CS50 Section

Week 2

Attendance Sign In: https://tinyurl.com/cs50ssi917

Hello!

I’m Raymond.

- Senior / Currier House
- From Vancouver, BC 🇨🇦
- Took CS50 Freshman Year
- Studies CS (Mind, Brain, Behavior Track), Economics Secondary
- Likes AI, ML, Marketplaces, IoT, Neuro, Synthetic Bio, Aerospace
- Can’t wait to experience CS50 with y’all!

Agenda

- How to CS50 & Section Logistics
- Week 1 Review
- Week 2 Key Topics + Coding Exercises
  - Arrays
  - Strings
  - Command Line Arguments

ROUNDTABLE!

INTROS

(Name, Year, Scratch Project, one thing you want to get out of CS50)

HOW TO CS50

GOALS OF THE COURSE

1. Learn to think computationally and algorithmically.
2. Improve your problem-solving skills—this will help you in every other course and life in general!
3. Gain practical programming skills in a bunch of languages: C, Python, SQL, etc.
4. Familiarize yourself with best practices for software design.
5. Build a community that will stay with you after this course.

COMPONENTS TO CS50

1. Lectures/Section + Quiz
2. Problem Sets
3. Test (Nov 4)
4. Final Projects: Web / Mobile / Game

You'll spend ~75% of your time in the class here.

Resources

- Lecture Notes
- Problem Set Walkthrough Videos
- “Shorts” and Section Practice Problems
- Peers! (Under Collab Policy)
- CS50 Discussion via Ed
- Section
- Tutorials and Sunday Office Hours

Tools

- check50 - Feedback on Code Correctness
- style50 - Feedback on Code Style
- help50 - Helps to Understand Error Messages
- man.cs50.io - Reference Documentation
- Debug50 - Helps with Code Debugging in CS50 IDE (Introduced Later)

A Week in the Life

- Lecture on **Monday**.
- (Optional) supersection on **Monday**.
- Quizzes due on **Tuesday**.
- Section on **Tuesday** or **Wednesday**.
- (Optional) tutorials on **Wednesday – Saturday**.
- (Optional) office hours on **Sunday**.
- Problem set due on **Sunday**.

What we'll Typically Cover in Section

- Questions and Clarifications from Previous Week
- Review of Key Concepts from Lecture:
  - Come with any questions!
- Practice Problems
- Workshops (Debugging, Improving Design, etc.)
- *Your Requests!*
  - Send me an email ~24 Hours before section if there is something you want to cover

Attending Section

- You have to attend section, but it doesn't have to be this one
  - Just make sure you get marked off for attendance each week
- Tip: Shop different sections and find the one that best fits your learning style!

Getting in Touch

raymond_wang@college.harvard.edu

(617) 949-6919

Let’s Get Started

Week 1 Review

Week 1 Review - What Questions do You Have?

- Variables
- Typing
- Primitives
- Qualifiers
- Arithmetic Operators
- Shortcut Arithmetic Operators
- Other Operators
- Conditional Statements
- Switch Statement
- Loops

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 encouraged to think about design, which tries to encapsulate a small cross-section of all the above tradeoffs

STYLE TIPS

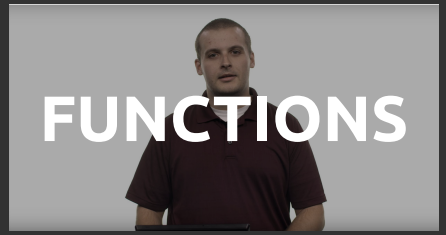
1. Use descriptive variable names (no `x`, `y`, `z`, unless it is used in a for loop and/or as a counter value)
2. Format your code cleanly—Keep your indentation, tabs, spaces, etc. clear
    a. CS50 Style guide: https://cs50.readthedocs.io/style/c/
3. Use comments to explain non-obvious parts of your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

Week 2 Review

Today We'll Cover

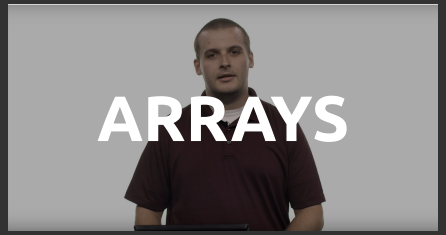
- Arrays
- Strings
- Command Line Arguments

HELPFUL “SHORTS” FOR THE WEEK



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

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



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

https://www.youtube.com/watch?v=w4TAY2HPLEg

Arrays

ARRAYS - HOW DO WE DECLARE THEM?

