An introduction to C programming

EECS 348: Software Engineering

Fall 2023

Learning objectives

- Learn how to write and compile a C program
- Learn what C libraries are
- Understand the C variable types
- Understand some control statements

What is in a name?

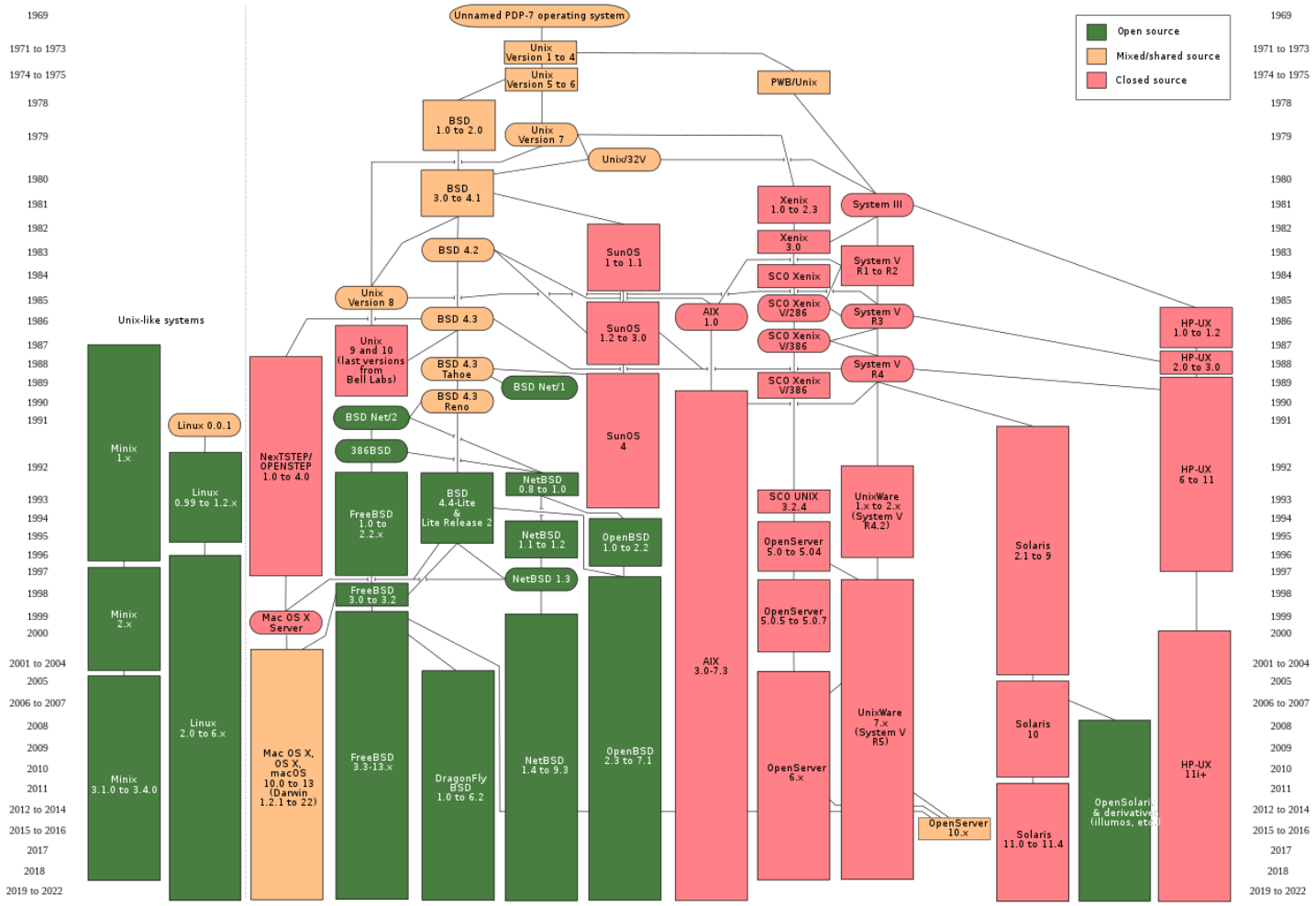
- MULTICS (Multiplexed Information and Computing Service) was an ambitious OS project
  - GE, Bell Labs, MIT
  - Supposed to be the most powerful OS
  - But also too complex
  - The project was not continued
- The birth of Unix (early 1970s)
  - Ken Thompson and Dennis Ritchie at Bell Labs
  - It was designed to be a simpler and more portable operating system than MULTICS

What is in a name?

