The background of the slide is a close-up, slightly blurred photograph of a green printed circuit board (PCB). A large, square, tan-colored integrated circuit (microcontroller) is the central focus, with its numerous pins visible. Other components like smaller chips, capacitors, and solder joints are also visible on the board.

MCT 4334

Embedded System Design

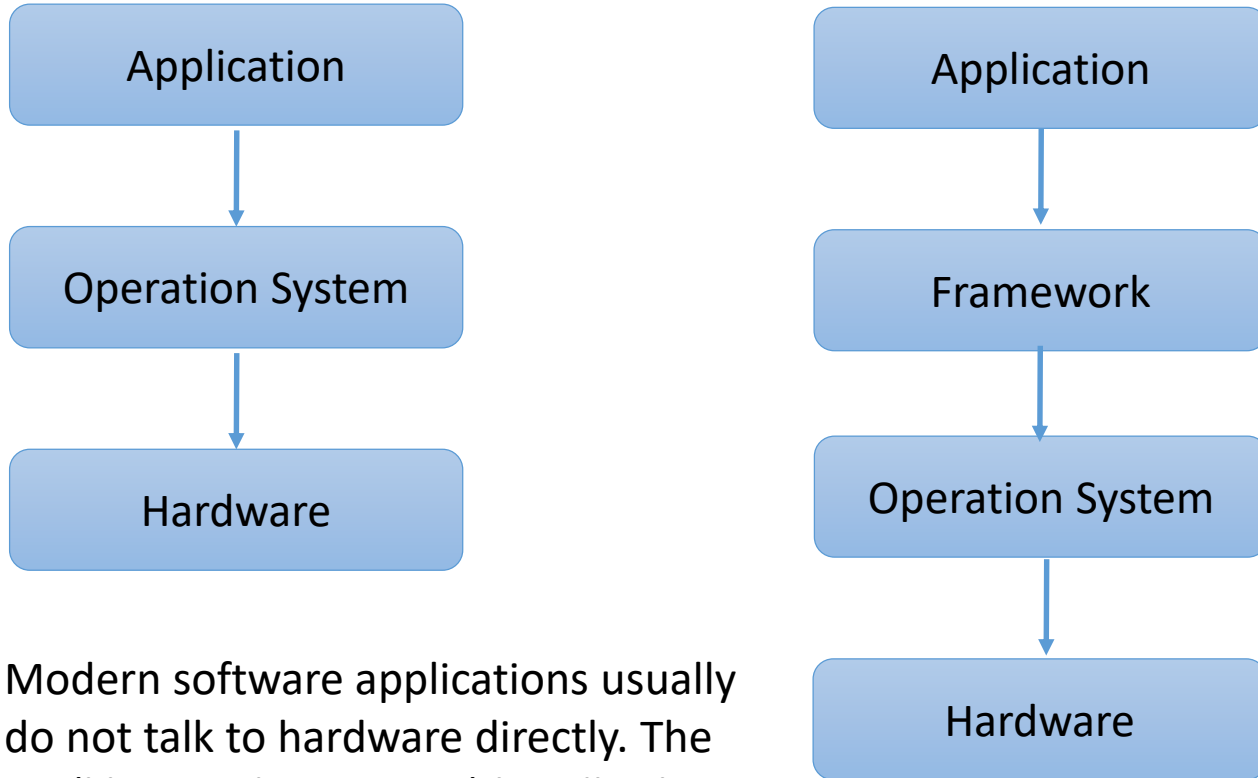
Week 02 Hardware C Programming

Outline

- Conceptual Software Layers
- Data types in C language
- Essential elements of C language
- Bit-wise operations
- Pointers and arrays
- Function pointers

Software Development

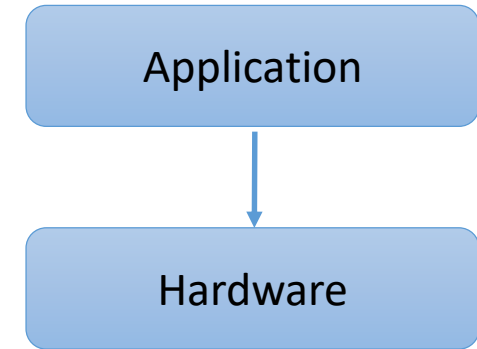
Desktop/Mobile Software



Modern software applications usually do not talk to hardware directly. The OS (like Windows, Linux) handles low-level work. Higher level languages like Java, C#, Matlab needs an additional framework on top of OS.

Applications written in Lower level languages (C, C++, Assembly) can talk to hardware directly.

