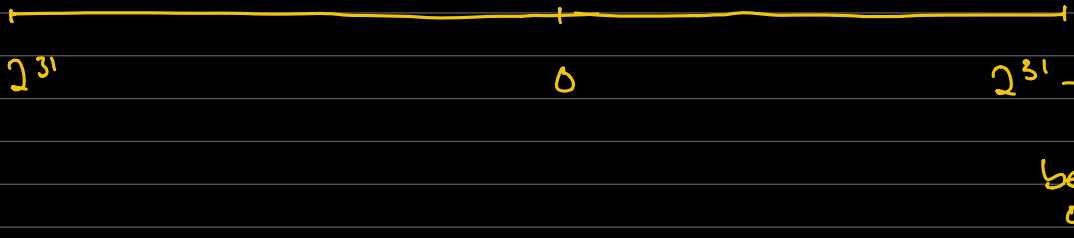
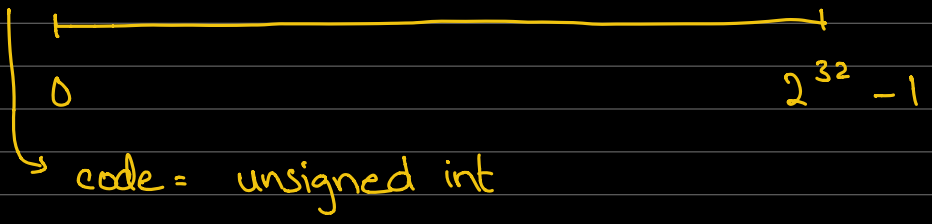
DATA TYPES

Integers
- A data type that stores positive or negative whole numbers
- Take up 4 bytes OR 32 bits of memory
- Since two's complement is being used, the leftmost digit is reserved for the negative value.
  - ↳ So only 31 bits are used to actually represent values
- The range of values that an int can represent:



- About -2 billion to 2 billion
- Unsigned int → can't show negative values.



Chars
- Vars that store single characters
- All chars take up one byte / 8 bits
- Follow ASCII code

Floating point numbers
- Decimal values
- Take up 4 bytes / 32 bits

Double
- Decimal values
- Take up 8 bytes / 64 bits

void → Type, NOT data type
- "Nothing"

Additional CS50 types from cs50.h

bool (boolean)
- Can store "true" or "false"

string
- A string of characters of variable length

DECLARING VARS.

```
int number;
char letter;

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

Note: It's good practice / design to only declare variables when required (scope issues)

Initialization vs. Declaration vs. Assignment

Initialization → Declaring and assigning a variable simultaneously.

Ex. `int number = 5;`

Declaration → Only "reserving space" for a variable, but not assigning anything to it

Ex. `int number;`

Assignment → Assigning a value to a variable that's already been declared

Ex. `number = 5;`

OPERATORS

Arithmetic Operators

Add → +
Subtract → -
Multiply → *
Divide → /

Modulus → % → returns the value of the remainder

upon division

Ex. 10 % 3 returns 1

Arithmetic Shorthand

num = num * 5; — is the same as → num *= 5;

Can be used for all the arithmetic operators written above

num++; — is the same as → num = num + 1;
num--; — is the same as → num = num - 1;

Boolean Expressions → returns only two possible values
- Logical Operators
- Relational Operators

↳ "True" or "False"

Note: In C, all non-zero values are true

Logical Operators

1. AND (&&)
   - Only true if both conditions are true
2. OR (||)
   - only false if both conditions are false
3. NOT (!)
   - Inverts the value of a variable/ expression / operand

Relational Operators

1. less than (<)
2. Greater than (>)
3. less than equal to (<=)
4. Greater than equal to (>=)
5. Equal to (==)
6. Not equal to (!=)

CONDITIONAL STATEMENTS

IF :

1. IF statement

```
if (bool)
{
    [code]
}
```

2. IF ELSE statement

```
if (bool)
{
  [main code]
}
else
{
  [alt code]
}
```

3. IF - ELSE IF - ELSE statement

