NOTES

2. C does not treat arrays as a single entity. Meaning you can't copy the values of one array to another by just setting them equal to one another.

“But why does C subject us to this insanity???” you may be asking.

A SHORT DIVERSION

What does the following program print?

```
int x = 4;
```

```
int y = x;
```

```
x = 7;
```

```
printf("%i\n", y);
```

A SHORT DIVERSION

What does the following program print?

```
int x = 4;
```

```
int y = x;
```

```
x = 7;
```

```
printf("%i\n", y);
```

It prints 4! This is because most variables in C are passed by value.

For example, this program sets `x` equal to 4 and then `y` equal to `x`. `y` is referencing 4 because C literally assigns 4 to `y`, not the memory location of `x` to `y`.

Had we been passing by reference, then anytime `x` changes, `y` would also change since it references what `x` is.

ARRAYS - IMPORTANT NOTES

2. C does not treat arrays as a single entity. Meaning you can't copy the values of one array to another by just setting them equal to one another.

“But why does C subject us to this insanity???” you may be asking.

So, C subjects us to this “insanity” because arrays are passed by reference! Setting two arrays equal to one another simply assigns one to the other’s memory location.

ARRAYS - IMPORTANT NOTES

3. We can declare an array, initialize some/all of its values to a specific list we provide, or we can initialize them all to zero.

Initialize All Values:

```
int arr[3] = {1,2,3};
```

This initializes all three
“slots” of the array

Initialize Some Values:

```
int arr[3] = {1};
```

This initializes `arr[0]` to 1
and sets the remaining
“slots” equal to zero

Initialize All to Zero:

```
int arr[3] = {0};
```

This sets all slots of the
array equal to zero

ARRAYS - WAIT...WHAT IS A STRING AGAIN?

Strings as we've dealt with so far were implemented by the CS50 library. But in reality, they are actually an array of characters

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

ARRAYS - WAIT...WHAT IS A STRING AGAIN?

Strings as we've dealt with so far were implemented by the CS50 library. But in reality, they are actually an array of characters

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

This is called a “null terminator.” It tells C that our char array has ended. More on the specifics later...

ARRAYS - MULTIDIMENSIONAL

We can even create multidimensional arrays in C:

```
bool arr[10][10];
```

The above gives us a 10x10 array to work with. How might multidimensional arrays be useful?

ARRAYS - MULTIDIMENSIONAL

We can even create multidimensional arrays in C:

```
bool arr[10][10];
```

The above gives us a 10x10 array to work with. How might multidimensional arrays be useful?

Representing complex physical things digitally (e.g. board games), working with multivariable functions, etc...

HANDS ON PRACTICE - TASK #2 (`temp2.c`)

<http://bit.ly/2DvR5nb>

Now continue to `temp2.c` and complete the second task.

HANDS ON PRACTICE - SOLUTION #2 (temp2.c)

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

float ctof(float temp);
float ftoc(float temp);

int main(void) {
    // Temperatures in Fahrenheit
    float temp[5] = {73.2, 24.1, 98.5, 101.9, 45.2};

    float temp_c[5];
    for(int i = 0; i < 5; i++) {
        temp_c[i] = ftoc(temp[i]);
        printf("%f\n", temp_c[i]);
    }
}

float ctof(float temp) {
    return (temp * (9.0 / 5.0)) + 32.0;
}

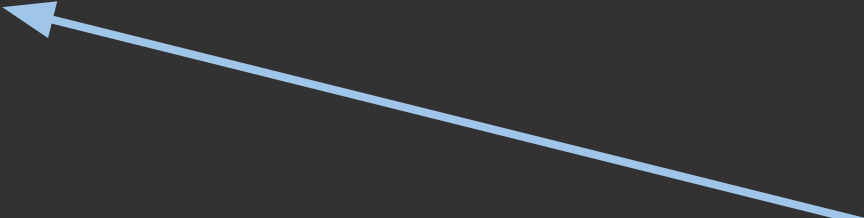
float ftoc(float temp) {
    return (temp - 32.0) * (5.0 / 9.0);
}
```

REVISITING THE MAIN FUNCTION

```
int main(void)
{
}

```

Does the input
always have to be
void?



REVISITING THE MAIN FUNCTION

