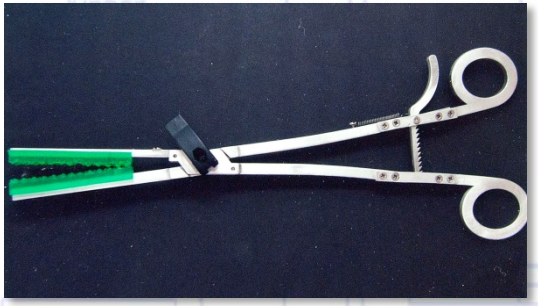
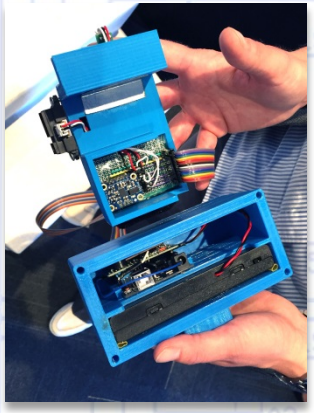
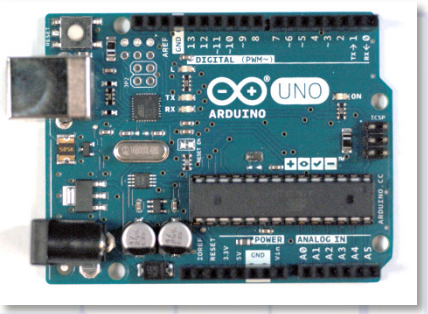
Introductory Medical Device Prototyping

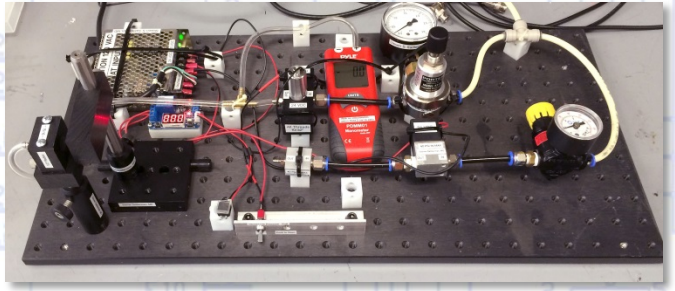
Programming in C

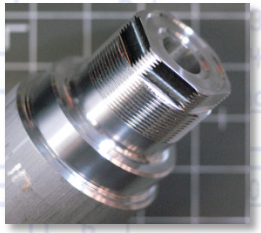
Prof. Steven S. Saliterman, http://saliterman.umn.edu/
Department of Biomedical Engineering, University of Minnesota











Programming

1. Software is the *smart* in your “smart device.”
2. An *algorithm* displayed as a *flowchart,* transforms your problem into various input, processing, decision and output steps
3. Lines of *code* are written to implement your algorithm.
4. Code may be written in machine language and/or higher level languages such as C, C++, and C#.
5. A *compiler* converts your program into *machine language* that the *microcontroller* understands.
6. The *compiled code* is then *uploaded* into a board containing the microcontroller, memory and various interface circuits.
7. Errors are then fixed by testing and *debugging*.
8. Rather than a microcontroller board, you might consider a *single-board computer*, such as *Raspberry Pi*, giving you a richer programming, processing and interface environment.

Programming in C

- The most widely used programming language.
- C was originally developed by *Dennis Ritchie* between 1969 and 1973 at Bell Labs.
- A *structured programming* computer language.
- Maps efficiently to machine instructions, largely replacing previous *assembly language* programing.
- Uses range from embedded systems to supercomputers.
- *Standardized* by the American National Standards Institute (ANSI) since 1989.
- Low-level access to computer memory is possible by converting machine addresses to *typed pointers*.
- Many later languages have borrowed directly or indirectly from C, including C++, Java, JavaScript, C#, Objective-C, Verilog (hardware description language), and others.

Author: Denise Panyik-Dale. Dennis Ritchie at Japan Prize Foundation ceremony in May 2011.

