YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

1224 CPL

Embedded C Programming Introduction to The C Programming Language

Agenda

- History of C
- Fundamentals of C
  - Data Types
  - Variables, Constants and Arrays
  - Keywords
  - Functions (Overview)
  - Declarations
  - printf() Library Function (Special use in this class)

Agenda

- Operators and Conditional Statements
- Statements and Expressions
- Control Statements: Making Decisions
- Functions
- Program Structure
- Arrays and Strings
- Pointers and Strings
- Structures and Unions
- Additional Features of C

Section 1.0
Using C in an Embedded Environment

Just the Facts

- C was developed in 1974 in order to write the UNIX operating system
- C is more "low level" than other high level languages (good for MCU programming)
- C is supported by compilers for a wide variety of MCU architectures
- C can do ***almost*** anything assembly language can do
- C is usually easier and faster for writing code than assembly language

Busting the Myths

The truth shall set you free…

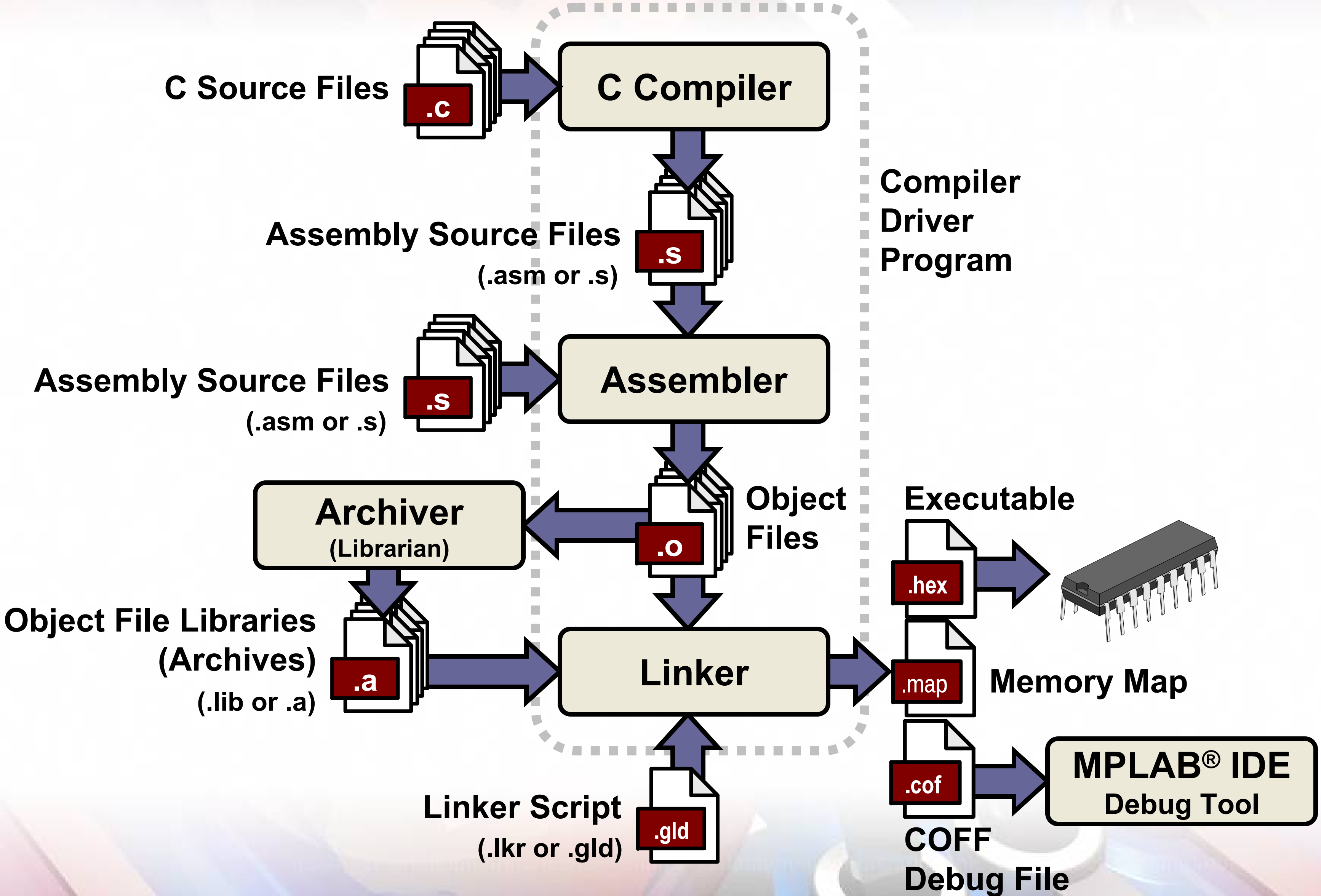
- **C is not as portable between architectures or compilers as everyone claims**
  - **ANSI language features ARE portable**
  - **Processor-specific libraries are NOT portable**
  - **Processor-specific code (peripherals, I/O, interrupts, special features) are NOT portable**
- **C is NOT as efficient as assembly**
  - **A *good* assembly programmer can *usually* do better than the compiler, no matter what the optimization level – C WILL use more memory**

Busting the Myths

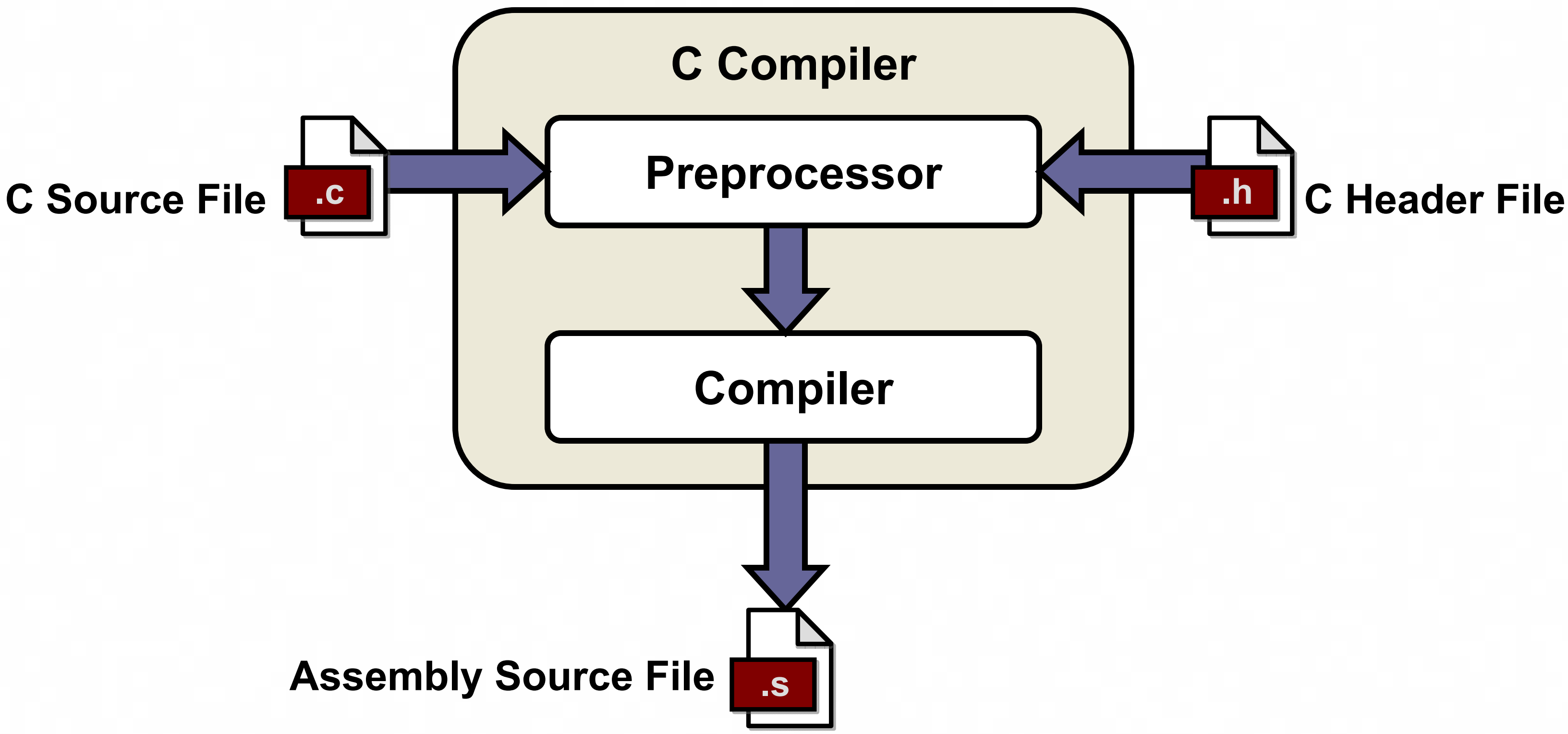
The truth shall set you free…

- **There is NO SUCH THING as self-documenting code – despite what many C proponents will tell you**
  - **C makes it possible to write very confusing code – just search the net for obfuscated C code contests… (www.ioccc.org)**
  - **Not every line needs to be commented, but most *blocks* of code should be**
- **Because of many shortcuts available, C is not always friendly to new users – hence the need for comments!**

Development Tools Data Flow



Development Tools Data Flow



C Runtime Environment

- C Compiler sets up a runtime environment
  - Allocates space for stack
  - Initialize stack pointer
  - Allocates space for heap
  - Copies values from Flash/ROM to variables in RAM that were declared with initial values
  - Clear uninitialized RAM
  - Disable all interrupts
  - Call main() function (where your code starts)

C Runtime Environment

- Runtime environment setup code is automatically linked into application by most PIC® MCU compiler suites
- *Usually* comes from either:
  - crt0.s / crt0.o (crt = C RunTime)
  - startup.asm / startup.o
- User modifiable if absolutely necessary
- Details will be covered in compiler specific classes

Fundamentals of C

A Simple C Program

Example

Preprocessor Directives

Header File

```c
#include <stdio.h>

#define PI 3.14159

int main(void)
{
    float radius, area;

    //Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = %f", area);
}
```

Constant Declaration (Text Substitution Macro)

Variable Declarations

Comment

Function

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.1
Comments

Comments

Definition

Comments are used to document a program's functionality and to explain what a particular block or line of code does. Comments are ignored by the compiler, so you can type anything you want into them.

- **Two kinds of comments may be used:**
  - **Block Comment**

```
/* This is a comment */
```

  - **Single Line Comment**

```
// This is also a comment
```

Comments

Using Block Comments

- **Block comments:**
  - **Begin with `/*` and end with `*/`**
  - **May span multiple lines**

```
/****************************************************************
* Program: hello.c
* Author:  R. Ostapiuk
****************************************************************/
#include <stdio.h>

/* Function: main() */
int main(void)
{
  printf("Hello, world!\n"); /* Display "Hello, world!" */
}
```

Comments

Using Single Line Comments

- **Single line comments:**
  - **Begin with // and run to the end of the line**
  - **May *not* span multiple lines**

```
//=======================================================
// Program: hello.c
// Author:  R. Ostapiuk
//=======================================================
#include <stdio.h>

// Function: main()
int main(void)
{
  printf("Hello, world!\n"); // Display "Hello, world!"
}
```

Comments

Nesting Comments

- **Block comments may not be nested within other delimited comments**
- **Single line comments may be nested**

Example: Single line comment within a delimited comment.

```
/*
    code here       // Comment within a comment
*/
```

Example: Delimited comment within a delimited comment.

Delimiters don't match up as intended!

```
/*
    code here       /* Comment within a comment */
    code here       /* Comment within a… oops!  */
*/
```

Dangling delimiter causes compile error

Comments

Best Practices

```c
/****************************************************************
* Program: hello.c
* Author:  R. Ostapiuk
****************************************************************/
#include <stdio.h>

/****************************************************************
* Function: main()
****************************************************************/
int main(void)
{
  /*
  int i;                          // Loop count variable
  char *p;                        // Pointer to text string
  */

  printf("Hello, world!\n"); // Display "Hello, world!"
}
```

Section 1.2
Variables, Identifiers, and Data Types

Variables and Data Types

A Simple C Program

Example

```
#include <stdio.h>

#define PI 3.14159

int main(void)
{
    float radius, area;

    //Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = %f", area);
}
```

Data Types → `int`, `float`

Variable Declarations → `float radius, area;`

Variables in use → `area = PI * radius * radius;`, `printf("Area = %f", area);`

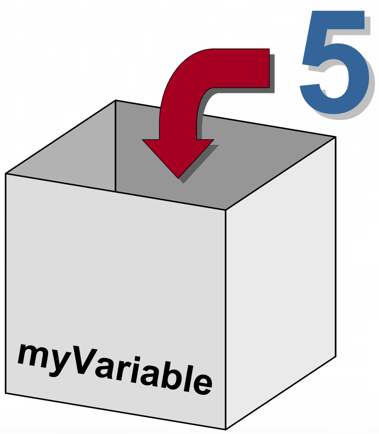
Variables

Definition

A <u>variable</u> is a name that represents one or more memory locations used to hold program data.

- A variable may be thought of as a container that can hold data used in a program

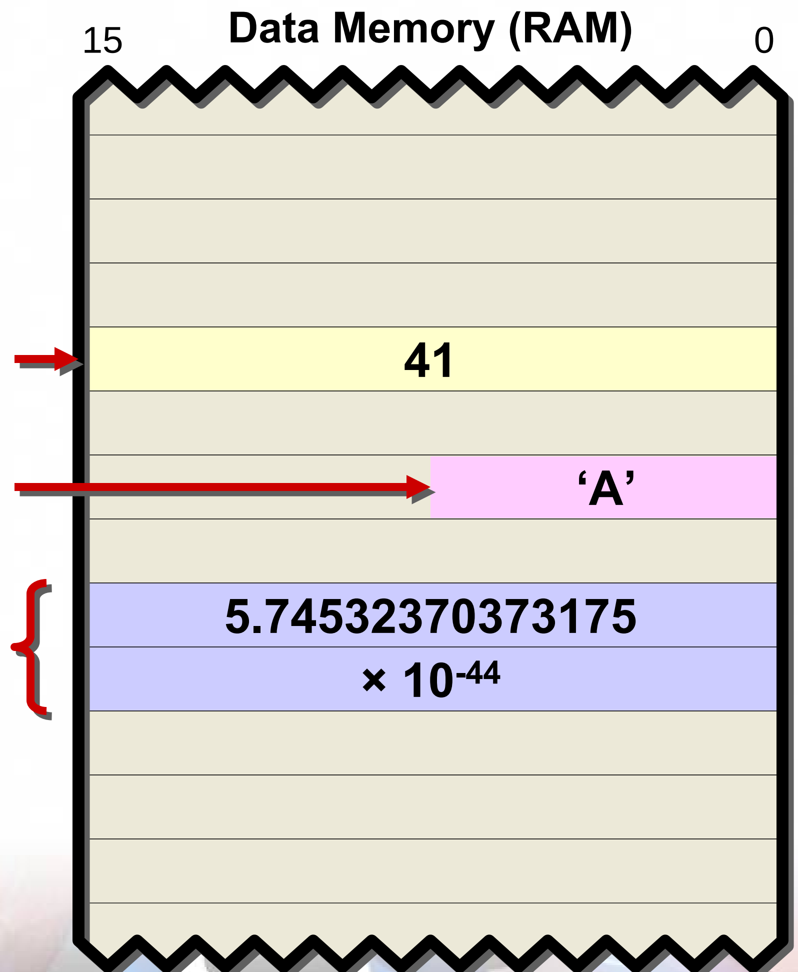
```
int myVariable;

myVariable = 5;
```



Variables

- **Variables are names for storage locations in memory**

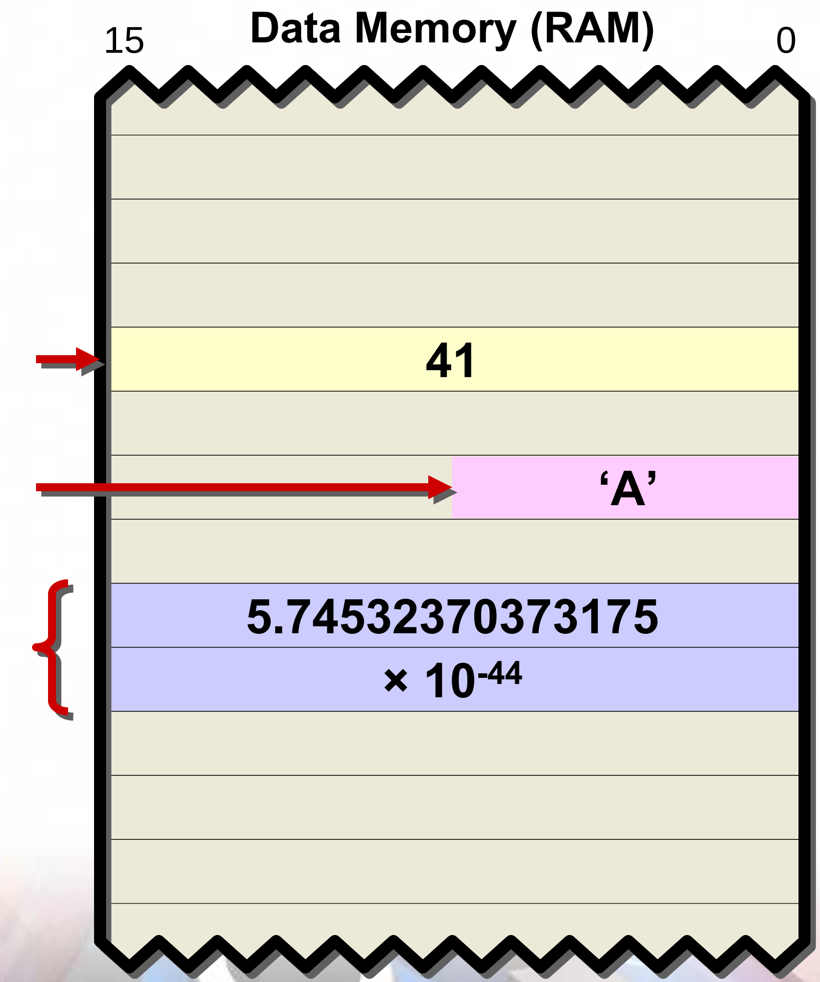
```
int warp_factor;
char first_letter;
float length;
```



Variables

- **Variable declarations consist of a unique <u>identifier</u> (name)…**

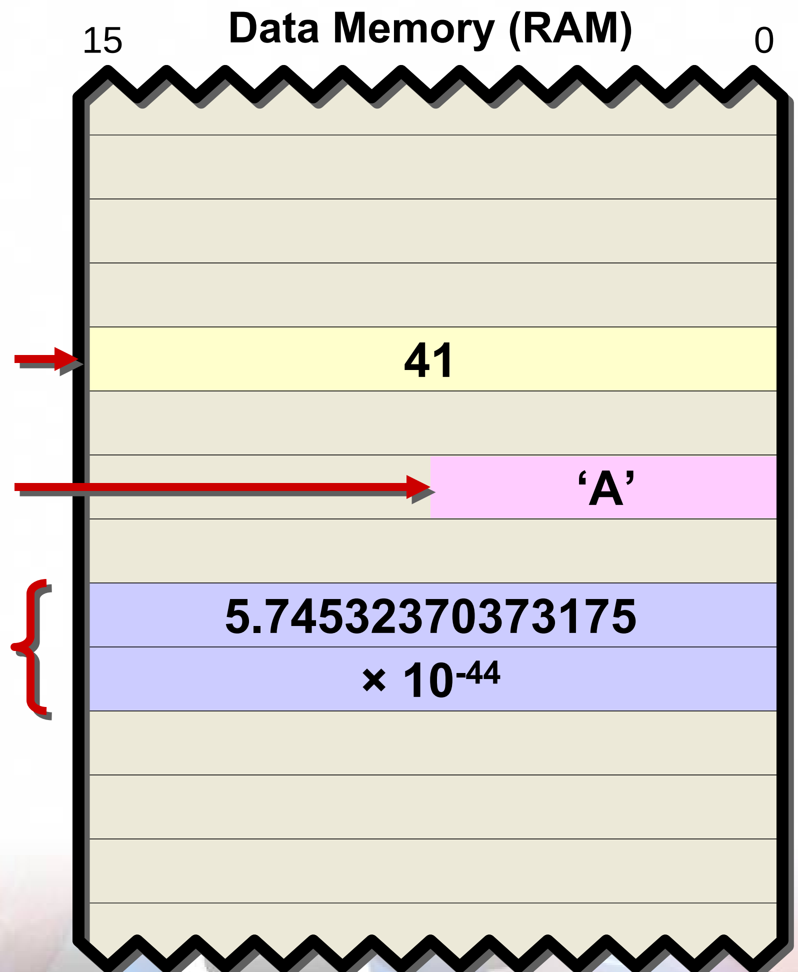
```
int warp_factor;
char first_letter;
float length;
```



Variables

- ...and a data type
  - Determines size
  - Determines how values are interpreted

```
int warp_factor;
char first_letter;
float length;
```



Identifiers

- Names given to program elements such as:
  - Variables
  - Functions
  - Arrays
  - Other elements

Identifiers

- **Valid characters in identifiers:**

`I d e n t i f i e r`

First Character
'_' (underscore)
'A' to 'Z'
'a' to 'z'

Remaining Characters
'_' (underscore)
'A' to 'Z'
'a' to 'z'
'0' to '9'

- **Case sensitive!**
- **Only first 31 characters significant***

ANSI C Keywords

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

- **Some compiler implementations may define additional keywords**

Data Types

Fundamental Types

Type	Description	Bits
`char`	single character	8
`int`	integer	16
`float`	single precision floating point number	32
`double`	double precision floating point number	64

The size of an `int` varies from compiler to compiler.

- MPLAB® C30 `int` is 16-bits
- MPLAB C18 `int` is 16-bits
- CCS PCB, PCM & PCH `int` is 8-bits
- Hi-Tech PICC `int` is 16-bits

Data Type Qualifiers

Modified Integer Types

Qualifiers: `unsigned`, `signed`, `short` **and** `long`

Qualified Type	Min	Max	Bits
`unsigned char`	0	255	8
`char, signed char`	-128	127	8
`unsigned short int`	0	65535	16
`short int, signed short int`	-32768	32767	16
`unsigned int`	0	65535	16
`int, signed int`	-32768	32767	16
`unsigned long int`	0	2^{32}-1	32
`long int, signed long int`	-2^{31}	2^{31}	32
`unsigned long long int`	0	2^{64}-1	64
`long long int, signed long long int`	-2^{31}	2^{31}	64

Data Type Qualifiers

Modified Floating Point Types

Qualified Type	Absolute Min	Absolute Max	Bits
`float`	$\pm\sim10^{-44.85}$	$\pm\sim10^{38.53}$	32
`double` (1)	$\pm\sim10^{-44.85}$	$\pm\sim10^{38.53}$	32
`long double`	$\pm\sim10^{-323.3}$	$\pm\sim10^{308.3}$	64

MPLAB® C30: (1)`double` is equivalent to `long double` if `–fno-short-double` is used

MPLAB C30 Uses the IEEE-754 Floating Point Format
MPLAB C18 Uses a modified IEEE-754 Format

Variables

How to Declare a Variable

Syntax

```
type identifier1, identifier2,…,identifiern;
```

- **A variable must be declared before it can be used**
- **The compiler needs to know how much space to allocate and how the values should be handled**

Example

```
int x, y, z;
float warpFactor;
char text_buffer[10];
unsigned index;
```

Variables

How to Declare a Variable

Variables may be declared in a few ways:

Syntax

One declaration on a line

```
type identifier;
```

One declaration on a line with an initial value

```
type identifier = InitialValue;
```

Multiple declarations of the same type on a line

$type\ identifier_1,\ identifier_2,\ identifier_3;$

Multiple declarations of the same type on a line with initial values

$type\ identifier_1 = Value_1,\ identifier_2 = Value_2;$

Variables

How to Declare a Variable

Examples

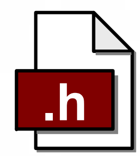
```
unsigned int x;
unsigned y = 12;
int a, b, c;
long int myVar = 0x12345678;
long z;
char first = 'a', second, third = 'c';
float big_number = 6.02e+23;
```

It is customary for variable names to be spelled using "camel case", where the initial letter is lower case. If the name is made up of multiple words, all words after the first will start with an upper case letter (e.g. `myLongVarName`).

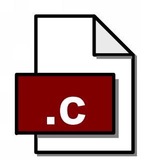
Variables

How to Declare a Variable

- Sometimes, variables (and other program elements) are declared in a separate file called a <u>header file</u>
- Header file names customarily end in .h
- Header files are associated with a program through the `#include` directive



MyProgram.h



MyProgram.c

#include Directive

- Three ways to use the #include directive:

Syntax

```
#include <file.h>
```

Look for file in the compiler search path
The compiler search path usually includes the compiler's directory and all of its subdirectories.
For example: C:\Program Files\Microchip\MPLAB C30*.*

```
#include "file.h"
```

Look for file in project directory only

```
#include "c:\MyProject\file.h"
```

Use specific path to find include file

#include Directive

main.h Header File and main.c Source File

main.h

```
unsigned int a;
unsigned int b;
unsigned int c;
```

main.c

```
#include <main.h>

int main(void)
{
        a = 5;
        b = 2;
        c = a+b;
}
```

The contents of main.h are *effectively* pasted into main.c starting at the `#include` directive's line

#include Directive

Equivalent main.c File

- After the preprocessor runs, this is how the compiler sees the main.c file
- The contents of the header file aren't *actually* copied to your main source file, but it will behave *as if* they were copied

main.c

```
unsigned int a;
unsigned int b;
unsigned int c;

int main(void)
{
    a = 5;
    b = 2;
    c = a+b;
}
```

Equivalent main.c file without #include

Lab 01

Variables and Data Types

On the CD

...\101_ECP\Lab01\Lab01.mcw

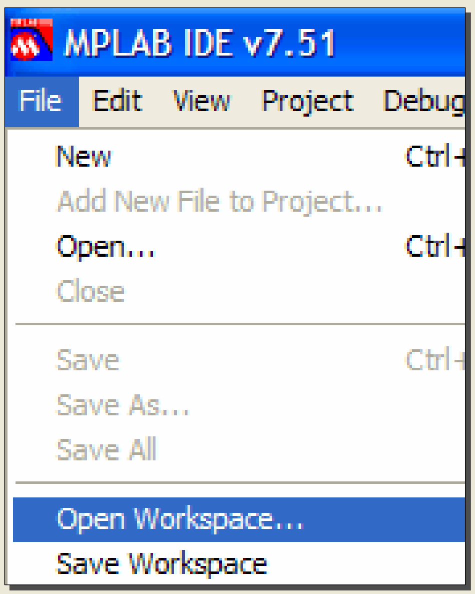
Lab 01

Variables and Data Types

- Open the project's workspace:

On the lab PC

C:\RTC\101_ECP\Lab01\Lab01.mcw



1. **Open MPLAB® IDE and select Open Workspace... from the File menu. Open the file listed above.**

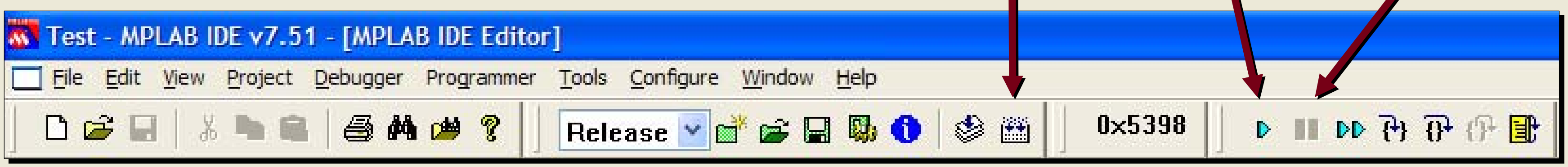
⚠ **If you already have a project open in MPLAB IDE, close it by selecting Close Workspace from the File menu before opening a new one.**

Lab 01

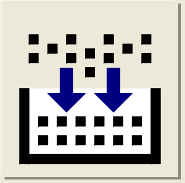
Variables and Data Types

- **Compile and run the code:**

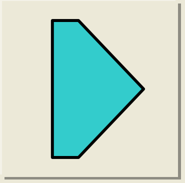
2 Compile (Build All) 3 Run 4 Halt



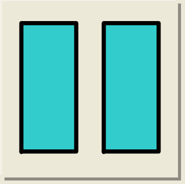
2 **Click on the Build All button.**



3 **If no errors are reported, click on the Run button.**



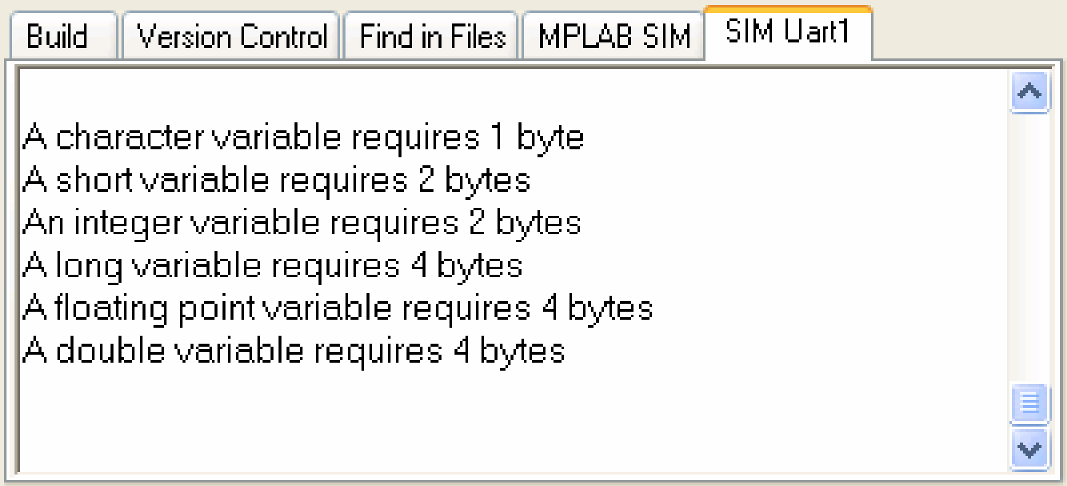
4 **Click on the Halt button.**



Lab 01

Variables and Data Types

- **Expected Results (1):**



5 **The** SIM Uart1 **window should show the text that is output by the program, indicating the sizes of C's data types in bytes.**

Lab 01

Variables and Data Types

- **Expected Results (2):**

Add SFR | AD1CHS | Add Symbol | __C30_UART

Address	Symbol Name	Value	Decimal
08AA	charVariable	0x32	50
08AC	shortVariable	0x0032	50
08AE	intVariable	0x0032	50
08B0	longVariable	0x00000032	50
08B4	floatVariable	50.00000	1112014848
08B8	doubleVariable	50.00000	1112014848

6 The watch window should show the values which are stored in the variables and make it easier to visualize how much space each one requires in data memory (RAM).

Lab 01

Variables and Data Types

16-bit Data Memory

7 Variables in Memory

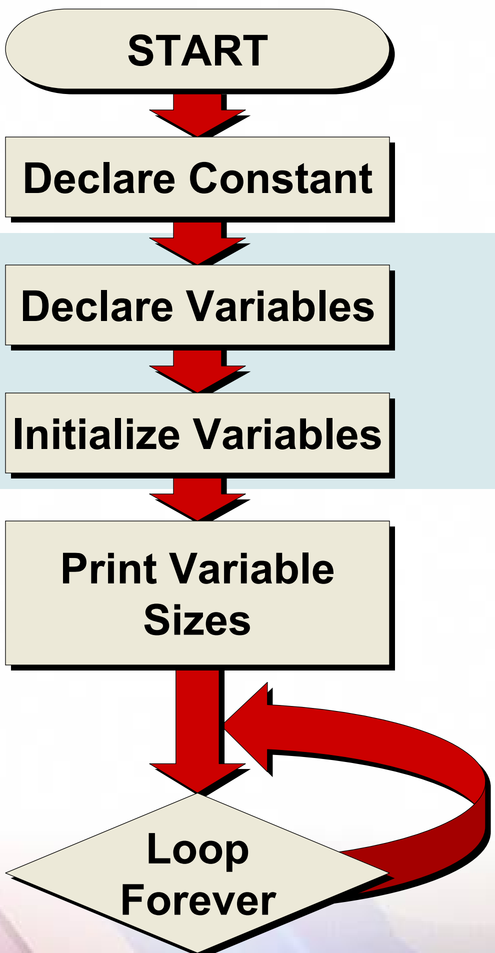
Odd address	High byte	Low byte	Even address	Variable
0x08A9			0x08A8	
0x08AB		32	0x08AA	char
0x08AD	00	32	0x08AC	short int
0x08AF	00	32	0x08AE	int
0x08B1	00	32	0x08B0	long int
0x08B3	00	00	0x08B2	long int
0x08B5	42	48	0x08B4	float
0x08B7	00	00	0x08B6	float
0x08B9	42	48	0x08B8	double
0x08BB	00	00	0x08BA	double
0x08BD			0x08BC	

Multi-byte values stored in "Little Endian" format on PIC® microcontrollers

Lab 01

Variables and Data Types

- **What does the code do?**



Example lines of code from the demo program:

```c
#define  CONSTANT1  50

int  intVariable;

intVariable = CONSTANT1;

printf("\nAn integer variable
        requires %d bytes.",
        sizeof(int));

while(1);
```

Lab 01

Conclusions

- Variables must be declared before used
- Variables must have a data type
- Data type determines memory use
- Most efficient data types:
  - int on 16-bit architectures*
  - char on 8-bit architectures
- Don't use float/double unless you really need them

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.3
Literal Constants

A Simple C Program

Literal Constants

Example

```
unsigned int a;
unsigned int c;
#define b 2

void main(void)
{
   a = 5;
   c = a + b;
   printf("a=%d, b=%d, c=%d\n", a, b, c);
}
```

Literal (← 2)

Literal (← 5)

Literal Constants

Definition

A literal or a literal constant is a value, such as a number, character or string, which may be assigned to a variable or a constant. It may also be used directly as a function parameter or an operand in an expression.

- Literals
  - Are "hard coded" values
  - May be numbers, characters or strings
  - May be represented in a number of formats (decimal, hexadecimal, binary, character, etc.)
  - Always represent the same value (5 always represents the quantity five)

Constant vs. Literal

What's the difference?