- MULTICS vs Unix

Feature	MULTICS	Unix
Design philosophy	Complex and powerful	Simple and portable
Target audience	Wide range of users	Computer scientists and other technical users
Programming language	PL/I	C
Major features	Hierarchical file system, dynamic linking, virtual memory	Hierarchical file system, shell, pipes, redirection
Success	Limited adoption	Widely adopted

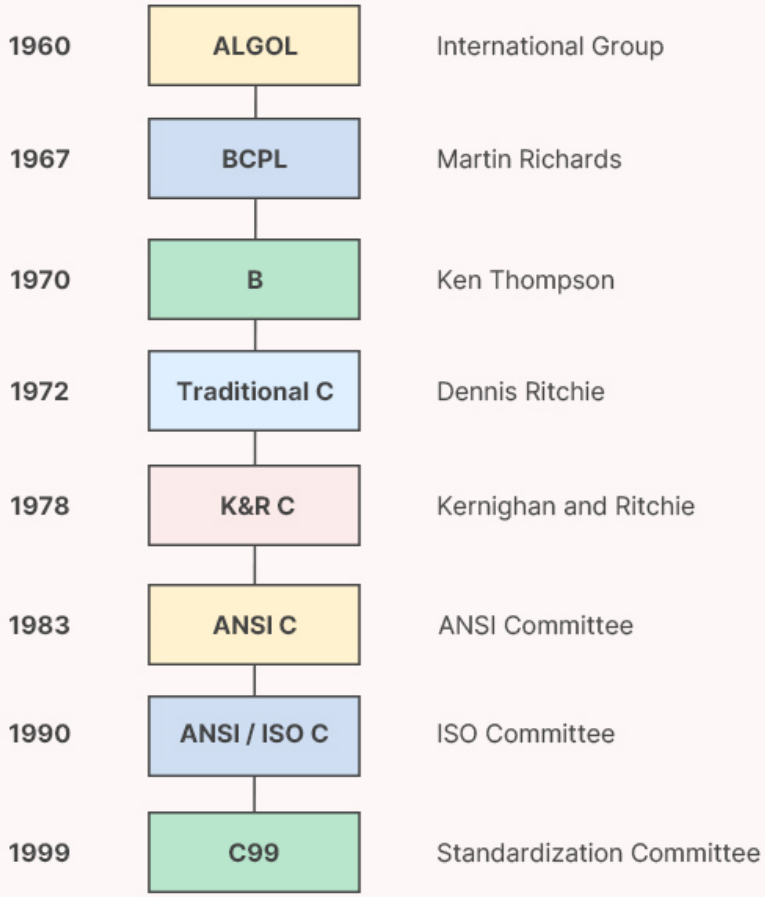
Unix history (Wikipedia)



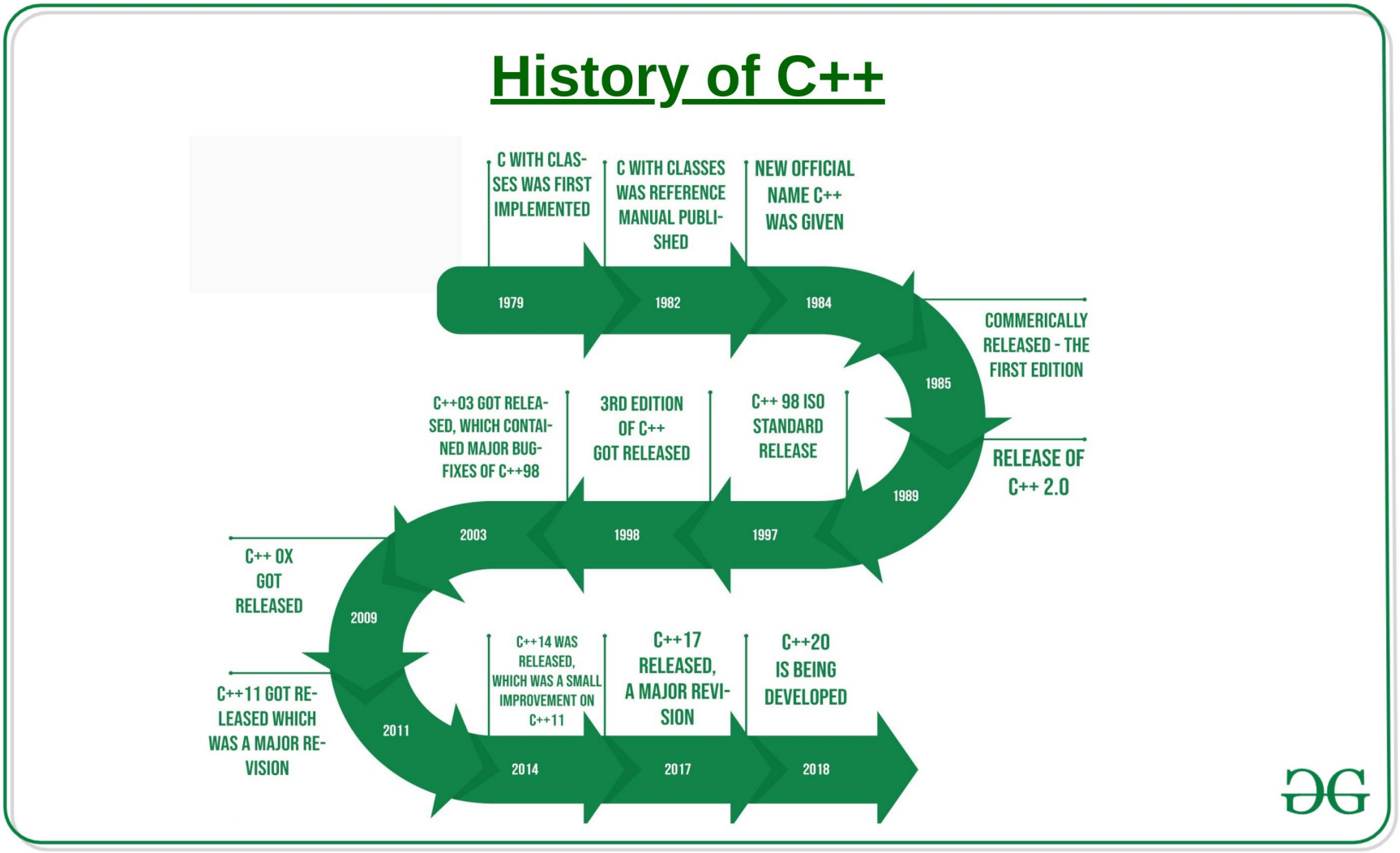
What is in a name?

