COS 217: Introduction to Programming Systems

Interfaces and Implementations*
Using the Smallest C Data Type: Character

*Note: We will have more lectures on this topic in a more general context later

PRINCETON UNIVERSITY

Agenda

Interface, implementation and design decisions for characters

Interfaces and implementations for the human reader

(Another difference from Java: Variable declarations in C89)

Getting Ready for an iClicker Question …

Q: Is the **if** statement in `upper` really necessary?

A. Gee, I don't know.
Let me check
the man page
(again)

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int c;
    while ((c = getchar()) != EOF) {
        if (islower(c))
            c = toupper(c);
        putchar(c);
    }
    return 0;
}
```

ctype.h Functions

```
$ man toupper
NAME
      toupper, tolower - convert letter to upper or lower case

SYNOPSIS
      #include <ctype.h>
      int toupper(int c);
      int tolower(int c);

DESCRIPTION
      toupper() converts the letter c to upper case, if possible.
      tolower() converts the letter c to lower case, if possible.

      If c is not an unsigned char value, or EOF, the behavior of
      these functions is undefined.

RETURN VALUE
      The value returned is that of the converted letter,
      or c if the conversion was not possible.
```

iClicker Question

Q: Is the **if** statement really necessary?

A. Yes, necessary for correctness.

B. Not necessary, but I'd leave it in.

C. Not necessary, and I'd get rid of it.

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int c;
    while ((c = getchar()) != EOF) {
        if (islower(c))
            c = toupper(c);
        putchar(c);
    }
    return 0;
}
```

To Use Characters in Programs, What Do We Need?

- A representation for characters
- Ways to input and output characters
- Ways to manipulate characters
  - Convert from lowercase to uppercase, etc.

Character Representation: The ASCII Standard

Mapping from integer values to characters:

ASCII (American Standard Code for Information Interchange) (/'æski/)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NUL									HT	LF					
16																
32	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

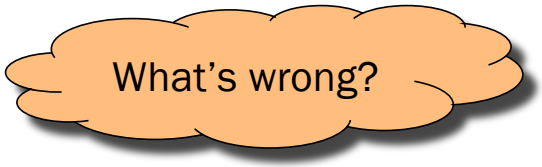
Notes: Many non-printing characters are left blank in table above

UPPER-CASE and lower-case letters are 32 apart

... but they're internally contiguous. So are digits 0 through 9.

Converting to Uppercase: A shot at `toupper()`

```
if ((c >= 97) && (c <= 122))
        c -= 32;
```



A Different Representation/Implementation: EBCDIC

Extended Binary Coded Decimal Interchange Code (/ˈɛbsɪdɪk/)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NUL					HT										
16																
32						LF										
48																
64	SP											.	<	(	+	|
80	&										!	$	*	)	;	
96	-	/									|	,	%	_	>	?
112										`	:	#	@	'	=	"
128		a	b	c	d	e	f	g	h	i		{				
144		j	k	l	m	n	o	p	q	r		}				
160		~	s	t	u	v	w	x	y	z						
176																
192		A	B	C	D	E	F	G	H	I						
208		J	K	L	M	N	O	P	Q	R						
224	\		S	T	U	V	W	X	Y	Z						
240	0	1	2	3	4	5	6	7	8	9						

Partial map

C Design Decision: Provide Character Literals

C will translate a literal to different integer values in different encodings

Single quote syntax: 'a' is a value of type char with value 97 in ASCII, 129 in EBCDIC

Use backslash to write special characters

- Backslash says: the next character isn't what you think it is; the backslash and it represent something else

```
'a'    the a character (97, 129)
'A'    the A character (65, 193)
'0'    the zero character (48, 240)
'\0'   the NUL (nullbyte) character (0, 0)
'\n'   the newline character (10, 37)
'\t'   the horizontal tab character (9, 5)
'\''   the single quote character (39, 125)
'"'    the double quote character (34, 127)
'\\'   the backslash character (92, 224)
```

Converting to Uppercase: Version 2

```
if ((c >= 'a') && (c <= 'z'))
        c += 'A' - 'a';
```

Arithmetic on chars?

What's wrong now?