```
int main(void)
{
}

```


Does the input
always have to be
void?

NO!

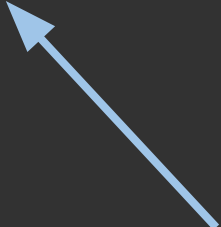
COMMAND LINE ARGUMENTS

We can use command line arguments to pass arguments into our program:

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



**argc represents the
number of arguments
we've passed via the
command line**



**argv is an array of
strings with the
different command line
arguments**

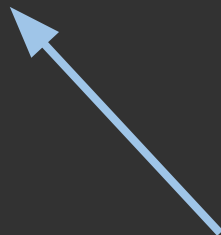
COMMAND LINE ARGUMENTS

```
$ ./main "x" "y" "z"
```

COMMAND LINE ARGUMENTS

```
$ ./ main "x" "y" "z"
```

argc would be 4



argv[0]	argv[1]	argv[2]	argv[3]
"main"	"x"	"y"	"z"

COMMAND LINE ARGUMENTS

```
$ ./main "x" "y" "z"
```

argc would be 4



argv[0] is always the
name of the program

argv[0]	argv[1]	argv[2]	argv[3]
"main"	"x"	"y"	"z"

COMMAND LINE ARGUMENTS - SOME NOTES

- `argv[]` gives us an array of strings
 - If you want command line arguments that are processed as integers, use `atoi(<string>)`
 - Likewise, you can use `atof(<string>)` for doubles and various other functions

COMMAND LINE ARGUMENTS - SOME NOTES

Because they're strings, we can treat them as a multidimensional array. How?

```
$ ./ main "bob" "gloria" "suzy"
```

COMMAND LINE ARGUMENTS - SOME NOTES

Because they're strings, we can treat them as a multidimensional array. How?

```
$ ./ main "bob" "gloria" "suzy"
```

`argv[1][1]` would give us "o".

HANDS ON PRACTICE - TASK #3 (`temp3.c`)

<http://bit.ly/2DvR5nb>

Now continue to `temp3.c` and complete the second task.

HANDS ON PRACTICE - SOLUTION #3 (temp3.c)

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

float ctof(float temp);
float ftoc(float temp);

int main(int argc, string argv[]) {
    if(argv[1][0] == 'f') {
        for(int i = 2; i < argc; i++) {
            printf("%f\n", ftoc(atof(argv[i])));
        }
    }
    else {
        for(int i = 2; i < argc; i++) {
            printf("%f\n", ctof(atof(argv[i])));
        }
    }
}

float ctof(float temp) {
    return (temp * (9.0 / 5.0)) + 32.0;
}

float ftoc(float temp) {
    return (temp - 32.0) * (5.0 / 9.0);
}
```

HANDS ON PRACTICE - SOLUTIONS

<http://bit.ly/2DsL0YD>

**Sample implementations for all three tasks are
contained in the Sandbox above.**

DEBUGGING

What types of bugs might we encounter?



DEBUGGING



What types of bugs might we encounter?

- **Syntax Error** - There is a mistake in the rules/grammar of the language with your code
 - Easiest to fix, but can be tough to find
- **Logic Error** - Something about your algorithm is wrong
 - Think through your algorithm, step-by-step (on paper!)
- **Input Error** - Your code didn't anticipate a particular type of input (e.g. a negative number) and did not include provisions to catch these errors
 - Think about edge cases to anticipate what these might be
- Any many more...

DEBUGGING



We have `help50` to assist us with debugging

- When you run into an error, simply prepend `help50` before the command that threw the error and it will interpret the problem and provide suggestions to you

DEBUGGING



Other tips to help you with debugging:

- Think through algorithm on paper before you attempt to code it
 - Writing pseudocode before real code prevents mistakes
- Use `printf()` statements to understand the flow of your program and whether certain areas are even being reached
 - Can also reveal what the inner contents of variables are at different steps to help you see where your algorithm is messing up
 - Later, we'll learn about `debug50` which helps automate this process
- Not debugging related, but use `style50` to help improve your style!

