functions

- known as procedures/subroutines/methods in other lang
- returns a value
  - ↳ return 0; or return void when it return nothing
- block of code that performs a certain task
  - ↳ performs a logically consistent task
- only defined once
- must declare them before using them
  - ↳ main would be "last"
- You can call them in other parts of the code
- main ()
  - ↳ special function
  - ↳ runs when program starts
  - ↳ other functions are subordinate

Things a function should do

define abstractions that are consistent
hide implementation
simplify code
- used for repeated sequence of code
they should never be arbitary sequences of statements

A function:

return_type function_name(parameters) { → head
// declarations
// function block / body

return_type
- defines the type of function's return value
- return type may be void or anything else
↳ cannot return an array

function name
- the name...

Parameters
- list of declarations seperated by a comma
- if not parameters use (void)

Declarations
- declare / init variables inside a function to make them local

Assignment Statements
- sets & restores value of variables, i/o, etc

Return values
- return a value, it can be void
- can return a char, int, float, etc
↳ also a pointer
- You can't return an array
- You can't name two functions the same

snake case for naming
↳ my_function_name

Parameters
- also known as arguments
- substitutions in algebraic functions is known as call-by-name
  ↳ instead macros with C preprocessor are used
- US mostly used: call by value- call-by reference
- C uses call-by-value, but not for arrays
formal parameters
  ↳ name of the parameter used inside of the function body
Actual parameters
  ↳ name of the value that is passed to the function
  - can be copied to the formal parameter
    ↳ passing a pointer using call-by-value
call-by-value: copy of the actual parameter is placed in the formal parameter

copy-in_copy2 = not only is it copied, the value is copied back to the actual parameter
  ↳ DC doesn't support it

Call-by-value

- all functions use this
- arguments are copied
  - ↳ no effect on the parameters inside the function
- copies the value of the actual parameters into a new set of variables (formal parameters) which are pushed into the call stack

Call-by-Refrence

- refrences of the arguments passed into a function are copied ~~mean~~. any changes made to the parameters inside the function has an effect on the actual values
  - ↳ the og variables are passed

although c doesn't use this, you can recreate it with pointer

- ↳ addresses instead of values are passed as arguments

function prototypes

cannot be called unless they have been declared and defined prior to function call

been prototyped at the beginning of the progr

- ↳ must be declared either at the beginning of the program or in a header file

#include

- preprocessor directive
- transfers program
- macros operate through text replacement
- used to include functions defined in other libraries

Define

- defines a macro
- goes in and replaces what you declared with the word
  - #define Pi 3.14159265
    - every time it sees Pi, it replaces it with 3.14159265

conditional directives

- use conditional statements to include code selectively

ifndef - execute statements when macro is undefined

ifdef - executes only when macro is defined

header files

- only contain things shared between files
  - function declaration
  - macro definitions
  - data structure & enumeration
  - global variables

-must be within a header guard, with #ifndef

Extern

- extends the visibility of variables & functions that can be called by any program file, if the declaration is known
- functions are implicitly prepended by extern
- used for global variables

Static

can be declared inside & outside
inside if the value of the variable needs to persist across function calls.
outside if the value of the variable needs to accessed my mult function

Recursion

- function may call itself

```
void function_name(){
    function_name();
}
```

- must always define an exit condition