# Week 3 Part 1

Kyle Dewey

# Overview

- Odds & Ends
- Constants
- Errors
- Functions
- Expressions versus statements

# Pre-Lab

# Underflow & Overflow

# Note

# Constants

# Constants

- Values which never change
- Specific values are constants
  - `55`
  - `27.2`
  - `'a'`
  - `"foobar"`

# Constants

- Specifically in the program text
  - Constant in that `52` always holds the same value
    - We cannot redefine `52` to be `53`

# Symbolic Constants

- Usually when programmers say "constant", they mean "symbolic constant"
- Values that never change, but referred to using some symbol
  - i.e. π (pi - `3.14...`)
- Mapping between symbol and value is explicitly defined somewhere

# In C

- Use `#define`
- By convention, constants should be entirely in caps

```
#define PI 3.14
...
int x = PI * 5;
```

# Mutability

- Constants are, well, constant!
- Cannot be changed while code runs

```
#define PI 3.14
...
PI = 4; // not valid C!
```

# What `#define` Does

- Defines a **text substitution** for the provided symbol
- This text is replaced during compilation by the **C preprocessor** (cpp)

# Example #1

Code

```
#define PI 3.14
...
int x = PI * 5;
```

After Preprocessor

```
int x = 3.14 * 5;
```

# Example #2

Code

```
#define PI 3.14
...
PI = 4;
```

After Preprocessor

```
3.14 = 4;
```

# Best Practices

- Generally, all constants should be made symbolic
  - Easier to change if needed later on
  - Gives more semantic meaning (i.e. `PI` is more informative than `3.14...`)
  - Possibly less typing

# Errors

# Errors

- Generally, expected result does not match actual result
- Four kinds of errors are relevant to CS16:
  - Syntax errors
  - Linker errors
  - Runtime errors
  - Logic errors

# Errors

- Four kinds of errors are relevant to CS16:
  - Syntax errors
  - Linker errors
  - Runtime errors
  - Logic errors

# Syntax Error

- A "sentence" was formed that does not exist in the language
- For example, "Be an awesome program" isn't valid C

# Syntax Error

- Easiest to correct
- Compiler will not allow it
- *Usually* it will say where it is exactly

# On Syntax Errors

- ...sometimes the compiler is really bad at figuring out where the error is

```
#include <stdio.h>

int main() {
  printf( "moo" )
  printf( "cow" );
  return 0;
}
```

# Reality

```
#include <stdio.h>

int main() {
   printf( "moo" )
   printf( "cow" );
   return 0;
}
```

- Missing semicolon at line 4

# GCC

```
#include <stdio.h>

int main() {
  printf( "moo" )
  printf( "cow" );
  return 0;
}
```

```
syntax.c: In function ‘main’:
syntax.c:5: error: expected ‘;’ before
‘printf’
```

# Ch

```
#include <stdio.h>

int main() {
   printf( "moo" )
   printf( "cow" );
     return 0;
  }
```

```
ERROR: multiple operands together
ERROR: syntax error before or at line 5
in file syntax.c
  ==>:   printf( "cow" );
  BUG:   printf( "cow" )<== ???
```

# The Point

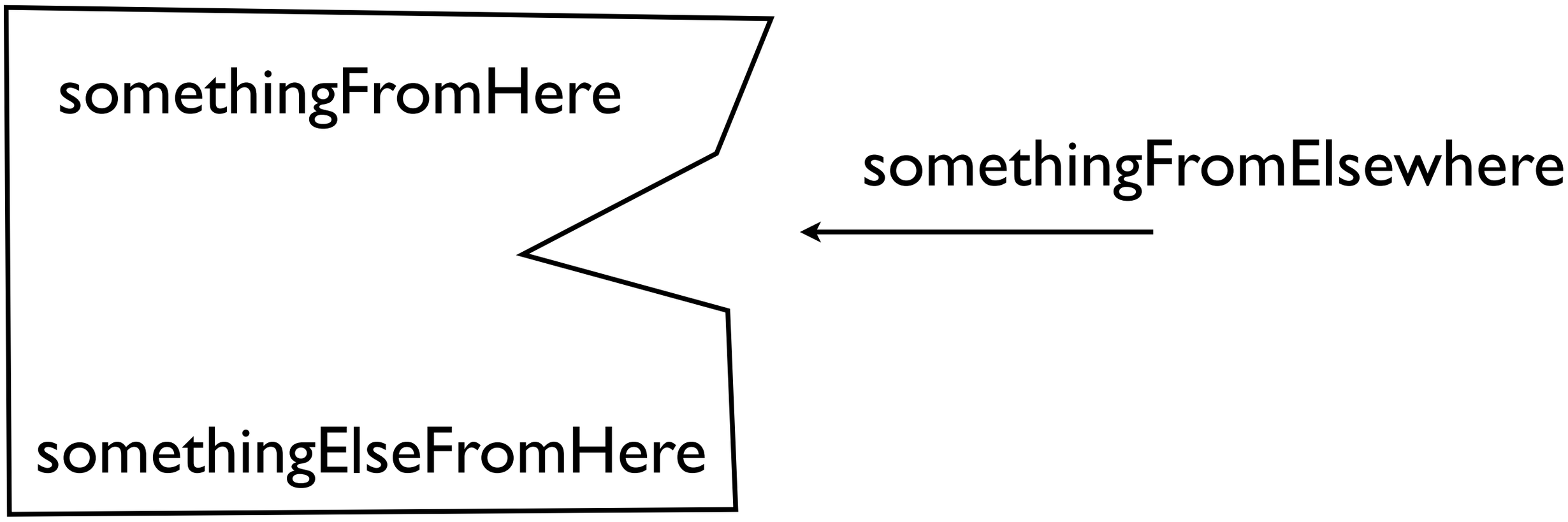
- Compilers are just other programs
- Programs can be wrong
- Programs are not as smart as people