PROBLEM SET 2

PREVIEW

PROBLEM SET 2 PREVIEW

Submit any two (2) of:

- `caesar.c`
- `vigenere.c`
- `crack.c`

If you submit all three, your highest two scores will count.

REFERENCE SHEETS

CS50

Operators

CS50

Overview

Arithmetic operators (`+`, `-`, `*`, `/`, `%`) perform basic arithmetic operations on numbers. Comparison operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) compare values and return a boolean result. Logical operators (`&&`, `&`, `||`, `|||`) combine boolean expressions. Bitwise operators (`&`, `|`, `^`, `~`, `<<`, `>>`) perform operations on the binary representation of numbers. Assignment operators (`=`, `+=`, `-=`, `*=`, `/=`, `%=`) assign values to variables. The ternary operator (`cond ? expr1 : expr2`) provides a concise way to write conditional expressions.

The precedence of operators determines the order in which operations are performed. The precedence table below shows the relative precedence of various operators. The operators in the same row have the same precedence. The operators in the same column have the same associativity. The operators in the same row and column have the same precedence and associativity.

Precedence	Operator(s)	Associativity
1	<code>++</code> , <code>--</code>	Left to right
2	<code>*</code> , <code>/</code> , <code>%</code>	Left to right
3	<code>+</code> , <code>-</code>	Left to right
4	<code><<</code> , <code>>></code>	Left to right
5	<code>&</code> , <code> </code> , <code>^</code> , <code>~</code>	Left to right
6	<code>&&</code> , <code>&</code> , <code> </code> , <code> </code>	Left to right
7	<code><</code> , <code>></code> , <code><=</code> , <code>>=</code> , <code>==</code> , <code>!=</code>	Left to right
8	<code>?:</code>	Right to left
9	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code>	Right to left

Key Terms

- Arithmetic operators
- Comparison operators
- Logical operators
- Bitwise operators
- Assignment operators
- Ternary operator

Arithmetic Operators

Arithmetic operators perform basic arithmetic operations on numbers. The operators are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulo). The modulo operator returns the remainder of a division operation. For example, `5 % 2` returns `1`.

When working with integers and floating-point numbers, it is important to use the appropriate operator. For example, using `+` to add two integers will result in an integer, while using `+` to add two floating-point numbers will result in a floating-point number.

Operator precedence determines the order in which operations are performed. For example, in the expression `2 + 3 * 4`, the multiplication is performed first, resulting in `2 + 12`, which equals `14`. To change the order of operations, parentheses can be used. For example, `(2 + 3) * 4` results in `20`.

Comparison Operators

Comparison operators compare two values and return a boolean result. The operators are `<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to), `==` (equal to), and `!=` (not equal to). The equality operators (`==` and `!=`) compare the values of two variables. The relational operators (`<`, `>`, `<=`, and `>=`) compare the values of two variables and return a boolean result.

The precedence of comparison operators is lower than that of arithmetic operators. For example, in the expression `2 + 3 < 4`, the addition is performed first, resulting in `5 < 4`, which is `false`. To change the order of operations, parentheses can be used. For example, `(2 + 3) < 4` results in `false`.

Logical Operators

Logical operators combine boolean expressions. The operators are `&&` (logical AND), `&` (bitwise AND), `||` (logical OR), and `|||` (bitwise OR). The logical AND operator (`&&`) returns `true` only if both operands are `true`. The logical OR operator (`||`) returns `true` if at least one operand is `true`. The bitwise AND operator (`&`) performs a bitwise AND operation on the binary representation of two numbers. The bitwise OR operator (`|||`) performs a bitwise OR operation on the binary representation of two numbers.

The precedence of logical operators is lower than that of comparison operators. For example, in the expression `2 < 3 && 4 < 5`, the comparisons are performed first, resulting in `true && true`, which is `true`. To change the order of operations, parentheses can be used. For example, `(2 < 3) && (4 < 5)` results in `true`.

Bitwise Operators

Bitwise operators perform operations on the binary representation of numbers. The operators are `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), `~` (bitwise NOT), `<<` (bitwise left shift), and `>>` (bitwise right shift). The bitwise AND operator (`&`) performs a bitwise AND operation on the binary representation of two numbers. The bitwise OR operator (`|`) performs a bitwise OR operation on the binary representation of two numbers. The bitwise XOR operator (`^`) performs a bitwise XOR operation on the binary representation of two numbers. The bitwise NOT operator (`~`) flips the bits of a number. The bitwise left shift operator (`<<`) shifts the bits of a number to the left. The bitwise right shift operator (`>>`) shifts the bits of a number to the right.