Microcontroller Software



- Software for micro-controller talks to the hardware directly.
- It is sometimes referred to as bare-metal programming

Programming Languages

- Prolog
- Matlab
- C#, Java, Python, etc
- C, C++, Pascal, etc
- Assembly
- Machine code

High Level



Level of abstraction

Low Level

- The lower the abstraction level, the easier it is to talk directly to the metal. The better the performance. (i.e. the program runs faster)
- The higher the abstraction level, the easier and faster to get the job done. (i.e. the developer process is faster)

C programming language

- C is a general-purpose imperative computer programming language.
- First appeared in 1972
- C language is low level enough that you can directly talk to hardware (access any register) using it.
- C language is high level enough that you do not need to worry about processor-specific instruction sets.



Structure of Basic C program

```
#include <stdio.h>

void main (void)
{
    printf ("Hello, World!\n");
}
```

- The main function is the entry-point of the program
- The main function can also return *int*
- It can also accept arguments/parameters

C Data Types

Type	Size
bool	8 bits
char	8 bits
unsigned char	8 bits
signed char	8 bits
int	16 or 32 bits*
unsigned int	16 or 32 bits*
short	16 or 32 bits*
unsigned short	16 or 32 bits*
long	32 bits or 64 bits*
unsigned long	32 bits or 64 bits*
float	32 bit
double	64 bit

char is just a data type to store an 8-bit number

*depends on hardware
On ATMEGA328p, int is 16 bit and long is 32-bit

Print function

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

int main()
{
    float a = 3.2;
    int b = 5;

    printf("%f \n", a);
    printf("%d \n", b);
    return 0;
}
```

Output:

```
3.200000
5
```

Char	Type	Interpretation
d,i	int	signed decimal notation
o	int	unsigned octal notation (without leading 0)
x	int	unsigned hexadecimal notation (without leading 0x, uses abcdef)
X	int	unsigned hexadecimal notation (without leading 0x, uses ABCDEF)
u	int	unsigned decimal notation
c	int	single character, after conversion to unsigned char)
s	char *	characters from the string until a \0 or precision is reached)
f	double	decimal notation of the form $[-]mmm.ddd$; precision controls ds
e,E	double	decimal notation of the form $[-]m.ddde\pm xx$; precision controls ds
g,G	double	selects the best choice between %e and %f
p	void *	print as a pointer (implementation-dependent representation)
n	int *	the number of characters output so far via this printf() is copied into the argument; no argument is converted
%		print a %; no argument is converted

Special characters

Control Char	Output	Description
<code>\n</code>	NL (LF)	newline
<code>\t</code>	HT	horizontal tab
<code>\v</code>	VT	vertical tab
<code>\b</code>	BS	backspace
<code>\r</code>	CR	carriage return
<code>\f</code>	FF	formfeed
<code>\a</code>	BEL	audible alert
<code>\\</code>	<code>\</code>	backslash
<code>\?</code>	<code>?</code>	question mark
<code>\'</code>	<code>'</code>	single quote
<code>\"</code>	<code>"</code>	double quote
<code>\ooo</code>	<i>ooo</i>	octal number
<code>\xhh</code>	<i>hh</i>	hexadecimal number

Address operator (&)

```
char a = 'Z';
```

```
&a //denotes the memory address of the variable named "a"
```

The address could be 64-bit, 32-bit or 16-bit depending on the architecture of the machine.

Example program

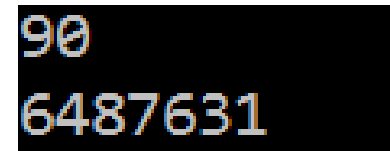
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
int main()
{
    char a = 90;

    printf("%d \n", a);
    printf("%d \n", &a);

    return 0;
}
```

Output:



```
90
6487631
```

Note: the second value is the address of the variable a. The value will differ from machine to machine and session to session.

Scanf function

Scanf function reads user input from console

You have to pass the address of a variable (not the actual variable) to scanf function.

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

int main()
{
    float input;

    printf("Please enter a number: ");
    scanf("%f", &input);

    printf("The square root is : %f \n", sqrt(input));
    return 0;
}
```

Output:

```
Please enter a number: 6
The square root is : 2.449490
```

Example program

What does the following program do?

```
void main (void)
{
    int fahrenheit;
    int celsius;
    int lower;
    int upper;
    int step;

    lower = 0;    /* lower limit */
    upper = 300;
    step = 20;
    fahrenheit = lower;
    while (fahrenheit <= upper)
    {
        celsius = 5 * (fahrenheit - 32) / 9;
        fahrenheit = fahrenheit + step;
        printf ("%d F = %d C\n", fahrenheit, celsius);
    }
}
```

Output:

```
20 F = -17 C
40 F = -6 C
60 F = 4 C
80 F = 15 C
100 F = 26 C
120 F = 37 C
140 F = 48 C
160 F = 60 C
180 F = 71 C
200 F = 82 C
220 F = 93 C
240 F = 104 C
260 F = 115 C
280 F = 126 C
300 F = 137 C
320 F = 148 C
```

Assignment operator (=)

All the following lines of code are identical