# Errors

- Four kinds of errors are relevant to CS16:
  - Syntax errors
  - Linker errors
  - Runtime errors
  - Logic errors

# Recall Linking

```
1: somethingFromHere();
2: somethingFromElsewhere();
3: somethingElseFromHere();
```



# Recall Linking

somethingFromElsewhere

somethingFromHere

somethingElseFromHere

# Linker Errors

- What if `somethingFromElsewhere` is nowhere to be found?
- Missing a piece
- Cannot make the executable

# Example

```
int something();

int main() {
    something();
    return 0;
}
```

- `int something();` **tells the compiler that** `something` **exists somewhere, but it does not actually give** `something`

# Example

```
int something();

int main() {
   something();
   return 0;
}
```

```
Undefined symbols for architecture
x86_64:
  "_something", referenced from:
      _main in ccM6c8aW.o
ld: symbol(s) not found for
architecture x86_64
```

# Errors

- Four kinds of errors are relevant to CS16:
  - Syntax errors
  - Linker errors
  - Runtime errors
  - Logic errors

# Runtime Errors

- Error that occurs while the code is running
- Compilation and linking must have succeeded to get to this point

# Examples

- Overflow

```
unsigned char x = 255;
x = x + 1;
```

- Underflow

```
unsigned char x =0;
x = x - 1;
```

# Examples

- Divide by zero (especially for integers!)

```
unsigned int x = 5 / 0;
```

- **Wrong** `printf` **placeholder**

```
printf( “%s”, 57 );
```

# Errors

- Four kinds of errors are relevant to CS16:
  - Syntax errors
  - Linker errors
  - Runtime errors
  - Logic errors

# Logic Errors

- It works!
- ...but it doesn't do what you wanted
- Like getting the wrong order at a restaurant

# Examples

- Transcribed an equation incorrectly
- Using the wrong variable
- Lack of understanding of problem
- etc. etc. etc...

# Logic Errors

- By far, the most difficult to debug
- It might be done **almost** correctly
- This is why testing is so important!

