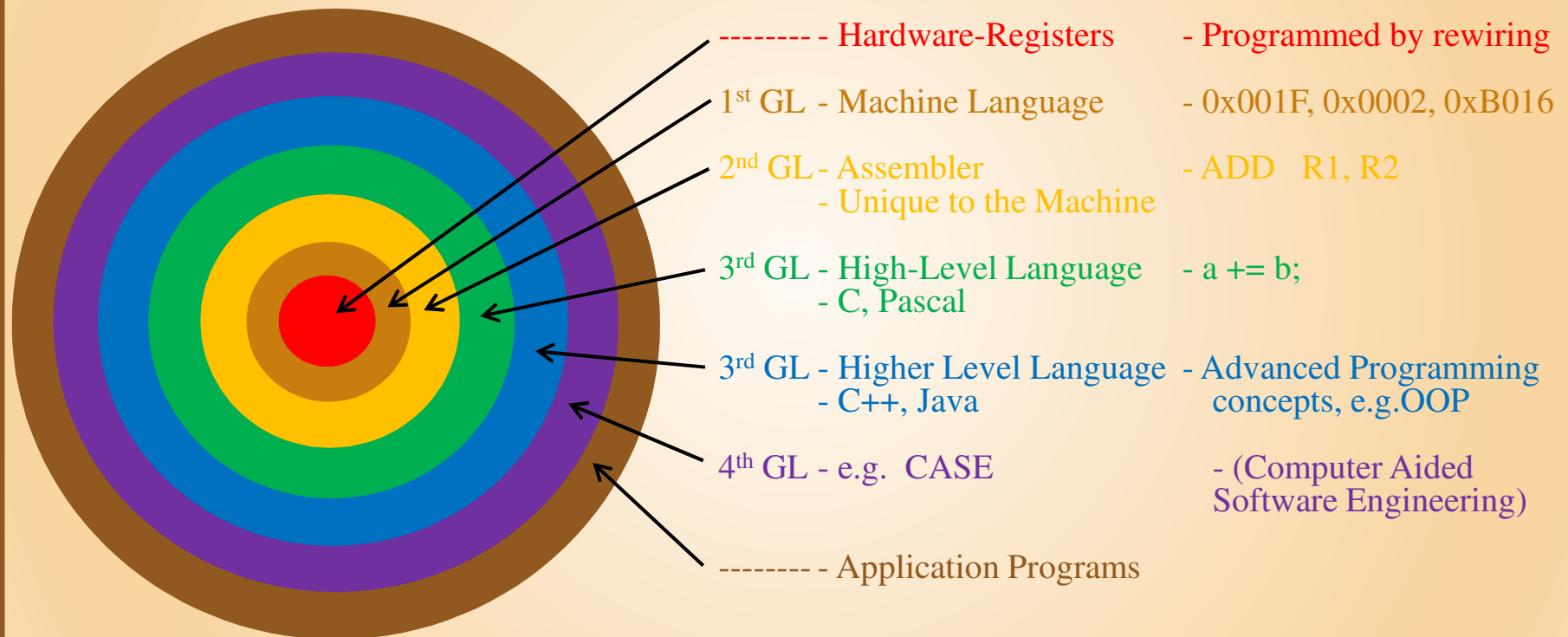


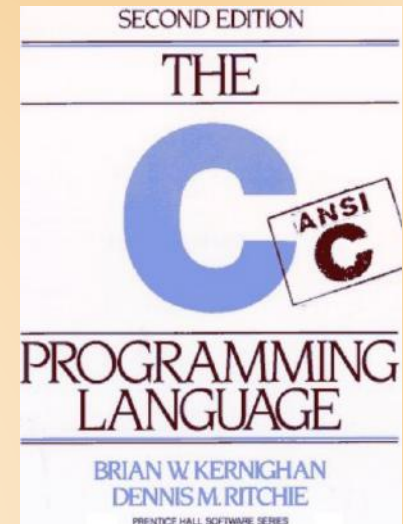
## Hardware Application Onion Model

- Figure 6.0 from textbook



## *C – Classic Reference:*

- [http://www.iups.org/media/meeting\\_minutes/C.pdf](http://www.iups.org/media/meeting_minutes/C.pdf)
- **Preface to the first edition (selections by Mr. Smith...)**
  - C is not a “very high level” language, nor a “big” one, not specialized to any particular area of application. But its generality makes it more convenient and effective for many tasks than supposedly more powerful languages.
  - C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie. The operating system, the C compiler, and essentially all UNIX applications programs (including all of the software used to prepare this book) are written in C. C is not tied to any particular hardware or system.
  - This book is meant to help the reader learn how to program in C. It contains a tutorial introduction to get new users started as soon as possible, separate chapters on each major feature, and a reference manual. For the most part, the examples are complete, real programs rather than isolated fragments.
  - The book is not an introductory programming manual; it assumes some familiarity with basic programming concepts like variables, assignment statements, loops, and functions. Nonetheless, a novice programmer should be able to read along and pick up the language, although access to more knowledgeable colleagues will help.



## **C - Variables, Controls, Arrays, Functions:**

- C is a language that bridges concepts in high-level programming and hardware.
- Assembly is low-level, you can nearly explicitly define actions on every cycle.
- Languages like Python nearly abstract hardware and hardware resource issues away completely.
- **In C, you have a lot of control over the processor.** You can even suggest what variables are stored in registers, but you don't have to code at the level that considers what is happening every clock cycle.
- **C can also be more dangerous.** C's type system and error checks exist only at compile-time. The compiled code runs in a stripped down run-time model with no safety checks for bad type casts, bad array indices, or bad pointers. There is no garbage collector to manage memory. Instead the programmer manages heap memory manually.
- **A good reference for getting started is:**
  - <http://cslibrary.stanford.edu/101/EssentialC.pdf>
- **-- And other references:**