Integrated Development Environment (IDE)…

1. Examples:
   - Arduino
   - Microchip MPLAB X for PIC
   - Microsoft Visual Studio for Windows
2. Editing – Entering the *Program* Code
3. Compiling – C, C++, C# & or other *Languages*
4. Running – *Executing* the Program
5. Debugging – Finding & *Correcting Errors*

C Variables and Modifiers…

1. *Basic Data Types* (Compiler Dependent)
    1. Char - typically one byte (8 bits or “1 byte”)
    2. Int – integer (16 bits)
    3. Float – a single precision floating point value (32 bits)
    4. Double – a double precision floating value (64 bits)
2. *Modifiers*
    1. Unsigned
    2. Short
    3. Long
3. *Boolean Type* – variable is either True or False

Data Types for Arduino (for example)…

1. boolean (8 bit) - simple logical true/false (1 byte = 8 bits)
2. byte (8 bit) - unsigned number from 0-255
3. char (8 bit) - signed number from -128 to 127. The compiler will attempt to interpret this data type as a character in some circumstances, which may yield unexpected results.
4. unsigned char (8 bit) - same as 'byte'; if this is what you're after, you should use 'byte' instead, for reasons of clarity.
5. word (16 bit) - unsigned number from 0-65535 (1 word = 2 bytes)
6. unsigned int (16 bit)- the same as 'word'. Use 'word' instead for clarity and brevity

https://learn.sparkfun.com/tutorials/data-types-in-arduino

7. int (16 bit) - signed number from -32768 to 32767. This is most commonly what you see used for general purpose variables in Arduino example code provided with the IDE.
8. unsigned long (32 bit) - unsigned number from 0-4,294,967,295. The most common usage of this is to store the result of the millis() function, which returns the number of milliseconds the current code has been running.
9. long (32 bit) - signed number from -2,147,483,648 to 2,147,483,647
10. float (32 bit) or double- signed number from -3.4028235E38 to 3.4028235E38. Floating point on the Arduino is not native; the compiler has to jump through hoops to make it work. If you can avoid it, you should.

https://learn.sparkfun.com/tutorials/data-types-in-arduino

Program Structure

```
//Typical Program Structure                          // Comment (begins with //)
#include <stdio.h>                                   // Specify standard libraries (one or more)
#include <stdbool.h>
int main (void)                                      // Beginning of main program
{                                                    // Signifies a group of declarations & statements
   int a = 5;                          //e.g.      // Declare global variable and assign a value
   float b;                            //e.g.      // Declaring a floating point value
   int val[4];                                      // Declaring an integer array: val[0], val[1], val[2], val[3]
   _Bool finished=false;                            // Declaring a boolean (true/false or 1/0)
  for (n=1; n<=200; n=n+1) ;           //e.g.      // Loop (also – while, do)
     { more statements }
  if (c=4)                             //e.g.      // Conditional (also –if-else, switch, condition)
    { more statements }
   printf;                             //e.g.      // Input and output command
}                                                    // End of main program
```


Kochan, S.G. *Programming in C*, 3rd ed., Developer's Library, Indianapolis, Indiana (2005).

Example Program…

```
//Add all even numbers from 0 to 100
#include <stio.h>
int main (void)
{
   int sum = 0,  n;
   for (n = 0; n <= 100; n = n+2) ;
     {
      sum = sum + n;
     }
  printf (“The sum is: ”, sum);
}
```

```
// Program Title
// Include standard input and output libraries
// Beginning of main program

// Declare “sum” and “n” as integers
// Loop, incrementing n from 0 to 100, by 2 each time

// Add previous sum to present n

//Loop is done – print the final sum
```

Relational Operators…

Operator	Meaning	Example
==	Equal to	count == 10
!=	Not equal to	person != cat
<	Less than	a < b
<=	Less than or equal	c <= e
>	Greater than	d > 7
>=	Greater than or equal	j >= k

Arithmetic Expressions…

Operation	Operator Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Power	e
Precedence	* or /, then + or – // parenthesis promotes ()
Modulus	% gives the remainder
Combining Operator with Assignment	e.g. count += 10; //same as: count = count +10;
	++count; //increments count by 1

Implicit Conversions…

1. Whenever a floating-point value is assigned to an integer, the decimal portion is truncated.
2. Assigning an integer value to floating point variable does not change the value.
3. Whenever two operands in an expression are integers, the operation is carried out under the rules of integer arithmetic. Decimal portions are lost even if assigned to a floating point variable.
4. Any operation between two values is performed as a floating point operation if either value is a floating point constant or variable.
5. *Type Cast Operator* e.g. (int) or (float) preceding the value converts the value for *the purpose of the calculation only*.