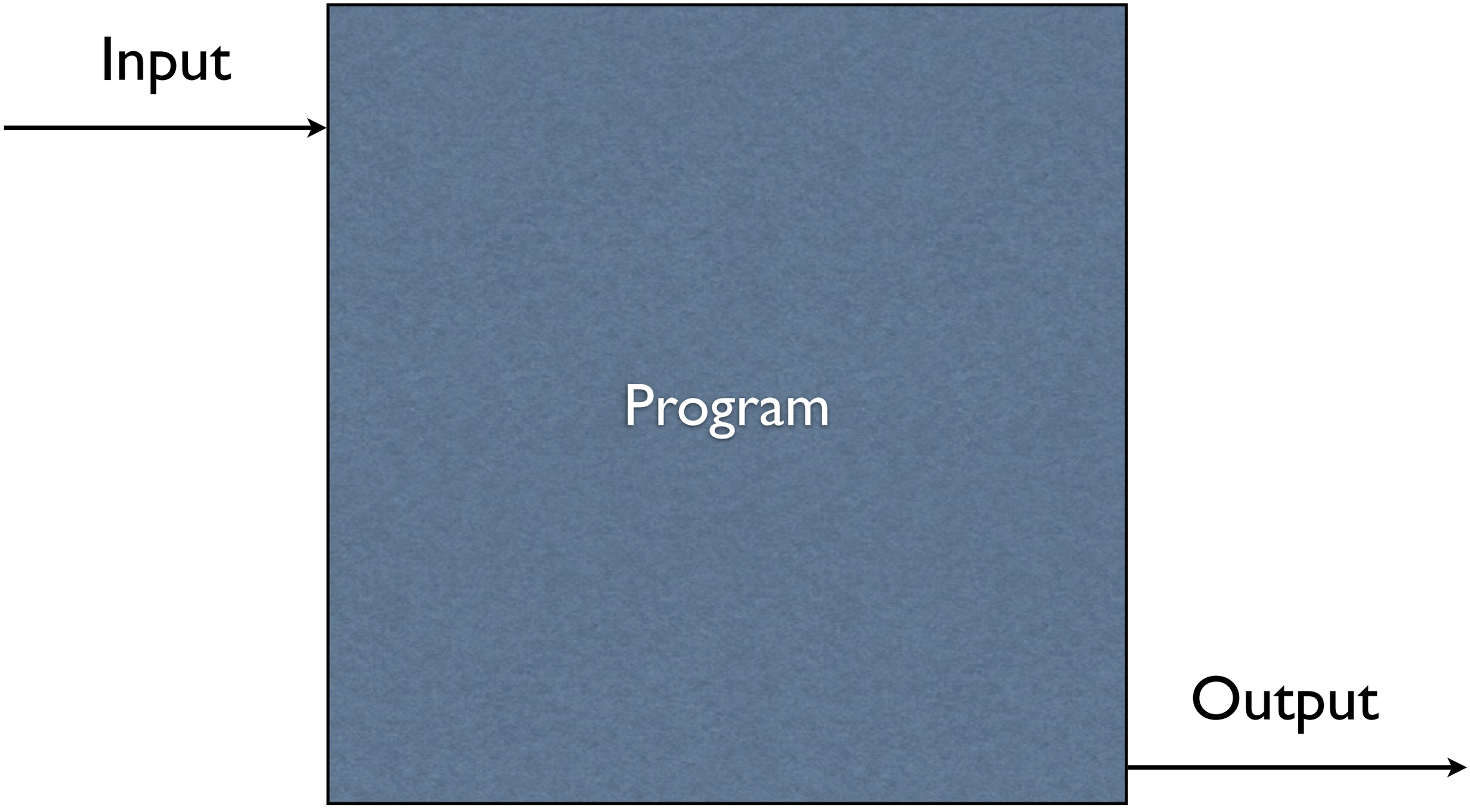
# Functions

## Divide and Conquer Revisited

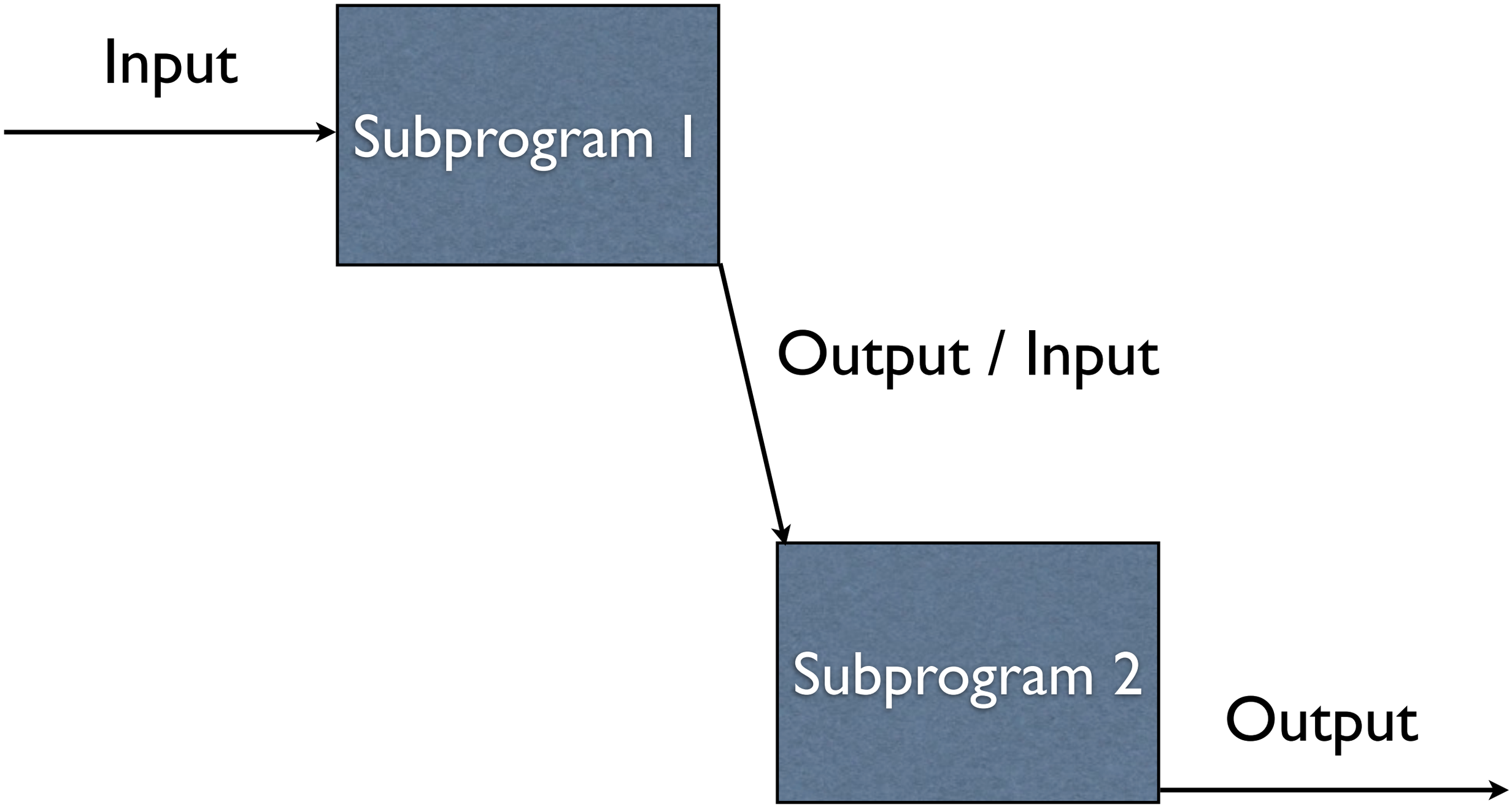
# Divide and Conquer

- Break a problem down into distinct subproblems, solve them individually, and finally combine them