- Terms are used interchangeably in most programming literature
- A literal is a constant, but a constant is not a literal
  - `#define MAXINT 32767`
  - `const int MAXINT = 32767;`
- For purposes of this presentation:
  - Constants are labels that represent a literal
  - Literals are values, often assigned to symbolic constants and variables

Literal Constants

- **Four basic types of literals:**
  - **Integer**
  - **Floating Point**
  - **Character**
  - **String**
- **Integer and Floating Point are *numeric type* constants:**
  - **Commas and spaces are not allowed**
  - **Value cannot exceed type bounds**
  - **May be preceded by a minus sign**

Integer Literals

Decimal (Base 10)

- **Cannot start with 0 (except for 0 itself)**
- **Cannot include a decimal point**
- **Valid Decimal Integers:**

```
0   5   127   -1021   65535
```

- **Invalid Decimal Integers:**

```
32,767   25.0   1 024   0552
```

Integer Literals

Hexadecimal (Base 16)

- Must begin with **0x** or **0X** (that's zero-x)
- May include digits 0-9 and A-F / a-f
- Valid Hexadecimal Integers:

```
0x  0x1    0x0A2B     0xBEEF
```

- Invalid Hexadecimal Integers:

```
0x5.3     0EA12     0xEG    53h
```

Integer Literals

Octal (Base 8)

- Must begin with **0** (zero)
- May include digits 0-7
- Valid Octal Integers:

```
0    01    012    073125
```

- Invalid Octal Integers:

```
05.3    0o12    080    53o
```

> While Octal is still part of the ANSI specification, almost no one uses it anymore.

Integer Literals

Binary (Base 2)

- Must begin with **0b** or **0B** (that's zero-b)
- May include digits 0 and 1
- Valid Binary Integers:

```
0b      0b1      0b0101001100001111
```

- Invalid Binary Integers:

```
0b1.0    01100    0b12    10b
```

> ⚠ **ANSI C does *not* specify a format for binary integer literals. However, this notation is supported by most compilers.**

Integer Literals
Qualifiers

- **Like variables, literals may be qualified**
- **A suffix is used to specify the modifier**
  - **'U' or 'u' for unsigned: `25u`**
  - **'L' or 'l' for long: `25L`**
- **Suffixes may be combined: `0xF5UL`**
  - **Note: `U` must precede `L`**
- **Numbers without a suffix are assumed to be signed and short**
- **Not required by all compilers**

Floating Point Literals

Decimal (Base 10)

- **Like decimal integer literals, but decimal point is allowed**
- **'e' notation is used to specify exponents** ($k\text{e}\pm\text{n} \Rightarrow k \cdot 10^{\pm\text{n}}$)
- **Valid Floating Point Literals:**

```
2.56e-5   10.4378   48e8   0.5
```

- **Invalid Floating Point Literals:**

```
0x5Ae-2    02.41      F2.33
```

Character Literals

- Specified within single quotes (`'`)
- May include any single printable character
- May include any single non-printable character using escape sequences (e.g. `'\0'` = NULL) (also called digraphs)
- Valid Characters: `'a'`, `'T'`, `'\n'`, `'5'`, `'@'`, `' '` (space)
- Invalid Characters: `'me'`, `'23'`, `'''`

String Literals

- Specified within double quotes (")
- May include any printable or non-printable characters (using escape sequences)
- Usually terminated by a null character ‘\0’
- Valid Strings: `"Microchip"`, `"Hi\n"`, `"PIC"`, `"2500"`,`"rob@microchip.com"`, `"He said, \"Hi\""`
- Invalid Strings: `"He said, "Hi""`

String Literals
Declarations

- Strings are a special case of <u>arrays</u>
- If declared without a dimension, the null character is automatically appended to the end of the string:

Example 1

```
char color[3] = "RED";
```

Is stored as:

```
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
```

Example 2

```
char color[] = "RED";
```

Is stored as:

```
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
color[3] = '\0'
```

String Literals

How to Include Special Characters in Strings

Escape Sequence	Character	ASCII Value
`\a`	BELL (alert)	7
`\b`	Backspace	8
`\t`	Horizontal Tab	9
`\n`	Newline (Line Feed)	10
`\v`	Vertical Tab	11
`\f`	Form Feed	12
`\r`	Carriage Return	13
`\"`	Quotation Mark (")	34
`\'`	Apostrophe/Single Quote (')	39
`\?`	Question Mark (?)	63
`\\`	Backslash (\)	92
`\0`	Null	0

String Literals

How to Include Special Characters in Strings

Example

```
char message[] = "Please enter a command…\n"
```

- This string includes a newline character
- Escape sequences may be included in a string like any ordinary character
- The backslash plus the character that follows it are considered a single character and have a single ASCII value

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.4
Symbolic Constants

Symbolic Constants

Definition

A constant or a symbolic constant is a label that represents a literal. Anywhere the label is encountered in code, it will be interpreted as the value of the literal it represents.

- Constants
  - Once assigned, never change their value
  - Make development changes easy
  - Eliminate the use of "magic numbers"
  - Two types of constants
    - Text Substitution Labels
    - Variable Constants (!!??)

Symbolic Constants

Constant Variables Using `const`

- Some texts on C declare constants like:

Example

```
const float PI = 3.141593;
```

- This is not efficient for an embedded system: A variable is allocated in program memory, but it cannot be changed due to the `const` keyword
- This is not the traditional use of `const`
- In the vast majority of cases, it is better to use `#define` for constants

Symbolic Constants

Text Substitution Labels Using #define

- **Defines a text substitution label**

Syntax

```
#define label text
```

- **Each instance of *label* will be replaced with *text* by the *preprocessor* unless *label* is inside a string**
- **No memory is used in the microcontroller**

Example

```
#define PI 3.14159
#define mol 6.02E23
#define MCU "PIC24FJ128GA010"
#define COEF 2 * PI
```

Symbolic Constants

#define Gotchas

- Note: a `#define` directive is **NEVER** terminated with a semi-colon (;), unless you want that to be part of the text substitution.

Example

```
#define MyConst 5;

c = MyConst + 3;

c = 5; + 3;
```

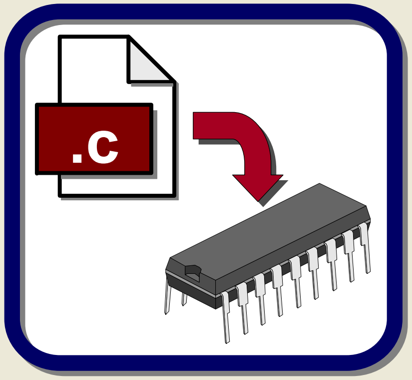
Symbolic Constants

Initializing Variables When Declared

- A constant declared with `const` may not be used to initialize a variable when it is declared

Example

```
#define CONSTANT1 5
const CONSTANT2 = 10;

int variable1 = CONSTANT1;
int variable2;
// Cannot do: int variable2 = CONSTANT2
```



Lab 02

Symbolic Constants

On the CD

...\101_ECP\Lab02\Lab02.mcw

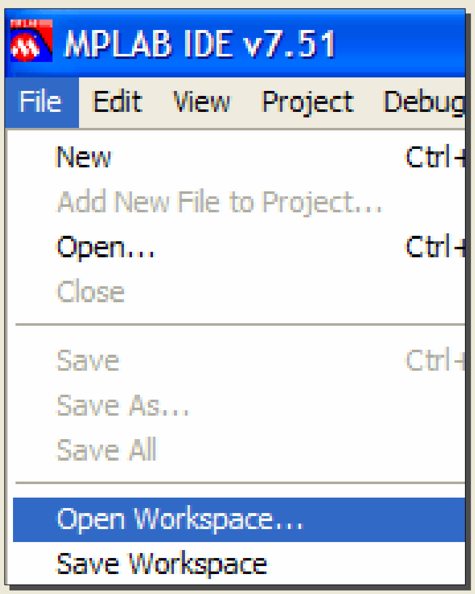
Lab 02

Symbolic Constants

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab02\Lab02.mcw



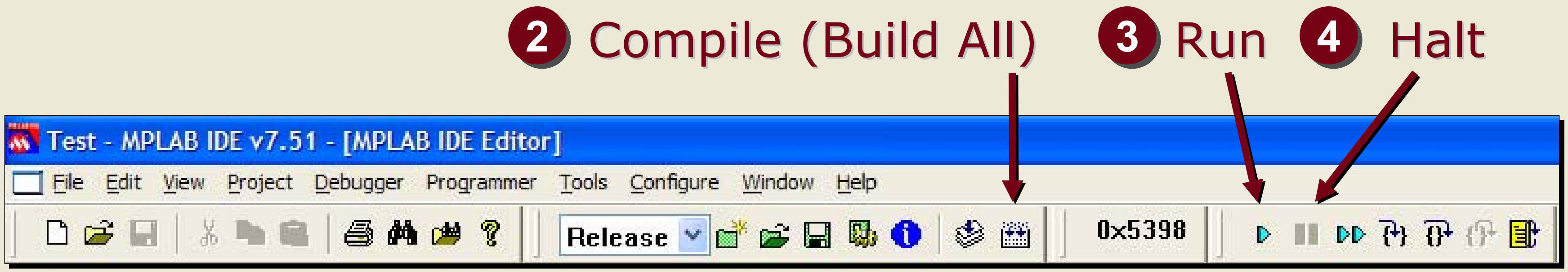
1. **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

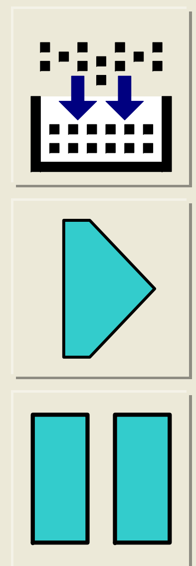
⚠ **If you already have a project open in MPLAB IDE, close it by selecting** Close Workspace **from the** File **menu before opening a new one.**

Lab 02

Symbolic Constants

- **Compile and run the code:**





2 **Click on the Build All button.**

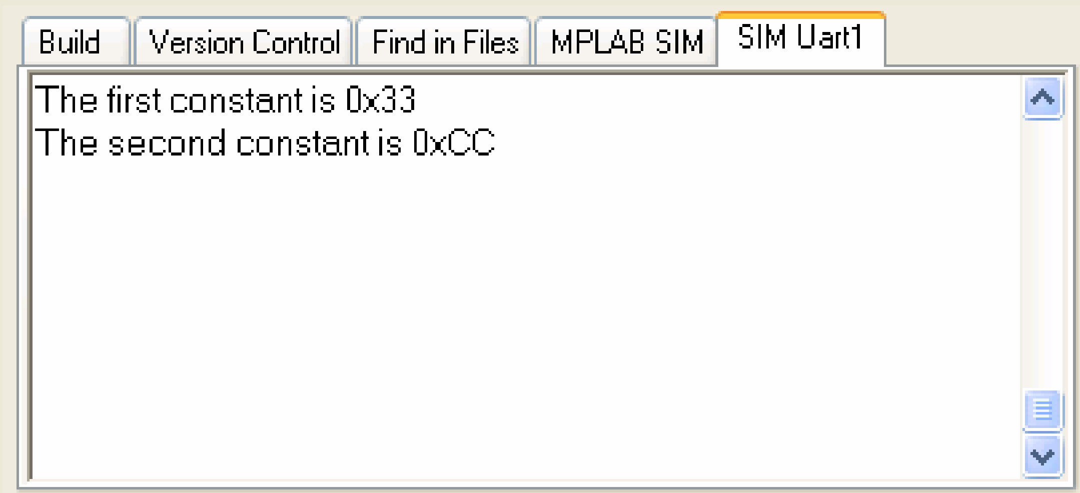
3 **If no errors are reported, click on the Run button.**

4 **Click on the Halt button.**

Lab 02

Symbolic Constants

- **Expected Results (1):**



5 **The** SIM Uart1 **window should show the text that is output by the program, indicating the values of the two symbolic constants in the code.**

Lab 02

Symbolic Constants

- **Expected Results (2):**

CONSTANT1 has <u>no</u> address

CONSTANT2 has a program memory address (P)

Add SFR | AD1CHS | Add Symbol | _C30_UART

	Address	Symbol Name	Value	Decimal
	08C2	variable1	0x0033	51
	08C4	variable2	0x00CC	204
		CONSTANT1	0x33	51
P	011D0	CONSTANT2	0x00CC	204

6 **The watch window should show the two symbolic constants declared in code. CONSTANT1 was declared with `#define`, and therefore uses no memory. CONSTANT2 was declared with `const` and is stored as an immutable variable in Flash program memory.**

Lab 02

Symbolic Constants

- **Expected Results (3):**

Address					ASCII
011A8	97B06F	97B0FF	97B90F	97B99F	o....... ....
011B0	060000	FA0000	848E80	884620	.... F..
011B8	200330	781F80	291D20	781F80	0. ...x. .)...x.
011C0	07F873	5787E4	F891D0	291EE0	s.....W. ......).
011C8	781F80	07F86E	5787E4	37FFFF	..x.n... ..W...7.
011D0	0000CC	006854	002065	006966	....Th.. e ..fi..
011D8	007372	002074	006F63	00736E	rs..t .. co..ns..
011E0	006174	00746E	006920	002073	ta..nt.. i..s ..
011E8	007830	005825	00000A	006854	0x..%X.. ....Th..
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Opcode Hex | Machine | Symbolic | PSV Mixed | PSV Data

7 **If we look in the program memory window, we can find CONSTANT2 which was created with `const` at address 0x011D0 (as was shown in the watch window)**

Lab 02

Symbolic Constants

- **Expected Results (4):**

lab02.map

```
External Symbols in Program Memory (by name):

        0x0011d0                 _CONSTANT2
        0x000e16                 __Atexit
        0x000b9c                 __Closreg
        0x00057c                 __DNKfflush
        0x0012d8                 __DefaultInterrupt
```

CONSTANT1 does not appear anywhere in the map file

8 **If we open the map file (in the lab02 project directory), we can see that memory has been allocated for CONSTANT2 at 0x011D0, but nothing has been allocated for CONSTANT1.**

Lab 02
Conclusions

- Constants make code more readable
- Constants improve maintainability
- #define should be used to define constants
- #define constants use no memory, so they may be used freely
- const should never be used in this context (it has other uses…)

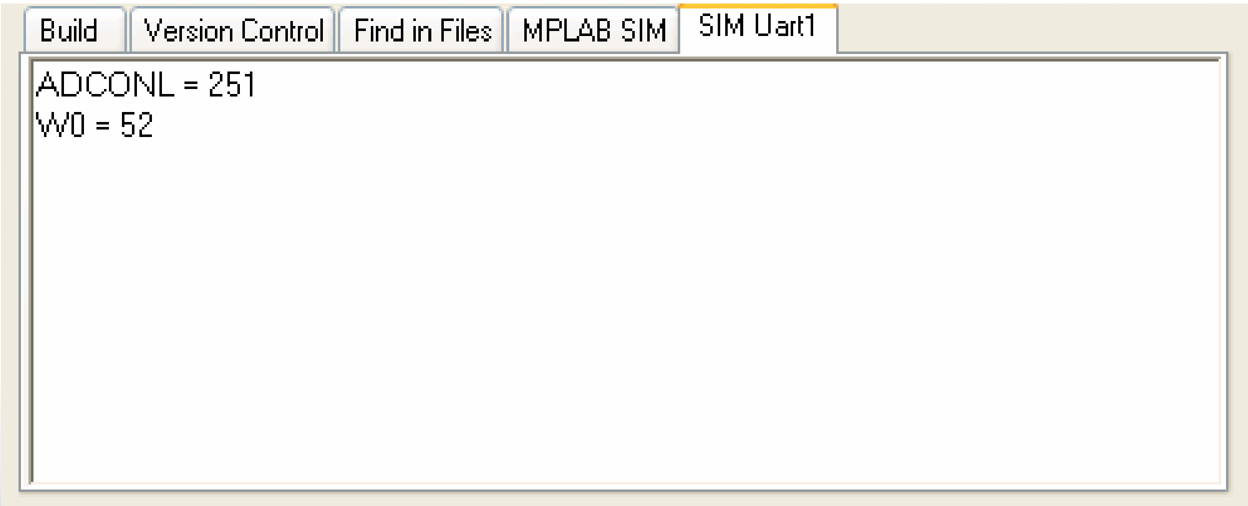
YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.5
printf() Function

printf()

Standard Library Function

- Used to write text to the "standard output"
- Normally a computer monitor or printer
- Often the UART in embedded systems
- SIM Uart1 window in MPLAB® SIM



printf()

Standard Library Function

Syntax

```
printf(ControlString, arg1,…argn);
```

- **Everything printed verbatim within string except %d's which are replaced by the argument values from the list**

Example

```
int a = 5, b = 10;
printf("a = %d\nb = %d\n", a, b);
```

Result:

```
a = 5
b = 10
```

printf()

Conversion Characters for Control String

Conversion Character	Meaning
`c`	Single character
`s`	String (all characters until '\0')
`d`	Signed decimal integer
`o`	Unsigned octal integer
`u`	Unsigned decimal integer
`x`	Unsigned hexadecimal integer with lowercase digits (`1a5e`)
`X`	As `x`, but with uppercase digits (e.g. `1A5E`)
`f`	Signed decimal value (floating point)
`e`	Signed decimal with exponent (e.g. `1.26e-5`)
`E`	As `e`, but uses `E` for exponent (e.g. `1.26E-5`)
`g`	As `e` or `f`, but depends on size and precision of value
`G`	As `g`, but uses `E` for exponent

printf()

Gotchas

- The value displayed is interpreted entirely by the formatting string:
  `printf("ASCII = %d", 'a');`
  will output: `ASCII = 97`
  A more problematic string:
  `printf("Value = %d", 6.02e23);`
  will output: `Value = 26366`
- Incorrect results may be displayed if the format type doesn't match the actual data type of the argument

printf()

Useful Format String Examples for Debugging

- Print a 16-bit hexadecimal value with a "0x" prefix and leading zeros if necessary to fill a 4 hex digit value:

```
printf("Address of x = %#06x\n", x_ptr);
```

Specifies that a 0x or 0X should precede a hexadecimal value (has other meanings for different conversion characters)

06 Specifies that 6 characters must be output (including 0x prefix), zeros will be filled in at left if necessary

x Specifies that the output value should be expressed as a hexadecimal integer

printf()

Useful Format String Examples for Debugging

- **Same as previous, but force hex letters to uppercase while leaving the 'x' in '0x' lowercase:**

```
printf("Address of x = 0x%04X\n", x_ptr);
```

04 Specifies that 4 characters must be output (no longer including 0x prefix since that is explicitly included in the string), zeros will be filled in at left if necessary

X Specifies that the output value should be expressed as a hexadecimal integer with uppercase A-F

Lab 03

`printf()` *Library Function*

On the CD

…\101_ECP\Lab03\Lab03.mcw

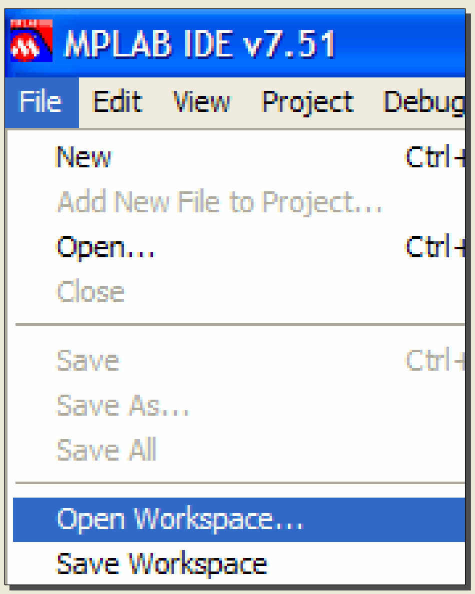
Lab 03

`printf()` Library Function

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab03\Lab03.mcw

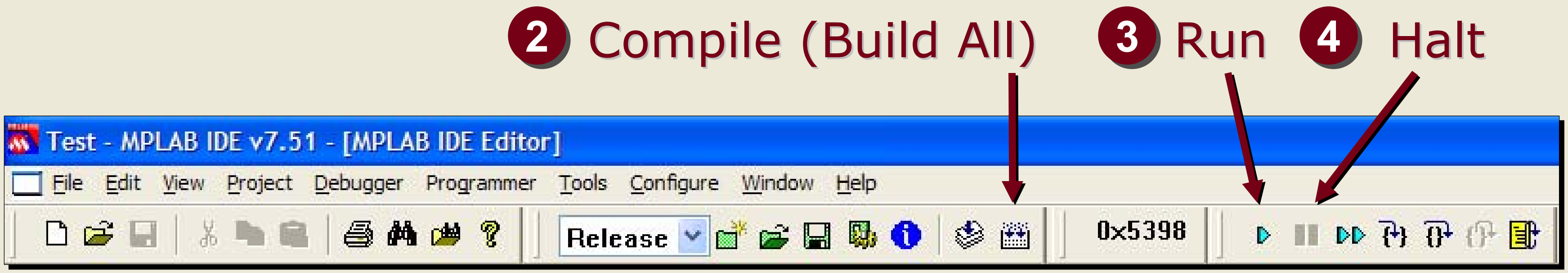


1. **Open MPLAB® IDE and select Open Workspace... from the File menu. Open the file listed above.**

⚠ **If you already have a project open in MPLAB IDE, close it by selecting Close Workspace from the File menu before opening a new one.**

Lab 03
printf() Library Function

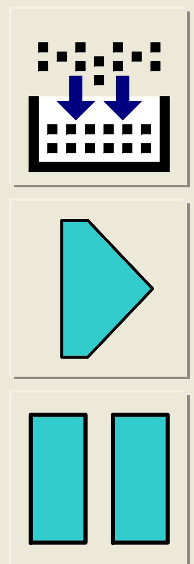
- **Compile and run the code:**



2 **Click on the Build All button.**

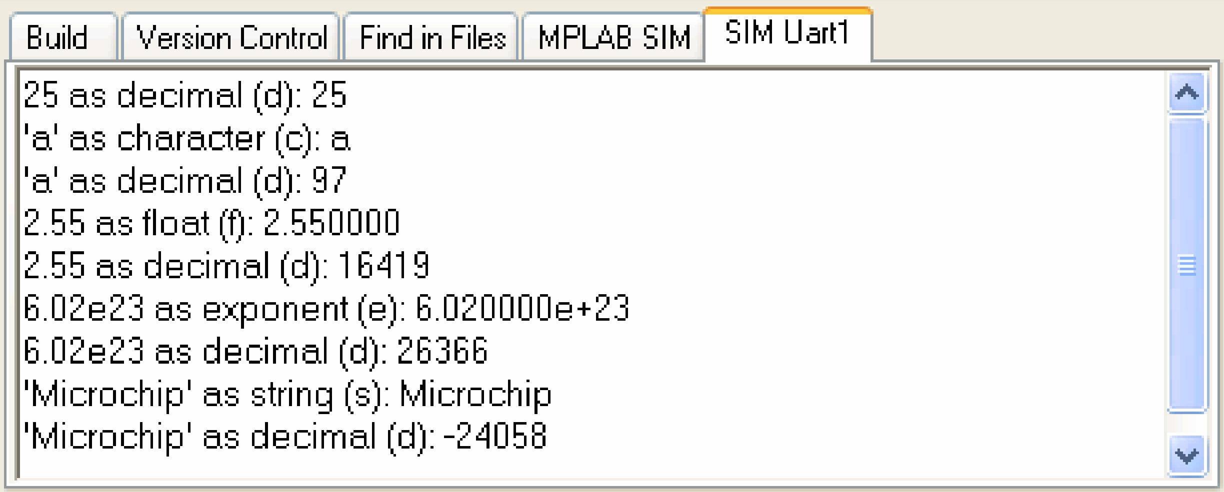
3 **If no errors are reported, click on the Run button.**

4 **Click on the Halt button.**



Lab 03
`printf()` Library Function

- **Expected Results (1):**



5 **The SIM Uart1 window should show the text that is output by the program by `printf()`, showing the how values are printed based on the formatting character used in the control string.**

Lab 03

`printf()` Library Function

- **Expected Results (2):**

Detailed Analysis:

Line of Code From Demo Project	Output
`printf("25 as decimal (d): %d\n", 25);`	25
`printf("'a' as character (c): %c\n", 'a');`	a
`printf("'a' as decimal (d): %d\n", 'a');`	97
`printf("2.55 as float (f): %f\n", 2.55);`	2.550000
`printf("2.55 as decimal (d): %d\n", 2.55);`	16419
`printf("6.02e23 as exponent (e): %e\n", 6.02e23);`	6.020000e+23
`printf("6.02e23 as decimal (d): %d\n", 6.02e23);`	26366
`printf("'Microchip' as string (s): %s\n", "Microchip");`	Microchip
`printf("'Microchip' as decimal (d): %d\n", "Microchip");`	-24058

Lab 03
Conclusions

- printf() has limited use in embedded applications themselves
- It is very useful as a debugging tool
- It can display data almost any way you want
- Projects that use printf() must:
  - Configure a heap (done in MPLAB® IDE)
  - Include the stdio.h header file

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.6
Operators

Operators

How to Code Arithmetic Expressions

Definition

An arithmetic expression is an expression that contains one or more operands and arithmetic operators.

- Operands may be variables, constants or functions that return a value
  - A microcontroller register is usually treated as a variable
- There are 9 arithmetic operators that may be used
  - Binary Operators: `+`, `-`, `*`, `/`, `%`
  - Unary Operators: `+`, `-`, `++`, `--`

Operators
Arithmetic

Operator	Operation	Example	Result
`*`	Multiplication	`x * y`	Product of `x` and `y`
`/`	Division	`x / y`	Quotient of `x` and `y`
`%`	Modulo	`x % y`	Remainder of `x` divided by `y`
`+`	Addition	`x + y`	Sum of `x` and `y`
`-`	Subtraction	`x - y`	Difference of `x` and `y`
+ (unary)	Positive	`+x`	Value of `x`
− (unary)	Negative	`-x`	Negative value of `x`

NOTE - An `int` divided by an `int` returns an `int`:
10/3 = 3
Use modulo to get the remainder:
10%3 = 1

Operators

Division Operator

- If both operands are an integer type, the result will be an integer type (int, char)
- If one or both of the operands is a floating point type, the result will be a floating point type (float, double)

Example: Integer Divide

```
int a = 10;
int b = 4;
float c;
c = a / b;
```

c = 2.000000 ✗

Because: int / int ➡ int

Example: Floating Point Divide

```
int a = 10;
float b = 4.0f;
float c;
c = a / b;
```

c = 2.500000 ✓

Because: float / int ➡ float

Operators

Implicit Type Conversion

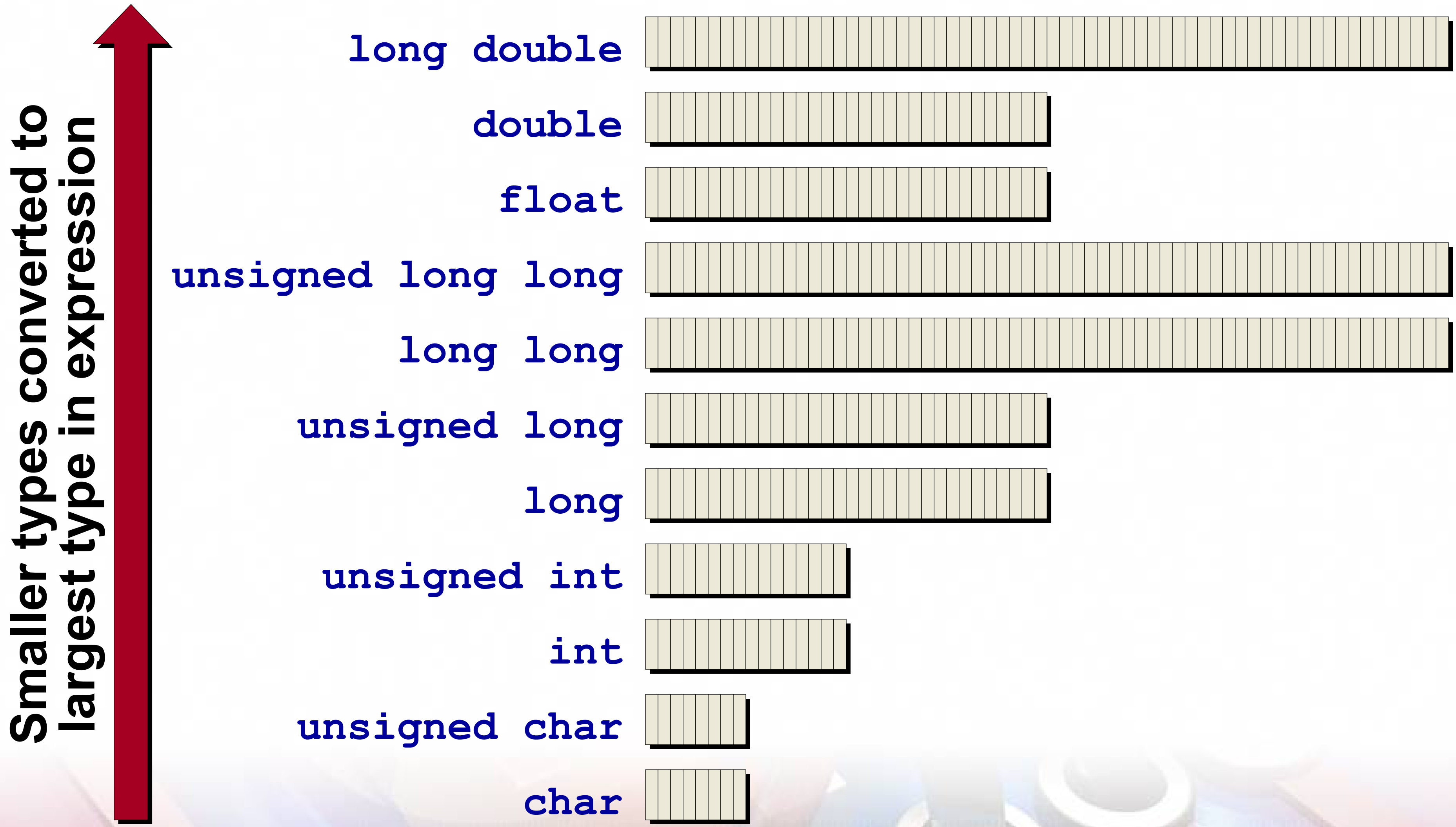
- In many expressions, the type of one operand will be temporarily "promoted" to the larger type of the other operand

Example

```
int x = 10;
float y = 2.0, z;
z = x * y;              // x promoted to float
```

- A smaller data type will be promoted to the largest type in the expression for the duration of the operation

Implicit Arithmetic Type Conversion Hierarchy



Operators

Arithmetic Expression Implicit Type Conversion

- Example implicit type conversions

Assume x is defined as:

```
short x = -5;
```

Expression	Implicit Type Conversion	Expression's Type	Result
`-x`	`x` is promoted to `int`	`int`	5
`x * -2L`	`x` is promoted to `long` because `-2L` is a `long`	`long`	10
`8/x`	`x` is promoted to `int`	`int`	-1
`8%x`	`x` is promoted to `int`	`int`	3
`8.0/x`	`x` is promoted to `double` because `8.0` is a `double`	`double`	-1.6

Operators

Applications of the Modulus Operator (%)

- **Truncation: `x % 2`^{n} where `n` is the desired word width (e.g. 8 for 8 bits: `x % 256`)**
  - **Returns the value of just the lower `n`-bits of `x`**
- **Can be used to break apart a number in any base into its individual digits**

Example

```
#define MAX_DIGITS 6
long number = 123456;
int i, radix = 10; char digits[MAX_DIGITS];

for (i = 0; i < MAX_DIGITS; i++)
{
   if (number == 0) break;
   digits[i] = (char)(number % radix);
   number /= radix;
}
```

Operators

Arithmetic: Increment and Decrement

Operator	Operation	Example	Result
`++`	Increment	`x++`	Use **x** then increment **x** by 1
		`++x`	Increment **x** by 1, then use **x**
`--`	Decrement	`x--`	Use **x** then decrement **x** by 1
		`--x`	Decrement **x** by 1, then use **x**

Postfix Example

```
x = 5;
y = (x++) + 5;
// y = 10
// x = 6
```

Prefix Example

```
x = 5;
y = (++x) + 5;
// y = 11
// x = 6
```

Operators

How to Code Assignment Statements

Definition

An <u>assignment statement</u> is a statement that assigns a value to a variable.

- Two types of assignment statements
  - Simple assignment
    `variable = expression;`
    The expression is evaluated and the result is assigned to the variable
  - Compound assignment
    `variable = variable op expression;`
    The variable appears on both sides of the =

Operators

Assignment

Operator	Operation	Example	Result
`=`	Assignment	`x = y`	Assign `x` the value of `y`
`+=`	Compound Assignment	`x += y`	`x = x + y`
`-=`		`x -= y`	`x = x - y`
`*=`		`x *= y`	`x = x * y`
`/=`		`x /= y`	`x = x / y`
`%=`		`x %= y`	`x = x % y`
`&=`		`x &= y`	`x = x & y`
`^=`		`x ^= y`	`x = x ^ y`
`|=`		`x |= y`	`x = x | y`
`<<=`		`x <<= y`	`x = x << y`
`>>=`		`x >>= y`	`x = x >> y`

Operators

Compound Assignment

- **Statements with the same variable on each side of the equals sign:**

Example

```
x = x + y;
```

This operation may be thought of as: The new value of **x** will be set equal to the current value of **x** plus the value of **y**

- **May use the shortcut assignment operators (compound assignment):**

Example

```
x += y;      //Increment x by the value y
```

Operators

Compound Assignment

Example

```
int x = 2;    //Initial value of x is 2

x *= 5;       //x = x * 5
```

Before statement is executed: `x` = 2

After statement is executed: `x` = 10

```
                          x *= 5;
     Is equivalent to:    x = (x * 5);
Evaluate right side first: x = (2 * 5);
     Assign result to x:  x = 10;
```

Operators

Relational

Operator	Operation	Example	Result (FALSE = 0, TRUE ≠ 0)
`<`	Less than	`x < y`	1 if **x** less than **y**, else 0
`<=`	Less than or equal to	`x <= y`	1 if **x** less than or equal to **y**, else 0
`>`	Greater than	`x > y`	1 if **x** greater than **y**, else 0
`>=`	Greater than or equal to	`x >= y`	1 if **x** greater than or equal to **y**, else 0
`==`	Equal to	`x == y`	1 if **x** equal to **y**, else 0
`!=`	Not equal to	`x != y`	1 if **x** not equal to **y**, else 0

In conditional expressions, any non-zero value is interpreted as TRUE. A value of 0 is always FALSE.

Operators

Difference Between = and ==

> **Be careful not to confuse = and ==. They are not interchangeable!**

- **= is the assignment operator**
  `x = 5` assigns the value 5 to the variable `x`
- **== is the 'equals to' relational operator**
  `x == 5` tests whether the value of `x` is 5

```
if (x == 5)
{
    do if value of x is 5
}
```

Operators

Difference Between = and ==

- What happens when the following code is executed?

Example

```
void main(void)
{
  int x = 2;                    //Initialize x
  if (x = 5)                    //If x is 5,…
  {
    printf("Hi!");              //…display "Hi!"
  }
}
```

Operators
Logical

Operator	Operation	Example	Result (FALSE = 0, TRUE ≠ 0)
`&&`	Logical AND	`x && y`	1 if both **x** ≠ 0 and **y** ≠ 0, else 0
`||`	Logical OR	`x || y`	0 if both **x** = 0 and **y** = 0, else 1
`!`	Logical NOT	`!x`	1 if **x** = 0, else 0

In conditional expressions, any non-zero value is interpreted as TRUE. A value of 0 is always FALSE.