  - <http://cslibrary.stanford.edu/101/EssentialC.pdf>
  - [http://publications.gbdirect.co.uk/c\\_book/](http://publications.gbdirect.co.uk/c_book/)

## *C and Java:*

- **C is a procedural language**
  - Problem solving centers on defining functions that perform a single service like `getValidInt( )`, `search( )` and `inputPersonData( )`.
  - Data is global or passed to functions as parameters
  - No classes
- **Java and C++ are object-oriented programming (OOP) languages**
  - Problem solving centers on defining classes that model “things” like Trucks, Persons, Marbles, Strings, and Candy Machine
  - Classes encapsulate data (instance variables) and code (methods)

## *If you know Java...*

- **C and Java syntax have much in common**
  - Some Data Types
  - Arithmetic operators
  - Logical Operators
  - Control structures
  - Other Operators
- We assume that you are proficient in Java in this course

## **Libraries:**

- **Because C is a procedural language, its library consists of predefined functions as opposed to having any classes as in other languages.**
  - Char/string functions (strcpy, strcmp)
  - Math functions (floor, ceil, sin)
  - Input/Output functions (printf, scanf)
- **On-line C/Unix manual --the “man” command**
  - Description of many C library functions and Unix commands
  - Usage: ‘man <function name>’ for C library functions or ‘man <command name>’ for Unix commands (example)
    - man printf
    - man dir
  - Search for applicable man pages using the “apropos” command or “man -k”
  - Learn to use the man command using “man man”



## **The C Standard:**

- **The first standard for C** was published by the American National Standards Institute (ANSI) in 1989 and is widely referred to as “ANSI C” (or sometimes C89)
- **A slightly modified version** of the ANSI C standard was adopted in 1990 and is referred to as “C90”. “C89” and “C90” refer to essentially the same language.
- In March 2000, ANSI adopted the **ISO/IEC 9899:1999** standard. This standard is commonly referred to as **C99**, and it is the current standard for the C programming language.
- **The C99 standard is not fully implemented in all versions of C compilers.**

## **C99 on GL:**

- The GNU C compiler on the GL systems (gcc version 4.1.2) appears to support several useful C99 features.
- These notes include those C99 features supported by gcc on GL since our course use that compiler.
- These features will be noted as C99 features when presented.