Program Statements

- Loop Statements
  - For
  - While
  - Do-While
- Decision Statements
  - Break and Continue
  - If
  - If-Else
  - Switch-Case

"For" Statement (a Loop)…

Statement Format

```
for (initialization; condition; increment)
   {program statement(s);}
```

Example – What is the value of the a[49] element?

Example Code

```
…
int a[100];
for (int n = 0; n < 100; n = n + 1)
 {
   a[n] = n * 2;
  }
…
```

"While" Statement (a Loop)…

while (*expression – a boolean that is true or false)*
{program statement(s);}

Example – What is the value of a[30] element?

```
…
int a[100];
int n = 0;
while (n < 100) {
    a[n] = n * 3;
    n = n + 1;  // Could also use
                "++n"
   }
…
```

“Do-while” Statement (a Loop)…

do
  {program *statement(s)}*
while *(test condition);*

Example – What is the value of a[75] element?

```
…
int a[100];
int n = 0;
do {
      a[n] = n * 4;
      n = n + 1;
 }
while n < 100;
…
```

“Break” and “Continue”…

break;

Based on a conditional statement, the action will be to leave the loop (or the present loop if nested).

continue;

Causes the loop in which it is executed to skip ahead to the next cycle of the loop. Any statements after the “continue” within the loop are ignored.

"If" Statement (a Decision)…

if (*expression)*

{program statements;}

Example – What is the value of n?

```
…
int a = 4, n = 0;
if a <= 5 {
   n = n + 50;
}
…
```

"If-Else" Statement (a Decision)…

```
if (expression)
      {program statements;}
else
     {program statements};
```

Example – What is the value of n?

```
…
int a = 10, n = 0;
if a <= 5 {
   n = n + 50;
}
else {
   n = n  + 25;
}
…
```

"Switch – Case" Statement…

```
switch (expression)
{
   case label1:
          program statements;
          break;
   case label2:
          program statements;
          break;
  default:
          program statements;
           break;
}
```

For example:

```
int a;
_Bool buy;
…
a = 2;
switch (a)
{
   case 1:                    // if a =1
            buy = true;
             break;
   case 2:                     // if a =2
            buy = false;
             break;
}
…
```

Case Statement Rules…

1. Case label must be unique.
2. Case labels must ends with colon.
3. Case labels must have constants / constant expression.
4. Case label must be of integral Type (Integer, Character), e.g. 10, 10+2, ‘j’.
5. Case label should not be ‘floating point number. ‘
6. Switch case should have at most one default label.
7. Default label is Optional.
8. Default can be placed anywhere in the switch.

9. *Break* statement takes control out of the switch.
10. Two or more cases may share one break statement.
11. Nesting (switch within switch) is allowed.
12. Relational Operators are not allowed in Switch Statement.
13. Macro Identifier are allowed as Switch Case Label.
14. Const Variable is allowed in switch Case Statement.
15. Empty Switch case is allowed.

Conditional Operator

condition ? expression1 : expression2

Usually *condition* is a relational expression.

If TRUE, then expression1 is evaluation, if FALSE then expression2 is evaluated.

For example:

```
int s, x
…
s = (x < 0)  ?  -1 : x* x
```

So, if x < 0 then ‘s’ equates to -1, otherwise ‘s’ equates to x^2

Arrays

1. The first element is indexed with zero, e.g. a[3] has 3 elements, a[0], a[1], and a[2].
2. Declare as usual, e.g. int a[3], float a[3], and char a[3].
3. Initialize: int a[3] = {2, 6, 1}.
4. Ok to initialize using a “for” loop.
5. If number of elements is not stated, the initialization will determine it, e.g. int a[ ] = {2, 6, 1} – elements will be three.
6. Arrays may be multidimensional, e.g. a[3, 5].
7. Two dimensional (rows and columns) can also be written, e.g. int M[4] [5] *(remember there is a zero row and column).*
8. Number of elements may be determined by variable – in which case range check first.

Functions

- A group of statements called by your main program or another function.
- Key words – *void, argument, formal parameter and local variables*.
  - *"void"* specifies that the function does not return a value.
  - *Arguments* are values passed to the function.
  - *Formal parameter* is the declared variable in the function that refers to the argument passed to it.
  - *Local variables* are declared and exist only in the function.
  - Multiple arguments are permitted.
- *Recursive* - functions may call themselves. Conceptualize as calling a new function (new local variables).

