Algorithms, Data Structures, and Problem Solving

Masoumeh Taromirad

Hamlstad University

DT4002, Fall 2016

Course Objectives

- A course on algorithms, data structures, and problem solving
- Learn about algorithm complexity, algorithm design, and classical data structures
- Improve programming abilities in a programming language (we use C).

Intended Learning Outcomes

- Knowledge and Understanding
  - recognize data structures and algorithms for search and sorting, such as quick sort, binary search trees, hash tables
  - recognize techniques for algorithm design such as divide and conquer, recursion, dynamic programming
  - explain how to estimate the execution time of programs
- Skills and Ability
  - identify the need and use data structures as modules to solve larger problems
  - use techniques for algorithm design in solving larger problems
- Judgement and Approach
  - judge how suitable a program is given its execution time
  - choose adequate implementations of data structures from program libraries

Teaching Format

1. Lecture Sessions: 2-hour sessions per week
   - main lecture by me
   - guest lecturer
   - in-class activity or quiz
   - 15-minute BREAK!

2. Lab Sessions: 2-hour sessions per week
   - organised by Süleyman Savas
   - computer-based exercises

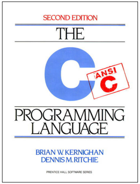
Assessment

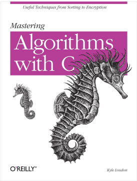
1. Project
   - at the end of the term
   - project result
   - written report
2. Examination
   - written examination of theory

Course Materials

Textbooks

- [K+R] Kernighan, Brian W., Ritchie, Dennis M. **The C Programming Language**. Prentice Hall, 1989.
- [L] Loudon, Kyle. **Mastering Algorithms with C**. OReilly & Associates, 1999.
- Sedgewick. **Algorithms in C**, Parts 1-4, Third Edition. ISBN 0-201-31452-5. [recommended]





Course Web Page

- Lecture handouts and slides are available at

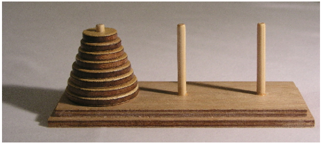
  http://ceres.hh.se/mediawiki/DA_4002_2016

- Check the page for updates and announcements!

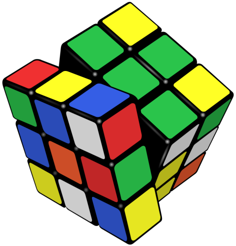
Any Question?

Algorithms

- recurring solutions to **recurring problems**
- recipes for solving **new problems**



$\Rightarrow$



- understanding trade-offs to make **informed choices** in specific situations

Objectives

Study algorithms

- procedure for solving a problem
- finite number of steps
- involves repetition of an operation

Objectives

- lay foundations **reusable** software development in any language (we'll use C)
- **master** daily programming
- **handle** trickier tasks

Let's start!

C Tutorial*

(*slides form previous years)

C Tutorial

[K+R] chapter 1 *(skip section 1.10)*

Hello, World

```
#include <stdio.h>

int main (int argc, char **argv)
{
  printf (“hello, world\n”);
  return 0;
}
```

