Features

- a "general purpose" language
  - equally usable for applications programming and systems programming, for example:
    - develop a network protocol
    - develop a database management system
    - write a compiler for another language (C++, Eiffel, …)
- it's ubiquitous: where you find UNIX you usually find C
- it provides the basis for understanding other languages, most notably C++

Features

- most C toolchains have a small footprint
  - popular choice for developing **embedded systems**
  - operating systems research and development
  - good choice for systems programs that one expects to port
- Compile-time features
  - ANSI-compliant compilers provide extensive compile-time diagnostics
  - ANSI-compliant compilers provide a continuum of optimizations; from **none** to **conservative** to **aggressive**

Features

- Run-time features (i.e., “pluses”):
  - easy to adapt a C compiler’s output (executables) to the execution environment on a platform: Windows, Mac, UNIX
- Run-time features missing (i.e., also could be considered as “efficiency pluses”!):
  - no native array access bounds checking
  - no native null-pointer checks (use a custom library for this)
  - no native checks on uninitialized variables (some scenarios can be checked at compile-time)

How do I use C?

- Write an application

```
$ vim hello.c
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return (0);
}
```

- Compile the source file into an object file

```
$ gcc -ansi -Wall -c hello.c
```

- Link the object file to the "Standard C Runtime Library" to produce an executable (hello)

```
$ gcc -o hello hello.o
          OR
$ gcc -o hello -lm hello.o   # use floating point math
```

How do I use C ?

- Run the executable:

```
$ ./hello
Hello, World!
```

- Basic rules:
  - all C stand-alone programs must have a function called "`main()`"
  - keywords are always lowercase; you cannot use a keyword as an identifier
  - statements must be terminated with a semicolon

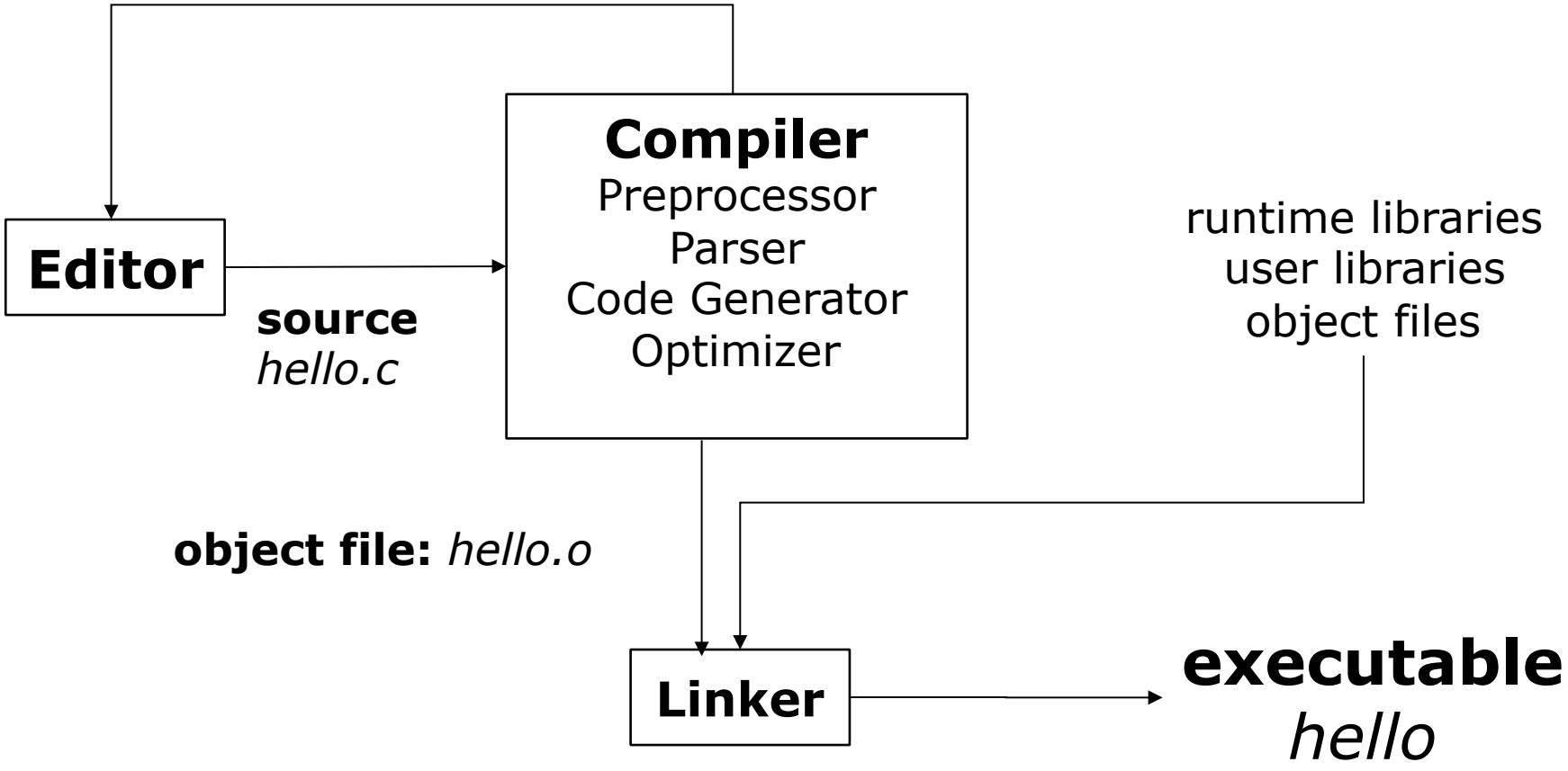
How do I use C ?