The precedence of bitwise operators is lower than that of logical operators. For example, in the expression `2 < 3 & 4 < 5`, the comparisons are performed first, resulting in `true & true`, which is `true`. To change the order of operations, parentheses can be used. For example, `(2 < 3) & (4 < 5)` results in `true`.

Assignment Operators

Assignment operators assign values to variables. The operators are `=` (assignment), `+=` (addition assignment), `-=` (subtraction assignment), `*=` (multiplication assignment), `/=` (division assignment), and `%=` (modulo assignment). The assignment operator (`=`) assigns the value of the right-hand operand to the left-hand operand. The compound assignment operators (`+=`, `-=`, `*=`, `/=`, and `%=`) perform an arithmetic operation on the left-hand operand and assign the result to the left-hand operand.

The precedence of assignment operators is lower than that of bitwise operators. For example, in the expression `2 + 3 = 4`, the addition is performed first, resulting in `5 = 4`, which is `false`. To change the order of operations, parentheses can be used. For example, `(2 + 3) = 4` results in `false`.

Ternary Operator

The ternary operator (`cond ? expr1 : expr2`) provides a concise way to write conditional expressions. It takes a boolean condition (`cond`) and two expressions (`expr1` and `expr2`). If the condition is `true`, the first expression (`expr1`) is evaluated and its value is returned. If the condition is `false`, the second expression (`expr2`) is evaluated and its value is returned. For example, `x < 0 ? -x : x` returns the absolute value of `x`.

The precedence of the ternary operator is lower than that of assignment operators. For example, in the expression `2 + 3 ? 4 : 5`, the addition is performed first, resulting in `5 ? 4 : 5`, which is `4`. To change the order of operations, parentheses can be used. For example, `(2 + 3) ? 4 : 5` results in `4`.

CS50

Page 1 of 1

CS50

Operators

Overview

Today's agenda: review **operators** (roughly, like a C++ primer, we explore the **arithmetic**, **relational**, **logical**, and **assignment** operators, and the **conditional** and **iteration** statements. Then, we'll look at **arrays** and **strings**, and the **algorithm** design techniques of **recursion** and **dynamic programming**.

1

2 + 2 = 6;

10

2

4 * 10 = 7;

7

3

10 * 2 = 7;

7

4

10 * 2 = 7;

7

5

10 * 2 = 7;

7

6

10 * 2 = 7;

7

Arithmetic Operators

Arithmetic operators perform mathematical operations on numeric values. The **arithmetic operators** are **addition** (**+**), **subtraction** (**-**), **multiplication** (*****), **division** (**/**), and **modulo** (**%**). The **modulo** operator returns the remainder of an integer division. For example, **10 % 3** is **1**, because **10** divided by **3** is **3** with a remainder of **1**.

7

4 * 10 = 7;

7

8

10 * 2 = 7;

7

9

10 * 2 = 7;

7

10

10 * 2 = 7;

7

11

10 * 2 = 7;

7

Assignment Operators

Assignment operators assign values to variables. There are several types of **assignment operators**: **simple assignment** (**=**), **addition assignment** (**+=**), **subtraction assignment** (**-=**), **multiplication assignment** (***=**), **division assignment** (**/=**), and **modulo assignment** (**%=**). The **simple assignment operator** (**=**) assigns the value of the expression on the right to the variable on the left.

12

7 = 4 * 10;

2

13

7 = 10 * 2;

2

14

7 = 10 * 2;

2

15

7 = 10 * 2;

2

16

7 = 10 * 2;

2

Arrays and Strings

Arrays are a collection of **elements** of the same type. They are used to store multiple values of the same type under a single name. **Strings** are a sequence of **characters** used to represent text. They are stored as **arrays of characters** in memory.

