CSE 3100: Systems Programming

Lecture 2: Expressions and Basic Data Types in C

Department of Computer Science and Engineering

University of Connecticut

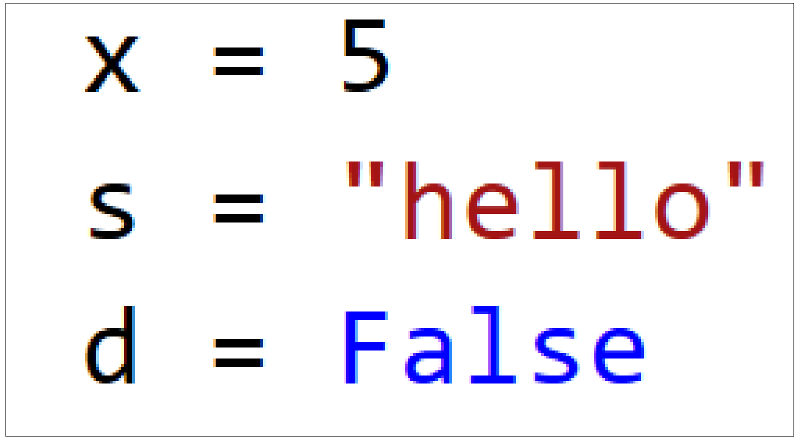
Lecture Overview

1. Static and Dynamic Typing

2. Expressions and Operators

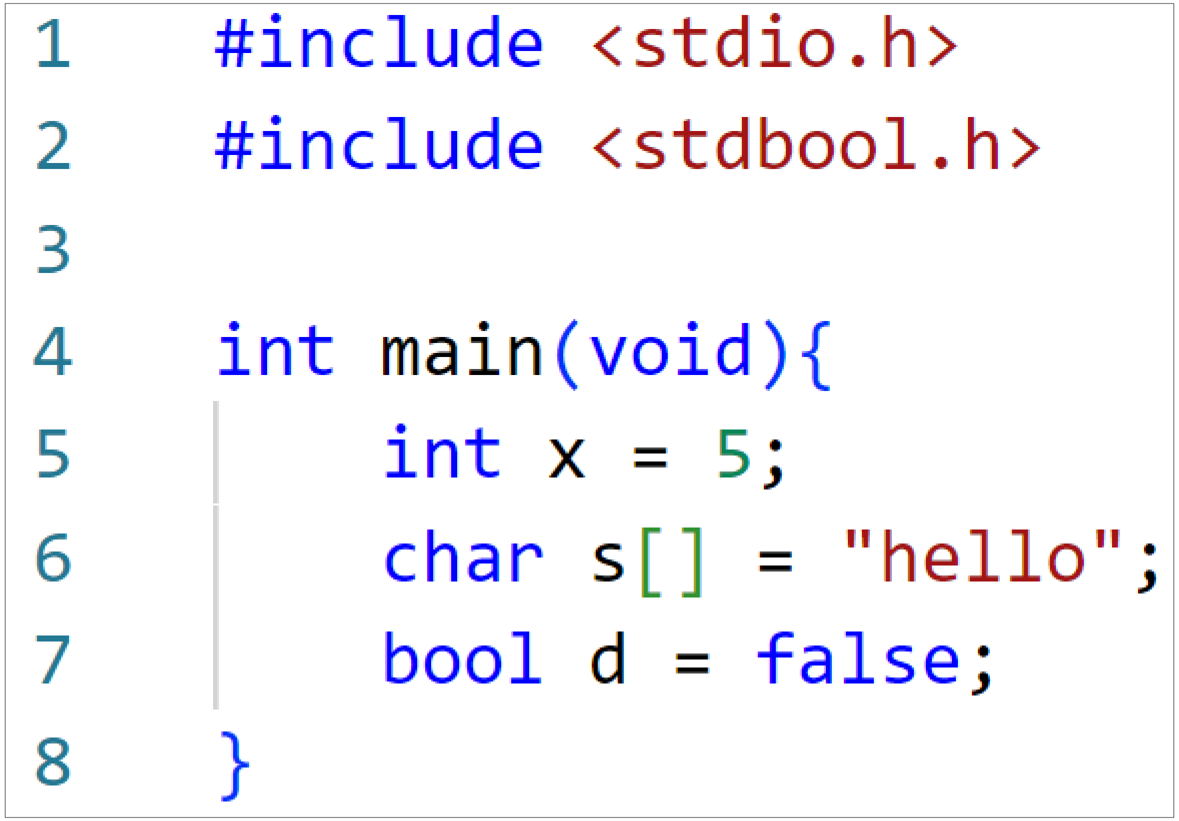
3. Memory for Data Types and Characters

Recall: What did an expression look like in Python?



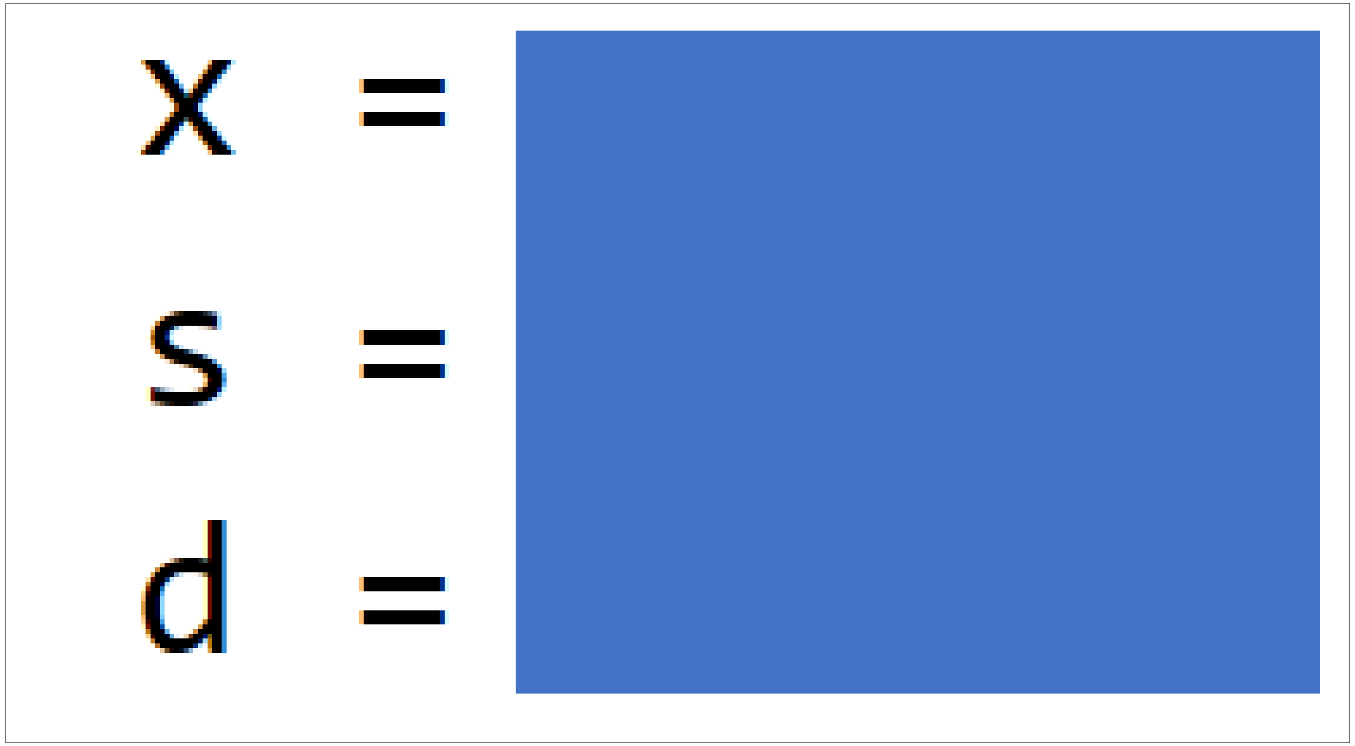
```
x = 5
s = "hello"
d = False
```

What does an expression look like in C?



