

Lecture 2: C Programming Intro

Vikram Iyer

Administrative

- Are people having issues with Ed Discussion?
- Lab kit pickup
Mon-Fri: 9am-12pm and 1pm-5pm
- Partner matching form: <https://bit.ly/cse474-24sp-partners>
- Questions about course format or resources?



Last time: What are embedded systems?

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



Intro to C programming

- 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;
```

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

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:

```

Assembly Language

Assembler

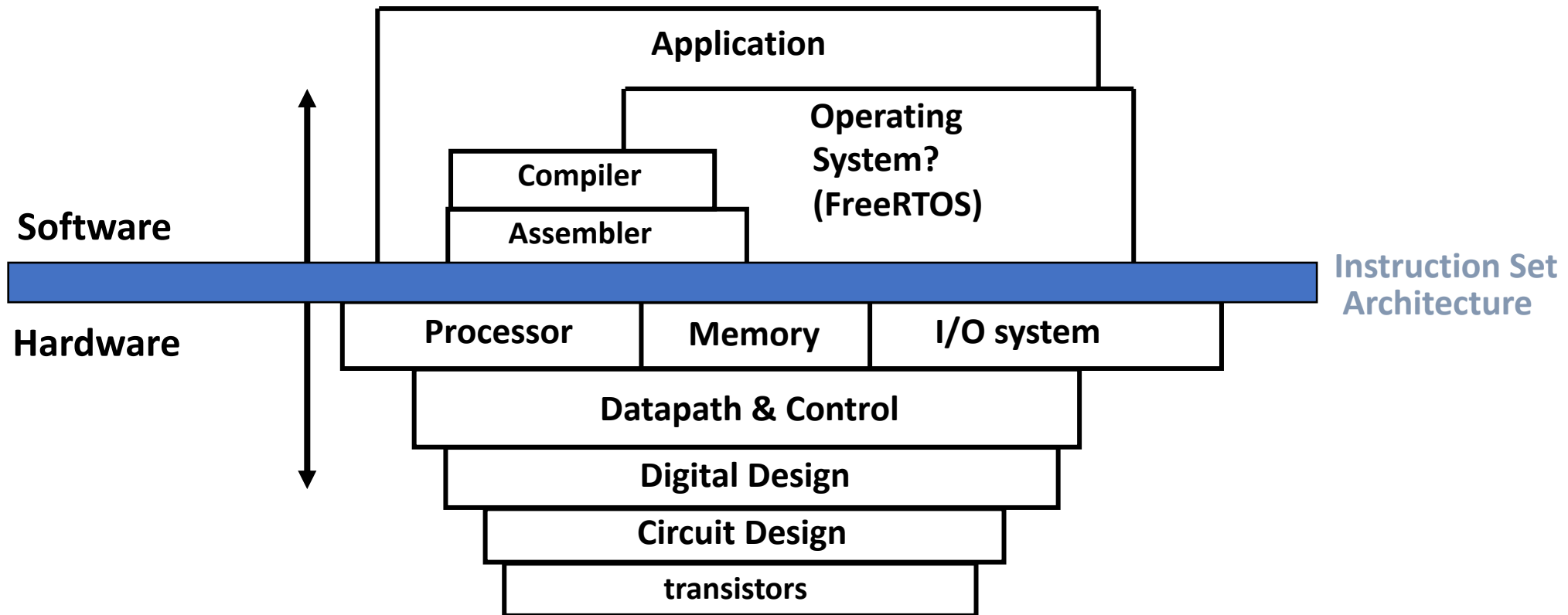
```

1000001101111100001001000001110000000000
0111010000011000
10001011010001000010010000010100
10001011010001100010010100010100
1000110100000100000000010
1000100111000010
110000011111101000011111
11110111011111000010010000011100

```

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	←	$Rd \cdot Rr$	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 (`Makefiles`) allow only modified files to be recompiled

C vs. Java™ Compilation Disadvantages

- 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 Disadvantages

- 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 Disadvantages

- 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

The image shows the Visual Studio Code website header and a screenshot of the application interface. The website header includes the Visual Studio Code logo, navigation links (Docs, Updates, Blog, API, Extensions, FAQ, Learn), a search bar for docs, and a blue 'Download' button. Below the header, a message states 'Version 1.77 is now available! Read about the new features and fixes from March.'

