Lecture 2: C Programming Intro

Vikram Iyer

Adapted from material by Blake Hannaford and Dan Garcia

Administrative

- Are people having issues with Ed Discussion?
- Lab kit pickup
  - MONDAY: 12:00 - 2:00
  - TUESDAY: 11:30 - 5:30
  - WEDNESDAY: 12:00 - 2:00
  - THURSDAY: 10:00 - 2:00
  - FRIDAY: 10:00 - 12:00
- Partner matching form: https://forms.gle/oK9LYjTZyt3pJSYz7
- Questions about course format or resources?

Last time: What are embedded systems?

Tiny computers that are *embedded* or hidden inside something else.









Plan for today

- Intro to C programming
  - Overview and comparison to Java
  - What happens when you compile a program?
  - Structure of a C program
- “Hello world” Program
- Structure of C program
- Arduino Demo

C vs. Java™ Overview (1/2)

Java

- Object-oriented (OOP)
- "Methods"
- Class libraries of data structures
- Automatic memory management

C

- No built-in object abstraction. Data separate from methods.
- "Functions"
- C libraries are lower-level
- Manual memory management
  - Need to use pointers

C vs. Java™ Overview (2/2)

Java

- High memory overhead from class libraries
- Relatively Slow
- Arrays initialize to zero
- Syntax:

```
/* comment */
// comment
System.out.print
```

C

- Low memory overhead
- Relatively Fast
- Arrays initialize to garbage
- Syntax:

```
/* comment */
// comment
printf
```

C vs. Java™ Compilation

C *compilers* take C and convert it into an architecture specific machine code (string of 1s and 0s).

- Unlike Java which converts to architecture independent bytecode.
- Unlike interpreted code in Python.
- These differ mainly in when your program is converted to machine instructions.
- For C, generally a 2 part process of compiling .c files to .o files, then linking the .o files into executables. Assembling is also done (but is hidden, i.e., done automatically, by default)

What does compiling do?

```
if (x != 0) y = (y+z)/x;
```

Compiler

```
    cmpl    $0, -4(%ebp)
    je      .L2
    movl    -12(%ebp), %eax
    movl    -8(%ebp), %edx
    leal    (%edx,%eax), %eax
    movl    %eax, %edx
    sarl    $31, %edx
    idivl   -4(%ebp)
    movl    %eax, -8(%ebp)
.L2:
```

Assembler

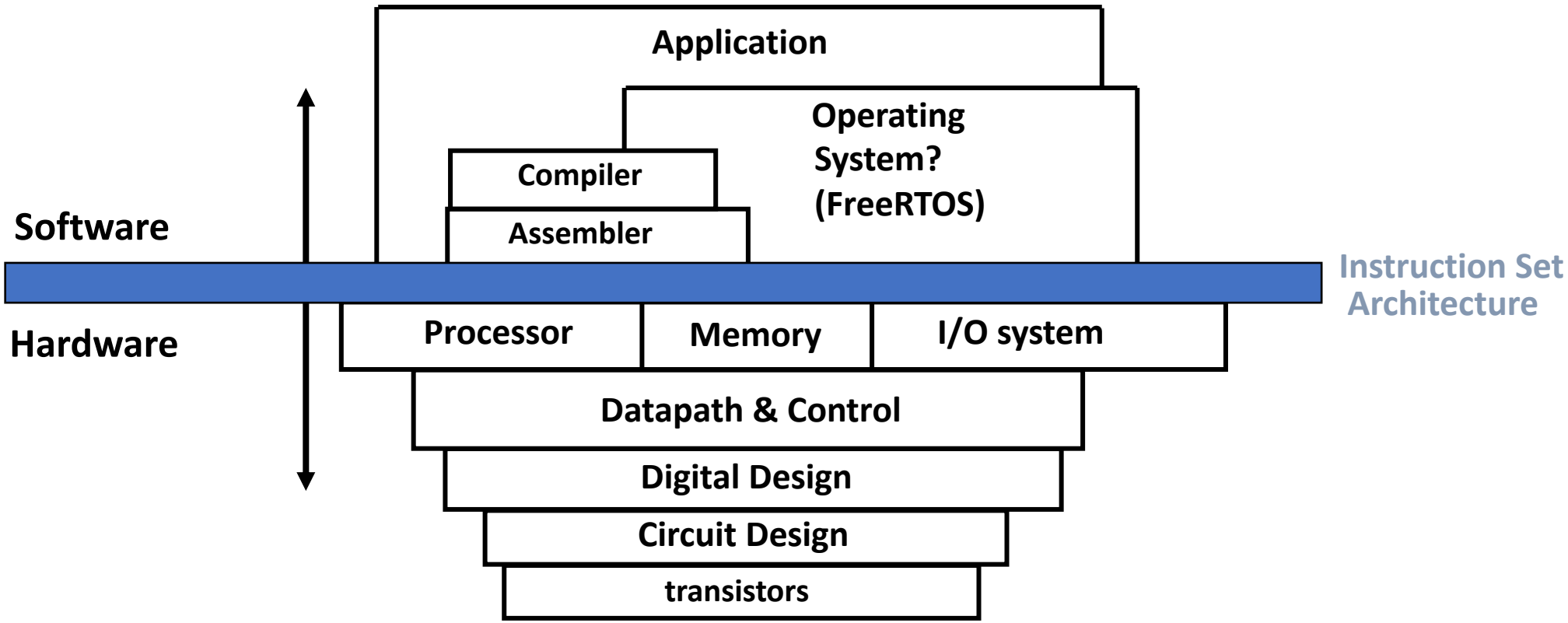
```
10000011011111000010010000011100000000000
0111010000011000
10001011010001000010010000010100
10001011010001100010010100010100
10001101000001000000010
1000100111000010
11000001111110100011111
11110111011111000010010000011100
```

