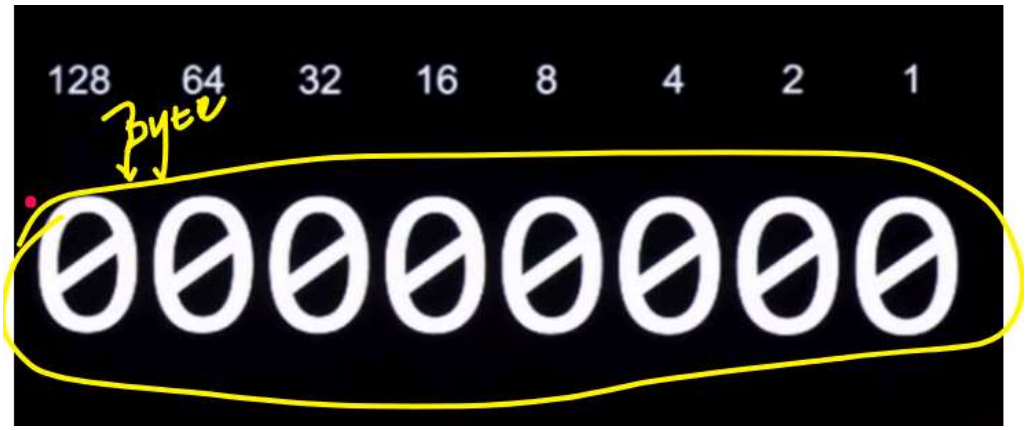
CS 50: week-1

Lecture -0

🔗 Cs50 Codespace & Scratch.Mit.edu

base-1 : Unary,
base-2: Binary digit, (bit)

1 Byte = 8 bits



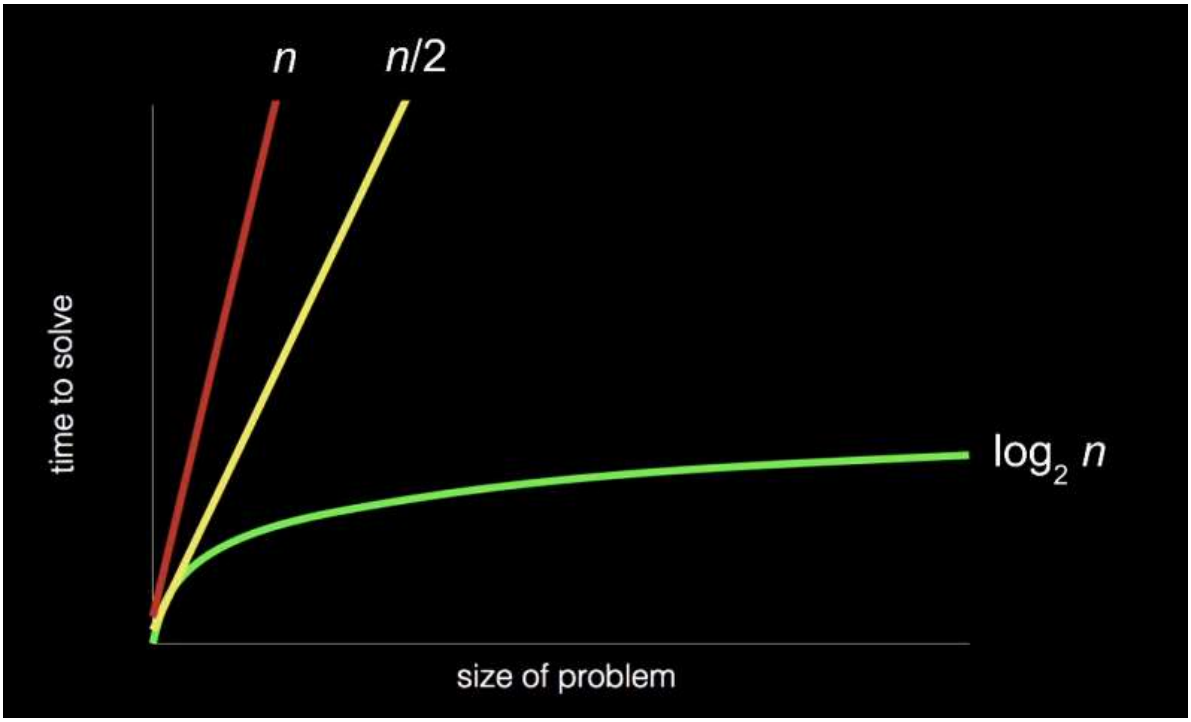
0	1	2	3	4	5	6	7	8	9	0	.	,	!	?	/	"	'	space	_
48	49	50	51	52	53	54	55	56	57	48	46	44	33	63	47	34	39	32	95

\	(	)	*	+	=	-	@	A	B	C	D	E	F	G	H	I	J	K	L
92	40	41	42	43	61	45	64	65	66	67	68	69	70	71	72	73	74	75	76

M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
77	78	79	80	81	82	83	84	85	86	87	88	89	90	97	98	99	100	101	102

g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122

Uppercase - 65=A, 66=B, Lowercase - 97=a, 98=b..**Algorithm** (means steps acquired to solve problems)