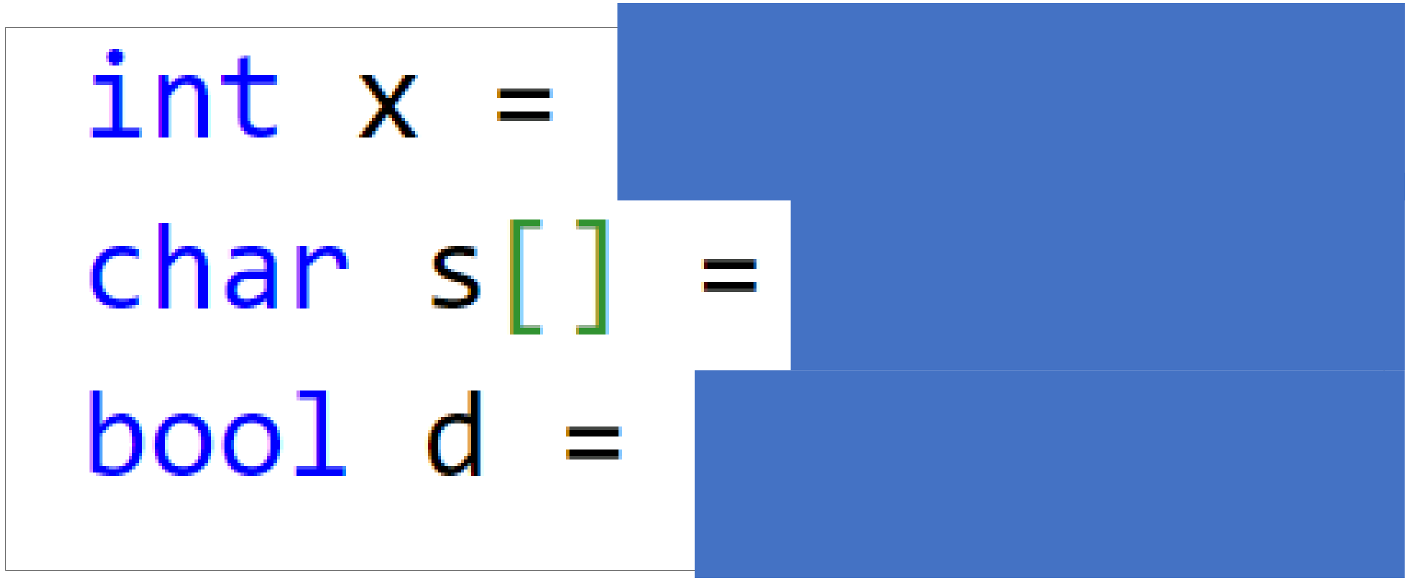
```c
#include <stdio.h>
#include <stdbool.h>

int main(void){
    int x = 5;
    char s[] = "hello";
    bool d = false;
}
```

Tough Question Time: *In Python can you tell if x, s and d are the same data type?*



```
x =

s =

d =
```

What about in C?



```
int x = [redacted]
char s[] =
bool d =
```

Typing for **expressions**

- Languages like C are statically typed.
- Statically typed programming languages do type checking at *compile-time*.

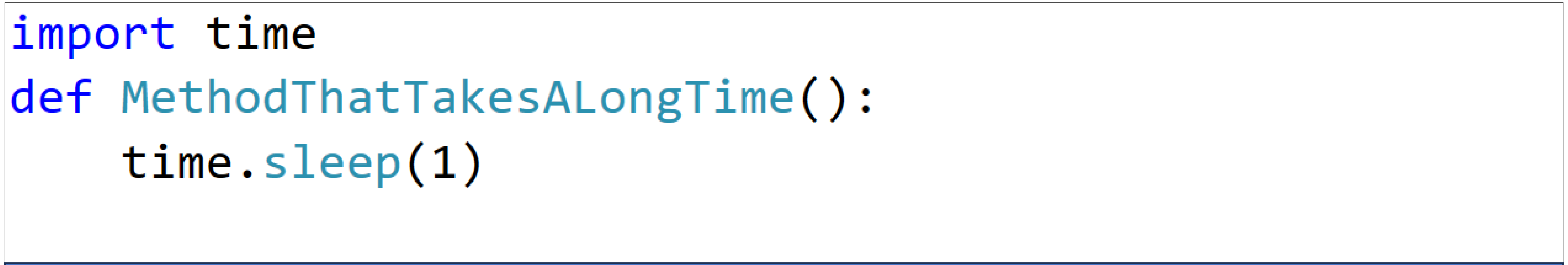
- Languages like Python are dynamically typed.
- Dynamically typed programming languages do type checking at *run-time*.

Why do we care?

Link: https://stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages

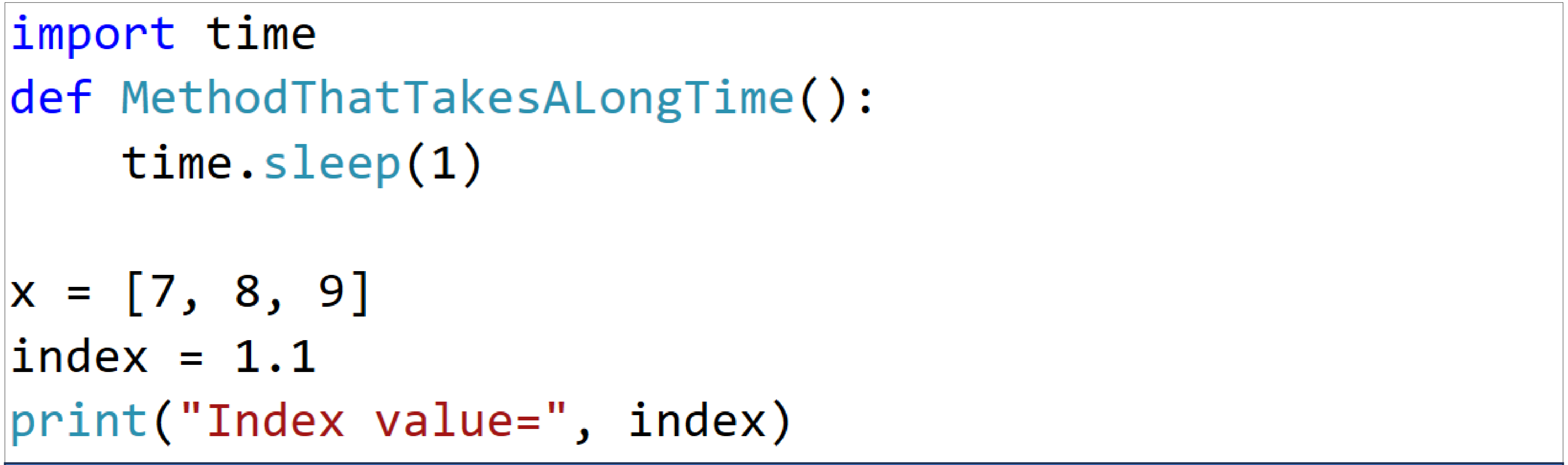
What is an advantage of a statically typed language?

We can illustrate the advantage with a buggy code…



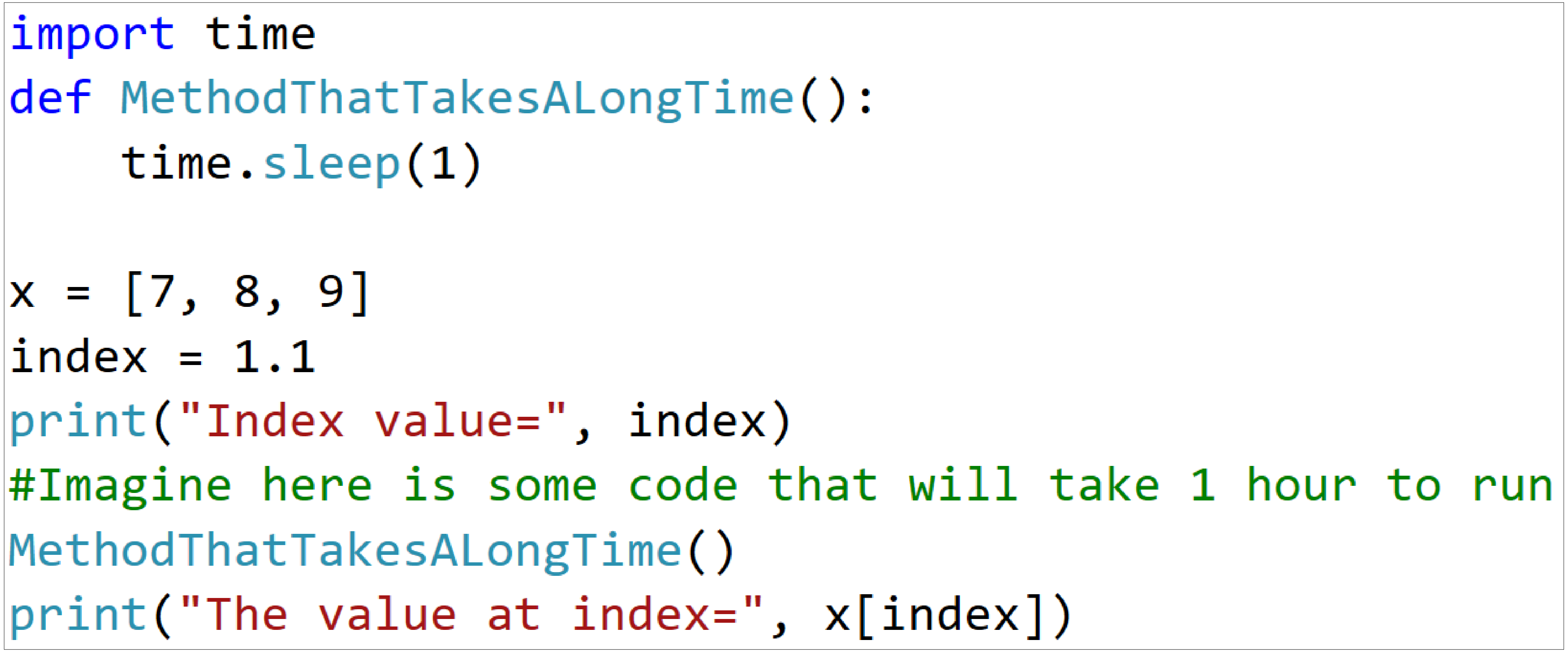
```python
import time
def MethodThatTakesALongTime():
    time.sleep(1)
```