- Basic rules (continued):
  - Comments are delimited by `/* ... */`

```
/* Everything between "slash star" and
    "star slash" is a comment, even if it
    spans several lines. Be careful not
    to nest comments; some compilers are
    unable to handle them. */
```

  - Single line comments are not ANSI C (//)
- **Upcoming labs:**
  - introduce the GNU toolchain
  - aspects of the C execution model

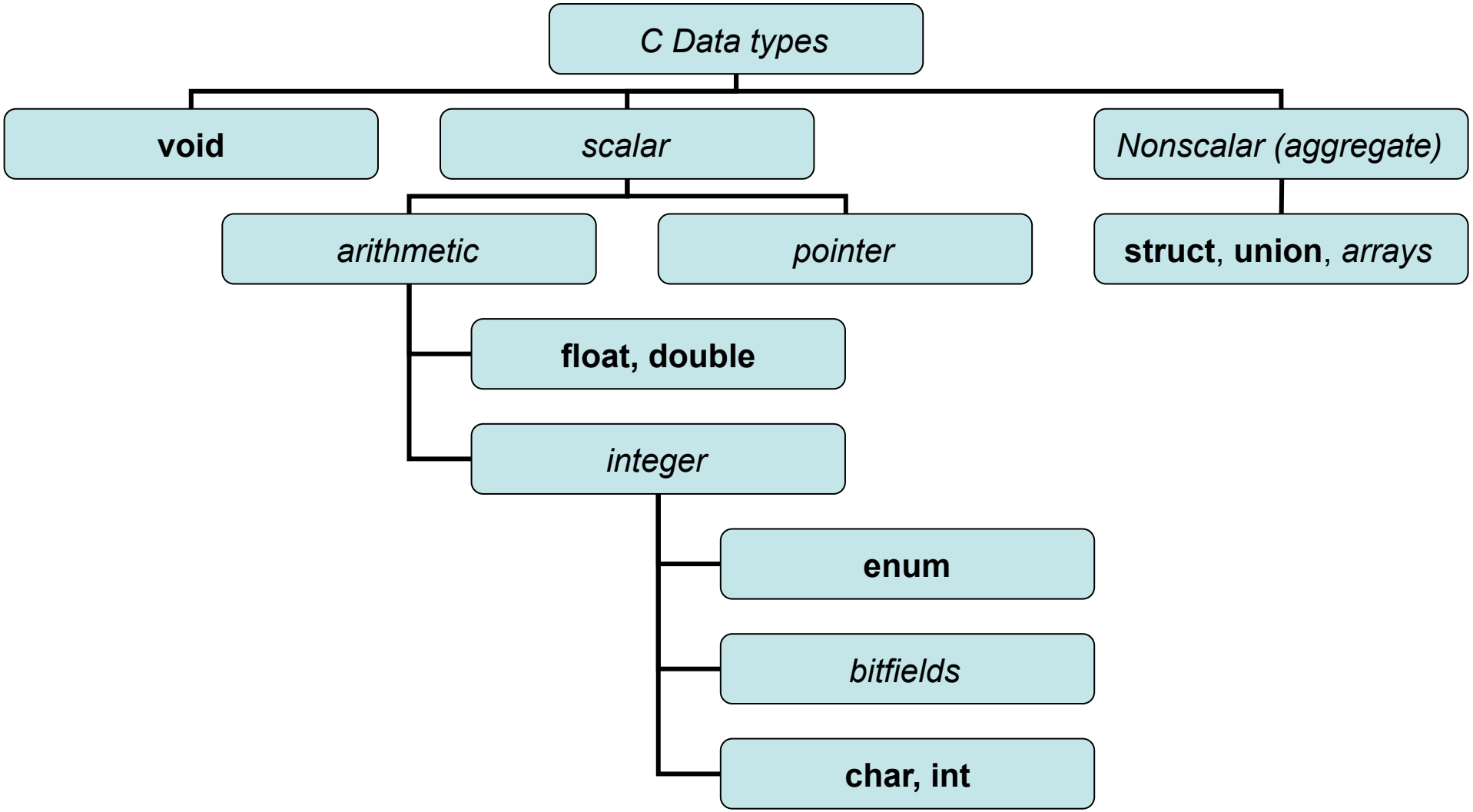
How do I use C ?


