ECE3411 – Fall 2016

Lecture 1a.

Course Outline
Introduction to C-Programming

Marten van Dijk

Department of Electrical & Computer Engineering

University of Connecticut

Email: vandijk@engr.uconn.edu

UCONN

Course Outline

- Syllabus
- Calendar

- This class requires you to be on top of the material by reading and studying book chapters and slides, coding lab solutions, and summarizing the execution flow of coded solutions in (bulleted + diagram) reports.
- We asks you to come prepared so that you can answer 5-min questions at the start of each class, write reports, and study for 7 quizzes + lab tests. This forces you to follow a study discipline with a number of successful study strategies.

Method of Study

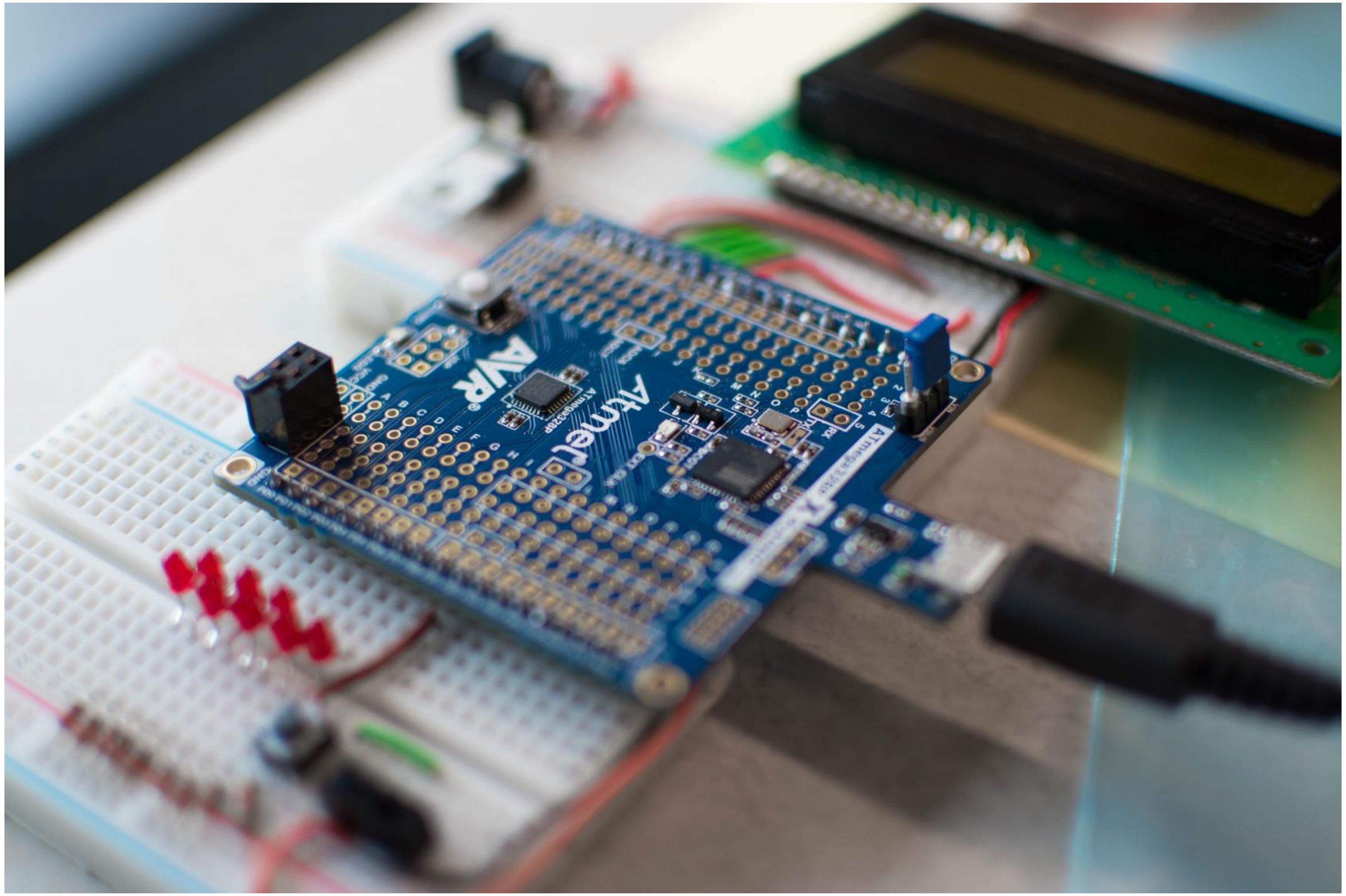
- Read the solution, cut and paste into code and get it to work on the MCU, parse the code and understand it
  - Students are struggling the most – bottom of class