What is an advantage of a statically typed language?



```python
import time
def MethodThatTakesALongTime():
    time.sleep(1)

x = [7, 8, 9]
index = 1.1
print("Index value=", index)
```

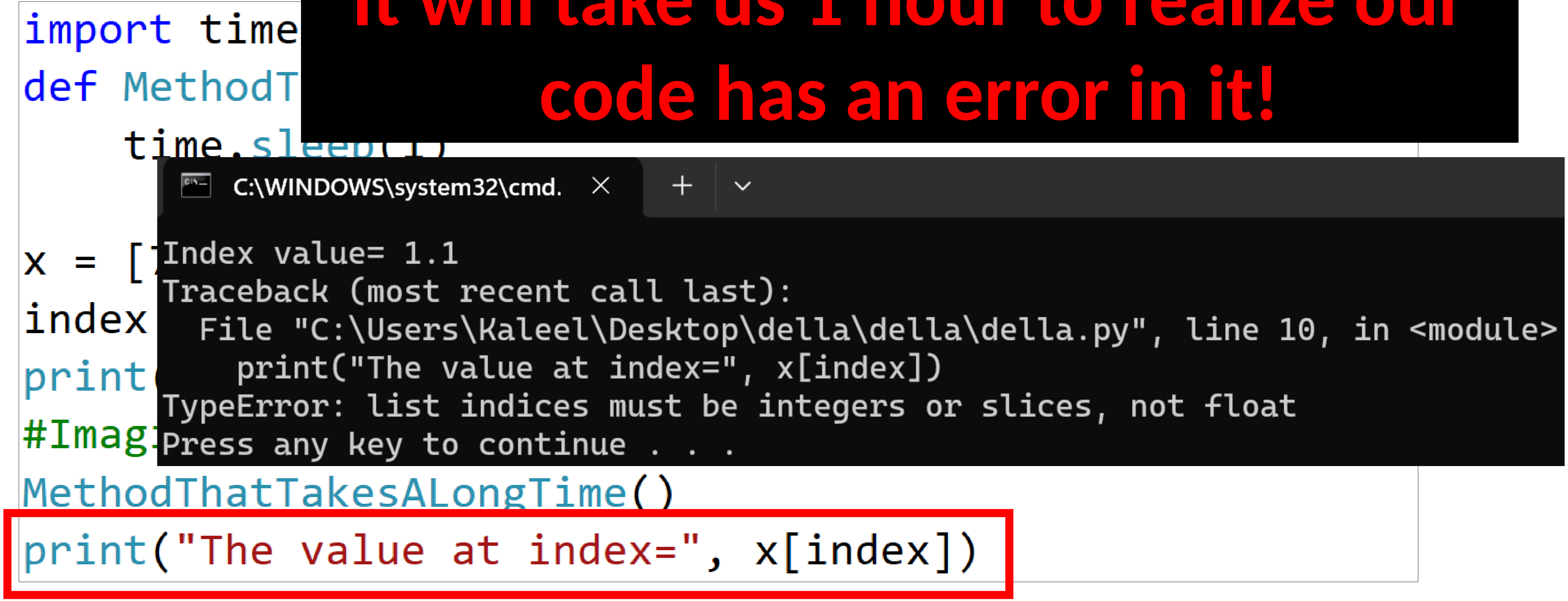
Where is the error in our code?



```python
import time
def MethodThatTakesALongTime():
    time.sleep(1)

x = [7, 8, 9]
index = 1.1
print("Index value=", index)
#Imagine here is some code that will take 1 hour to run
MethodThatTakesALongTime()
print("The value at index=", x[index])
```

Where is the error in our code?

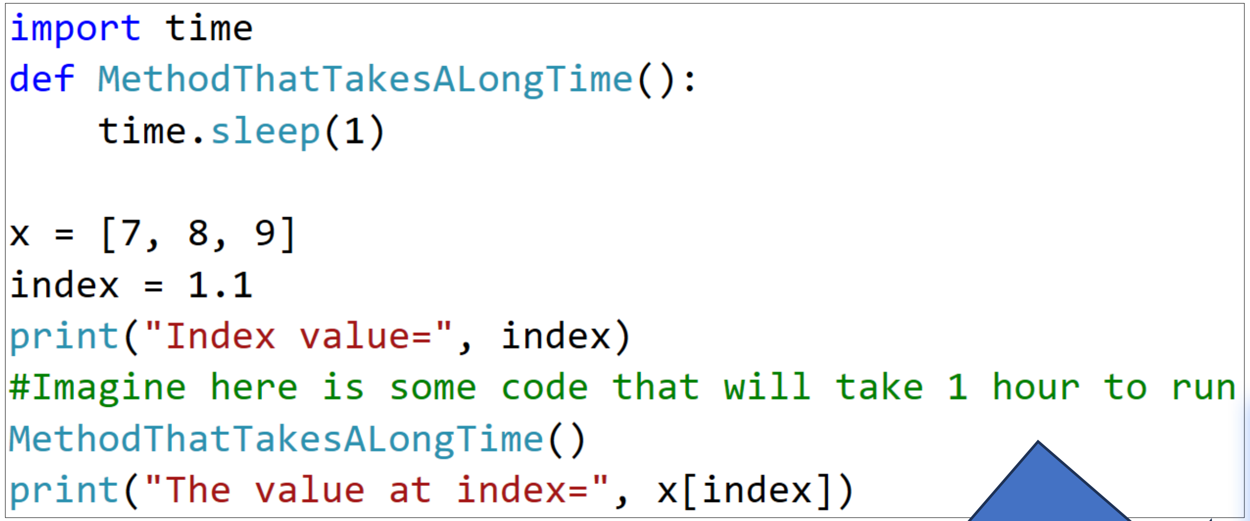
It will take us 1 hour to realize our code has an error in it!



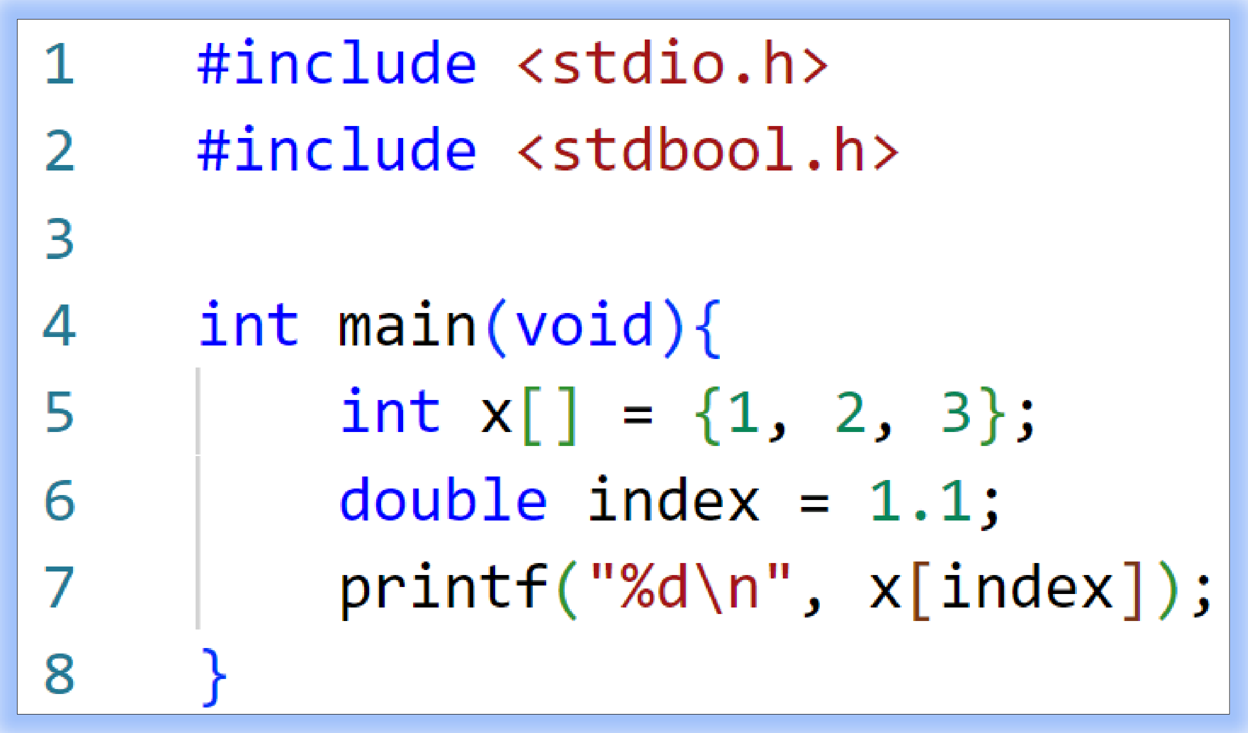
```
import time
def MethodT
    time.sleep(1)

x = [
index
print(
#Imag
MethodThatTakesALongTime()
print("The value at index=", x[index])
```

```
Index value= 1.1
Traceback (most recent call last):
  File "C:\Users\Kaleel\Desktop\della\della\della.py", line 10, in <module>
    print("The value at index=", x[index])
TypeError: list indices must be integers or slices, not float
Press any key to continue . . .
```

What would happen if we try the same code in C?