```
char a = 'Z';           //Assign an ASCII character
char a = 90;            //Assign an decimal value
char a = 0x5A;          //Assign a hexadecimal value
char a = 0b1011010;     //Assign a binary value
```

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

```
int main()
{
```

```
    char a = 'Z';
    printf("%c \n", a);
```

```
    a--;
    printf("%c \n", a);
```

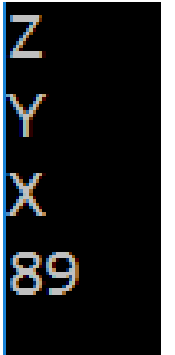
```
    int b = 88;
    printf("%c \n", b);
```

```
    printf("%d \n", a);
```

```
    return 0;
```

```
}
```

Output



```
Z
Y
X
89
```

Type conversion

```
char c;  
short s;  
  
/* suppose s is assigned some value prior to this assignment */  
c = (char) s;  
  
/* suppose c is assigned some value prior to this assignment */  
s = (short) c;
```

- One data type can be converted to others.
- This process is called *type casting* (or *casting* for short).

Enum

```
typedef enum
{
    FALSE,
    TRUE
} Boolean;

/* skipping many lines */

void SomeFunction (void)
{
    int i;
    Boolean answer;
    char j;

    /* skipping many lines */

    answer = FALSE;
    while (answer != TRUE)
    {
        /* skipping many lines */
    }
}
```

New data
types can be
defined using
enum
keyword

Mathematical operators

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division (for non-floats, quotient is returned)
%	modulo (for non-floats, remainder is returned)

Comparison operators

Operator	Operation
>	greater-than
>=	greater-than or equal-to
<	less-than
<=	less-than or equal-to
==	equal-to
!=	not equal-to
&&	and
	or
!	unary negation (non-zero \rightarrow 0, 0 \rightarrow 1)

Increment operators

Operator	Operation
++	increment value by 1; either before or after the variable is used
--	decrement value by 1; either before or after the variable is used

Suppose x = 10 initially

Statement	n After	x After
n = x++;	10	11
n = ++x;	11	11
n = x--;	10	9
n = --x;	9	9

Conditional expression

```
/* This conditional expression... */  
z = (a > b) ? c : d;  
  
/* ...is the same as the following code. */  
if (a > b)  
{  
    z = c;  
}  
else  
{  
    z = d;  
}
```

Note: it is a good practice to always put { } after if and else statements.

Avoid this kind of practice! 

```
void main (void)  
{  
    if (a < 2)  
        doSomething();  
    else  
        doSomethingelse();  
        doMore();  
}
```

Switch statement

```
switch (expression)
{
    case constant-expression1:
        /* Performed when expression == constant-expression1 */
        statements
        break;

    case constant-expression2:
        /* Performed when expression == constant-expression2 */
        statements
        break;

    /* skipping other cases */

    default :
        /* Performed when expression != any constant expression */
        statements
        break;
}
```

Loops

```
for (expression1; expression2; expression3)  
    statement
```

```
do  
    statement  
while (expression);
```

```
expression1;  
while (expression2)  
{  
    statement  
    expression3;  
}
```

Infinite loop

```
for (;;)
    statement
```

```
while (1)
    statement
```

```
do
    statement
while (1);
```

Structs

```
struct TagPoint  
{  
    int x;  
    int y;  
};
```

```
struct TagPoint maxPoint;
```

```
maxPoint.x == 320;  
maxPoint.y == 200;
```

Good programming practice

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

Good code 

```
void main (void )
```

```
{
```

```
    int a,b,c;
```

```
    for (int a=0; a<5; a++)
```

```
    {
```

```
        for (int b=0; b<5; b++)
```

```
        {
```

```
            for (int c=0; c<5; c++)
```

```
            {
```

```
                while(1)
```

```
                {
```

```
                }
```

```
                if (a < 2)
```

```
                {
```

```
                    dosomething;
```

```
                }
```

```
                else
```

```
                {
```

```
                    do
```

```
                    {
```

```
                }while(1);
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

- Always put {} where they belong.
- After { character, insert a new line and a tab character

Bad code 

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

```
void main (void){
```

```
    int a,b,c;
```

```
        for (int a=0; a<5; a++) {
```

```
    for (int b=0; b<5; b++) {
```

```
        for (int c=0; c<5; c++)
```

```
        {
```

```
            while(1){
```

```
            }
```

```
                if (a < 2) dosomething;
```

```
                else do
```

```
                    while(1);
```

```
        }
```

```
    }
```

```
}
```

```
}
```

```
}
```


Exercise 1

Write a program that checks whether a given number is a prime number.