- Read the solution, now try coding yourself the whole solution and where stuck or where it does not work refer back to the solution
  - The average/middle student

- First code a solution yourself, get it to work, finally read the posted solution
  - Students are on top of the material – top of class

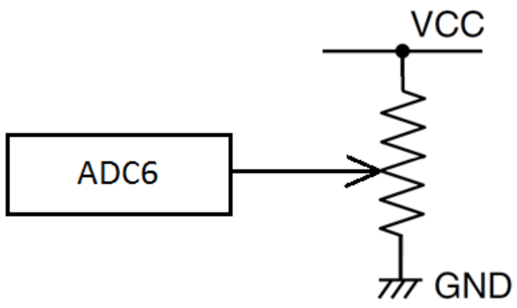
Verified by asking students about there study method and correlating the answers with their grades

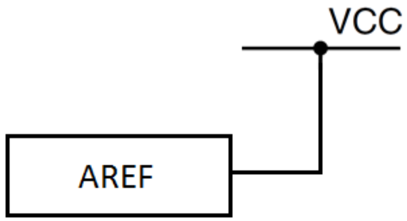
ATmega328P Development Board

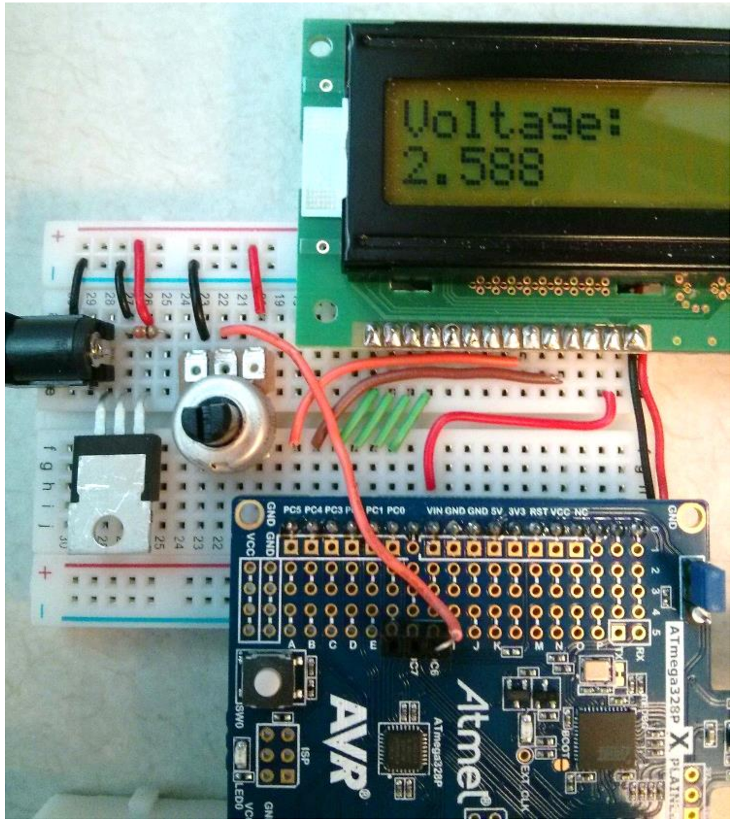


Interesting bits (1): Interfacing Analog Sensors

- Real world is Analog, whereas our computing systems are Digital
- Interfacing of Analog Sensors with the MCU is crucial component of Embedded Systems design
- In this course, you'll interface Temperature and Ambient Light sensors with the MCU to perform various control tasks.