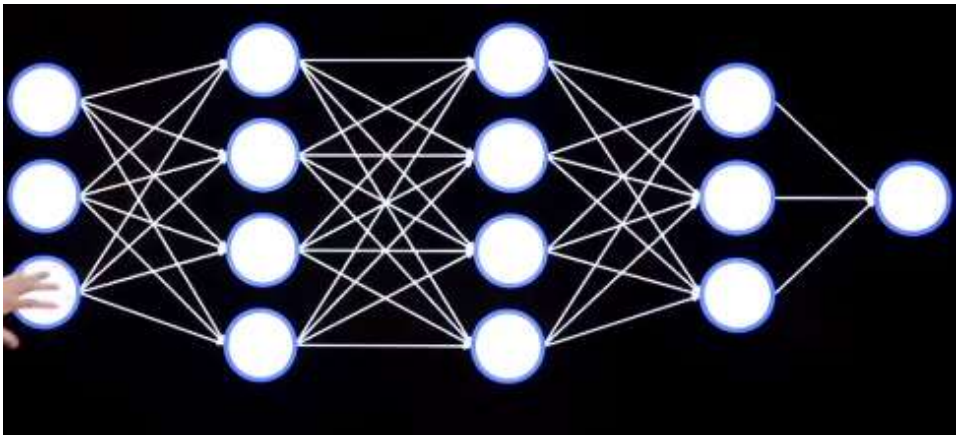
Pseudocode

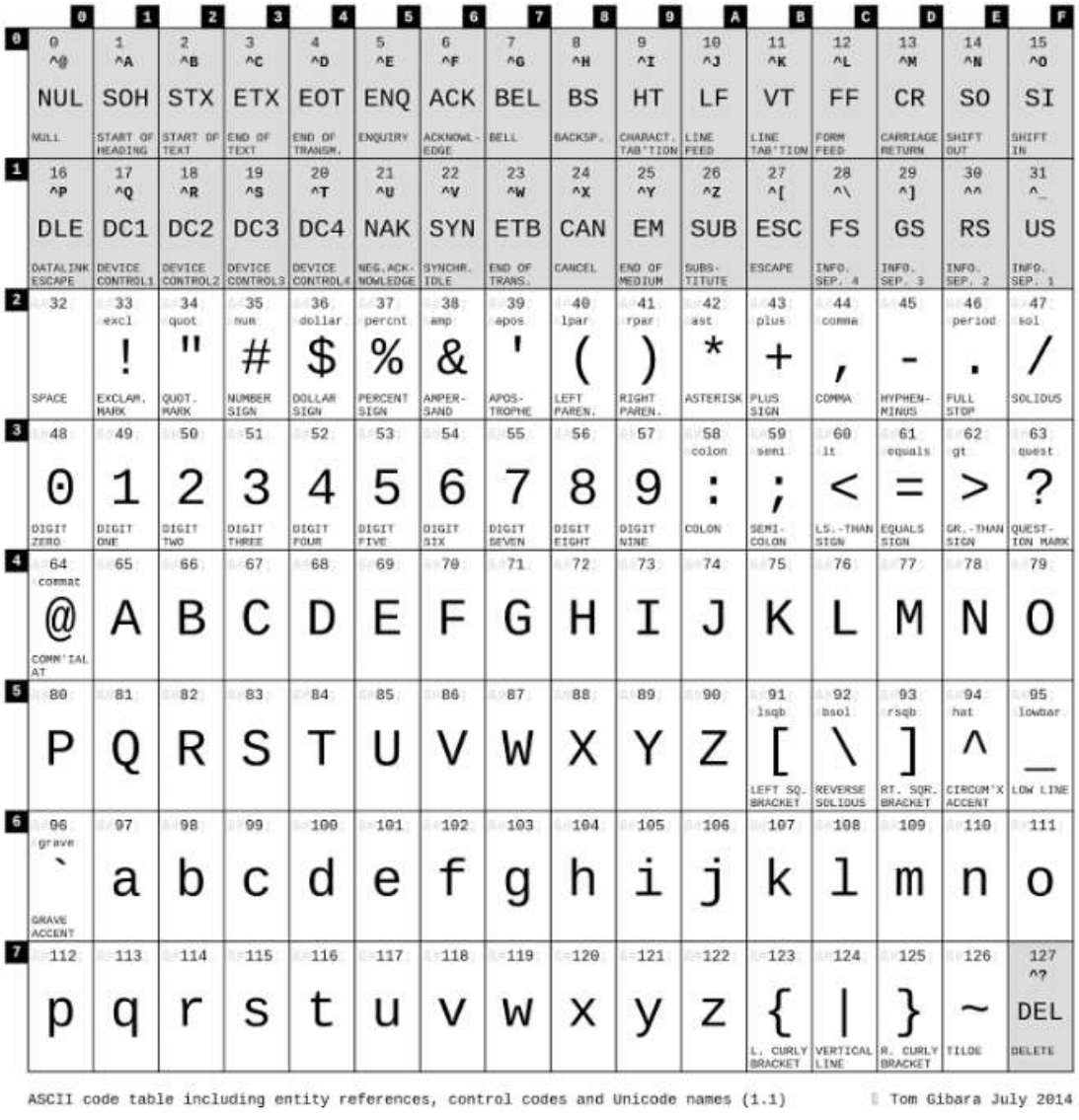
```
1   Pick up phone book
2   Open to middle of phone book
3   Look at page
4   If person is on page
5       Call person
6   Else if person is earlier in book
7       Open to middle of left half of book
8       Go back to line 3
9   Else if person is later in book
10      Open to middle of right half of book
11      Go back to line 3
12  Else
13      Quit
```

- functions
  - arguments, return values, variables
- conditionals
- Boolean expressions
- loops
- ...

Conditional

Boolean Expression

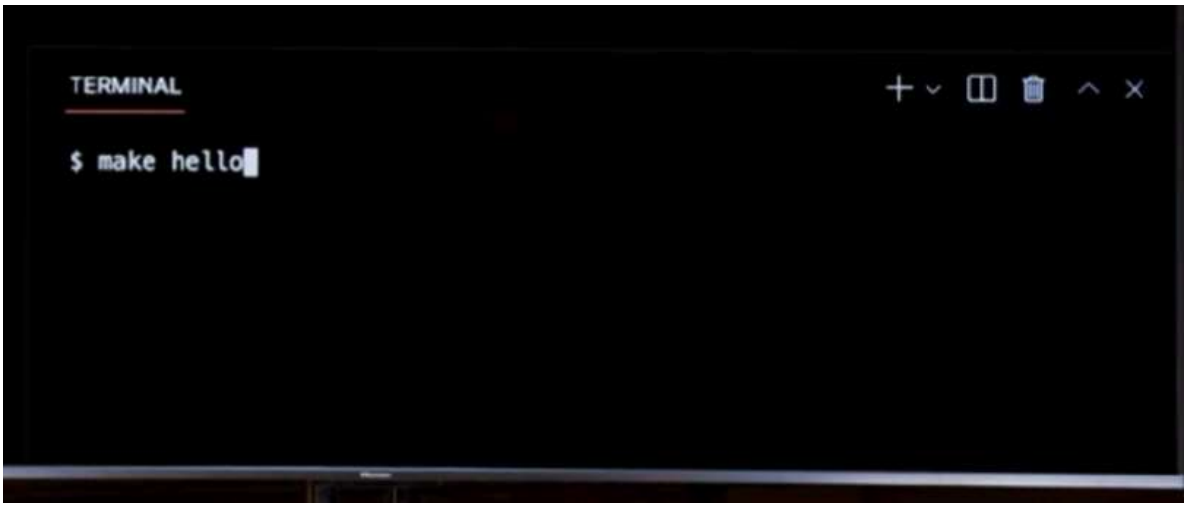




	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0 ^@ NUL NULL	1 ^A SOH START OF HEADING	2 ^B STX START OF TEXT	3 ^C ETX END OF TEXT	4 ^D EOT END OF TRANSM.	5 ^E ENQUIRY	6 ^F ACK ACKNOWL-EDGE	7 ^G BEL BELL	8 ^H BS BACKSP.	9 ^I HT CHARACT. TAB'TION	10 ^J LF LINE FEED	11 ^K VT LINE TAB'TION	12 ^L FF FORM FEED	13 ^M CR CARRIAGE RETURN	14 ^N SO SHIFT OUT	15 ^O SI SHIFT IN
1	16 ^P DLE DATALINK ESCAPE	17 ^Q DC1 DEVICE CONTROL1	18 ^R DC2 DEVICE CONTROL2	19 ^S DC3 DEVICE CONTROL3	20 ^T DC4 DEVICE CONTROL4	21 ^U NAK NEG.ACK-NOWLEDGE	22 ^V SYN SYNCHR. IDLE	23 ^W ETB END OF TRANS.	24 ^X CAN CANCEL	25 ^Y EM END OF MEDIUM	26 ^Z SUB SUBS-TITUTE	27 ^[ESC ESCAPE	28 ^\ FS INFO. SEP. 4	29 ^] GS INFO. SEP. 3	30 ^^ RS INFO. SEP. 2	31 ^_ US INFO. SEP. 1
2	32 SPACE	33 excl ! EXCLAM. MARK	34 quot " QUOT. MARK	35 num # NUMBER SIGN	36 dollar $ DOLLAR SIGN	37 percnt % PERCENT SIGN	38 amp & AMPER-SAND	39 apos ' APOS-TROPHE	40 lpar (LEFT PAREN.	41 rpar) RIGHT PAREN.	42 ast * ASTERISK	43 plus + PLUS SIGN	44 comma , COMMA	45 - HYPHEN-MINUS	46 period . FULL STOP	47 sol / SOLIDUS
3	48 0 DIGIT ZERO	49 1 DIGIT ONE	50 2 DIGIT TWO	51 3 DIGIT THREE	52 4 DIGIT FOUR	53 5 DIGIT FIVE	54 6 DIGIT SIX	55 7 DIGIT SEVEN	56 8 DIGIT EIGHT	57 9 DIGIT NINE	58 colon : COLON	59 semi ; SEMI-COLON	60 lt < LS.-THAN SIGN	61 equals = EQUALS SIGN	62 gt > GR.-THAN SIGN	63 quest ? QUEST-ION MARK
4	64 commat @ COMM'IAL AT	65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
5	80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y	90 Z	91 lsqb [LEFT SQ. BRACKET	92 bsol \ REVERSE SOLIDUS	93 rsqb] RT. SQR. BRACKET	94 hat ^ CIRCUM'X ACCENT	95 lowbar _ LOW LINE
6	96 grave ` GRAVE ACCENT	97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
7	112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 { L. CURLY BRACKET	124 | VERTICAL LINE	125 } R. CURLY BRACKET	126 ~ TILDE	127 ^? DEL DELETE