Ask for user for a number

Respond back whether the number is a prime number

Repeat until the user enters a negative number

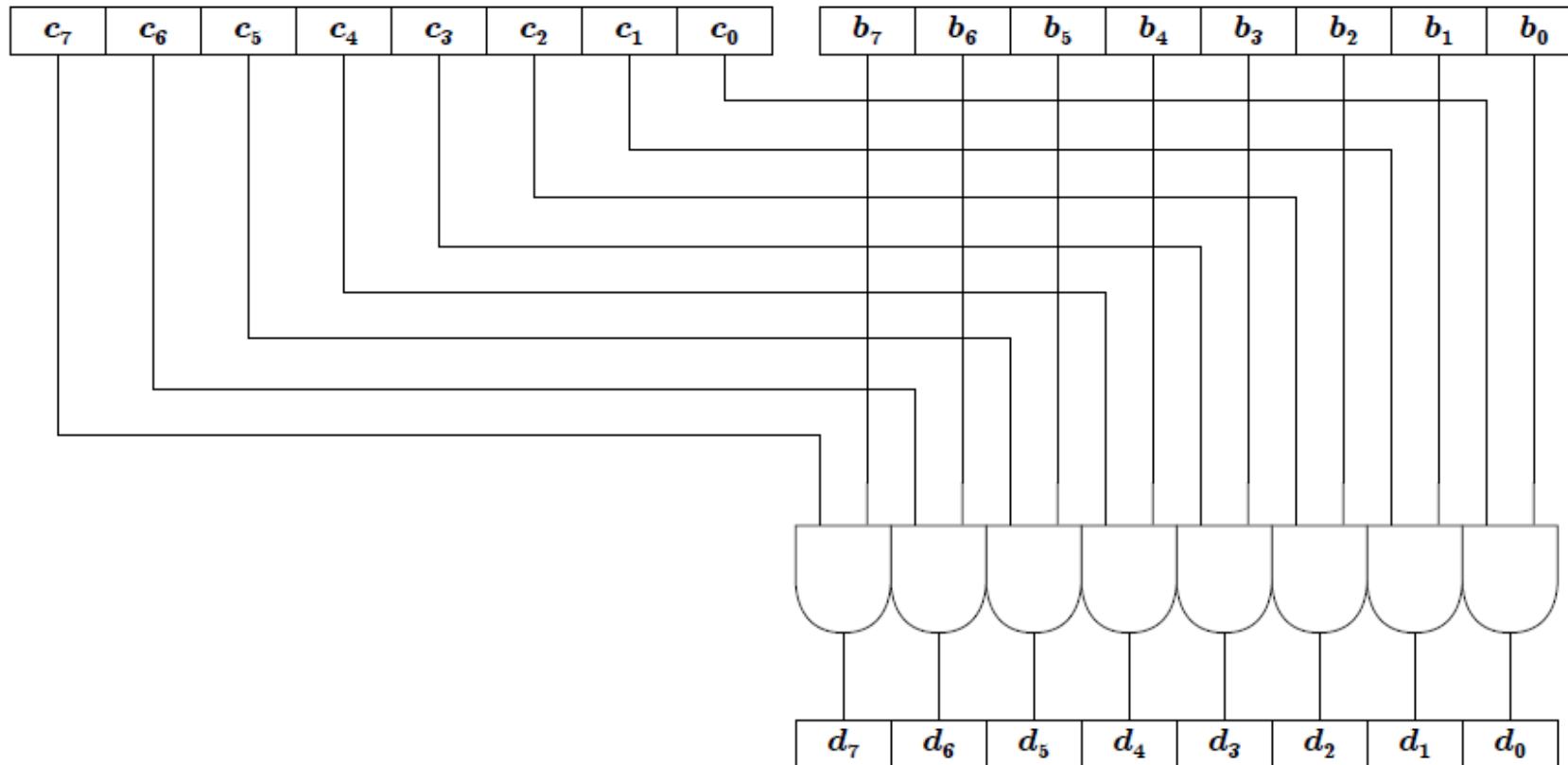
End the program

Bitwise operators

Bitwise operators manipulate individual bits of variables

Operator	Operation
&	AND (boolean intersection)
	OR (boolean union)
^	XOR (boolean exclusive-or)
<<	left shift
>>	right shift
~	NOT (boolean negation, i.e., ones' complement)

Bitwise AND operation of two 8-bit numbers



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

int main()
{
    unsigned char a = 90;    //0101 1010
    unsigned char b = 23;    //0001 0111
    unsigned char c;

    c = ~a;
    printf("%d \n", c);

    c = a&b;
    printf("%d \n", c);

    c = a|b;
    printf("%d \n", c);

    c = a^b;
    printf("%d \n", c);

    c = b<<2;
    printf("%d \n", c);

    c = a>>3;
    printf("%d \n", c);

    return 0;
}
```

Output



165
18
95
77
92
11

Bit-masking

Suppose we have an 8-bit unknown number named “a”. We want to set the third LSB (make it HIGH) and keep the remaining bits same as before. How can we accomplish this?

We have this → a = XXXX XXXX

We want this → a = XXXX X1XX

Bit-masking

Write a program that sets the third LSB of an 8-bit unsigned number entered by the user. What is the output if the user enters 98.

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