Hello, World

import standard I/O functions

every program has a main function

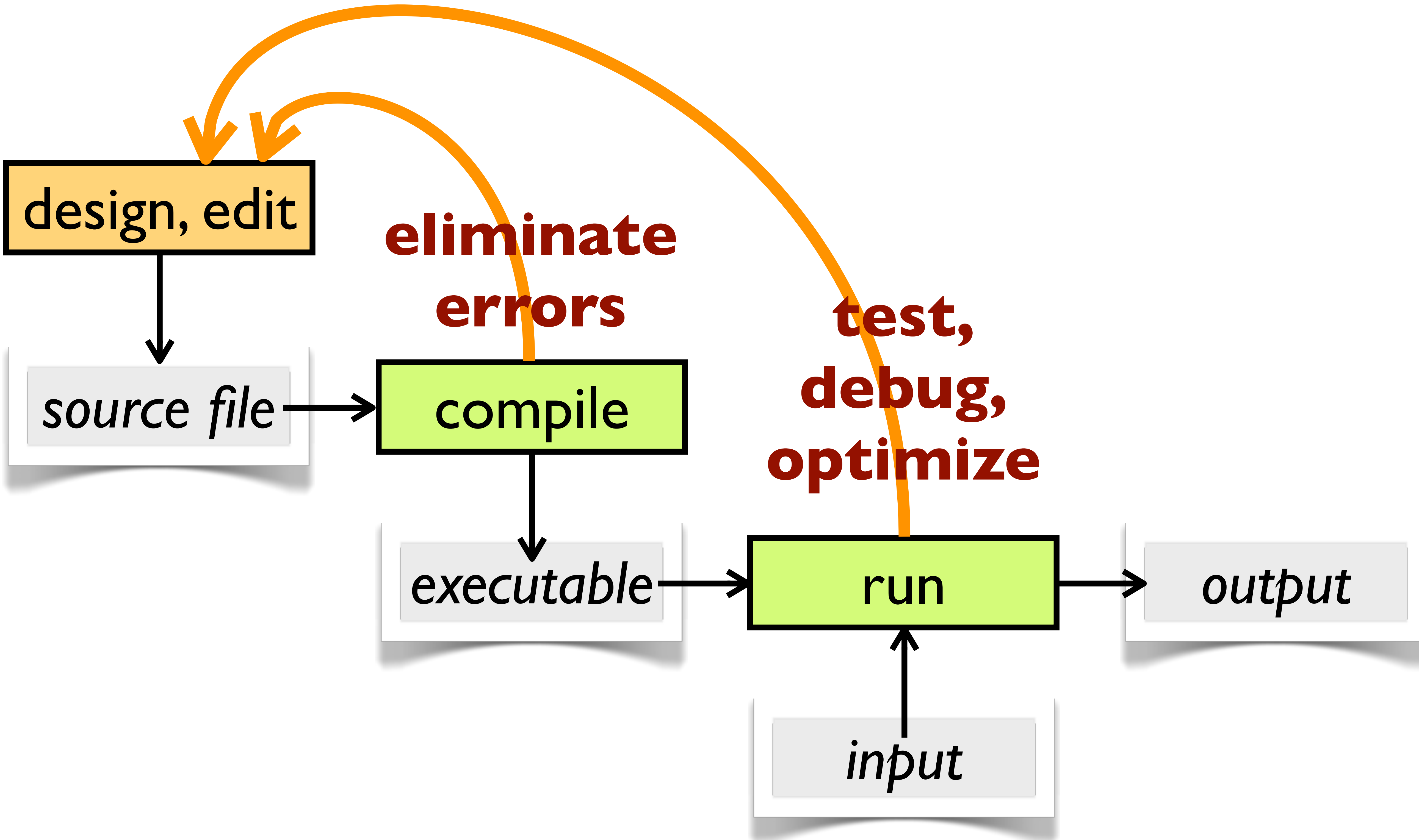
```
#include <stdio.h>

int main (int argc, char **argv)
{
  printf (“hello, world\n”);
  return 0;
}
```