For example:

```
#include <stdio.h>                    // include library

void circumference (float radius) // function
{ float cir;                              // local variable
  cir = (2 * 3.1415 * radius);
  printf (…, cir);                    // print result
}

int main (void)                        // main program
{ …
  circumference (10.5);             // function call 0
  …
}
```

Returning Function Results…

- *return expression* indicates that the function is to return to the value of expression.
- You must declare the type of value the function will return.
- In the example, the function *circumference* is called with the argument 8.4, and the value returned is *result.*

For example:

```
#include <stdio.h>                    // include library

circumference (float radius)          // function
{float circ;                          // local variable
  circ = (2 * 3.14 * radius);
  return circ
}

int main (void)                       // main program
{float result
 …
  result = circumference (8.4);       // function call
 …
}
```

Global and Static Variables

1. Global variables have initial value of zero even arrays. Local variables must be explicitly initialized.
2. Although global variables reduce the number of arguments that need to be passed to a function, they decrease readability. It is not clear what the function needs as input or produces as output.

Automatic & Static Variables…

- *Local variables* in a function are also called *automatic variables*, meaning they do not retain their value upon leaving the function.
- *Static variables* do not come and go as the function is called. It will have the *same value returning to the function as it had when it left*.
- Static Variables will have a default value of zero.

Static variable example:

```
#include <stdio.h>                          // include library

circumference (float radius)                // function
{ static int itemschecked ;                 // local variable
  float circ;
  cir c= (2 * 3.14 * radius);
  ++itemschecked;
  return circ
}

int main (void)                               // main program
{float result
 …
  result = circumference (8.4);             // function call
 …
}
```

Structures

1) Similar to an array *element*, a structure has *members*.
2) A structure is defined, and then variables are *declared* of that type.
3) The variable name and its members are separated by a *period*.
4) Assign values to each member.
5) Assignment can be done in a single line using *compound literals*.

For example:

```
#include <stdio.h>        // include library
…
int main (void)
{
   struct date                  // defining a new structure type date
   {
      int month;               // members of the structure
      int day;
      int year;
   };
…
struct date today;          // declaring variable today of type struct date

today.month = 7 ;           // values of the variables of today*
today.day = 23;
today.year = 2016;
…
return = 0;
}
          // *could also initialize as:  struct date today = {7, 23, 2016};
               or today = (struct date) {7, 23, 2016};
               or today = (struct date) { .month = 7, .day = 23, .year = 2016};
```


Kochan, S.G. *Programming in C*, 3rd ed., Developer's Library, Indianapolis, Indiana (2005).

Structure Rules..