```
int main()
{
```

```
    unsigned char a;
    printf("Please enter a number: ");
    scanf("%d", &a);
```

This is called a mask

```
    a = a | 0b00000100;    // a = a | (1<<2) is also the same
    printf("%d \n", a);
```

```
    return 0;
```

```
}
```

Output:

```
Please enter a number: 98
102
```

Bit-masking

Suppose we have an 8-bit unknown unsigned number named “a”. We want to clear the third LSB and keep the remaining bits same as before. How can we accomplish this?

We have this → a = XXXX XXXX

We want this → a = XXXX X0XX

Bit-masking

Write a program that clears the third LSB of an 8-bit unsigned number entered by the user.

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

```
int main()
```

```
{
```

```
    unsigned char a;
```

```
    printf("Please enter a number: ");
```

```
    scanf("%d", &a);
```

```
    a = a & 0b11111011;
```

```
    printf("%d \n", a);
```

```
    return 0;
```

```
}
```

This is called a mask

// a = a & ~(1<<2) is also the same

Bit-masking

Write a program that **checks** the third LSB bit of an signed number entered by the user. Output “HIGH” if the bit is 1 and “LOW” if the bit is 0.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main()
6  {
7      signed char a;
8      printf("Please enter a number: ");
9      scanf("%d", &a);
10
11     if (a & 0b00000100 == 0)
12     {
13         printf("LOW");
14     }
15     else
16     {
17         printf("HIGH");
18     }
19
20     return 0;
21 }
```

Output:

```
Please enter a number: -9
HIGH
```

Note: Line 11 to 18 can be simplified using ternary operator

```
char *text = (a & 0b00000100 == 0) ? "LOW" : "HIGH";
printf(text);
```

The above code can be simplified further

```
printf((a & 0b00000100 == 0) ? "LOW" : "HIGH");
```

Bit-masking

Summary of bit-masking operations

Statement	c	mask	d	Embedded usefulness
<code>d = (c & mask);</code>	0x55	0x0F	0x05	Clear bits that are 0 in the mask
<code>d = (c mask);</code>	0x55	0x0F	0x5F	Set bits that are 1 in the mask
<code>d = (c ^ mask);</code>	0x55	0x0F	0x5A	Invert bits that are 1 in the mask
<code>d = (c << 3);</code>	0x55		0xA8	Multiply by a power of 2
<code>d = (c >> 2);</code>	0x55		0x15	Divide by a power of 2
<code>d = ~c;</code>	0x55		0xAA	Invert all bits

Comparison between bitwise and logical operators

Statement	x	y	z After	Operation
<code>z = (x & y);</code>	1	2	0	Bitwise AND
<code>z = (x && y);</code>	1	2	1	Logical AND
<code>z = (x y);</code>	1	2	3	Bitwise OR
<code>z = (x y);</code>	1	2	1	Logical OR

Assignment operators

Operator	Syntax	Equivalent Operation
<code>+=</code>	<code>i += j;</code>	<code>i = (i + j);</code>
<code>-=</code>	<code>i -= j;</code>	<code>i = (i - j);</code>
<code>*=</code>	<code>i *= j;</code>	<code>i = (i * j);</code>
<code>/=</code>	<code>i /= j;</code>	<code>i = (i / j);</code>
<code>%=</code>	<code>i %= j;</code>	<code>i = (i % j);</code>
<code>&=</code>	<code>i &= j;</code>	<code>i = (i & j);</code>
<code> =</code>	<code>i = j;</code>	<code>i = (i j);</code>
<code>^=</code>	<code>i ^= j;</code>	<code>i = (i ^ j);</code>
<code><<=</code>	<code>i <<= j;</code>	<code>i = (i << j);</code>
<code>>>=</code>	<code>i >>= j;</code>	<code>i = (i >> j);</code>