Recall EBCDIC

Extended Binary Coded Decimal Interchange Code

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NUL					HT										
16																
32						LF										
48																
64	SP											.	<	(	+	|
80	&										!	$	*	)	;	
96	-	/									|	,	%	_	>	?
112										`	:	#	@	'	=	"
128		a	b	c	d	e	f	g	h	i		{				
144		j	k	l	m	n	o	p	q	r		}				
160		~	s	t	u	v	w	x	y	z						
176																
192		A	B	C	D	E	F	G	H	I						
208		J	K	L	M	N	O	P	Q	R						
224	\		S	T	U	V	W	X	Y	Z						
240	0	1	2	3	4	5	6	7	8	9						

Partial map

Note: UPPER CASE characters not contiguous; same for lower case.

Converting to Uppercase: Version 3

- Provide a real **interface**:
  - Character data type
  - API for operations on characters
    - Works on all machines and representations
    - **Implemented** differently for different machines and representations (ASCII, EBCDIC, etc)

```
#include <ctype.h>

if (islower(c))
        c = toupper(c);
```

What about Input/Output?

C does not provide I/O facilities in the language :

- They're provided in the standard library, declared in stdio.h
  - Constant: EOF
  - Data type: FILE (described later in course)
  - Variables: stdin, stdout, and stderr
  - Functions: (numerous)