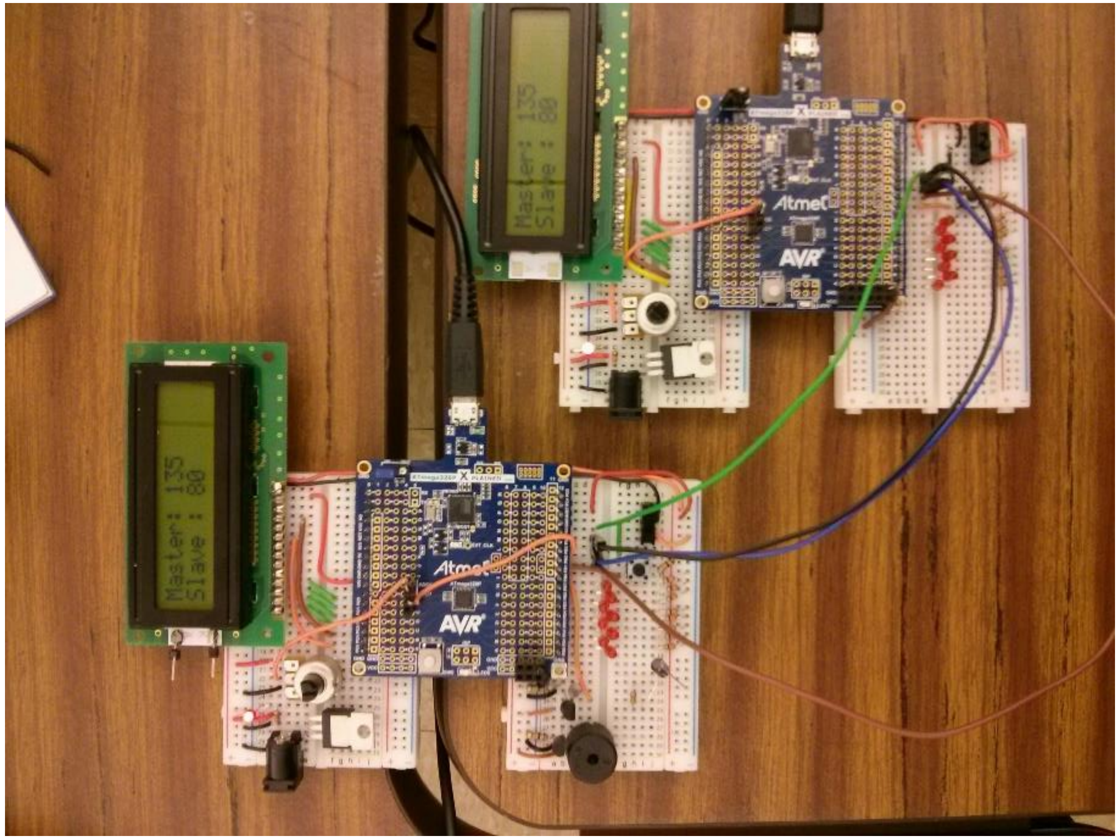






Interesting bits (2): Communication Across Devices

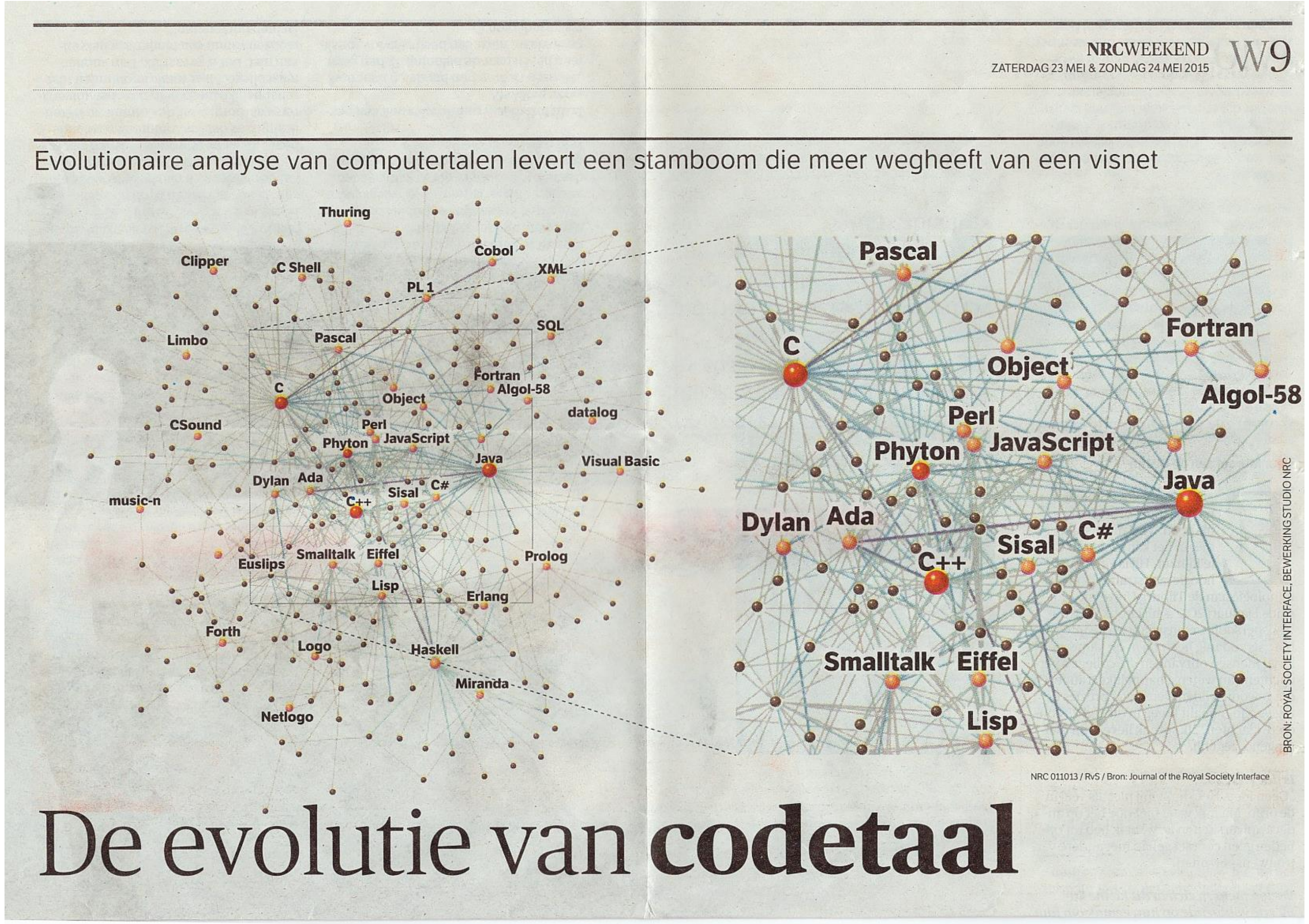
- Communication across devices is a vital part of Embedded Systems
- You will explore two important communication protocols namely
  - SPI
  - I2C



Interesting bits (3): Playing with Timers & Interrupts

- A lot of Embedded Systems handle time-triggered and time-critical tasks!
- Timers of Microcontrollers serve several useful purposes related to embedded system tasks.
- We will be designing:
  - Timer based applications such as Stopwatch
  - Multi-tasking applications with time-triggered tasks
  - Pulse Width Modulation applications