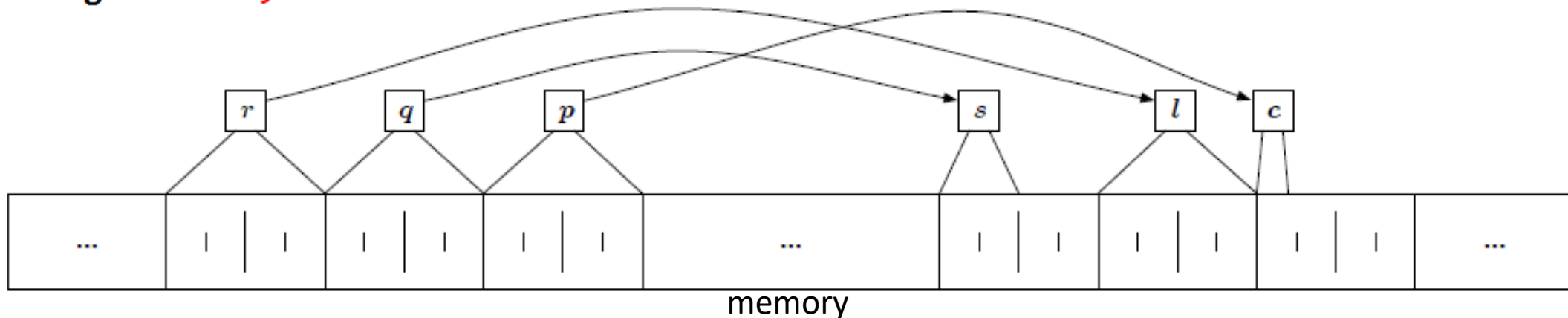
Pointers

- A pointer is a variable that contains the address of another variable.
- It “points” to another variable.

```
char c;  
short s;  
long l;
```

All pointers have the same size (64-bit, 32-bit, or 16-bit depending on the architecture of the machine).

```
char *p = &c;  
short *q = &s;  
long *r = &l;
```



Pointers

* is called the dereferencing operator

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

int main()
{
    char c;
    char *p = &c; //pointer p points to c

    c = 32;

    printf("%d \n", p); // The first line prints the value of p (which is the address of c)
    printf("%d \n", *p); // The second line prints the value of the variable that p points to (value of c)
    printf("%d \n", &p); // The third line prints the address of p
}
```

Output

```
6487631
32
6487616
```

Pointers

- You can declare a pointer that points to anything. For example:
 - Pointers to arrays
 - Pointers that point to other pointers
 - Pointers to structure types
 - Pointers to functions

Exercise: What is the output?

```
char c = 5;  
char *p = &c;
```

```
c = *p + 1;    // Increments c  
*p += 1;      // Increments c  
++(*p);       // Increments c  
(*p)++;      // Increments c
```

```
printf("%d", c);
```

Output: 9

Array

Arrays are blocks of consecutive types.

Style 1

```
int array[10];
```

```
array[0] = 3;  
array[1] = 4;  
array[2] = 5;  
array[3] = 1;  
array[4] = 9;  
array[5] = 2;  
array[6] = 6;  
array[7] = 7;  
array[8] = 10;  
array[9] = 2;
```

Style 2

```
int array[] = {3,4,5,1,9,2,6,7,10,2};
```

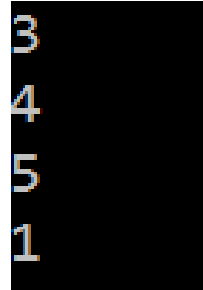
An array is just like a pointer.

```
int array[] = {3,4,5,1,9,2,6,7,10,2};
```

```
int *p = array; This statement is same as int *p = &array[0];
```

```
printf("%d \n", *(p));  
printf("%d \n", *(p+1));  
printf("%d \n", *(p+2));  
printf("%d \n", *(p+3));
```

output



```
3  
4  
5  
1
```

*(p) is same as array[0]
*(p+1) is same as array[1]
*(p+2) is same as array[2]
*(p+3) is same as array[3]

Exercise: what is the output of the following program?

```
int array[] = {3,4,5,1,9,2,6,7,10,2};
```

```
int *p = &array[3];
```

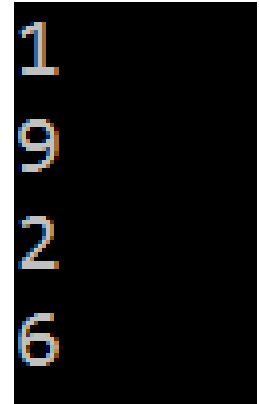
```
printf("%d \n", *(p));
```

```
printf("%d \n", *(p+1));
```

```
printf("%d \n", *(p+2));
```

```
printf("%d \n", *(p+3));
```

output



```
1
9
2
6
```

Pointers & Bitwise Operators

The concept of pointers combined with bitwise operators allows a program to set, clear or read any bit of any address of the main memory.

Example:

Write a program that sets the MSB bit of register at address #7.

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

int main()
{
    unsigned char *a = 7;

    *a |= 0x80; //1000 0000

    printf("%d", *a);

    return 0;
}
```

Note: if you try to run this program on a PC, the program will crash

Address (hex)	MSB						LSB
0	1	0	1	0	1	0	0
1	1	0	0	1	0	1	0
2	0	1	0	0	1	0	1
3	1	0	1	0	0	1	0
4	0	1	0	1	0	1	0
5	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0
7	0	0	0	0	0	0	1
8	0	0	1	0	0	0	0
9	0	0	0	0	1	0	0
10	0	0	1	1	0	0	0
11	1	0	1	0	0	1	0
12	0	0	0	1	0	0	0
13	0	0	1	1	0	0	1
14	1	0	0	0	1	0	0
15	1	0	0	1	0	0	0
A	0	0	0	1	0	0	0
B	0	1	0	0	0	0	0
C	0	0	0	0	0	0	0
D	0	0	0	1	0	0	1
E	1	0	0	0	0	0	0