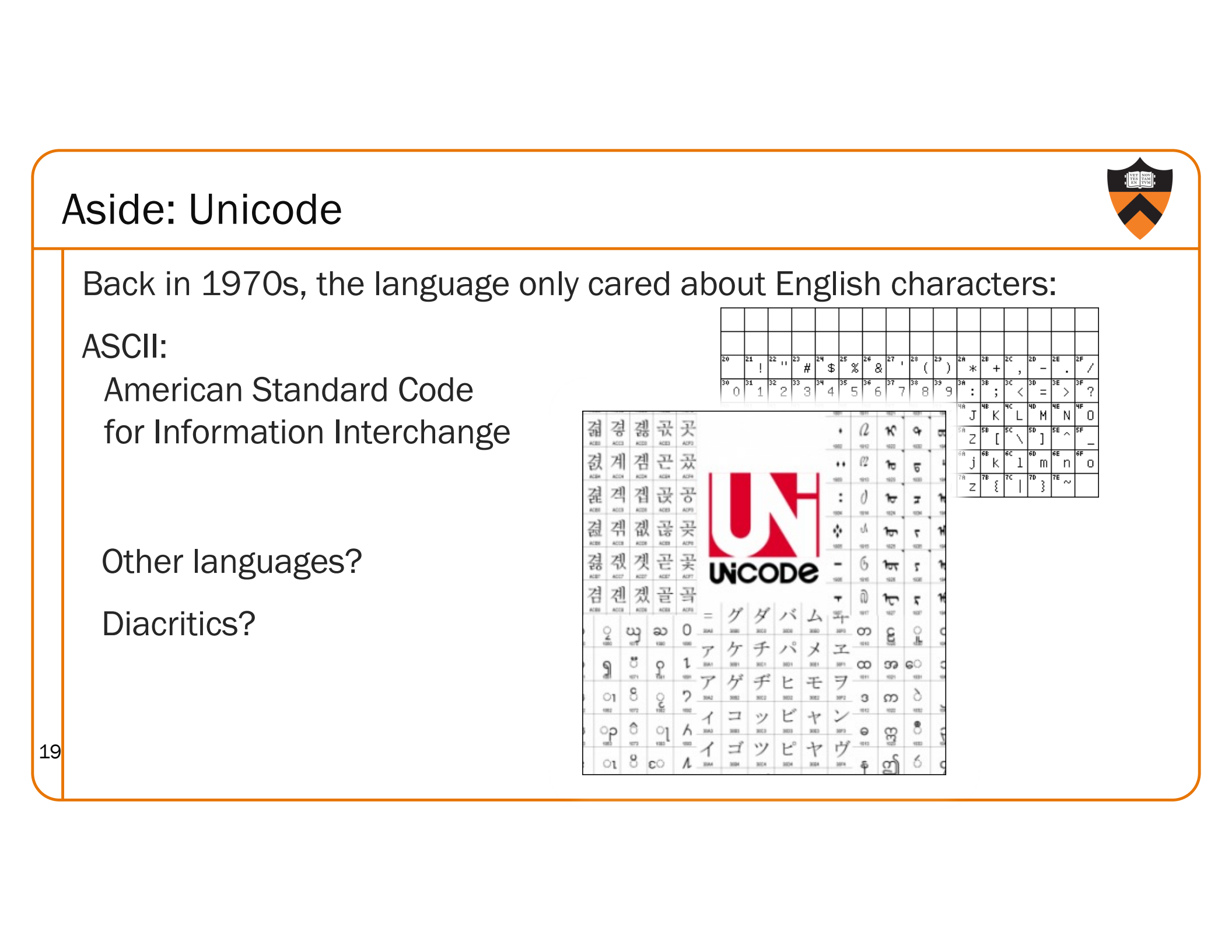
Reading characters

- getchar() function with return type wider than char (specifically, int)
- Returns EOF (a special non-character int) to indicate failure
- Reminder: there is no such thing as "the EOF character"

Writing characters

- putchar() function accepting one parameter
- Doesn't need to be an int, but for symmetry with getchar(), parameter is an int

Aside: Unicode



Back in 1970s, the language only cared about English characters:

ASCII:

American Standard Code
for Information Interchange

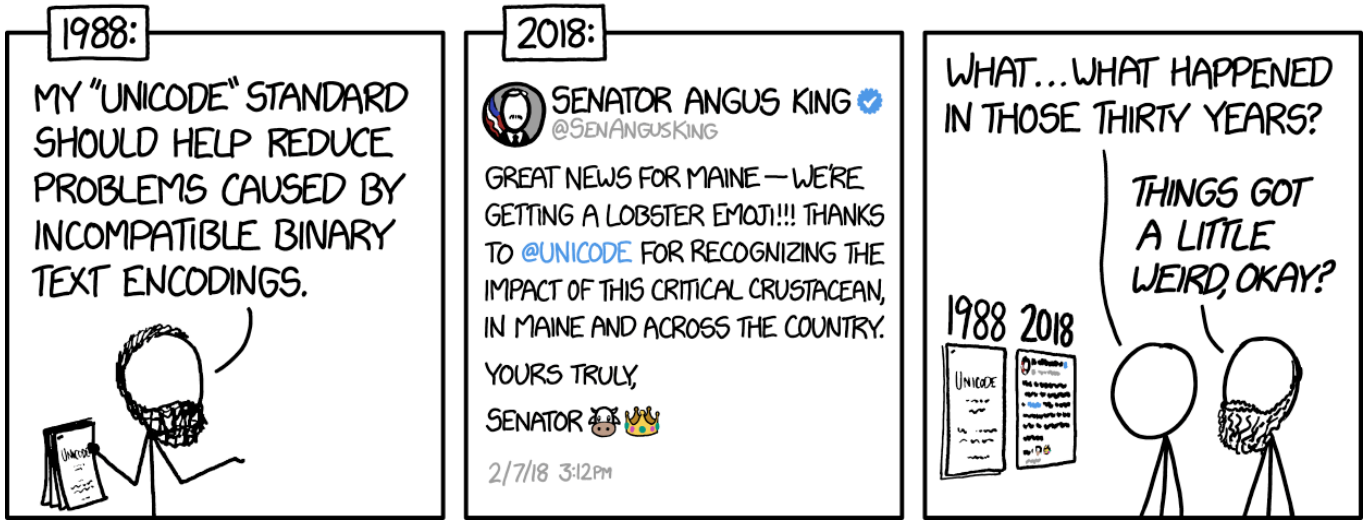
Other languages?

Diacritics?

Modern Unicode

When C was designed, characters fit in 8 (really 7) bits, so C's chars are 8 bits long.

When Java was designed, Unicode fit in 16 bits, so Java's chars are 16 bits. Then …



https://xkcd.com/1953/

Result: modern systems use *variable length* (UTF-8/16/32) encoding for Unicode.