Programming Languages



Introduction to C-Programming

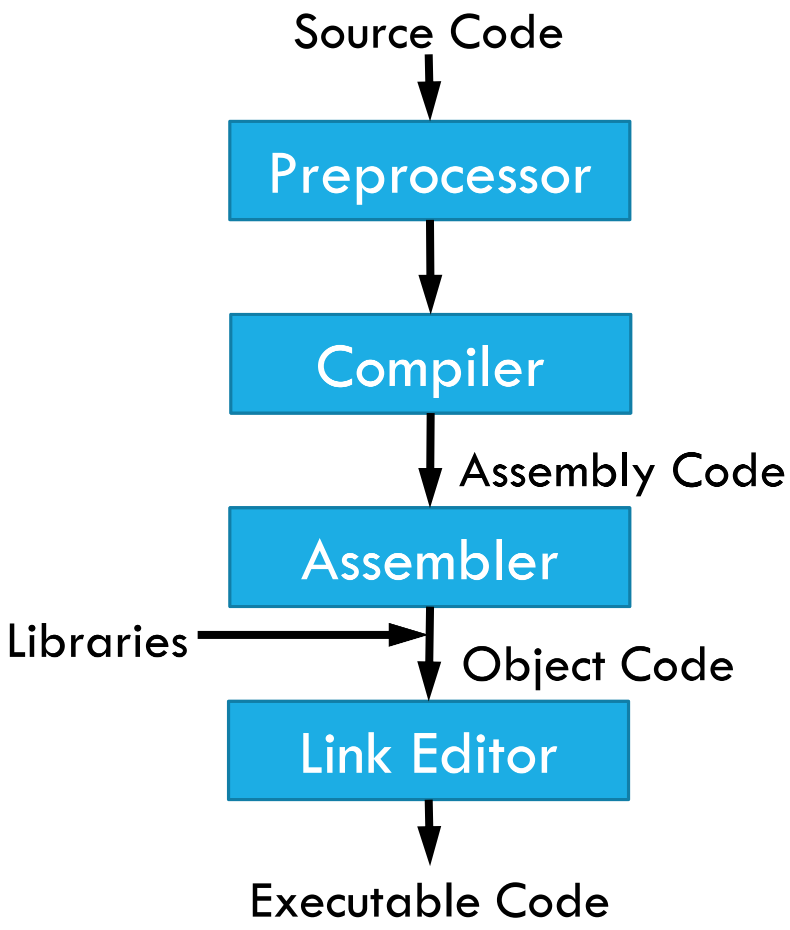
- The C programming language was designed by Dennis Ritchie at Bell Laboratories in the early 1970s.

- C is mother language of all programming language used for systems programming.

- It is procedure-oriented and also a mid level programming language.

The C Compilation Model

- **The Preprocessor** accepts source code as input and is responsible for
  - Removing comments
  - Interpreting special preprocessor directives denoted by #.
  - Examples: **#include <stdio.h>** , **#define begin {** , **#define end }**
- **The C compiler** translates source to assembly code.
- **The assembler** creates object code.
- **The Link Editor** combines any library functions referenced in the source code with the **main()** function to create an executable file.



A simple C program : Printing ‘Hello World’

```c
#include <stdio.h>
int main ()
{
    printf(“Hello World”);
    return 0;
}
```

stdio.h

```c
#ifndef     _STDIO_H_
#define     _STDIO_H_
.....
#include <sys/cdefs.h>
#include <machine/ansi.h>
.....
int             printf(const char *, ...);
int             scanf(const char *, ...);
...
```

- **#include <stdio.h>**
  - Preprocessor directive which loads contents of a certain file
  - `<stdio.h>` allows standard input/output operations
- **int main ()**
  - **main** is the driver function of a c program where execution starts.
  - **int** means that **main** returns an integer value
- Bodies of all functions must be contained in curly braces
  - ‘ { ’ start of function
  - ‘ } ’ end of function
- **printf(“Hello World”);**
  - Prints the string of characters within quotes
  - Entire line is called a statement
  - All statements must end with a semicolon
- **return 0;**
  - A way to exit a function
  - Here it means that the program terminated normally