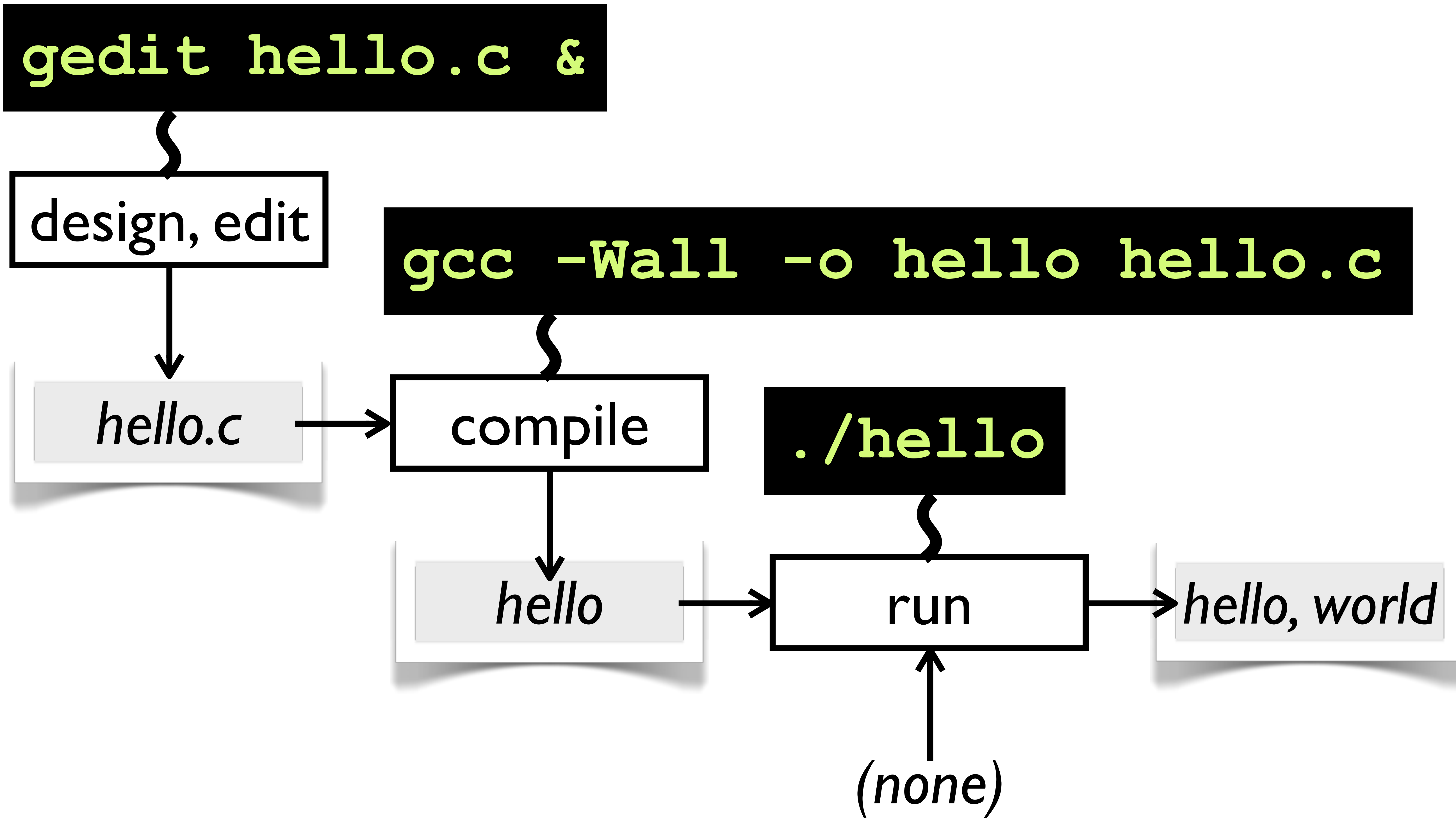
call a function to print a string

return an exit code

C Programming Workflow



C Programming Workflow



Variables, Operators, Flow

```
/* print Fahrenheit to Celsius table */
int main (int argc, char **argv)
{
  int fahr, cels;
  fahr = 0;
  while (fahr <= 300) {
    cels = 5 * (fahr - 32) / 9;
    printf (“%d\t%d\n”, fahr, cels);
    fahr = fahr + 20;
  }
  return 0;
}
```

Variables, Operators, Flow

comment

```
/* print Fahrenheit to Celsius table */
int main (int argc, char **argv)
{
  int fahr, cels;
  fahr = 0;
  while (fahr <= 300) {
    cels = 5 * (fahr - 32) / 9;
    printf ("%d\t%d\n", fahr, cels);
    fahr = fahr + 20;
  }
  return 0;
}
```

variables

while loop

Variables, Operators, Flow

```
/* print Fahrenheit to Celsius table */
int main (int argc, char **argv)
{
   int fahr, cels;
   fahr = 0;
   while (fahr <= 300) {
     cels = 5 * (fahr - 32) / 9;
     printf (“%d\t%d\n”, fahr, cels);
     fahr = fahr + 20;
   }
   return 0;
}
```

type

name

loop condition

loop body

Variables, Operators, Flow

```
/* print Fahrenheit to Celsius table */
int main (int argc, char **argv)
{
              cels;
   fahr = 0;
   while (fahr <= 300) {
      cels = 5 * (fahr - 32) / 9;
               \t%d\n    , cels);
      fahr = fahr + 20;
   }
   return 0;
}
```

assign

compare

divide

multiply

subtract

add

Types

```
int    an_integer;
short  a_small_int;
long   a_big_int;
char   a_single_character;
float  almost_a_real_number;
double more_precise;

unsigned int   a_positive_int;
unsigned short a_small_positive_int;
unsigned long  a_big_positive_int;
unsigned char  an_unsigned_char;

/* no “unsigned” float or double */
```