- Language B
  - Derived from BCPL (Basic Combined Programming Language)
  - Created by Ken Thompson at the Bell Labs in 1960
  - A small and simple language, but it was difficult to use and had many limitations
- Language C
  - Dennis Ritchie, another Bell Labs researcher, joins Thompson to develop a new language that was power but easier to use
  - They called it C (because the previous one was called B)
  - A very popular language
- C++
  - The name C++ is a play on the word "C" and the word "++"
  - Developed Bjarne Stroustrup in 1979 (Ph.D. work)

History of C

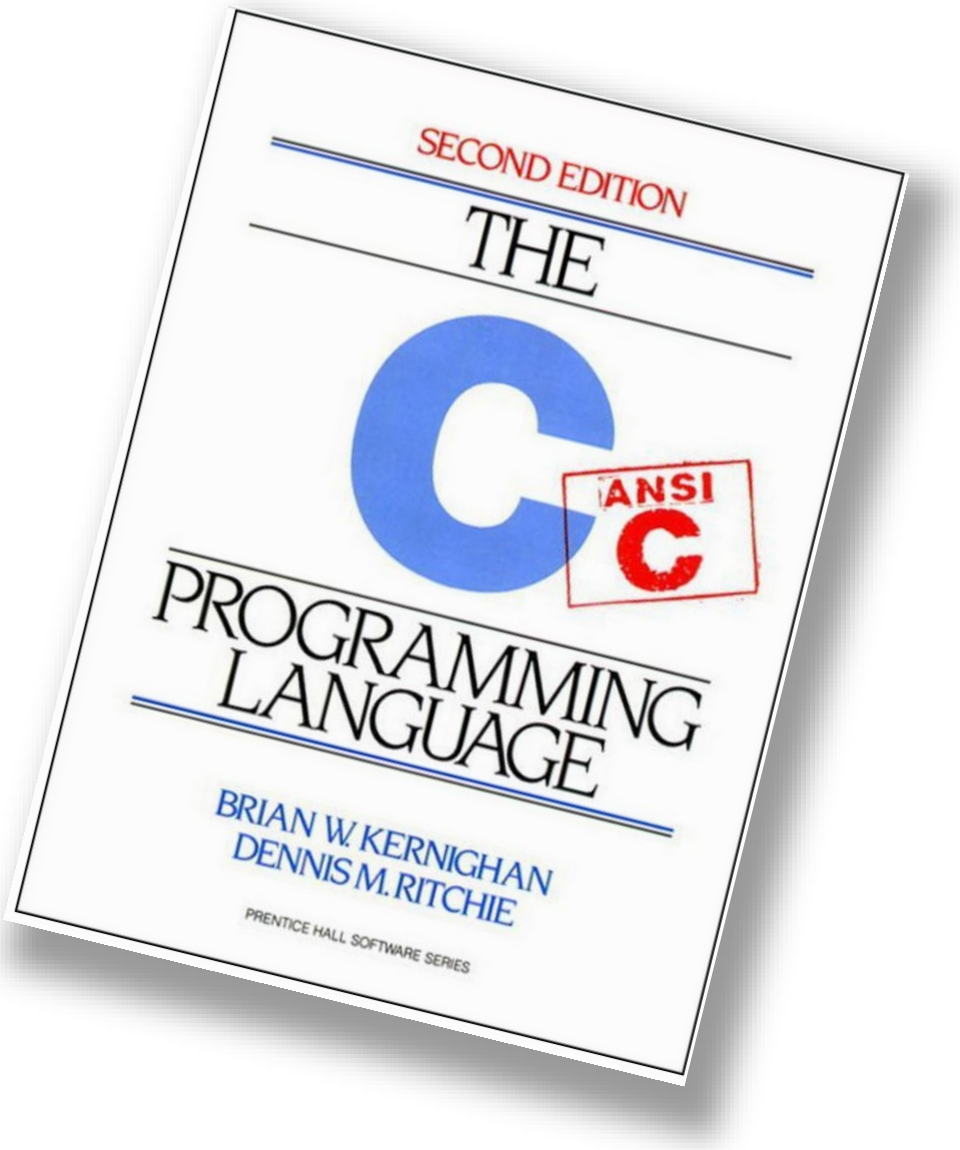

1960
ALGOL
International Group
1967
BCPL
Martin Richards
1970
B
Ken Thompson
1972
Traditional C
Dennis Ritchie
1978
K&R C
Kernighan and Ritchie
1983
ANSI C
ANSI Committee
1990
ANSI / ISO C
ISO Committee
1999
C99
Standardization Committee

History of C++


History of C++
C WITH CLASSES WAS FIRST IMPLEMENTED
1979
C WITH CLASSES WAS REFERENCE MANUAL PUBLISHED
1982
NEW OFFICIAL NAME C++ WAS GIVEN
1984
COMMERICALLY RELEASED - THE FIRST EDITION
1985
RELEASE OF C++ 2.0
1989
C++ 98 ISO STANDARD RELEASE
1997
3RD EDITION OF C++ GOT RELEASED
1998
C++03 GOT RELEASED, WHICH CONTAINED MAJOR BUGFIXES OF C++98
2003
C++ 0X GOT RELEASED
2009
C++11 GOT RELEASED WHICH WAS A MAJOR REVISION
2011
C++14 WAS RELEASED, WHICH WAS A SMALL IMPROVEMENT ON C++11
2014
C++17 RELEASED, A MAJOR REVISION
2017
C++20 IS BEING DEVELOPED