## *Hello World:*

- This source code is in a file such as the following hello.c:

```
/*  
file header block comment  
*/  
  
#include <stdio.h>  
  
int main( )  
  
{  
    // print the greeting ( // allowed with C99 )  
    printf( "Hello World\n" );  
    return 0;  
}
```

## Compiling on Unix:

- Traditionally the name of the C compiler that comes with Unix is “**cc**”.
- The UMBC GL systems use the “GNU Compiler Collection” named “**gcc**” for compiling C (and C++) programs.
- The default name of the executable program that is created by **gcc** is **a.out**  
`unix> gcc hello.c`

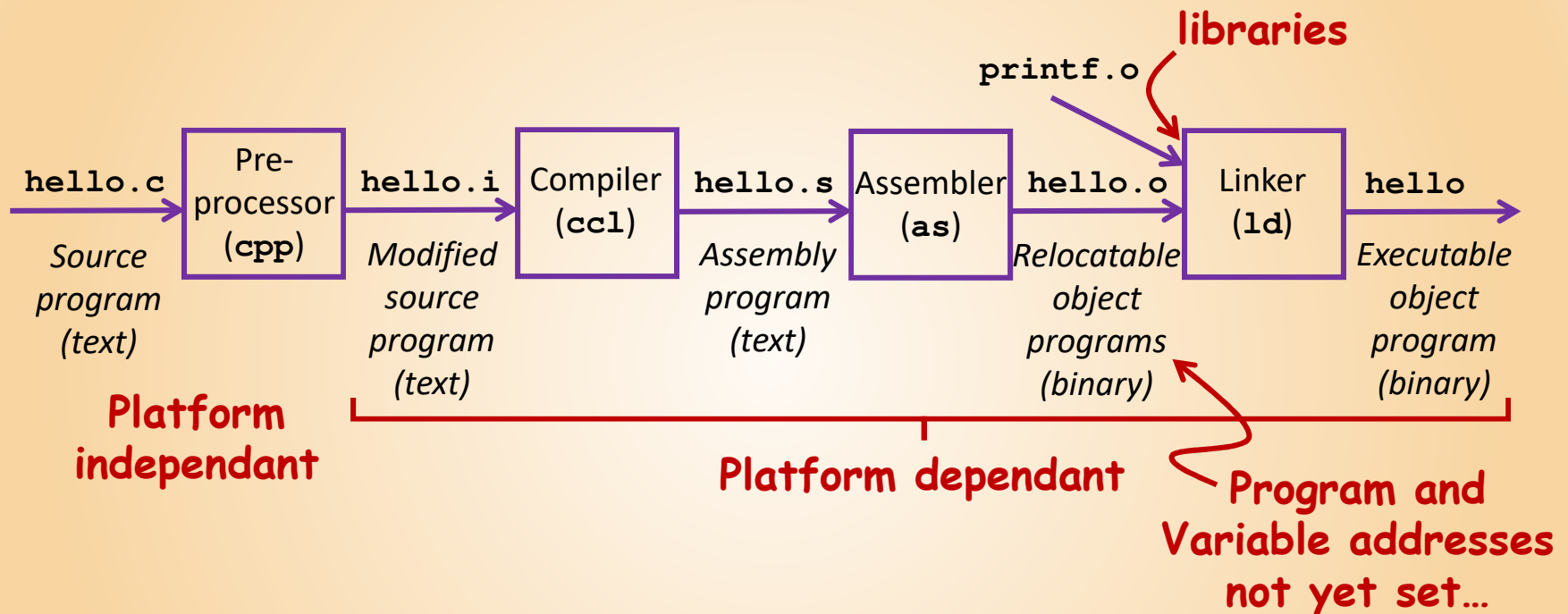
## Compiling Options:

- **-c**
  - Compile only (create a .o file), don't link (create an executable)  
`gcc -c hello.c`
- **-o filename**
  - Name the executable filename instead of a.out  
`gcc -o hello hello.c`
- **-Wall**
  - Report all warnings  
`gcc -Wall hello.c`
- **Use them together**  
`gcc -Wall -o hello hello.c`
- Note: the **-ansi** switch enforces the original ANSI C standard and disables C99 features.