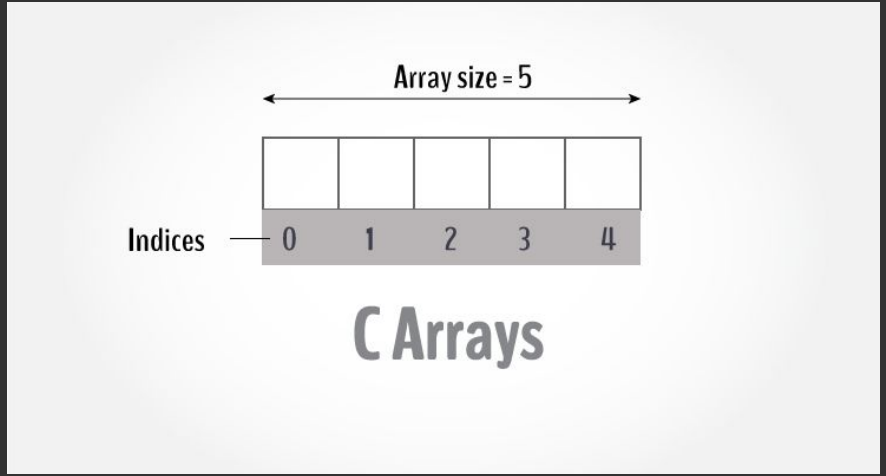
```
type name[size];
```

- 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 index value
- Note that arrays are indexed from zero—If want to access the nth element, we use `name[n-1]`

ARRAYS - OUT OF BOUNDS

```c
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

```c
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

```c
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]);
```

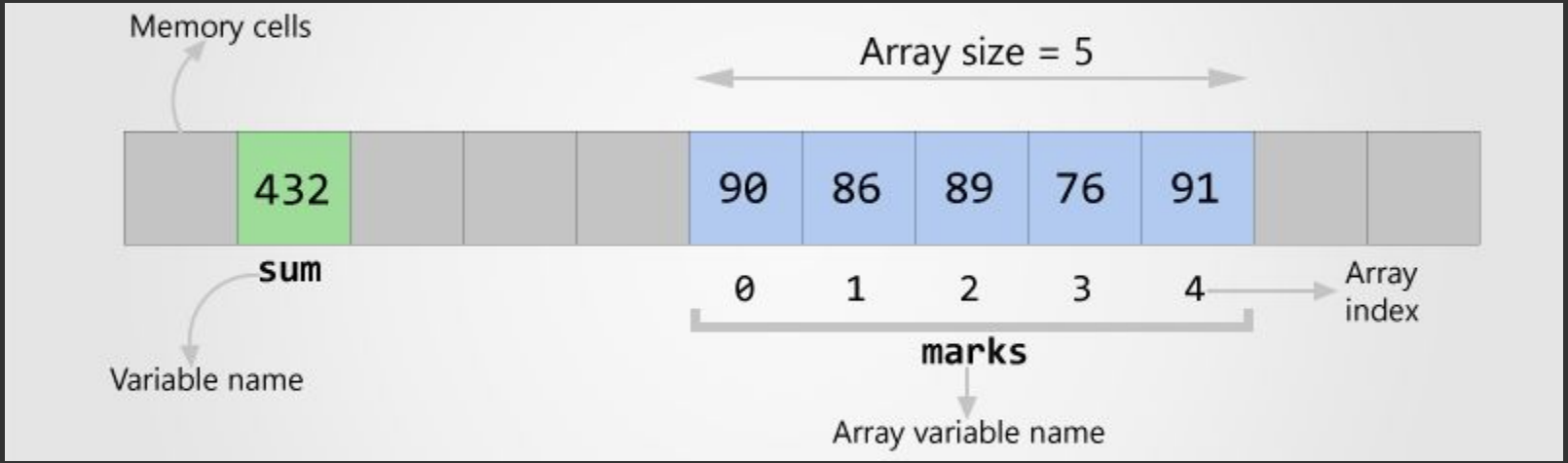
Other times you'll get an error:

```
Segmentation fault
```