In C, this is supported not in the language but via libraries

Agenda

Interfaces, implementations and design decisions for characters

Interfaces and implementations for the human reader

Another difference from Java

- Variable declarations in C89

`Recall: upper1` Program Specification

Read all chars from stdin

Capitalize the first letter of each 'word:'

- series of alphabetical characters terminated by non-alphabetical characters

Write result to stdout

upper1 Version 3

```
#include <stdio.h>
#include <ctype.h>
enum Statetype {NOT_INWORD, INWORD};

enum Statetype handleNotInwordState(int c)
{
   enum Statetype state;
   if (isalpha(c)) {
      putchar(toupper(c));
      state = INWORD;
   } else {
      putchar(c);
      state = NOT_INWORD;
   }
   return state;
}

enum Statetype handleInwordState(int c)
{
   enum Statetype state;
   if (!isalpha(c)) {
      putchar(c);
      state = NOT_INWORD;
   } else {
      putchar(c);
      state = INWORD;
   }
   return state;
}
```

```
int main(void)
{
   int c;
   enum Statetype state = NOT_INWORD;
   while ((c = getchar()) != EOF) {
      switch (state) {
         case NORMAL:
            state = handleNoNotInwordState(c);
            break;
         case INWORD:
            state = handleInwordState(c);
            break;
      }
   }
   return 0;
}
```

That's an A-, at best.
No comments!

`Upper1:` Improving the Interface

Problem:

- The program works, but…
- It's too hard for the human reader

Solution:

- Add function-level comments as part of the interface

Function Comments: A Key Part of Interfaces

Function comment should describe

what the function does (from the caller's viewpoint)

- Data coming into the function
  - Parameters, input streams
- What it does to those data (at a high level)
- Data going out from the function
  - Return value, output streams, call-by-reference parameters

Function comment should **not** describe

how the function works

Function Comment Examples

Bad main() function comment: Describes how the function works

```
Read a character from stdin using getchar.
Depending upon the current DFA state, pass the
character to an appropriate state-handling
function. The value returned by the state-
handling function is the next DFA state. Repeat
until end-of-file. Return 0.
```

Good main() comment: Describes what the function does (from caller's perspective)

```
Read text from stdin. Convert the first character
of each "word" to uppercase, where a word is a
Contiguous sequence of uppercase or lowercase
letters. Write the result to stdout. Return 0.
```

Upper1: Other Interface Comments

```
/* defines constants representing each state in the DFA */
enum Statetype {NOT_INWORD, INWORD};
```

```
/* Implement the NOT_INWORD state of the DFA. c is the current DFA
   character. Write c's uppercase equivalent, if it has one, or
   otherwise c itself, to stdout. Return the next state specified
   by the DFA. */

enum Statetype handleNotInwordState(int c) {
```

```
/* Implement the INWORD state of the DFA. c is the current
   DFA character. Write c to stdout. Return the next DFA state. */

enum Statetype handleInwordState(int c) {
```

```
/* Read text from stdin. Convert the first character of each
   "word" to uppercase, where a word is a sequence of
   letters. Write the result to stdout. Return 0. */

int main(void) {
   /* Use a DFA approach. state is the current DFA state. */
   enum Statetype state = NOT_INWORD;
```

Other Good Things for the Human Reader

- Comments in the implementation
- Readable code
  - Often a tradeoff with efficiency and “showing coding prowess”
  - Lack of language support for other data types can make it worse

Other Good Things for the Human Reader

Q: There are several ways to structure a loop – which is best?

A.
```
for (c = getchar(); c != EOF; c = getchar())
   putchar(toupper(c));
```

B.
```
while ((c = getchar()) != EOF)
   putchar(toupper(c));
```

C.
```
for (;;)
{  c = getchar();
   if (c == EOF)
      break;
   putchar(toupper(c));
}
```

D.
```
c = getchar();
while (c != EOF)
{putchar(toupper(c));
   c = getchar();
}
```

Other Good Things for the Human Reader

- Comments in the implementation
- Readable code
  - Often a tradeoff with efficiency and “showing coding prowess”
  - Lack of language support for other data types can make it worse