2018

The very classic C Book

- The *K&R C* book
- Millions of copies sold
- Translated to 25 languages



A sample program

```c
#include <stdio.h>


int main() {
    int year;

    printf("\n");
    printf("Enter a year: ");
    scanf("%d", &year);

    // leap year if perfectly divisible by 400

    if (year % 400 == 0) {
        printf("%d is a leap year.", year);
    }

    // not a leap year if divisible by 100
    // but not divisible by 400

    else if (year % 100 == 0) {
        printf("%d is not a leap year.", year);
    }

    // leap year if not divisible by 100
    // but divisible by 4

    else if (year % 4 == 0) {
        printf("%d is a leap year.", year);
    }

    // all other years are not leap years

    else {
        printf("%d is not a leap year.\n\n", year);
    }

    return 0;
}
```

Writing a C program

1. Write the code for a program (source code) using an editor such as vi or nano, save as file `my_pgm.c`

```
#include <studio.h>

int main ( ) {
    printf("Hello, world!\n");
}
```

Compiling a C program

2. Compile the program to convert from the *source code* to an "executable" or "binary" (or *object code*):

```
$ gcc -o my_pgm.exe my_pgm.c
```

3. If the compiler produces any errors, fix them and recompile

Executing a C program

2. Once there are now programming errors and you have a n executable code, run it:

```
$ my_pgm.exe
```

Hello, world!