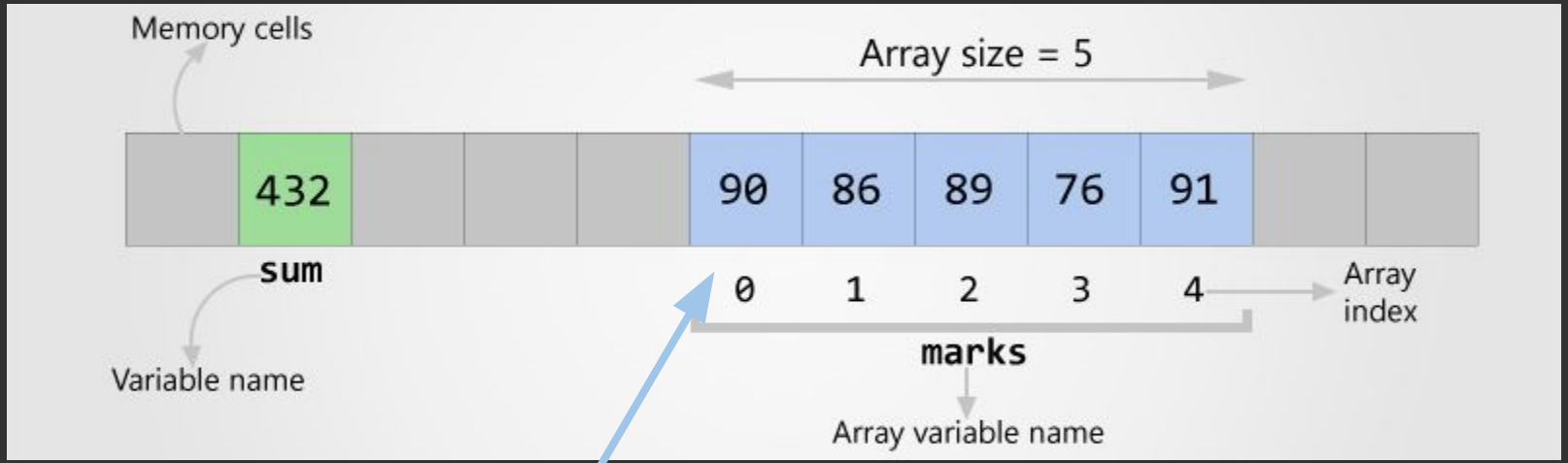
This means you're trying to read/write to an illegal memory location

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:

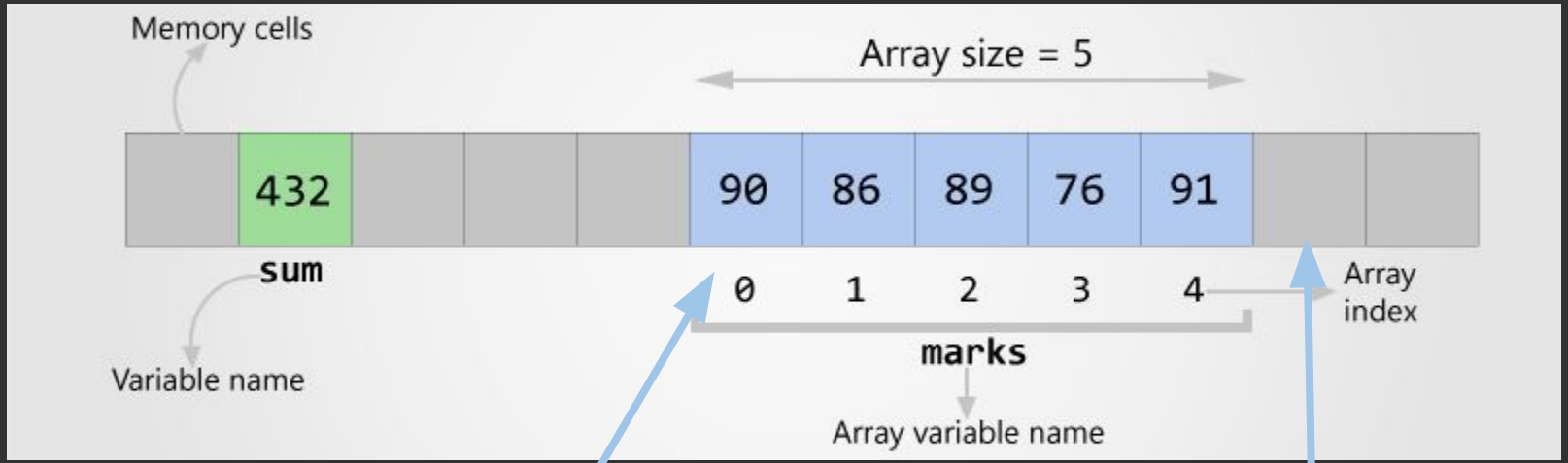


ARRAYS - UNDER THE HOOD



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?”

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.

Strings

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

`s[0]`

`s[1]`

Pair Exercise:
Reverse the String

http://bit.ly/2V0AIcu

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

Solution

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

int main(void)
{
    string s = get_string("Text: ");

    // Loop through string in reverse order
    for (int i = strlen(s) - 1; i >= 0; i--)
    {
        printf("%c", s[i]);
    }

    printf("\n");
}
```

ASCII

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Character Representation

Displayed	C	S	5	0	\0
In ASCII	43	53	35	30	0

Command-Line Arguments

Command-Line Arguments

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


}
```

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

```
./caesar 2
```

argv[0] argv[1]

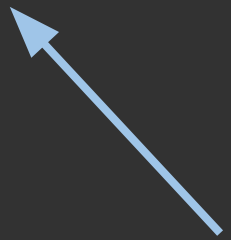
COMMAND LINE ARGUMENTS

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

COMMAND LINE ARGUMENTS

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

`argc` would be 4

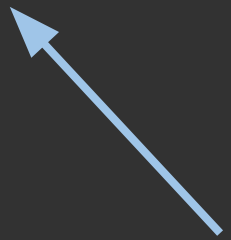


argv[0]	argv[1]	argv[2]	argv[3]
“main”	“x”	“y”	“z”

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".

PROBLEM SET 2 PREVIEW

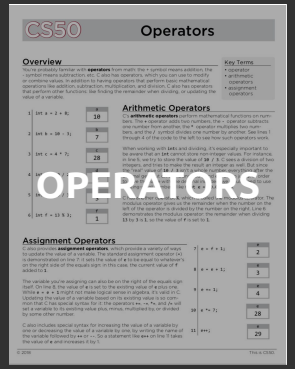
Due Sun 9/22 @ 11:59pm

You will need to complete:

- **`Readability`**
- `One of` **`Caesar`** `or` **`Substitution`**

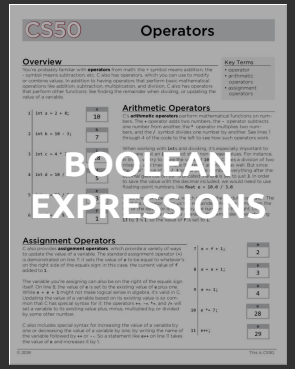
APPENDIX - BASICS FROM WEEK 1

REFERENCE SHEETS



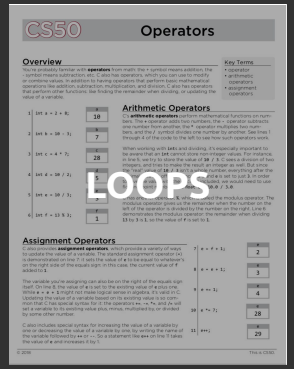
OPERATORS

https://www.dropbox.com/sh/5y662ey1hc4sde4/AABpC6MbC5rzo81wNK9CP_NZa/Operators.pdf?dl=0



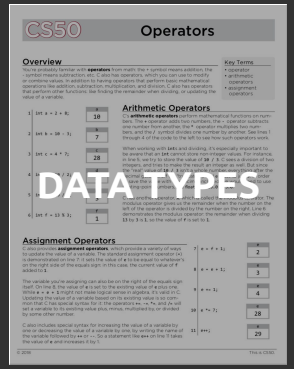
BOOLEAN EXPRESSIONS

https://www.dropbox.com/sh/5y662ey1hc4sde4/AAD30Rk8Fcmz_HG2hE-_bXoRa/Boolean%20Expressions.pdf?dl=0