High Level Language (*e.g.* C, Java)

Assembly Language

Machine Code

Overview of HW/SW interface



Atmel AVR Instruction set

Table 4-2. Arithmetic and Logic Instructions

Mnemonic	Operands	Description		Op		Flags	#Clocks AVR	#Clocks AVRxm	#Clocks AVRxt	#Clocks AVRrc
ADD	Rd, Rr	Add without Carry	Rd	←	Rd + Rr	Z,C,N,V,S,H	1	1	1	1
ADC	Rd, Rr	Add with Carry	Rd	←	Rd + Rr + C	Z,C,N,V,S,H	1	1	1	1
ADIW	Rd, K	Add Immediate to Word	Rd	←	Rd + 1:Rd + K	Z,C,N,V,S	2	2	2	N/A
SUB	Rd, Rr	Subtract without Carry	Rd	←	Rd - Rr	Z,C,N,V,S,H	1	1	1	1
SUBI	Rd, K	Subtract Immediate	Rd	←	Rd - K	Z,C,N,V,S,H	1	1	1	1
SBC	Rd, Rr	Subtract with Carry	Rd	←	Rd - Rr - C	Z,C,N,V,S,H	1	1	1	1
SBCI	Rd, K	Subtract Immediate with Carry	Rd	←	Rd - K - C	Z,C,N,V,S,H	1	1	1	1
SBIW	Rd, K	Subtract Immediate from Word	Rd + 1:Rd	←	Rd + 1:Rd - K	Z,C,N,V,S	2	2	2	N/A
AND	Rd, Rr	Logical AND	Rd	←	Rd • Rr	Z,N,V,S	1	1	1	1

C vs. Java™ Compilation Advantages

- Great run-time performance: generally much faster than Java for comparable code (because it optimizes for a given architecture)

- OK compilation time: enhancements in compilation procedure (`Makefile`s) allow only modified files to be recompiled

C vs. Java™ Compilation Disdvantages

- All compiled files (including the executable) are architecture specific, depending on *both* the CPU type and the operating system.

- Executable must be rebuilt on each new system.
  - Called "porting your code" to a new architecture.

- The "change→compile→run [repeat]" iteration cycle is slow

C vs. Java™ Compilation Disdvantages

- All compiled files (including the executable) are architecture specific, depending on *both* the CPU type and the operating system.

- Executable must be rebuilt on each new system.
  - Called "porting your code" to a new architecture.

- The "change→compile→run [repeat]" iteration cycle is slow

C vs. Java™ Compilation Disdvantages

- All compiled files (including the executable) are architecture specific, depending on *both* the CPU type and the operating system.