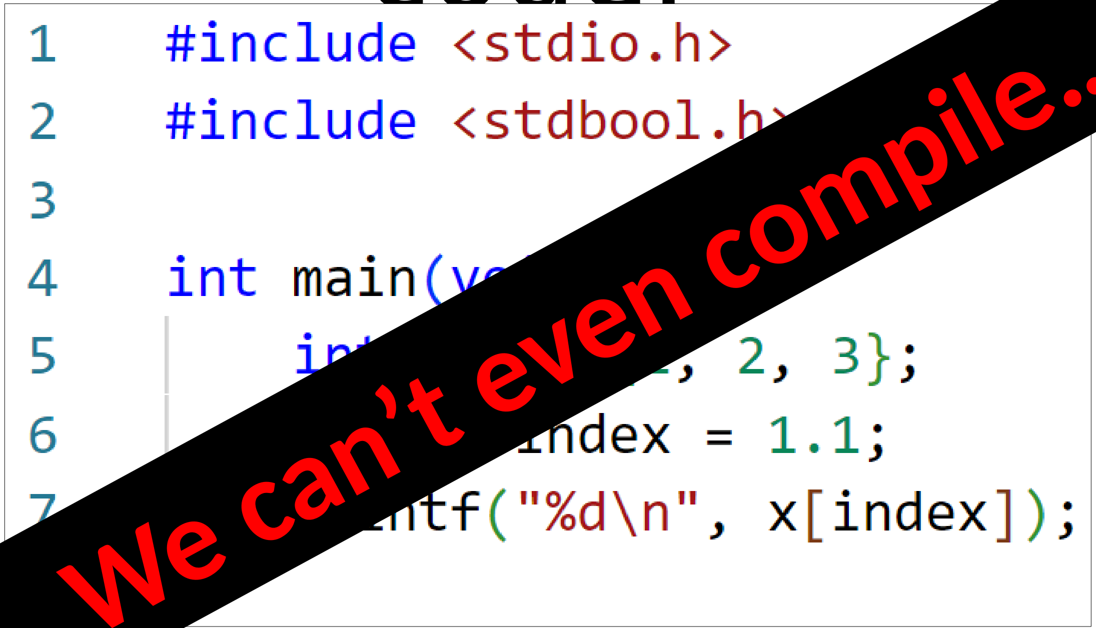


```python
import time
def MethodThatTakesALongTime():
    time.sleep(1)

x = [7, 8, 9]
index = 1.1
print("Index value=", index)
#Imagine here is some code that will take 1 hour to run
MethodThatTakesALongTime()
print("The value at index=", x[index])
```



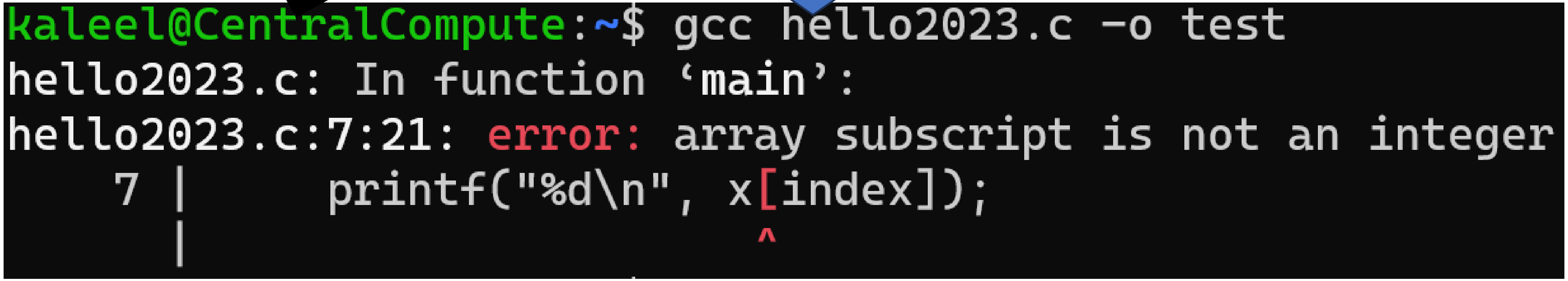
```c
#include <stdio.h>
#include <stdbool.h>

int main(void){
    int x[] = {1, 2, 3};
    double index = 1.1;
    printf("%d\n", x[index]);
}
```

Trying to compile wrongly typed C code:



```
1    #include <stdio.h>
2    #include <stdbool.h
3
4    int main(vo
5        in        , 2, 3};
6             index = 1.1;
7          ntf("%d\n", x[index]);
```

We can’t even compile…



```
kaleel@CentralCompute:~$ gcc hello2023.c -o test
hello2023.c: In function ‘main’:
hello2023.c:7:21: error: array subscript is not an integer
    7 |     printf("%d\n", x[index]);
      |                     ^
```

Statically Typed Languages

- It can clearly be seen that a lot of error checking can be done by the compiler BEFORE the code ever runs for statically typed languages.
- Are there any disadvantage to having a statically typed language?

Programmer hands after 1 day of typing out data types for all expressions:

Lecture Overview

~~1. Static and Dynamic Typing~~

2. Expressions and Operators

3. Memory for Data Types and Characters

Expressions in C

- All C expressions have a type
  - Constants have a type
  - Variables have a type
  - Function return values have a type
  - Every sub-expression of a larger expression has a type
- Adding a semicolon to an expression makes it a statement.

A Few Basic Data Types in C

- int
  - An integer
- char
  - A single byte that can store a character in ASCII
  - An 8-bit integer
- float
  - Floating point numbers

- Quick refresher: What is a bit and what is a byte?
  - A bit is simply a value 1 or 0.
  - A byte is 8 bits.

Constants (of basic types)

```
// Constants cannot be changed
// char
'a', 'b', '\n'

// integer (note that compiler stores them in binary)
200, -34
0x7fffFFFF // hex
07112      // octal

// floating point numbers
3.1415, -0.34, 1.3E20
```

Variables

- **All variables must be declared and initialized before use**
- Variable declarations specify the type and name
  - Compiler allocates memory based on type
  - Valid names consist of letters (case sensitive!), digits, and '_', but cannot start with digits
  - Multiple variables of the same type can be declared together
  - Variables can be initialized when declared (“variable definition”) or using separate assignments

Examples:

```
char c;
int  i, j, k = 1;
float f;
```

Operators

- Conventional arithmetic, bitwise, and logical operators

```
+   -   *   /   %
&   |   ~   ^   <<   >>
&&  ||  !
```

- Pre/post increment/decrement (as in Java, C++, etc.)

```
i++  ++i  j--  --j
c = i++;        //  c will be (i - 1)
c = ++i;        //  c will be the same as i
```

Something seems off…

The ++ and -- Operators in Practice

```
int i = 4;
int c = i++;
```

- First set i to 4.
- Second set c to same the value of i. So c will be 4.
- Third add one to i. Now i will be 5.

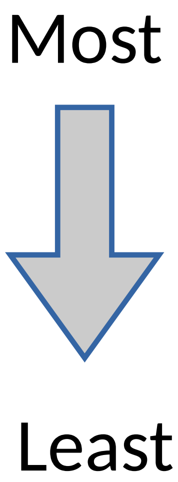
```
int i = 4;
int c = ++i;
```

- First set i to 4.
- Second increment i. So now i is 5.
- Third set c to the same value of i. So now c will be 5.

Precedence and associativity

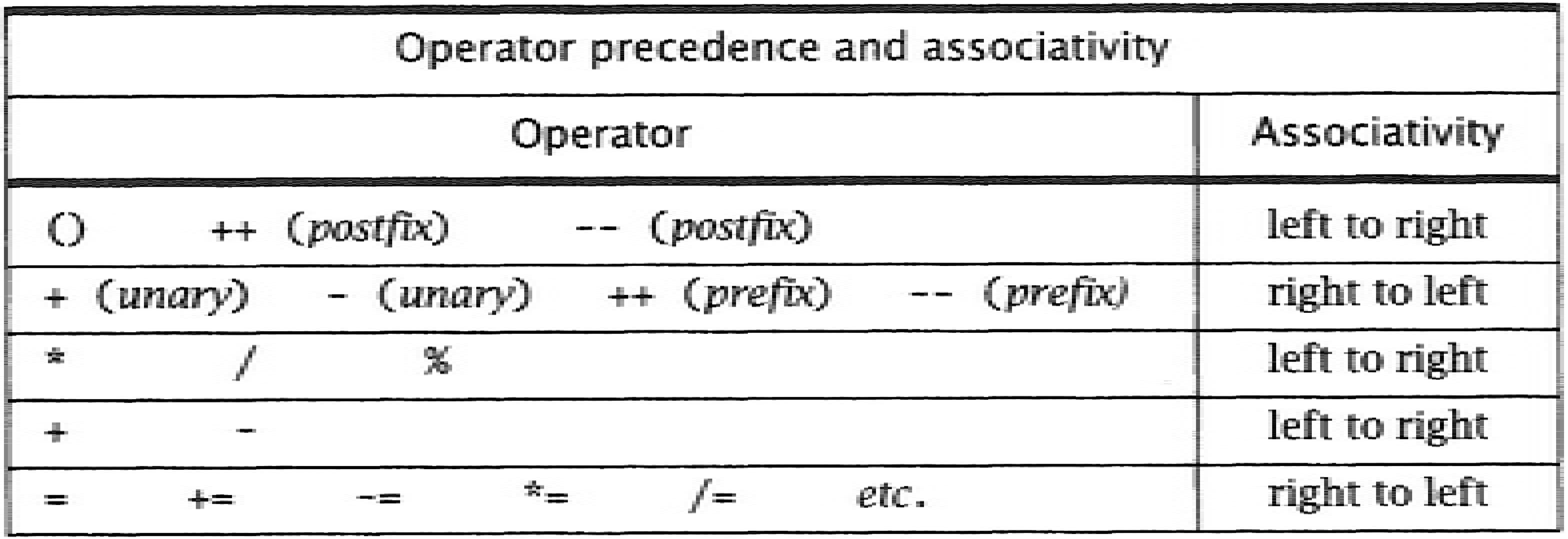
- Precedence determines which operation is done first
  - If operators have the same precedence, use associativity
  - Use parentheses

```
i + j * 10 – k / 20
(i + (j * 10)) – (k / 20)
```