## Compiling and Running a C Program:

```
Unix> gcc -Wall -o hello hello.c
```



➤ **Execute your program** by typing the name of the executable at the Unix Prompt:

```
Unix> hello
```

## *Compile time - Vocabulary:*

- **Preprocessor:** prepares file for compiler, handles duties like processing macros, sources selection, processing preprocessor directives (indicated by # in C) and file includes
- **Compiler:** converts nearly machine independent C code to machine-dependent assembly code
- **Assembler:** converts assembly language to machine language, but result is a relocatable object file, meaning addresses of code and variables have not all be resolved
- **Linker:** combines all object files and resolves addressing issues among them and determines final addresses for code and variables
- **Loader:** when we execute the program, loads the executable file into memory –it makes sure that main function is in a address that reflects the start of program.
  - More details on linker-loader:
    - <http://www.lurklurk.org/linkers/linkers.html>
    - <http://www.linuxjournal.com/article/6463>
- **Cross Compiler:** compiler that runs on one platform but outputs code for another target machine (our AVR code is compiled Intel Processor)

## **Identifiers:**

Identifiers are a name of a function or a variable

- ANSI/ISO C Standards
  - case sensitive
  - first character must be alpha or \_
  - May not be a C keyword such as return or int
  - No length limit imposed by standard (but read compiler documentation for limitations)
- Good Coding practices
  - Choose convention for capitalization of variables and functions
  - Symbolic constants should be all caps
  - Choose descriptive names over short names
  - Which variable name is most useful when reading code 3 years later or when somebody else has to?

	<b>temp1</b>	<b>TempDegC</b>
<b>t1</b>		
	<b>temperature1</b>	

- Treat identifiers as documentation, i.e. something for humans as much as for the compiler. Don't be lazy with naming, put effort into documentation.

## Assignments:

- Assignments set values to variables. They use the equal "=" character and end with a semicolon.

```
OutsideTempDegC  = 16;  
InsideTempDegC   = 20;  
InsideTempForNow = InsideTempDegC;
```

- What would be the line to show the temp in degrees Fahrenheit?

## Initialization:

- Initialization refers to the first assignment whether in declaration or afterword.
  - Until initialization, a variables should usually be considered uninitialized, meaning its "contents" are unknown/unspecified/garbage.
    - Exception: All objects with static storage duration, variables declared with the *static* keyword and global variables, are zero-initialized unless they have a user-supplied initialization value. Even so, it is a good practice to provide an explicit initialization as documentation that zero-initialization was intended.

note: initialization is not "free" in terms of run time or program space. It is equivalent to assembly ldi. In most cases initialization with a definition is a good idea. The compiler may add/remove it anyway as needed/unneeded.

## *Types: - Integral Data Types:*

- C data types for storing integer values are
  - **int** (the basic integer data type - int should be used unless there's a very good reason to use one of the others)
  - **short int** (typically abbreviated just as short)
  - **long int** (typically abbreviated just as long)
  - **long long int** (C99)
  - **char** (C does not have “byte”)
- Number of Bytes
  - **char** is stored in 1 byte
  - The number of bytes used by the other types depends on the machine being used.

## *Types: - Integral Specifiers:*

- Each of the integral types may be specified as either
  - **signed** (positive, negative, or zero)
  - **unsigned** (positive or zero only) (allows larger numbers)
  - **signed** is the default qualifier
- Much more on this later...
- Note: be sure to **pay attention to signed vs. unsigned** representations when transferring data between system. **Don't assume.**



## *Types - Integral Type Sizes:*

- The C standard is specifically vague regarding the size of the integral types
  - A **short int** must not be larger than an **int**.
  - An **int** must not be larger than a **long int**.
  - A **short int** must be at least 16 bits long.
  - An **int** must be at least 16 bits long.
  - A **long int** must be at least 32 bits long.
  - A **long long int** must be at least 64 bits long.
- The standard does not require that any of these sizes be necessarily different.
- So... **check the documentation for your compiler.** --Especially for embedded systems. --You need to know!