Operators

Bitwise

Operator	Operation	Example	Result (for each bit position)
`&`	Bitwise AND	`x & y`	1, if 1 in both **x** and **y** 0, if 0 in **x** or **y** or both
`|`	Bitwise OR	`x | y`	1, if 1 in **x** or **y** or both 0, if 0 in both **x** and **y**
`^`	Bitwise XOR	`x ^ y`	1, if 1 in **x** or **y** but not both 0, if 0 or 1 in both **x** and **y**
`~`	Bitwise NOT (One's Complement)	`~x`	1, if 0 in **x** 0, if 1 in **x**

- **The operation is carried out on each bit of the first operand with each corresponding bit of the second operand**

Operators

Difference Between & and &&

> ⚠ **Be careful not to confuse & and &&. They are not interchangeable!**

- **& is the bitwise AND operator**

  `0b1010 & 0b1101 → 0b1000`

- **&& is the logical AND operator**

  `0b1010 && 0b1101 → 0b0001` **(TRUE)**

  <*Non-Zero Value*> && <*Non-Zero Value*> → 1 (TRUE)

```
if (x && y)
{
    do if x and y are both TRUE (non-zero)
}
```

Operators

Difference Between & and &&

- What happens when each of these code fragments are executed?

Example 1 – Using A *Bitwise* AND Operator

```
char x = 0b1010;
char y = 0b0101;
if (x & y) printf("Hi!");
```

Example 2 – Using A *Logical* AND Operator

```
char x = 0b1010;
char y = 0b0101;
if (x && y) printf("Hi!");
```

Operators

Logical Operators and Short Circuit Evaluation

- **The evaluation of expressions in a logical operation stops as soon as a TRUE or FALSE result is known**

Example

If we have two expressions being tested in a logical AND operation:

```
expr1 && expr2
```

The expressions are evaluated from left to right. If *expr1* is 0 (FALSE), then *expr2* would not be evaluated at all since the overall result is already known to be false.

Truth Table for AND (&&)
FALSE = 0
TRUE = 1

expr1	*expr2*	Result
0	X (0)	0
0	X (1)	0
1	0	0
1	1	1

expr2 is not evaluated in the first two cases since its value is not relevant to the result.

Operators

Logical Operators and Short Circuit Evaluation

- **The danger of short circuit evaluation**

Example

If z = 0, then c will not be evaluated

```
if !((z = x + y) && (c = a + b))
{
    z += 5;
    c += 10;
}
```

← Initial value of c may not be correct

It is perfectly legal in C to logically compare two assignment expressions in this way, though it is not usually good programming practice.
A similar problem exists when using function calls in logical operations, which is a very common practice. The second function may never be evaluated.

Operators

Shift

Operator	Operation	Example	Result
<<	Shift Left	`x << y`	Shift **x** by **y** bits to the left
>>	Shift Right	`x >> y`	Shift **x** by **y** bits to the right

Shift Left Example:

```
x = 5;          // x = 0b00000101 = 5
y = x << 2;     // y = 0b00010100 = 20
```

- In both shift left and shift right, the bits that are shifted out are lost
- For shift left, 0's are shifted in (Zero Fill)

Operators

Shift – Special Cases

- **Logical Shift Right (Zero Fill)**

  If x is UNSIGNED (`unsigned char` in this case):

```
x = 250;        // x = 0b11111010 = 250
y = x >> 2;     // y = 0b00111110 = 62
```

- **Arithmetic Shift Right (Sign Extend)**

  If x is SIGNED (`char` in this case):

```
x = -6;         // x = 0b11111010 = -6
y = x >> 2;     // y = 0b11111110 = -2
```

Operators

Power of 2 Integer Divide vs. Shift Right

- If you are dividing by a power of 2, it will usually be more efficient to use a right shift instead

$$y = x / 2^n \quad \rightarrow \quad y = x >> n$$

0	0	0	0	1	0	1	0	>>	0	0	0	0	0	1	0	1

10_{10} Right Shift 5_{10}

- Works for integers or fixed point values

Operators

Power of 2 Integer Divide vs. Shift in MPLAB® C30

Example: Divide by 2

```
int x = 20;
int y;
y = x / 2;
```

y = 10

```
10:                    y = x / 2;
00288   804000         mov.w 0x0800,0x0000
0028A   200022         mov.w #0x2,0x0004
0028C   090011         repeat #17
0028E   D80002         div.sw 0x0000,0x0004
00290   884010         mov.w 0x0000,0x0802
```

Example: Right Shift by 1

```
int x = 20;
int y;
y = x >> 1;
```

y = 10

```
9:                     y = x >> 1;
00282   804000         mov.w 0x0800,0x0000
00284   DE8042         asr 0x0000,#1,0x0000
00286   884010         mov.w 0x0000,0x0802
```

Operators

Power of 2 Integer Divide vs. Shift in MPLAB® C18

Example: Divide by 2

```
int x = 20;
int y;
y = x / 2;
```

y = 10

```
10:                 y = x / 2;
0132    C08C        MOVFF 0x8c, 0x8a
0134    F08A        NOP
0136    C08D        MOVFF 0x8d, 0x8b
0138    F08B        NOP
013A    0E02        MOVLW 0x2
013C    6E0D        MOVWF 0xd, ACCESS
013E    6A0E        CLRF 0xe, ACCESS
0140    C08A        MOVFF 0x8a, 0x8
0142    F008        NOP
0144    C08B        MOVFF 0x8b, 0x9
0146    F009        NOP
0148    EC6B        CALL 0xd6, 0
014A    F000        NOP
014C    C008        MOVFF 0x8, 0x8a
014E    F08A        NOP
0150    C009        MOVFF 0x9, 0x8b
0152    F08B        NOP
```

Example: Right Shift by 1

```
int x = 20;
int y;
y = x >> 1;
```

y = 10

```
9:                  y = x >> 1;
0122    C08C        MOVFF 0x8c, 0x8a
0124    F08A        NOP
0126    C08D        MOVFF 0x8d, 0x8b
0128    F08B        NOP
012A    0100        MOVLB 0
012C    90D8        BCF 0xfd8, 0, ACCESS
012E    338B        RRCF 0x8b, F, BANKED
0130    338A        RRCF 0x8a, F, BANKED
```

16-Bit Shift on 8-Bit Architecture

Operators

Memory Addressing

Operator	Operation	Example	Result
`&`	Address of	`&x`	Pointer to `x`
`*`	Indirection	`*p`	The object or function that `p` points to
`[]`	Subscripting	`x[y]`	The `y`th element of array `x`
`.`	Struct / Union Member	`x.y`	The member named `y` in the structure or union `x`
`->`	Struct / Union Member by Reference	`p->y`	The member named `y` in the structure or union that `p` points to

These operators will be discussed later in the sections on arrays, pointers, structures, and unions. They are included here for reference and completeness.

Operators
Other

Operator	Operation	Example	Result
`()`	Function Call	`foo(x)`	Passes control to the function with the specified arguments
`sizeof`	Size of an object or type in bytes	`sizeof x`	The number of bytes **`x`** occupies in memory
`(type)`	Explicit type cast	`(short) x`	Converts the value of **`x`** to the specified type
`?:`	Conditional expression	`x ? y : z`	The value of **`y`** if **`x`** is true, else value of **`z`**
`,`	Sequential evaluation	`x, y`	Evaluates **`x`** then **`y`**, else result is value of **`y`**

Operators

The Conditional Operator

Syntax

```
(test-expr) ? do-if-true : do-if-false;
```

Example

```c
int x = 5;

(x % 2 != 0) ?
      printf("%d is odd\n", x) :
      printf("%d is even\n", x);
```

Result:

```
5 is odd
```

Operators

The Conditional Operator

```
x = (condition) ? A : B;
```

x = A if condition is true
x = B if condition is false

Operators

The Explicit Type Cast Operator

- **Earlier, we cast a literal to type float by entering it as: `4.0f`**
- **We can cast the variable instead by using the cast operator: `(type)variable`**

Example: Integer Divide

```
int x = 10;
float y;

y = x / 4;
```

y = 2.000000 ✗

Because: int / int ➔ int

Example: Floating Point Divide

```
int x = 10;
float y;

y = (float)x / 4;
```

y = 2.500000 ✓

Because: float / int ➔ float

Operators

Precedence

Operator	Description	Associativity
`( )`	Parenthesized Expression	Left-to-Right
`[ ]`	Array Subscript	
`.`	Structure Member	
`->`	Structure Pointer	
`+ -`	Unary + and – (Positive and Negative Signs)	Right-to-Left
`++ --`	Increment and Decrement	
`! ~`	Logical NOT and Bitwise Complement	
`*`	Dereference (Pointer)	
`&`	Address of	
`sizeof`	Size of Expression or Type	
`(type)`	Explicit Typecast	

Continued on next slide…

Operators

Precedence

Operator	Description	Associativity
* / %	Multiply, Divide, and Modulus	Left-to-Right
+ -	Add and Subtract	Left-to-Right
<< >>	Shift Left and Shift Right	Left-to-Right
< <=	Less Than and Less Than or Equal To	Left-to-Right
> >=	Greater Than and Greater Than or Equal To	Left-to-Right
== !=	Equal To and Not Equal To	Left-to-Right
&	Bitwise AND	Left-to-Right
^	Bitwise XOR	Left-to-Right
|	Bitwise OR	Left-to-Right
&&	Logical AND	Left-to-Right
||	Logical OR	Left-to-Right
?:	Conditional Operator	Right-to-Left

Continued on next slide…

Operators

Precedence

Operator	Description	Associativity
`=`	Assignment	Right-to-Left
`+=` `-=`	Addition and Subtraction Assignments	
`/=` `*=`	Division and Multiplication Assignments	
`%=`	Modulus Assignment	
`<<=` `>>=`	Shift Left and Shift Right Assignments	
`&=` `|=`	Bitwise AND and OR Assignments	
`^=`	Bitwise XOR Assignment	
`,`	Comma Operator	Left-to-Right

- **Operators grouped together in a section have the same precedence – conflicts within a section are handled via the rules of associativity**

Operators
Precedence

- **When expressions contain multiple operators, their precedence determines the order of evaluation**

Expression	Effective Expression
`a - b * c`	`a - (b * c)`
`a + ++b`	`a + (++b)`
`a + ++b * c`	`a + ((++b) * c)`

If functions are used in an expression, there is no set order of evaluation for the functions themselves.

e.g. `x = f() + g()`
There is no way to know if `f()` or `g()` will be evaluated first.

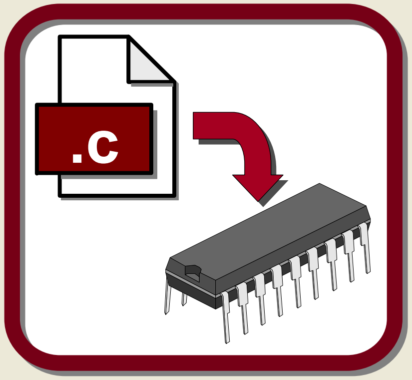
Operators

Associativity

- If two operators have the same precedence, their associativity determines the order of evaluation

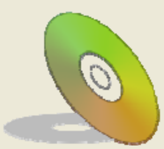
Expression	Associativity	Effective Expression
`x / y % z`	Left-to-Right	`(x / y) % z`
`x = y = z`	Right-to-Left	`x = (y = z)`
`~++x`	Right-to-Left	`~(++x)`

- You can rely on these rules, but it is good programming practice to explicitly group elements of an expression



Lab 04

Operators



On the CD

…\101_ECP\Lab04\Lab04.mcw

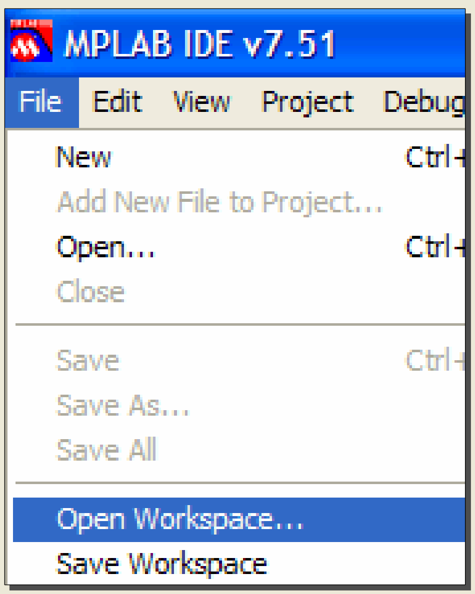
Lab 04

Operators

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab04\Lab04.mcw



1. **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 04

Operators

Solution: Steps 1 and 2

```
/*############################################################################
# STEP 1:  Add charVariable1 to charVariable2 and store the result in
#          charVariable1.  This may be done in two ways.  One uses the
#          ordinary addition operator, the other uses a compound assignment
#          operator.  Write two lines of code to perform this operation
#          twice - once for each of the two methods.
#          Don't forget to end each statement with a semi-colon!
############################################################################*/

//Add using addition operator
charVariable1 = charVariable1 + charVariable2;
//Add using compound assignment operator
charVariable1 += charVariable2;

/*############################################################################
# STEP 2:  Increment charVariable1.  There are several ways this could be
#          done.  Use the one that requires the least amount of typing.
############################################################################*/

//Increment charVariable1
charVariable1++;
```

Lab 04

Operators

Solution: Steps 3 and 4

```
/*###############################################################################
# STEP 3: Use the conditional operator to set longVariable1 equal to
#         intVariable1 if charVariable1 is less than charVariable2.
#         Otherwise, set longVariable1 equal to intVariable2
# NOTE:   The comments below are broken up into 3 lines, but the code you
#         need to write can fit on a single line.
###############################################################################*/

//If charVariable1 < charVariable2, then
//longVariable1 = intVariable1, otherwise
//longVariable1 = intVariable2
longVariable1 = (charVariable1 < charVariable2) ? intVariable1 : intVariable2;

/*###############################################################################
# STEP 4: Shift longVariable2 one bit to the right.  This can be accomplished
#         most easily using the appropriate compound assignment operator.
###############################################################################*/

//Shift longVariable2 one bit to the right
longVariable2 >>= 1;
```

Lab 04
Operators

Solution: Step 5

```
/*###############################################################################
# STEP 5: Perform the operation (longVariable2 AND 0x30) and store the result
#         back in longVariable2.  Once again, the easiest way to do this is
#         to use the appropriate compound assignment operator that will
#         perform an equivalent operation to the one in the comment below.
###############################################################################*/

//longVariable2 = longVariable2 & 0x30
longVariable2 &= 0x30;
```

Lab 04

Conclusions

- Most operators look just like their normal mathematical notation
- C adds several shortcut operators in the form of compound assignments
- Most C programmers tend to use the shortcut operators

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.7

Expressions and Statements

Expressions

- Represents a single data item (e.g. character, number, etc.)
- May consist of:
  - A single entity (a constant, variable, etc.)
  - A combination of entities connected by operators (+, -, *, / and so on)

Expressions

Examples

Example

```
a + b
x = y
speed = dist/time
z = ReadInput()
c <= 7
x == 25
count++
d = a + 5
```

Statements

- Cause an action to be carried out
- Three kinds of statements in C:
  - Expression Statements
  - Compound Statements
  - Control Statements

Expression Statements

- An expression followed by a semi-colon
- Execution of the statement causes the expression to be evaluated

Examples

```
i = 0;
i++;
a = 5 + i;
y = (m * x) + b;
printf("Slope = %f", m);
;
```

Compound Statements

- A group of individual statements enclosed within a pair of curly braces { and }
- Individual statements within may be any statement type, including compound
- Allows statements to be embedded within other statements
- Does NOT end with a semicolon after }
- Also called Block Statements

Compound Statements

Example

Example

```
{
   float start, finish;

   start = 0.0;
   finish = 400.0;
   distance = finish – start;
   time = 55.2;
   speed = distance / time;
   printf("Speed = %f m/s", speed);
}
```

Control Statements

- Used for loops, branches and logical tests
- Often require other statements embedded within them

Example

```
while (distance < 400.0)
{
   printf("Keep running!");
   distance += 0.1;
}
```

(while syntax: `while` *`expr statement`*)

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

<u>Section 1.8</u>
Making Decisions

Boolean Expressions

- C has no Boolean data type
- Boolean expressions return integers:
  - 0 if expression evaluates as FALSE
  - non-zero if expression evaluates as TRUE (usually returns 1, but this is not guaranteed)

```
int main(void)
{
    int x = 5, y, z;

    y = (x > 4);      y = 1  (TRUE)
    z = (x > 6);
    while (1);        z = 0  (FALSE)
}
```

Boolean Expressions

Equivalent Expressions

- If a variable, constant or function call is used alone as the conditional expression: `(MyVar)` or `(Foo())`
- This is the same as saying: `(MyVar != 0)` or `(Foo() != 0)`
- In either case, if `MyVar` ≠ 0 or `Foo()` ≠ 0, then the expression evaluates as TRUE (non-zero)
- C Programmers almost always use the first method (laziness always wins in C)

`if` Statement

Syntax

```
if (expression) statement
```

- *`expression`* is evaluated for boolean TRUE (≠0) or FALSE (=0)
- If TRUE, then *`statement`* is executed

Note

Whenever you see ***`statement`*** in a syntax guide, it may be replaced by a compound (block) statement.

Remember: spaces and new lines are not significant.

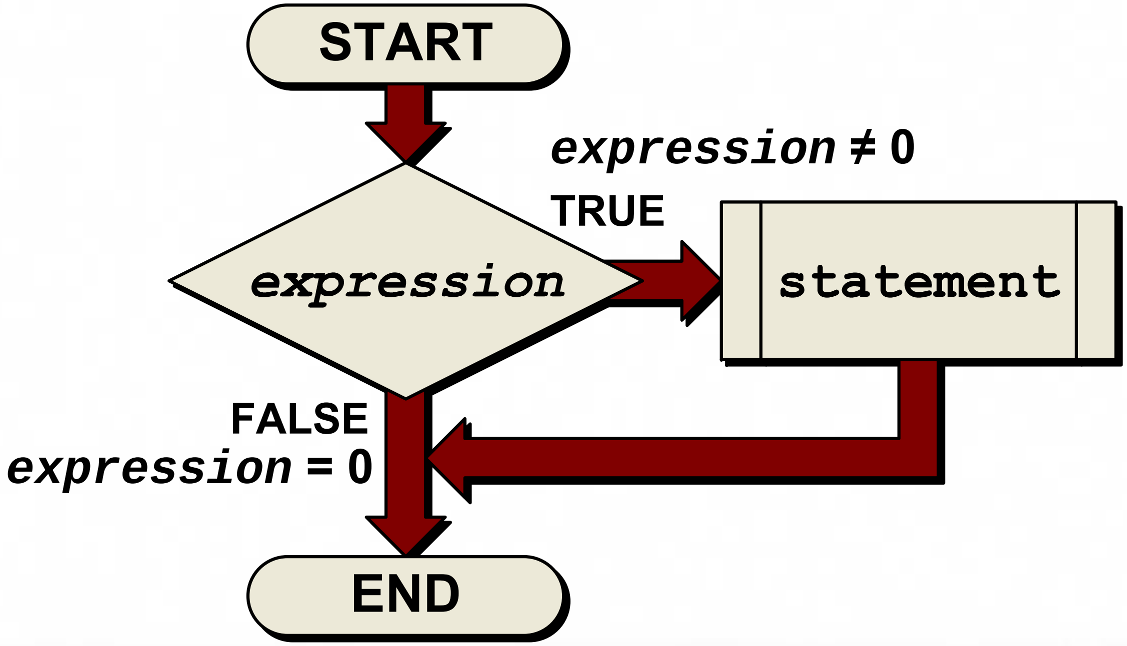
```
if (expression)
{
    statement1
    statement2
}
```

`if` Statement

Flow Diagram

Syntax

```
if (expression) statement
```



if Statement

Example

```
{
  int x = 5;

  if (x)                          If x is TRUE (non-zero)…
  {
     printf("x = %d\n", x);       …then print the value of x.
  }
  while (1);
}
```

- What will print if x = 5?   … if x = 0?
- …if x = -82?
- …if x = 65536? ?

`if` Statement

Testing for TRUE

- `if (x)` vs. `if (x == 1)`
  - `if (x)` only needs to test for not equal to 0
  - `if (x == 1)` needs to test for equality with 1
  - Remember: TRUE is defined as non-zero, FALSE is defined as zero

Example: if (x)

```
if (x)

8:                     if (x)
011B4   E208C2         cp0.w 0x08c2
011B6   320004         bra z, 0x0011c0
```

Example: if (x ==1)

```
if (x == 1)

11:                    if (x == 1)
011C0   804610         mov.w 0x08c2,0x0000
011C2   500FE1         sub.w 0x0000,#1,[0x001e]
011C4   3A0004         bra nz, 0x0011ce
```

Nested `if` Statements

Example

```
int power = 10;
float band = 2.0;
float frequency = 146.52;

if (power > 5)
{
   if (band == 2.0)
   {
      if ((frequency > 144) && (frequency < 148))
      {
        printf("Yes, it's all true!\n");
      }
   }
}
```

`if-else` Statement

Syntax

```
if (expression) statement1
else statement2
```

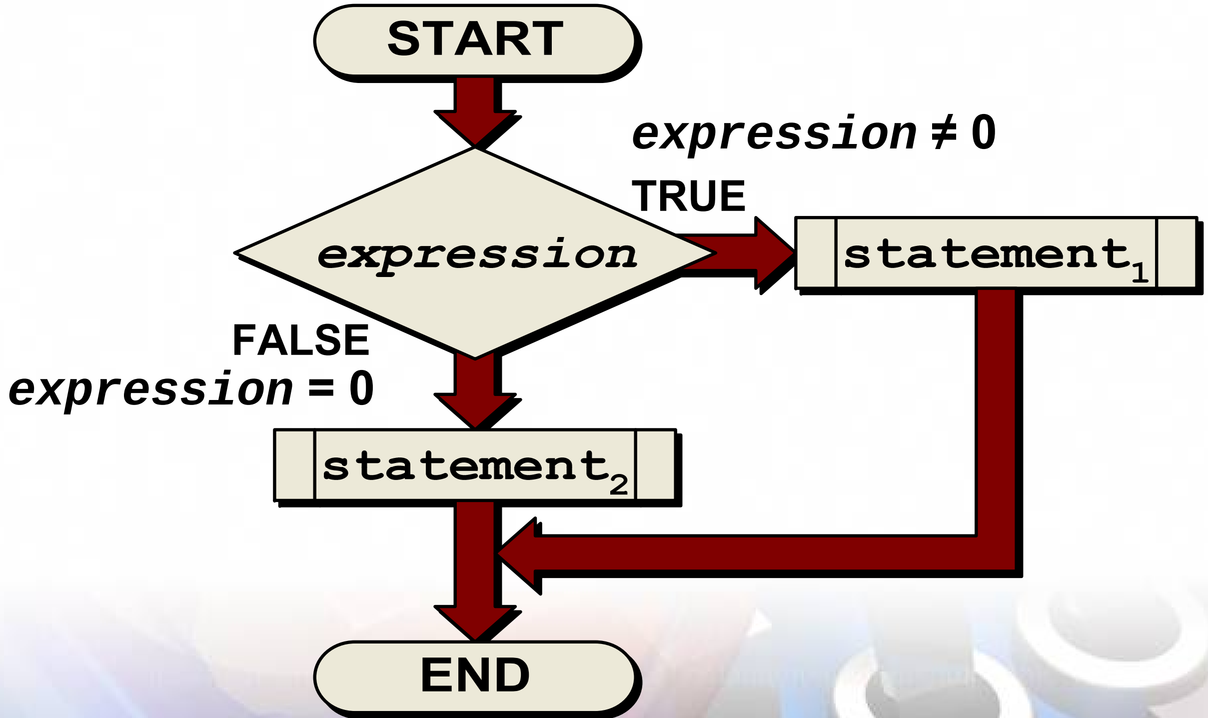
- *`expression`* is evaluated for boolean TRUE (≠0) or FALSE (=0)
- If TRUE, then *`statement`*$_1$ is executed
- If FALSE, then *`statement`*$_2$ is executed

`if-else` Statement

Flow Diagram

Syntax

```
if (expression) statement1
else statement2
```



if-else Statement

Example

```
{
  float frequency = 146.52;  //frequency in MHz

  if ((frequency > 144.0) && (frequency < 148.0))
    {
      printf("You're on the 2 meter band\n");
    }
  else
    {
      printf("You're not on the 2 meter band\n");
    }
}
```

`if-else if` Statement

Syntax

```
if (expression₁) statement₁
else if (expression₂) statement₂
else statement₃
```

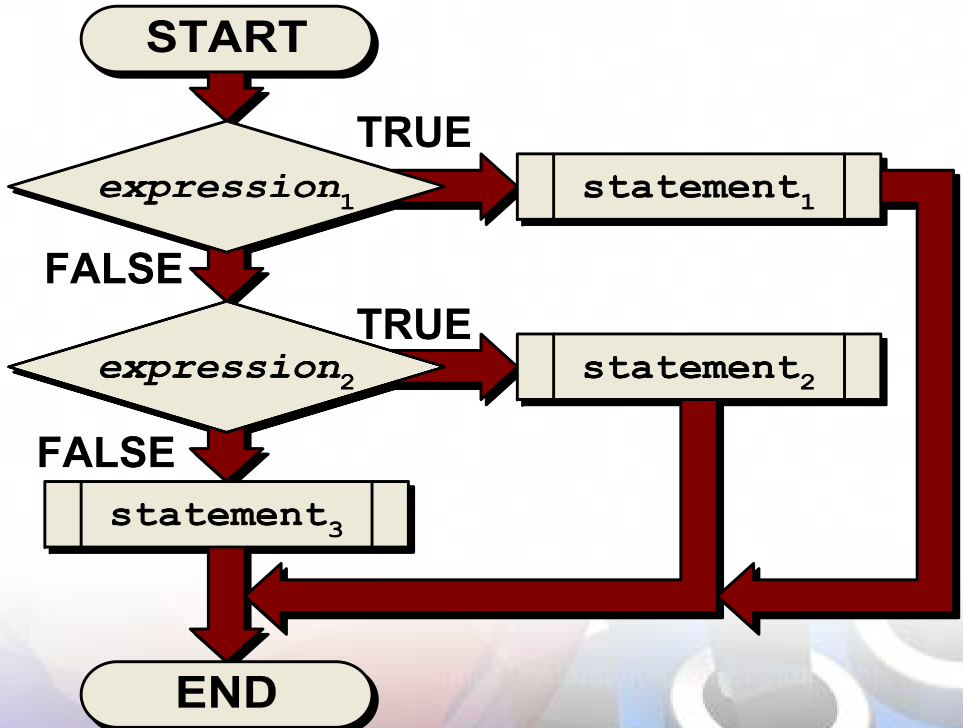
- *expression*$_1$ is evaluated for boolean TRUE (≠0) or FALSE (=0)
- If TRUE, then *statement*$_1$ is executed
- If FALSE, then *expression*$_2$ is evaluated
- If TRUE, then *statement*$_2$ is executed
- If FALSE, then *statement*$_3$ is executed

`if-else if` Statement

Flow Diagram

Syntax

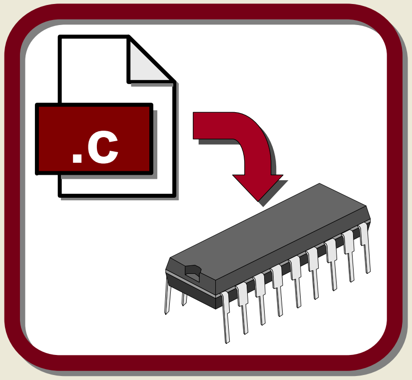
```
if (expression₁) statement₁
else if (expression₂) statement₂
else statement₃
```



if-else if Statement

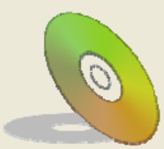
Example

```c
if ((freq > 144) && (freq < 148))
    printf("You're on the 2 meter band\n");
else if ((freq > 222) && (freq < 225))
    printf("You're on the 1.25 meter band\n");
else if ((freq > 420) && (freq < 450))
    printf("You're on the 70 centimeter band\n");
else
    printf("You're somewhere else\n");
```



Lab 05

Making Decisions (if)



On the CD

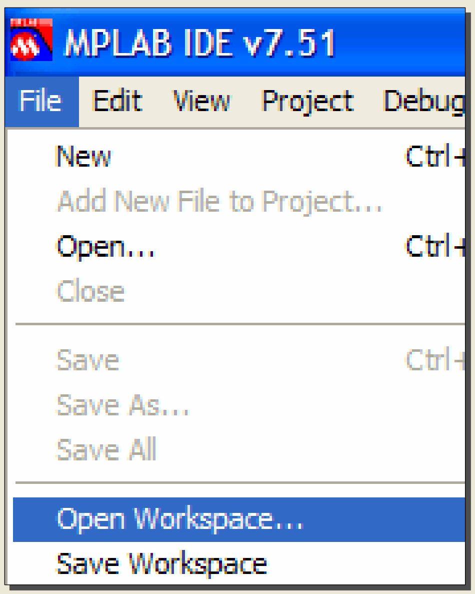
…\101_ECP\Lab05\Lab05.mcw

Lab 05
Making Decisions (`if`)

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab05\Lab05.mcw



1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 05

Making Decisions (`if`)

Solution: Steps 1 and 2

```
/*###############################################################################
# STEP 1: Increment intVariable1 if BOTH the following conditions are true:
#          * floatVariable2 is greater than or equal to floatVariable1
#          * charVariable2 is greater than or equal to charVariable1
#          Remember to use parentheses to group logical operations.
###############################################################################*/
//Write the if condition
if((floatVariable2 >= floatVariable1) && (charVariable2 >= charVariable1))
{
        intVariable1++;           //Increment intVariable1
}

/*###############################################################################
# STEP 2: If the above is not true, and floatVariable1 is greater than 50
#          then decrement intVariable2.  (HINT: else if)
###############################################################################*/
//Write the else if condition
else if(floatVariable1 > 50)
{
        intVariable2--;            //Decrement intVariable2
}
```

Lab 05

Making Decisions (`if`)

Solution: Step 3

```
/*###############################################################################
# STEP 3: If neither of the above are true, set charVariable2 equal to 1.
#         (HINT: else)
###############################################################################*/
//Write the else condition
else
{
        charVariable2 = 1;        //Set charVariable2 equal to 1

}
```

Lab 05
Conclusions

- **if statements make it possible to conditionally execute a line or block of code based on a logic equation**
- **else if / else statements make it possible to present follow-up conditions if the first one proves to be false**

`switch` Statement

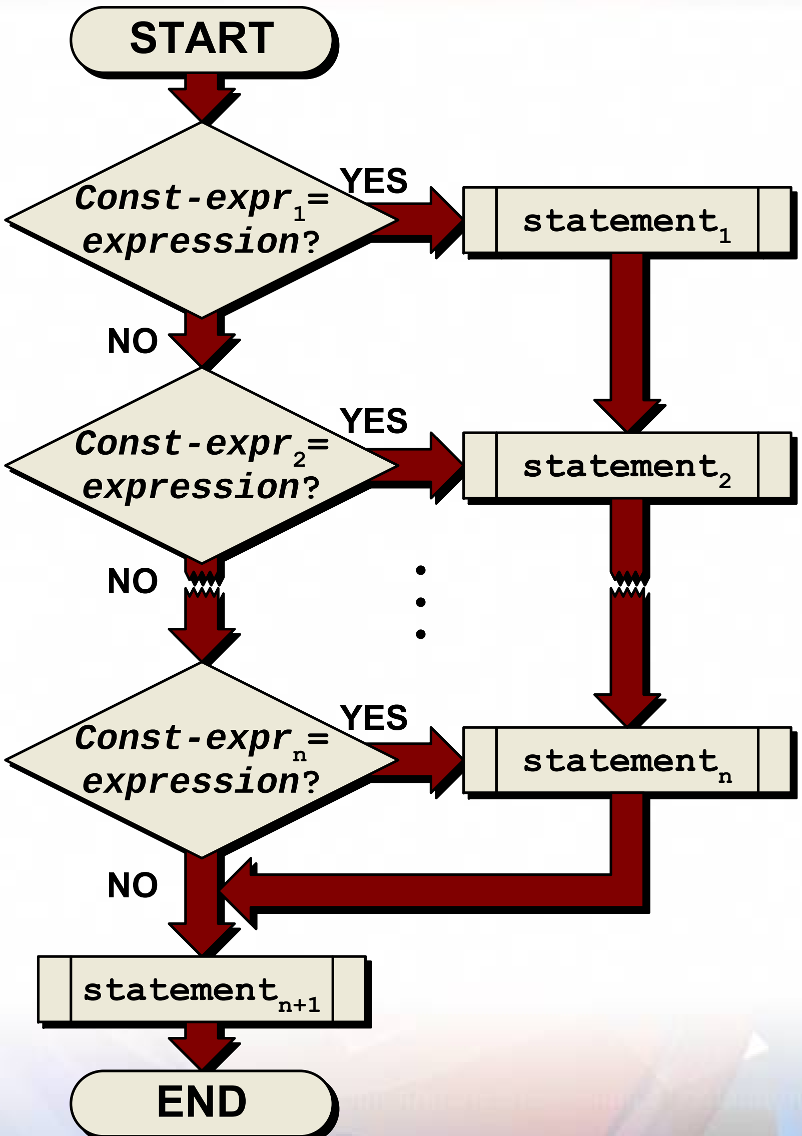
Syntax

```
switch (expression)
{
   case const-expr1: statements1
                 :
                 :
   case const-exprn: statementsn
   default: statementsn+1
}
```

- *`expression`* is evaluated and tested for a match with the *`const-expr`* in each `case` clause
- The *`statements`* in the matching `case` clause is executed

switch Statement

Flow Diagram (default)