Most

Least



Operator precedence and associativity

Operator						Associativity
()	++ *(postfix)*	-- *(postfix)*				left to right
+ *(unary)*	- *(unary)*	++ *(prefix)*	-- *(prefix)*			right to left
*	/	%				left to right
+	-					left to right
=	+=	-=	*=	/=	*etc.*	right to left

Assignment operators

- Assignment operator

  `LHS = Expression`

  - LHS (Left Hand Side) is something that can be ***written to*** (e.g, a variable)
  - LHS and Expression have “compatible” types
  - The value of Expression is assigned to LHS and becomes the value of the assignment operation
- Compound assignment operators (+=, *=, …)
- `var op= expr` ⟷ `var = var op expr`

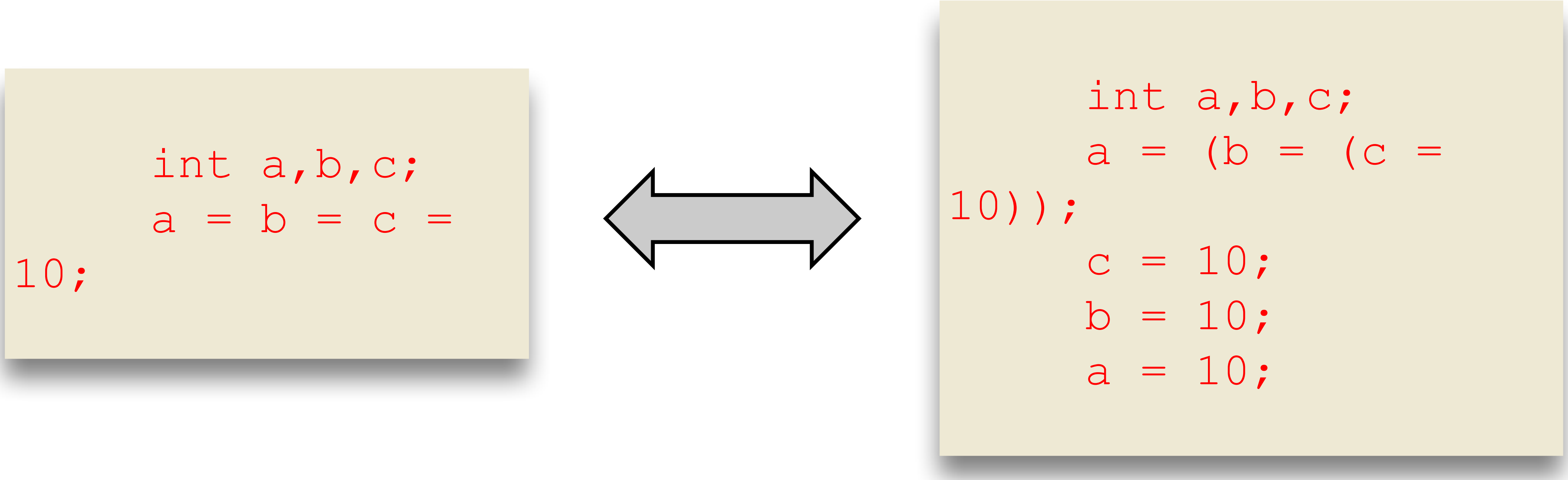
Compound Operators

Examples:

```
a = x + y;        b = c = d = 0;
i += 10;               // i = i + 10
j *= 5;                // j = j * 5
```

Assignments **ARE NOT** Statements

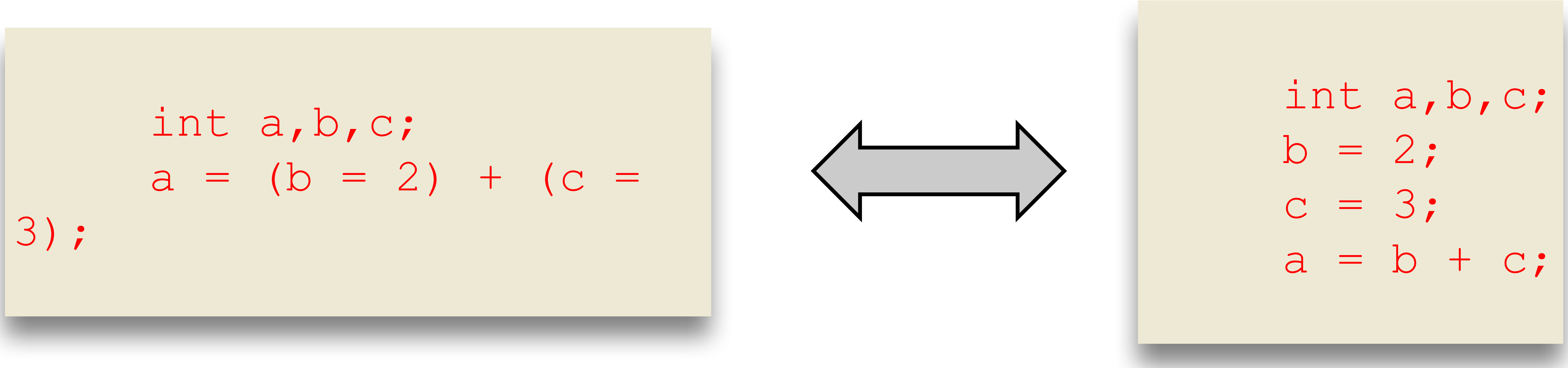
- Assignments are ***expressions*** and "=" is an operator
  - You can chain them!
  - You can use them inside larger expressions



```
     int a,b,c;
     a = b = c =
10;
```

```
     int a,b,c;
     a = (b = (c =
10));
     c = 10;
     b = 10;
     a = 10;
```

One more assignment example:

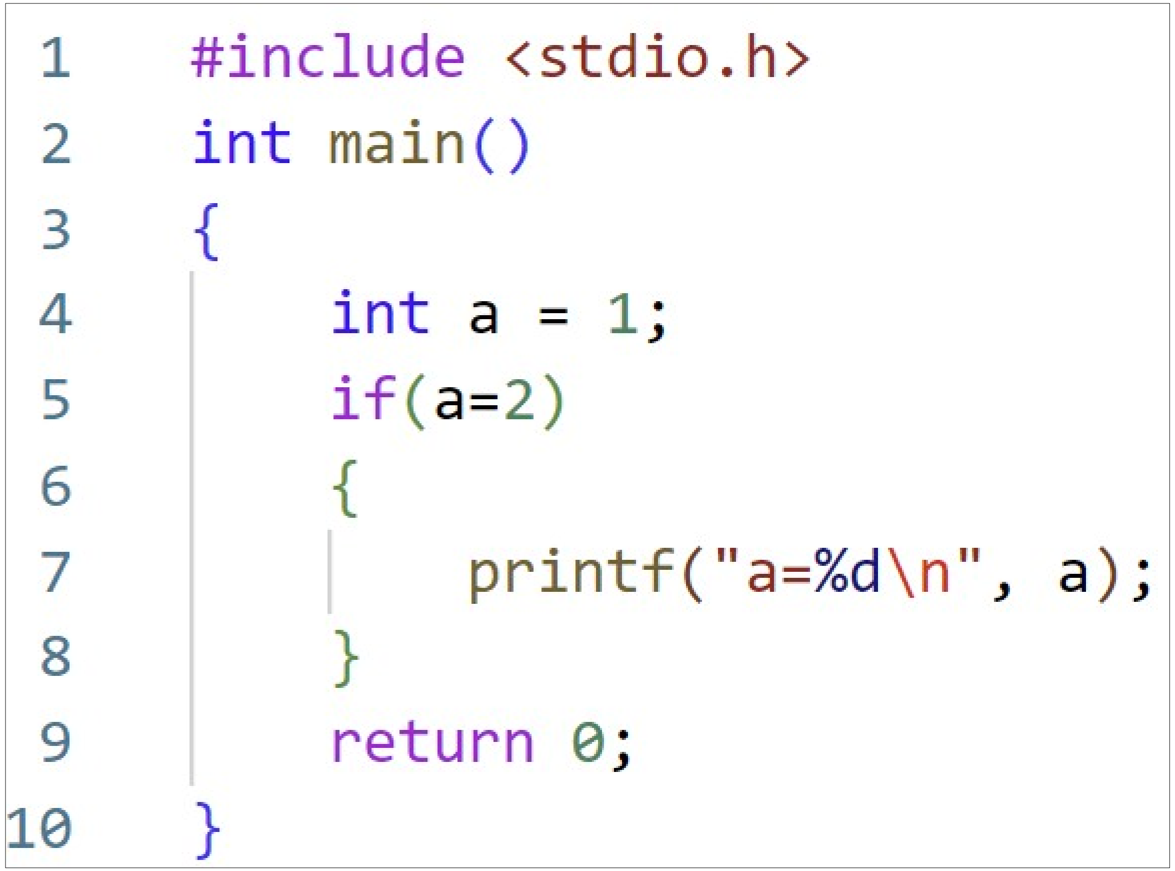


```
     int a,b,c;
     a = (b = 2) + (c =
3);
```

```
int a,b,c;
b = 2;
c = 3;
a = b + c;
```

Open question: would this be considered good practice?