Another ‘Hello World’ Program

```
#include <stdio.h>
#define   begin      {
#define   end        }
int main ()
begin
    printf("Hello World");
    return 0;
end
```

- You can define your own macros
- **begin** represents the opening brace ‘{’
- **end** represents the closing brace ‘}’
- The body of **main ()** can be enclosed in **begin** and **end**
- However, the recommended way of enclosing the function body is to use the braces ‘{ }’
- You can define other macros as well, e.g.
  - #define   MAX_ARRAY_SIZE   100

Tokens in C

- Keywords
  - These are reserved words of the C language.
  - For example int, float, if, else, for, while etc.
- Identifiers
  - An Identifier is a sequence of letters and digits, but must start with a letter.
  - Identifiers are used to name variables, functions etc.
  - Identifiers are case sensitive.
  - Valid: Root, _getchar, __sin, x1, x2, x3, x_1, If
  - Invalid: 324, short, price$, My Name
- Constants
  - 13, 'a', 1.3e-5 etc.
- String Literals
  - A sequence of characters enclosed in double quotes as "…".
  - For example "13" is a string literal and not number 13.
  - 'a' and "a" are different.
- Operators
  - Arithmetic operators: +, -, *, / ,%
  - Logical operators: ||, &&, !
- White Spaces
  - Spaces, new lines, tabs, comments (A sequence of characters enclosed in /* and */) etc.
  - These are used to separate the adjacent identifiers, keywords and constants.

Basic data types

char	Stored as 8 bits. Unsigned 0 to 255. Signed -128 to 127.
short int	Stored as 16 bits. Unsigned 0 to 65535. Signed -32768 to 32767.
int	Same as either short int or long int
long int	Stored as 32 bits. Unsigned 0 to 4294967295. Signed -2147483648 to 2147483647
float	Approximate precision of 6 decimal digits (single precision).
double	Approximate precision of 14 decimal digits (double precision).

Constants

- Numerical Constants
  - Constants like 12, 253 are stored as int type (No decimal point).
  - Numbers with a decimal point (21.53) are stored as float or double.
- **Character and string constants**
  - **‘c’** , a single character in single quotes are stored as char.
  - Some special character are represented as two characters in single quotes.
    - **‘\n’** = newline,
    - **‘\t’**= tab,
    - **‘\\’** = backlash,
    - **‘\”’** = double quotes.
  - A sequence of characters enclosed in double quotes is called a string constant or string literal.
    - For example : “**Hello**”

Variables

- Variable names correspond to locations in the computer's memory
- Every variable has a name, a type, a size and a value
- Naming a Variable
  - Must be a valid identifier
  - Must not be a keyword
  - Names are case sensitive
- Declaring a Variable
  - Each variable used must be declared. Example : `data-type var1, var2,…;`
  - Declaration announces the data type of a variable and allocates appropriate memory location.
  - Initializing value to a variable in the declaration itself: `data-type var = expression;`
  - Examples: `int sum = 0; char newLine = '\n'; float epsilon = 1.0e-6;`

Global and Local variables

- Global Variables
  - These variables are declared outside all functions.
  - Life time of a global variable is the entire execution period of the program.
  - Can be accessed by any function defined below the variable's declaration, in a file.
- Local Variables
  - These variables are declared inside some functions.
  - Life time of a local variable is the entire execution period of the function in which it is defined.
  - Cannot be accessed by any other function.
  - In general variables declared inside a block are accessible only in that block.