17

7 = 10 * 2;

2

18

7 = 10 * 2;

2

19

7 = 10 * 2;

2

20

7 = 10 * 2;

2

21

7 = 10 * 2;

2

CS50

Operators

Overview

Today's agenda: review **operators** (roughly, like a C++ primer, we explore the **arithmetic**, **relational**, **logical**, and **assignment** operators, and the **conditional** and **iteration** statements. Then, we'll look at **arrays** and **strings**, and the **algorithm** design techniques of **recursion** and **dynamic programming**.

1

2 + 2 = 6;

10

2

4 * 10 = 7;

7

3

10 * 2 = 7;

7

4

10 * 2 = 7;

7

5

10 * 2 = 7;

7

6

10 * 2 = 7;

7

Arithmetic Operators

Arithmetic operators perform mathematical operations on numeric values. The **arithmetic operators** are **addition** (**+**), **subtraction** (**-**), **multiplication** (*****), **division** (**/**), and **modulo** (**%**). The **modulo** operator returns the remainder of an integer division. For example, **10 % 3** is **1**, because **10** divided by **3** is **3** with a remainder of **1**.

7

4 * 10 = 7;

7

8

10 * 2 = 7;

7

9

10 * 2 = 7;

7

10

10 * 2 = 7;

7

11

10 * 2 = 7;

7

Assignment Operators

Assignment operators assign values to variables. There are several types of **assignment operators**: **simple assignment** (**=**), **addition assignment** (**+=**), **subtraction assignment** (**-=**), **multiplication assignment** (***=**), **division assignment** (**/=**), and **modulo assignment** (**%=**). The **simple assignment operator** (**=**) assigns the value of the expression on the right to the variable on the left.

12

7 = 4 * 10;

2

13

7 = 10 * 2;

2

14

7 = 10 * 2;

2

15

7 = 10 * 2;

2

16

7 = 10 * 2;

2

Arrays and Strings

Arrays are a collection of **elements** of the same type. They are used to store multiple values of the same type under a single name. **Strings** are a sequence of **characters** used to represent text. They are stored as **arrays of characters** in memory.

17

7 = 10 * 2;

2

18

7 = 10 * 2;

2

19

7 = 10 * 2;

2

20

7 = 10 * 2;

2

21

7 = 10 * 2;

2

CS50

Operators

Overview

Today's agenda: review **operators** (roughly, like a C++ primer, we explore the **arithmetic**, **relational**, **logical**, and **assignment** operators, and the **conditional** and **iteration** statements. Then, we'll look at **arrays** and **strings**, and the **algorithm** design techniques of **recursion** and **dynamic programming**.

1

2 + 2 = 6;

10

2

4 * 10 = 7;

7

3

10 * 2 = 7;

7

4

10 * 2 = 7;

7

5

10 * 2 = 7;

7

6

10 * 2 = 7;

7

Arithmetic Operators

Arithmetic operators perform mathematical operations on numeric values. The **arithmetic operators** are **addition** (**+**), **subtraction** (**-**), **multiplication** (*****), **division** (**/**), and **modulo** (**%**). The **modulo** operator returns the remainder of an integer division. For example, **10 % 3** is **1**, because **10** divided by **3** is **3** with a remainder of **1**.

7

4 * 10 = 7;

7

8

10 * 2 = 7;

7

9

10 * 2 = 7;

7

10

10 * 2 = 7;

7

11

10 * 2 = 7;

7

Assignment Operators

Assignment operators assign values to variables. There are several types of **assignment operators**: **simple assignment** (**=**), **addition assignment** (**+=**), **subtraction assignment** (**-=**), **multiplication assignment** (***=**), **division assignment** (**/=**), and **modulo assignment** (**%=**). The **simple assignment operator** (**=**) assigns the value of the expression on the right to the variable on the left.

12

7 = 4 * 10;

2

13

7 = 10 * 2;

2

14

7 = 10 * 2;

2

15

7 = 10 * 2;

2

16

7 = 10 * 2;

2

Arrays and Strings

Arrays are a collection of **elements** of the same type. They are used to store multiple values of the same type under a single name. **Strings** are a sequence of **characters** used to represent text. They are stored as **arrays of characters** in memory.