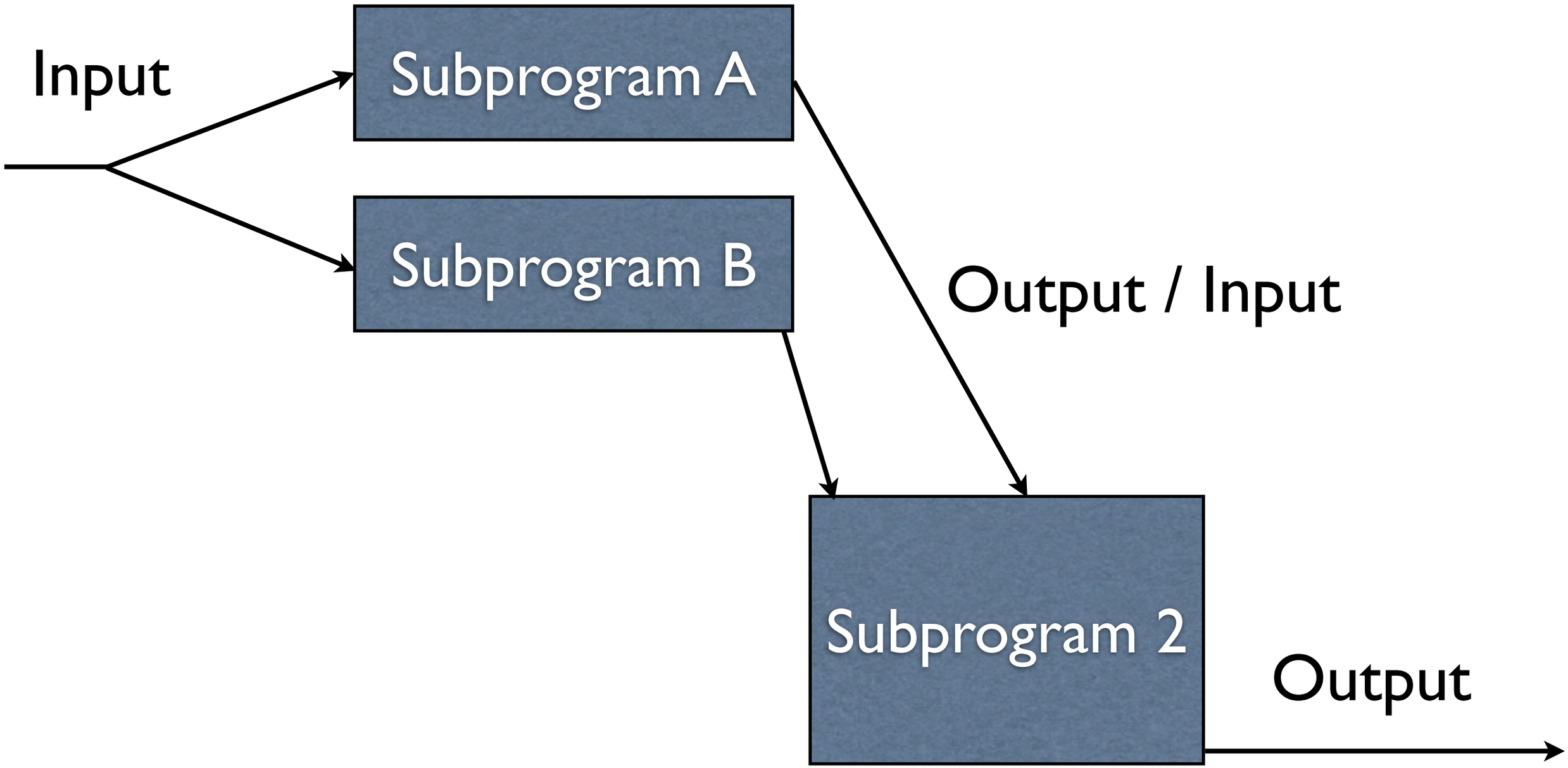
# Divide and Conquer



# Divide and Conquer

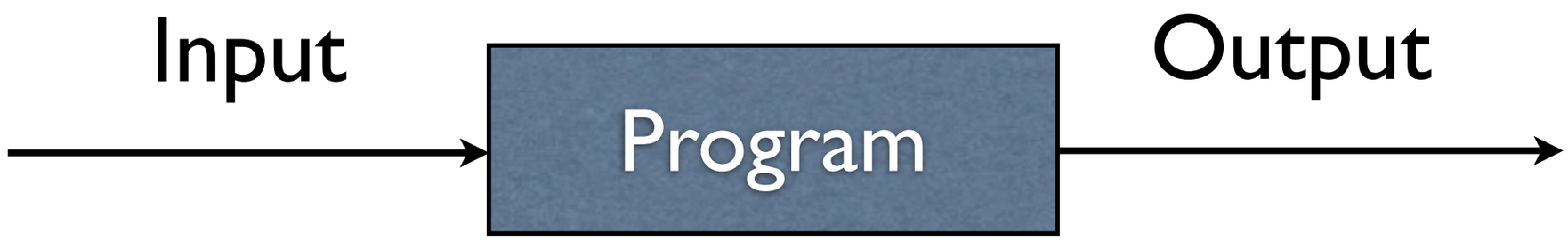


# Divide and Conquer



# Input and Output

- Intentionally left ambiguous