## *Types - typedef:*

- C allows variables to be declared using intrinsic types and type-synonyms defined using the typedef keyword.
  - Intrinsic Types
    - "Fundamental" or "built-in" types
      - Integral Types
        - e.g. int, char
      - Floating-Point Types (disused later in course)
  - Type synonyms (aliases) using typedef
    - Keyword typedef can be used to give a new name to an existing type
    - By convention we refer to this as "defining a type", but no new type is created, only a new name
    - Example:

```
typedef unsigned int my_type;  
my_type a;
```
    - especially useful for structures, to be covered more later

**Types–typedef–named types that make the size apparent should be used:**

- To avoid ambiguity of variable sizes on embedded systems, named types that make the size apparent should be used.
- C language allows defining user types. For this typedef keyword is used:  

```
typedef unsigned char byte;    //create byte type
typedef unsigned int word;     //create word type
```
- In other words defining custom types description structure is used:  

```
typedef standard_type custom_type;
```
- WinAVR compiler has predefined custom types:  

```
typedef signed char int8_t;           //located in header file inttypes.h
typedef unsigned char uint8_t;        //located in inttypes.h
typedef int int16_t;                   //located in header inttypes.h
typedef unsigned int uint16_t;         //located in inttypes.h
typedef long int32_t;                  //located in inttypes.h
typedef unsigned long uint32_t;        //located in inttypes.h
typedef long long int64_t;             //located in inttypes.h
typedef unsigned long long uint64_t;   //located in inttypes.h
typedef struct {int quot; int rem} div_t; //located in stdlib.h
```
- It is used for standard function ldiv();
- <http://winavr.scienceprog.com/avr-gcc-tutorial/more-about-c-types-in-avr-gcc.html>

## *"Defining" Variables:*

- C allows you to **declare** and **define** different kinds of variables.
- A **declaration** puts a variable's name in the namespace, so that it can be referred to, but no memory is allocated. Declaration sets the identifier (the name) and the type.
- A **definition** allocates the memory. The amount of memory needed depends on the variable type.
- A **initialization**, which is optional, sets an initial value to be stored in the variable.

## *C declaration, definition and initialization:*

- In C, a combined declaration and definition is typical and is the default.

type	name	
char	example1;	//definition and declaration
int	example2 = 5;	//def. and decl. and init.
void	example3(void)	//def. and decl. of function


```
{  
    int x =7;  
}
```

Pasted from [http://en.wikipedia.org/wiki/Declaration\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Declaration_(computer_science))

- The extern keyword may be added to declare that the definition will be provided elsewhere.

```
extern char    example1;  
extern int     example2;  
void          example3(void);
```

For a function, simply not providing the definition suffices, and in that case the declaration is called a prototype.



Pasted from [http://en.wikipedia.org/wiki/Declaration\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Declaration_(computer_science))



## *Integral Variables:*

- Each of the following is a valid variable declaration:

```
int age = 42;  
signed int age = -33;  
long area = 123456;  
short int height = 4;  
unsigned char IQ = 102;  
unsigned int length = 8282;  
unsigned long int SATscore = 800;
```

### *Floating Point Data Types:*

- C data types for storing floating point values (those with a decimal part) are
  - **float**, the smallest floating point type
  - **double**, a larger type with a larger range of values
  - **long double**, an even larger type with an even large range of values
- **double** is typically used for all floating point values unless there's a compelling need to use one of the others
- Floating point variables may store integer values

### *Size of Floating Point Type:*

- A **double** variable can be marked as being a **long double**, which the compiler may use to select a larger floating point representation than a plain **double**.
- The standard is unspecific on the relative sizes of the floating point values, and only requires a **float** not to be larger than a **double**, which should not be larger than a **long double**.