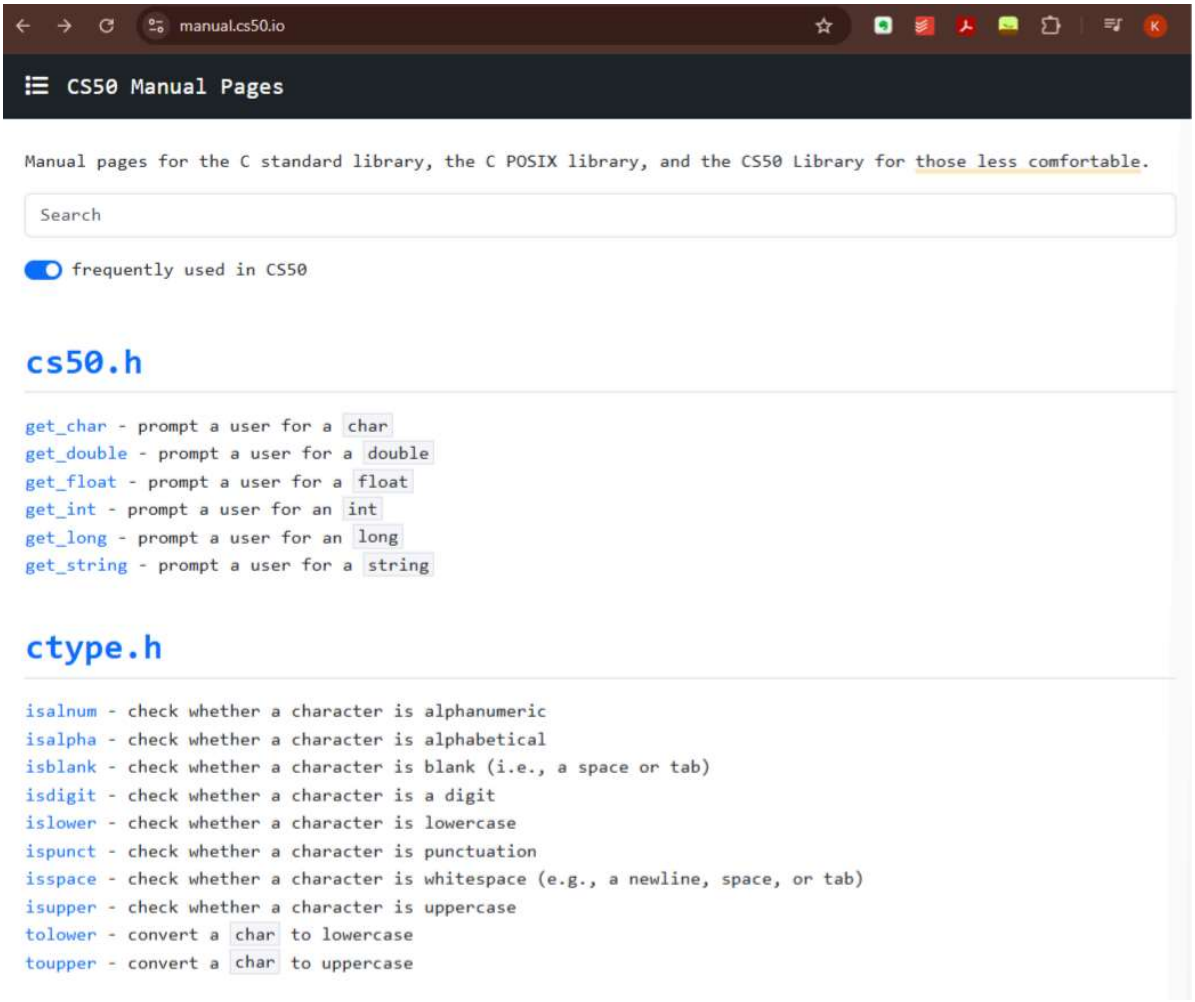
ASCII code table including entity references, control codes and Unicode names (1.1) Tom Gibara July 2014

Week 1




```
#include <stdio.h>
int main(void)
{
printf("hello, world\n");
}
```

https://manual.cs50.io/



https://manual.cs50.io/3/get_char

NAME

less comfortable

get_char - prompt a user for a `char`

SYNOPSIS

less comfortable

Header File

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

Prototype

```
char get_char(string prompt, ...);
```

DESCRIPTION

less comfortable

This function prompts the user for a `char`. If the user inputs more or less than one `char`, the function prompts the user again.

This function expects at least one argument, `prompt`. If `prompt` contains any format codes, a la printf, this function accepts additional arguments as well, one per format code.

RETURN VALUE

less comfortable

This function returns the user's input as a `char`.