Example of global and local variable

```
/* Compute Area of  a circle */
#include <stdio.h>
float pi = 3.14159;  /* Global variable */

int main() {
  float rad;   /* Local variable*/

  printf( “Enter the radius “ );
  /* scanf obtains a value from user */
  /* Value is stored in rad */
  /* %f indicates that value should be float */
  scanf(“%f” , &rad);

  if ( rad > 0.0 ) {
    float area = pi * rad * rad;
    printf( “Area = %f\n” , area );
  }
  else {
    printf( “Negative radius\n”);
  }
   return 0;
}
```

Arithmetic Operators

- A = B → Assignment: A gets the value of B
- A + B → Add A and B together
- A – B → Subtract B from A
- A * B → A multiplied by B
- A / B → A divided by B
- A % B → Modulo: Integer remainder of A/B

Example:

```
int A = 11;
int B = 4;
int X = A / B;      // X gets the value 2. Since X is an integer, the fractional part is ignored.
int Y = A % B;      // Y gets the value 3 since A=BX+Y
```

Comparison Operators

- A == B → A is equal to B?
- A != B → A is NOT equal to B?
- A > B → A is greater than B?
- A < B → A is less than B?
- A >= B → A is greater than/equal to B?
- A =< B → A is less than/equal to B?

Logical Operators

Logical Operators map the inputs to either **TRUE** (Logical 1) or **FALSE** (logical 0)

These operators result in a single bit output

- !A → **NOT** A
- A && B → A **AND** B
- A || B → A **OR** B

Example:

```
if (A || (B && C) || !D)
{
   //do something;
}
```

if statement is only satisfied if

- A is logical high **OR,**
- B **AND** C are logical high **OR,**
- D is logical low.

Bitwise Operators

Bitwise operators map input bit vectors to the same sized output bit vector

- ~A → Bitwise complement of A
- A & B → Bitwise AND of A and B
- A | B → Bitwise OR of A and B
- A ^ B → Bitwise XOR of A and B
- A << B → Bitwise left shift A by B positions
- A >> B → Bitwise right shift of A by B positions

Bitwise Operators Examples

Let A = 0b11 and B = 0b01 then

- A represents the bit vector 11
- B represents the bit vector 01

~A	= 0b00		
A & B	= 0b11 & 0b01	= 0b01	
A | B	= 0b11 | 0b01	= 0b11	
A ^ B	= 0b11 ^ 0b01	= 0b10	
A << B	= 0b11 << 0b01	= 0b11 << 1	= 0b10
A >> B	= 0b11 >> 0b01	= 0b11 >> 1	= 0b01

We use bitwise operators frequently to manipulate the register values.

Prefix & Postfix Increment/Decrement

- ++A → The value of A is incremented before assigning it to variable A
- --A → The value of A is decremented before assigning it to variable A
- A++ → The value is incremented after assigning it to the variable A
- A-- → The value is decremented after assigning it to the variable A

Pre/Post Increment Examples

```
int x = 0;
while(++x < 5)
{
   printf("%d ", x);
}
```

- This prints 1, 2, 3, 4
- x is incremented BEFORE the comparison. Since 1 is less than 5, a '1' is printed. This is repeated until x = 4.
- Then the condition for the while loop fails, since x will be assigned a value of 5 before the values are compared.

```
int x = 0;
while(x++ < 5)
{
   printf("%d ", x);
}
```

- This prints 1, 2, 3, 4, 5.
- x is incremented AFTER the comparison, therefore, it meets the criteria of the while loop until x = 5.

Compound Assignments

- A += B → A = A + B
- A -= B → A = A – B
- A *= B → A = A * B
- A /= B → A = A/B
- A %= B → A = A%B
- A &= B → A = A&B
- A |= B → A= A|B
- A <<= B → A = A<<B
- A >>= B → A = A>>B

Control Structures: if/else statement

```
if(expression)
        <statement>
```

```
if(expression)
{
        /* Block of statements */
}
```

```
if(expression){
        /* Block of statements */
} else {
        /* other statements */
} else if (expression) {
        /* other statements */
} else if (..){
        /* ... */
}
```

- if statement can be used to execute some code if the condition in the expression is met.
- It can be used to execute a single code statement or a block of statements.
- if/else statement defines the alternate code to execute if the if-condition is not met.
- Note: if/else statements can be strung together with more if/else statements to add conditions to the 'else' parts.