- This general model is widely applicable

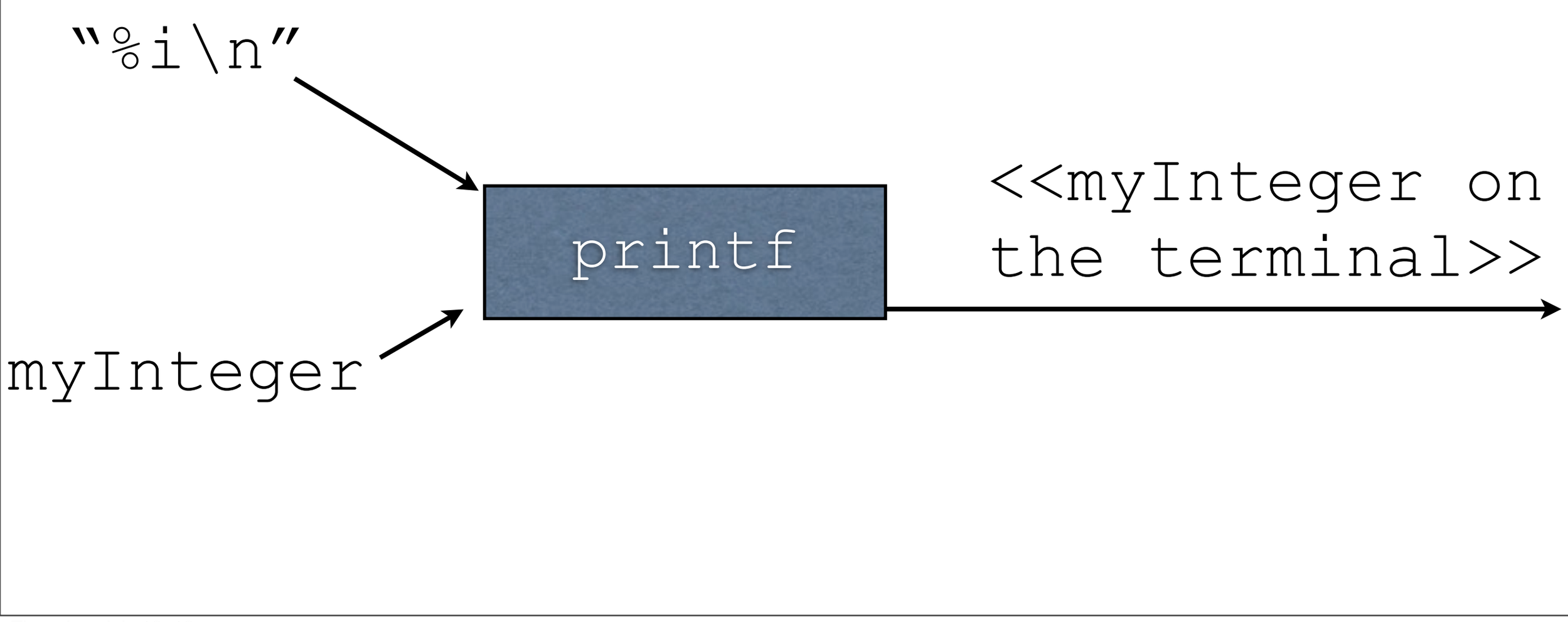


# Relation to Functions

- Consider the function `printf`

# printf Function

```
printf( "%i\n", myInteger )