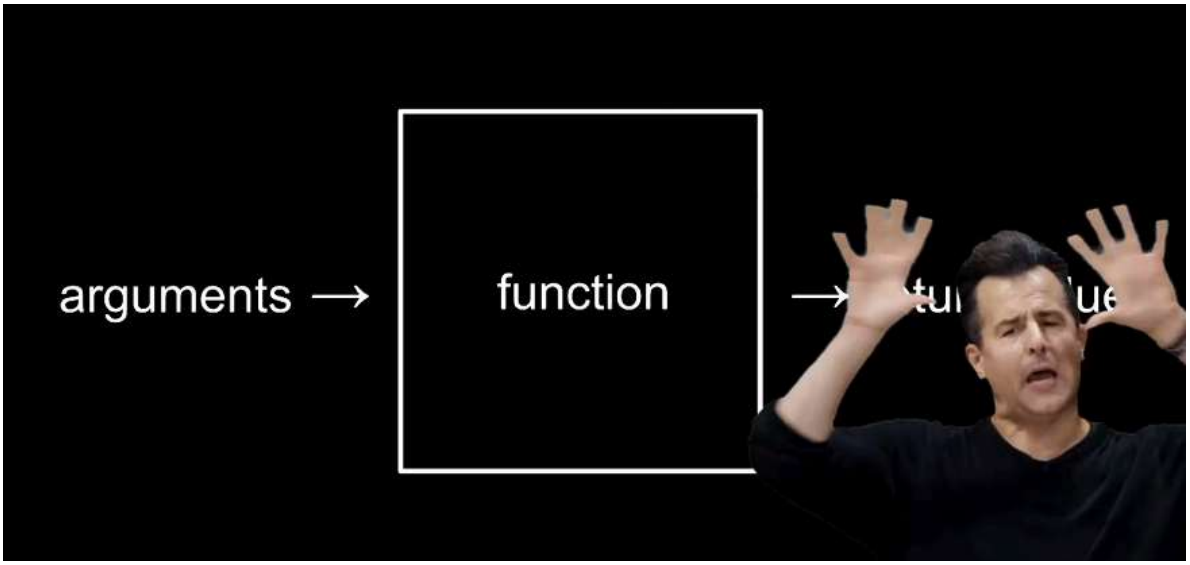
EXAMPLE

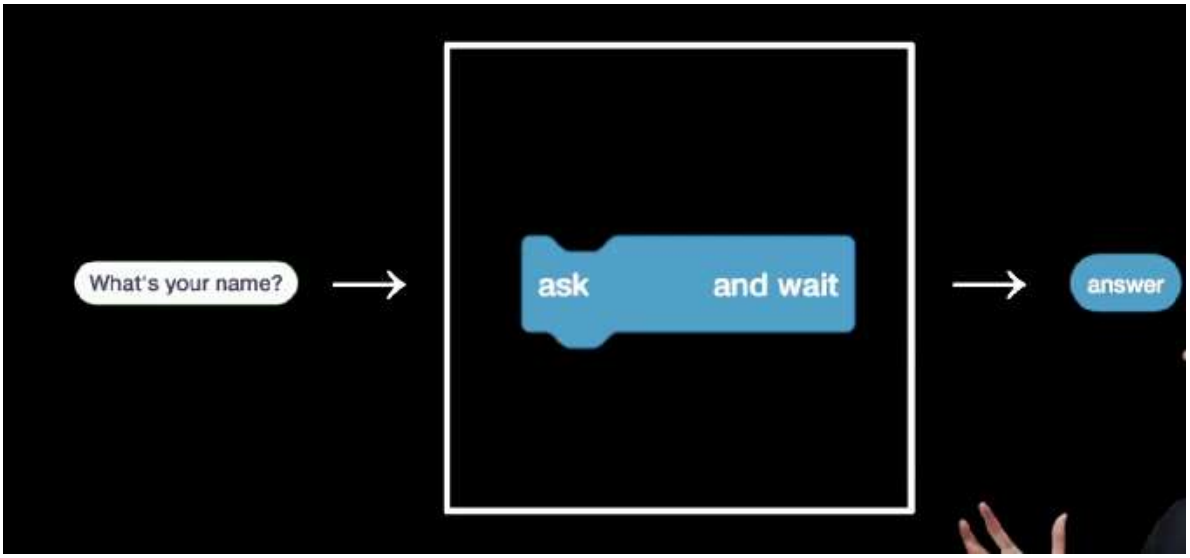
less comfortable

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

int main(void)
{
    char c = get_char("Input: ");
    printf("Output: %c.\n", c);
}
```

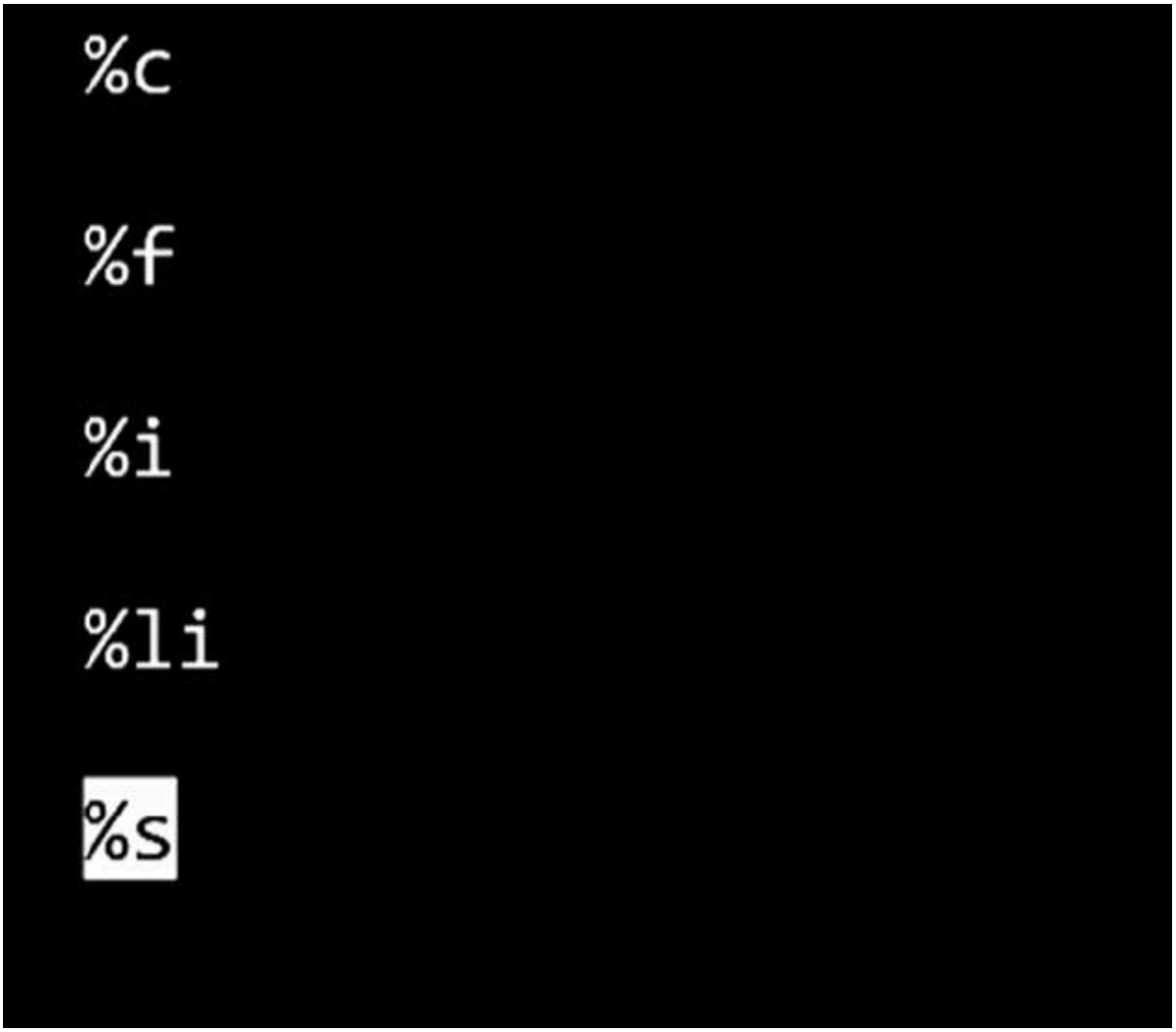
SEE ALSO