Two-dimensional arrays

```
#define MAX_ROWS 2
#define MAX_COLS 5

const short m_table[MAX_ROWS][MAX_COLS] =
{
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10}
};
```

The following statements are true

```
m_table[0][1] == 2;
m_table[1][4] == 10;
```

Three-dimensional array

```
#define MAX_DIM0 3
#define MAX_DIM1 2
#define MAX_DIM2 5

const short m_table[MAX_DIM0][MAX_DIM1][MAX_DIM2] =
{
    {
        {1, 2, 3, 4, 5},
        {6, 7, 8, 9, 10}
    },
    {
        {11, 12, 13, 14, 15},
        {16, 17, 18, 19, 20}
    },
    {
        {21, 22, 23, 24, 25},
        {26, 27, 28, 29, 30}
    },
};
```

The following statements are true

```
m_table[0][1][2] == 8;
m_table[2][1][3] == 29;
```

Passing by value

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

int main()
{
    short a = 10;
    short b = 13;
    swap (a,b);

    printf("a = %d, b= %d", a, b);
    return 0;
}

void swap (short x, short y)
{
    short temp ;
    temp = x;
    x = y;
    y = temp;
}
```

- The swap function swaps the values of the inputs.
- What is the output of the program?

a = 10, b= 13

- The swap function does not affect a and b because it only swaps the local copies (x and y)

Passing by reference

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

int main()
{
    short a = 10;
    short b = 13;
    swap (&a, &b);

    printf("a = %d, b= %d", a, b);
    return 0;
}
```

```
void swap (short *x, short *y)
{
    short temp ;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

- What is the output of the program?

a = 13, b= 10

- The swap function affects the values of a and b

Function pointers

Function pointers store memory locations that point to functions.

Function pointers are very tricky, but often used in embedded programming for various reasons including:

- callbacks into RTOSes;
- ISR handling;
- I/O port interfacing to higher level;

Format:

```
return_type (* variableName)(argument_list);
```

function

```
int newfunction(char x)
{
    return (x < 512);
}
```

function pointer m

```
int (*m) (char);
```

function

```
void swap (short x, short y)
{
    short temp ;
    temp = x;
    x = y;
    y = temp;
}
```

function pointer p

```
void (*p) (short, short);
```

Function pointer example.

What is the output?

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

```
int newfunction (char x)
{
    return (x+1);
}
```

```
int main()
{
    int (*m) (char) = newfunction;

    int value = (*m)(32);

    printf("%d", value);
    return 0;
}
```

Output:

33

Note: newfunction can be invoked through m (which is a pointer to newfunction)

Object Oriented Programming (OOP)

- C++ language is an extension of C language with OOP support and other new features. Initially called “*C with classes*”.
- C++ language first appeared in 1983.
- The Arduino library actually is written in C++.
- OOP is a programming paradigm based on the concept of “objects”.
Examples of objects in Arduino are Serial, Servo, SPI, I2C.

- A class contains private and public members.
- Private members can only be accessed by members of the class.
- Members can be variables or functions.
- In this example, the class called “vector” has 3 variable members (x, y, z), which are also called properties.
- It has one method called getMagnitude.

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

class vector
{
    private:
        //No private members

    public:
        double x;
        double y;
        double z;

        double getMagnitude()
        {
            return sqrt(x*x+y*y+z*z);
        }
};
```

Name of class

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

```
class vector
```

```
{
```

```
private:
```

```
    //No private members
```

```
public:
```

```
    double x;
```

```
    double y;
```

```
    double z;
```

```
    double getMagnitude()
```

```
    {
```

```
        return sqrt(x*x+y*y+z*z);
```

```
    }
```

```
};
```

output

```
Magnitude of v1 = 5.678908
Magnitude of v2 = 10.212737
Magnitude of v2 = 9.205976
```

continuation from left-hand side

```
int main ()
```

```
{
```

```
    vector v1;
```

```
    vector v2;
```

```
    vector v3;
```

```
    v1.x = 3.5;
```

```
    v1.y = 4;
```

```
    v1.z = 2;
```

```
    v2.x = 2.5;
```

```
    v2.y = 9.3;
```

```
    v2.z = 3.4;
```

```
    v3.x = 7.1;
```

```
    v3.y = 5.3;
```

```
    v3.z = 2.5;
```

```
    printf("Magnitude of v1 = %f \n", v1.getMagnitude());
```

```
    printf("Magnitude of v2 = %f \n", v2.getMagnitude());
```

```
    printf("Magnitude of v2 = %f \n", v3.getMagnitude());
```

```
    return 0;
```

```
}
```

```

class vector
{
    private:
        double x;
        double y;
        double z;

    public:
        void setX(double value)
        {
            x = value;
        }
        void setY(double value)
        {
            y = value;
        }
        void setZ(double value)
        {
            z = value;
        }
        double getMagnitude()
        {
            return sqrt(x*x+y*y+z*z);
        }
};