Coding Assignment Example



```c
#include <stdio.h>
int main()
{
    int a = 1;
    if(a=2)
    {
        printf("a=%d\n", a);
    }
    return 0;
}
```

```
a=2
```

Lecture Overview

~~1. Static and Dynamic Typing~~

~~2. Expressions and Operators~~

3. Memory for Data Types and Characters

Integer Data Types

```
char
short int       🛍Ⓟ    short
int
long int        🛍Ⓟ    long
long long int   🛍Ⓟ    long long
```

And unsigned versions like “unsigned char”, “unsigned short”, etc.

- How many bytes does each take?
  - Depends on CPU architecture and compiler

Consider x86_64 (64-bit architecture)

size (in bits)	signed	unsigned
8	char -128 .. 127	unsigned char 0..255
16	short -32768..32767	unsigned short 0..65535
32	int $- 2^{31}$ .. $2^{31} - 1$	unsigned int $0..2^{32}-1$
64	long $- 2^{63}$ .. $2^{63} - 1$	unsigned long $0..2^{64}$ -1
64	long long $- 2^{63}$ .. $2^{63} - 1$	unsigned long long $0..2^{64}$ -1

Consider i386 (32-bit architecture)

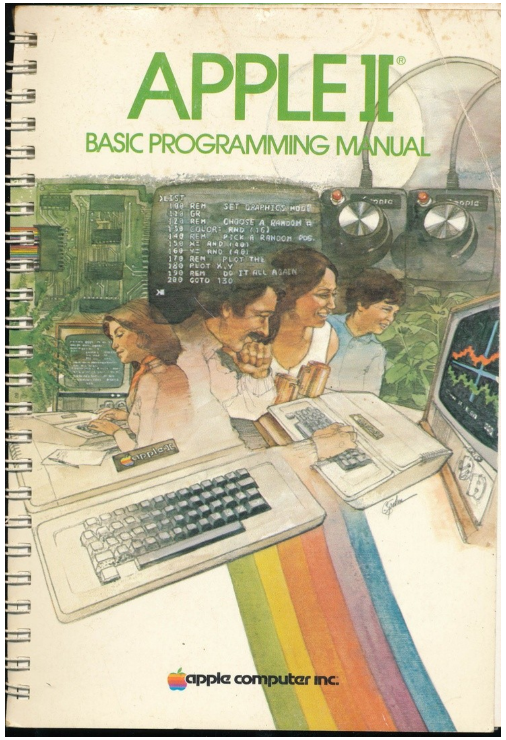
size (in bits)	signed	unsigned

8	`char`
	`-128 .. 127`
		`unsigned char`
		`0..255`

16	`short`
	`-32768..32767`
		`unsigned short`
		`0..65535`

32	`int` $- 2^{31} .. 2^{31} - 1$	`unsigned int` $0..2^{32}-1$
32	`long` $- 2^{31} .. 2^{31} - 1$	`unsigned long` $0..2^{32}-1$
64	`long long` $- 2^{63} .. 2^{63} - 1$	`unsigned long long` $0..2^{64} -1$

How to determine memory (space) requirements when running my program?



Answer: Read and memorize the hardware manual for every computer architecture design from 1980 onward.

How much space?

- How to determine the amount of space for some type?
- Operator sizeof gives the number of bytes needed for a type or a variable
  - You will need this later to dynamically allocate space!

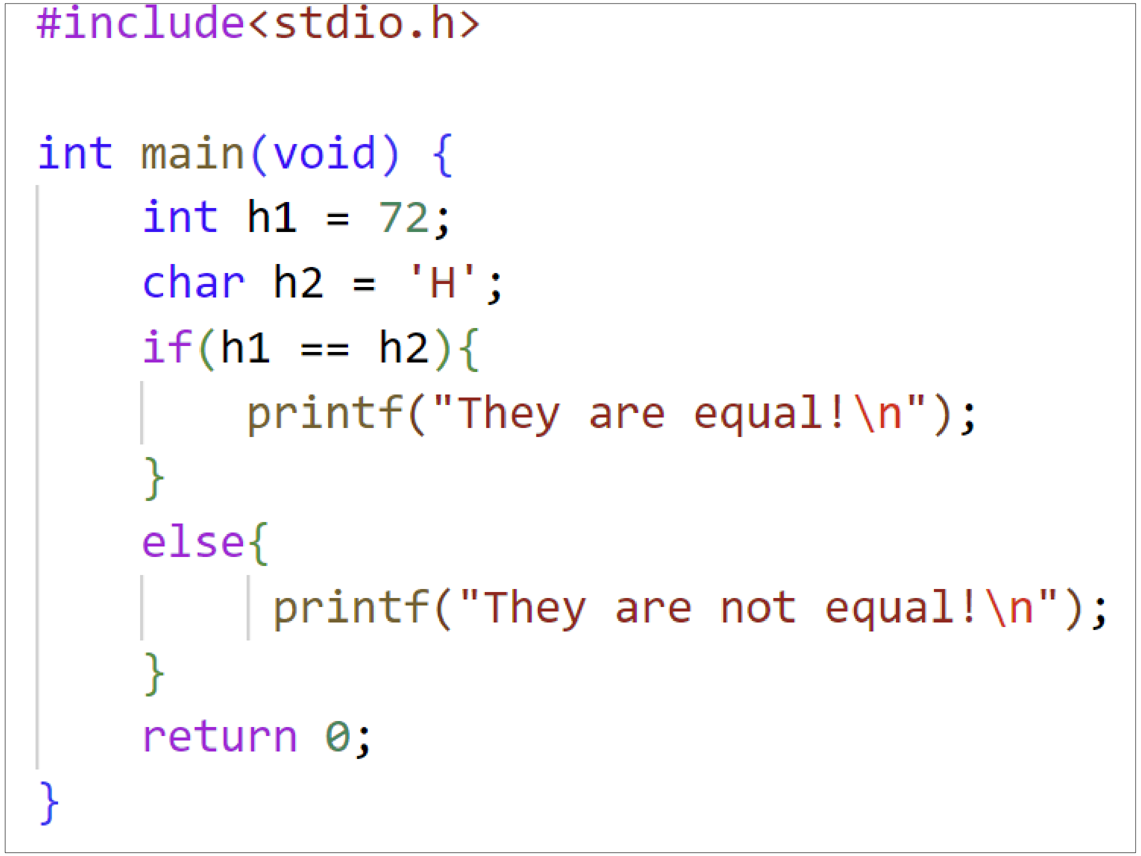
```
sizeof(T)
```

Examples:

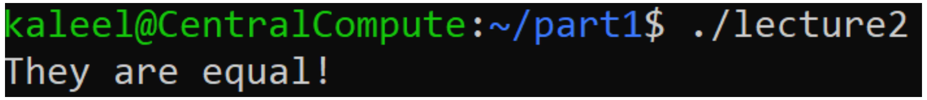
```
int i;
sizeof(int);  sizeof(i);
// 4 on the machines we use in this course
```

An Example Code to Check Equality

What will be printed?