```
get_double(3), get_float(3), get_int(3), get_long(3),
get_string(3), printf(3)
```





get_string( )


%c
%f
%i
%li
%s

=

<

<=

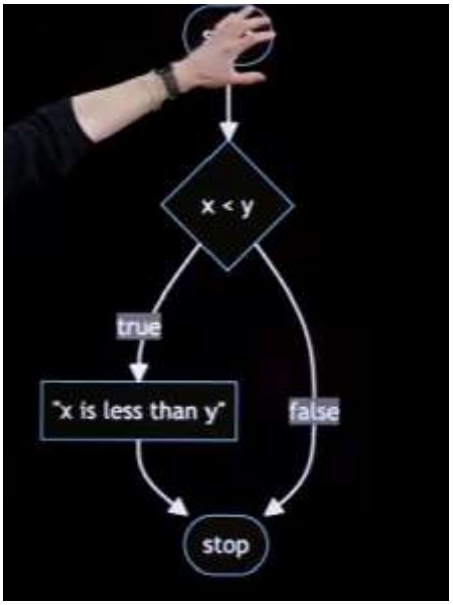
>

>=

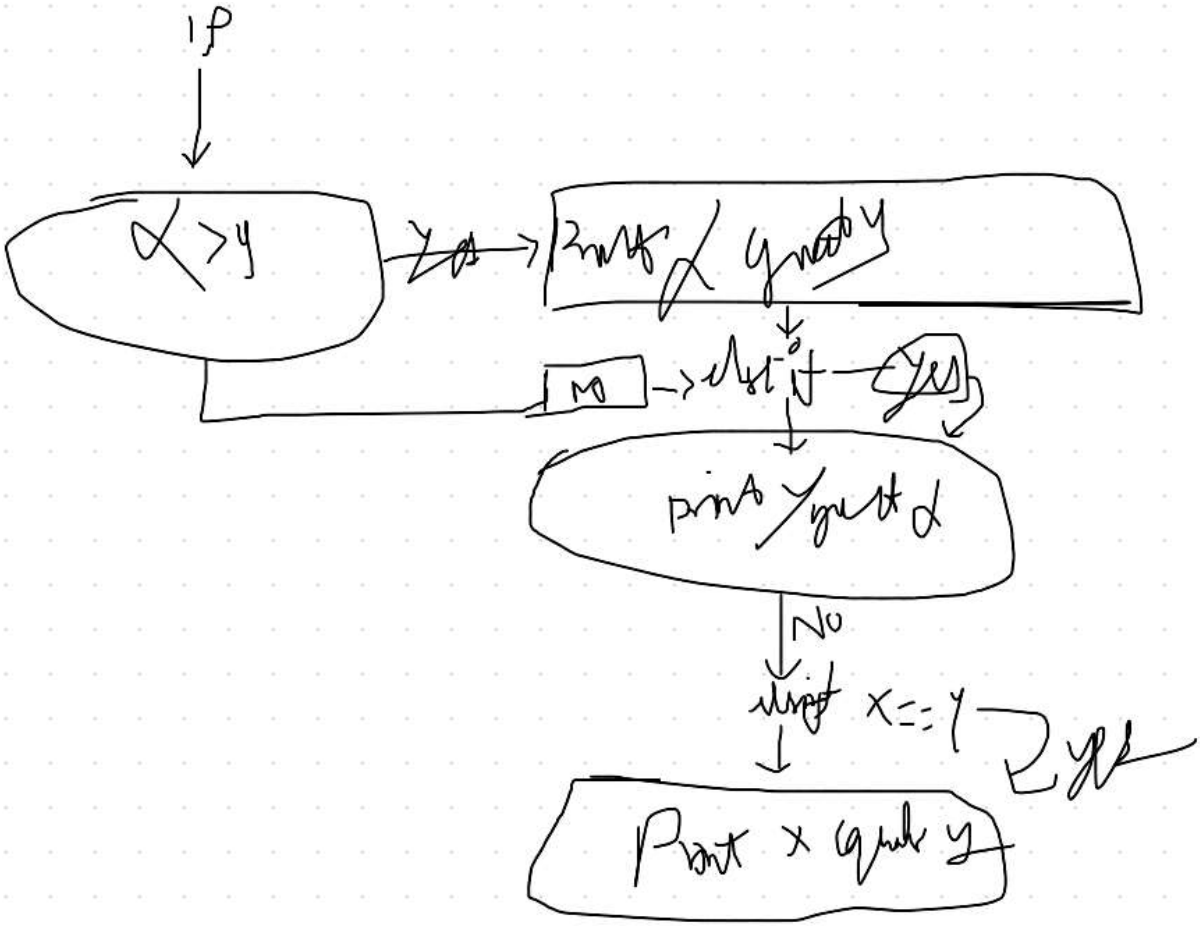
==

!=

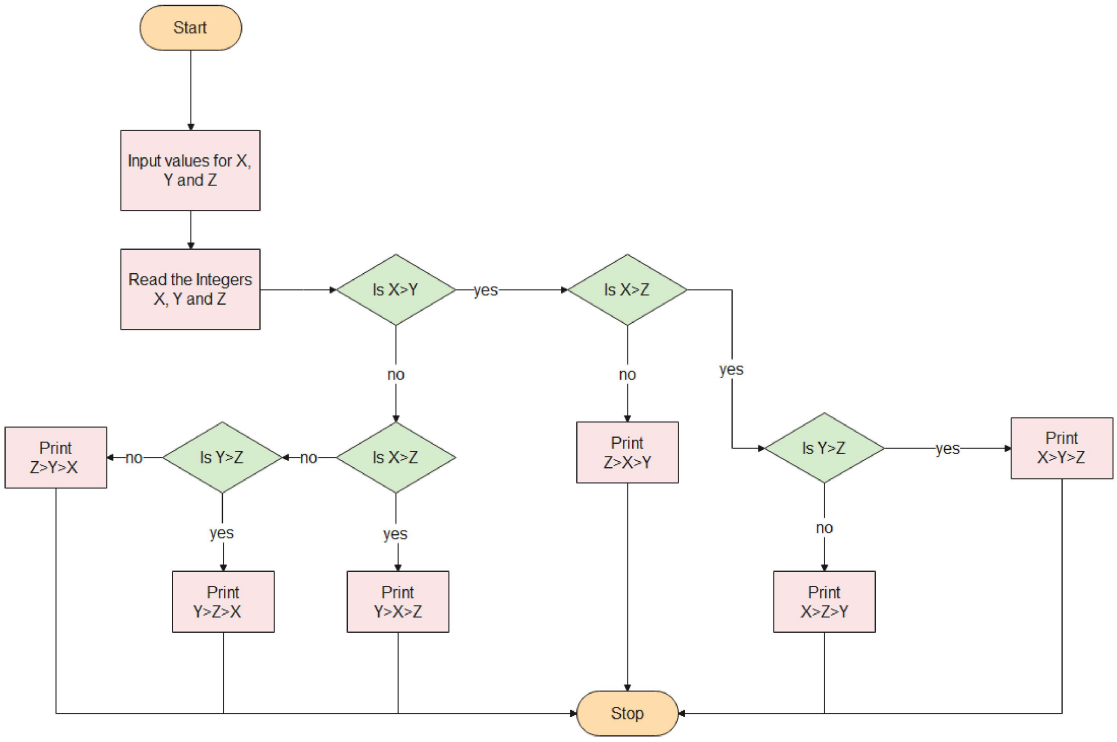
...

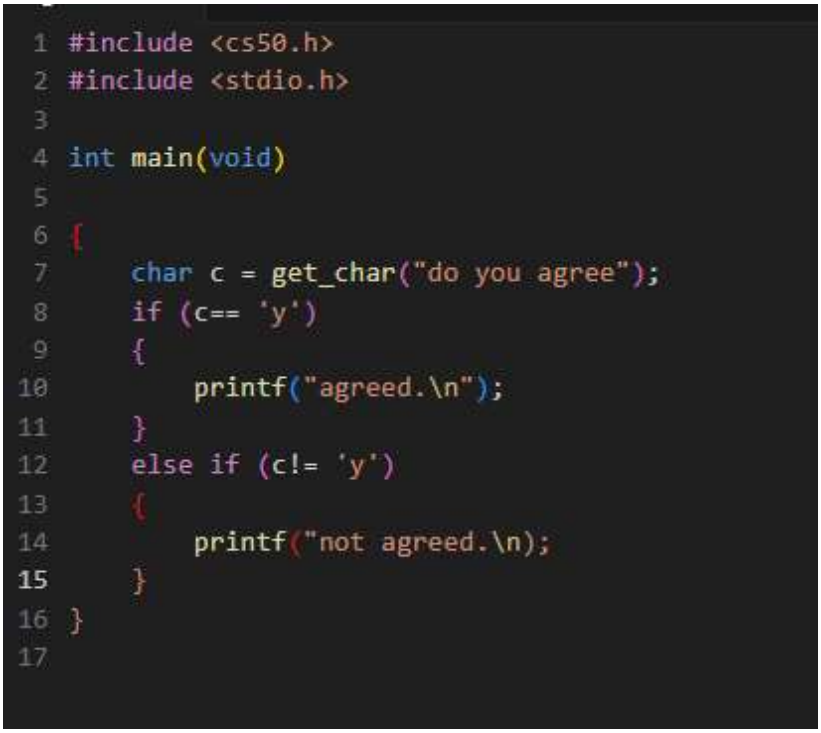



```
bool
char
double
float
int
long
string
...
```



Arrange the numbers X, Y, Z in descending order





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

int main(void)

{
    char c = get_char("do you agree");
    if (c== 'y')
    {
        printf("agreed.\n");
    }
    else if (c!= 'y')
    {
        printf("not agreed.\n);
    }
}

```

agree.c × agree design50 style50

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

int main(void)

{
    char c = get_char("do you agree?");
    if (c == 'y')
    {
        printf("Agreed.\n");
    }
    else if (c == 'n')
    {
        printf(" Not agreed.\n");
    }
}

```

PROBLEMS OUTPUT TERMINAL PORTS 2

```
$ make agree
$ ./agree
do you agreey
Agreed.
$
```

agree.c × agree design50 style50

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

int main(void)

{
    char c = get_char("do you agree?");
    if (c == 'y')
    {
        printf("Agreed.\n");
    }
    else if (c == 'n')
    {
        printf(" Not agreed.\n");
    }
}

```

PROBLEMS OUTPUT TERMINAL PORTS 2

```
$ make agree
$ ./agree
do you agreey
Agreed.
$
$ ./agree
do you agreen
 Not agreed.
$
```

TERMINAL

```
$ make calculator
$ ./calculator
Here's $1. Double it and give it to the next person? y
Here's $2. Double it and give it to the next person? y
Here's $4. Double it and give it to the next person? y
Here's $8. Double it and give it to the next person? y
Here's $16. Double it and give it to the next person? y
Here's $32. Double it and give it to the next person? y
Here's $64. Double it and give it to the next person? y
Here's $128. Double it and give it to the next person? y
Here's $256. Double it and give it to the next person? y
Here's $512. Double it and give it to the next person? y
Here's $1024. Double it and give it to the next person? y
Here's $2048. Double it and give it to the next person? y
Here's $4096. Double it and give it to the next person? y
Here's $8192. Double it and give it to the next person? y
Here's $16384. Double it and give it to the next person? y
Here's $32768. Double it and give it to the next person? y
Here's $65536. Double it and give it to the next person? y
Here's $131072. Double it and give it to the next person? y
Here's $262144. Double it and give it to the next person?
```

integer overflow

```
3
4  int main(void)
5  {
6      long dollars = 1;
7      while (true)
8      {
9          char c = get_char("Here's $%li. Double it and give it to
10         if (c == 'y')
11         {
12             dollars *= 2;
13         }
14         else
15         {
16             break;
17         }
18     }
19     printf("Here's $%li.\n", dollars);
20 }
21
```

Variables

```
int calls = 4;
```

type name | value

assignment operator

calls

4

"Create an **integer** named **calls** that **gets** the **value 4**."