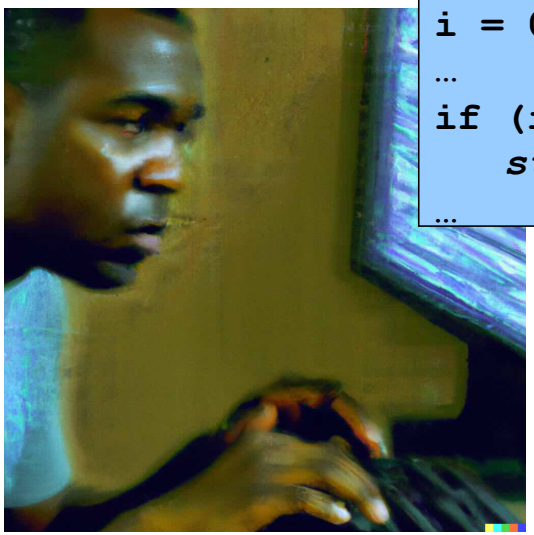
Unlike Java, C has No Boolean (Logical) Data Type

- Represent logical data using type char or int
  - Or any primitive type! 😱
- Conventions:
  - Statements (if, while, etc.) use 0 ⇒ FALSE, ≠0 ⇒ TRUE
  - Relational operators (<, >, etc.) and logical operators (!, &&, ||) produce the result 0 or 1
- Fine, but would have been nice to have a Boolean type
  - Note: the character literal ‘t’ is TRUE (nonzero int value), but ‘f’ is also TRUE (nonzero int)
  - Note: 0 and 1 are not assured values for FALSE AND TRUE. Could #define TRUE 6.023e23 …

Issues with Lack of Logical Data Type

Imagine this type of code in Java code and in C.
What happens in each case?

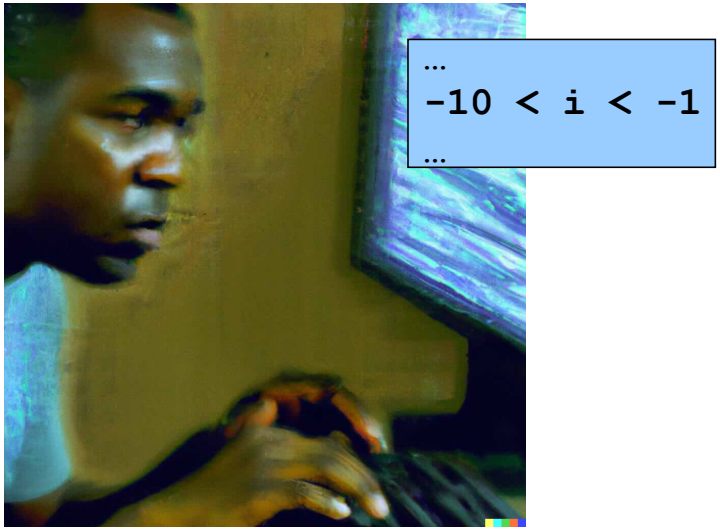
```
…
int i;
…
i = 0;
…
if (i = 5)
   statement1;
…
```



DALL·E 2
prompt: impressionist painting of a computer programmer with a lack of sleep debugging late at night

Sample Exam Question (Spring 2016, Exam 1)

Indicate what value this expression evaluates to:



DALL·E 2
prompt: impressionist painting of a computer programmer with a lack of sleep debugging late at night

Doing Logic with Integers

Using integers to represent logical data permits shortcuts

```
…
int i;
…
if (i)  /* same as (i != 0) */
   statement1;
else
   statement2;
…
```

But it also permits really bad code…

```
i = (1 != 2) + (3 > 4);
```

Agenda

Interfaces, implementations and design decisions for characters

Interfaces and implementations for the human reader

Another difference from Java

- Variable declarations in C89

Declaring Variables

Like Java, C requires variable declarations (some languages don't: awk, bash)

Declaring variables requires more from the programmer

- Extra verbiage
- Thinking ahead about how the variable will be used

But it has many benefits

- Allows compiler to check "spelling:" catch typos and type mismatches
- Allows compiler to allocate memory more efficiently
- Declaring the types of variables produces fewer surprises at runtime

Declaring Variables

Declaration statement specifies type of variable (and other attributes too)

Examples:

```
int i;
int i, j;
int i = 5;
const int i = 5; /* value of i cannot change */
static int i; /* covered later in course */
extern int i; /* covered later in course */
```

- Can have multiple declarations per statement. But can be risky …
- Unlike in Java, can use a local variable before assigning a value to it
  - But value is undefined

Declaring Variables (contd.)