Control Structures: switch statement

```
switch (<expression>)
{
   case <label1> :
      <statements 1>
       break;
   case <label2> :
      <statements 2>
      break;
   default :
      <statements 3>
}
```

- Used as a substitute for lengthy if statements that look for several conditions of some variable.

Control Structures: Loops

```
while ( <expression> )
{
    <statements>
}
```

```
for ( <expression1>; <expression2>; <expression3> )
{
    <statements>
}
```

```
do
{
    <statements>
}
while ( <expression> );
```

- while loop: While the condition in the expression statement is true, execute the statements in the loop.
- for loop: Similar to the while loop. expression1 initializes a variable, expression2 is a conditional expression, expression3 is a modifier, like an increment (x++).
- do-while loop is similar to while loop. It ensures that the block of statements is executed at least once.

for Loop Example

Temperature units conversion from Fahrenheit to Celsius:

```
#include <stdio.h>
int main() {
        int f;
        for (f=0; f <= 300; f += 20) {
                printf("%3d %6.1f \n", f, (5.0 / 9.0) * (f – 32.0));
        }
        return 0;
}
```

- %3d
  - % means “Print a variable here”
  - 3 means “Use at least 3 spaces to display, padding as needed”
  - d means “The variable will be an integer”
- %6.1f means “Print a float using 6 digits and round up to 1 decimal digit”.

Interesting Fact:

- To approximate Celsius from Fahrenheit in your head:
  - Subtract 32 from F
  - Take half of the result and increase it by 10%

Conditional Expressions

- Conditional expressions

```
expr1? expr2 : expr3;
```

- If `expr1` is true then execute `expr2` else execute `expr3`

Example:

```
for (int i=0; i<n; i++){
        printf("%d %c", a[i], (i%10==9 || i==(n-1))? '\n' : ' ');
}
```

Break and Continue statements

- **break** is used to terminate a loop immediately.
- **continue** is used to skip the subsequent statements inside the loop.

Examples:

```
while(test expression){
    <statements>
    if(test expression)
        break;
    <statements>
}
```

```
while(test expression){
    <statements>
    if(test expression)
        continue;
    <statements>
}
```

Type conversion

- The operands of a binary operator must have the same type and the result is also of the same type.
- Integer division: `c = (9 / 5)*(f - 32)`
- The operands of the division are both int and hence the result also would be int.
- For correct results, one may write `c = (9.0 / 5.0)*(f - 32)`
- In case the two operands of a binary operator are different, but compatible, then they are converted to the same type by the compiler. The mechanism (set of rules) is called Automatic Type Casting.

```
c = (9.0 / 5)*(f - 32)
```

- It is possible to force a conversion of an operand. This is called Explicit Type casting.

```
c = ((float) 9 / 5)*(f - 32)
```

Functions

- Functions are blocks of code that perform a number of pre-defined commands to accomplish something productive.
  - Library Functions
  - User Defined Functions
- Function prototypes are usually declared in the header files.
- General format for a function prototype

```
return-type function_name ( arg_type arg1, ..., arg_type argN );
```

- General format for a function body

```
return-type function_name ( arg_type arg1, ..., arg_type argN )
{
      /* Code for function body  */
}
```

Functions Example

```
#include <stdio.h>
int mult ( int x, int y );              // Function Prototype

int main()
{
    int x, y, z;
    printf( "Please input two numbers to be multiplied: " );
    scanf( "%d", &x );                  // Call to a library function
    scanf( "%d", &y );                  // Call to a library function
    z = mult( x, y );                   // Call to a user-defined function
    printf( "The product of your two numbers is %d\n", z );
}

/* Function Body */
int mult (int x, int y)
{
    return x * y;
}
```