Some common properties of C

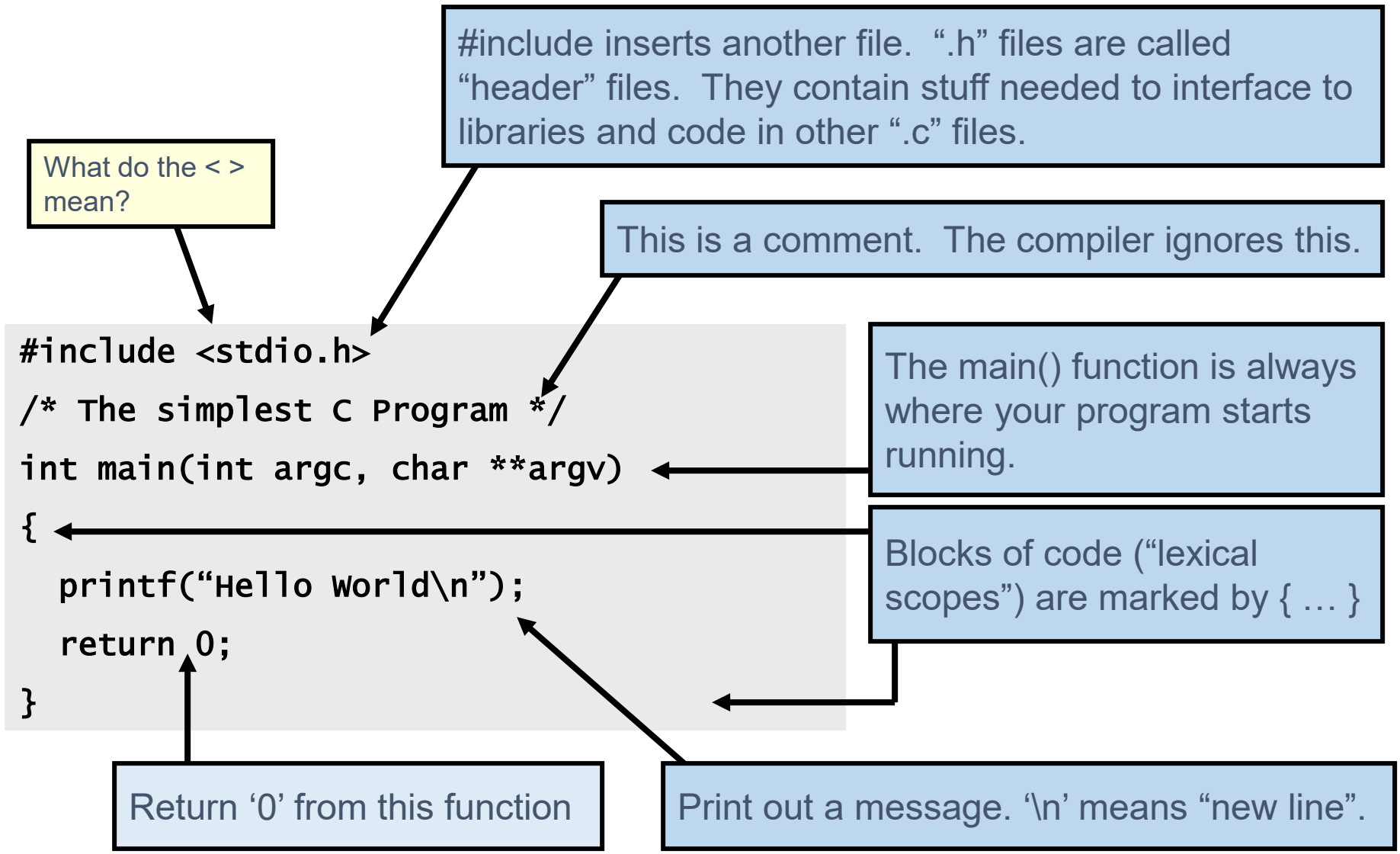
- Case matters, white space does not
- Comments go between /* and */
- Each statement is followed by a semicolon
- Execution begins in the main function

```
#include <stdio.h>
int main(int argc, char* argv[]) {
   /* start here */
   printf("Hello World\n");
   return 0;
   /*end here */
}
```

A sample program

```c
#include <stdio.h>


int main() {
    int year;

    printf("\n");
    printf("Enter a year: ");
    scanf("%d", &year);

    // leap year if perfectly divisible by 400

    if (year % 400 == 0) {
        printf("%d is a leap year.", year);
    }

    // not a leap year if divisible by 100
    // but not divisible by 400

    else if (year % 100 == 0) {
        printf("%d is not a leap year.", year);
    }

    // leap year if not divisible by 100
    // but divisible by 4

    else if (year % 4 == 0) {
        printf("%d is a leap year.", year);
    }

    // all other years are not leap years

    else {
        printf("%d is not a leap year.\n\n", year);
    }

    return 0;
}
```