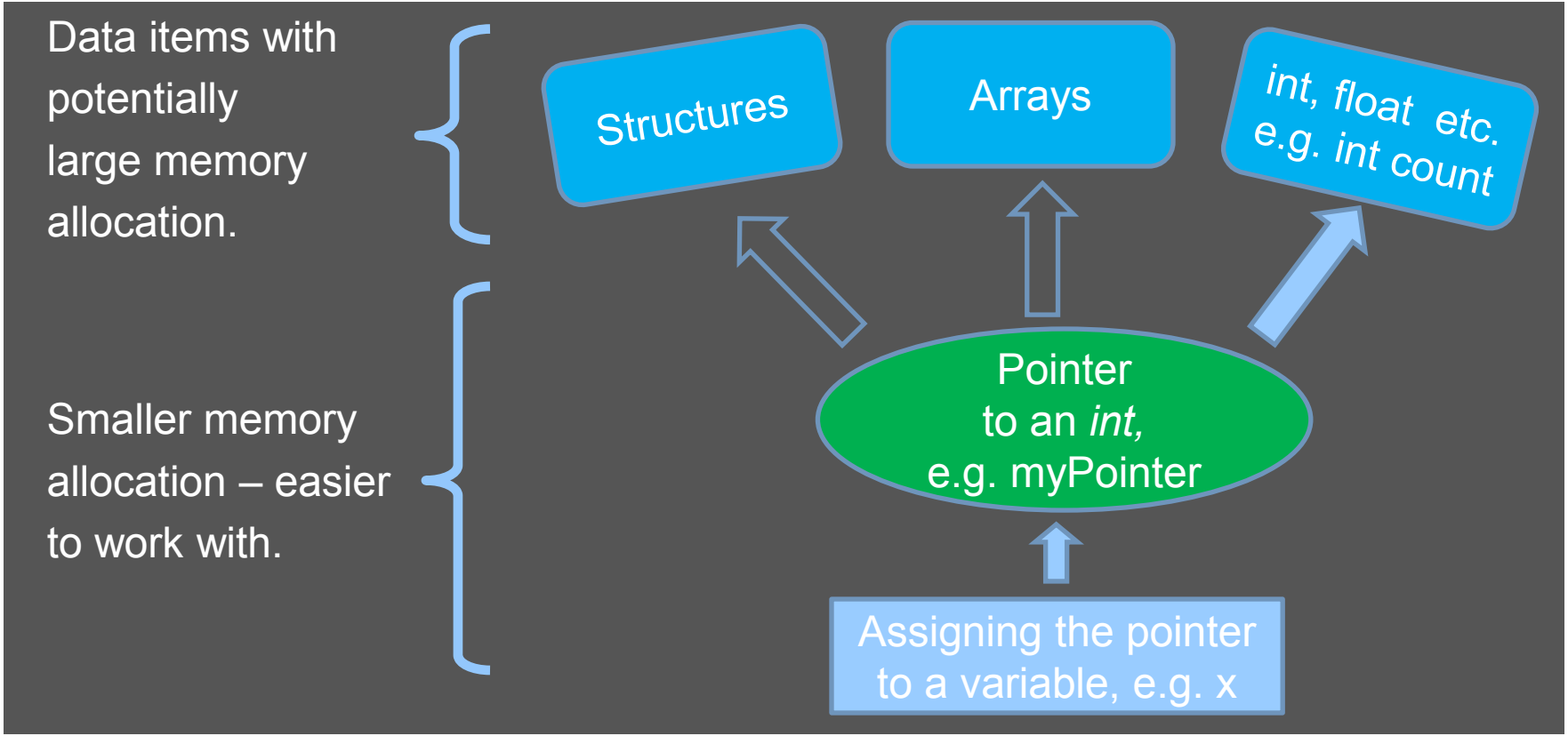
1. Structure members may be used in expressions just as any other variable.
2. Define ahead of your functions, making them global.
3. Structures may be passed as arguments.
   - Any changes made by the function to the values contained in a structure argument have no effect on the original structure.
4. Members may be other structures or arrays.

Character Strings

1. *Double quotation* marks are used to delimit a *character string*: e.g. “Hello world!”
2. Recall the type *char*, and declaration: e.g. *char n*, and assignment *n = ‘t’* (or any other character we would like), in *single quotations*.
3. Any combination of letters, numbers or special characters may be used.
4. Consider an arrays of characters:
   - *char phone [ ]={ ‘P’, ‘h’, ‘o’, ‘n’, ‘e’};*
5. Alternatively this could be written as:
   - *char phone [ ] = {“Phone”};* (curly brackets are optional).
6. If you explicitly size the array, add one place at the end for the *null character*. This character is automatically appended to the end of a string to signal to the compiler that the string has ended.
   - *char phone [6] = “Phone”;*

7. There are various programs you can write to accomplish the following:
   - Concatenating two string.
   - Determining the number of characters in a string.
   - Testing for the equality of two strings.

Pointers


Data items with potentially large memory allocation.
Smaller memory allocation – easier to work with.
Structures
Arrays
int, float etc.
e.g. int count
Pointer
to an int,
e.g. myPointer
Assigning the pointer
to a variable, e.g. x

*Indirect Operator * and Address Operator &…*

1. A pointer allows you an indirect means of accessing the value of a particular data item.
2. The *indirection operator*, *, defines the variable myPointer as a *type pointer* to *int*.
3. The *address operator*, &, is used to make a *pointer* to count.

```
#include<stdio.h>;
int main (void)
{
   int count = 10;
   int *myPointer;              // declaring a pointer to a int
   myPointer = &count;          // set the pointer to count
   x = *myPointer;              // assigning the pointer to x
   printf ("count = %i, x = %i/n", count, x);
   return 0;
}
```