More precise arithmetic

```
int main (int argc, char **argv)
{
  int fahr, cels;
  fahr = 0;
  while (fahr <= 300) {
    cels = 5 * (fahr - 32) / 9;
    printf ("%d\t%d\n", fahr, cels);
    fahr = fahr + 20;
  }
  return 0;
}
```

truncated integer division

More precise arithmetic

```
int main (int argc, char **argv)
{
   double fahr, cels;
   fahr = 0.0;
   while (fahr <= 300.0) {
      cels = 5.0 * (fahr - 32.0) / 9.0;
      printf ("%f\t%f\n", fahr, cels);
      fahr = fahr + 20.0;
   }
   return 0;
}
```

precise floating-point division

with `int`	with `double`
60 15 80 26 100 37	60.00 15.555 80.00 26.667 100.00 37.778

There's a more compact way...

```
int main (int argc, char **argv)
{
  double fahr, cels;
  fahr = 0.0;
  while (fahr <= 300.0) {
    cels = 5.0 * (fahr - 32.0) / 9.0;
    printf (“%f\t%f\n”, fahr, cels);
    fahr = fahr + 20.0;
  }
  return 0;
}
```

There's a more compact way...

```
int main (int argc, char **argv)
{
  double fahr;
  for (fahr = 0; fahr <= 300; fahr += 20)
    printf ("%f\t%f\n", fahr, 5*(fahr-32)/9);
  return 0;
}
```

There's a more compact way...

```
int main (int argc, char **argv)
{
   double fahr;
   for (fahr = 0; fahr <= 300; fahr += 20)
     printf ("%f\t%f\n", fahr, 5*(fahr-32)/9);
   return 0;
}
```

also: −= *= /= ...

converted to floating-point

- convenient for-loop syntax
- convenient update operators
- no need for explicit **`cels`** variable
- single-statement bodies don't need {}
- automatic type conversion *(but be careful...)*

Character Input & Output

```
int c;
unsigned long n;
n = 0;
c = getchar();
while (EOF != c) {
  putchar (c);
  ++n;
  c = getchar();
}
printf ("\ntotal: %lu\n", n);
```

More Compact Character I/O

```
int c;
unsigned long n = 0;
```

combined initialization and declaration

```
while (EOF != (c = getchar())) {
  putchar (c);
  ++n;

}
printf ("\ntotal: %lu\n", n);
```

the value of an assignment is the value of its left-hand side

More Compact Character I/O

```
int c;
unsigned long n = 0;
while (EOF != (c = getchar())) {
  putchar (c);
  ++n;
}
printf ("\ntotal: %lu\n", n);
```

Counting Lines

```
int c;
unsigned long n = 0;
while (EOF != (c = getchar())) {
   putchar (c);
   if ('\n' == c)
     ++n;
}
printf ("\n%lu lines\n", n);
```

Counting Words

```
#define IN  1
#define OUT 0

int c, state = OUT;
unsigned long n = 0;
while (EOF != (c = getchar())) {
   putchar (c);
   if (ispunct(c) || isspace(c))
     state = OUT;
   else if (state == OUT) {
     state = IN;
     ++n;
   }
}
printf ("\n%lu words\n", n);
```