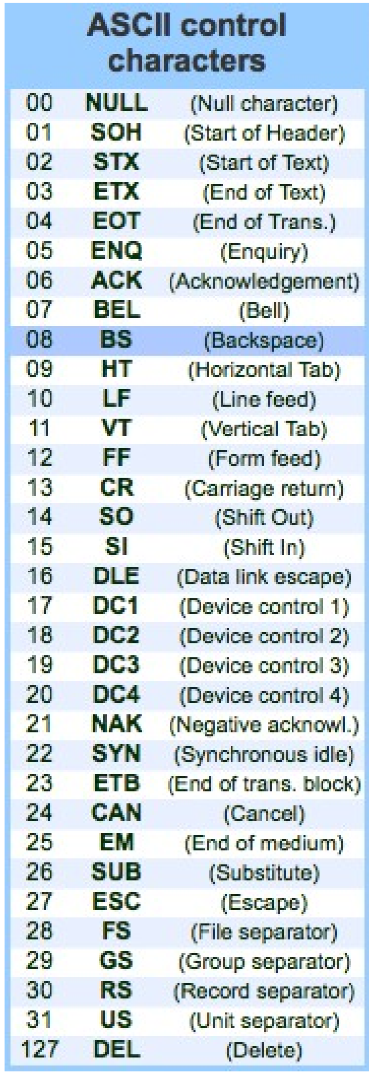
```c
#include<stdio.h>

int main(void) {
    int h1 = 72;
    char h2 = 'H';
    if(h1 == h2){
        printf("They are equal!\n");
    }
    else{
        printf("They are not equal!\n");
    }
    return 0;
}
```



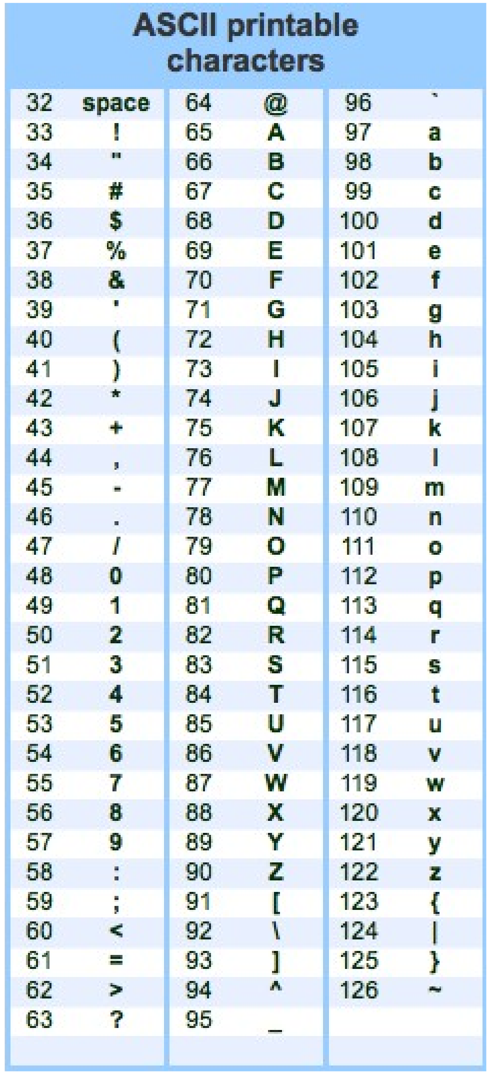
```
kaleel@CentralCompute:~/part1$ ./lecture2
They are equal!
```

Character (char) Data Type

- char has 8 bits (a byte)
- ASCII code
  - Characters are mapped to an integer in 0..127
  - An ASCII character can be stored in char
- Classes in ASCII
  - 0..31: “Control” character (aka, non-printable)
  - 48..57: Digits
  - 65..90: Upper case letters
  - 97..122: Lower case letters



ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)



ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

So…

- 65..90: Upper case letters
- The character ‘H’ is none other than …. 72

- Observe how…
    - ‘0’ through ‘9’ are consecutive!
    - ‘A’ through ‘Z’ are consecutive!
    - ‘a’ through ‘z’ are consecutive!

Want to see ASCII table in your terminal?

```
man ascii
```

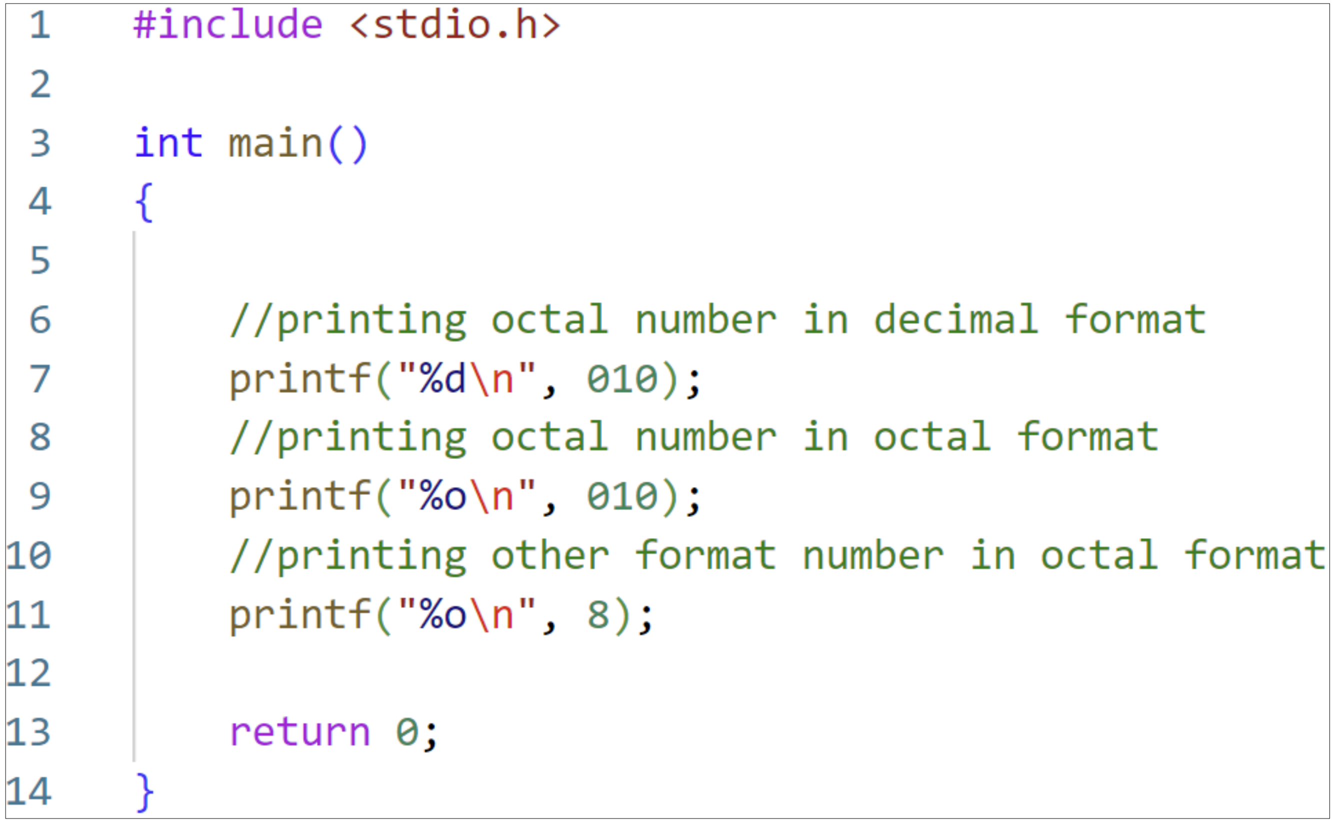
```
char ch = '8';
int  x = ch – '0';             // What is the value of x?
```

Non-printable characters

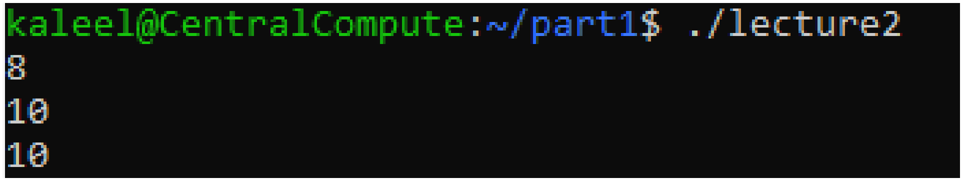
- These are sometimes useful
  - Showing the constant (literal)

`'\n'`	newline
`'\r'`	carriage-return
`'\f'`	form-feed
`'\t'`	tabulation
`'\b'`	backspace
`'\x7'`	audible bell (x indicates hexadecimal)
`'\07'`	audible bell (0 indicates octal)

Printing Code Example



```c
#include <stdio.h>

int main()
{

    //printing octal number in decimal format
    printf("%d\n", 010);
    //printing octal number in octal format
    printf("%o\n", 010);
    //printing other format number in octal format
    printf("%o\n", 8);

    return 0;
}
```