Editor
source
hello.c
Compiler
Preprocessor
Parser
Code Generator
Optimizer
runtime libraries
user libraries
object files
object file: hello.o
Linker
executable
hello

A word about formatting style

- any amount of white space is considered a single space
  - tabs and spaces can be used liberally
- white space improves code readability
- commenting is important as a maintenance tool
- use tabbing in conjunction with curly braces (`{ , }`) to indicate different levels of nested functions.
- in C, type declarations must appear at the beginning of a scope
- scope begins and ends with curly braces (`{ , }`)
- use **underscores_for_variables** rather **thanCamelCaps**

C Data Types


C Data types
void
scalar
Nonscalar (aggregate)
arithmetic
pointer
struct, union, arrays
float, double
integer
enum
bitfields
char, int

Basic Data Types

C	Java
`char`	`char`
`int`	`int`
`enum`	`enum`
`float`	`float`
`double`	`double`
`--`	`Boolean`

Type qualifiers

- C supports four type qualifiers; keywords which qualify certain scalar arithmetics:
  - **`long`** , **`short`** : affect the range of an integer or floating point numbers
  - **`signed`**, **`unsigned`**: just that, indicates that an integer is signed or unsigned

Qualified Basic Types

Basic type	Qualified basic type

`char`	`char`
	`signed char`
	`unsigned char`

`int`	`int`
	`short int`
	`long int`
	`long long int`
	`unsigned int`
	`unsigned short int`
	`unsigned long int`
	`unsigned long long int`

`double`	`long double`

Aggregate Data Types

- C supports the following aggregate data types:
  - **`struct`** types: one mechanism to declare user-defined types
    - like records in Pascal/Modula/Oberon
    - we'll look at these later
    - **significantly different from Java classes**
  - array types: you can define an array of any scalar or aggregate type
    - describe these later
  - **`union`** types: similar to structs, but members are overlaid (sharing storage)

Literals

- Character constants (8-bit ASCII):
  - `char ch = 'A', bell = '\b', formfeed = '\f';`
- Numeric literals
  - Integer:
    - `int a = 10, b = 0x1CE, c = 0777;`
    - `unsigned int x = 0xfffU;`
    - `long int y = 2L;`
  - Floating point:
    - `float x = 3.1415F;`
    - `double x = 1.25, y = 2.5E10, z = -2.5e-10;`
    - `long double x = 3.5e3L;`

Danger! String literals

- String literals
  - `char *s = "unable to open file\n";`
- We will get to C strings in due course, but here is an early warning:
  - The variable "**s**" above might appear to act like a string…
  - … but it is actually a variable holding an address to a "static string table"

Storage classes

- C provides the following four storage classes:
  - **auto**: applies only to variables declared at function scope
  - **register**: a **hint** to the compiler to place a variable in a CPU register
  - **static**: internal linkage, and static storage allocation
  - **extern**: external linkage, not a definition
- Storage classes are used to modify a variable declaration or definition
- Current practice:
  - avoid using "auto" or "register"
  - use "static" and "extern" to control variable visibility, and these will be the only storage classes we will use in this course