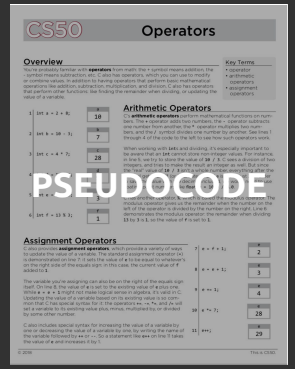
LOOPS

https://www.dropbox.com/sh/5y662ey1hc4sde4/AAA3J_QHkJ5GFeTi2YuEJplYa/Loops.pdf?dl=0



DATA TYPES

https://www.dropbox.com/sh/5y662ey1hc4sde4/AAC10N2PXZrLldLKZz21hCp2a/Data%20Types.pdf?dl=0



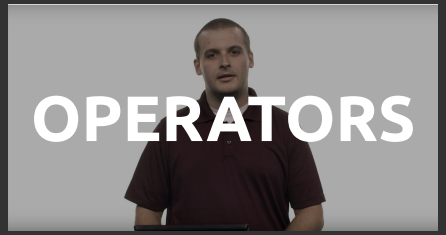
PSEUDOCODE

https://www.dropbox.com/sh/5y662ey1hc4sde4/AAAac4DxJ3fRQiaohQ3dts65a/Pseudocode.pdf?dl=0

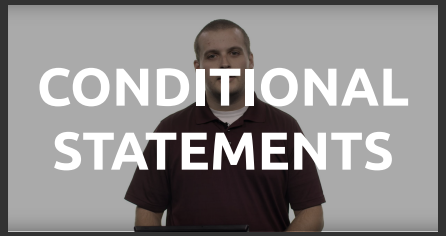
Some Shorts

https://www.youtube.com/watch?v=lnYKOnz9ln8

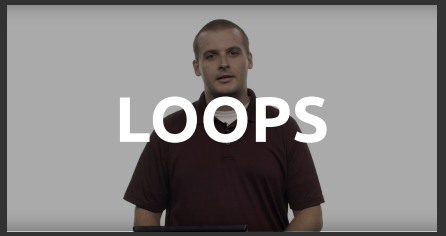
https://www.youtube.com/watch?v=q6K8KMqt8wQ



https://www.youtube.com/watch?v=7apBtlEkJzk



https://www.youtube.com/watch?v=FqUeHzvci10



https://www.youtube.com/watch?v=QOvo-xFL9II

VARIABLES - THE BASICS

```c
int main(void)
{
    int x = 14;

    int y;
    y = 14;
}
```

What is the difference between these two approaches?

VARIABLES - THE BASICS

```c
int main(void)
{
    int x = 14;

    int y;
    y = 14;
}
```

What is the difference between these two approaches?

In the first one, we declare and initialize a variable at the same time (*instantiation*). In the second one, we separate these operations.

VARIABLES - TYPING

C is a **strongly typed** language - Every variable you declare must include a type associated with it.

What is the purpose for this? What advantages and disadvantages does this confer?

VARIABLES - TYPING

C is a **strongly typed** language - Every variable you declare must include a type associated with it.

What is the purpose for this? What advantages and disadvantages does this confer?

VARIABLES - PRIMITIVES

DATA TYPE	RANGE	MEMORY
`int`	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647	2 or 4 bytes
`char`	-128 to 127 or 0 to 255	1 byte
`long`	-2,147,483,648 to 2,147,483,647	4 bytes
`float`	1.2E-38 to 3.4E+38	4 bytes
`double`	2.3E-308 to 1.7E+308	8 bytes

VARIABLES - PRIMITIVES

DATA TYPE	RANGE	MEMORY
`int`	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647	2 or 4 bytes
`char`	-128 to 127 or 0 to 255	1 byte
`long`	-2,147,483,648 to 2,147,483,647	4 bytes
`float`	1.2E-38 to 3.4E+38	4 bytes
`double`	2.3E-308 to 1.7E+308	8 bytes

VARIABLES - PRIMITIVES

DATA TYPE	RANGE	MEMORY
`int`	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647	2 or 4 bytes
`char`	-128 to 127 or 0 to 255	1 byte
`long`	-2,147,483,648 to 2,147,483,647	4 bytes
`float`	1.2E-38 to 3.4E+38	4 bytes
`double`	2.3E-308 to 1.7E+308	8 bytes

?