Getting input

```
int calls = get_int("Calls:
```

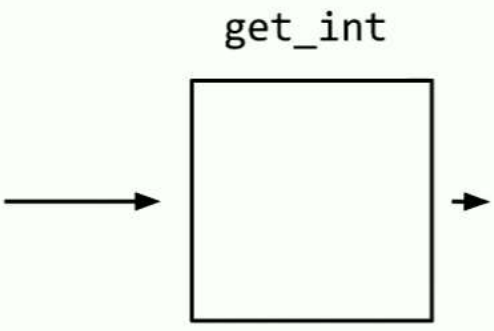
type name |

assignment operator

function

Functions

```
int calls = get_int("Calls: ");
```



Storing return values

```
int calls = 4;
```

type name | value

assignment operator

calls

4

"Create an **integer** named **calls** that **gets** the **value 4**."

Printing values

```
int calls = 4;
printf("calls equals %i", calls);
```

Printing values

```
int calls = 4;
printf("calls equals %i", calls);
```

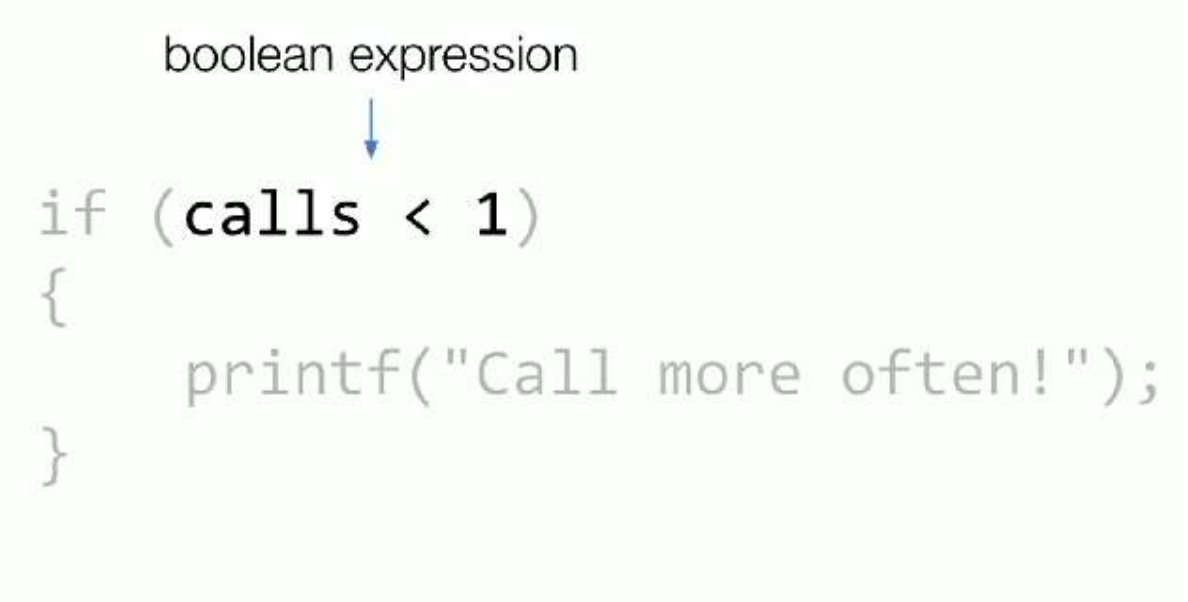
placeholder value

Types and format codes

Numbers	Text	True/False
int (%i)	char (%c)	bool (%i)
float (%f)	string (%s)	

```
// src/index.js
// jsut posthog code for my knowledge

import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

import { PostHogProvider} from 'posthog-js/react'

const options = {
api_host: process.env.REACT_APP_PUBLIC_POSTHOG_HOST,
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
<React.StrictMode>
<PostHogProvider
```

```
apiKey={process.env.REACT_APP_PUBLIC_POSTHOG_KEY}
options={options}
>
<App />
</PostHogProvider>
</React.StrictMode>
);
```



initialization

```
int i = 0;
while (i < 2)
{
    printf("%i\n", i);
    i = i + 1;
}
```

boolean expression

```
int i = 0;
while (i < 2)
{
    printf("%i\n", i);
    i = i + 1;
}
```

```c
int i = 0;
while (i < 2)
{
    printf("%i\n", i);
    i = i + 1;
}
```

increment

boolean expression

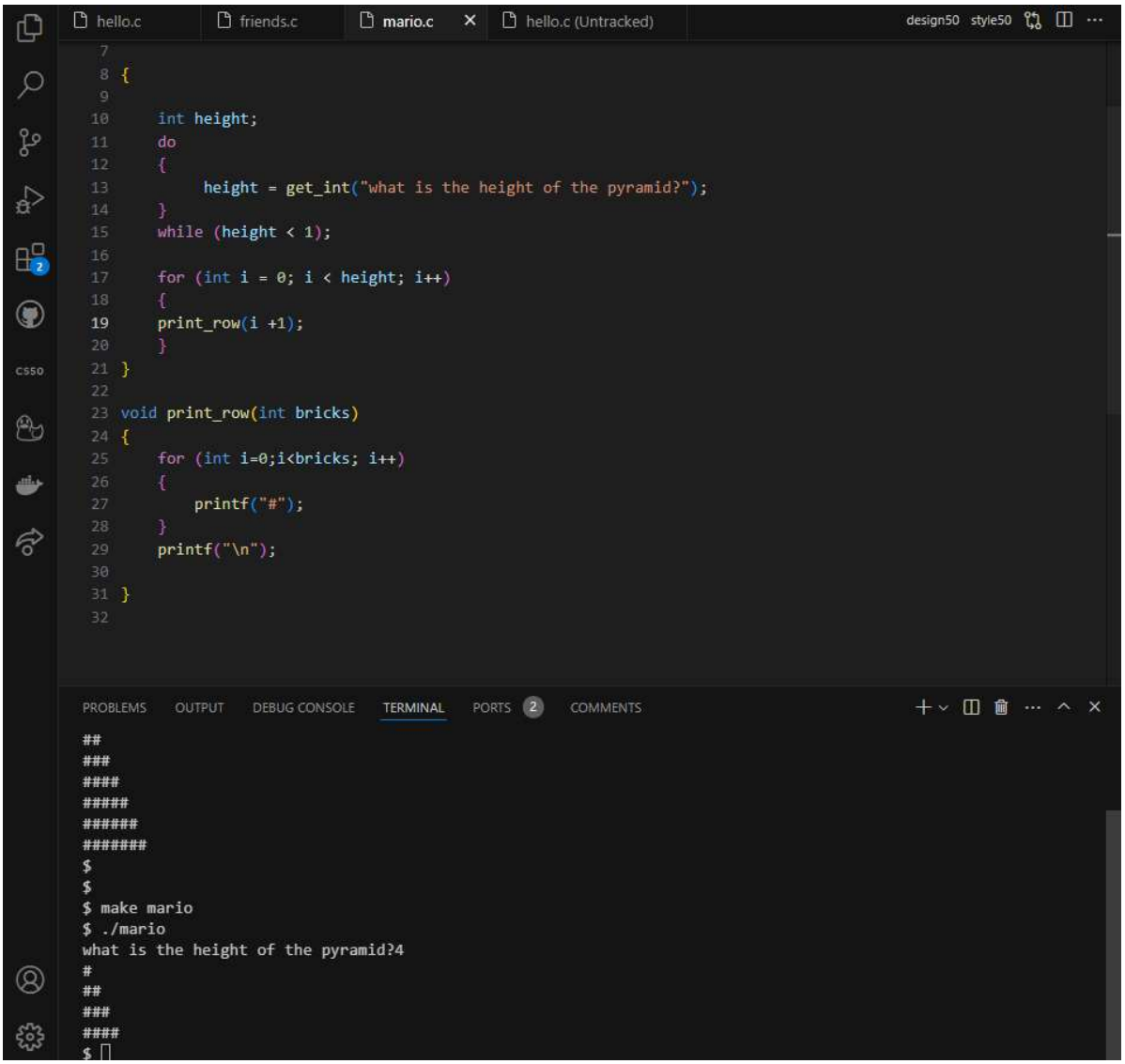
```c
for (int i = 0; i < 2; i++)
{
    printf("%i\n", i);
}
```