### *Floating Point Data Types:*

- Each of the following is a valid floating point variable declaration:

```
float Avg = 10.6;  
double Median = 88.54;  
double HomeCost = 10000;
```

## Character Data Types:

- C has just one data type for storing characters: char, which is just one byte
- Because a char is just one byte, C only supports the ASCII character set (more on this later)

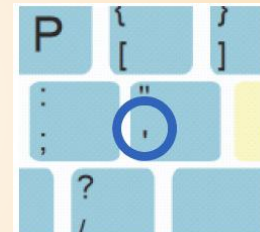
- Example Assignments:

```
char x = 'A';
```



single quote

'A' is a literal with value 65  
(why is the value 65?)



not



- Equivalent example

```
char x = 65;
```

## *'const' Qualifier:*

- Any of the variable types discussed may be qualified as **const**.
- **const** variables may not be modified by your code. Any attempt to do so will result in a compiler error.
- Since they may not be modified, **const** variables must be initialized when declared

```
const double PI = 3.14159;  
const int MyAge = 55;
```

- Not Allowed:

```
const float PI;  
PI = 3.14159;
```

## *sizeof( )*

- Because the sizes (number of bits/bytes) of the C data types are vaguely specified, C provides the **sizeof( )** operator to determine the size of any data type (**in bytes**).
- **sizeof( )** should be used everywhere the size of a data type is required so that your code is portable to other hardware on which the size of the data types may be different.

## **Variable Declaration:**

- ANSI C requires that all variables be declared at the beginning of the “block” in which they are defined, before any executable line of code.

- C99 allows variables to be declared anywhere in the code (like Java and C++)

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

```
int temp;  
temp = a  
a = b  
b = temp;
```

executable code

The older standards required  
this declaration to be at the top

- In any case, variables must be declared before they can be used.

## **Boolean Data Type:**

- ANSI C has no Boolean type
- The C99 standard supports the Boolean data type
- To use **bool**, **true**, and **false**, your code must include **<stdbool.h>**

```
#include <stdbool.h>  
bool isRaining = false;  
if ( isRaining )  
    printf( "Bring your umbrella\n");
```



## Arithmetic Operators:

- Arithmetic operators are the same as Java
  - = (is used for assignment)
  - +, - (plus, minus)
  - \*, /, % (times, divide, mod)
  - ++, -- (increment, decrement (pre and post))
- Combinations are the same
  - +=, -= (plus equal, minus equal)
  - \*=, /=, %= (times equal, divide equal, mod equal)
- Arithmetic Practice
- Assignment Practice

Example:  
a += 2;  
same as  
a = a+2;

## Logical Operators:

- Logical operators are closely similar in C and Python and result in a Boolean value.

**&&** (and)

**||** (or)

**==, !=** (equal and not equal)

**<, <=** (less than, less than or equal)

**>, >=** (greater than, greater than or equal)

- Integral types may also be treated as Boolean expressions
  - Zero is considered “false”
  - Any non-zero value is considered “true”

- Boolean  
Logic  
Practice

another **VERY** common error:

```
if (x = 0) {  
}
```

**INSTEAD**, you want:

```
if (x == 0) {  
}
```

Example:

```
x = -1;  
if (! (x) ) {  
    x = x+1;  
}
```

Result:

```
x = -1;
```

**Materials to Review for Quiz 2:**

class: 5, slide: 3 → AVR IO Ports - Notes  
class: 5, slide: 5 → AVR IO Ports - Pull Up Resistors  
class: 5, slide: 7-8 → AVR IO Ports – Programming I/O Ports –Assembly  
class: 6, slide: 1 → Overview, Addressing Modes in General  
class: 6, slide: 2-10 → Addressing Modes - Matching...  
class: 7, slide: 3-6 → Useful Assembler Features  
class: 7, slide: 10 → Stack and Functions  
class: 7, slide: 13-15 → Implementing Delays  
class: 7, slide: 16 → Conditional, Unconditional jumps  
class: 8, slide: 1 → Hardware Application Onion Model  
class: 8, slide: 5 → Libraries  
class: 8, slide: 9 → Compiling and Running a C Program  
class: 8, slide: 10 → Compile time - Vocabulary  
class: 8, slide: 13 → Types: Integral Data Types, Specifiers  
class: 8, slide: 17-19 → 'Defining' Variables  
class: 8, slide: 24-25 → Operators