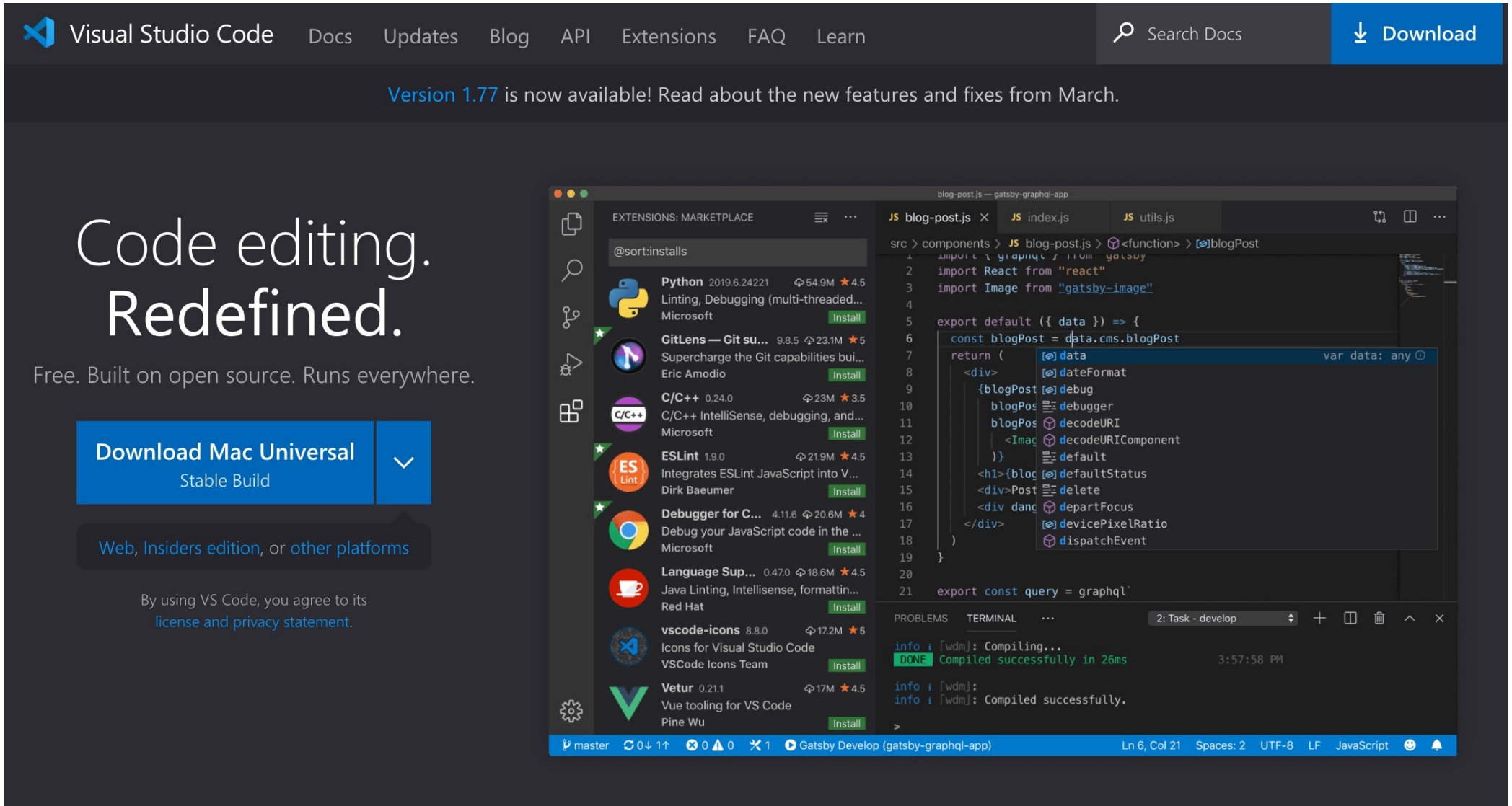
- Executable must be rebuilt on each new system.
  - Called "porting your code" to a new architecture.

- The "change→compile→run [repeat]" iteration cycle is slow

“Hello World” in C

```
// Simple C program to display "Hello World"
// Header file for input output functions
#include <stdio.h>
// main function - where program execution starts

int main()
{
    // prints hello world
    printf("Hello World\n");
    return 0;
}
```

Visual Studio Code: code.visualstudio.com



Writing a C program: `main()`

- Execution starts with first line of `main()`.