Counting Words

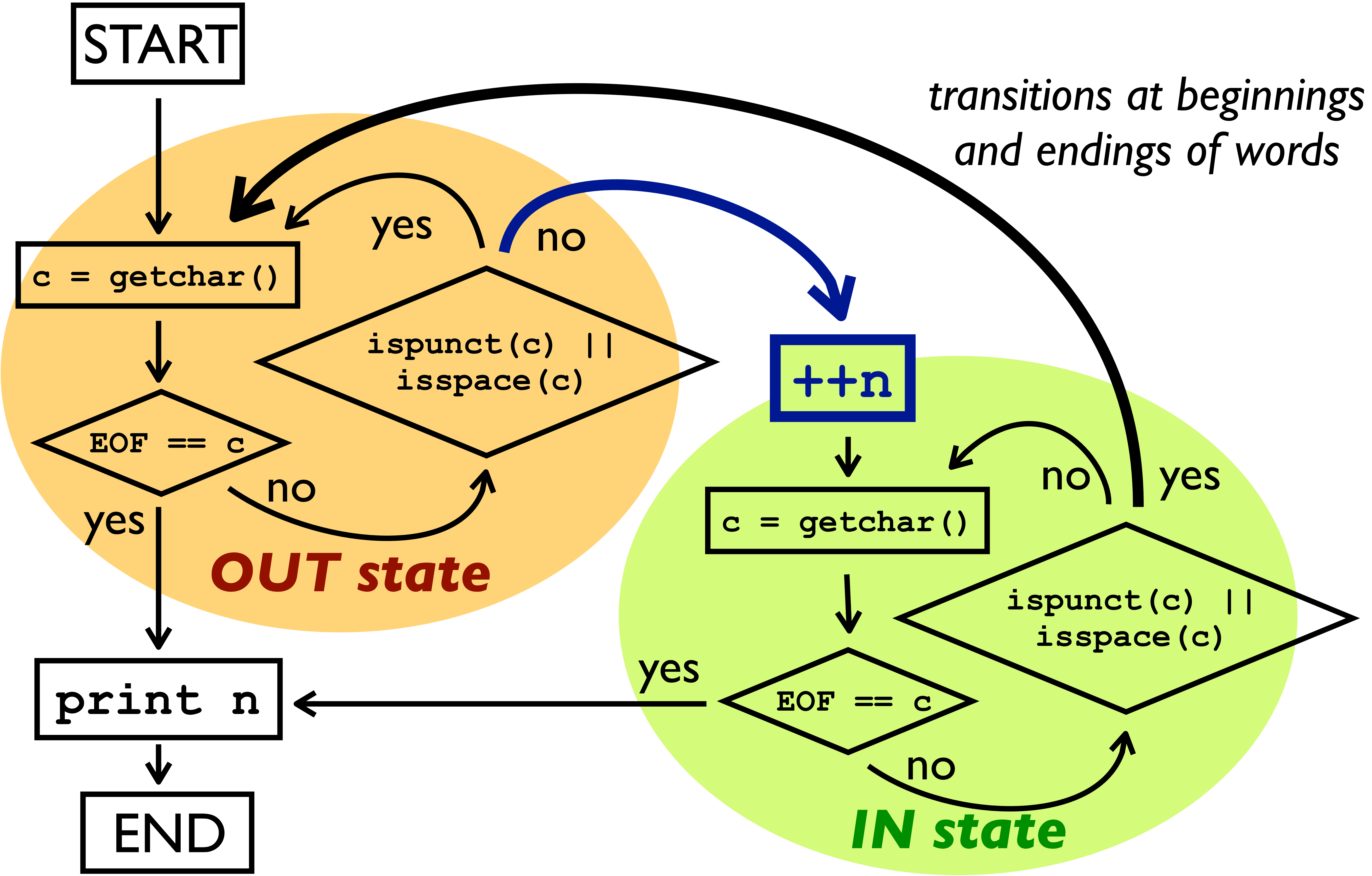
```
#define IN  1
#define OUT 0

int c, state = OUT;
unsigned long n = 0;
while (EOF != (c = getchar())) {
   putchar (c);
   if (ispunct(c) || isspace(c))
    state = OUT;
   else if (state == OUT) {
    state = IN;
    ++n;
   }
}
printf ("\n%lu words\n", n);
```

compile-time constants

from <ctype.h>

Counting Words Finite State Machine



Arrays

idx	num[idx]
0	0
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81

```
int idx;
int num[10];
for (idx = 0; idx < 10; ++idx)
   num[idx] = idx * idx;
```

- block of N identically typed values
- in declarations:
  `[]` reserves memory for N values
- in expressions:
  `[]` accesses individual values
- the **first** element is at index ***zero***
- the **last** element is at index ***N-1***