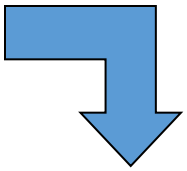
Some common properties of C



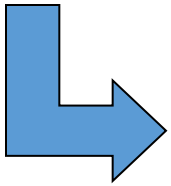
The compilation process

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

Preprocess



```
__extension__ typedef  unsigned long long int   __dev_t;
__extension__ typedef  unsigned int   __uid_t;
__extension__ typedef  unsigned int   __gid_t;
__extension__ typedef  unsigned long int   __ino_t;
__extension__ typedef  unsigned long long int   __ino64_t;
__extension__ typedef  unsigned int   __nlink_t;
__extension__ typedef  long int   __off_t;
__extension__ typedef  long long int   __off64_t;
extern void flockfile (FILE *__stream)  ;
extern int ftrylockfile (FILE *__stream)  ;
extern void funlockfile (FILE *__stream)  ;
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```



Compile

my_program

Compilation occurs in two steps: "Preprocessing" and "Compiling"

Why ?

In Preprocessing, source code is "expanded" into a larger form that is simpler for the compiler to understand. Any line that starts with '#' is a line that is interpreted by the Preprocessor.

- Include files are "pasted in" (#include)
- Macros are "expanded" (#define)
- Comments are stripped out (/* */ , //)
- Continued lines are joined (\ )

\ ?

The compiler then converts the resulting text into binary code the CPU can run directly.

C functions

A Function is a series of instructions to run. You pass Arguments to a function and it returns a Value.

"main()" is a Function. It's only special because it always gets called first when you run your program.

Return type, or void

Function Arguments

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

Calling a Function: "printf()" is just another function, like main(). It's defined for you in a "library", a collection of functions you can call from your program.

Returning a value

C functions

```
#include <stdio.h>

// Define a function to greet someone by name

void greet (char name[ ]) {
  printf("Hello, %s!\n", name);
}

int main( ) {
  // Call the greet function with the name "Bard".
  greet (“Idena");

  return 0;
}
```

C functions

```
#include <stdio.h>

// Define a function to compute the sum of two integers.
int add(int a, int b) {
  return a + b;
}

int main( ) {
  int num1 = 10;
  int num2 = 20;
  int sum = add (num1, num2);
  printf ("The sum of %d and %d is %d.\n", num1, num2, sum);
  return 0;
}
```

Memory locations

Memory is like a big table of numbered slots where bytes can be stored.

The number of a slot is its Address.
One byte Value can be stored in each slot.

Some "logical" data values span more than one slot, like the character string "Hello\n"

A Type names a logical meaning to a span of memory. Some simple types are:

`char`	a single character (1 slot)
`char [10]`	an array of 10 characters
`int`	signed 4 byte integer
`float`	4 byte floating point
`int64_t`	signed 8 byte integer

not always…

Signed?…

Addr	Value
0	
1	
2	
3	
4	'H' (72)
5	'e' (101)
6	'l' (108)
7	'l' (108)
8	'o' (111)
9	'\n' (10)
10	'\0' (0)
11	
12	

72?

What are C libraries?

- C is a lightweight language
  - Most of its intelligence is compartmentalized in libraries
  - Almost all c programs use the "stdio" or standard input/output library
  - Many also use the "math" library
- To use a library, include the header file (i.e., **stdio.h**) at the top of the file
- For most special purpose libraries (i.e., math) you need to include the math library

C variable types

- The most common types are: char, int, float, and double
- Strings are arrays of characters (will cover arrays later)
- Declare a variable before you use it:

```
/* declares an integer called x.  Its value is not assigned.*/
int x;
/* declares two floating point numbers;  set z equal to pi */
float y, z = 3.14159;
z = 4;   /* now z is equal to 4 */
myVal = 2;   /* An error because myVal is not declared. */
```

C variables

A Variable names a place in memory where you store a Value of a certain Type.

You first Define a variable by giving it a name and specifying the type, and optionally an initial value

declare vs define?

```
char x;
y='e';
```

Initial value of x is undefined

Initial value

Name

What names are legal?

Type is single character (char)

extern? static? const?

symbol table?

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	'e' (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

The compiler puts them somewhere in memory.

Expressions and evaluation

Expressions combine Values using Operators, according to precedence.

```
1 + 2 * 2       → 1 + 4       → 5
(1 + 2) * 2     → 3 * 2       → 6
```

Symbols are evaluated to their Values before being combined.

```
int x=1;
int y=2;
x + y * y       → x + 2 * 2     → x + 4     → 1 + 4     → 5
```

Comparison operators are used to compare values.
In C, 0 means “false”, and *any other value* means “true”.

```
int x=4;
(x < 5)                  → (4 < 5)                  → <true>
(x < 4)                  → (4 < 4)                  → 0
((x < 5) || (x < 4))     → (<true> || (x < 4))      → <true>
```

Not evaluated because first clause was true

Comparison operators

```
== equal to
<  less than
<= less than or equal
>  greater than
>= greater than or equal
!= not equal
&& logical and
|| logical or
!  logical not
```

