

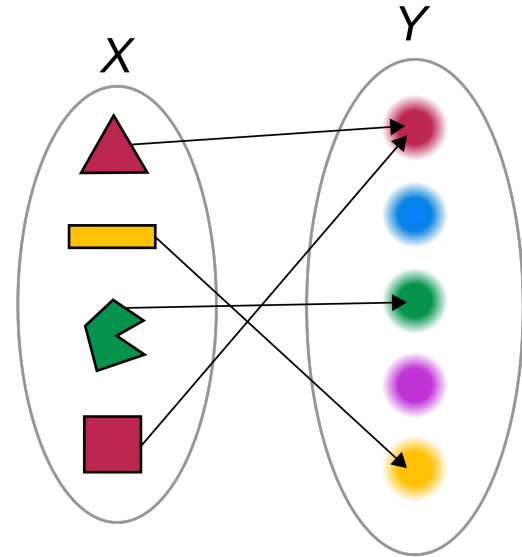


# Functions



# Functions

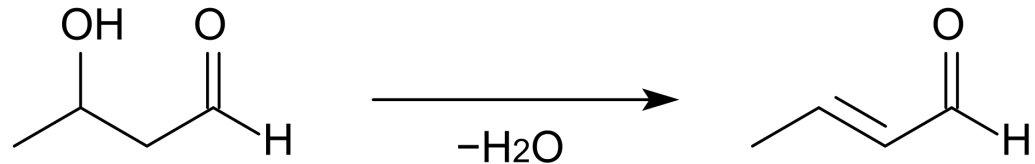
- Allow a program to be broken down into component parts
- Algorithms are not commonly treated as single entities in computer science - they are usually broken down into sub-algorithms



# Functions

- **Functional Decomposition**

- Consists of breaking down large-scale problems into manageable “bits”, then allowing those “bits” to be solved by individual programmers (or small teams)



# Functional Decomposition

Example: Home design / building



# Functional Decomposition



- This decomposition can work, as long as everyone involved agrees on *interfaces*
  - Interface: standard method of sharing information between two different functions

# Function interfaces: Programming

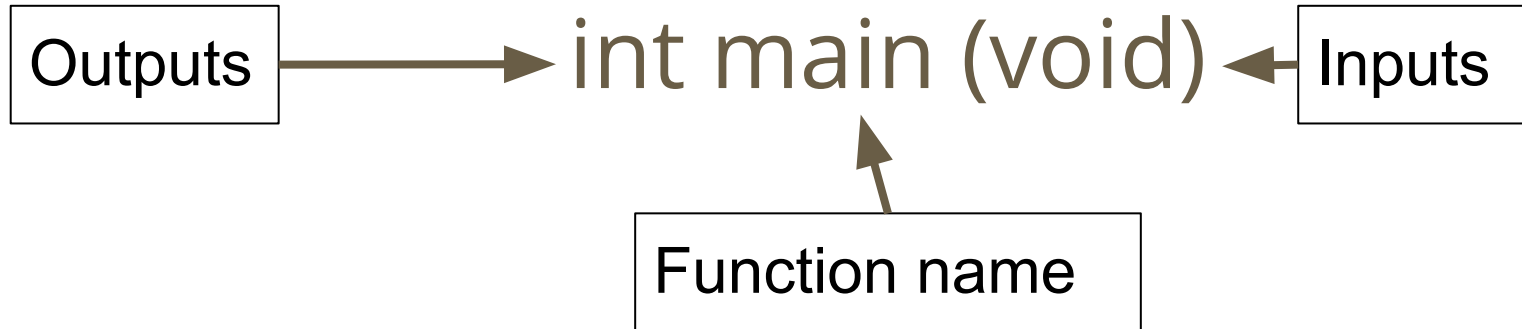


- A function (in programming) has three parts:
  - 1) A name
  - 2) Inputs → also known as *parameters* or *arguments*
  - 3) Outputs → also known as *return values*

In C, a function may have 1 or 0 outputs - NO MORE

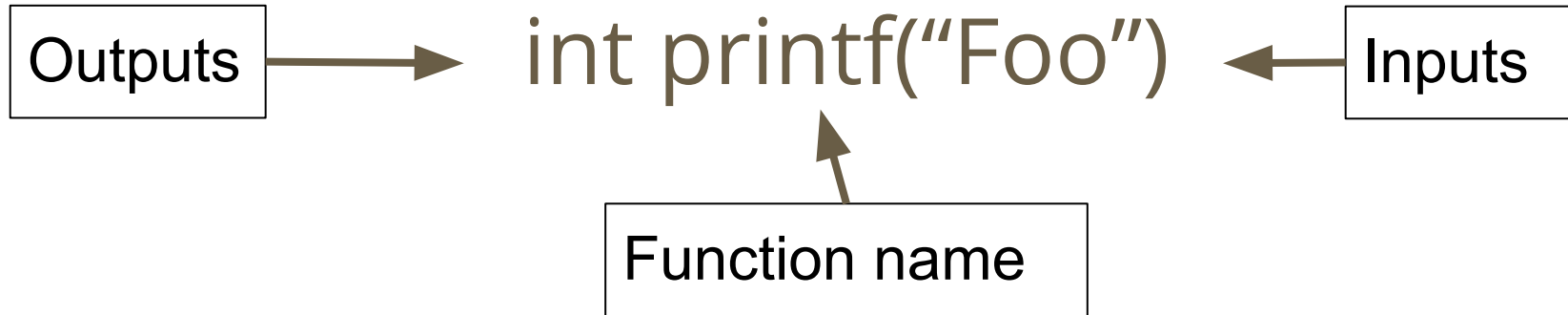
# Functions interfaces

- We've seen this before...