```
if (bool)
{
  [main code]
}
else if (otherBool)
{
  [other Main Code]
}
else
{
    [alt code]
}
```

Note: All branches are mutually exclusive in one if statement.
To have not-mutually-exclusive branches, use multiple if statements

SWITCH

```
switch (x)
{
  case possibleValue:
      code;
      break;
  case anotherPossibleValue:
      code;
      break;
  case anotherPositiveValue:
      code;
      break;
  default finalResort:
      alt code;
}
```

→ This is important because if not included, the first true case from the top will be executed and so will all the following cases.

↳ ie. you will "fall through" the rest of the switch statement

break; is the C equivalent of sys.exit() in Python.

TERNARY OPERATOR (? :)

- "Shorthand" (but not exactly) for if-else
- Really concise way of writing a simple if-else statement

```
int num1 = (booleanCondition) ? 5 : 6
```

↳ this use of the ternary operator is the same as the following code:

```
int num1;
if (booleanCondition)
{
    x = 5;
}
else
{
    x = 6;
}
```

LOOPS

Infinite loop

```
while (true)
{
    infiniteCode; → Infinite no. of iterations
}
```

Pre-condition loop

```
while (BoolExpression)
{
    code; → Unknown no. of iterations, possibly 0 iterations
}
```

Post-condition loop

```
do
{
    code;
}
while (BoolExpression);
```

↳ Unknown no. of iterations, but at least 1 iteration

Count-controlled loop

→ 10 iterations

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

↳ limited and discrete number of iterations

LINUX COMMAND LINE (Unix-based commands)

ls (list)
- list immediate contents of the current directory

cd (change directory)

cd [name] → where [name] must be an immediate child directory of the current directory

cd . → change directory to the current directory

↳ I know this command is kinda stupid, but it's important to know that "." is shorthand for "current directory"

cd .. → change directory to the parent directory

↳ shorthand for "parent directory"

pwd → returns the name of current directory, or "present working directory"

cd ../.. → move up two levels

cd → go back to home directory

mkdir (make directory)

mkdir [name] → Create a directory called [name] in the current directory

mkdir path/[name] → Create a directory called [name] where the specified absolute path leads to

↑

mkdir ./path/[name] → Same thing as this but the path is now relative (to the current directory) instead of being absolute

cp → copy

cp <source> <destination>

cp hello.txt hi.txt

↳ Copies the contents of hello.txt into a new file called hi.txt

cp -r home/work/ user/

↳ Copies the work directory and its contents (because -r → recursive) into a new directory called user in the home directory

rm → remove rm <fileName>

rm [file] → Deletes file, asks for confirmation
↳ stands for "force"
rm -f [file] → Deletes file, does not ask for confirmation

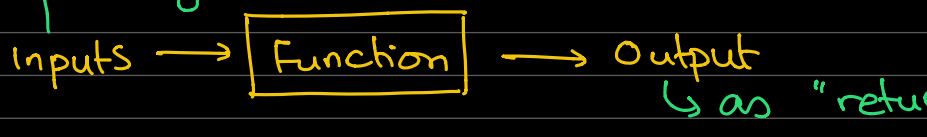
rm -r [dir] → Deletes directory + its contents, asks for confirmation
↳ "recursive"

rm -rf [dir] → Deletes directory + it's contents, does not ask for confirmation

mv → move

mv old.c new.c → essentially renaming
↳ Move contents of old.c to new.c and get rid of old.c

FUNCTIONS aka. procedures or methods or subroutines



Why are functions used?
Help in organizing the code by breaking down the problem at hand into smaller, more manageable tasks.

Ease of implementation – It's generally easier to design, implement, and debug smaller chunks of code that collectively solve the problem

at hand.

Functions can be reused and recycled in numerous programs

Using functions:

1. Declaration
   Tells compilers that this function that's later called in the program, although defined after the main function, exists

   Function declarations are conventionally written before the main function.

return-type functionName (arg-type arg-name, arg2-type arg2-name)

↳ function declaration format