- To get the main function to accept arguments, use this:

```
int main (int argc, char *argv[])
```

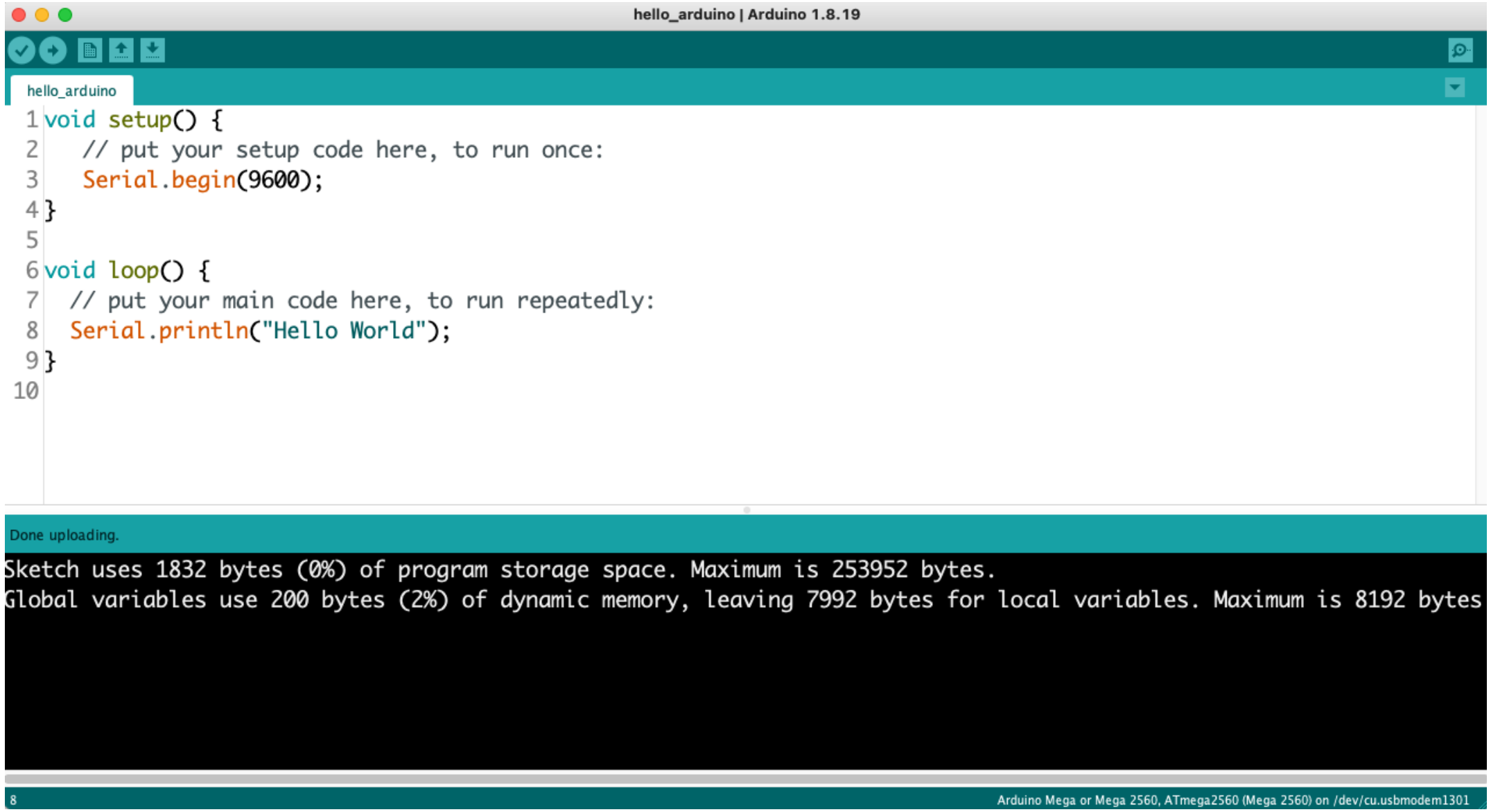
- What does this mean?
  - `argc` will contain the number of strings on the command line (the executable counts as one, plus one for each argument).
    Example Here `argc` is 2: `unix% sort myFile`

  - `argv` is a pointer to an array containing the arguments as strings (more on pointers later).