- Unlike Java (and later versions of C), declaration statements in C89 must appear before any other kind of statement in any compound statement.
  - Note this **doesn't** mean that all declarations must be at the top of a *function* exclusively. Can be at start of a block

```
{
   int i;

   /* Non-declaration
      statements, even if
      they don't use j */

   int j;
}
```

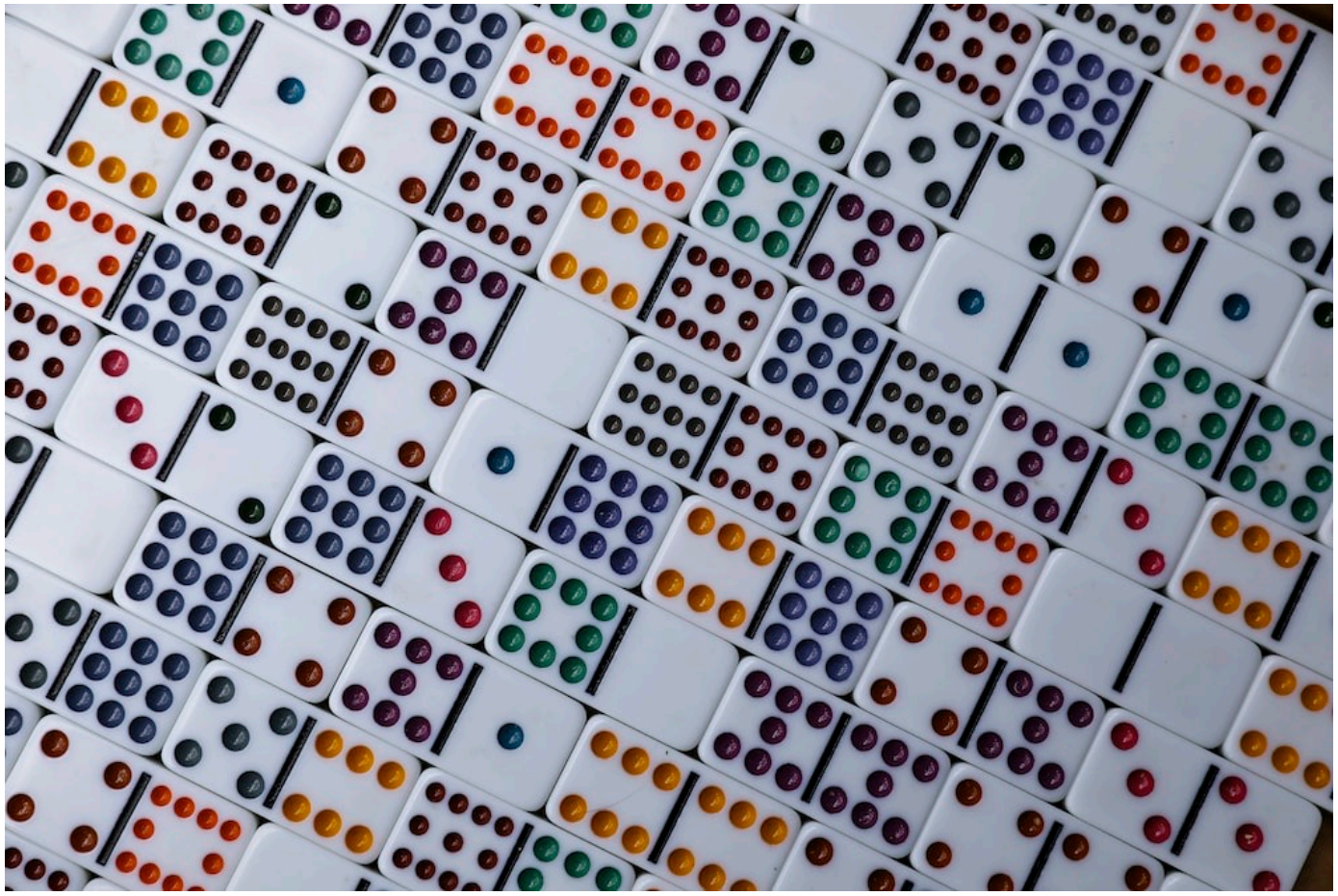
Illegal in C89

```
{
   int i;
   int j;

   /* Non-declaration
   statements that use
   i and/or j. */
}
```

Legal in C89

Next time ... Numbers



Mick Haupt