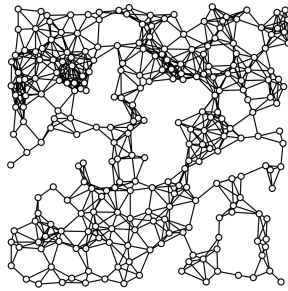
# Functions interfaces

- And again...





# Built-in functions



- C libraries (stdio.h, math.h, stdlib.h, etc...) all have pre-defined functions
  - All have defined names, inputs, and outputs
  - What does scanf() take as input? How about output?
  - What does rand() take as input? What does it output?

# Function names

- Can be anything
- Follow the same rules as variable names
  - No whitespace, etc...



Hello  
my name is

# Function inputs

- Must be specified: this allows the interface to be consistent across code
- If there are NO inputs...
  - Write "void"
- If there ARE inputs...
  - Have to specify type + name



# Function inputs

```
int main(void);
```

```
int rand(void);
```

- Each built-in function has a specific input
  - `main()` / `rand()` have no inputs (therefore *void*)



# Function outputs

- Must be specified: this allows the interface to be consistent across code
- If there are NO outputs
  - Write “void”
- If there IS an output
  - Have to specify type



# Function outputs

```
int main(void);
```

```
int rand(void);
```

- Both main() and rand() return a value → specifically, both return an integer
- Therefore can be assigned to a value...

```
int num1 = rand();
```



# Function Declarations

- In order to be used within a C program, a function must be declared above main()
  - Main() is the only exception to this rule

```
#include <stdio.h>
void do_stuff(void);
int main(void) {
    /*Code goes here*/
    return 0;
}
```

IN CONGRESS, JULY 4, 1776.

The unanimous Declaration of the thirteen united States of America.

# Function Declarations

- The *#include* statement declares pre-built functions
- The *do\_stuff()* function is declared as well (will be using it as an example)

```
#include <stdio.h>
void do_stuff(void);
int main(void) {
    /*Code goes here*/
    return 0;
}
```

IN CONGRESS, JULY 4, 1776.

The unanimous Declaration of the thirteen united States of America.



# Function Declarations

- These declarations are called *function prototypes*
  - Are necessary to using / creating functions in C

```
#include <stdio.h>
void do_stuff(void);
int main(void) {
    /*Code goes here*/
    return 0;
}
```

IN CONGRESS, JULY 4, 1776.

The unanimous Declaration of the thirteen united States of America.

# Function invocation

- The function *invocation* “runs” the function
- Has 3 parts:
  - 1) Function name
  - 2) Parentheses (directly after name)
  - 3) Zero or more arguments (must be comma separated)



# Function invocation

- The function *invocation* “runs” the function

```
void do_stuff(void);  
int main(void) {  
    do_stuff();  
    return 0;  
}
```



# Function invocation

- The function *invocation* “runs” the function

```
#include <stdio.h>
void do_stuff(void);
int main(void) {
    do_stuff();
    printf("Hello World");
    return 0;
}
```



# Function invocation

- Multiple arguments to a function are separated by commas

```
#include <stdio.h>
void do_stuff(void);
int main(void) {
    int num1 = 1;
    printf("Hello World: %d", num1);
    return 0;
}
```



# Function Definition

- 3rd (and most important) part of creating a function
- Definition of a function goes BELOW main()

```
#include <stdio.h>
void do_stuff(void);
int main(void) {
    do_stuff();
    return 0;
}
void do_stuff(void) {
    printf("I'm a function!\n");
}
```



# Function Definition

- Begins with a repetition of the function declaration
  - Has to be the same → otherwise, the compiler will give an error

```
#include <stdio.h>
void do_stuff(void);
int main(void) {
    do_stuff();
    return 0;
}
void do_stuff(void) {
    printf("I'm a function!\n");
}
```



# Function Definition

- Includes curly braces {}
  - Similar to if statements or loops

```
#include <stdio.h>
void do_stuff(void);
int main(void) {
    do_stuff();
    return 0;
}
void do_stuff(void) {
    printf("I'm a function!\n");
}
```





# Function Definition

- All code for the function goes inside the braces
  - Is executed **WHENEVER** the function is invoked

```
#include <stdio.h>
void do_stuff(void);
int main(void) {
    do_stuff();
    return 0;
}
void do_stuff(void) {
    printf("I'm a function!\n");
}
```



# Function task

- Write the “Hello World” program
  - The line `printf(“Hello World\n”);` has to be in its own function
- *Hint: Use the function declaration **`void hello(void);`***