“Hello World” on Arduino

```
// Simple Arduino program to display "Hello World"
void setup() {
   // put your setup code here, to run once:
   Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.println("Hello World");
}
```

Arduino IDE


hello_arduino | Arduino 1.8.19
hello_arduino
1 void setup() {
2 // put your setup code here, to run once:
3 Serial.begin(9600);
4 }
5
6 void loop() {
7 // put your main code here, to run repeatedly:
8 Serial.println("Hello World");
9 }
10
Done uploading.
Sketch uses 1832 bytes (0%) of program storage space. Maximum is 253952 bytes.
Global variables use 200 bytes (2%) of dynamic memory, leaving 7992 bytes for local variables. Maximum is 8192 bytes
8
Arduino Mega or Mega 2560, ATmega2560 (Mega 2560) on /dev/cu.usbmodem1301

Arduino programs: `setup()` and `loop()`

`setup()`

- Execution starts with first line of `setup()` upon power cycling or RESET
- This function only runs once at the start of your program
- Used for initialization

`loop()`

- This function runs infinitely and executes the main code

https://github.com/arduino/ArduinoCore-avr/blob/master/cores/arduino/main.cpp

```
#include <Arduino.h>

/*Various inits and defines omitted*/

int main(void) {
    init();
    initVariant();
    #if defined(USBCON)
        USBDevice.attach();
    #endif setup();
    for (;;) {
        loop();
        if (serialEventRun)
            serialEventRun();
        }
    return 0;
}
```

Organizing C programs

- A C program is typically written in multiple files
  - Starts with a C-Source file (.c file) that contains the function `main()`
  - **Arduino** starts with functions `setup()` and `loop()` .ino
  - Main or loop call other functions
  - These are often defined in other files
- Simpler editing
- Helps modularity and multiple authors
  - Can have library that you can easily import into other projects

Organizing C programs

File: main.c

Hello_arduino.ino

```
main ()
   {
     ... code   ...
   }

fcn1 ()
   {
   ... code ...
   }
```

File: functions.c

```
fnc2 ()
   {
     ... code   ...
   }

fcn3 ()
   {
   ... code ...
   }

... etc ...
```

Organizing C programs

File: main.c

```
main ()
   {
     ... code   ...
   }
```

```
   {
   ... code ...
   }
```

File: functions.c

```
fnc2 ()
   {
     ... code   ...
   }
```

```
   {
   ... code ...
   }

... etc ...
```

How do we tell our compiler about these other functions?

Function prototypes

- All functions (except `main()`) require a **function prototype** in any file in which they will be called or defined.
- Defines function name, input types and output types
- All functions are visible from any file (but prototype must be supplied in the file)
- **Arduino** will try to define function prototypes for you, but *it doesn't always work*. For this class define prototypes.

```
// Function prototype
int square_int(int);


// Function definition
int square_int(int x)
{
    return x * x;
}
```

Organizing prototypes in header files (.h)

Header files gets read into your C-Source file just prior to compilation

File: main.c

```
#include square.h
main()
   {
     ... code   ...
   }
```

File: square.h

```
// Function prototype
int square_int(int);
```

Organizing prototypes in header files (.h)

Header files read into C-Source file by the pre-processor during compilation

Pre-processor output:

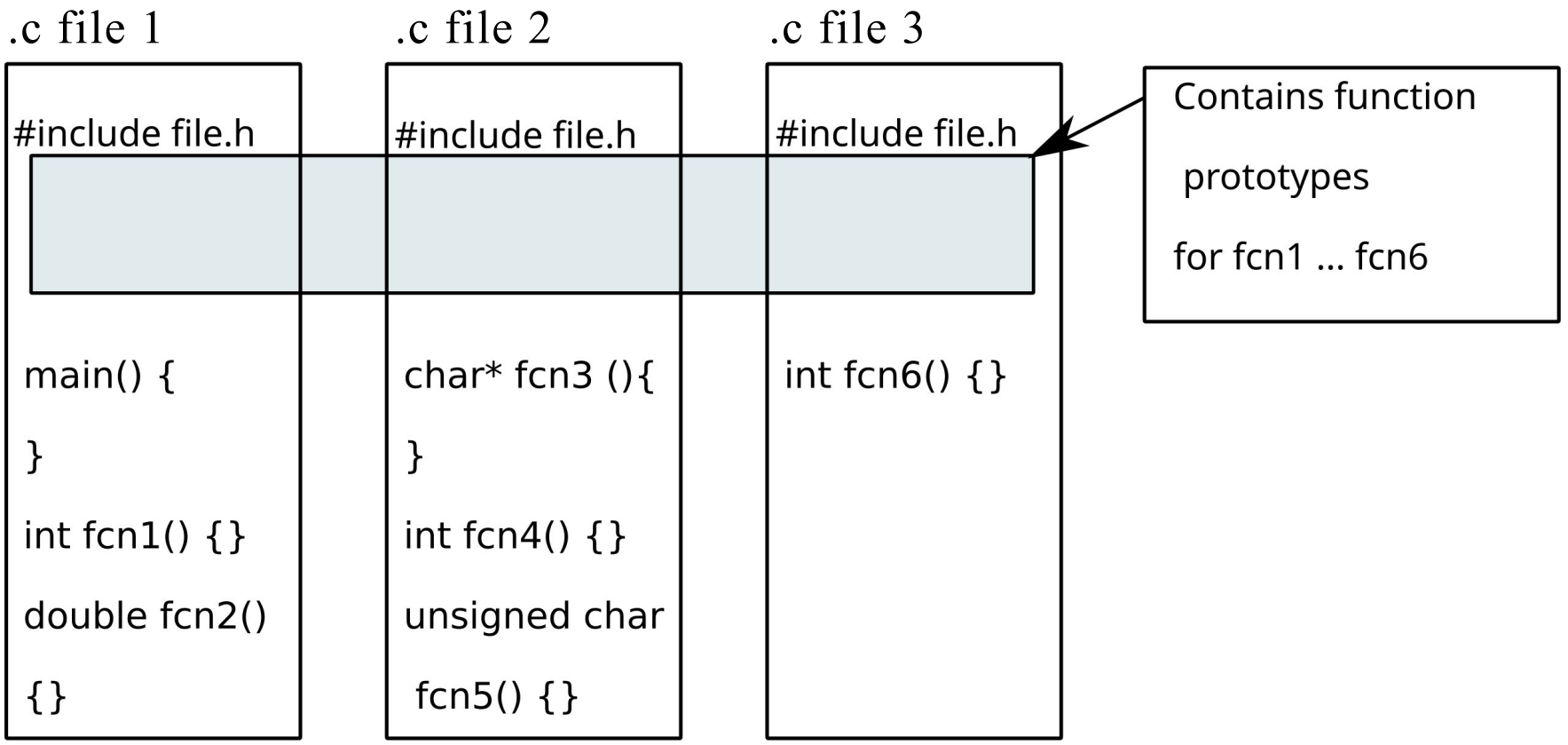
```
// Function prototype
int square_int(int);

main()
   {
     ... code   ...
   }
```

Organizing prototypes in header files (.h)

Why is it important to do this?

- `#include` text will be the same in every ".c" files
- Can edit just 1 copy
- Less typing and ensures consistence

Organizing prototypes in header files (.h)



A single ".h" file introduces the exact same function prototypes and data declarations into multiple files without having to maintain separate identical copies in multiple files

Other C pre-processor functions: `#define`

- like the search-replace function in your word processor.
- `#define TRUE 1`
- C pre-processor will replace all occurrences of `TRUE` with `1`.