Scalar Variable Definitions

- Defining variables
  - general definition syntax:

```
<type> <name>;
```

  - definition with initialization:

```
<type> <name> = <value>;
```

  - with a storage class modifier:

```
<storage class> <type> <name>;
<storage class> <type> <name> = <value>;
```

Scalar Variable Definitions

- Examples:

```
extern int tics;

double long int x = 4L;
int a, b, c;
unsigned int a, b = 0x1fU;

char c = 'A';
static unsigned char esc = '\0x27';

double pi;
long double ptime;
enum { red, green, blue } colour;
```

Introduction to C Programming (cont)

- Aggregate Data Type: C arrays
- Statements
- Simple I/O
- Control flow
- User defined types
  - type definitions **`(typedef)`**
  - enumerations **`(enum)`**
  - Aggregate data type: structures **`(struct)`**

C Arrays

- An array is a group of data elements of the same type, accessed using the same identifier; e.g., `X[3], X[11]`
- Arrays may be statically or dynamically allocated. Static arrays cannot grow at runtime. Dynamic arrays can grow at runtime (using standard library functions).
- Arrays may be multidimensional; e.g., `X[row][column]`
- Access to the elements of an array is accomplished using integer indices
- If an array is dimensioned to hold `size` elements, the elements are indexed from `0` up to `size-1`
- **C provides no array bounds checking**, so accessing elements beyond index `size-1`, or below index `0` can cause a segmentation fault
- Static arrays can be auto-initialized at runtime

C Arrays (2)

- syntax for a one-dimensional array declaration:
  `<storage class> <type> <identifier>[<size>]`
  e.g. `double vector[3];`
  `char buffer[256];`
- `<size>` must be known at compile time
- `<size>` **is not a part of an array data structure**. Programmer has to manage correct access to array!

- Examples:

```
double f[3] = {0.1, 2.2, -100.51};

int freq[10] = {20,12}; /* freq[0] = 20,
                           freq[1] = 12,
                           freq[2] = 0,
                              ...
                           freq[9] = 0 */
```

C Statements

- ```
  S = S; S;
  | x = e
  | f(e1,…,en)
  | if (bexpr) {S} [else if {S}] [else {S}]
  | switch(e) { case e1: S case e2: S … default: S }
  | while (bexpr) {S}
  | do {S} while (bexpr)
  | for (e1; bexpr; e2) {S}
  | break
  | continue
  | return e
  | ε
  ```
- where,
  - ***S*** is a statement
  - ***e,e1,en*** are general expressions
  - ***bexpr*** is a boolean expression
  - **ε** is the empty or null statement
  ```

Simple I/O (Text I/O)

- **standard input (stdin)**
- **`char *fgets(char *buf, int n, FILE *stream)`**
  - read at most **`n-1`** characters from **`stream`** and copy to location **`buf`**; input terminates when newline encountered or n-1 characters input. Appends a null character to end of buffer.
  - returns **`NULL`** if error or end-of-file encountered
  - set **`stream`** to **`stdin`** to accept input from standard input
- **`int scanf( char *format, [...] )`**
  - read formatted data from standard input
  - returns **`EOF`** when end-of-file encountered, otherwise it returns the number of fields successfully converted
  - the format specifiers are encoded in the string **`format`**
  - takes a variable number of arguments

Simple I/O (Text output)

- **standard output (stdout)**
- `int printf( char *format, […])`
  - print formatted output to standard output
  - returns the number of characters printed
  - the format specifiers are encoded in the string **`format`**
  - takes a variable number of arguments
- Examples:
  - `printf("My name is %s\n", name); /* char array */`
  - ```
    printf("My name is %s and my age is %d\n", name, age);
    /* name is a char array, age is an int */
    ```
  - ```
    printf("The temperature today is %f\n", temp_celsius);
    /* temp_celsius is a float */
    ```
  - ```
    printf("%d/%d/%d", year, month, day);
    /* year, month and day are ints; there is no newline */
    ```
    ```

Input/Output Model: example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAXLINELEN 80

char line[MAXLINELEN];

void process_line(char *in_line) {
    printf("%d\n", strlen(in_line));
}


int main() {
    while(fgets(line, MAXLINELEN, stdin) != NULL) {
        process_line(line);
    }

    if (ferror(stdin) != 0) {
        perror("<stdin>"); /* print error message */
        exit(1);
    }

    exit(0);
}
```