Let's start with a left-aligned pyramid first!

```
#
##
###
####
#####
######
```

return type

```
void print_row(int bricks)
{
    # Print row of bricks
}
```

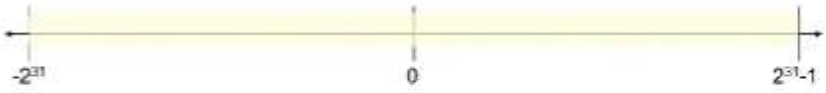


Data Types

Data Types and Variables

- `int`
  - The `int` data type is used for variables that will store integers.
  - Integers always take up 4 bytes of memory (32 bits). This means the range of values they can store is necessarily limited to 32 bits worth of information.

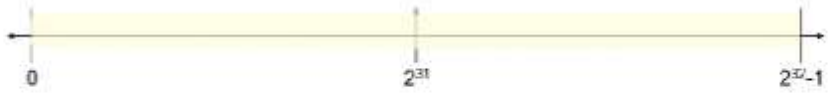
Integer Range



Data Types and Variables

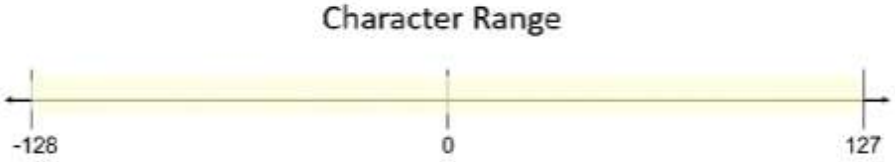
- `unsigned int`
  - `unsigned` is a *qualifier* that can be applied to certain types (including `int`), which effectively doubles the positive range of variables of that type, at the cost of disallowing any negative values.
  - You'll occasionally have use for `unsigned` variables in CS50.

Unsigned Integer Range



Data Types and Variables

- `char`
  - The `char` data type is used for variables that will store single characters.
  - Characters always take up 1 byte of memory (8 bits). This means the range of values they can store is necessarily limited to 8 bits worth of information.
  - Thanks to ASCII, we've developed a mapping of characters like A, B, C, etc... to numeric values in the positive side of this range.



Data Types and Variables

- `float`
  - The `float` data type is used for variables that will store floating-point values, also known as *real numbers*.
  - Floating points values always take up 4 bytes of memory (32 bits).
  - It's a little complicated to describe the range of a `float`, but suffice it to say with 32 bits of precision, some of which might be used for an integer part, we are limited in how *precise* we can be.

Data Types and Variables

- `double`
  - The `double` data type is used for variables that will store floating-point values, also known as *real numbers*.
  - The difference is that doubles are *double precision*. They always take up 8 bytes of memory (64 bits).
  - With an additional 32 bits of precision relative to a `float`, `double`s allow us to be specify much more precise real numbers.

Data Types and Variables

- `void`
  - Is a type, but not a *data type*.
  - Functions can have a `void` return type, which just means they don't return a value.
  - The parameter list of a function can also be `void`. It simply means the function takes no parameters.
  - For now, think of `void` more as a placeholder for "nothing". It's more complex than that, but this should suffice for the better part of the course.

Data Types and Variables

- `bool`
  - The `bool` data type is used for variables that will store a Boolean value. More precisely, they are capable only of storing one of two values: `true` and `false`.
  - Be sure to `#include <cs50.h>` atop your programs if you wish to use the `bool` type.

Data Types and Variables

- `string`
  - The `string` data type is used for variables that will store a series of characters, which programmers typically call a *string*.
  - Strings include things such as words, sentences, paragraphs, and the like.
  - Be sure to `#include <cs50.h>` atop your programs if you wish to use the `string` type.

Data Types and Variables

- Later in the course we'll also encounter structures (`structs`) and defined types (`typedefs`) that afford great flexibility in creating data types you need for your programs.
- Now, let's discuss how to create, manipulate, and otherwise work with variables using these data types.

Data Types and Variables

- Creating a variable
  - To bring a variable into existence, you need simply specify the data type of the variable and give it a name.

```
int number;
char letter;
```

  - If you wish to create multiple variables of the same type, you specify the type name *once*, and then list as many variables of that type as you want.

```
int height, width;
float sqrt2, sqrt3, pi;
```

  - In general, it's good practice to only *declare* variables when you need them.

variables: Data type & Name

int, bool get string row, column

Data Types and Variables

- Using a variable
  - After a variable has been *declared*, it's no longer necessary to specify that variable's type. (In fact, doing so has some unintended consequences!)

```
int number;          // declaration
number = 17;         // assignment
char letter;         // declaration
letter = 'H';        // assignment
```

  - If you are simultaneously declaring and setting the value of a variable (sometimes called *initializing*), you can consolidate this to one step.

```
int number = 17;    // initialization
char letter = 'H'; // initialization
```

```
int number; // declaration
number = 17; // assignment

char letter; // declaration
letter = 'H'; // assignment

ont number = 17; // initialization
```

Operators

Arithmetic Operators

- In C we can add (+), subtract (-), multiply (*) and divide (/) numbers, as expected.

```
int x = y + 1;
x = x * 5;
```

- We also have the modulus operator, (%) which gives us the remainder when the number on the left of the operator is divided by the number on the right.

```
int m = 13 % 4; // m is now 1
```

Arithmetic Operators

- C also provides a shorthand way to apply an arithmetic operator to a single variable.

```
x = x * 5;
x *= 5;
```

- This trick works with all five basic arithmetic operators. C provides a further shorthand for incrementing or decrementing a variable by 1:

```
x++;
x--;
```

Boolean Expressions

- Boolean expressions are used in C for comparing values.
- All Boolean expressions in C evaluate to one of two possible values – `true` or `false`.
- We can use the result of evaluating a Boolean expression in other programming constructs such as deciding which branch in a *conditional* to take, or determining whether a *loop* should continue to run.

conditional to take, Loop!

Boolean Expressions

- Logical operators
  - Logical AND (&&) is true if and only if both operands are true, otherwise false.

x	y	(x && y)
true	true	true
true	false	false
false	true	false
false	false	false

Boolean Expressions

- Logical operators
  - Logical OR (||) is true if and only if at least one operand is true, otherwise false.

x	y	(x || y)
true	true	true
true	false	true
false	true	true
false	false	false

Boolean Expressions

- Logical operators
  - Logical NOT (!) inverts the value of its operand.

x	!x
true	false
false	true

Boolean Expressions

- Relational operators
  - These behave as you would expect them to, and appear syntactically similar to how you may recall them from elementary arithmetic.
    - Less than (`x < y`)
    - Less than or equal to (`x <= y`)
    - Greater than (`x > y`)
    - Greater than or equal to (`x >= y`)

Boolean Expressions

- Relational operators
  - C also can test two variables for equality and inequality.
    - Equality (`x == y`)
    - Inequality (`x != y`)
  - Be careful! It's a common mistake to use the assignment operator (=) when you intend to use the equality operator (==).