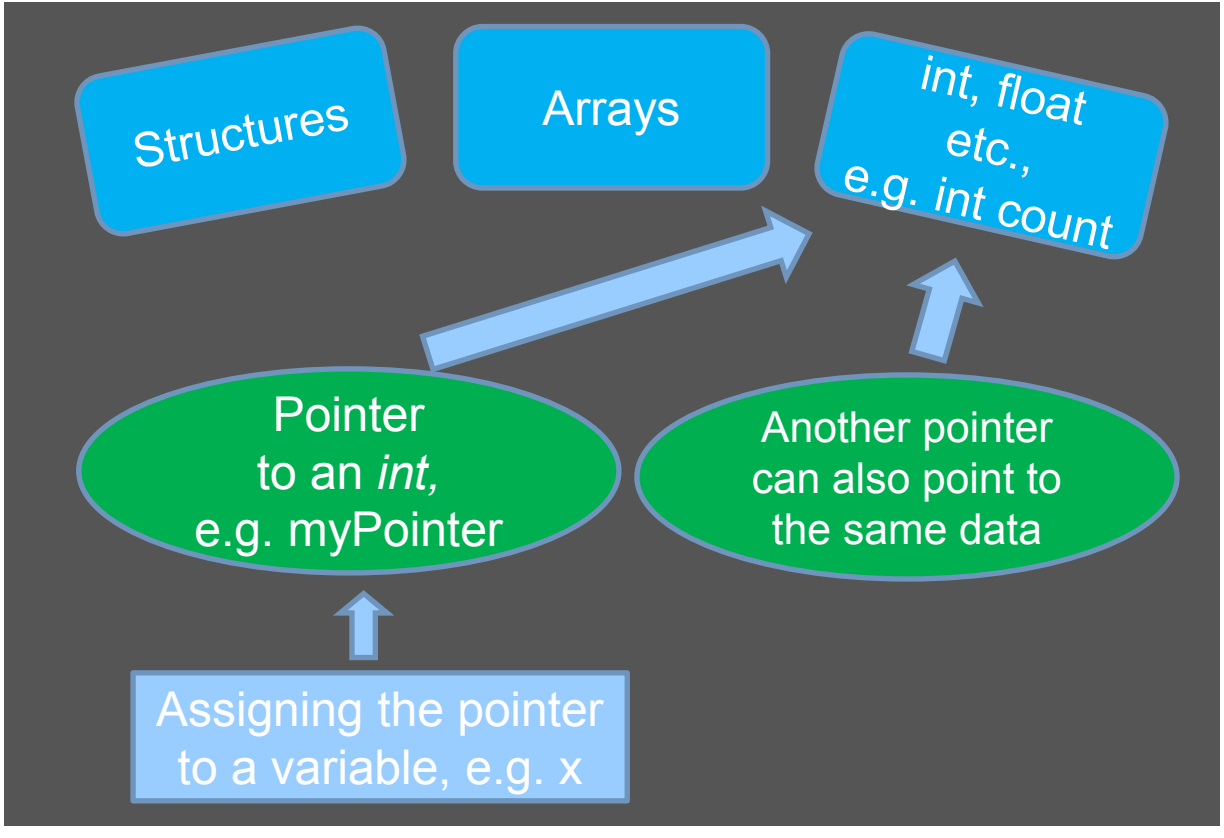
Output: count = 10, x = 10 (we will discuss printf formatting later)