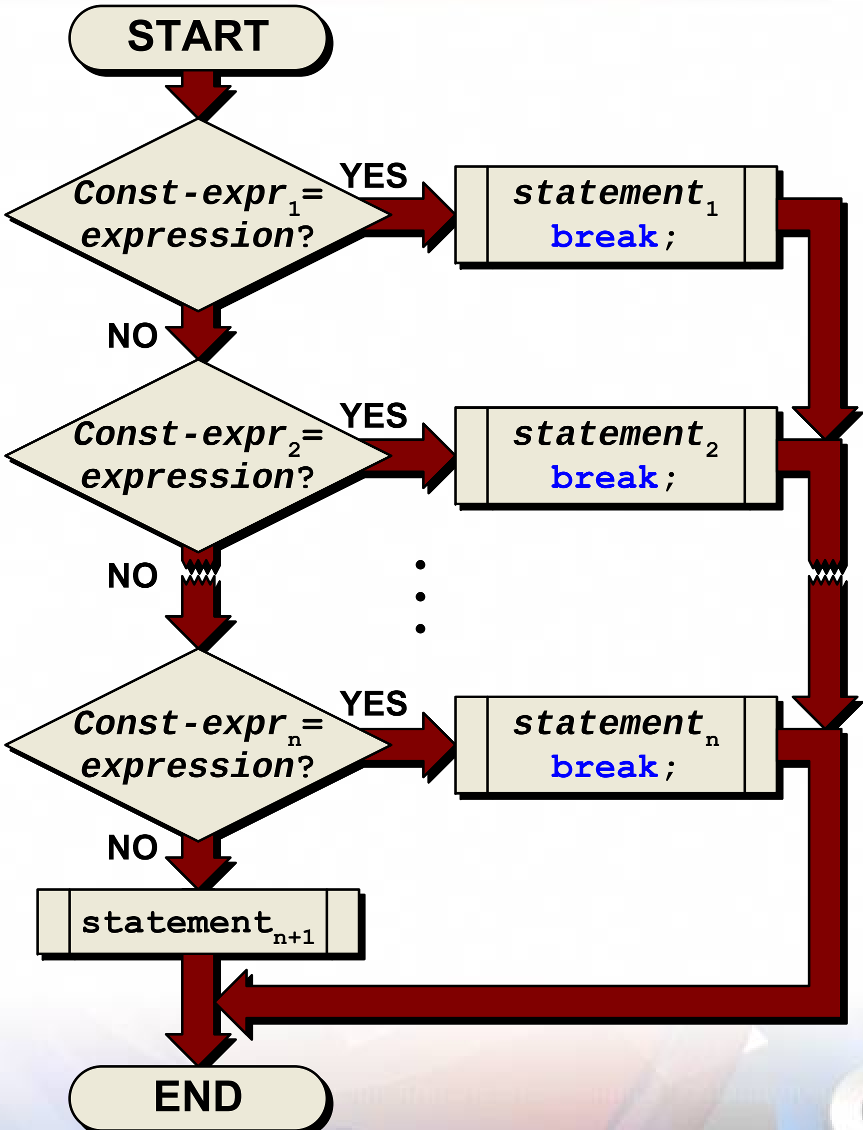


Notice that each statement falls through to the next

This is the default behavior of the `switch` statement

switch Statement

Flow Diagram (modified)



Adding a **break** statement to each statement block will eliminate fall through, allowing only one case clause's statement block to be executed

switch Statement

switch Example 1

```
switch(channel)
{
      case 2:  printf("WBBM Chicago\n"); break;
      case 3:  printf("DVD Player\n"); break;
      case 4:  printf("WTMJ Milwaukee\n"); break;
      case 5:  printf("WMAQ Chicago\n"); break;
      case 6:  printf("WITI Milwaukee\n"); break;
      case 7:  printf("WLS Chicago\n"); break;
      case 9:  printf("WGN Chicago\n"); break;
      case 10: printf("WMVS Milwaukee\n"); break;
      case 11: printf("WTTW Chicago\n"); break;
      case 12: printf("WISN Milwaukee\n"); break;
      default: printf("No Signal Available\n");
}
```

switch Statement

switch Example 2

```
switch(letter)
{
      case 'a':
         printf("Letter 'a' found.\n");
         break;
      case 'b':
         printf("Letter 'b' found.\n");
         break;
      case 'c':
         printf("Letter 'c' found.\n");
         break;
      default:   printf("Letter not in list.\n");
}
```

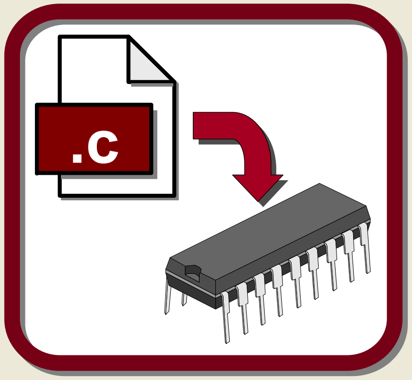
switch Statement

switch Example 3

```
switch(channel)
{
      case 4 ... 7:
         printf("VHF Station\n"); break;
      case 9 ... 12:
         printf("VHF Station\n"); break;
      case 3:
      case 8:
      case 13:
         printf("Weak Signal\n"); break;
      case 14 ... 69:
         printf("UHF Station\n"); break;
      default:
         printf("No Signal Available\n");
}
```

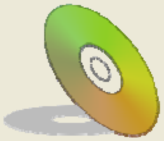
Apply this case to channel 4, 5, 6, and 7

Case 3 and 8 are allowed to fall through to case 13



Lab 06

Making Decisions (switch)



On the CD

…\101_ECP\Lab06\Lab06.mcw

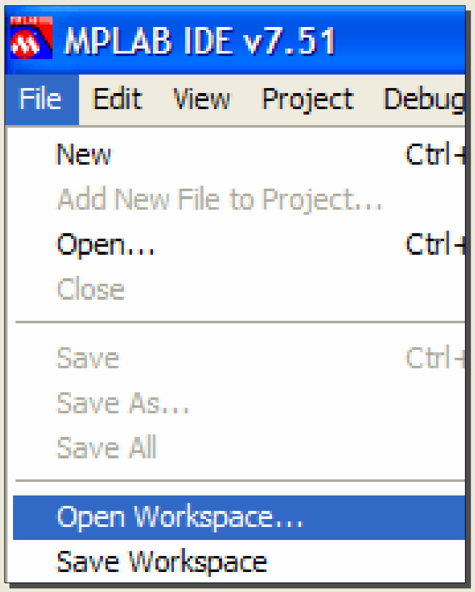
Lab 06

Making Decisions (`switch`)

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab06\Lab06.mcw



1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 06

Making Decisions (switch)

Solution: Step 1

```
/*###############################################################################
# TASK:   Write a switch statement to print the network's initials with the
#         channel (based on Chicago TV stations).
#         * If channel = 2, print "CBS 2" to the output window.
#         * If channel = 5, print "NBC 5" to the output window.
#         * If channel = 7, print "ABC 7" to the output window.
#         * For all other channels, print "--- #" to the output window,
#           where "#" is the channel number.
#         (HINT: Use printf(), and use the newline character '\n' at the end
#          of each string you print to the output window.)
# NOTE:   The switch statement is in a loop that will execute 9 times.  Each
#         pass through the loop, 'channel' will be incremented.  The output
#         window should display a line of text for channels 2 to 10.
#
# STEP 1: Open a switch statement on the variable 'channel'
###############################################################################*/
//Begin switch statement
switch(channel)
{
```

Lab 06

Making Decisions (`switch`)

Solution: Steps 2 and 3

```
/*###############################################################################
# STEP 2: Write case for channel = CBS (CBS is a constant defined to equal 2)
###############################################################################*/
case CBS:                            //If channel = CBS (CBS = 2)
{

    printf("CBS %d\n", channel);  //Display string "CBS 2" followed by newline
    break;                          //Prevent fall through to next case
}

/*###############################################################################
# STEP 3: Write case for channel = NBC (NBC is a constant defined to equal 5)
#         This should look almost identical to step 2.
###############################################################################*/
case NBC:                            //If channel = NBC (NBC = 5)
{
    printf("NBC %d\n", channel);  //Display string "NBC 5" followed by newline
    break;                          //Prevent fall through to next case
}
```

Lab 06

Making Decisions (`switch`)

Solution: Steps 4 and 5

```
/*###############################################################################
# STEP 4: Write case for channel = ABC (ABC is a constant defined to equal 7)
#         This should look almost identical to step 2.
###############################################################################*/
case ABC:                               //If channel = ABC (ABC = 7)
{
    printf("ABC %d\n", channel);  //Display string "ABC 7" followed by newline
    break;                        //Prevent fall through to next case
}

/*###############################################################################
# STEP 5: Write default case.  If channel is anything other than those
#         listed above, this is what should be done.  For these cases, you
#         need to print the string "--- #" where "#" is the channel number.
#         For example, if channel = 6, you should print "--- 6".
###############################################################################*/
default:                                //For all other channels
{
    printf("--- %d\n", channel);  //Display string "--- #" followed by newline
}
```

Lab 06

Conclusions

- `switch` provides a more elegant decision making structure than `if` for multiple conditions (if – else if – else if – else if…)
- The drawback is that the conditions may only be constants (match a variable's state to a particular value)

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.9
Loops

`for` Loop

Syntax

$$\texttt{for (}expression_1\texttt{; }expression_2\texttt{; }expression_3\texttt{)}$$
$$\quad statement$$

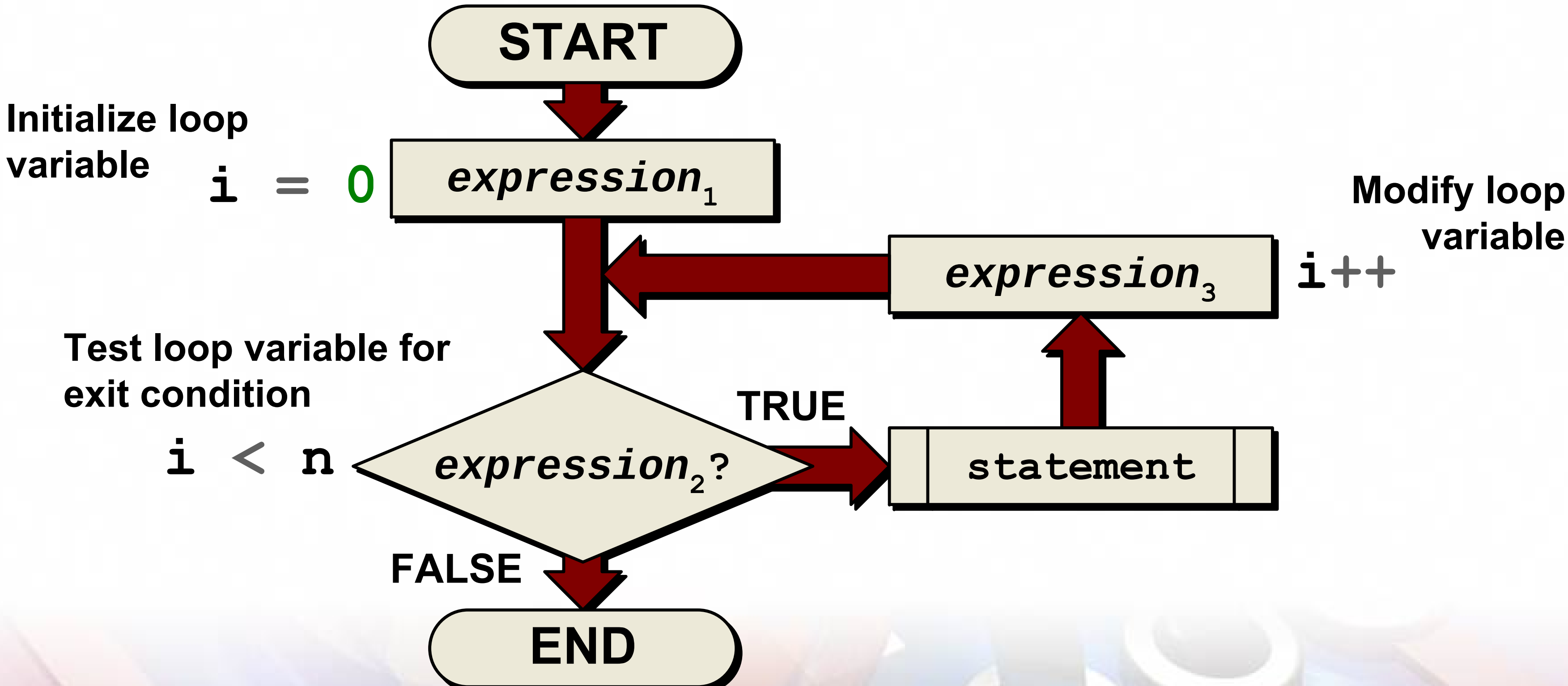
- $expression_1$ initializes a loop count variable once at start of loop (e.g. `i = 0`)
- $expression_2$ is the test condition – the loop will continue while this is true (e.g. `i <= 10`)
- $expression_3$ is executed at the end of each iteration – usually to modify the loop count variable (e.g. `i++`)

for Loop

Flow Diagram

Syntax

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



for Loop

Example (Code Fragment)

```
int i;

for (i = 0; i < 5; i++)
{
        printf("Loop iteration #%d\n", i);
}
```

Expected Output:

```
Loop iteration 0
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
```

`for` Loop

- Any or all of the three expressions may be left blank (semi-colons must remain)
- If *expression*$_1$ or *expression*$_3$ are missing, their actions simply disappear
- If *expression*$_2$ is missing, it is assumed to always be true

Note

Infinite Loops

A `for` loop without any expressions will execute indefinitely (can leave loop via `break` statement)

```
for ( ; ; )
{
    …
}
```

`while` Loop

Syntax

```
while (expression) statement
```

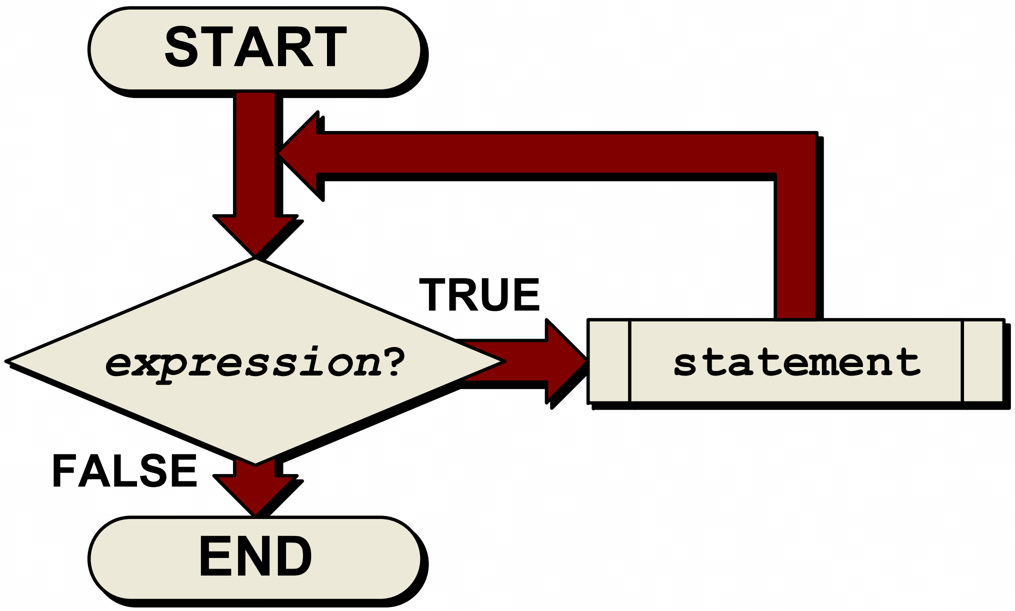
- If *expression* is true, *statement* will be executed and then *expression* will be re-evaluated to determine whether or not to execute *statement* again
- It is possible that *statement* will never execute if *expression* is false when it is first evaluated

`while` Loop

Flow Diagram

Syntax

```
while (expression) statement
```



while Loop

Example

Example (Code Fragment)

```c
int i = 0;

while (i < 5)
{
    printf("Loop iteration #%d\n", i++);
}
```

- `int i = 0;` ← Loop counter initialized outside of loop
- `while (i < 5)` ← Condition checked at start of loop iterations
- `i++` ← Loop counter incremented manually inside loop

Expected Output:

```
Loop iteration 0
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
```

`while` Loop

- The *expression* must always be there, unlike with a `for` loop
- `while` is used more often than `for` when implementing an infinite loop, though it is only a matter of personal taste
- Frequently used for main loop of program

Note

Infinite Loops

A `while` loop with *expression* = 1 will execute indefinitely (can leave loop via `break` statement)

```
while (1)
{
    ...
}
```

`do-while` Loop

Syntax

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

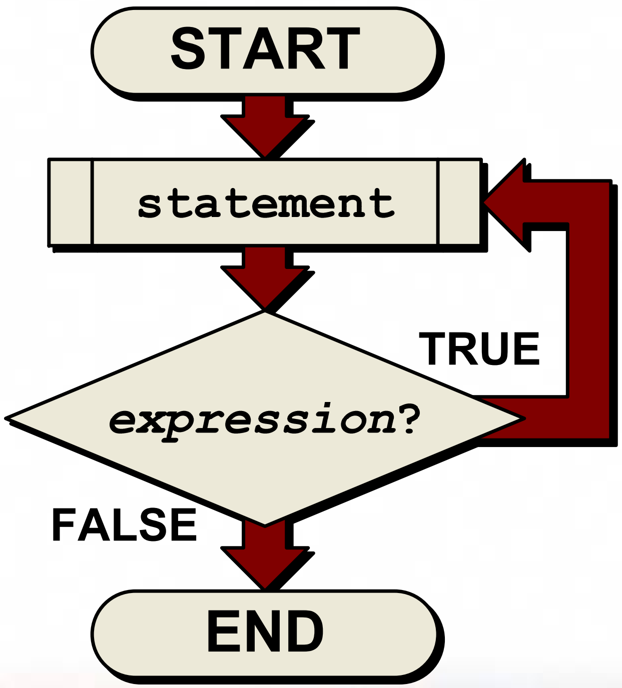
- *`statement`* is executed and then *`expression`* is evaluated to determine whether or not to execute *`statement`* again
- *`statement`* will always execute at least once, even if the expression is false when the loop starts

do-while Loop

Flow Diagram

Syntax

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



do-while Loop

Example

Example (Code Fragment)

```
int i = 0;        // Loop counter initialized outside of loop

do
{
    printf("Loop iteration #%d\n", i++);
} while (i < 5);  // Condition checked at end of loop iterations
```

Loop counter initialized outside of loop

Loop counter incremented manually inside loop

Condition checked at end of loop iterations

Expected Output:

```
Loop iteration 0
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
```

`break` Statement

Syntax

```
break;
```

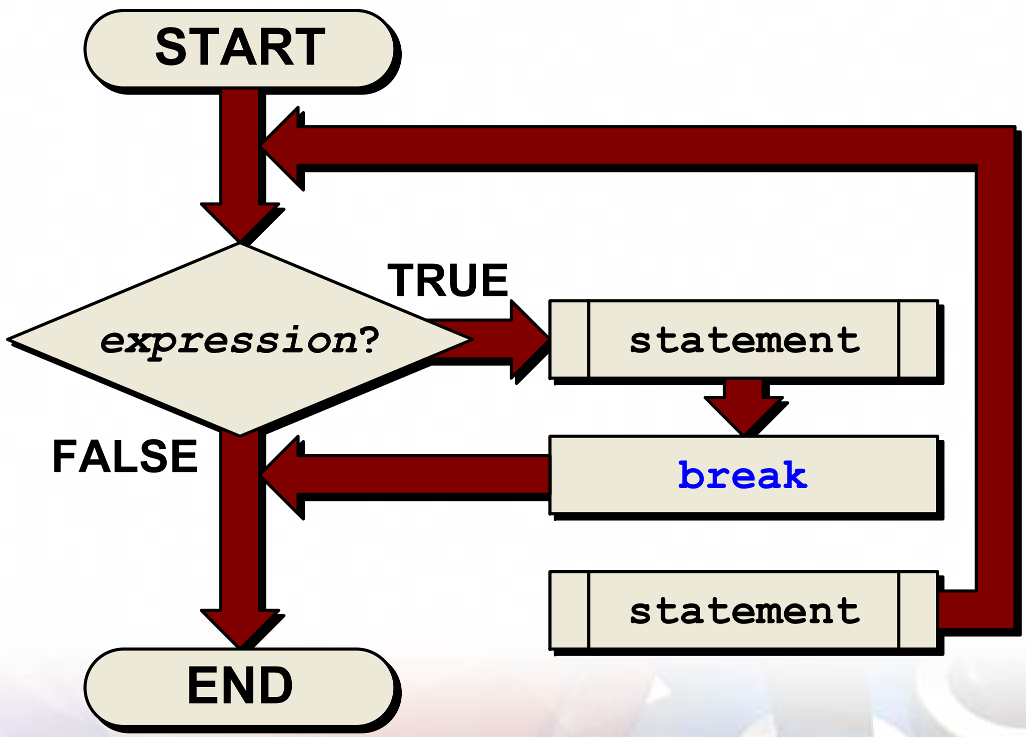
- **Causes immediate termination of a loop even if the exit condition hasn't been met**
- **Exits from a `switch` statement so that execution doesn't fall through to next `case` clause**

break Statement

Flow Diagram Within a while Loop

Syntax

```
break;
```



break Statement

Example

Example (Code Fragment)

```
int i = 0;

while (i < 10)
{
    i++;
    if (i == 5) break;
    printf("Loop iteration #%d\n", i);
}
```

Exit from the loop when i = 5.
Iteration 6-9 will not be executed.

Expected Output:

```
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
```

`continue` Statement

Syntax

```
continue;
```

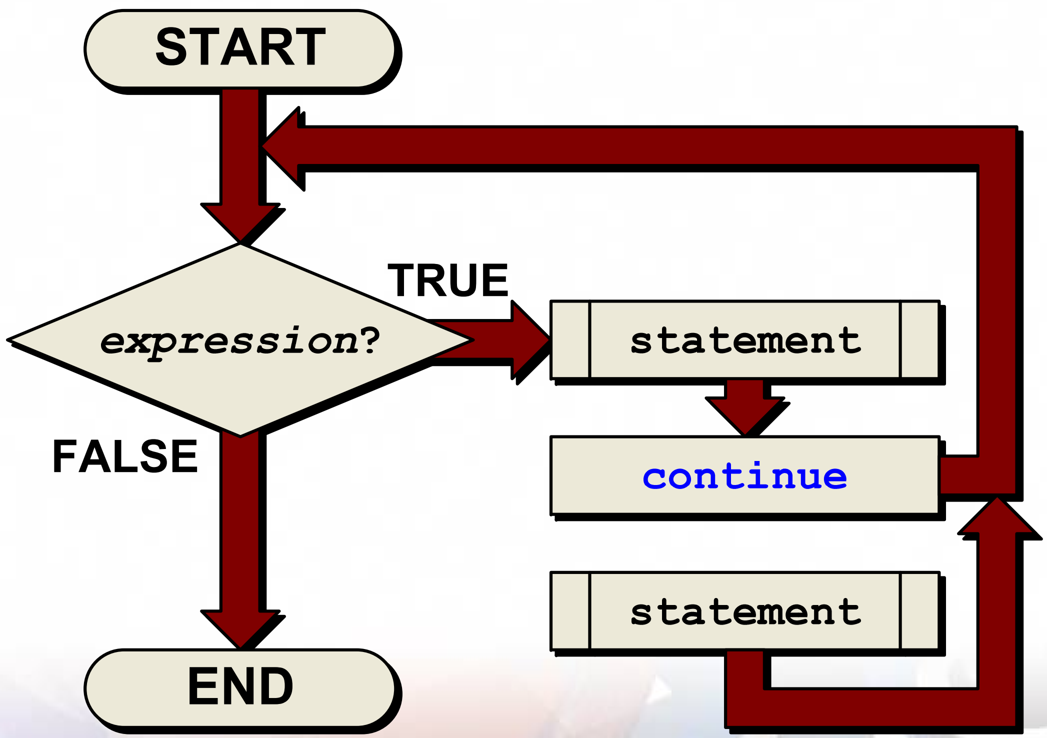
- **Causes program to jump back to the beginning of a loop without completing the current iteration**

continue Statement

Flow Diagram Within a while Loop

Syntax

```
continue;
```



continue Statement

Example

Example (Code Fragment)

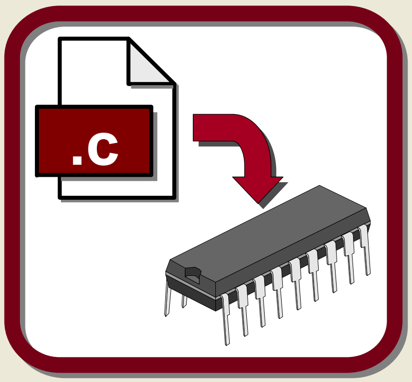
```
int i = 0;

while (i < 6)
{
    i++;
    if (i == 2) continue;
    printf("Loop iteration #%d\n", i);
}
```

Skip remaining iteration when i = 2. Iteration 2 will not be completed.

Expected Output:

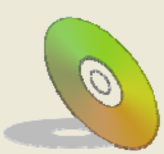
```
Loop iteration 1
Loop iteration 3
Loop iteration 4
Loop iteration 5
```

Iteration 2 does not print



Lab 07

Loops



On the CD

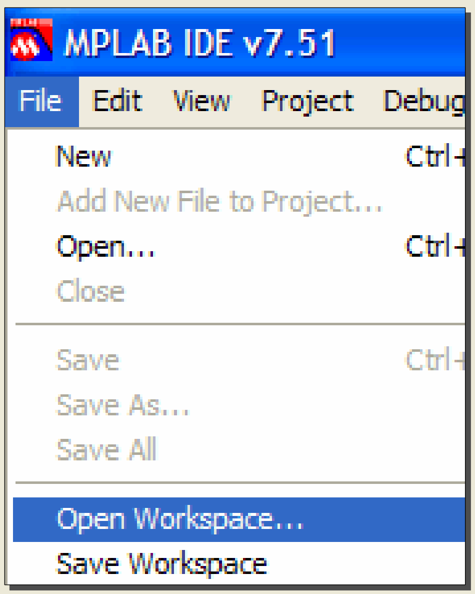
…\101_ECP\Lab07\Lab07.mcw

Lab 07
Loops

- Open the project's workspace:

On the lab PC

C:\RTC\101_ECP\Lab07\Lab07.mcw



1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 07
Loops

Solution: Step 1

```
/*###############################################################################
# STEP 1: Create a for loop to iterate the block of code below.  The loop
#         should do the following:
#         * Initialize counter1 to 1
#         * Loop as long as counter1 is less than 5
#         * Increment counter1 on each pass of the loop
#         (HINT: for(init; test; action))
###############################################################################*/
//Write the opening line of the for loop
for( counter1 = 1 ; counter1 < 5 ; counter1++)
{
     intVariable1 *= counter1;
     printf("FOR: intVariable1 = %d, counter1 = %d\n", intVariable1, counter1);
}
//end of for loop block
```

Lab 07

Loops

Solution: Step 2

```
/*###############################################################################
# STEP 2: Create a while loop to iterate the block of code below.  The loop
#         should run until charVariable1 is 0.
###############################################################################*/
//Loop as long as charVariable1 is not 0
while( charVariable1 != 0)
{
     charVariable1--;
     charVariable2 += 5;
     printf("WHILE: charVariable1 = %d, charVariable2 = %d\n",
                    charVariable1, charVariable2);
}
//end of while loop block
```

Lab 07

Loops

Solution: Step 3

```
/*###############################################################################
# STEP 3: Create a do...while loop to iterate the block of code below.
#         The loop should run until counter1 is greater than 100
###############################################################################*/
do                                //Write opening line of do loop
{
    counter1 += 5;
    counter2 = counter1 * 3;
    printf("DO: counter1 = %d, counter2 = %d\n", counter1, counter2);
} while(counter1 <= 100);         //Write closing line of loop - test counter1
//end of do...while block
```

Lab 07
Conclusions

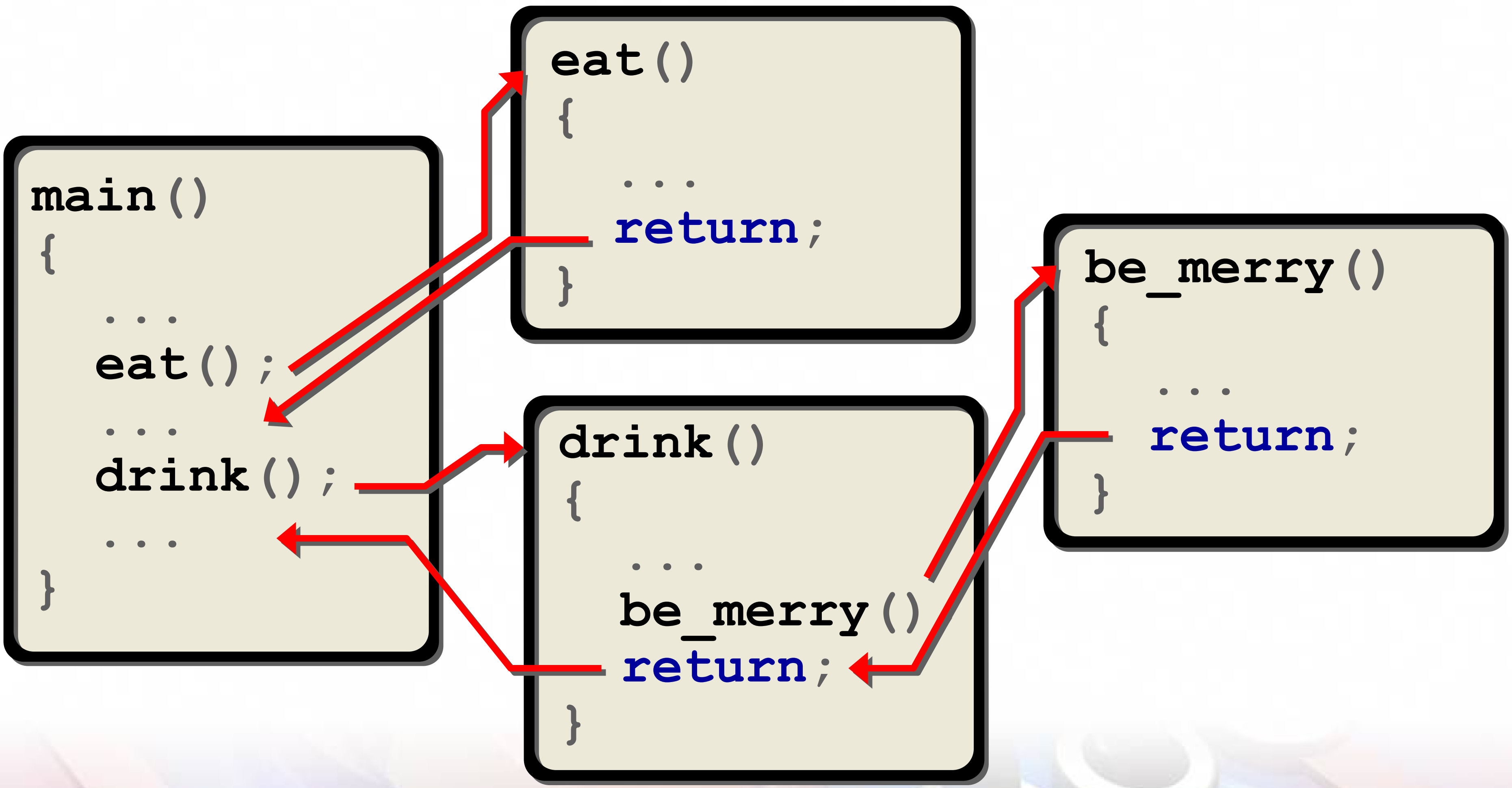
- C Provides three basic looping structures
  - for – checks loop condition at top, automatically executes iterator at bottom
  - while – checks loop condition at top, you must create iterator if needed
  - do…while – checks loop condition at bottom, you must create iterator if needed

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.10
Functions

Functions

Program Structure



Functions

What is a function?

Definition

Functions are self contained program segments designed to perform a specific, well defined task.

- **All C programs have one or more functions**
- **The `main()` function is required**
- **Functions can accept parameters from the code that calls them**
- **Functions usually return a single value**
- **Functions help to organize a program into logical, manageable segments**

Functions

Remember Algebra Class?

- Functions in C are conceptually like an algebraic function from math class…

Function Definition

Function Name → $f(x) = x^2 + 4x + 3$

↑ Function Parameter

- If you pass a value of 7 to the function: f(7), the value 7 gets "copied" into x and used everywhere that x exists within the function definition: $f(7) = 7^2 + 4*7 + 3 = 80$

Functions

Definitions

Syntax

Data type of return *expression*

Name

Parameter List (optional)

```
type identifier(type1 arg1,…,typen argn)
{
    declarations
    statements
    return expression;
}
```

Body

Header

Return Value (optional)

Functions

Function Definitions: Syntax Examples

Example

```
int maximum(int x, int y)
{
  int z;

  z = (x >= y) ? x : y;
  return z;
}
```

Example – A more efficient version

```
int maximum(int x, int y)
{
  return ((x >= y) ? x : y);
}
```

Functions

Function Definitions: Return Data Type

Syntax

```
type identifier(type1 arg1,…,typen argn)
{
  declarations
  statements
  return expression;
}
```

- A function's *type* must match the type of data in the return *expression*

Functions

Function Definitions: Return Data Type

- **A function may have multiple return statements, but only one will be executed and they must all be of the same type**

Example

```
int bigger(int a, int b)
{
      if (a > b)
            return 1;
      else
            return 0;
}
```

Functions

Function Definitions: Return Data Type

- The function type is `void` if:
  - The `return` statement has no *expression*
  - The `return` statement is not present at all
- This is sometimes called a *procedure function* since nothing is returned

Example

```
void identifier(type1 arg1,…,typen argn)
{
    declarations
    statements
    return;
}
```

`return;` may be omitted if nothing is being returned

Functions

Function Definitions: Parameters

- **A function's parameters are declared just like ordinary variables, but in a comma delimited list inside the parentheses**
- **The parameter names are only valid inside the function (local to the function)**

Syntax

```
type identifier(type1 arg1,…,typen argn)
{
  declarations
  statements
  return expression;
}
```

Function Parameters: $type_1\ arg_1, \ldots, type_n\ arg_n$

Functions

Function Definitions: Parameters

- Parameter list may mix data types
  - `int foo(int x, float y, char z)`
- Parameters of the same type must be declared separately – in other words:
  - `int maximum(int x, y)` will not work
  - `int maximum(int x, int y)` is correct

Example

```
int maximum(int x, int y)
{
  return ((x >= y) ? x : y);
}
```

Functions

Function Definitions: Parameters

- If no parameters are required, use the keyword `void` in place of the parameter list when defining the function

Example

```
type identifier(void)
{
      declarations
      statements
      return expression;
}
```

Functions

How to Call / Invoke a Function

Function Call Syntax

- **No parameters and no return value**

```
foo();
```

- **No parameters, but with a return value**

```
x = foo();
```

- **With parameters, but no return value**

```
foo(a, b);
```

- **With parameters and a return value**

```
x = foo(a, b);
```

Functions

Function Prototypes

- **Just like variables, a function must be declared before it may be used**
- **Declaration must occur before main() or other functions that use it**
- **Declaration may take two forms:**
  - **The entire function definition**
  - **Just a function prototype – the function definition itself may then be placed anywhere in the program**

Functions

Function Prototypes

- Function prototypes may be take on two different formats:
  - An exact copy of the function header:

Example – Function Prototype 1

```
int maximum(int x, int y);
```

  - Like the function header, but without the parameter names – only the types need be present for each parameter:

Example – Function Prototype 2

```
int maximum(int, int);
```

Functions

Declaration and Use: Example 1

Example 1

```
int a = 5, b = 10, c;

int maximum(int x, int y)
{
  return ((x >= y) ? x : y);
}

int main(void)
{
  c = maximum(a, b);
  printf("The max is %d\n", c)
}
```

Function is *declared* and *defined* before it is used in main()

Functions

Declaration and Use: Example 2

Example 2

```
int a = 5, b = 10, c;

int maximum(int x, int y);

int main(void)
{
  c = maximum(a, b);
  printf("The max is %d\n", c)
}

int maximum(int x, int y)
{
  return ((x >= y) ? x : y);
}
```

Function is ***declared*** with prototype **before** use in main()

Function is ***defined*** after it is used in main()

Functions

Passing Parameters by Value

- Parameters passed to a function are passed *by value*
- Values passed to a function are copied into the local parameter variables
- The original variable that is passed to a function cannot be modified by the function since only a copy of its value was passed

Functions

Passing Parameters by Value

Example

```
int a, b, c;

int foo(int x, int y)
{
  x = x + (++y);
  return x;
}

int main(void)
{
  a = 5;
  b = 10;
  c = foo(a, b);
}
```

The *value* of `a` is *copied* into `x`.
The *value* of `b` is *copied* into `y`.
The function does not change the value of `a` or `b`.