?

VARIABLES - QUALIFIERS

- The `unsigned` qualifier means you have NO signs on your data type (i.e. you can't have negatives)
- The `signed` qualifier allows you to have signs

VARIABLES - QUALIFIERS

- The **`unsigned`** qualifier means you have NO signs on your data type (i.e. you can't have negatives)
- The **`signed`** qualifier allows you to have signs

How might we find the size of a data type in C?

Arithmetic Operators

Let’s brainstorm together: What arithmetic operators do we have in C?

Arithmetic Operators

Let’s brainstorm together: What arithmetic operators do we have in C?

+ - * / %

Arithmetic Operators - Shortcuts

Take advantage of shortcuts:

```
int main(void)
{
    int x;

    x = x + 1;
    x += 1;
    x++;
}
```

Other Operators

TYPE	OPERATOR	PURPOSE
Logical	&& || !	AND OR NOT
Relational	< <= >= > == !=	LESS THAN LESS THAN OR EQUAL TO GREATER THAN OR EQUAL TO GREATER THAN EQUALS NOT EQUAL TO

Other Operators

TYPE	OPERATOR	PURPOSE
Logical	&& || !	AND OR NOT
Relational	< <= >= > == !=	LESS THAN LESS THAN OR EQUAL TO GREATER THAN OR EQUAL TO GREATER THAN EQUALS NOT EQUAL TO

What's the difference between = and == in C?

MORE ON OPERATORS

What is `!(P && Q)` equivalent to?

MORE ON OPERATORS

What is `!(P && Q)` equivalent to?

`!P || !Q`

How about `!(P || Q)`?

MORE ON OPERATORS

What is `!(P && Q)` equivalent to?

```
!P || !Q
```

These are called De Morgan's Laws.

How about `!(P || Q)`?

```
!P && !Q
```

CONDITIONAL STATEMENTS

How do we actually use those logical operators we just learned?

CONDITIONAL STATEMENTS

How do we actually use those logical operators we just learned?

By writing boolean expressions and utilizing conditional statements!

```
if (condition)
   {
       // code goes here if the
condition is met
   }
```

```
if (condition)
   {
       // code goes here
   }
else
   {
       // if condition fails
   }
```

```
if (condition)
   {
       // code goes here
   }
else if (another condition)
   {
       // if condition fails
   }
else
   {
```

CONDITIONAL STATEMENTS - PRACTICE

How might I write a statement that prints “hello, world” to the screen only if the user enters an integer greater than 10?

CONDITIONAL STATEMENTS - PRACTICE

How might I write a statement that prints “hello, world” to the screen only if the user enters an integer greater than 10?

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

int main(void)
{
    int x = get_int("Please enter an integer: ");

    if(x > 10) {
        printf("hello, world\n");
    }
}
```

THE SWITCH STATEMENT

```c
#include <stdio.h>
int main () {

  /* local variable definition */
  char grade = 'B';

  switch(grade) {
     case 'A' :
        printf("Excellent!\n" );
        break;
     case 'B' :
        printf("Well done\n" );
        break;
     case 'C' :
        printf("You passed\n" );
        break;
     case 'D' :
        printf("Better try again\n" );
        break;
     default :
        printf("Invalid grade\n" );
  }

  printf("Your grade is  %c\n", grade );
  return 0;
}
```

THE SWITCH STATEMENT

```c
#include <stdio.h>
int main () {

  /* local variable definition */
  char grade = 'B';

  switch(grade) {
     case 'A' :
        printf("Excellent!\n" );
        break;
     case 'B' :
        printf("Well done\n" );
        break;
     case 'C' :
        printf("You passed\n" );
        break;
     case 'D' :
        printf("Better try again\n" );
        break;
     default :
        printf("Invalid grade\n" );
  }

  printf("Your grade is  %c\n", grade );
  return 0;
}
```

Must be constant values like a character or number

THE SWITCH STATEMENT