The main content area of the website features the text 'Code editing. Redefined.' and 'Free. Built on open source. Runs everywhere.' Below this is a blue button for 'Download Mac Universal Stable Build' and a link for 'Web, Insiders edition, or other platforms'. A disclaimer at the bottom of the website section reads: 'By using VS Code, you agree to its license and privacy statement.'

The screenshot of the VS Code application shows the 'EXTENSIONS: MARKETPLACE' sidebar on the left, listing various extensions like Python, GitLens, C/C++, ESLint, Debugger for C++, Language Support, vscode-icons, and Vetur. The main editor area displays a JavaScript file named 'blog-post.js' with code for a GraphQL query and a React component. The code includes imports for 'react', 'gatsby-image', and 'graphql', and defines a 'blogPost' component that uses 'data.cms.blogPost' and 'data.defaultStatus'. The bottom status bar shows the current file is 'blog-post.js' in the 'gatsby-graphql-app' workspace, with the cursor at 'Ln 6, Col 21'.

Writing a C program: `main()`

- Execution starts with first line of `main()` .
- To get the main function to accept arguments, use this:

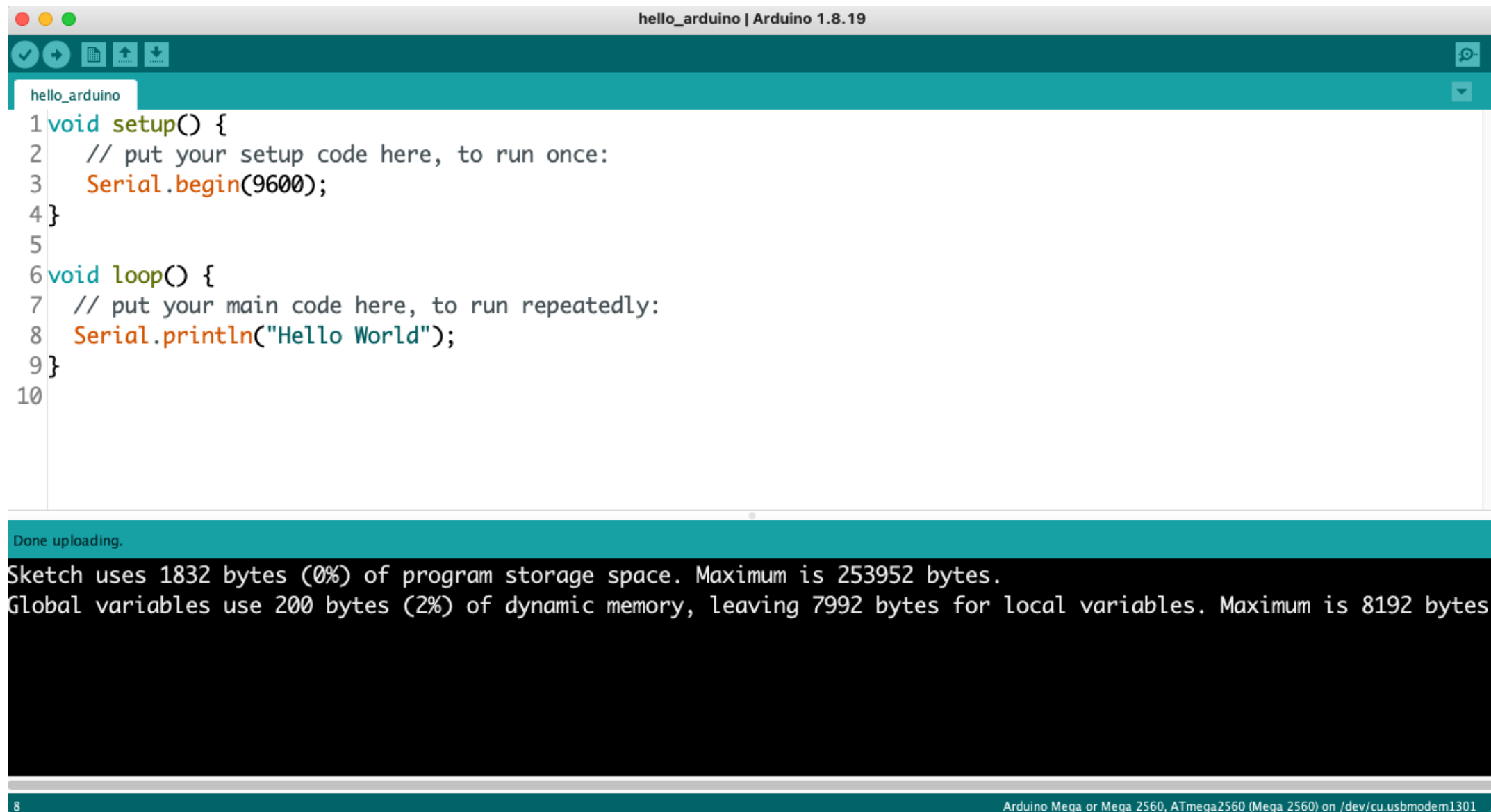
```
int main (int argc, char *argv[])
```
- What does this mean?
 - `argc` will contain the number of strings on the command line (the executable counts as one, plus one for each argument).
Example Here `argc` is 2: `unix% sort myFile`
 - `argv` is a pointer to an array containing the arguments as strings (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