Functions

Recursion

- A function can call itself repeatedly
- Useful for iterative computations (each action stated in terms of previous result)
- Example: Factorials (5! = 5 * 4 * 3 * 2 * 1)

Example

```
long int factorial(int n)
{
        if (n <= 1)
                return(1);
        else
                return(n * factorial(n - 1));
}
```

Functions

Evaluation of Recursive Functions

- **Evaluation of 5!**
  (based on code from previous slide)

Recursive iterations of function		Partial results pushed on stack	Factorial term replaced with result of expression above	Result evaluated from TOS downward
[0]	1! =	1	= 1	
[1]	2! =	2 * 1!	= 2 * 1 = 2	
[2]	3! =	3 * 2!	= 3 * 2 = 6	
[3]	4! =	4 * 3!	= 4 * 6 = 24	
[4]	5! =	5 * 4!	= 5 * 24 = 120	

Conceptual evaluation of recursive function

Functions and Scope

Parameters

- A function's parameters are local to the function – they have no meaning outside the function itself
- Parameter names may have the same identifier as a variable declared outside the function – the parameter names will take precedence inside the function

These are not the same n.

```
int n;
long int factorial(int n){…}
```

Functions and Scope

Variables Declared Within a Function

- **Variables declared within a function block are local to the function**

Example

```
int x, y, z;

int foo(int n)
{
        int a;

        a += n;
}
```

The n refers to the function parameter n

The a refers to the a declared locally within the function body

Functions and Scope

Variables Declared Within a Function

- **Variables declared within a function block are not accessible outside the function**

Example

```
int x;
int foo(int n)
{
        int a;
        return (a += n);
}
int main(void)
{
        x = foo(5);
        x = a;
}
```

This will generate an error. a may not be accessed outside of the function where it was declared.

Functions and Scope

Global versus Local Variables

Example

```
int x = 5;

int foo(int y)
{
  int z = 1;
  return (x + y + z);
}

int main(void)
{
  int a = 2;
  x = foo(a);
  a = foo(x);
}
```

x can see be seen by everybody

foo's local parameter is y
foo's local variable is z
foo cannot see main's a
foo can see x

main's local variable is a
main cannot see foo's y or z
main can see x

Functions and Scope

Parameters

- "Overloading" variable names:

n Declared Locally and Globally

```
int n;

int foo(int n)
{
      ...
      y += n;
      ...
}
```

local n hides global n

n Declared Globally Only

```
int n;

int foo(int x)
{
      ...
      y += n;
      ...
}
```

A locally defined identifier takes precedence over a globally defined identifier.

Functions and Scope

Parameters

Example

```
int n;

int foo(int n)
{
      y += n;
}

int bar(int n)
{
      z *= n;
}
```

- **Different functions may use the same parameter names**
- **The function will only use its own parameter by that name**

Functions and Scope

#define Within a Function

Example

```
#define x 2

void test(void)
{
    #define x 5
    printf("%d\n", x);
}

void main(void)
{
    printf("%d\n", x);
    test();
}
```

Running this code will result in the following output in the Uart1 IO window:

```
5
5
```

Why?

Remember: #define is used by the preprocessor to do text substitution before the code is compiled.

Functions

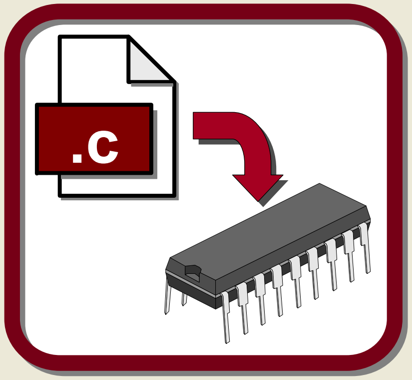
Historical Note

- C originally defined functions like this:

```
int maximum(x, y)
int x, int y
{
   return ((x >= y) ? x : y);
}
```

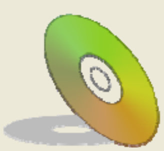
- Do not use the old method – use the new one only:

```
int maximum(int x, int y)
{
   return ((x >= y) ? x : y);
}
```



Lab 08

Functions



On the CD

...\101_ECP\Lab08\Lab08.mcw

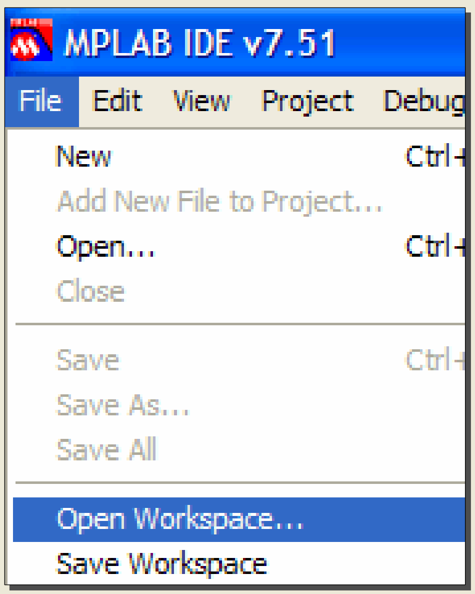
Lab 08

Functions

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab08\Lab08.mcw



1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 08

Functions

Solution: Step 1

```
/*###############################################################################
# STEP 1: Write two function prototypes based on the following information:
#         + Function Name: multiply_function()
#           - Parameters: int x, int y
#           - Return type: int
#         + Function Name: divide_function()
#           - Parameters: float x, float y
#           - Return type: float
###############################################################################*/

int multiply_function( int x, int y);       //multiply_function() prototype

float divide_function( float x, float y );  //divide_function() prototype
```

Lab 08

Functions

Solution: Step 2

```
/*##########################################################################
# STEP 2: Call the multiply_function() and divide_function().
#         (a) Pass the variables intVariable1 and intVariable2 to the
#             multiply_function().
#         (b) Store the result of multiply_function() in the variable "product".
#         (c) Pass the variables floatVariable1 and floatVariable2 to the
#             divide_function().
#         (d) Store the result of divide_function() in the variable "quotient".
##########################################################################*/

//Call multiply_function
product = multiply_function( intVariable1 , intVariable2 );

//Call divide_function
quotient = divide_function( floatVariable1 , floatVariable2 );

// intQuotient will be 0 since it is an integer
intQuotient = divide_function( floatVariable1 , floatVariable2 );
```

Lab 08
Functions

Solution: Steps 3 and 4

```
/*#############################################################################
# STEP 3: Write the function multiply_function().  Use the function prototype
#         you wrote in STEP 1 as the function header.  In the body, all you
#         need to do is return the product of the two input parameters (x * y)
#############################################################################*/
//Function Header
int multiply_function( int x, int y)
{
    return (x * y);                   //Function Body
}

/*#############################################################################
# STEP 4: Write the function divide_function().  Use the function prototype
#         you wrote in STEP 1 as the function header.  In the body, all you
#         need to do is return the quotient of the two input parameters (x / y)
#############################################################################*/
//Function Header
float divide_function( float x, float y )
{
    return (x / y);                   //Function Body
}
```

Lab 08

Conclusions

- Functions provide a way to modularize code
- Functions make code easier to maintain
- Functions promote code reuse

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.11
Multi-File Projects and Storage Class Specifiers

Storage Class Specifiers

Scope and Lifetime of Variables

- Scope and lifetime of a variable depends on its storage class:
  - Automatic Variables
  - Static Variables
  - External Variables
  - Register Variables
- Scope refers to where in a program a variable may be accessed
- Lifetime refers to how long a variable will exist or retain its value

Storage Class Specifiers

Automatic Variables

- Local variables declared inside a function
  - Created when function called
  - Destroyed when exiting from function
- `auto` keyword *usually* not required – local variables are automatically automatic*
- Typically created on the stack

```
int foo(int x, int y)
{
  int a, b;
  ...
```

Automatic Variables

*Except when the compiler provides an option to make parameters and locals static by default.

Storage Class Specifiers

`auto` Keyword with Variables

```
int foo(auto int x, auto int y)
{
   ... ;
}
```

- `auto` is almost never used
- Many books claim it has no use at all
- Some compilers still use `auto` to explicitly specify that a variable should be allocated on the stack when a different method of parameter passing is used by default

Storage Class Specifiers

Static Variables

- Given a permanent address in memory
- Exist for the entire life of the program
  - Created when program starts
  - Destroyed when program ends
- Global variables are always static (cannot be made automatic using `auto`)

```
int x;      ← Global variable is always static

int main(void)
{
   ...
```

Storage Class Specifiers

static Keyword with Variables

- A variable declared as static inside a function retains its value between function calls (not destroyed when exiting function)
- Function parameters cannot be static with some compilers (MPLAB® C30)

```
int foo(int x)
{
    static int a = 0;
    ...
    a += x;
    return a;
}
```

a will remember its value from the last time the function was called.
If given an initial value, it is only initialized when first created – not during each function call

Storage Class Specifiers

External Variables

- Variables that are ***defined*** outside the scope where they are used
- Still need to be ***declared*** within the scope where they are used
- `extern` keyword used to tell compiler that a variable defined elsewhere will be used within the current scope

External Variable Declaration Syntax:

```
extern type identifier;
```

External Variable Declaration Example:

```
extern int x;
```

Storage Class Specifiers

External Variables

- A variable *declared* as `extern` within a function is analogous to a function prototype – the variable may be *defined* outside the function after it is used

Example

```
int foo(int x)
{
    extern int a;
    ...
    return a;
}

int a;
```

Storage Class Specifiers

External Variables

- A variable *declared* as `extern` outside of any function is used to indicate that the variable is *defined* in another source file – memory only allocated when it's *defined*

Main.c

```c
extern int x;

int main(void)
{
    x = 5;
    ...
}
```

SomeFileInProject.c

```c
int x;

int foo(void)
{
    ...
}
```


Storage Class Specifiers

Register Variables

- Variables placed in a processor's "hardware registers" for higher speed access than with external RAM (mostly used for microprocessor-based systems)
- Doesn't *usually* make sense in embedded microcontroller system where RAM is integrated into processor package
- May be done with PIC® MCU/dsPIC® DSC, but it is architecture/compiler specific…

Storage Class Specifiers

Scope of Functions

- **Scope of a function depends on its storage class:**
  - **Static Functions**
  - **External Functions**
- **Scope of a function is either local to the file where it is defined (static) or globally available to any file in a project (external)**

Storage Class Specifiers

External Functions

- Functions by default have global scope within a project
- `extern` keyword not required, but function prototype is required in calling file (or .h)

Main.c

```c
int foo(void);

int main(void)
{
    ...
    x = foo();
}
```

SomeFileInProject.c

```c
int foo(void)
{
    ...
}
```

Storage Class Specifiers

Static Functions

- If a function is declared as `static`, it will only be available within the file where it was declared (makes it a local function)

Main.c

```
int foo(void);

int main(void)
{
    ...
    x = foo();
}
```

SomeFileInProject.c

```
static int foo(void)
{
    ...
}
```

Storage Class Specifiers

Library Files and Header Files

INCLUDE DECLARATIONS

Main.c

```c
#include <LibFile.h>

int x;

int main(void)
{
      x = foo();
      myVar = x;
}
```

USE

DECLARATIONS

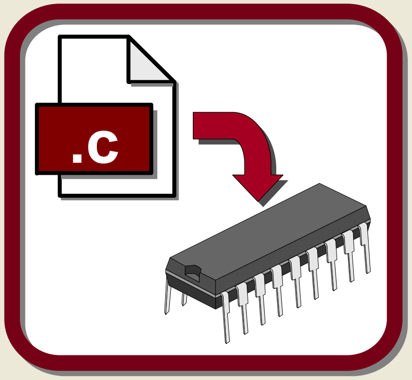
LibFile.h

```c
extern int myVar;

int foo(void);
```

LibFile.c

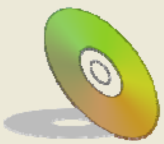
```c
int myVar;

int foo(void)
{
      ...
}
```

DEFINITIONS



Lab 09

Multi-File Projects



On the CD

...\101_ECP\Lab09\Lab09.mcw

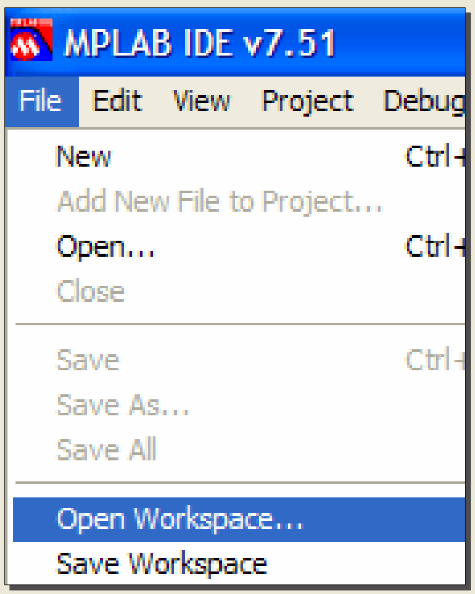
Lab 09

Multi-File Projects

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab09\Lab09.mcw



1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 09

Multi-File Projects

Solution: Step 1a and 1b (File1_09.h)

```
/*############################################################################
# STEP 1a: Add variable declarations to make the variables defined in
#          File1_09.c available to any C source file that includes this
#          header file. (intVariable1, intVariable2, product)
############################################################################*/
//Reference to externally defined "intVariable1"
extern int intVariable1;
//Reference to externally defined "intVariable2"
extern int intVariable2;
//Reference to externally defined "product"
extern int product;

/*###############################################################################
# STEP 1b: Add a function prototype to make multiply_function() defined in
#          File1_09.c available to any C source file that includes this header
#          file.
###############################################################################*
//Function prototype for multiply_function()
int multiply_function(int x, int y);
```

Lab 09

Multi-File Projects

Solution: Step 2a and 2b (File2_09.h)

```
/*############################################################################
# STEP 2a: Add variable declarations to make the variables defined in
#          File2_09.c available to any C source file that includes this header
#          file.(floatVariable1, floatVariable2, quotient, intQuotient)
############################################################################*/
//Reference to externally defined "floatVariable1"
extern float floatVariable1;
//Reference to externally defined "floatVariable2"
extern float floatVariable2;
//Reference to externally defined "quotient"
extern float quotient;
//Reference to externally defined "intQuotient"
extern int intQuotient;

/*############################################################################
# STEP 2b: Add a function prototype to make divide_function() defined in
#          File2_09.c available to any C source file that includes this header
#          file.
############################################################################*/
//Function prototype for divide_function()
float divide_function(float x, float y );
```

Lab 09

Conclusions

- **Multi-file projects take the concept of functions further, by providing an additional level of modularization**
- **Globally declared variables and all normal functions are externally available if extern declarations and function prototypes are available**
- **Static functions are not available externally**

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.12

Arrays

Arrays

Definition

Arrays are variables that can store many items of the same type. The individual items known as **elements**, are stored sequentially and are uniquely identified by the array **index** (sometimes called a **subscript**).

- **Arrays:**
  - **May contain any number of elements**
  - **Elements must be of the same type**
  - **The index is zero based**
  - **Array size (number of elements) must be specified at declaration**

Arrays

How to Create an Array

Arrays are declared much like ordinary variables:

Syntax

```
type arrayName[size];
```

- *size* refers to the number of elements
- *size* must be a constant integer

Example

```
int a[10];      // An array that can hold 10 integers

char s[25];     // An array that can hold 25 characters
```

Arrays

How to Initialize an Array at Declaration

Arrays may be initialized with a list when declared:

Syntax

```
type arrayName[size] = {item1,…,itemn};
```

- The items must all match the *type* of the array

Example

```
int a[5] = {10, 20, 30, 40, 50};

char b[5] = {'a', 'b', 'c', 'd', 'e'};
```

Arrays

How to Use an Array

Arrays are accessed like variables, but with an index:

Syntax

```
arrayName[index]
```

- *index* may be a variable or a constant
- The first element in the array has an index of 0
- C does not provide any bounds checking

Example

```
int i, a[10];    //An array that can hold 10 integers

for(i = 0; i < 10; i++) {
  a[i] = 0;      //Initialize all array elements to 0
}
a[4] = 42;       //Set fifth element to 42
```

Arrays

Creating Multidimensional Arrays

Add additional dimensions to an array declaration:

Syntax

```
type arrayName[size1]...[sizen];
```

- Arrays may have any number of dimensions
- Three dimensions tend to be the largest used in common practice

Example

```
int a[10][10];          //10x10 array for 100 integers

float b[10][10][10];    //10x10x10 array for 1000 floats
```

Arrays

Initializing Multidimensional Arrays at Declaration

Arrays may be initialized with lists within a list:

Syntax

```
type arrayName[size0]…[sizen] =
                         {{item,…,item},
                                 ⋮
                          {item,…,item}};
```

Example

```
char a[3][3] = {{'X', 'O', 'X'},
                {'O', 'O', 'X'},
                {'X', 'X', 'O'}};

int b[2][2][2] = {{{0, 1},{2, 3}},{{4, 5},{6, 7}}};
```

Arrays

Visualizing 2-Dimensional Arrays

```
int a[3][3] = {{0, 1, 2},
               {3, 4, 5},
               {6, 7, 8}};
```

Row, Column

```
a[y][x]
```

Row 0

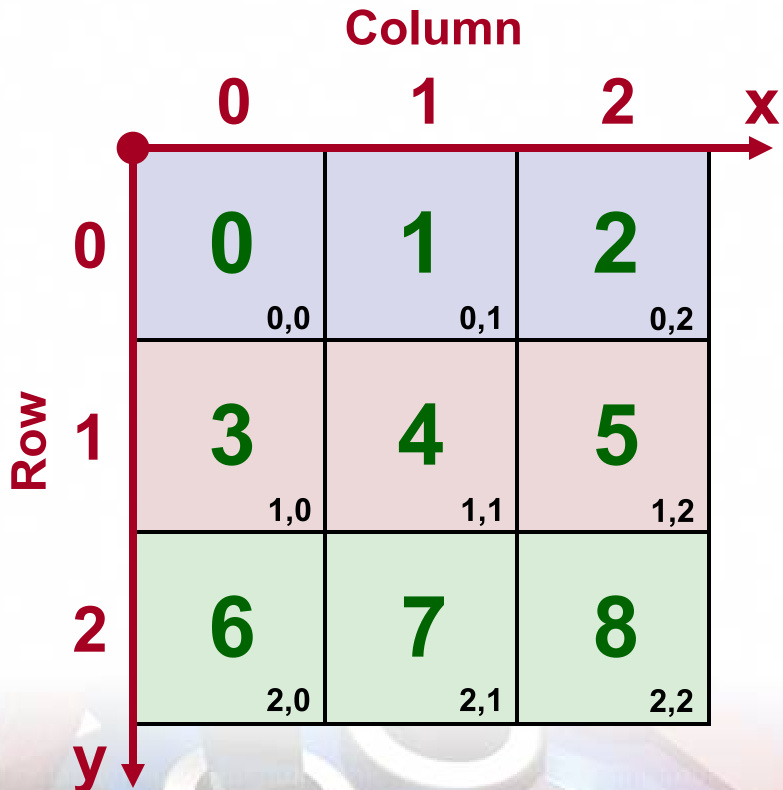
```
a[0][0] = 0;
a[0][1] = 1;
a[0][2] = 2;
```

Row 1

```
a[1][0] = 3;
a[1][1] = 4;
a[1][2] = 5;
```

Row 2

```
a[2][0] = 6;
a[2][1] = 7;
a[2][2] = 8;
```



Arrays

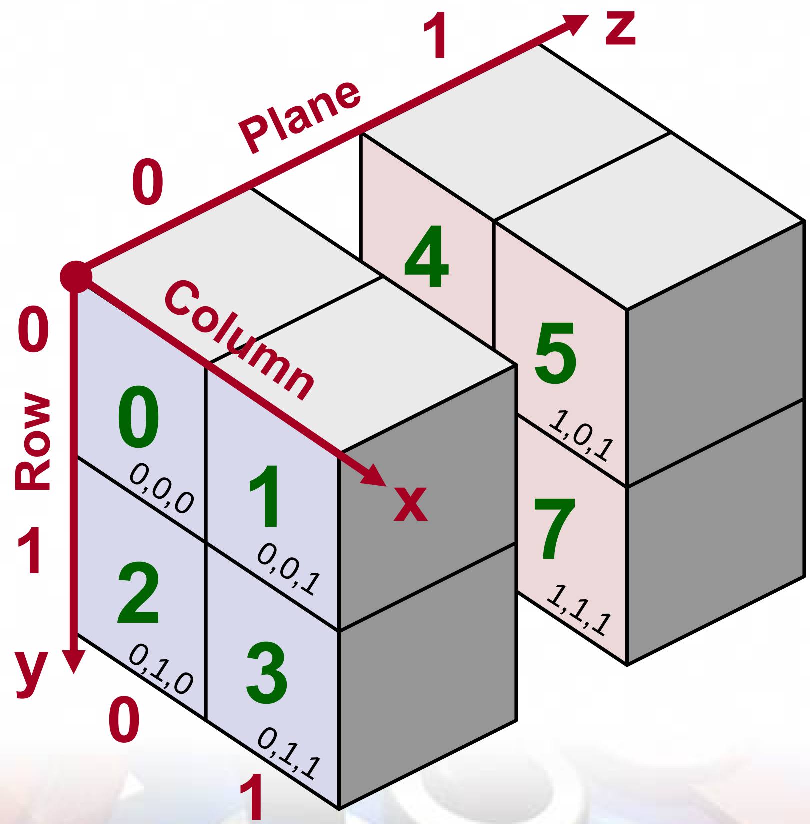
Visualizing 3-Dimensional Arrays

```
int a[2][2][2] = {{{0, 1},{2, 3}},
                  {{4, 5},{6, 7}}};
```

Plane, Row, Column

```
a[z][y][x]
```

Plane 0	`a[0][0][0] = 0;`
	`a[0][0][1] = 1;`
	`a[0][1][0] = 2;`
	`a[0][1][1] = 3;`
Plane 1	`a[1][0][0] = 4;`
	`a[1][0][1] = 5;`
	`a[1][1][0] = 6;`
	`a[1][1][1] = 7;`



Arrays

Example of Array Processing

```
/**************************************************
* Print out 0 to 90 in increments of 10
**************************************************/
int main(void)
{
      int i = 0;
      int a[10] = {0,1,2,3,4,5,6,7,8,9};

      while (i < 10)
      {
            a[i] *= 10;
            printf("%d\n", a[i]);
            i++;
      }

      while (1);
}
```

Strings

Character Arrays and Strings

Definition

Strings are arrays of `char` whose last element is a null character `'\0'` with an ASCII value of 0. C has no native string data type, so strings must always be treated as character arrays.

- **Strings:**
  - **Are enclosed in double quotes `"string"`**
  - **Are terminated by a null character `'\0'`**
  - **Must be manipulated as arrays of characters (treated element by element)**
  - **May be initialized with a string literal**

Strings

Creating a String Character Array

Strings are created like any other array of `char`:

Syntax

```
char arrayName[length];
```

- *length* must be one larger than the length of the string to accommodate the terminating null character '\0'
- A `char` array with n elements holds strings with n-1 `char`

Example

```
char str1[10];        //Holds 9 characters plus '\0'

char str2[6];         //Holds 5 characters plus '\0'
```

Strings

How to Initialize a String at Declaration

Character arrays may be initialized with string literals:

Syntax

```
char arrayName[] = "Microchip";
```

- Array size is not required
- Size automatically determined by length of string
- NULL character `'\0'` is automatically appended

Example

```
char str1[] = "Microchip"; //10 chars "Microchip\0"

char str2[6] = "Hello";    //6 chars "Hello\0"

//Alternative string declaration - size required
char str3[4] = {'P', 'I', 'C', '\0'};
```


Strings

How to Initialize a String in Code

In code, strings must be initialized element by element:

Syntax

```
arrayName[0] = char1;
arrayName[1] = char2;
             •
             •
             •
arrayName[n] = '\0';
```

- **Null character `'\0'` must be appended manually**

Example

```
str[0] = 'H';
str[1] = 'e';
str[2] = 'l';
str[3] = 'l';
str[4] = 'o';
str[5] = '\0';
```

Strings

Comparing Strings

- Strings cannot be compared using logical operators (==, !=, etc.)
- Must use standard C library string manipulation functions
- `strcmp()` returns 0 if strings equal

Example

```
char str[] = "Hello";

if (!strcmp( str, "Hello"))
  printf("The string is \"%s\".\n", str);
```

Functions

Array Parameters

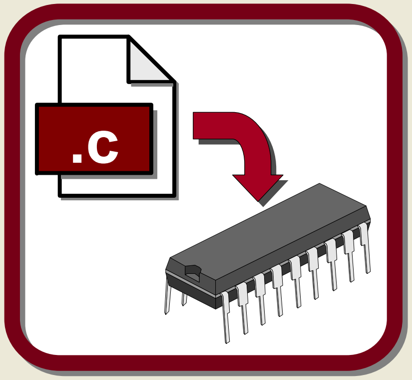
- **Arrays are passed by reference rather than by value for greater efficiency**
- **A pointer to the array, rather than the array itself is passed to the function**

This declaration…

```
void WriteLCD(char greetings[]){…}
```

…is equivalent to this declaration.

```
void WriteLCD(char *greetings){…}
```



Lab 10

Arrays

On the CD

...\101_ECP\Lab10\Lab10.mcw

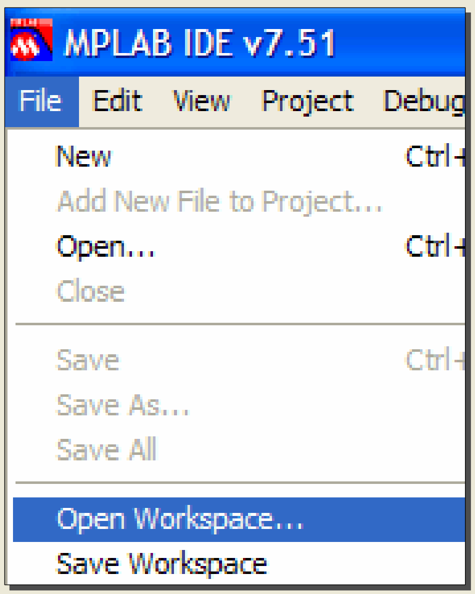
Lab 10

Arrays

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab10\Lab10.mcw



1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 10

Arrays

Solution: Step 1

```
/*###############################################################################
# STEP 1: Create two initialized arrays with 10 elements each named array1 and
#         array2 (you may use the pre-defined constant ARRAY_SIZE as part of
#         the array declaration).
#         The arrays should be initialized with the following values:
#         + array1: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
#         + array2: 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
#         Note: the elements are all of type int
###############################################################################*/
// array1 declaration & definition
int array1[ARRAY_SIZE] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
// array2 declaration & definition
int array2[ARRAY_SIZE] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

Lab 10

Arrays

Solution: Step 2

```
/*############################################################################
# STEP 2: Pass the two arrays you declared above (array1 & array2) to the
#         function add_function() (see its definition below).  Store the
#         result of the function call in the array result[].  The idea here is
#         to add each corresponding element of array1 and array2 and store the
#         result in result[].  In other words, add the first element of
#         array1[] to the first element of array2[] and store the result in
#         the first element of result[].  Next add the second elements…
############################################################################*/
// result = sum of elements of array1 & array2
result[i] = add_function(array1[i], array2[i]);
i++;
```

Lab 10

Conclusions

- **Arrays may be used to store a group of related variables of the same type under a common name**
- **Individual elements are accessed by using the array index in conjunction with the array name**
- **Arrays may be used in many places that an ordinary variable would be used**

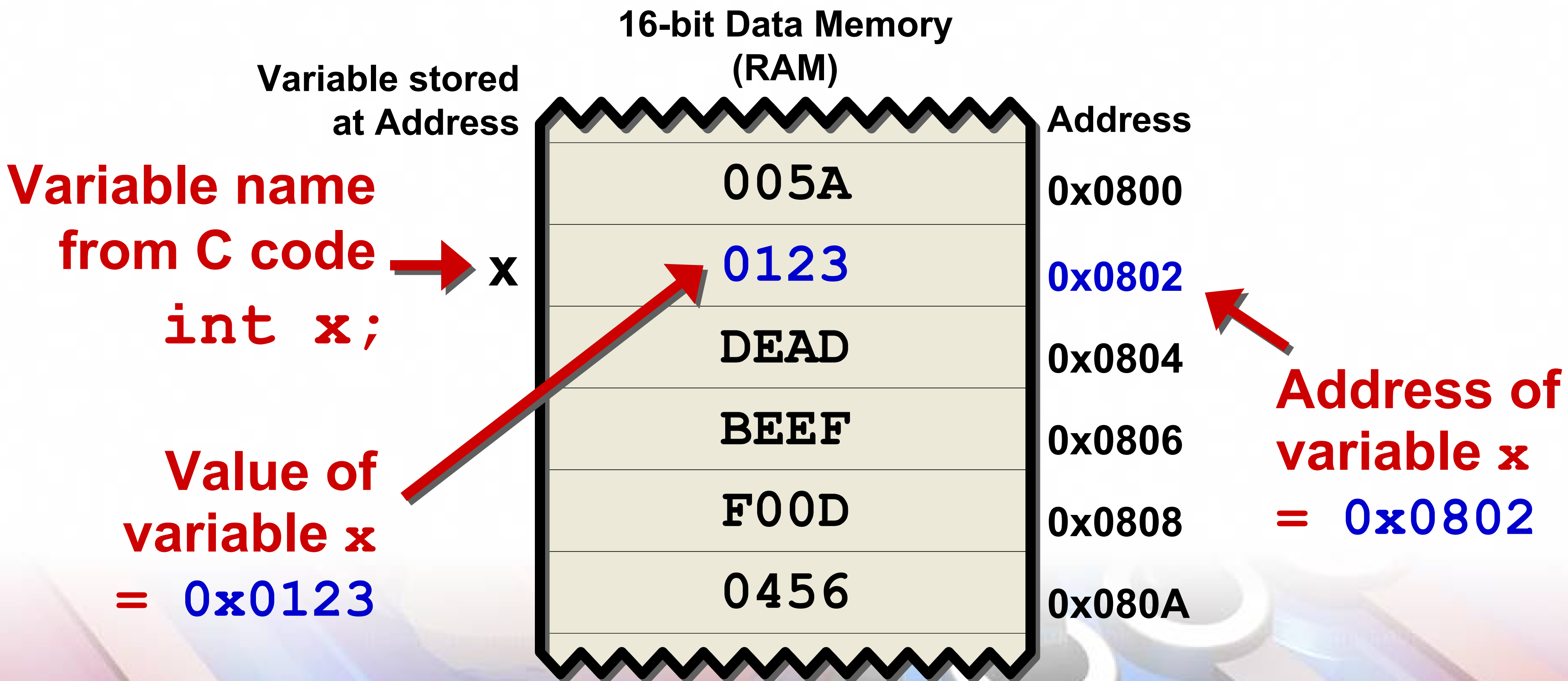
YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.13
Data Pointers

Pointers

A Variable's Address versus A Variable's Value

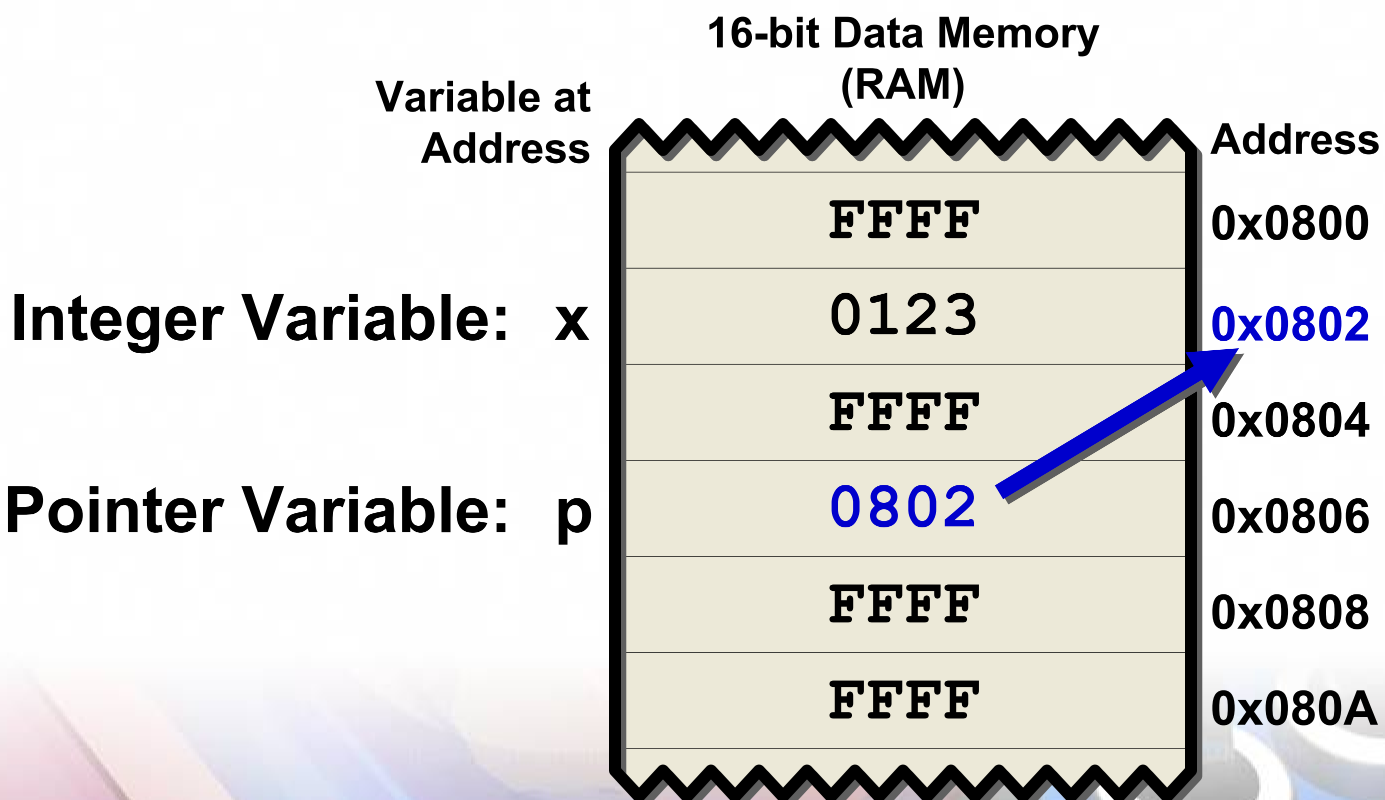
- In some situations, we will want to work with a variable's address in memory, rather than the value it contains…



Pointers

What are pointers?

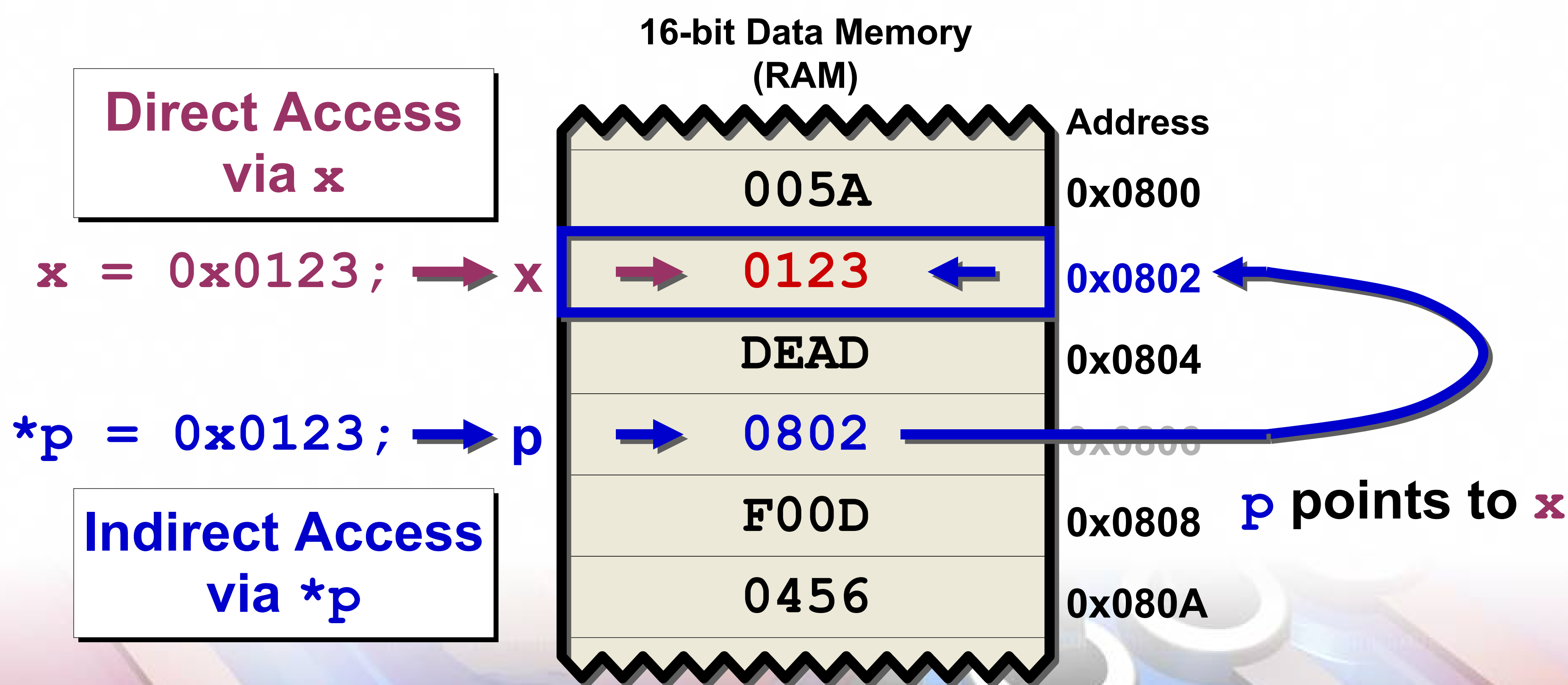
- **A pointer is a variable or constant that holds the address of another variable or function**



Pointers

What do they do?

- A pointer allows us to indirectly access a variable (just like indirect addressing in assembly language)



Pointers

Why would I want to do that?

- Pointers make it possible to write a very short loop that performs the same task on a range of memory locations / variables.

Example: Data Buffer