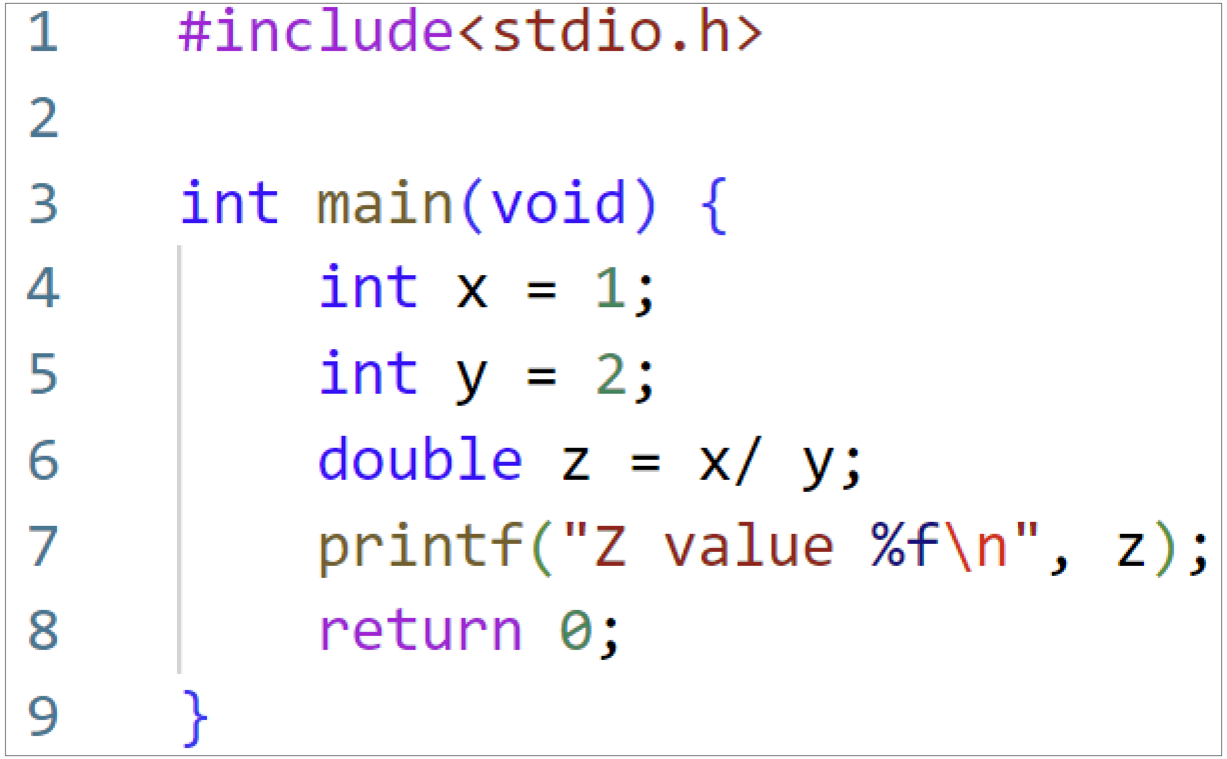
```
kaleel@CentralCompute:~/part1$ ./lecture2
8
10
10
```

Automatic Type Conversion

- When an operator has operands of different types, the operands are **automatically** converted to a common type by the compiler
  - In general, a lower rank operand is converted into the type of the higher rank one, where

  char < short < int < long < long long < float < double < long double
  - E.g., 1 gets converted to double before performing the addition in the expression 1 + 2.5
- Automatic conversion can also occur across assignments
  - The value of the expression on right hand side may be **widened** to the type of the LHS, e.g., double d = 1;
  - Or **narrowed** (possibly with information loss), e.g., int i = 2.5;
- Read the book for more details!

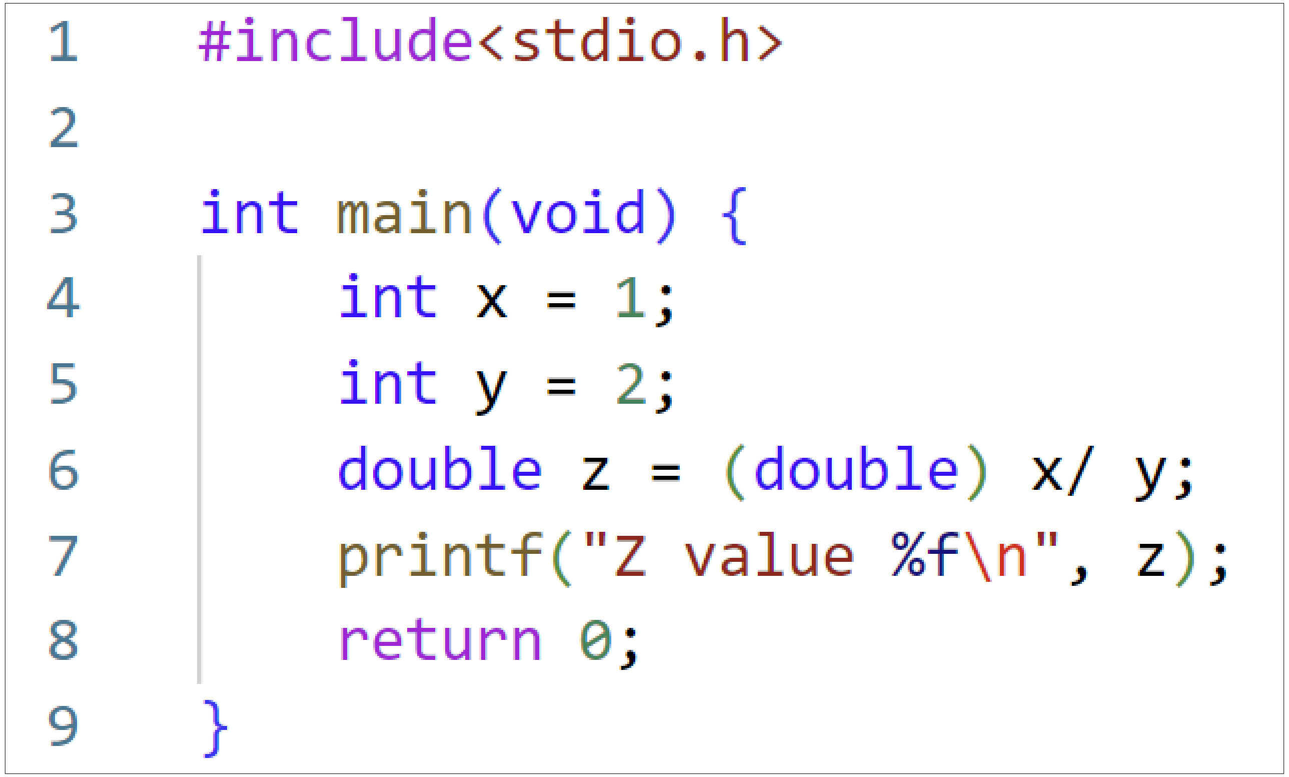
Type Conversion Example 1



```c
#include<stdio.h>

int main(void) {
    int x = 1;
    int y = 2;
    double z = x/ y;
    printf("Z value %f\n", z);
    return 0;
}
```

0

Type Conversion Example 2



```c
#include<stdio.h>

int main(void) {
    int x = 1;
    int y = 2;
    double z = (double) x/ y;
    printf("Z value %f\n", z);
    return 0;
}
```

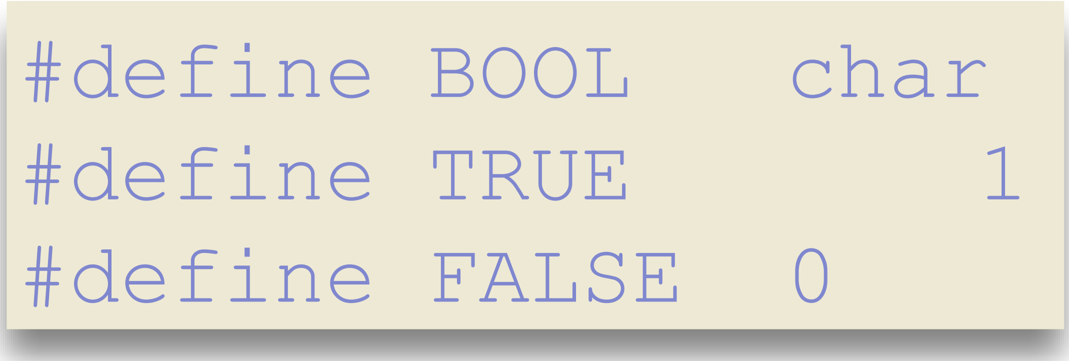
0.5

Would this work?

```
z = (double)(x / y)
```

What About Booleans?

- K&R and C89/C90 do not have a Boolean data type
  - 0 “means” FALSE and anything else “means” TRUE
  - Common to use int or char to store Boolean values and define convenience macros



```
#define BOOL    char
#define TRUE       1
#define FALSE   0
```

- C99 introduced _Bool
  - A variable of _Bool type is either 0 or 1

Integers of specific sizes (C99)

```
#include <stdint.h>
int8_t         // signed 8 bits integers
int16_t
int32_t
int64_t
uint8_t        // unsigned 8 bits integers
uint16_t
uint32_t
uint64_t
// Many projects have their own *standard* types
```

Strings

- String is not a basic type
- A string is an array of characters
  - Will be discussed more later

```
"Hello world!\n"
"Hello " "world!\n"   // Useful for long strings
"It's \"Mickey\"\n"   // Escape double quotations
"A bell \007\n"       // Can use ASCII code
```

Be careful mixing types

- Sometimes the results may not be as expected!
  - What is the size (in bits) of each operand?
  - Are your operands signed or unsigned ?

```
_Bool b1;
char  b2, b3;
int   i = 256; // 0x100

b1 = i;
b2 = i;
b3 = i != 0;
```

b1 is 1 because i is not 0
b2 is 0 because lowest 8 bits in i are 0
b3 is 1 because i is not 0

Do you want b2 or b3?

Figure Sources

1. https://i.imgflip.com/5wgm69.jpg
2. https://upload.wikimedia.org/wikipedia/commons/thumb/e/e0/SNice.svg/800px-SNice.svg.png
3. https://i.imgflip.com/1rflc0.jpg
4. https://st.depositphotos.com/1726977/1219/i/450/depositphotos_12193152-stock-photo-boney-skeleton-hands-working-a.jpg
5. https://sonder.com.au/wp-content/uploads/2020/10/quality-over-quantity-blog@2x.png
6. https://i.pinimg.com/736x/8b/f2/2a/8bf22a209224c73dd1618d6c12f2474d--basic-programming-apple-ii.jpg
7. https://humornama.com/wp-content/uploads/2020/12/You-Are-A