```



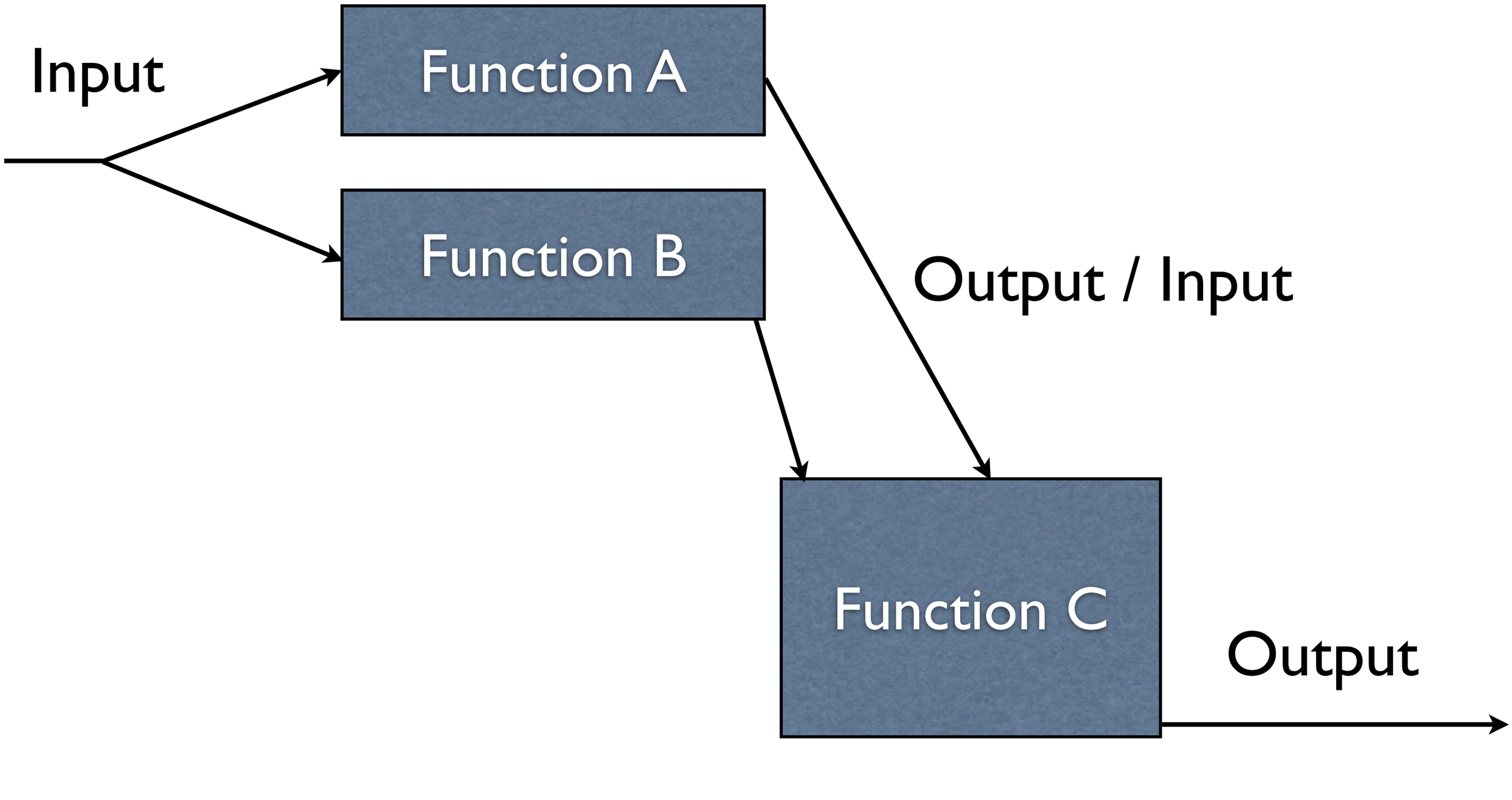
# Functions

- A way of breaking down programs into (more or less) independent units
- Can be tested individually
- Easier to think about