```
//Point to RAM buffer starting address
char *bufPtr = &buffer;

while ((DataAvailable) && (*bufPtr != '/0'))
{
    //Read byte from UART and write it to RAM buffer
    ReadUART(bufPtr);
    //Point to next available byte in RAM buffer
    bufPtr++;
}
```


Pointers

Why would I want to do that?

Example: Data Buffer

RAM buffer allocated over a range of addresses (perhaps an array)

Pseudo-code:

(1) Point arrow to first address of buffer
(2) Write data from UART to location pointed to by arrow
(3) Move arrow to point to next address in buffer
(4) Repeat until data from UART is 0, or buffer is full (arrow points to last address of buffer)

16-bit Data Memory (RAM)

16-bit Data Memory (RAM)	Address
0123	0x08BA
4567	0x08BC
89AB	0x08BE
CDEF	0x08C0
1357	0x08C2
9BDF	0x08C4
0246	0x08C6
8ACE	0x08C8

Pointers

Where else are they used?

- Used in conjunction with dynamic memory allocation (creating variables at runtime)
- Provide method to pass arguments *by reference* to functions
- Provide method to pass more than one piece of information into and out of a function
- A more efficient means of accessing arrays and dealing with strings

Pointers

How to Create a Pointer Variable

Syntax

```
type *ptrName;
```

- In the context of a declaration, the `*` merely indicates that the variable is a pointer
- *`type`* is the type of data the pointer may point to
- Pointer usually described as "*a pointer to* ***`type`***"

Example

```
int *iPtr;          // Create a pointer to int

float *fPtr;        // Create a pointer to float
```

Pointers

How to Create a Pointer Type with `typedef`

Syntax

```
typedef type *typeName;
```

- A pointer variable can now be declared as type *`typeName`* which is a synonym for *`type`*
- The `*` is no longer needed since *`typeName`* explicitly identifies the variable as a pointer to *`type`*

Example

```
typedef int *intPtr;   // Create pointer to int type

intPtr p;              // Create pointer to int
                       // Equivalent to: int *p;
```

No * is used

Pointers
Initialization

- To set a pointer to point to another variable, we use the `&` operator (address of), and the pointer variable is used <u>without</u> the dereference operator `*`:

```
p = &x;
```

- This assigns the address of the variable `x` to the pointer `p` (`p` now points to `x`)
- Note: `p` must be declared to point to the type of `x` (e.g. `int x; int *p;`)

Pointers

Usage

- When accessing the variable pointed to by a pointer, we use the pointer <u>with</u> the dereference operator `*`:

```
y = *p;
```

- This assigns to the variable `y`, the value of what `p` is pointing to (`x` from the last slide)
- Using `*p`, is the same as using the variable it points to (e.g. `x`)

Pointers

Another Way To Look At The Syntax

Example

```
int x, *p;          //int and a pointer to int

p = &x;             //Assign p the address of x
*p = 5;             //Same as x = 5;
```

- `&x` is a constant pointer
  - It represents the address of `x`
  - The address of `x` will never change
- `p` is a variable pointer to int
  - It can be assigned the address of any int
  - It may be assigned a new address any time

Pointers

Another Way To Look At The Syntax

Example

```
int x, *p;          //1 int, 1 pointer to int

p = &x;             //Assign p the address of x
*p = 5;             //Same as x = 5;
```

- `*p` represents the data pointed to by `p`
  - `*p` may be used anywhere you would use `x`
  - `*` is the dereference operator, also called the indirection operator
  - In the pointer declaration, the only significance of `*` is to indicate that the variable is a pointer rather than an ordinary variable

Pointers

How Pointers Work

Example

```c
{
  int x, y;
  int *p;

  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;

  *p = 0x0100;
  p = &y;
  *p = 0x0200;
}
```

16-bit Data Memory (RAM)

Variable at Address	16-bit Data Memory (RAM)	Address
	0000	0x08BA
x	0000	0x08BC
y	0000	0x08BE
p	0000	0x08C0
	0000	0x08C2
	0000	0x08C4
	0000	0x08C6
	0000	0x08C8

Pointers

How Pointers Work

Example

```
{
  int x, y;
  int *p;

  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;

  *p = 0x0100;
  p = &y;
  *p = 0x0200;
}
```

16-bit Data Memory (RAM)

Variable at Address	16-bit Data Memory (RAM)	Address
	0000	0x08BA
x	DEAD	0x08BC
y	0000	0x08BE
p	0000	0x08C0
	0000	0x08C2
	0000	0x08C4
	0000	0x08C6
	0000	0x08C8

Pointers

How Pointers Work

Example

```
{
  int x, y;
  int *p;

  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;

  *p = 0x0100;
  p = &y;
  *p = 0x0200;
}
```

16-bit Data Memory (RAM)

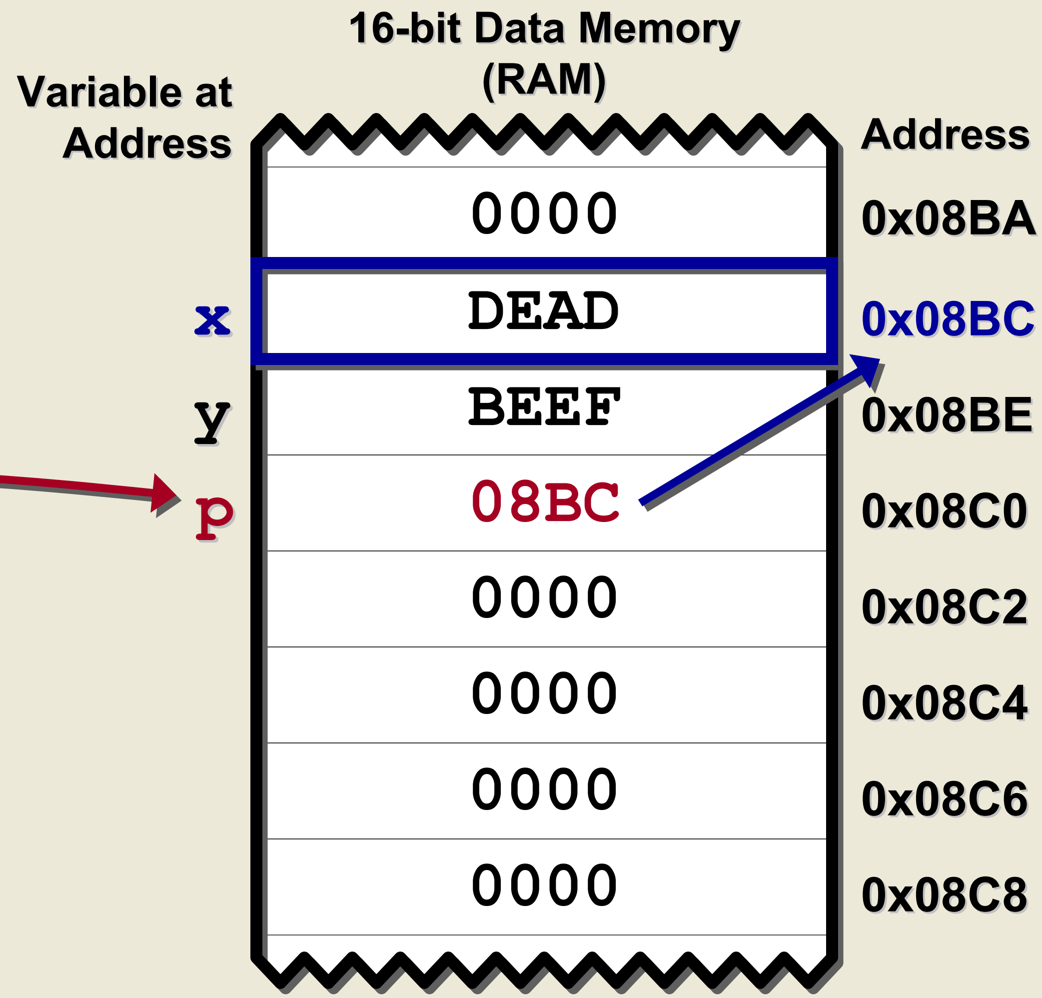
Variable at Address	16-bit Data Memory (RAM)	Address
	0000	0x08BA
x	DEAD	0x08BC
y	BEEF	0x08BE
p	0000	0x08C0
	0000	0x08C2
	0000	0x08C4
	0000	0x08C6
	0000	0x08C8

Pointers

How Pointers Work

Example

```
{
  int x, y;
  int *p;

  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;

  *p = 0x0100;
  p = &y;
  *p = 0x0200;
}
```

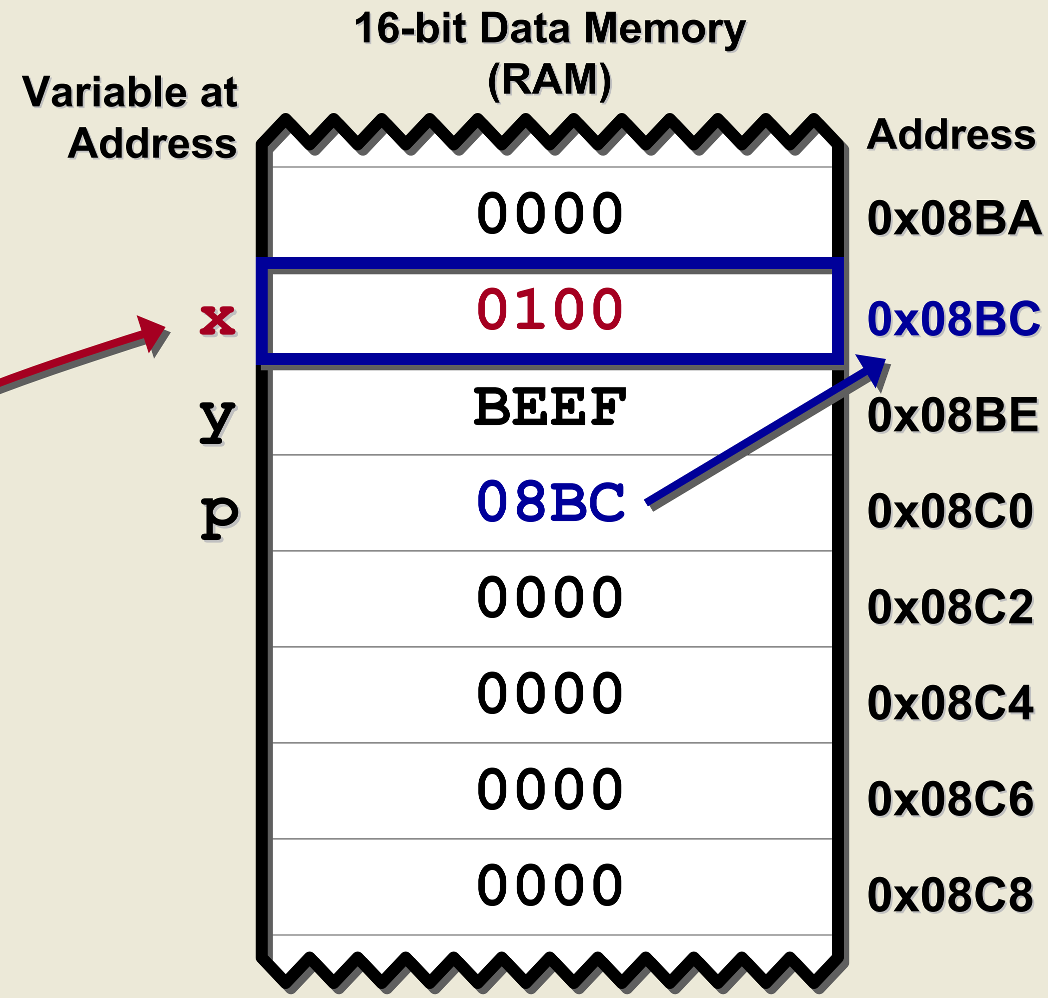


Pointers

How Pointers Work

Example

```
{
  int x, y;
  int *p;

  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;

  *p = 0x0100;
  p = &y;
  *p = 0x0200;
}
```



Pointers

How Pointers Work

Example

```
{
  int x, y;
  int *p;

  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;

  *p = 0x0100;
  p = &y;
  *p = 0x0200;
}
```

16-bit Data Memory (RAM)

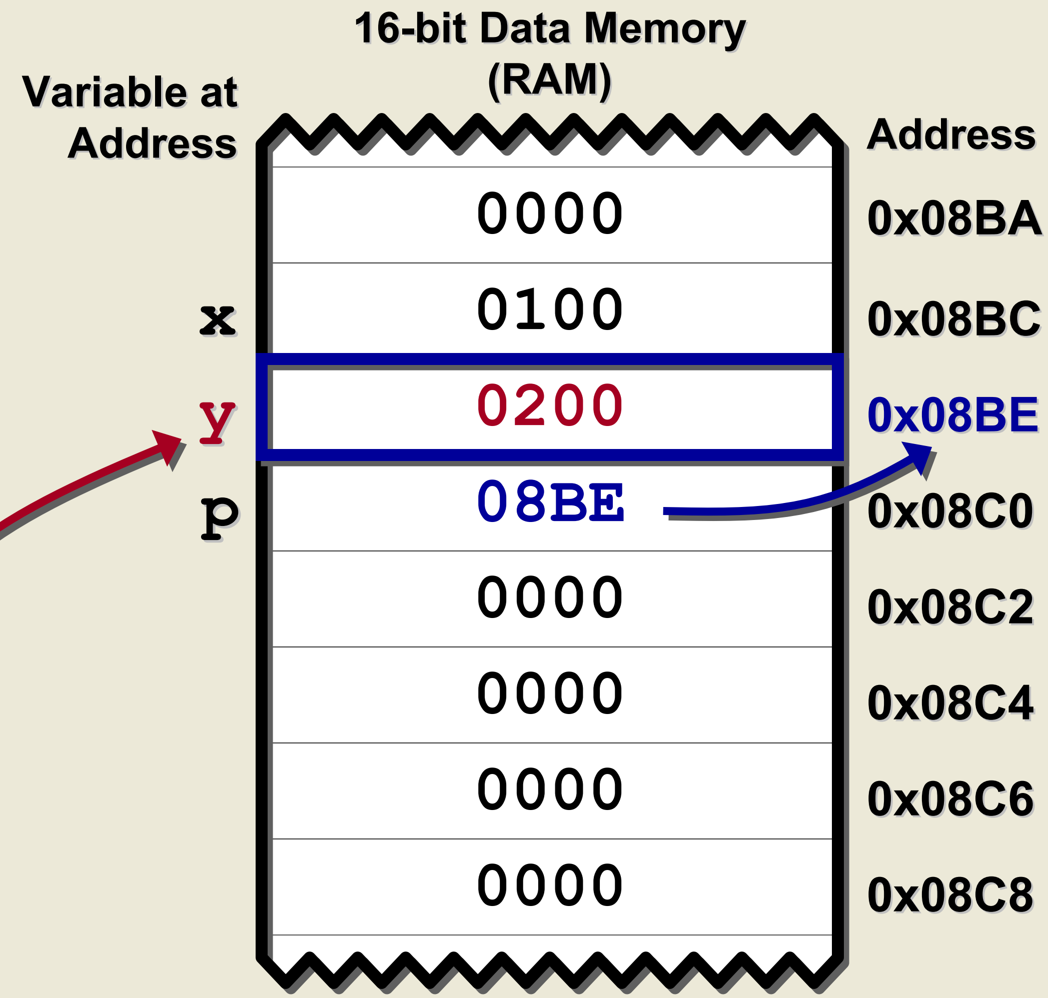
Variable at Address	16-bit Data Memory (RAM)	Address
	0000	0x08BA
x	0100	0x08BC
y	BEEF	0x08BE
p	08BE	0x08C0
	0000	0x08C2
	0000	0x08C4
	0000	0x08C6
	0000	0x08C8

Pointers

How Pointers Work

Example

```
{
  int x, y;
  int *p;

  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;

  *p = 0x0100;
  p = &y;
  *p = 0x0200;
}
```

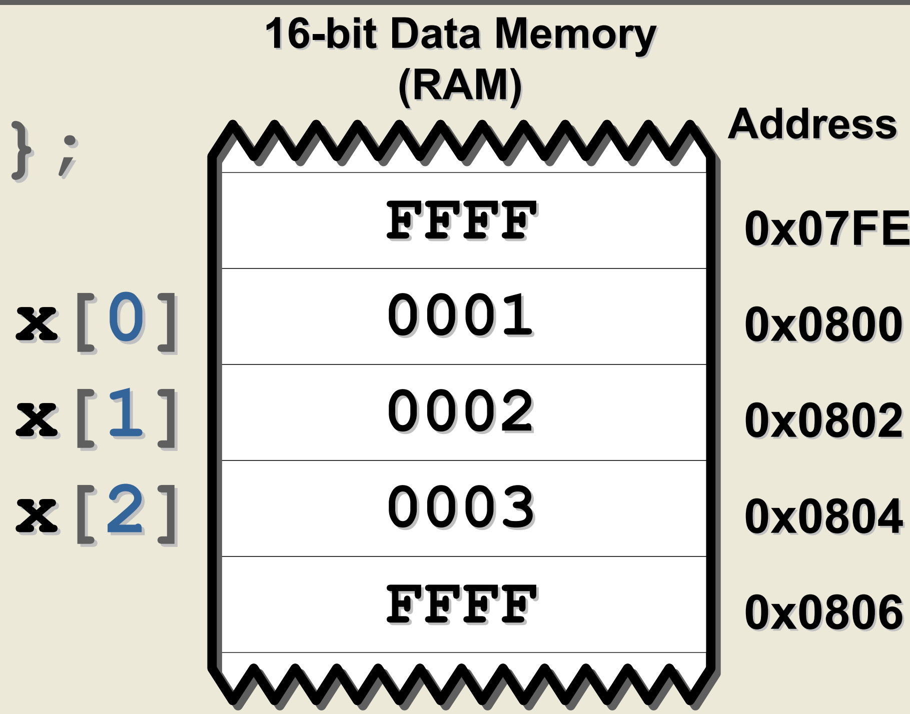


Pointers and Arrays

A Quick Reminder…

- Array elements occupy consecutive memory locations

```
int x[3] = {1,2,3};
```



- Pointers can provide an alternate method for accessing array elements

Pointers and Arrays

Initializing a Pointer to an Array

- The array name is the same thing as the address of its first (0^{th}) element

If we declare the following array and pointer variable:

```
int x[5] = {1,2,3,4,5};
int *p;
```

We can initialize the pointer to point to the array using any one of these three methods:

```
p = x;          //Works only for arrays!
p = &x;         //Works for arrays or variables
p = &x[0];      //This one is the most obvious
```

Pointers and Arrays

A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

```
int x[3] = {1,2,3};
int *p;

p = &x;
p++;
```

16-bit Data Memory (RAM)

		Address
	FFFF	0x07FE
x[0]	0001	0x0800
x[1]	0002	0x0802
x[2]	0003	0x0804
p	FFFF	0x0806

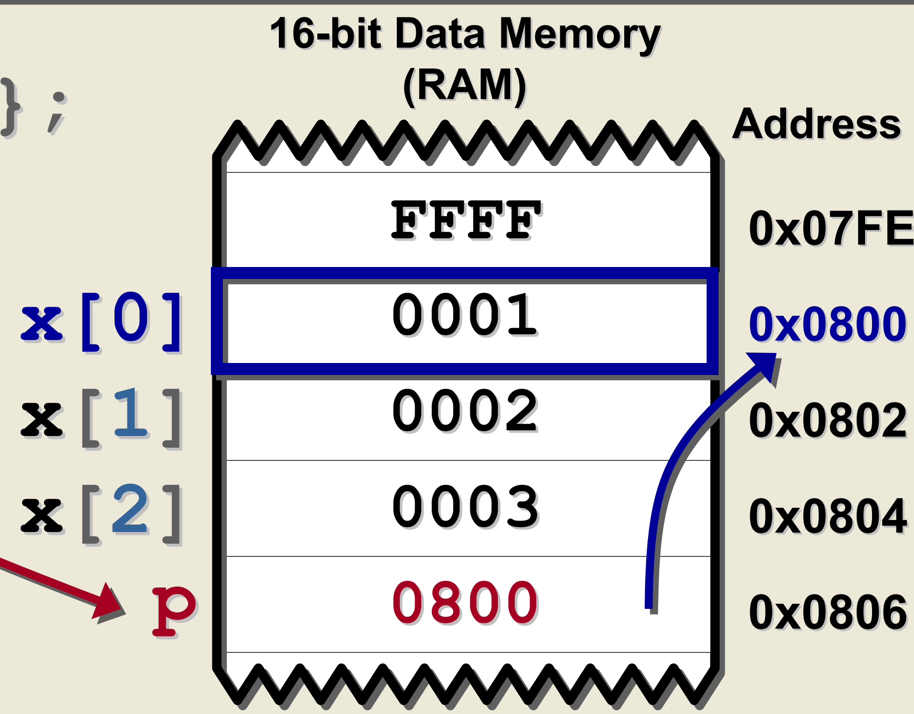
- More on this in just a bit…

Pointers and Arrays

A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

```
int x[3] = {1,2,3};
int *p;

p = &x;
p++;
```



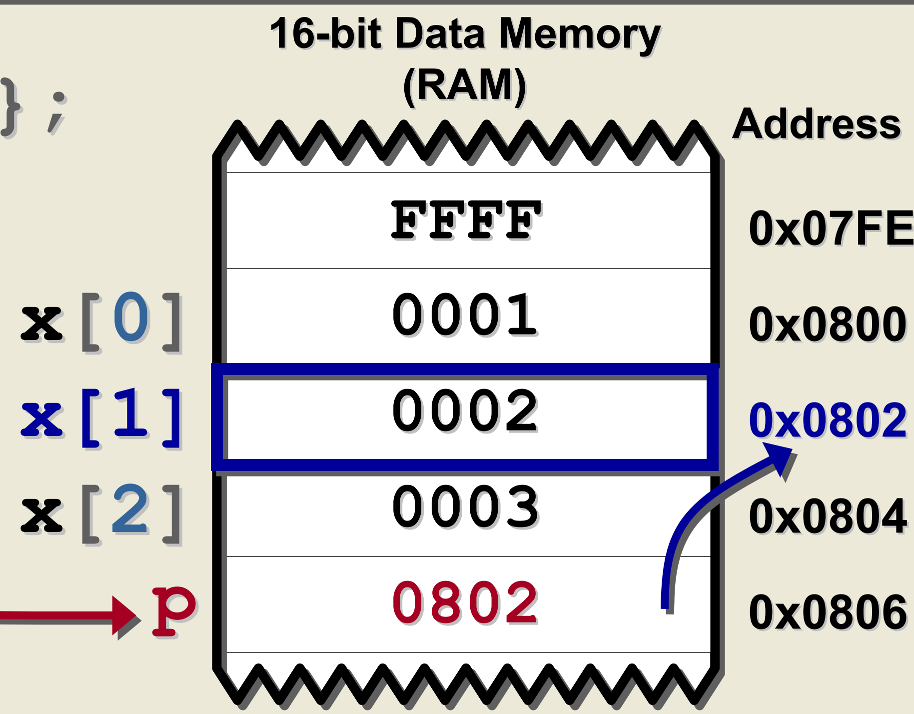
- More on this in just a bit…

Pointers and Arrays

A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

```
int x[3] = {1,2,3};
int *p;

p = &x;
p++;
```



- More on this in just a bit…

Pointer Arithmetic

Incrementing Pointers

- Incrementing or decrementing a pointer will add or subtract a multiple of the number of bytes of its type
- If we have:

```
float x;
float *p = &x;
p++;
```

We will get `p = &x + 4` since a `float` variable occupies 4 bytes of memory

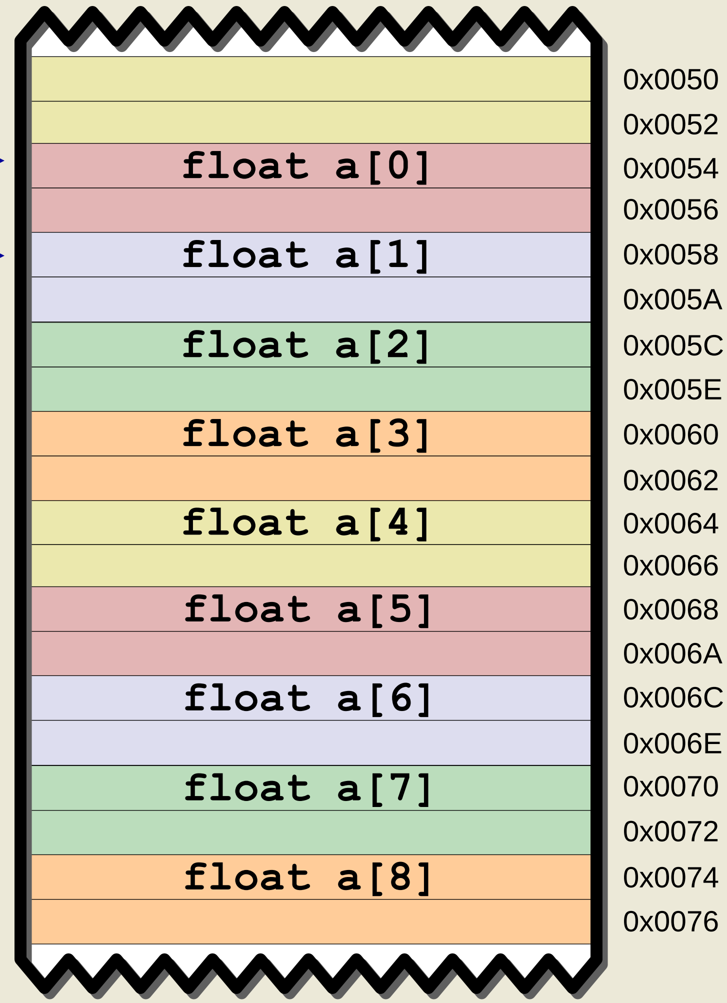
Pointer Arithmetic

Incrementing Pointers

Example

```
float *ptr;

          ptr = &a;

             ptr++;
```

Incrementing `ptr` moves it to the next sequential `float` array element



16-bit Data Memory Words

Pointer Arithmetic
Larger Jumps

- Adding or subtracting any other number with the pointer will change it by a multiple of the number of bytes of its type
- If we have

```
int x;
int *p = &x;
p += 3;
```

We will get `p = &x + 6` since an `int` variable occupies 2 bytes of memory

Pointer Arithmetic

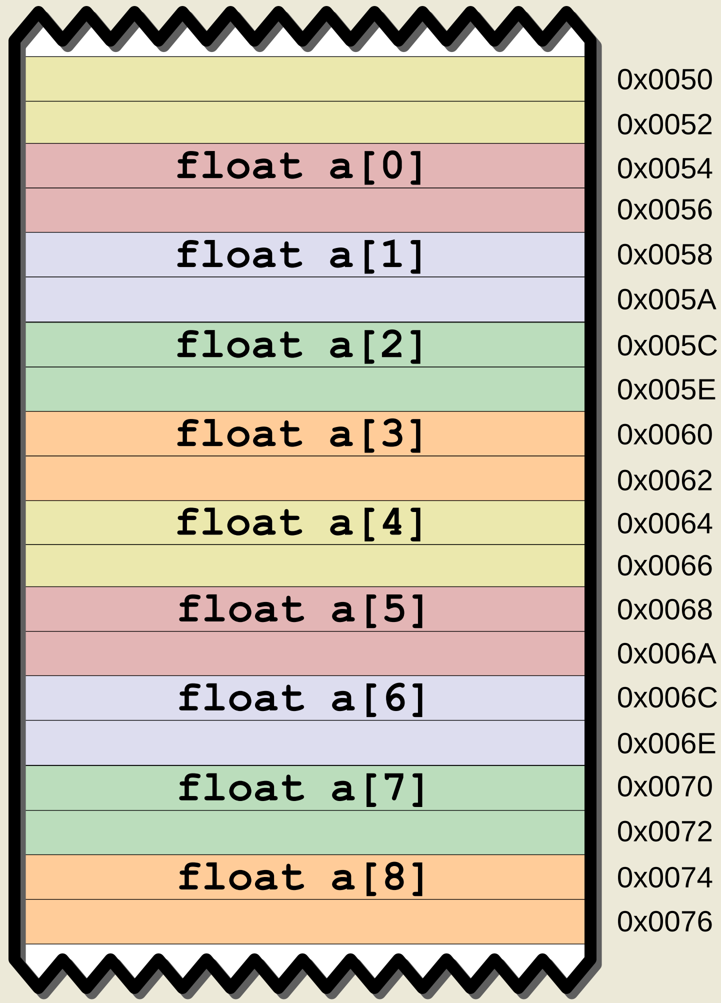
Larger Jumps

Example

```
float *ptr;

ptr = &a;
```

Adding 6 to `ptr` moves it 6 `float` array elements ahead (24 bytes ahead)

```
ptr += 6;
```



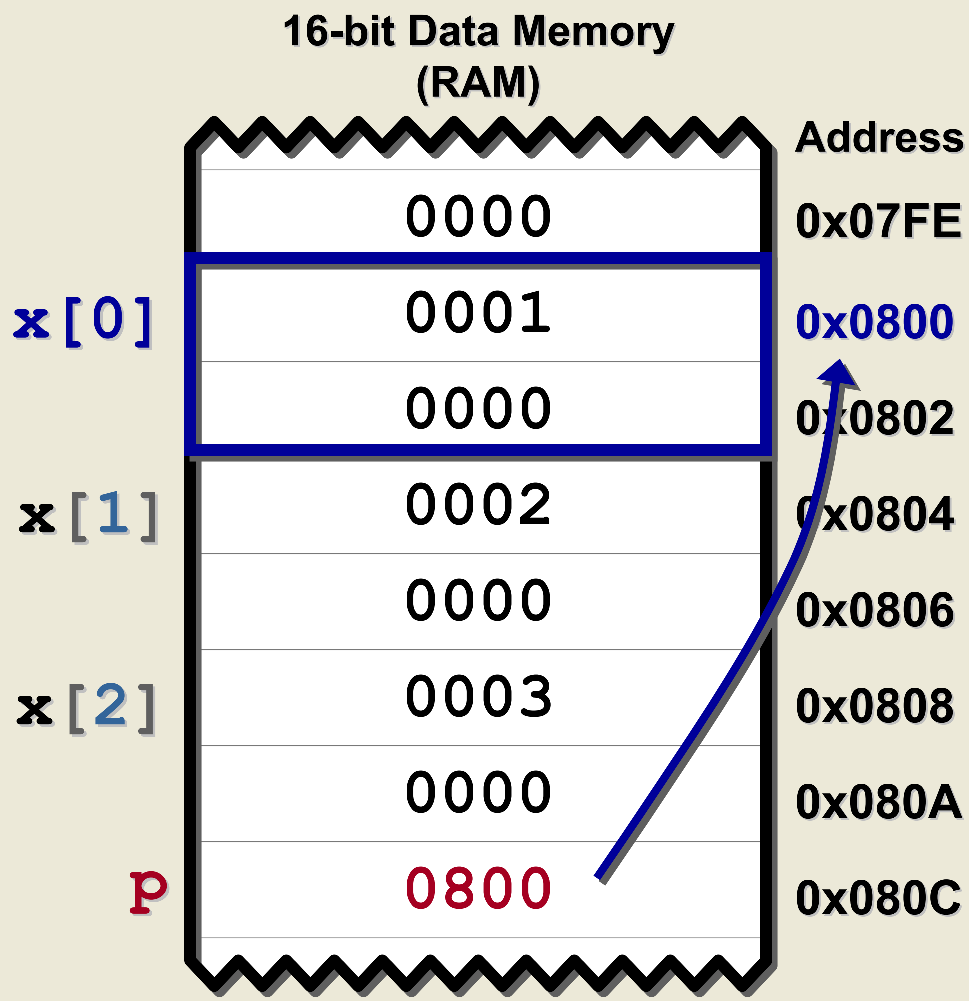
16-bit Data Memory Words

Pointers

Pointer Arithmetic

Example