```

```

int main ()
{
    vector v1;
    vector v2;
    vector v3;

    v1.setX(3.5);
    v1.setY(4);
    v1.setZ(2);

    v2.setX(2.5);
    v2.setY(9.3);
    v2.setZ(3.4);

    v3.setX(7.1);
    v3.setY(5.3);
    v3.setZ(2.5);

    printf("Magnitude of v1 = %f \n", v1.getMagnitude());
    printf("Magnitude of v2 = %f \n", v2.getMagnitude());
    printf("Magnitude of v2 = %f \n", v3.getMagnitude());
    return 0;
}

```

In this example, x, y, and z are private members. But their values can be set through public functions. The output is the same as in the previous example.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

class vector
{
private:
    double x;
    double y;
    double z;

public:
    vector(double c1, double c2, double c3)
    {
        x = c1;
        y = c2;
        z = c3;
    }
    double getMagnitude()
    {
        return sqrt(x*x+y*y+z*z);
    }
};

int main ()
{
    vector v1(3.5, 4, 2);
    vector v2(2.5, 9.3, 3.4);
    vector v3(7.1, 5.3, 2.5);

    printf("Magnitude of v1 = %f \n", v1.getMagnitude());
    printf("Magnitude of v2 = %f \n", v2.getMagnitude());
    printf("Magnitude of v2 = %f \n", v3.getMagnitude());
    return 0;
}

```

Constructor

Invokes

- A function of the class with the same name of the class is called a **constructor**.
- The constructor gets called whenever an object of that class gets created.
- A class can have multiple constructors.
- The output of this example is same as the previous example.


```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

class vector
{
private:
    double x;
    double y;
    double z;

public:
    vector(double c1, double c2, double c3)
    {
        x = c1;
        y = c2;
        z = c3;
    }
    vector ()
    {
        x = 1;
        y = 1;
        z = 1;
    }
    double getMagnititude()
    {
        return sqrt(x*x+y*y+z*z);
    }
};

int main ()
{
    vector v1(3.5, 4, 2);
    vector v2(2.5, 9.3, 3.4);
    vector v3(7.1, 5.3, 2.5);
    vector v4;

    printf("Magnitude of v1 = %f \n", v1.getMagnititude());
    printf("Magnitude of v2 = %f \n", v2.getMagnititude());
    printf("Magnitude of v2 = %f \n", v3.getMagnititude());
    return 0;
}

```

Second constructor



- In this example, the first constructor gets called when v1,v2, and v3 get declared.
- The second constructor gets called when v4 is declared.
- v4 will have x=1, y=1 and z=1.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

class vector
{
private:
    double x;
    double y;
    double z;

public:
    vector(double c1, double c2, double c3)
    {
        x = c1;
        y = c2;
        z = c3;
    }
    void Add(vector another)
    {
        x += another.x;
        y += another.y;
        z += another.z;
    }
    double getMagnitude()
    {
        return sqrt(x*x+y*y+z*z);
    }
    void Print()
    {
        printf("x = %f  y = %f  z = %f \n", x, y, z);
    }
};

```

```

int main ()
{
    vector v1(3.5, 4, 2);
    vector v2(2.5, 9.3, 3.4);
    v1.Add(v2);
    v1.Print();
    return 0;
}

```

Output

```
x = 6.000000  y = 13.300000  z = 5.400000
```

The Add function add a vector to the current vector.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

class vector
{
private:
    double x;
    double y;
    double z;

public:
    vector(double c1, double c2, double c3)
    {
        x = c1;
        y = c2;
        z = c3;
    }
    void Add(vector another)
    {
        x += another.x;
        y += another.y;
        z += another.z;
    }
    static vector Add(vector a, vector b)
    {
        double x_new = a.x + b.x;
        double y_new = a.y + b.y;
        double z_new = a.z + b.z;
        vector newvector(x_new, y_new, z_new);
        return newvector;
    }
    void Print()
    {
        printf("x = %f  y = %f  z = %f \n", x, y, z);
    }
};


```

Static member


```

int main ()
{
    vector v1(3.5, 4, 2);
    vector v2(2.5, 9.3, 3.4);
    vector v3 = vector::Add(v1, v2);
    v3.Print();
    return 0;
}

```



- Static members can be directly accessed from the class without creating objects.
- Static members are accessed using `::` keyword.
- The output is the same as in the previous example.