Pointer Example…

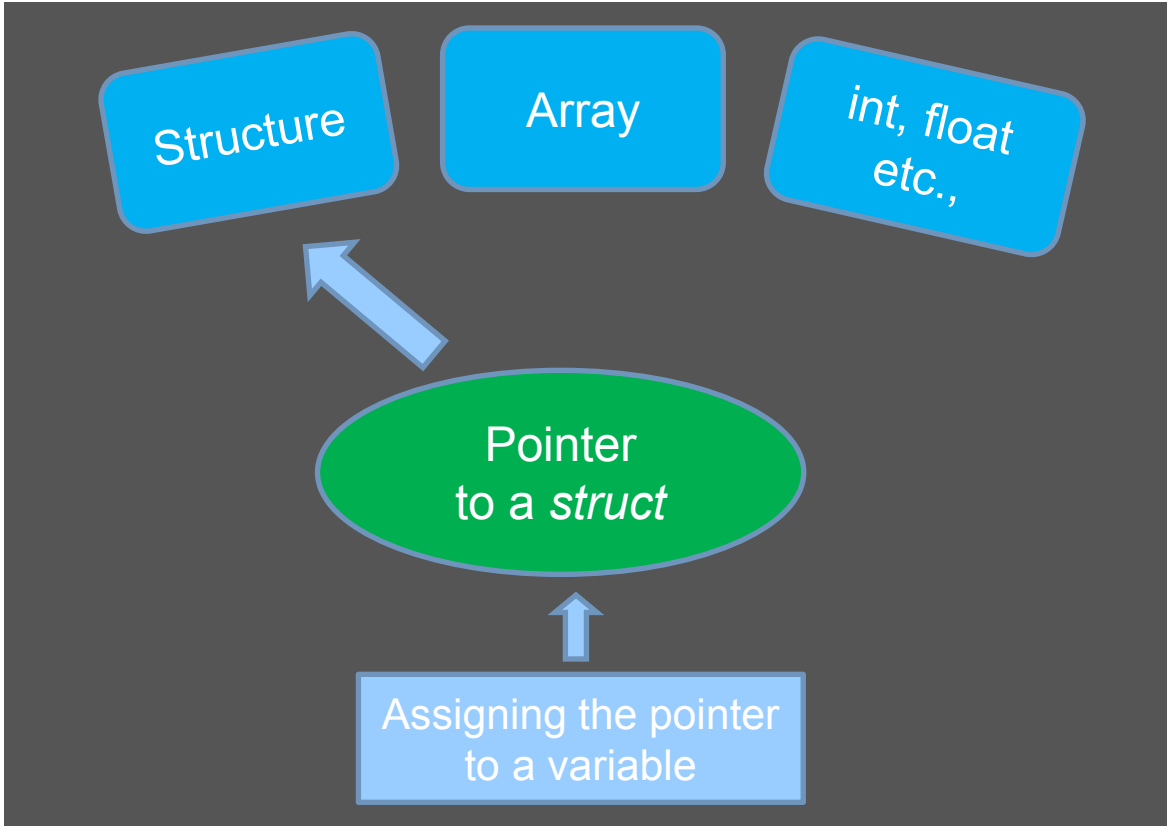
same as:
char *myPointer;
myPointer = &c;

```
#include <stdio.h>
int main (void)
{
   char c = 'Q';                          // Assign "Q" to a new char variable c
   char *myPointer = &c;                  // Declare  myPointer as a pointer to  char, then set it to c.

   printf ("%c %c\n", c, *myPointer);  // Output will be Q Q


c = '/';                                  // Assign "/" to the char variable c
   printf ("%c %c\n", c, *myPointer);  // Output will be //

   *char_pointer = '(';                    // Assign "(" to the char variable c
   printf ("%c %c\n", c, *myPointer);  // Output will be ( (

   return 0;
}
```

More than One Pointer…



Pointer to a Structure…



Example of a Structure Pointer…

```
#include<stdio.h>
int main (void)
{
   struct date
     {
       int month;
       int day;
       int year;
     };

     struct date  today, *datePtr;       //today is type struct, and *datePtr is a pointer to struct
date

     datePtr = &today;                   //setting datePtr to point to today

     datePtr  -> monthly = 9;             //same as saying (*datePtr).month = 9
     datePtr ->  day = 25;
     datePtr -> year = 2004,

     printf ( "Today's date is %i/%i/%.2i.\n",
        datePtr -> month,  datePtr -> day, datePtr -> year % 100);
     return 0;
}
Output: Today's date is 9/25/04.
```


Kochan, S.G. *Programming in C*, 3rd ed., Developer's Library, Indianapolis, Indiana (2005).

Preprocessor Command: #define

- #define - assigns symbolic names to a constant
  - e.g. *#define CARD 6* – defines the name card and assigns a value of 6. (Capitalized is optional)
  - Anywhere (except in a character string) that 'card' is used, it will be substituted by the value 6.
  - May appear anywhere in the program.
  - Examples: *#define PI 3.1415926*, *#define TWO_PI 2.0 * 3.1415926, #define AND && , #define OR ||, or #define EQUALS ==.*

- #define is also known as a *macro* because it can take an argument like a function.

  e.g. #define SQUARE(x) x*x

  y = SQUARE (v); //v^2 is assigned to y

  - The type of the argument is unimportant.
  - Becomes resident in the program (more memory but faster execution).

… and *#include*

- A method of grouping all of your macros together into a separate file, then including them into your program. Typically placed at the beginning. Examples: *<stdio.h>, <float.h>, <limit.h>*
- These files end with *.h*
- May be contained in a *libraries folder* when working with Arduino and other microcontrollers.
- Placing in < > tells the compiler to look for the file in a specific location.
- Once created, they can be used in any program.

Summary

- Flowchart a problem for easier coding
- Relational operators and arithmetic expressions
- Variables and data types
- Statements
  - Loop statements – for, while, do
  - Decision statements – if, if-else, switch-case
- Arrays
- Functions
- Structures
- Pointers
- Preprocessor Commands