```c
{
  long x[3] = {1,2,3};
  long *p = &x;

  *p += 4;
  p++;
  *p = 0xDEADBEEF;
  p++;
  *p = 0xF1D0F00D;
  p -= 2;
  *p = 0xBADF00D1;
}
```

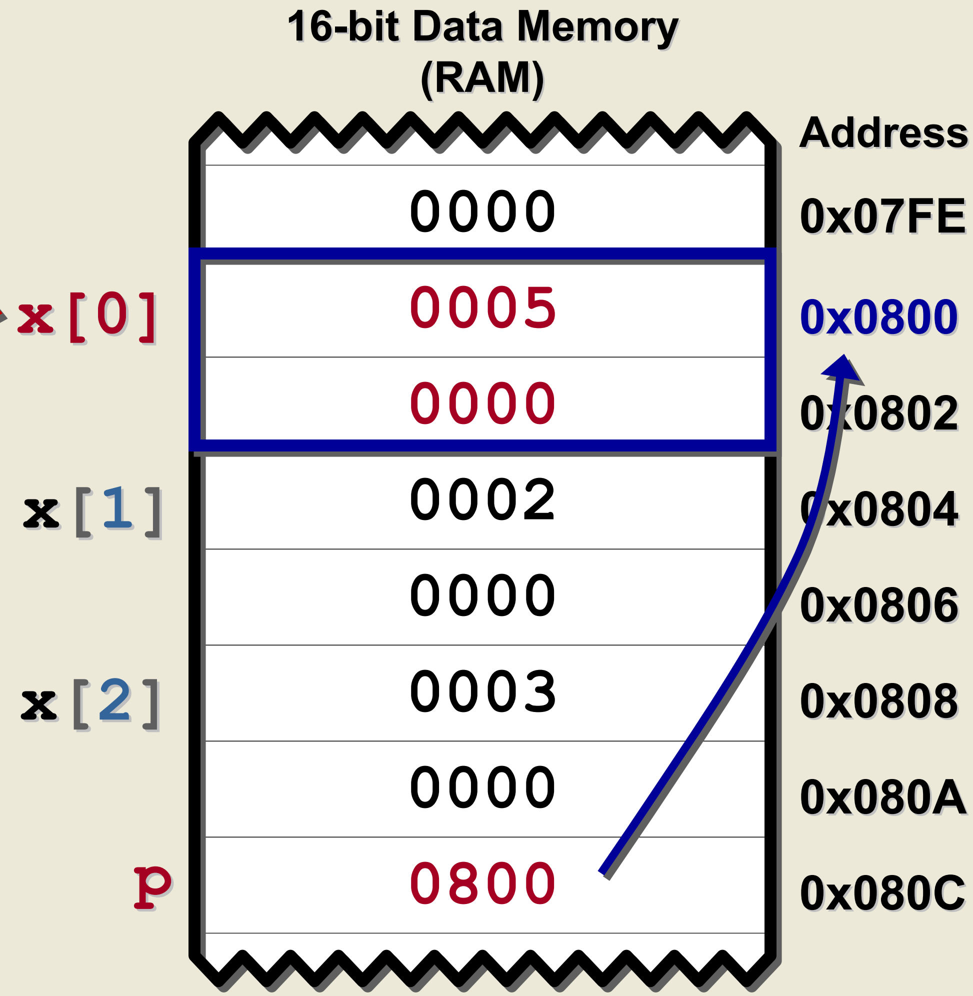


Pointers

Pointer Arithmetic

Example

```
{
  long x[3] = {1,2,3};
  long *p = &x;

  *p += 4;
  p++;
  *p = 0xDEADBEEF;
  p++;
  *p = 0xF1D0F00D;
  p -= 2;
  *p = 0xBADF00D1;
}
```

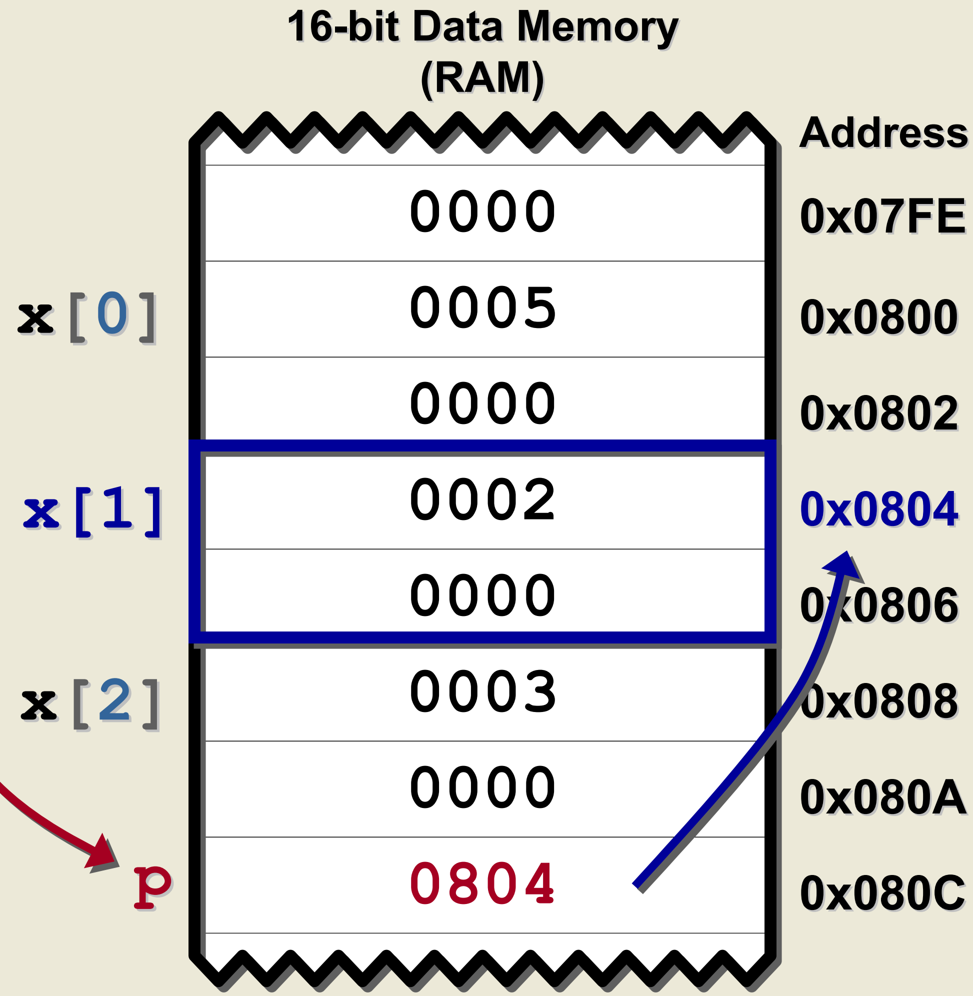


Pointers

Pointer Arithmetic

Example

```
{
  long x[3] = {1,2,3};
  long *p = &x;

  *p += 4;
  p++;
  *p = 0xDEADBEEF;
  p++;
  *p = 0xF1D0F00D;
  p -= 2;
  *p = 0xBADF00D1;
}
```

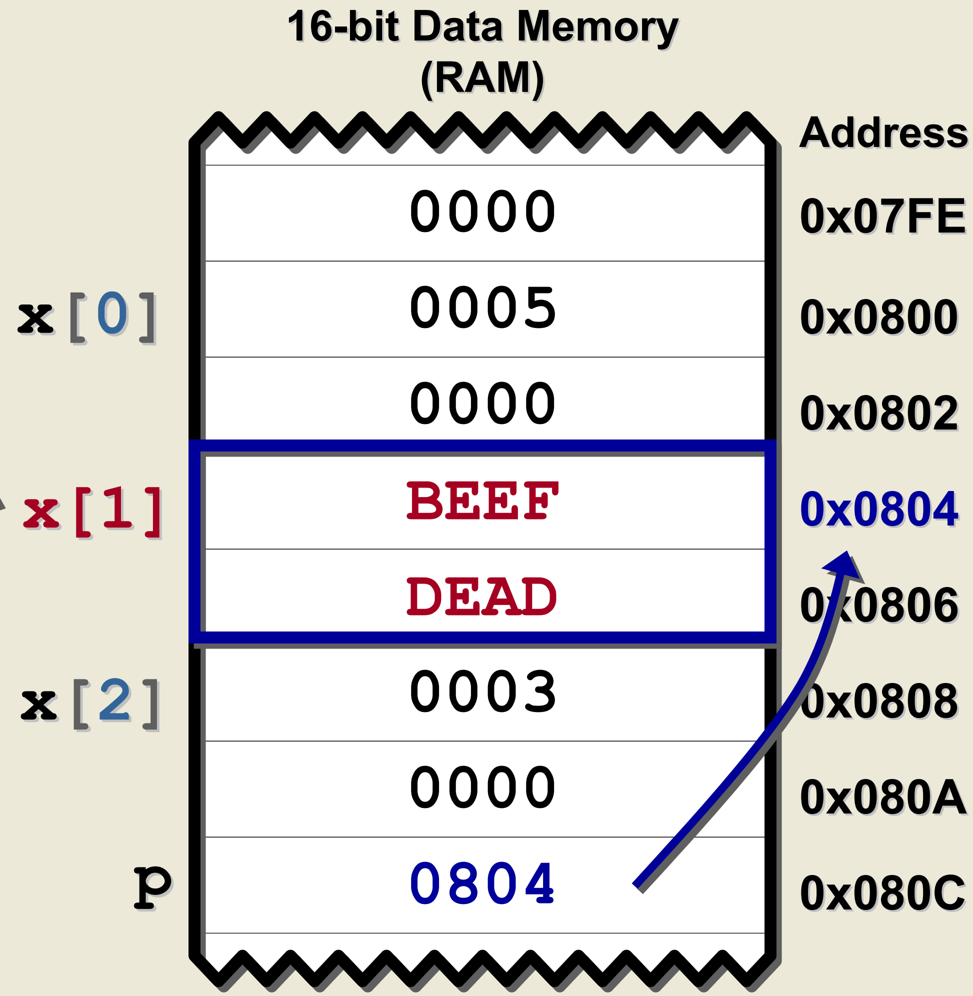


Pointers

Pointer Arithmetic

Example

```
{
  long x[3] = {1,2,3};
  long *p = &x;

  *p += 4;
  p++;
  *p = 0xDEADBEEF;
  p++;
  *p = 0xF1D0F00D;
  p -= 2;
  *p = 0xBADF00D1;
}
```

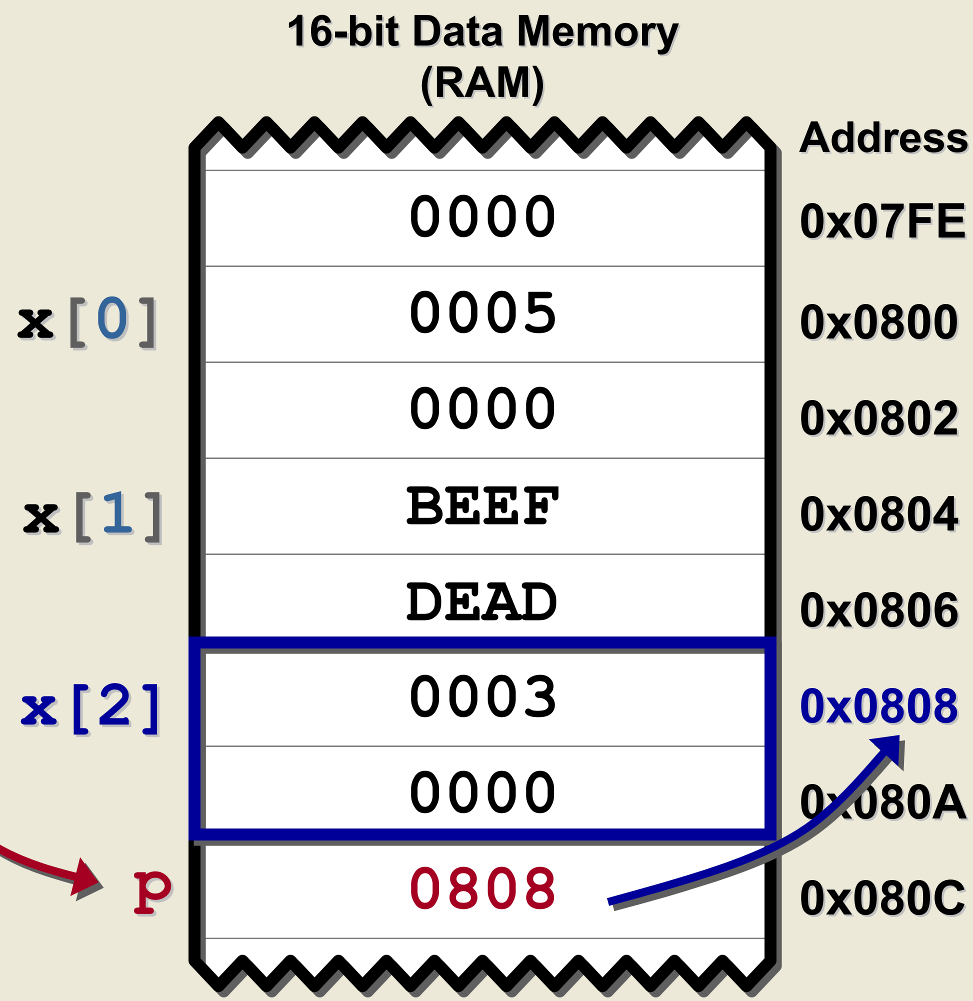


Pointers

Pointer Arithmetic

Example

```
{
  long x[3] = {1,2,3};
  long *p = &x;

  *p += 4;
  p++;
  *p = 0xDEADBEEF;
  p++;
  *p = 0xF1D0F00D;
  p -= 2;
  *p = 0xBADF00D1;
}
```

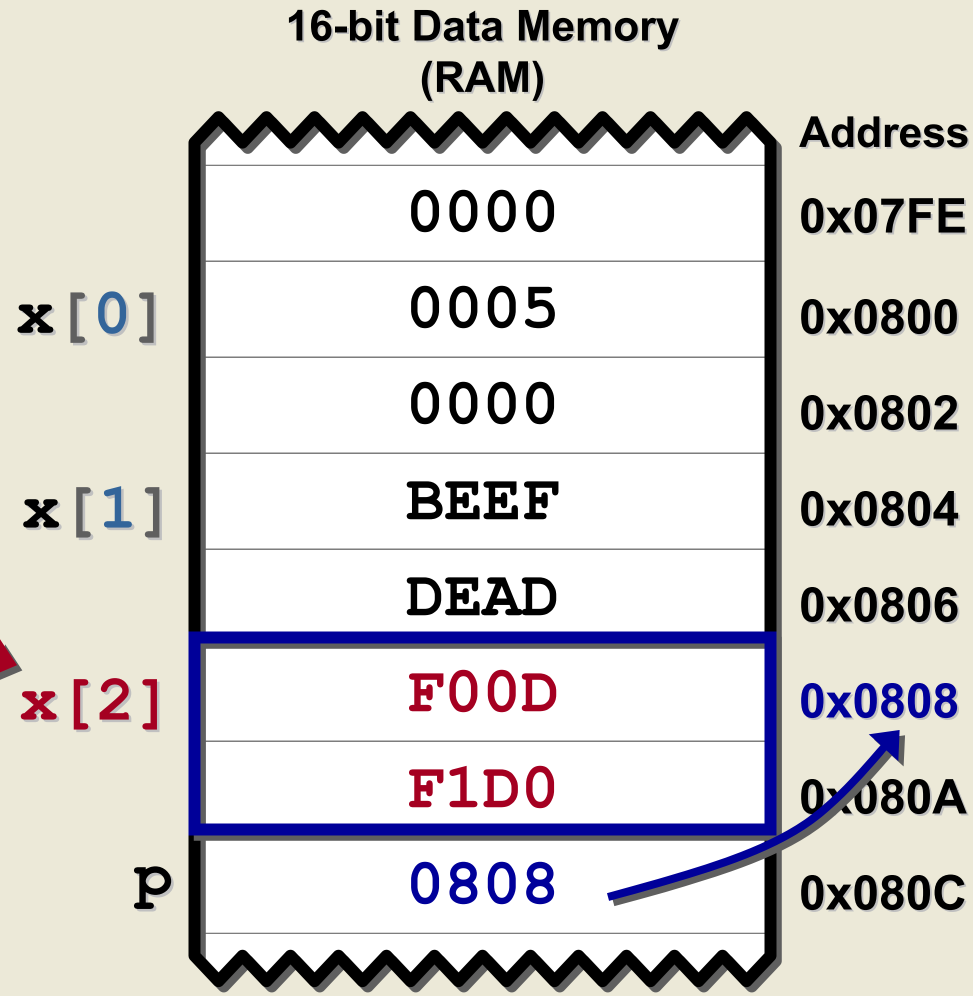


Pointers

Pointer Arithmetic

Example

```
{
  long x[3] = {1,2,3};
  long *p = &x;

  *p += 4;
  p++;
  *p = 0xDEADBEEF;
  p++;
  *p = 0xF1D0F00D;
  p -= 2;
  *p = 0xBADF00D1;
}
```

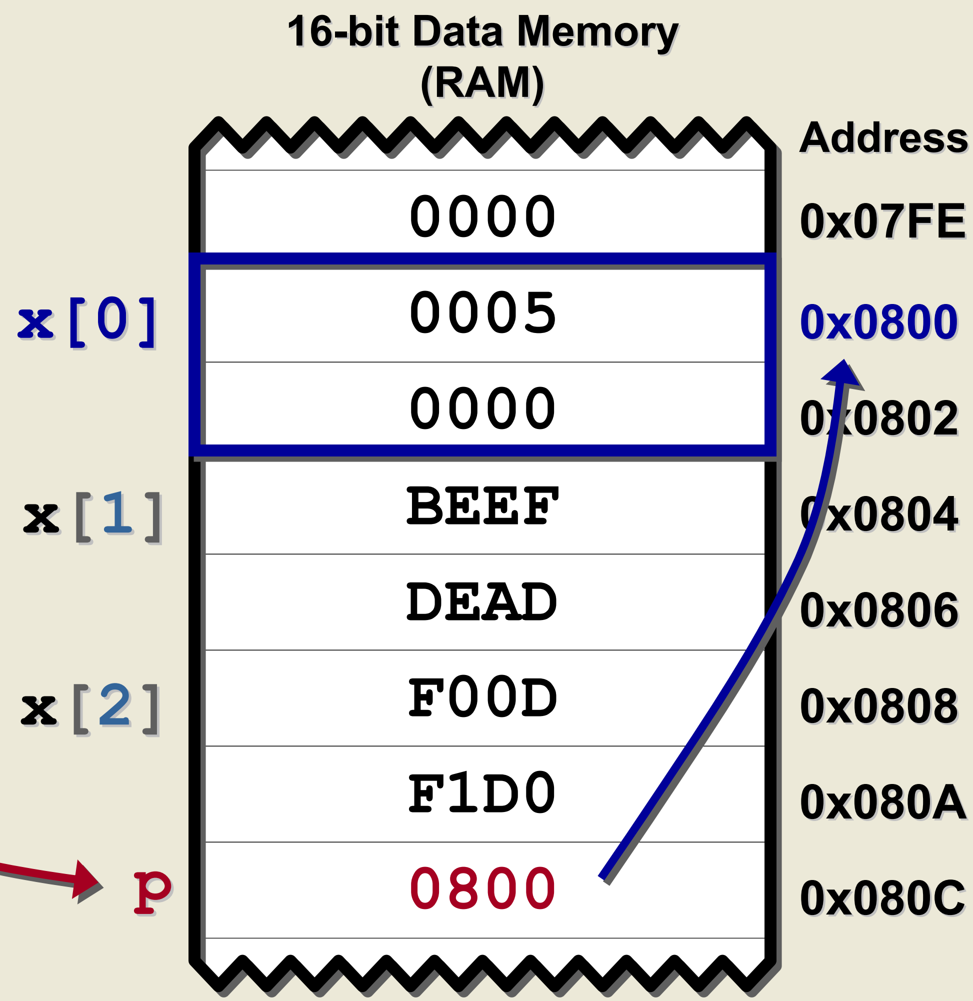


Pointers

Pointer Arithmetic

Example

```
{
  long x[3] = {1,2,3};
  long *p = &x;

  *p += 4;
  p++;
  *p = 0xDEADBEEF;
  p++;
  *p = 0xF1D0F00D;
  p -= 2;
  *p = 0xBADF00D1;
}
```

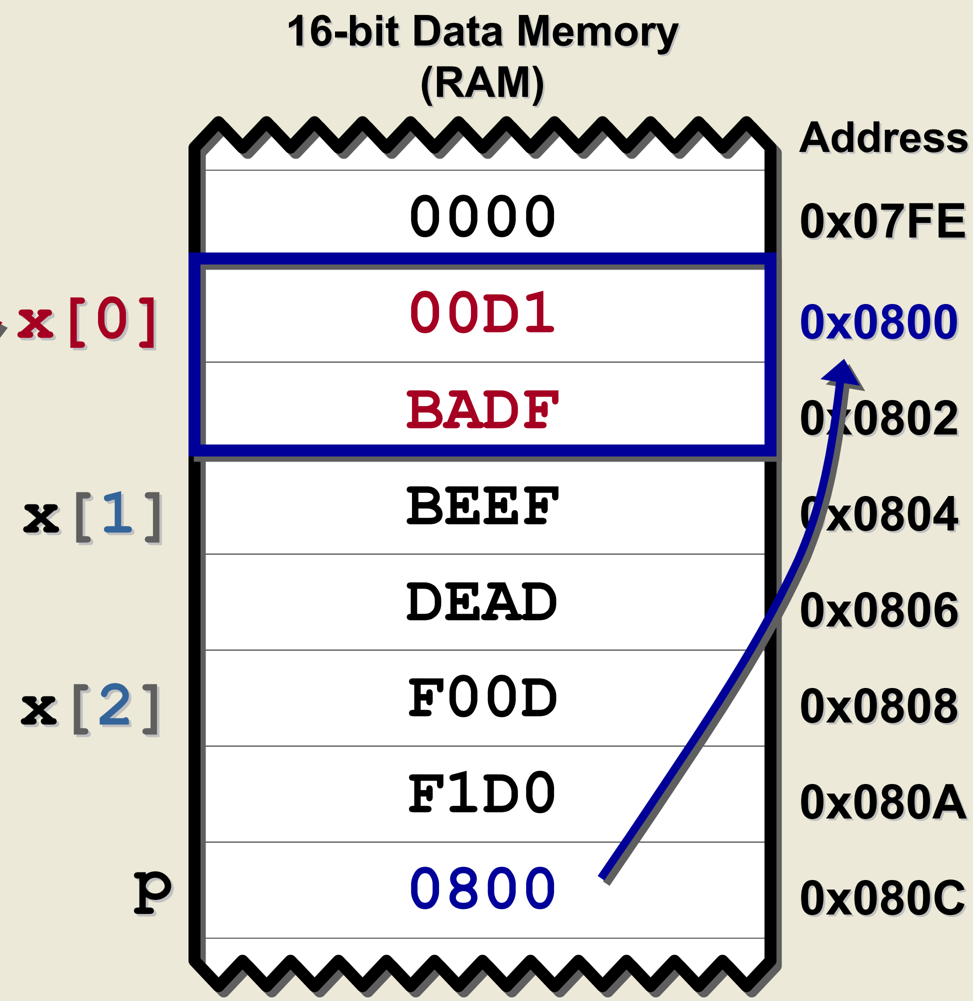


Pointers

Pointer Arithmetic

Example

```
{
  long x[3] = {1,2,3};
  long *p = &x;

  *p += 4;
  p++;
  *p = 0xDEADBEEF;
  p++;
  *p = 0xF1D0F00D;
  p -= 2;
  *p = 0xBADF00D1;
}
```



Pointers

Post-Increment/Decrement Syntax Rule

- **Care must be taken with respect to operator precedence when doing pointer arithmetic:**

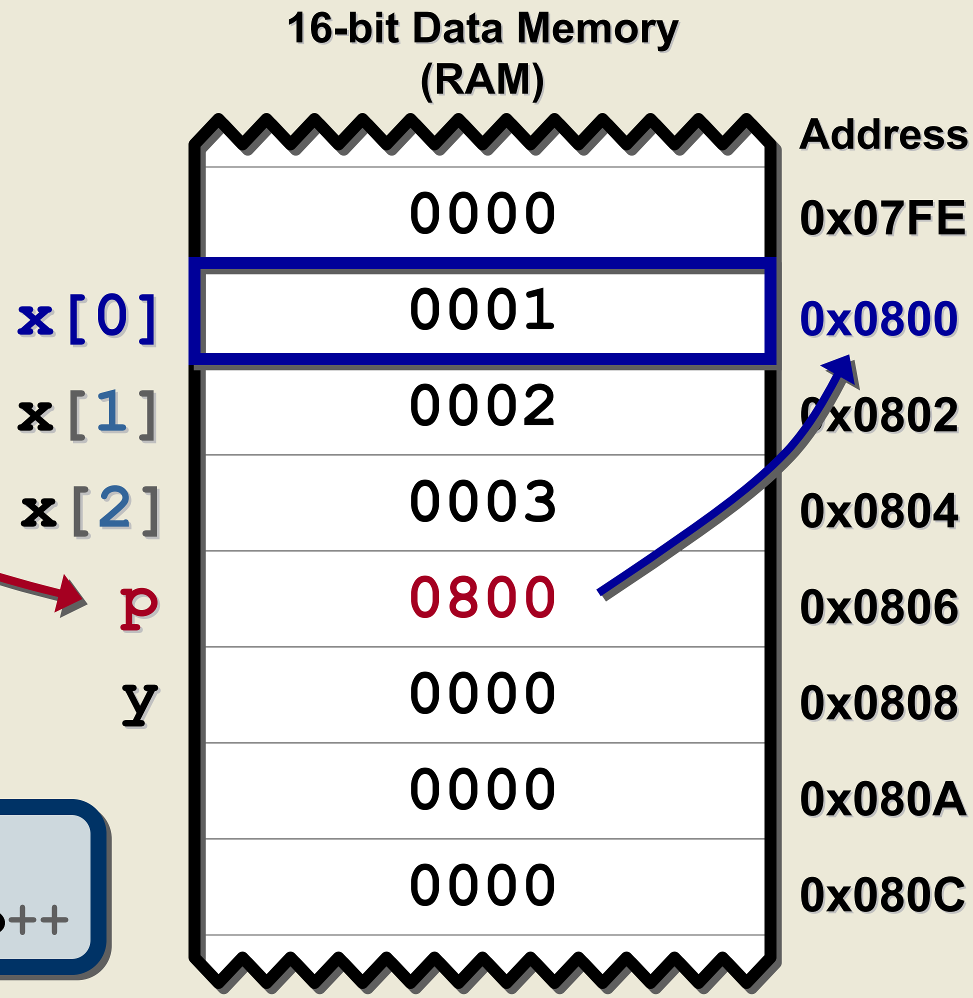
Syntax	Operation	Description by Example
`*p++` `*(p++)`	Post-Increment Pointer	`z = *(p++);` is equivalent to: `z = *p;` `p = p + 1;`
`(*p)++`	Post-Increment data pointed to by Pointer	`z = (*p)++;` is equivalent to: `z = *p;` `*p = *p + 1;`

Pointers

Post-Increment / Decrement Syntax

Example

```
{
  int x[3] = {1,2,3};
  int y;
  int *p = &x;

  y = 5 + *(p++);

  y = 5 + (*p)++;
}
```



Remember:
`*(p++)` is the same as `*p++`

Pointers

Post-Increment / Decrement Syntax

Example

```
{
  int x[3] = {1,2,3};
  int y;
  int *p = &x;

  y = 5 + *(p++);

  y = 5 + (*p)++;
}
```

16-bit Data Memory (RAM)

		Address
	0000	0x07FE
x[0]	0001	0x0800
x[1]	0002	0x0802
x[2]	0003	0x0804
p	0800	0x0806
y	0006	0x0808
	0000	0x080A
	0000	0x080C

Remember:
`*(p++)` is the same as `*p++`

Pointers

Post-Increment / Decrement Syntax

Example

```
{
  int x[3] = {1,2,3};
  int y;
  int *p = &x;

  y = 5 + *(p++);

  y = 5 + (*p)++;
}
```

16-bit Data Memory (RAM)

		Address
	0000	0x07FE
x[0]	0001	0x0800
x[1]	0002	0x0802
x[2]	0003	0x0804
p	0802	0x0806
y	0006	0x0808
	0000	0x080A
	0000	0x080C

Remember:
`*(p++)` is the same as `*p++`

Pointers

Post-Increment / Decrement Syntax

Example

```
{
  int x[3] = {1,2,3};
  int y;
  int *p = &x;

  y = 5 + *(p++);

  y = 5 + (*p)++;
}
```

16-bit Data Memory (RAM)

		Address
	0000	0x07FE
x[0]	0001	0x0800
x[1]	0002	0x0802
x[2]	0003	0x0804
p	0802	0x0806
y	0007	0x0808
	0000	0x080A
	0000	0x080C

Remember:
`*(p++)` is the same as `*p++`

Pointers

Post-Increment / Decrement Syntax

Example

```
{
  int x[3] = {1,2,3};
  int y;
  int *p = &x;

  y = 5 + *(p++);

  y = 5 + (*p)++;
}
```

16-bit Data Memory (RAM)

	Value	Address
	0000	0x07FE
x[0]	0001	0x0800
x[1]	0003	0x0802
x[2]	0003	0x0804
p	0802	0x0806
y	0007	0x0808
	0000	0x080A
	0000	0x080C

Remember:
`*(p++)` is the same as `*p++`

Pointers

Pre-Increment/Decrement Syntax Rule

- **Care must be taken with respect to operator precedence when doing pointer arithmetic:**

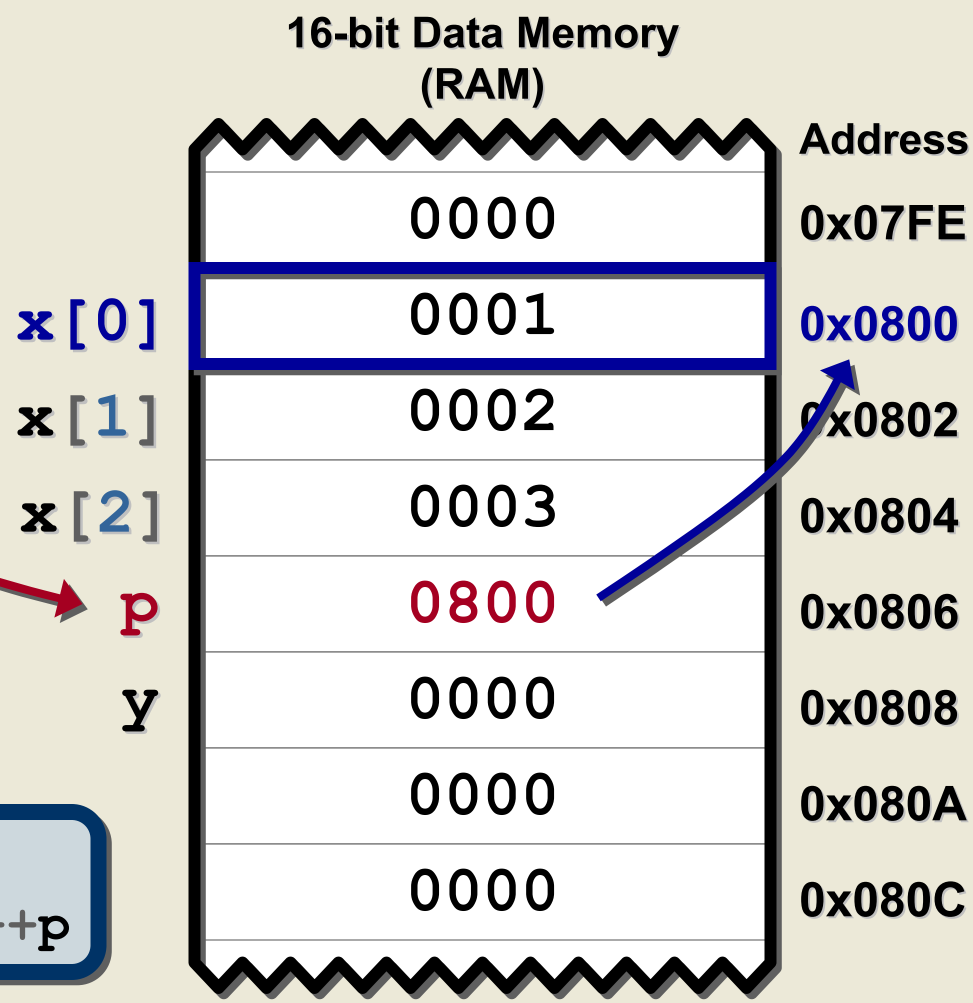
Syntax	Operation	Description by Example
`++*p` `*(++p)`	Pre-Increment Pointer	`z = *(++p);` is equivalent to: `p = p + 1;` `z = *p;`
`++(*p)`	Pre-Increment data pointed to by Pointer	`z = ++(*p);` is equivalent to: `*p = *p + 1;` `z = *p;`

Pointers

Pre-Increment / Decrement Syntax

Example