Counting Digits

```
int c;
unsigned long n[10];
memset (n, 0, sizeof(n));
while (EOF != (c = getchar())) {
  putchar (c);
  if ('0' <= c && '9' >= c)
    ++n[c - '0'];
}
printf ("\ndigits:\n");
for (c = 0; c < 10; ++c)
  printf ("  %d:\t%lu\n", c, n[c]);
```

digit	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII code	48	49	50	51	52	53	54	55	56	57

from `<string.h>`

array length ***in bytes***

```
int c;
unsigned long n[10];
memset (n, 0, sizeof(n));
while (EOF != (c = getchar())) {
   putchar (c);
   if ('0' <= c && '9' >= c)
      ++n[c - '0'];
}
printf ("\ndigits:\n");
for (c = 0; c < 10; ++c)
   printf ("  %d:\t%lu\n", c, n[c]);
```

this trick works only for ASCII

Functions

- make it possible to focus on **what** is done, without worrying about ***how*** it is done
- two steps to create a C function
  1. **declaration**:
     - give it a name
     - describe what it depends on
     - describe what it produces
  2. **definition**:
     - write down the sequence of statements that implement it

these two steps can be combined into one

Functions

```
int power (int base, int n)
{
  int i, p;
  p = 1;
  for (i = 1; i <= n; ++i)
    p = p * base;
  return p;
}

int main (int argc, char ** argv)
{
  int i;
  for (i = 0; i < 10; ++i)
    printf("%d %d", power(2, i), power(-3, i));
  return 0;
}
```

Functions

```
int power (int base, int n)
{
   int i, p;
   p = 1;
   for (i = 1; i <= n; ++i)
      p = p * base;
   return p;
}
```

combined declaration + definition

function calls

```
int main (int argc, char ** argv)
{
   int i;
   for (i = 0; i < 10; ++i)
      printf("%d %d", power(2, i), power(-3, i));
   return 0;
}
```

Functions

```
int power (int base, int n)
{
   int i, p;
   p = 1;
   for (i = 1; i <= n; ++i)
      p = p * base;
   return p;
}
```

signature

body

```
int main (int argc, char ** argv)
{
   int i;
   for (i = 0; i < 10; ++i)
      printf(“%d %d”, power(2, i), power(-3, i));
   return 0;
}
```

Functions

function name

argument list

type

name

return type

local variables

return statement

```
int power (int base, int n)
{
    int i, p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Function Calls

```
    p = p * base;
  return p;
}

int main (int argc, char ** argv)
{
  int i;
  for (i = 0; i < 10; ++i)
    printf("%d %d", power(2, i), power(-3, i));
  return 0;
}
```

after call:
place return value
from function
into caller

before call:
copy arguments
from caller
into function

```
int power (int base, int n)
{
  int i, p;
```

Call by Value

```
int power (int base, int n)
{
   int i, p;
   p = 1;
   for (i = 1; i <= n; ++i)
      p = p * base;
   return p;
}

int main (int argc, char ** argv)
{
   int i;
   for (i = 0; i < 10; ++i)
      printf(“%d %d”, power(2, i), power(-3, i));
   return 0;
}
```

Call by Value

```
int power (int base, int n)
{
  int i, p;
  p = 1;
  for (i = 1; i <= n; ++i)
    p = p * base;
  return p;
}

int main (int argc, char ** argv)
{
  int i;
  for (i = 0; i < 10; ++i)
    printf("%d %d", power(2, i), power(-3, i));
  return 0;
}
```

the function gets its own fresh copy of each argument!

Call by Value

```
int power (int base, int n)
{
  int p;

  for (p = 1; n > 0 ; --n)
    p = p * base;
  return p;
}

int main (int argc, char ** argv)
{
  int i;
  for (i = 0; i < 10; ++i)
    printf("%d %d", power(2, i), power(-3, i));
  return 0;
}
```

modifying an argument in here has no outside effect!

Call by Value

```
int power (int base, int n)
{
   int p;

   for (p = 1; n > 0 ; --n)
      p = p * base;
   return p;
}

int main (int argc, char ** argv)
{
```

function arguments are local variables that get initialized by the caller

we've been *using* functions all this time already!