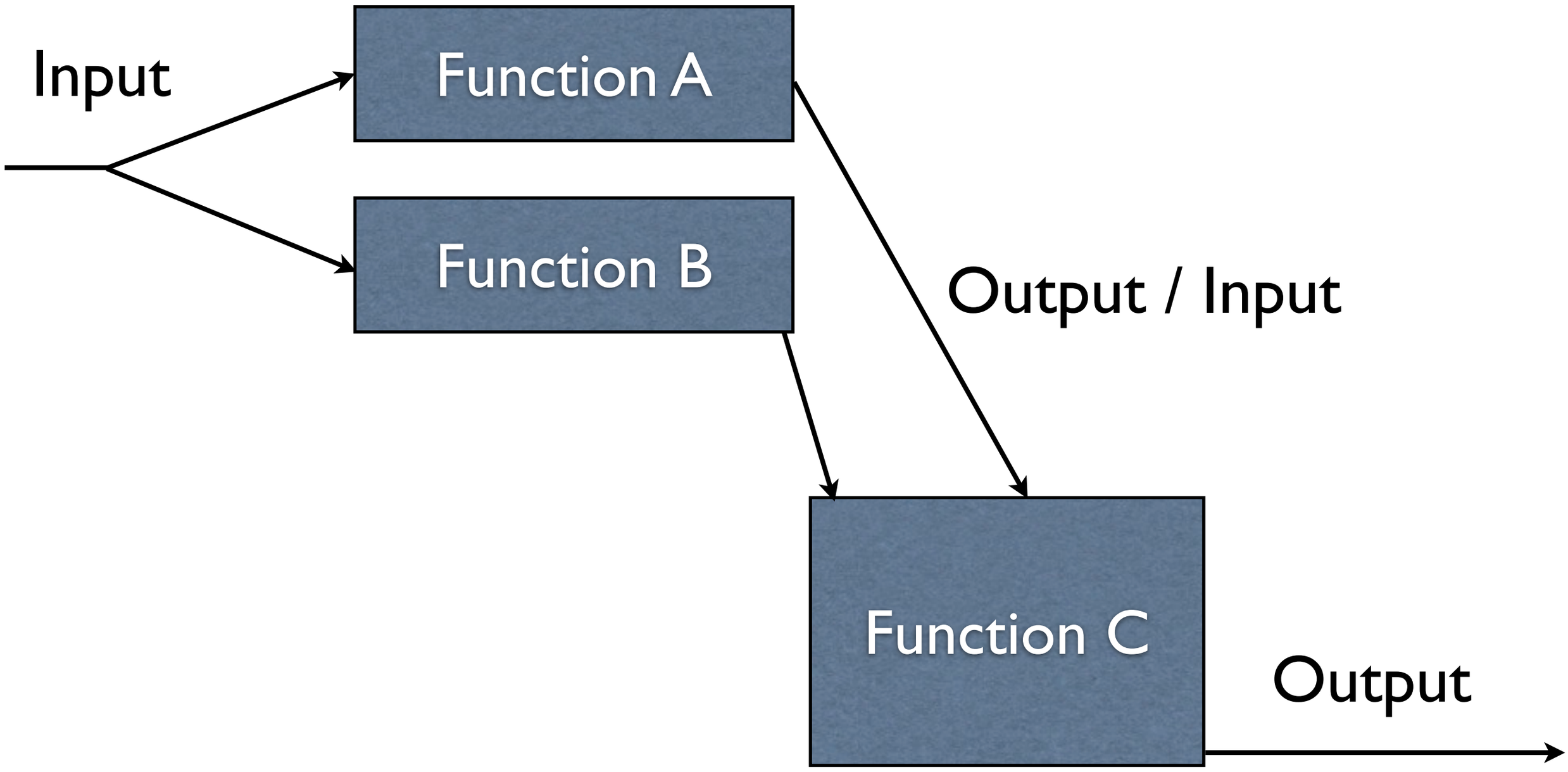
# Function Input / Output

- Input: **parameters** (more on this later)
- Output: **return value**

# Functions



# Functions




```
functionC( functionA( Input ),
           functionB( Input ) )
```

# Using Functions in C

- Function names have the same rules as variables
- Functions are **called** like so:

```
noArguments();
printf( “hi” );
printf( “%i”, myInteger );
```

# Making Functions in C

- Function definition template (`p` = parameter):

```
returnType functionName(p1,p2,...,pN)
{
   // function body
}
```

# Returning

- Functions can optionally **return** values using the `return` reserved word
- This is a special output mechanism

# Examples

```
int toSecondPower( int number ) {
   return number * number;
}
```

```
double doubleIt1( double number ) {
   return number + number;
}

double doubleIt2( double number ) {
   return number * 2;
}
```

# Bigger Example

```
int craziness( double number ) {
   int x = (int)(number * 2);
   double y = x + 2;
   int z = (int)(y * y) + x;
   return z;
}
```

# Question

- Return type doesn't match what's returned
- What happens?

```
int mismatch( double number ) {
   return number;
}
```

# Answer

- Treated as a cast to the return type

```
int mismatch( double number ) {
   return number;
}

int main() {
   // prints out 5
   printf( "%i\n", mismatch( 5.5 ) );
   return 0;
}
```

# Question

- Two returns - What happens?

```
int craziness2( double number ) {
   int x = (int)(number * 2);
   return x;
   double y = x + 2;
   int z = (int)(y * y) + x;
   return z;
}
```

# Answer

- Functions can return at most once
- Everything past the first one is ignored

```
int craziness2( double number ) {
   int x = (int)(number * 2);
   return x;
   double y = x + 2;
   int z = (int)(y * y) + x;
   return z;
}
```

# Question

- There is no explicit return
- What happens?

```
int craziness3( double number ) {
   int x = (int)(number * 2);
   double y = x + 2;
   int z = (int)(y * y) + x;
}
```

# Answer

- It will return something...but who knows what
- Undefined behavior
- `gcc` **will give a warning if given the** `-Wall` **flag**

```
gcc -Wall myProgram.c
```

# “May Return”

- Functions don’t necessarily need to return values
- Can still be useful

# `void` Return Type

- If a function has a return type of void, this mean it does not return anything

```
int globalVariable = 0;

void incrementGlobal() {
  globalVariable++;
}
```

# `void` and Return

- `return` can still be used with `void` functions
- Simply ends the execution of the function
- Important in discussing control flow

```
void something() {
  return;
}
```

# Question

- Returning something with void
- What happens?

```
void function() {
   return 5;
}
```

# Answer

- Nothing is returned
- `gcc` gives a warning about this

```
void function() {
   return 5;
}
```

# Function Prototypes

- Needed to tell the compiler a function exists
- Without them, functions have to be ordered carefully or the compiler can get confused

# Without Prototypes

```
void something() {
   return;
}

int getFive() {
   something();
   return 5;
}
```

- Compiles fine

# Without Prototypes

```
int getFive() {
  something();
  return 5;
}

void something() {
  return;
}
```

```
prototypes.c:6: warning: conflicting
types for ‘something’
prototypes.c:2: warning: previous
implicit declaration of ‘something’
was here
```

# Function Prototypes

- Look just like the definition, but they lack a body

```
returnType functionName(p1,p2,...,pN);
```

# Prototype Example

```
int getFive();
void something();

int getFive() {
   something();
   return 5;
}

void something() {
   return;
}
```

- Compiles fine

# Parameters Revisited

```
int addTen( int input ) {
   return input + 10;
}

...

addTen( 5 );
```

# Parameters Revisited

Formal Parameter

```
int addTen( int input ) {
   return input + 10;
}

...

addTen( 5 );
```

Actual Parameter

# Putting it All Together

# Example

- A program reads in a signed integer
- The program adds 50 to the integer
- The program prints the result out to the user
- `functionExample.c`