The screenshot shows the Arduino IDE interface. The title bar reads "hello_arduino | Arduino 1.8.19". The code editor contains the following code:

```
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
```

Below the code editor, a status bar indicates "Done uploading." The console window shows the following output:

```
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
```

At the bottom of the IDE, the board and port are specified as "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()`
 - 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`

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

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

File: `functions.c`

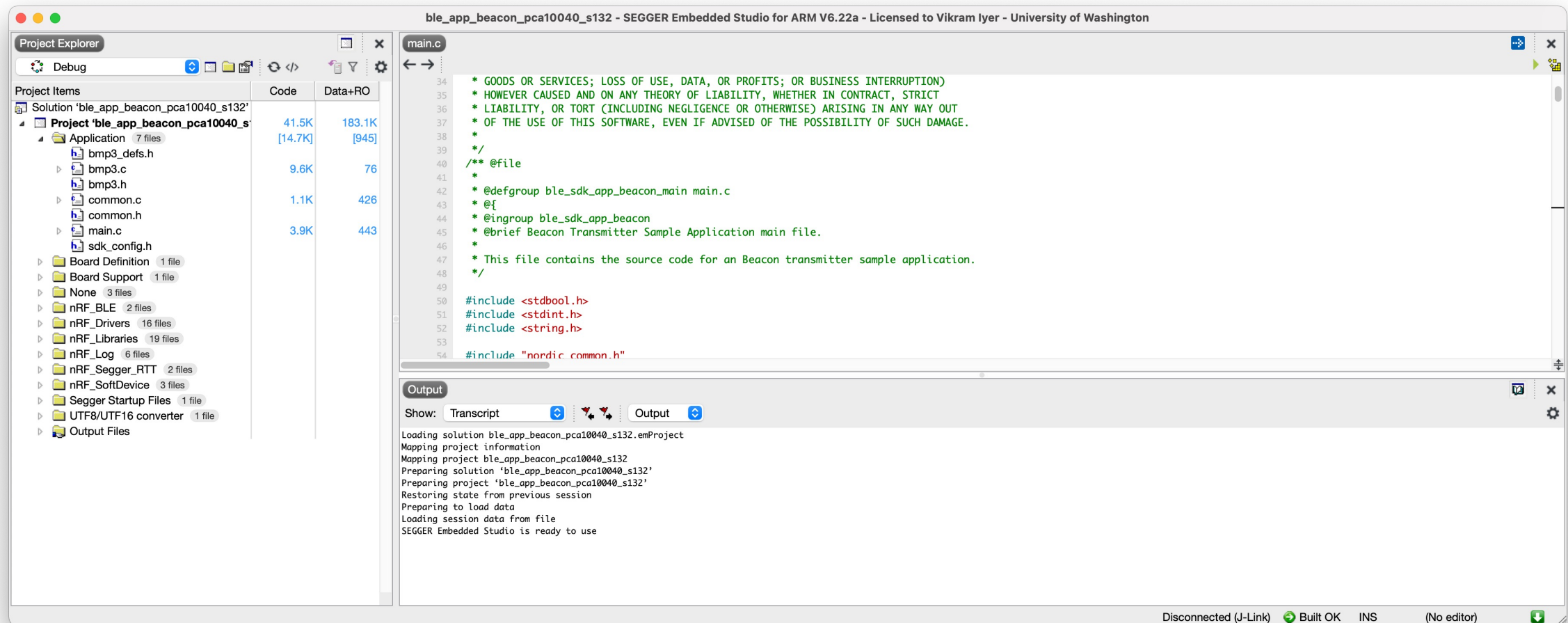
```
fcn2 ()
{
    ... code ...
}

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

... etc ...
```