```
     printf("%d %d", power(2, i), power(-3, i));
   return 0;
}
```

...almost done...

strings and a first look at pointers

Strings

- remember arrays?
  block of N identically typed values
- strings are arrays of characters
- strings always end with a zero
  so we can use them without knowing their size beforehand

Strings

- for example: “**`itads2013`**”

- remember arrays?
  block of N identically typed values
- strings are arrays of characters
- strings always end with a zero
  so we can use them without knowing their size beforehand

idx	char[idx]	ASCII
0	`'i'`	105
1	`'t'`	116
2	`'a'`	97
3	`'d'`	100
4	`'s'`	115
5	`'2'`	50
6	`'0'`	48
7	`'1'`	49
8	`'3'`	51
9	`'\0'`	0

Strings

```
#include <stdio.h>

int main (int argc, char **argv)
{
  printf (“hello, world\n”);
  return 0;
}
```

h	**e**	**l**	**l**	**o**	**,**		**w**	**o**	**r**	**l**	**d**	**\n**	**\0**
104	**101**	**108**	**108**	**111**	**44**	**32**	**119**	**111**	**114**	**108**	**100**	**10**	**0**

Strings

```
#include <stdio.h>

int main (int argc, char **argv)
{
  printf (“hello, world\n”);
  return 0;
}
```

but how does C pass this into a function?

h	e	l	l	o	,		w	o	r	l	d	\n	\0
104	101	108	108	111	44	32	119	111	114	108	100	10	0

a first look at

Pointers

- data “live” in RAM at various **addresses**
- variables usually hide that detail
- but it can be necessary to use addresses

- for now just show the principle:
    - “*****“ in <u>declarations</u>
    - “*****“ and “**&**“ in <u>statements</u>
- *later in this course we will use pointers for*
    - *links between data elements*
    - *customizing data structure functions*
    - *mutable function arguments*

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;
```

memory *(hexadecimal)*

address	value
7028	02
7029	82
702a	a7
702b	af
702c	ef
702d	
702e	
702f	
7030	
7031	
7032	
7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;
```

memory *(hexadecimal)*

	address	value
foo:	7028	02
	7029	82
	702a	a7
	702b	af
bar:	702c	ef
	702d	
	702e	
	702f	
	7030	
	7031	
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;
```

"*****" in <u>declaration</u> means:
`fp` stores an address
where an **`int`** is stored

memory *(hexadecimal)*

	address	value
foo:	7028	02
	7029	82
	702a	a7
	702b	af
bar:	702c	ef
	702d	
fp:	702e	
	702f	
	7030	
	7031	
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;


fp = &foo;
```

memory *(hexadecimal)*

	address	value
foo:	7028	02
	7029	82
	702a	a7
	702b	af
bar:	702c	ef
	702d	
fp:	702e	
	702f	
	7030	
	7031	
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;

fp = &foo;
```

“**&**” means: get the **address** of the variable

memory *(hexadecimal)*

	address	value
foo:	7028	02
	7029	82
	702a	a7
	702b	af
bar:	702c	ef
	702d	
fp:	702e	
	702f	
	7030	
	7031	
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;

fp = &foo = 0x7028
```

memory *(hexadecimal)*

	address	value
foo:	7028	02
	7029	82
	702a	a7
	702b	af
bar:	702c	ef
	702d	
fp:	702e	**70**
	702f	**28**
	7030	
	7031	
	7032	
	7033	

"**=**" means: store it in the variable **`fp`**

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;

fp = &foo;
```

pointers are variables that store an address

memory *(hexadecimal)*

	address	value
foo:	7028	02
	7029	82
	702a	a7
	702b	af
bar:	702c	ef
	702d	
fp:	702e	70
	702f	28
	7030	
	7031	
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;
char *bp = &bar;

fp = &foo;
```

the same principle applies to any type

memory *(hexadecimal)*

	address	value
foo:	7028	02
	7029	82
	702a	a7
	702b	af
bar:	702c	ef
	702d	
fp:	702e	70
	702f	28
bp:	7030	**70**
	7031	**2c**
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;
char *bp = &bar;

fp = &foo;
*fp = -17;
```

memory *(hexadecimal)*

	address	value
foo:	7028	02
	7029	82
	702a	a7
	702b	af
bar:	702c	ef
	702d	
fp:	702e	70
	702f	28
bp:	7030	70
	7031	2c
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;
char *bp = &bar;

fp = &foo;
*fp = -17;
```

"*" in <u>statement</u> means:
(the **int** value) stored at
(the address stored at **fp**)

memory *(hexadecimal)*

	address	value
foo:	7028	02
	7029	82
	702a	a7
	702b	af
bar:	702c	ef
	702d	
fp:	702e	70
	702f	28
bp:	7030	70
	7031	2c
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;
char *bp = &bar;

fp = &foo;
*fp = -17;
```