```
+  plus
-  minus
*  mult
/  divide
%  modulo
```

```
&  bitwise and
|  bitwise or
^  bitwise xor
~  bitwise not
<< shift left
>> shift right
```

The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit. For oft-confused cases, the compiler will give you a warning "Suggest parens around …" – do it!

Beware division:

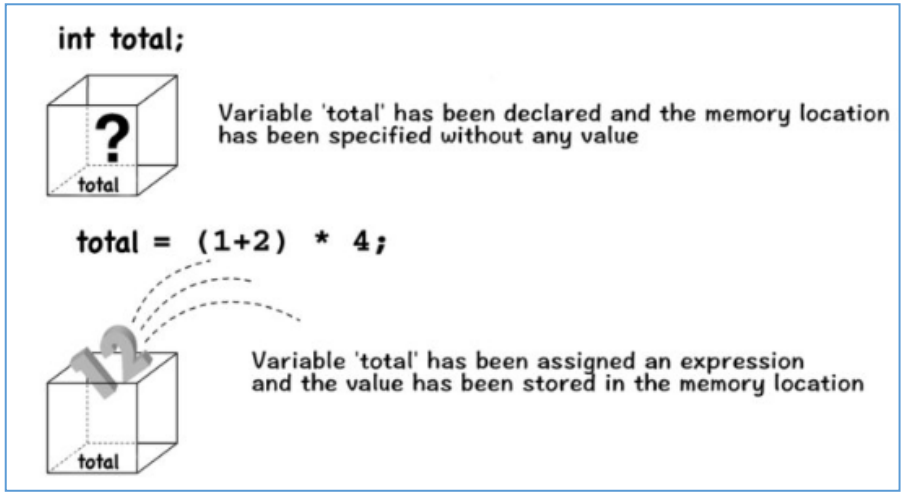
- If second argument is integer, the result will be integer (rounded):
  5 / 10 → 0 *whereas* 5 / 10.0 → 0.5
- Division by 0 will cause a FPE

Don't confuse & and &&..
1 & 2 → 0 *whereas* 1 && 2 → <true>

The assignment statement

- An Assignment statement is a statement that is used to set a value to the variable name in a program
  - It stores a value in the memory location which is denoted by a variable name
  - The assignment operator is = (not ==)

```
variable = expression;
total = (1 + 2) * 4;
```



The assignment statement

- Variations

  `+=`

  `-=`

  `*=`

  `/=`

- Example

  `a += b;`

- Chain or multiple assignment

  `a = b = c = d = expression;`

Compound statement

- A compound statement in C is a group of statements that are enclosed in curly braces
- Compound statements can be used to group related statements together, to make the code more readable and maintainable

```c
{
  // This is the first statement in the compound statement.
  printf("Hello, world!");

  // This is the second statement in the compound statement.
  getchar();
}
```

```c
if ( i > 0 )
{
    line[i] = x;
    x++;
    i--;
}
```

Scoping in C

```
int main() {
  int x = 10;
  { // Block1
    int y = 20; // accessible in Block1 only
    printf("x in Block 1: %d\n", x);
    printf("y in Block 1: %d\n", y);
  }
  printf("x in main: %d\n", x);
  return 0;
}
```

Scoping in C

```c
void greet(char* name) {
  int age = 30;

  printf("Hello, %s! You are %d years old.\n", name, age);
}

int main() {
  greet("Bard");

  return 0;
}
```

```c
#include <stdio.h>

int x = 10;

void print_x() {
  printf("x: %d\n", x);
}

int main() {
  int x = 20;

  print_x(); // Prints 10, not 20

  return 0;
}
```

The if statement

- Syntax: **if (expression) statement;**
- If the expression is true (not zero), the statement is executed. If the expression is false, it is not executed.
- You can group multiple expressions together with braces:

```
if (expression) {
  statement 1;
  statement 2;
  statement 3;
}
```

The if/else statement

- Syntax: **if (expression) statement1; else statement2;**
- If the expression is true, statement1 will be executed, otherwise, statement2 will be

```
if (myVal < 3){
   printf("myVal is less than 3.\n");
}
else {
    printf("myVal is >= to 3.\n");
}
```

Assignment operators

```
x = y    assign y to x
x++      post-increment x
++x      pre-increment x
x--      post-decrement x
--x      pre-decrement x
```

```
x += y   assign (x+y) to x
x -= y   assign (x-y) to x
x *= y   assign (x*y) to x
x /= y   assign (x/y) to x
x %= y   assign (x%y) to x
```

Note the difference between ++x and x++:

```
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Don't confuse = and ==! The compiler will warn "suggest parens".

```
int x=5;
if (x==6)   /* false */
{
  /* ... */
}
/* x is still 5 */
```

```
int x=5;
if (x=6)   /* always true */
{
  /* x is now 6 */
}
/* ... */
```

The "while" loop

- Syntax: **while (condition) statement;**
- The condition is evaluated, if it is true, the body of loop will be executed

```
while(condition){
        //code to be executed
}
```

The for loop

- Syntax: **for (expr1; expr2; expr3) statement;**
- Syntax: **for (initialization; test; increment) statement;**