Conditional Statements

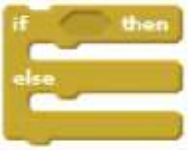
Conditionals

```
if (boolean-expression)
{

}
```

- If the `boolean-expression` evaluates to `true`, all lines of code between the curly braces will execute in order from top-to-bottom.
- If the `boolean-expression` evaluates to `false`, those lines of code will not execute.

Conditionals

```
if (boolean-expression)
{

}
else
{

}
```



- If the `boolean-expression` evaluates to `true`, all lines of code between the first set of curly braces will execute in order from top-to-bottom.
- If the `boolean-expression` evaluates to `false`, all lines of code between the second set of curly braces will execute in order from top-to-bottom.

Conditionals

```
if (boolean-expr1)
{
   // first branch
}
else if (boolean-expr2)
{
   // second branch
}
else if (boolean-expr3)
{
   // third branch
}
else
{
   // fourth branch
}
```

- In C, it is possible to create an `if-else if-else` chain.
  - In Scratch, this required nesting blocks.
- As you would expect, each branch is mutually exclusive.

Conditionals

```
int x = GetInt();
switch(x)
{
   case 1:
      printf("One!\n");
      break;
   case 2:
      printf("Two!\n");
      break;
   case 3:
      printf("Three!\n");
      break;
   default:
      printf("Sorry!\n");
}
```

- C's `switch()` statement is a conditional statement that permits enumeration of discrete cases, instead of relying on Boolean expressions.
- It's important to `break;` between each case, or you will "fall through" each case (unless that is desired behavior).

Conditionals

```
int x;
if (expr)
{
   x = 5;
}
else
{
   x = 6;
}
```

```
int x = (expr) ? 5 : 6;
```

- These two snippets of code act identically.
- The ternary operator (?:) is mostly a cute trick, but is useful for writing trivially short conditional branches. Be familiar with it, but know that you won't need to write it if you don't want to.

Loops

Loops

- Loops allow your programs to execute lines of code repeatedly, saving you from needing to copy and paste or otherwise repeat lines of code.

- C provides a few different ways to implement loops in your programs, some of which likely look familiar from Scratch.

Loops

```
while (true)
{

}
```

- This is what we call an *infinite loop*. The lines of code between the curly braces will execute repeatedly from top to bottom, until and unless we break out of it (as with a `break;` statement) or otherwise kill our program.

Loops

```
while (boolean-expr)
{

}
```

- If the `boolean-expr` evaluates to `true`, all lines of code between the curly braces will execute repeatedly, in order from top-to-bottom, until `boolean-expr` evaluates to `false`.
- Somewhat confusingly, the behavior of the Scratch block is reversed, but it is the closest analog.

Loops

```
do
{

}
while (boolean-expr);
```

- This loop will execute all lines of code between the curly braces once, and then, if the `boolean-expr` evaluates to `true`, will go back and repeat that process until `boolean-expr` evaluates to `false`.

Loops

```
for (int i = 0; i < 10; i++)
{

}
```

- Syntactically unattractive, but `for` loops are used to repeat the body of a loop a specified number of times, in this example 10.
- The process undertaken in a `for` loop is:
  - The counter variable(s) (here, `i`) is set
  - The Boolean expression is checked.
    - If it evaluates to `true`, the body of the loop executes.
    - If it evaluates to `false`, the body of the loop does not execute.
  - The counter variable is incremented, and then the Boolean expression is checked again, etc.

Loops

```
for (start; expr; increment)
{

}
```

- Syntactically unattractive, but `for` loops are used to repeat the body of a loop a specified number of times, in this example 10.
- The process undertaken in a `for` loop is:
  - The statement(s) in `start` are executed
  - The `expr` is checked.
    - If it evaluates to `true`, the body of the loop executes.
    - If it evaluates to `false`, the body of the loop does not execute.
  - The statement(s) in `increment` are executed, and then the `expr` is checked again, etc.

Loops

`while`

- Use when you want a loop to repeat an unknown number of times, and possibly not at all.

`do-while`

- Use when you want a loop to repeat an unknown number of times, but at least once.

`for`

- Use when you want a loop to repeat a discrete number of times, though you may not know the number at the moment the program is compiled.

Command Line

Using the Linux Command Line

- The CS50 IDE is a cloud-based machine running *Ubuntu*, one of the many flavors of the *Linux* OS.
- Many modern Linux distributions have graphical user interfaces (GUI) to allow easy mouse-based navigation.
- Still, as a programmer you'll likely be using your *terminal window* frequently, and you can do many of the same tasks with keyboard commands.

ls - list,

ls-- cd---ls

Using the Linux Command Line

`mkdir <directory>`

- Short for "make directory", this command will create a new subdirectory called `<directory>` located in the current directory.

Using the Linux Command Line

`cd <directory>`

- Short for "change directory", this command change your current directory to `<directory>`, which you specify, in your workspace or on your operating system.
- The shorthand name for the current directory is `.`
- The shorthand name for the parent directory of the current directory is `..`
- If ever curious about the name of the current directory, though the terminal prompt will often tell you, you can type pwd (present working directory).

mkdir ----> directory ---> Folder!

cp - copy
cp -r (-r -> everything)

Using the Linux Command Line

`rm <file>`

- Short for "remove", this command will delete `<file>` after it asks you to confirm (y/n) you want to delete it.
- You can skip the confirmation by typing:
  `rm -f <file>`
  But use at your own peril! There's no undo.
- To delete entire directories you need to use the `-r` flag, just as was the case with `cp`.
  `rm -r <directory>`
- You can also combine the `-r` and `-f` flags into `-rf`. Again, careful! There's no undo!

Using the Linux Command Line

`mv <source> <destination>`

- Short for "move", this command will allow you to effectively rename a file, moving it from `<source>` to `<destination>`.

Using the Linux Command Line

- To be sure, there are many more basic command line utilities at your disposal, and we'll discuss many of them in the future in CS50.
- If you wish to explore other interesting ones before we see them in the class, read up on:

chmod	ln	touch
rmdir	man	diff
sudo	clear	telnet

Magic Numbers

Magic Numbers

- Some of the programs we've written in CS50 have some weird numbers thrown in there.
  - The height of Mario's pyramid is capped at 23, for example.
- What do those numbers mean? If someone looks at your program, is the meaning of 23 immediately obvious?
- Directly writing constants into our code is sometimes referred to as using **magic numbers**.

Magic Numbers

```
card deal_cards(deck name)
{
    for (int i = 0; i < 52; i++)
    {
        // deal the card
    }
}
```

- We've got a magic number in here. Do you see what it is?
  - More importantly, do you see a potential problem here? Particularly if this function is just one of many in a suite of programs that manipulate decks of cards.

Magic Numbers

```
card deal_cards(deck name)
{
    int deck_size = 52;
    for (int i = 0; i < deck_size; i++)
    {
        // deal the card
    }
}
```

- This fixes one problem, but introduces another.
  - Even if globally declared, what if some other function in our suite inadvertently manipulates `deck_size`. Could spell trouble.

Magic Numbers

- C provides a **preprocessor directive** (also called a **macro**) for creating symbolic constants.

```
#define NAME REPLACEMENT
```

- At the time your program is compiled, `#define` goes through your code and replaces `NAME` with `REPLACEMENT`.
  - If `#include` is similar to copy/paste, then `#define` is analogous to find/replace.

Magic Numbers

- C provides a **preprocessor directive** (also called a **macro**) for creating symbolic constants.

```
#define PI 3.14159265
```

- At the time your program is compiled, `#define` goes through your code and replaces `PI` with `3.14159265`.
  - If `#include` is similar to copy/paste, then `#define` is analogous to find/replace.

Magic Numbers

- C provides a **preprocessor directive** (also called a **macro**) for creating symbolic constants.

```
#define COURSE "CS50"
```

- At the time your program is compiled, `#define` goes through your code and replaces `COURSE` with `"CS50"`.
  - If `#include` is similar to copy/paste, then `#define` is analogous to find/replace.

Magic Numbers

```
#define DECKSIZE 52

card deal_cards(deck name)
{
    for (int i = 0; i < DECKSIZE; i++)
    {
        // deal the card
    }
}
```