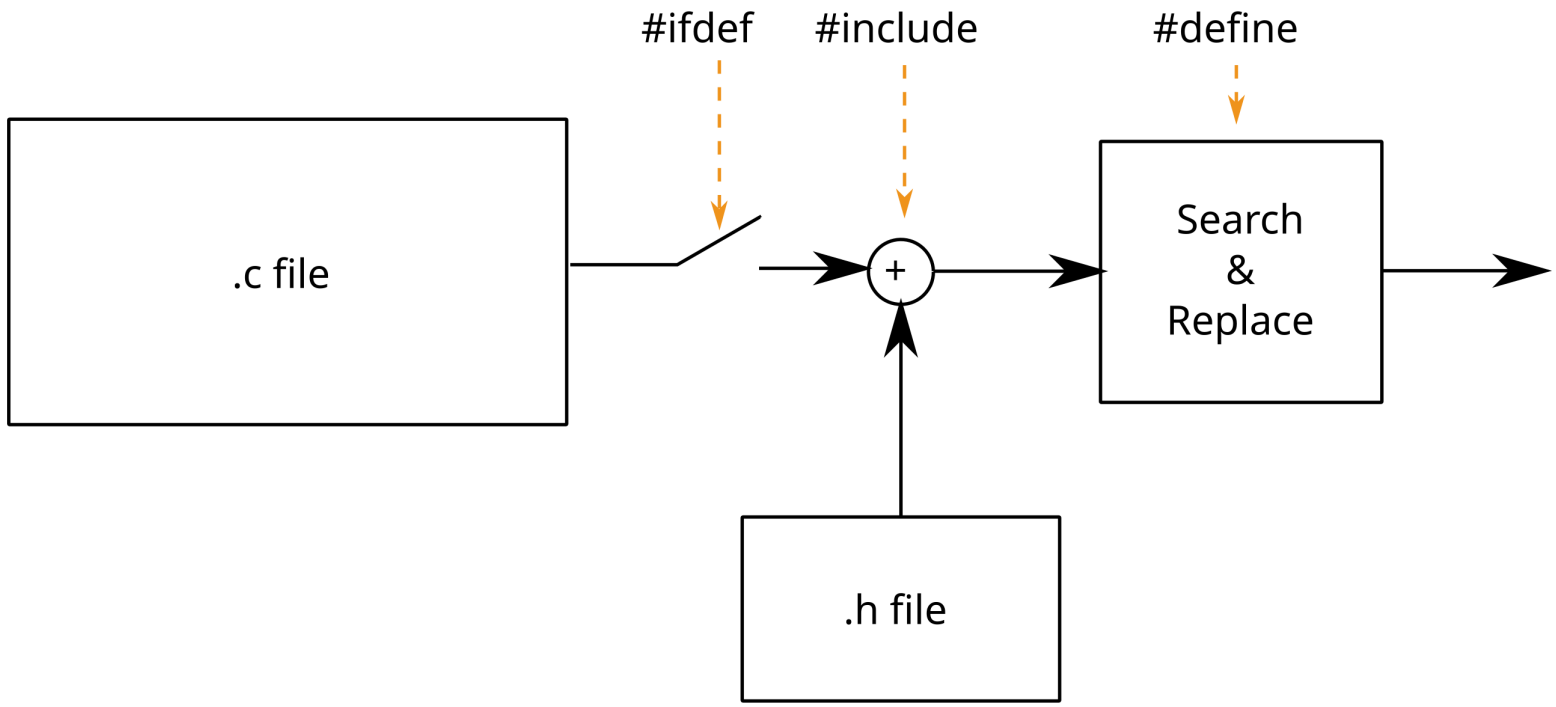
Other C pre-processor functions:

#ifdef / #endif

Can use this to turn on and off entire blocks of code

```
#define LINUX 1
//#define WINDOWS 1

...
#ifdef WINDOWS
      ... windows-specific
#endif

#ifdef LINUX
      ... linux-specific
#endif
```

Other C pre-processor overview



The C pre-processor applies some convenience features to your code before compilation. Specifically, the `#ifdef` macro can turn off C-code until the `#endif`. You can use this to define a debug mode, compatibility with different peripherals, etc

What does the compiler do with your code?

For each ".c" file:

1. Run C pre-processor on the ".c" file
2. Compile output into "object" file.
    - Incomplete machine language program
    - ".o" file for each ".c" file

Then: "Link" all ".o" files, and library files, to create "executable" binary file.

Variable scope

- Variable declared inside a function is local.
- Variable declared outside a function is visible anywhere inside that ".c" file.
- Variable declared outside a function can also be visible in other ".c" files — if `extern` is used.
- Variables declared with `extern` must be declared outside a function in another ".c" file.

```
fcn1 ()
{
  // Local variable
  int x = 1;
}

// Global variable
int x = 2;
```

Function scope

- Functions cannot be used without a function prototype.
- Each ".c" file must have a function prototype for each function which is used in that ".c" file.
- `#include` can help manage your function prototypes.