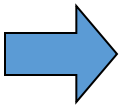
- The for loop will first perform the initialization. Then, as long is test is TRUE, it will execute statements. After each execution, it will increment

```
for (i = 0; i < 3; i++) {
  printf("Counter = %d\n", i);
}
```

The for loop

The "for" loop is just shorthand for this "while" loop structure.

```
float pow(float x, uint exp)
{
  float result=1.0;
  int i;
  i=0;
  while (i < exp) {
    result = result * x;
    i++;
  }
  return result;
}

int main(int argc, char **argv)
{
  float p;
  p = pow(10.0, 5);
  printf("p = %f\n", p);
  return 0;
}
```



```
float pow(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  return result;
}

int main(int argc, char **argv)
{
  float p;
  p = pow(10.0, 5);
  printf("p = %f\n", p);
  return 0;
}
```

Summary

- Learned how to write and compile a C program
- Learned what C libraries are
- Introduced the C variable types
- Introduced how to use if and if/else statements
- Introduced how to use the for and while statements

- References: some slides from Lewis Girod, CENS Systems Lab

Try yourself

- Find all three-digit numbers that are equal to the sum of the cube of their digits

Try yourself

```
#include <stdio.h>


int main() {
  int num, hundredsDigit, tensDigit, unitsDigit;
  for (num = 100; num <= 999; num++) {
    // Extract the hundreds digit, tens digit, and units digit of the number
    hundredsDigit = num / 100;
    tensDigit = (num % 100) / 10;
    unitsDigit = num % 10;
    int sumOfCubes = hundredsDigit * hundredsDigit * hundredsDigit + tensDigit *
tensDigit * tensDigit + unitsDigit * unitsDigit * unitsDigit;
    if (sumOfCubes == num) {
      printf("%d is equal to the sum of the cube of its digits.\n", num);
    }
  }
  return 0;
}
```

Try yourself

```
#include <stdio.h>
int main() {
  int num, originalNum, remainder, result;
  for (num = 100; num <= 999; num++) {
    originalNum = num;
    result = 0;
    while (originalNum != 0) {
        remainder = originalNum % 10;
        result += remainder * remainder * remainder;
        originalNum /= 10;
    }
    if (result == num) {
        printf("%d is an Armstrong number.\n", num);
    }
  }
  return 0;
}
```

Where is printf declared?

```
#include <stdio.h>

int main (void) {
 printf("Hello, World!\n");
 return 0;
}
```

#include <file.h>

- AKA header file
- A file of *C* code that is *copied into your program* at compile time
  - By the preprocessor
- Spells out the *contract* of the interface between implementer and client
- Declares everything that your program needs to know about the "standard I/O facilities" of *C* **…**

Where is printf declared?

```
#include <stdio.h>

int main (void) {
 printf("Hello, World!\n");
 return 0;
}
```

Formatting with printf

%c	character
%d	decimal (integer) number (base 10)
%e	exponential floating-point number
%f	floating-point number
%i	integer (base 10)
%o	octal number (base 8)
%s	a string of characters
%u	unsigned decimal (integer) number
%x	number in hexadecimal (base 16)
%%	print a percent sign
\%	print a percent sign

Formatting with printf

- The following are from Alvin Alexander's website
- A printf format reference page or cheat sheet (C, Java, Scala, etc.)
- https://alvinalexander.com/programming/printf-format-cheat-sheet/

printf special characters

- The following character sequences have a special meaning when used as printf format specifiers:

\a	audible alert
\b	backspace
\f	form feed
\n	newline, or linefeed
\r	carriage return
\t	tab
\v	vertical tab
\\	backslash

Controlling integer width with printf

- The %3d specifier is used with integers, and means a minimum width of three spaces, which, by default, will be right-justified

printf("%3d", 0);	0
printf("%3d", 123456789);	123456789
printf("%3d", -10);	-10
printf("%3d", -123456789);	-123456789

Left-justifying printf integer output

- To left-justify integer output with printf, just add a minus sign (-) after the % symbol, like this:

printf("%-3d", 0);	0
printf("%-3d", 123456789);	123456789
printf("%-3d", -10);	-10
printf("%-3d", -123456789);	-123456789

Left-justifying printf integer output

- To left-justify integer output with printf, just add a minus sign (-) after the % symbol, like this:

printf("%-3d", 0);	0
printf("%-3d", 123456789);	123456789
printf("%-3d", -10);	-10
printf("%-3d", -123456789);	-123456789

The printf integer zero-fill option

- To zero-fill your printf integer output, just add a zero (0) after the % symbol, like this:

printf("%03d", 0);	000
printf("%03d", 1);	001
printf("%03d", 123456789);	123456789
printf("%03d", -10);	-10
printf("%03d", -123456789);	-123456789