Organizing C programs

This becomes very important in large projects



Organizing C programs

File: `main.c`

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

File: `functions.c`

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

How do we tell our compiler about these other functions?

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

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

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);

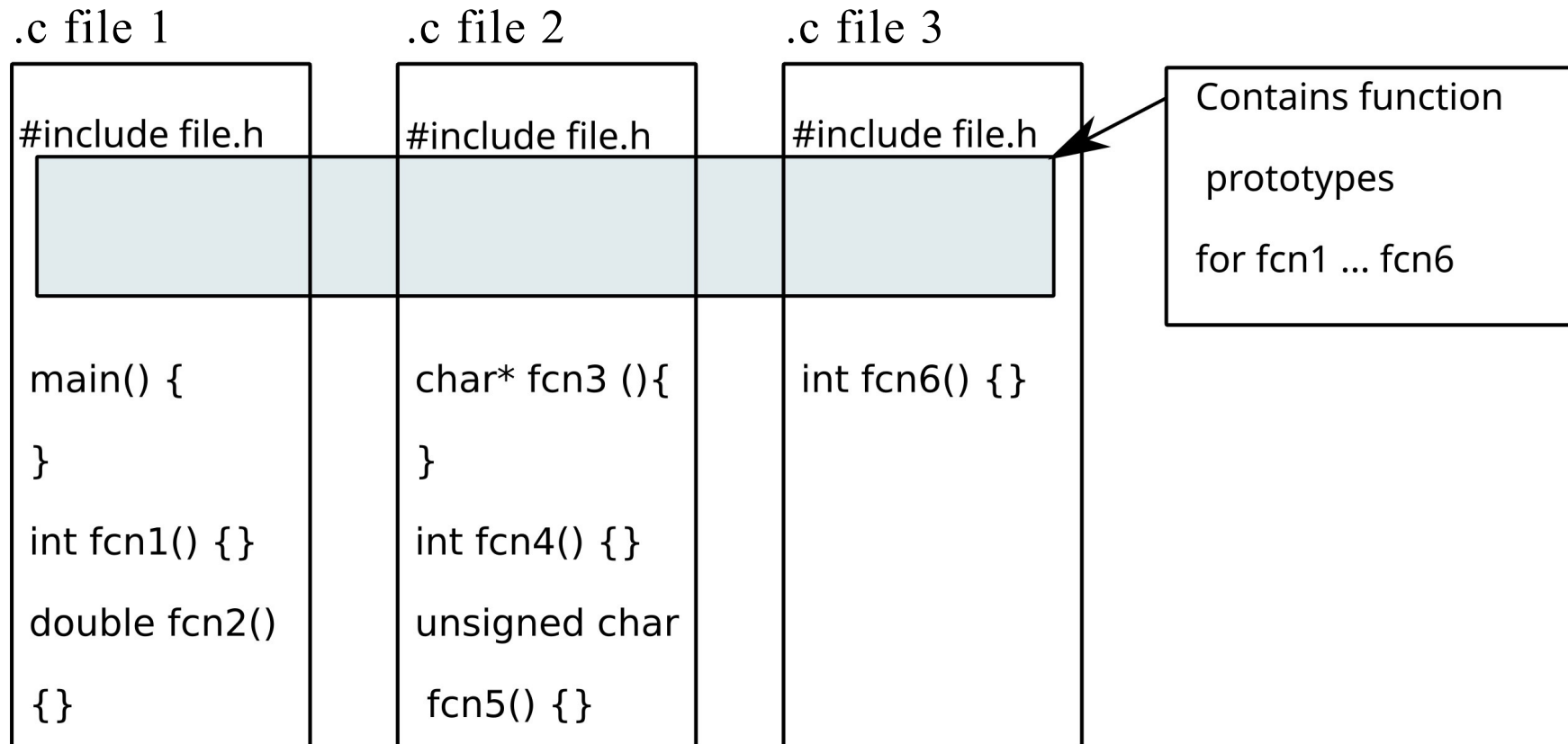
main()
{
    ... code ...
}
```

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

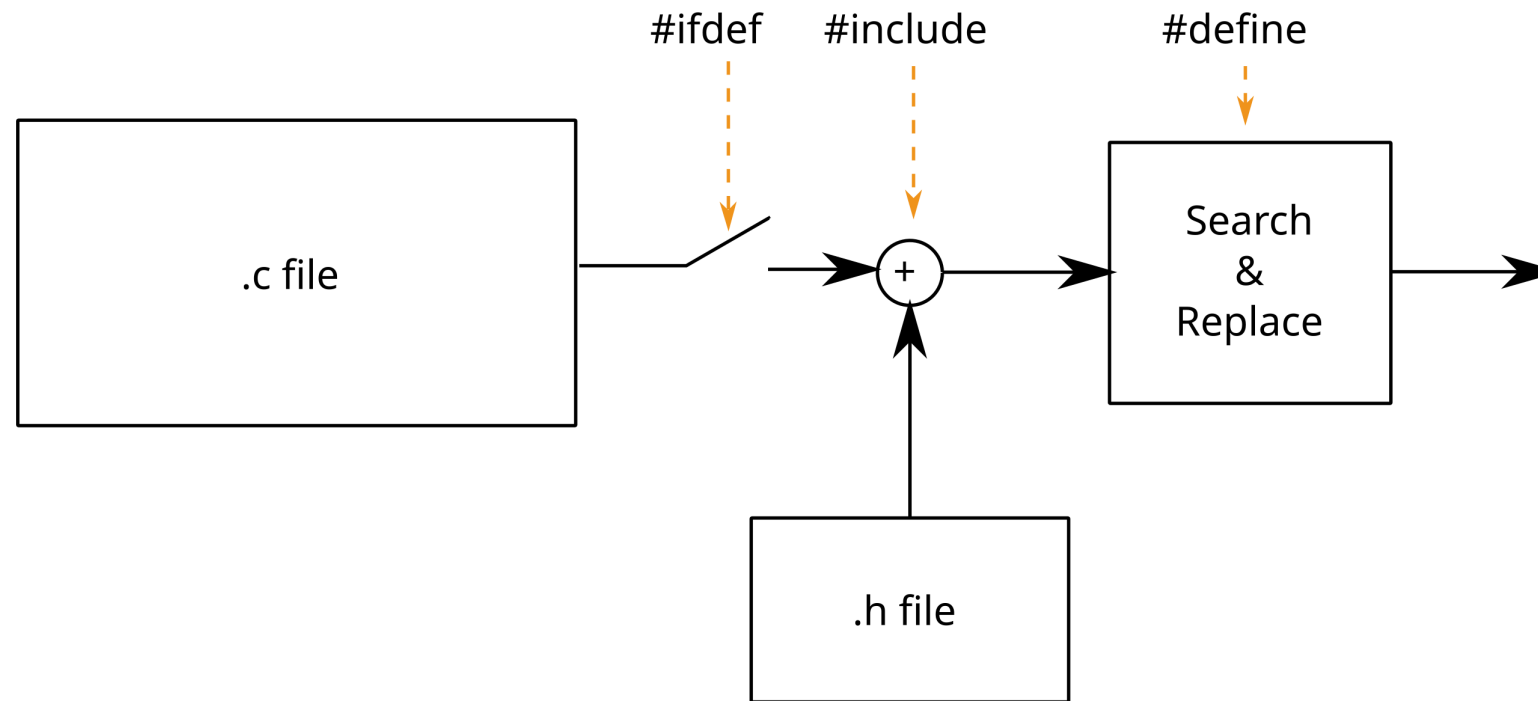
```
#define LINUX 1
//#define WINDOWS 1

...

#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.

Partner matching form

Fill this out if you need a lab partner and we'll match you with one

<https://bit.ly/cse474-24sp-partners>