"*****" in <u>statement</u> means:
(the **int** value) stored at
(the address **0x7028**)

memory *(hexadecimal)*

	address	value
foo:	7028	02
	7029	82
	702a	a7
	702b	af
bar:	702c	ef
	702d	
fp:	702e	70
	702f	28
bp:	7030	70
	7031	2c
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;
char *bp = &bar;

fp = &foo;
*fp = -17;
```

"=" means:
write -17

"*" in statement means:
(the **int** value) stored at
(the address **0x7028**)

memory *(hexadecimal)*

	address	value
foo:	7028	
	7029	
	702a	
	702b	
bar:	702c	ef
	702d	
fp:	702e	70
	702f	28
bp:	7030	70
	7031	2c
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;
char *bp = &bar;

fp = &foo;
*fp = -17;
```

"=" means:
write -17

"*" in statement means:
(the **int** value) stored at
(the address **0x7028**)

memory *(hexadecimal)*

	address	value
foo:	7028	**ff**
	7029	**ff**
	702a	**ff**
	702b	**ef**
bar:	702c	ef
	702d	
fp:	702e	70
	702f	28
bp:	7030	70
	7031	2c
	7032	
	7033	

a first look at

Pointers

```
int foo = 42117039;
char bar = -17;

int *fp;
char *bp = &bar;

fp = &foo;
*fp = -17;
```

when dealing with pointers, a table like this can help a lot

memory *(hexadecimal)*

	address	value
foo:	7028	ff
	7029	ff
	702a	ff
	702b	ef
bar:	702c	ef
	702d	
fp:	702e	70
	702f	28
bp:	7030	70
	7031	2c
	7032	
	7033	

Back to the Question

```
#include <stdio.h>

int main (int argc, char **argv)
{
  printf (“hello, world\n”);
  return 0;
}
```

but how does C pass this into a function?

h	**e**	**l**	**l**	**o**	**,**		**w**	**o**	**r**	**l**	**d**	**\n**	**\0**
104	**101**	**108**	**108**	**111**	**44**	**32**	**119**	**111**	**114**	**108**	**100**	**10**	**0**

Back to the Question

```
#include <stdio.h>

int main (int argc, char **argv)
{
   printf (    0x7028    );
   return 0;
}
```

just pass the address of the first letter!

7028	7029	702a	702b	702c	702d	702e	702f	7030	7031	7032	7033	7034	7035
h	e	l	l	o	,		w	o	r	l	d	\n	\0
104	101	108	108	111	44	32	119	111	114	108	100	10	0

Back to the Question

```
#include <stdio.h>

int main (       char **argv)
{
   printf
   return
}
```

thankfully, this is usually completely hidden from the programmer

7028	7029	702a	702b	702c	702d	702e	702f	7030	7031	7032	7033	7034	7035
h	e	l	l	o	,		w	o	r	l	d	\n	\0
104	101	108	108	111	44	32	119	111	114	108	100	10	0

Pointers?
Enough for Now!

- pointers seem to be the most confusing part of C for many people
- but once you understand them, everything becomes so easy:
  - strings are just pointers
  - arrays are just pointer
  - even function are just pointers

...exercises: practice makes perfect...

Take-Home Message

- values:
  - types and variables
  - arrays, strings, pointers
- flow:
  - if and else
  - for and while loops
- functions:
  - argument list
  - return type
  - pass by value

Reading List

- for this week *(for exercises 1 + 2)*
  [K+R] 2.1-10, 3.1-5, 3.7, 5.1, 5. 3, 5. 4, 7.1-2
- for next week *(lecture 2, exercises 3 + 4)*
  [K+R] 1.1-10, 4.1-2, 4.8, 4.10, 5.2, 6.1-4, 6.7