[illegible]

CS50

Operators

Overview

One of the primary tasks that **operators** accomplish is to **manipulate** the values of variables. In this section, we'll explore the various operators available in Python, including how to use them to perform arithmetic, logical, and bitwise operations. We'll also discuss how operators can be used to control the flow of a program and to manipulate data structures.

There are three categories of operators:

1) Arithmetic operators	30
2) Comparison operators	1
3) Assignment operators	7
4) Logical operators	3
5) Bitwise operators	28
6) Identity operators	1
7) Membership operators	1

Arithmetic Operators

Arithmetic operators are used to perform mathematical functions on numerical values. They include operators for addition, subtraction, multiplication, division, and modulus. These operators are used to perform basic arithmetic operations on variables and constants.

When working with variables and displaying the results, it's important to remember that Python uses floating-point arithmetic by default. This means that even if you perform integer operations, the result will be a float. For example, `5 + 3` will result in `8.0` instead of `8`.

Python also provides operators for integer division (`//`) and modulus (`%`). Integer division returns the quotient of two numbers as an integer, while the modulus operator returns the remainder of a division. For example, `5 // 3` results in `1`, and `5 % 3` results in `2`.

FUNCTIONS

Assignment Operators

Python provides **assignment operators**, which provide a variety of methods for assigning values to variables. These operators are used to assign values to variables, and they can be used to perform complex assignments. The most common assignment operator is the simple assignment operator (`=`), which assigns a value to a variable.

Python also provides operators for augmented assignment, which allow you to perform operations on a variable and assign the result back to the same variable. These operators include `+=`, `-=`, `*=`, `/=`, `%=`, `//=`, and `**=`. For example, `x += 1` is equivalent to `x = x + 1`.

Another useful assignment operator is the **chained assignment operator**, which allows you to assign multiple variables at once. For example, `x = y = z = 1` assigns the value `1` to all three variables. This is useful for initializing multiple variables with the same value.

Python also provides the **identity operator** (`is`), which is used to check if two variables refer to the same object in memory. This operator is useful for checking if a variable is `None` or if two objects are the same. For example, `x is None` checks if `x` is `None`, and `x is y` checks if `x` and `y` refer to the same object.

Finally, Python provides the **membership operator** (`in`), which is used to check if a value is a member of a sequence (such as a list, tuple, or string). This operator is useful for checking if a value is present in a collection. For example, `1 in [1, 2, 3]` returns `True`, while `4 in [1, 2, 3]` returns `False`.

Key Terms

- **operator**
- **manipulate**
- **variables**
- **arithmetic**
- **logical**
- **bitwise**

$x = y + z$	2
$x = y + z$	3
$x = y + z$	4
$x = y + z$	5
$x = y + z$	6
$x = y + z$	7
$x = y + z$	8
$x = y + z$	9
$x = y + z$	10
$x = y + z$	11
$x = y + z$	12
$x = y + z$	13
$x = y + z$	14
$x = y + z$	15
$x = y + z$	16
$x = y + z$	17
$x = y + z$	18
$x = y + z$	19
$x = y + z$	20
$x = y + z$	21
$x = y + z$	22
$x = y + z$	23
$x = y + z$	24
$x = y + z$	25
$x = y + z$	26
$x = y + z$	27
$x = y + z$	28
$x = y + z$	29
$x = y + z$	30

2/2/2020

David

<https://www.dropbox.com/sh/5y662ey1hc4sde4/AACTZ9s1PxHcq1nQ-qLOSmKDa/ASCII.pdf?dl=0>

https://www.dropbox.com/sh/5y662ey1hc4sde/4AAANVfQjHejlkx_HhI3rm4DSa/Arrays%20and%20Strings.pdf?dl=0

<https://www.dropbox.com/sh/5y662ey1hc4sde4/AADuQLORUhJBC8ITuLvCp5cTa/Command-Line%20Interaction.pdf?dl=0>

<https://www.dropbox.com/sh/5y662ey1hc4sde4/AABW3PttM7UdTY4nyTS1tePa/Functions.pdf?dl=0>

COME HAVE BRUNCH WITH ME!

This Sunday @ **11am** in **Winthrop Dining Hall**.

Survey Link

Finish the survey if you haven't already or are new to the section:

tinyurl.com/y7mxcdnu