```c
#include <stdio.h>
int main () {

  /* local variable definition */
  char grade = 'B';

  switch(grade) {
     case 'A' :
        printf("Excellent!\n" );
        break;
     case 'B' :
        printf("Well done\n" );
        break;
     case 'C' :
        printf("You passed\n" );
        break;
     case 'D' :
        printf("Better try again\n" );
        break;
     default :
        printf("Invalid grade\n" );
  }

  printf("Your grade is  %c\n", grade );
  return 0;
}
```

Must be constant values like a character or number

What does the break statement do?

THE SWITCH STATEMENT

```c
#include <stdio.h>
int main () {

  /* local variable definition */
  char grade = 'B';

  switch(grade) {
     case 'A' :
        printf("Excellent!\n" );
        break;
     case 'B' :
        printf("Well done\n" );
        break;
     case 'C' :
        printf("You passed\n" );
        break;
     case 'D' :
        printf("Better try again\n" );
        break;
     default :
        printf("Invalid grade\n" );
  }

  printf("Your grade is  %c\n", grade );
  return 0;
}
```

The break statement tells C to exit the switch statement. Without it, the program would flow into the other cases and execute their code as well.

THE SWITCH STATEMENT

```c
#include <stdio.h>
int main () {

  /* local variable definition */
  char grade = 'B';

  switch(grade) {
     case 'A' :
        printf("Excellent!\n" );
        break;
     case 'B' :
        printf("Well done\n" );
        break;
     case 'C' :
        printf("You passed\n" );
        break;
     case 'D' :
        printf("Better try again\n" );
        break;
     default :
        printf("Invalid grade\n" );
  }

  printf("Your grade is  %c\n", grade );
  return 0;
}
```

When would we want to use a switch statement vs. just if-else statements?

CONDITIONAL STATEMENTS - A SHORTCUT

We have the **ternary operator** in C to allow us to shorten our if-else statements:

```
if (a > b) {
    result = x;
}
else {
    result = y;
}
```

```
result = a > b ? x : y;
```

COMPOUND BOOLEAN EXPRESSIONS

How do we express two different boolean conditions?

COMPOUND BOOLEAN EXPRESSIONS

How do we express two different boolean conditions?

Using our logical operators! For example:

```
((x > 15) && (y == 7))
```

COMPOUND BOOLEAN EXPRESSIONS

Note that C reads left-to-right like in English:

```
((x > 15) && (y == 7) || (z > 12))
```

This will calculate the boolean expression off the first two parts and then compare that result with the `||` to the last part.

LOOPS

We have three types of loops in C:

- `while` loops
- `do-while` loops
- `for` loops

What is the difference between each of these?

LOOPS

while	do-while	for
Checks for a condition at the start of each time the loop runs	Runs the code in the loop body and then checks the condition	Checks a condition prior to running the loop, runs the code in the loop body, increments the counter, and then repeats

```c
int i = 0;
while (i < 10)
{
  printf("%i\n", i);
  i++;
}
```

```c
int j = 0;
do
{
  printf("%i\n", j);
  j++;
}
while (j < 10);
```

```c
for (int k = 0; k < 10; k++)
{
  printf("%i\n", k);
}
```

LOOPS

When do we use each different type of loop?

LOOPS

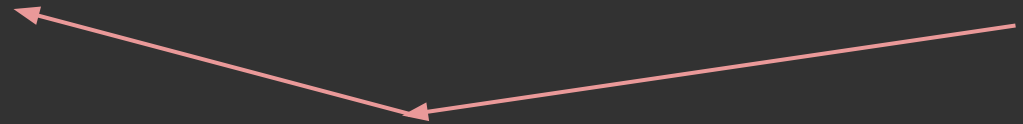
When do we use each different type of loop?

`while`	`do-while`	`for`
You want to run the code in the loop body until there is a state change (*nondeterministic*)	You want to run the code in the loop body until there is a state change, but guarantee the code runs at least once	You want to run the code in the loop body for a predetermined number of times (*deterministic*)

LOOPS

When do we use each different type of loop?

`while`	`do-while`	`for`
You want to run the code in the loop body until there is a state change (*nondeterministic*)	You want to run the code in the loop body until there is a state change, but guarantee the code runs at least once	You want to run the code in the loop body for a predetermined number of times (*deterministic*)



You can see the for loop as a subset of the while loop