```
{
  int x[3] = {1,2,3};
  int y;
  int *p = &x;

  y = 5 + *(++p);

  y = 5 + ++(*p);
}
```



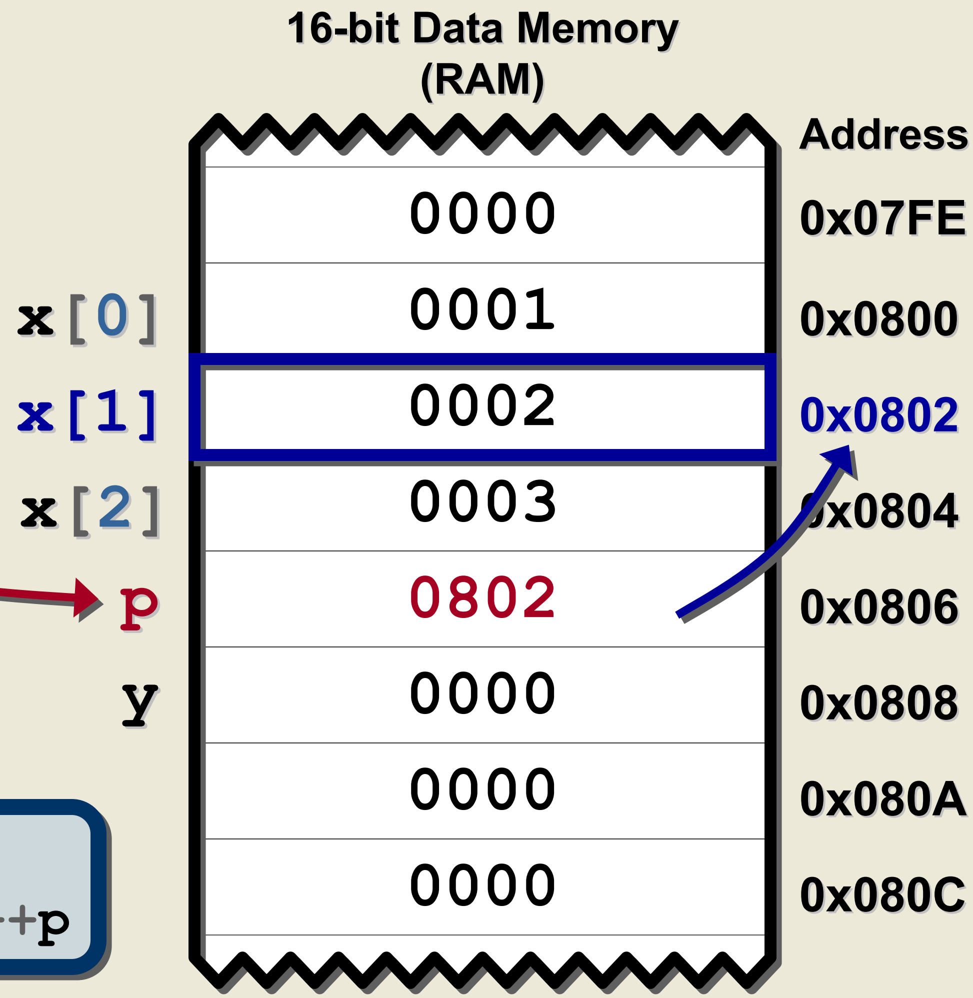
Remember:
`*(++p)` is the same as `*++p`

Pointers

Pre-Increment / Decrement Syntax

Example

```
{
  int x[3] = {1,2,3};
  int y;
  int *p = &x;

  y = 5 + *(++p);

  y = 5 + ++(*p);
}
```



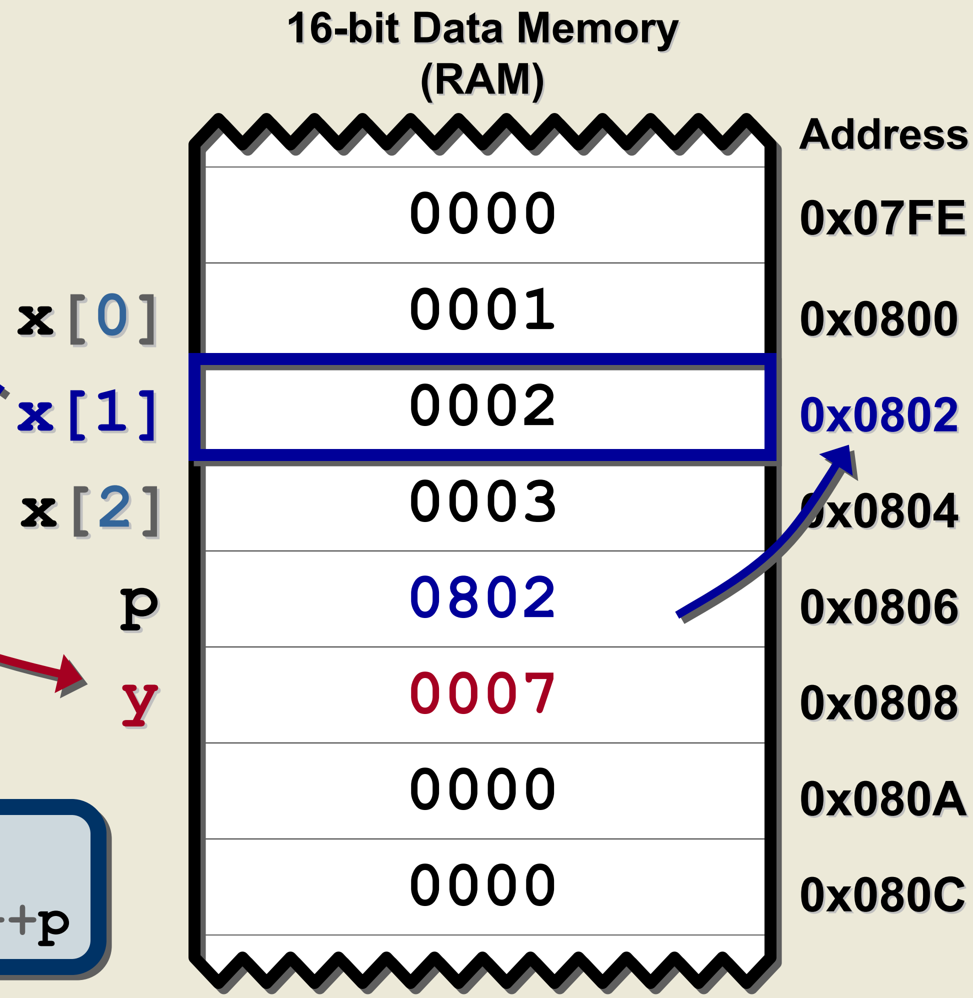
Remember:
`*(++p)` is the same as `*++p`

Pointers

Pre-Increment / Decrement Syntax

Example

```
{
  int x[3] = {1,2,3};
  int y;
  int *p = &x;

  y = 5 + *(++p);

  y = 5 + ++(*p);
}
```



Remember:
`*(++p)` **is the same as** `*++p`

Pointers

Pre-Increment / Decrement Syntax

Example

```
{
  int x[3] = {1,2,3};
  int y;
  int *p = &x;

  y = 5 + *(++p);

  y = 5 + ++(*p);
}
```

16-bit Data Memory (RAM)

		Address
	0000	0x07FE
x[0]	0001	0x0800
x[1]	0003	0x0802
x[2]	0003	0x0804
p	0802	0x0806
y	0007	0x0808
	0000	0x080A
	0000	0x080C

Remember:
`*(++p)` is the same as `*++p`

Pointers

Pre-Increment / Decrement Syntax

Example

```
{
  int x[3] = {1,2,3};
  int y;
  int *p = &x;

  y = 5 + *(++p);

  y = 5 + ++(*p);
}
```

16-bit Data Memory (RAM)

		Address
	0000	0x07FE
x[0]	0001	0x0800
x[1]	0003	0x0802
x[2]	0003	0x0804
p	0802	0x0806
y	0008	0x0808
	0000	0x080A
	0000	0x080C

Remember:
`*(++p)` **is the same as** `*++p`

Pointers

Pre- and Post- Increment/Decrement Summary

- The parentheses determine what gets incremented/decremented:

Modify the pointer itself

`*(++p)` or `*++p` and `*(p++)` or `*p++`

Modify the value pointed to by the pointer

`++(*p)` and `(*p)++`

Pointers

Initialization Tip

- If a pointer isn't initialized to a specific address when it is created, it is a good idea to initialize it as NUL (pointing to nowhere)
- This will prevent it from unintentionally corrupting a memory location if it is accidentally used before it is initialized

Example

```
int *p = NUL;
```

NULL is the character **'\0'** but **NUL** is the value of a pointer that points to nowhere

Lab 11

Pointers and Pointer Arithmetic

On the CD

...\101_ECP\Lab11\Lab11.mcw

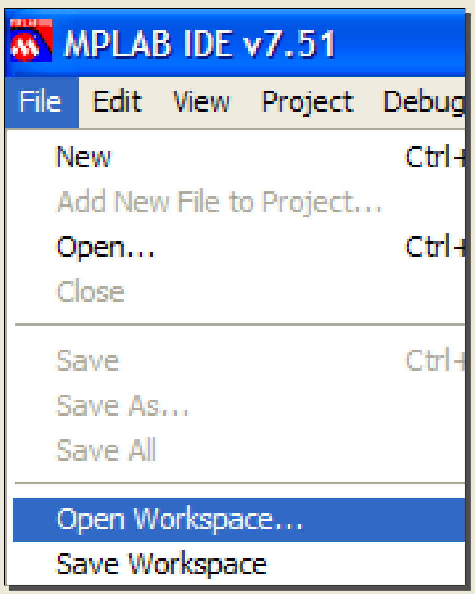
Lab 11

Pointers and Pointer Arithmetic

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab11\Lab11.mcw



1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 11

Pointers and Pointer Arithmetic

Solution: Steps 1, 2 and 3

```
/*###############################################################################
# STEP 1: Initialize the pointer p with the address of the variable x
###############################################################################*/
        //Point to address of x
        p = &x;

/*###############################################################################
# STEP 2: Complete the following printf() functions by adding in the
#         appropriate arguments as described in the control string.
###############################################################################*/
        printf("The variable x is located at address 0x%X\n", &x);
        printf("The value of x is %d\n", x);
        printf("The pointer p is located at address 0x%X\n", &p);
        printf("The value of p is 0x%X\n", p);
        printf("The value pointed to by *p = %d\n", *p);

/*###############################################################################
# STEP 3: Write the int value 10 to the location p is currently pointing to.
###############################################################################*/
        *p = 10;
```

Lab 11

Pointers and Pointer Arithmetic

Solution: Steps 4 and 5

```
/*###############################################################################
# STEP 4: Increment the value that p points to.
###############################################################################*/
        //Increment array element's value
        (*p)++;

        printf("y[%d] = %d\n", i, *p);
/*###############################################################################
# STEP 5: Increment the pointer p so that it points to the next item.
###############################################################################*/
        //Increment pointer to next array element
        p++;
```

Lab 11

Conclusions

- **Pointers are variables that hold the address of other variables**
- **Pointers make it possible for the program to change which variable is acted on by a particular line of code**
- **Incrementing and decrementing pointers will modify the value in multiples of the size of the type they point to**

Pointers and Functions

Passing Pointers to Functions

- Normally, functions operate on copies of the data passed to them (pass by value)

```
int x = 2, y = 0;

int square(int n)
{
      return (n * n);
}

int main(void)
{
      y = square(x);
}
```

Value of variable passed to function is copied into local variable n

After Function Call: y = 4
x = 2

x was not changed by function

Pointers and Functions

Passing Pointers to Functions

- **Pointers allow a function to operate on the original variable (pass by reference)**

```
int x = 2 , y = 0;

void square(int *n)
{
      *n *= *n;
}

int main(void)
{
      square(&x);
}
```

Address of variable passed to function and stored in local pointer variable n

After Function Call: x = 4
x was changed by function

Pointers and Functions

Passing Pointers to Functions

- A function with a pointer parameter:

Example

```
int foo(int *q)
```

- Must be called in one of two ways:
  (assume: `int x, *p = &x;`)

`foo(&x)`	Pass an address to the function so the address may be assigned to the pointer parameter: `q = &x`
`foo(p)`	Pass a pointer to the function so the address may be assigned to the pointer parameter: `q = p`

Pointers and Functions

Passing Parameters By Reference

Example – Part 1

We know where you live!

Swap function definition:

```
void swap(int *n1, int *n2)
{
    int temp;

    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

Addresses of parameters copied to local pointer variables: Function can now modify the original variables via pointers.

Pointers and Functions

Passing Parameters By Reference

Example – Part 2

Main function definition:

```
int main(void)
{
    int x = 5, y = 10;
    int *p = &y;

    swap(&x, p);

    while(1);
}
```

Swap function prototype:

```
void swap(int *n1, int *n2)
```

Tell function where x and y live...

```
n1 = &x
n2 = p
```

After running program:

x = 10

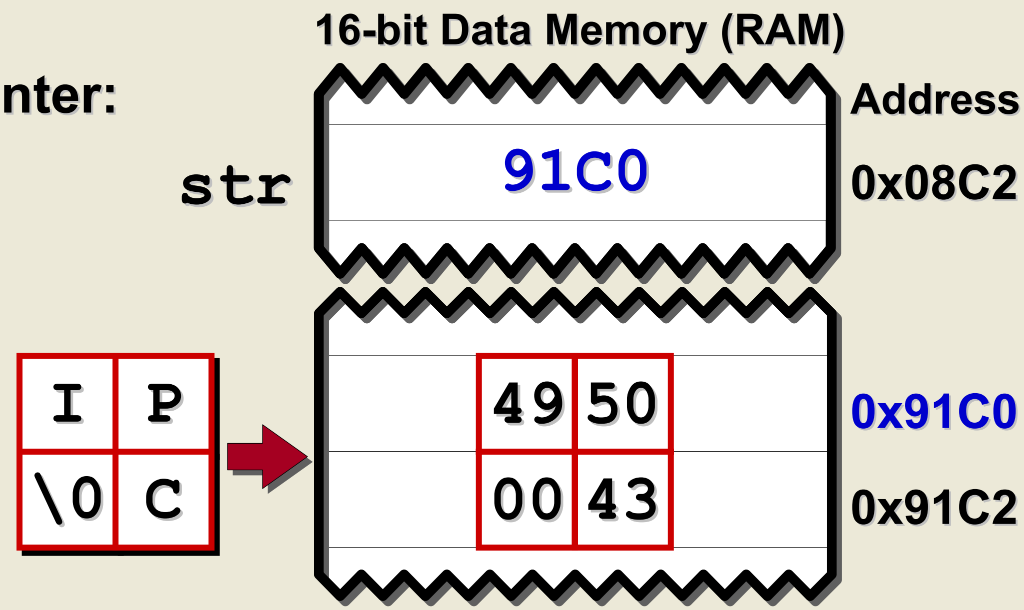
y = 5

Pointers and Strings

- So far, we have worked with strings strictly as arrays of `char`
- Strings may be created and used with pointers much more elegantly

String declaration with a pointer:

```
char *str = "PIC";
```

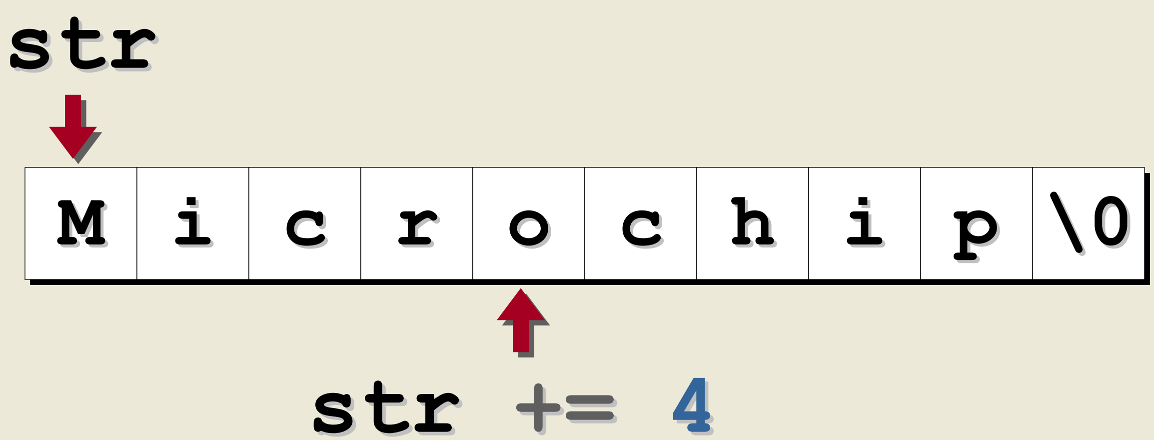


Implementation varies depending on compiler and architecture used.

Pointers and Strings

- When initialized, a pointer to a string points to the first character:

```
char *str = "Microchip";
```

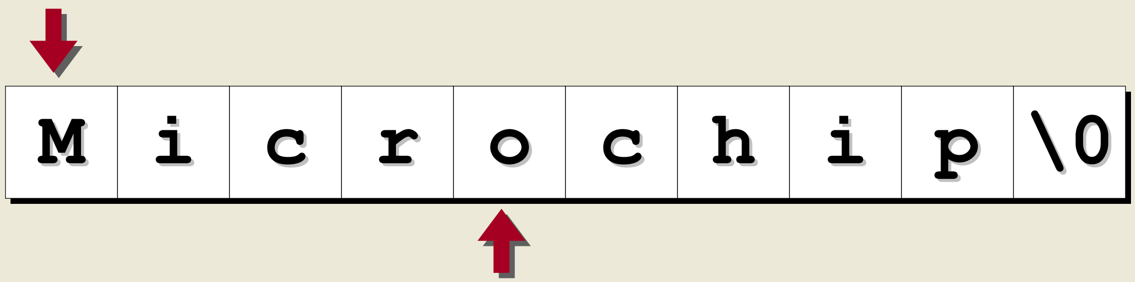


- Increment or add an offset to the pointer to access subsequent characters

Pointers and Strings

- Pointers may also be used to access characters via an offset:

```
char *str = "Microchip";
*str == 'M'
```




```
*(str + 4) == 'o'
```

- Pointer always points to "base address"
- Offsets used to access subsequent chars

Pointers and Strings

Pointer versus Array: Initialization at Declaration

- **Initializing a character string when it is declared is essentially the same for both a pointer and an array:**

Example: Pointer Variable

```
char *str = "PIC";
```

Example: Array Variable

```
char str[] = "PIC";
```

or

```
char str[4] = "PIC";
```

The NULL character '\0' is automatically appended to strings in both cases (array must be large enough).

Pointers and Strings

Pointer versus Array: Assignment in Code

- An entire string may be assigned to a pointer
- A character array must be assigned character by character

Example: Pointer Variable

```
char *str;

str = "PIC";
```

Example: Array Variable

```
char str[4];

str[0] = 'P';
str[1] = 'I';
str[2] = 'C';
str[3] = '\0';
```

Must explicitly add NULL character '\0' to array.

Pointers and Strings

Comparing Strings

- **If you want to test a string for equivalence, the natural thing to do is:**
  `if (str == "Microchip")`
- **This is not correct, though it might appear to work sometimes**
- **This compares the address in `str` to the address of the string literal `"Microchip"`**
- **The correct way is to use the `strcmp()` function in the standard library which compares strings character by character**

Pointers and Strings

Comparing Strings

- strcmp() prototype:

Function Prototype

```
int strcmp(const char *s1, const char *s2);
```

- strcmp() return values:
  - <0 if s1 is less than s2
  - 0 if s1 is equal to s2
  - >0 if s1 is greater than s2

The `strcmp()` prototype is in
`C:\Program Files\Microchip\MPLAB C30\include\string.h`

Pointers and Strings

Comparing Strings

Example

```
#include <string.h>

char *str = "Microchip";

int main(void)
{
    if (0 == strcmp(str, "Microchip"))
        printf("They match!\n");

    while(1);
}
```

Arrays of Pointers

Declaration

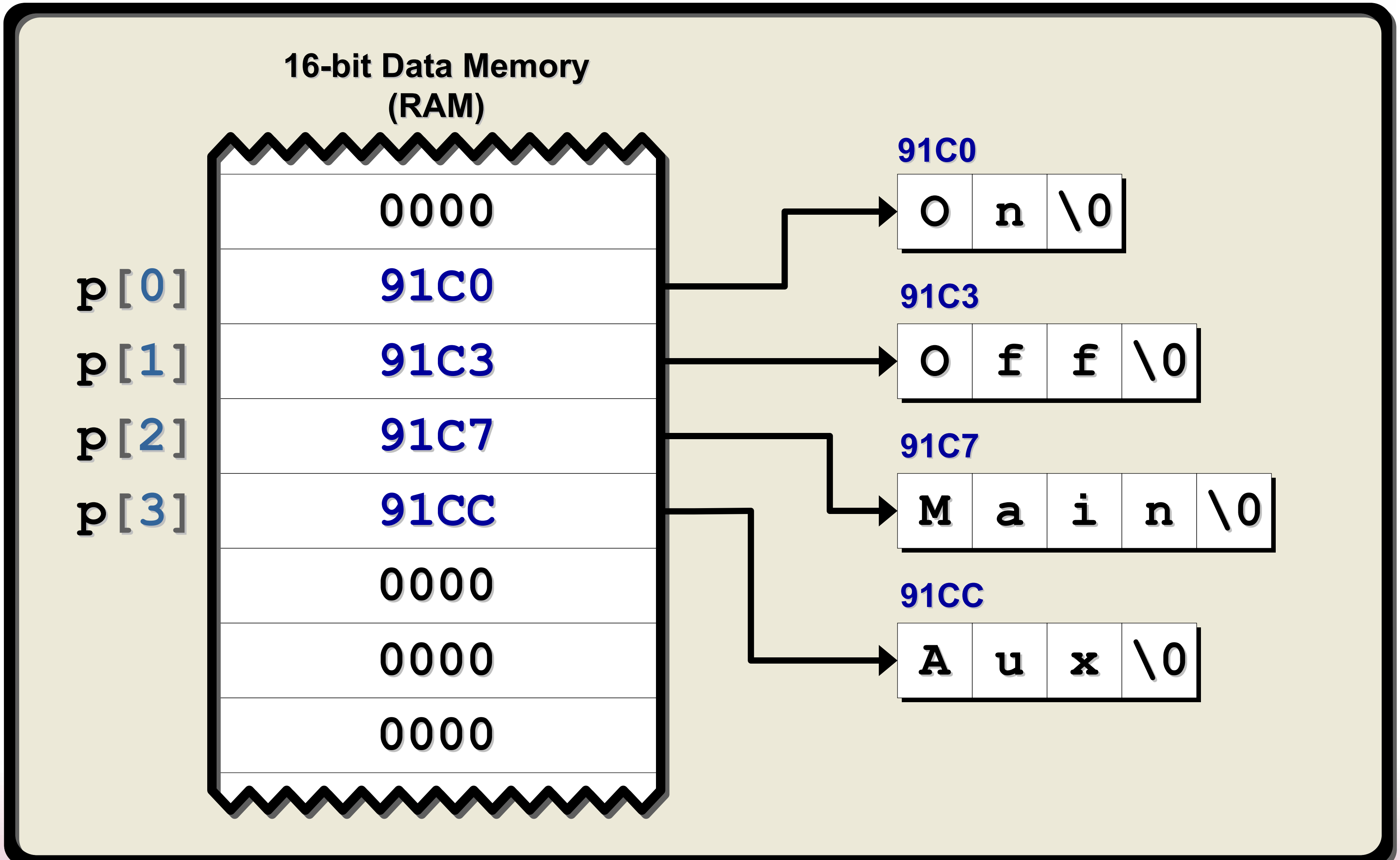
- An array of pointers is an ordinary array variable whose elements happen to all be pointers.

```
char *p[4];
```

- This creates an array of 4 pointers to `char`
  - The array `p[]` itself is like any other array
  - The elements of `p[]`, such as `p[1]`, are pointers to char

Arrays of Pointers

Array Elements are Pointers Themselves



Arrays of Pointers

Initialization

- A pointer array element may be initialized just like its ordinary variable counterpart:

```
p[0] = &x;
```

- Or, when working with strings:

```
p[0] = "My string";
```

Arrays of Pointers

Dereferencing

- To use the value pointed to by a pointer array element, just dereference it like you would an ordinary variable:

```
y = *p[0];
```

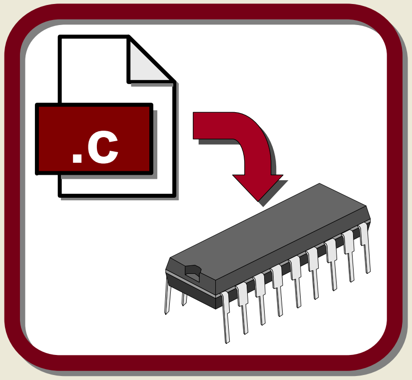
- Using `*p[0]` is the same as using the object it points to, such as `x` or the string literal `"My String"` from the previous slide

Arrays of Pointers

Accessing Strings

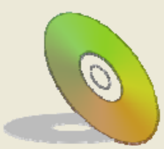
Example

```
int i = 0;
char *str[] = {"Zero", "One", "Two",
               "Three", "Four", "\0"};

int main(void)
{
    while(*str[i] != '\0')
        printf("%s\n", str[i++]);

    while(1);
}
```



Lab 12

Pointers, Arrays, and Functions



On the CD

…\101_ECP\Lab12\Lab12.mcw

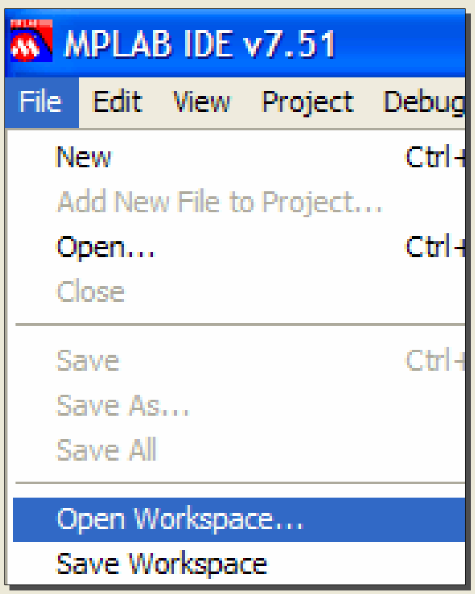
Lab 12

Pointers, Arrays, and Functions

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab12\Lab12.mcw



1. **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 12

Pointers, Arrays, and Functions

Solution: Steps 1 and 2

```
/*############################################################################
# STEP1: Pass the variable x to the function twosComplement such that the
#        value of x itself may be changed by the function.  Note: The function
#        expects a pointer (address) as its parameter.
############################################################################*/
//Perform twos complement on x
        twosComplement(&x);

/*############################################################################
# STEP 2: Pass the array 'a' to the function reverse1().  Use the constant
#         ARRAY_SIZE for the second parameter.
#         See definition of function reverse1() below.
############################################################################*/
//Reverse order of elements by passing array
        reverse1(a, ARRAY_SIZE);
```

Lab 12

Pointers, Arrays, and Functions

Solution: Steps 3 and 4

```
/*###############################################################################
# STEP 3: Pass a pointer to array 'a' to the function reverse2().  Use the
#          constant ARRAY_SIZE for the second parameter.
#          See definition of function reverse2() below.
#          Hint: You do not need to define a new pointer variable to do this.
###############################################################################*/
//Reverse order of elements by passing pointer
         reverse2(a, ARRAY_SIZE);

/*###############################################################################
# STEP 4: Complete the function header by defining a parameter called 'number'
#          that points to an integer (i.e. accepts the address of an integer
#          variable).
###############################################################################*/
//void twosComplement(/*### Your Code Here ###*/)
void twosComplement(int *number)
{
         *number = ~(*number);                  //Bitwise complement value
         *number += 1;                          //Add 1 to result
}
```

Lab 12

Conclusions

- **Pointers make it possible to pass a variable by reference to a function (allows function to modify original variable – not a copy of its contents)**
- **Arrays are frequently treated like pointers**
- **An array name alone represents the address of the first element of the array**

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.14
Function Pointers

Function Pointers

- Pointers may also be used to point to functions
- Provides a more flexible way to call a function, by providing a choice of which function to call
- Makes it possible to pass functions to other functions
- Not extremely common, but very useful in the right situations

Function Pointers

Declaration

- A function pointer is declared much like a function prototype:

```
int (*fp)(int x);
```

- Here, we have declared a function pointer with the name `fp`
  - The function it points to must take one int parameter
  - The function it points to must return an int

Function Pointers

Initialization

- A function pointer is initialized by setting the pointer name equal to the function name

If we declare the following:

```
int (*fp)(int x); //Function pointer
int foo(int x);   //Function prototype
```

We can initialize the function pointer like this:

```
fp = foo;    //fp now points to foo
```

Function Pointers

Calling a Function via a Function Pointer

- The function pointed to by fp from the previous slide may be called like this:

```
y = fp(x);
```

- This is the same as calling the function directly:

```
y = foo(x);
```

Function Pointers

Passing a Function to a Function

Example 1: Understanding the Mechanism

```
int x;
int foo(int a, int b);          //Function prototype
int bar(int a, int b);          //Function prototype

//Function definition with function pointer parameter
int foobar(int a, int b, int (*fp)(int, int))
{
   return fp(a, b);  //Call function passed by pointer
}

void main(void)
{
   x = foobar(5, 12, &foo);  //Pass address of foo
}
```

Function Pointers

Passing a Function to a Function

Example 2: Evaluate a Definite Integral (approximation)

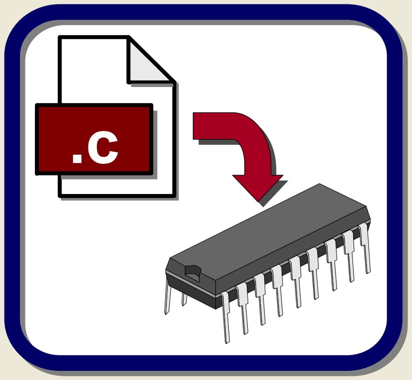
```
float integral(float a, float b, float (*f)(float))
{
    float sum = 0.0;
    float x;
    int n;

    //Evaluate integral{a,b} f(x) dx
    for (n = 0; n <= 100; n++)
    {
        x = ((n / 100.0) * (b - a)) + a;
        sum += (f(x) * (b - a)) / 101.0;
    }
    return sum;
}
```

bounds of integral (float a, float b)

function to be evaluated (float (*f)(float))

$$y = \int_a^b f(x)\,dx$$

Adapted from example at: http://en.wikipedia.org/wiki/Function_pointer



Lab 13

Function Pointers

On the CD

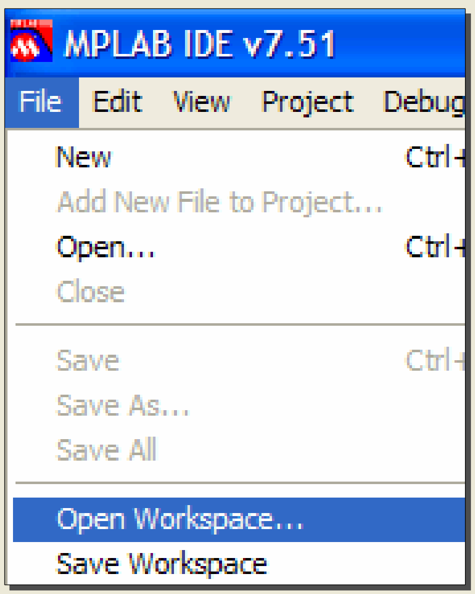
…\101_ECP\Lab13\Lab13.mcw

Lab 13
Function Pointers

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab13\Lab13.mcw



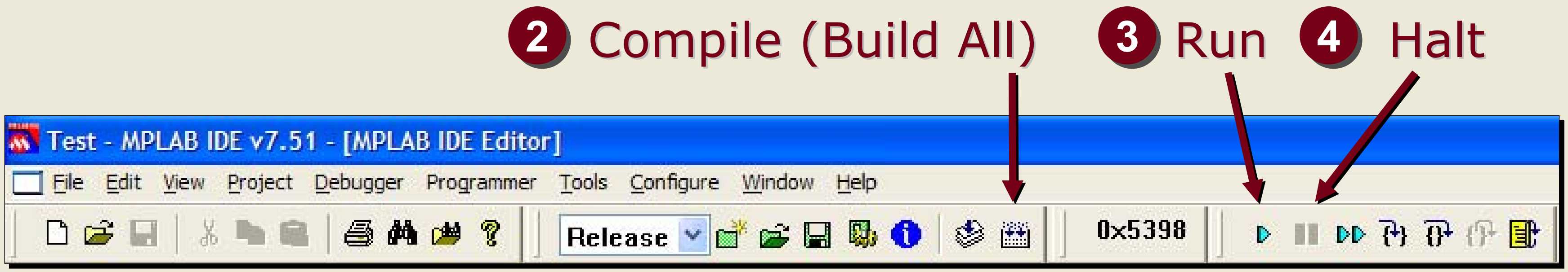
1. **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 13

Function Pointers

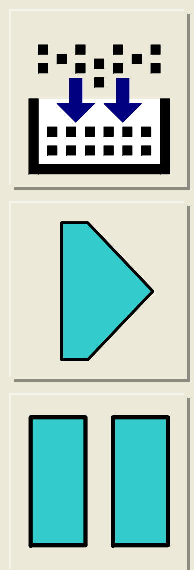
- **Compile and run the code:**



2 Click on the Build All button.

3 If no errors are reported, click on the Run button.

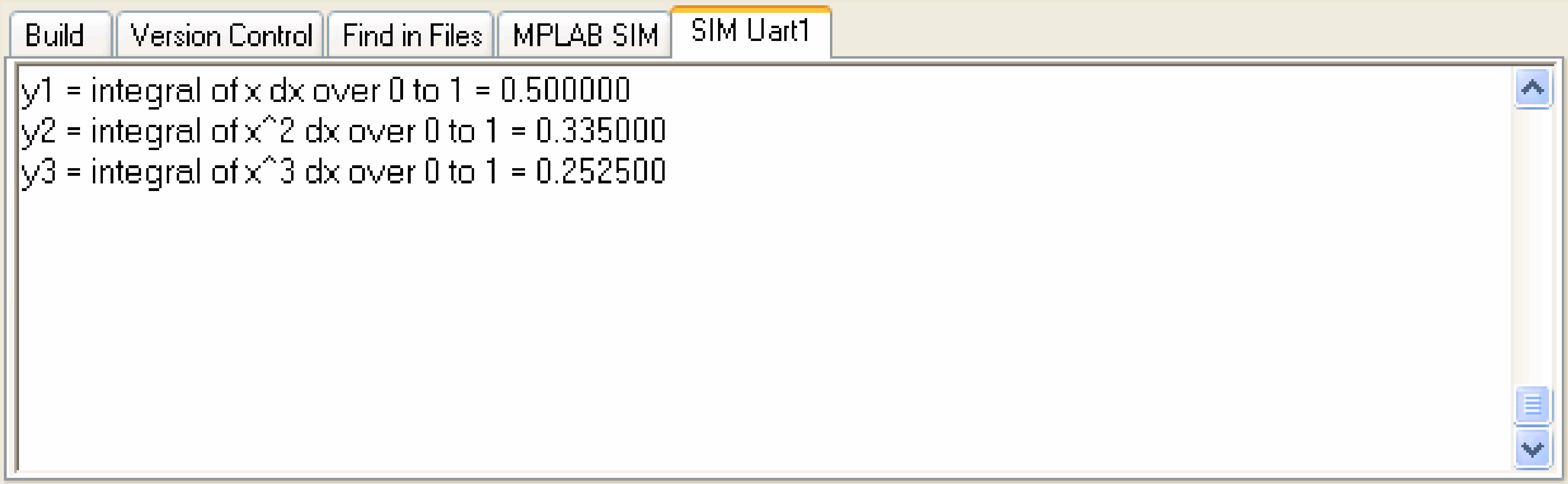
4 Click on the Halt button.



Lab 13

Function Pointers

- **Results**



Three separate functions are integrated over the interval 0 to 1:

$y_1 = \int x \, dx = \frac{1}{2} x^2 + C \, [0,1] = 0.500000$

$y_2 = \int x^2 \, dx = \frac{1}{3} x^3 + C \, [0,1] = 0.335000$

$y_3 = \int x^3 \, dx = \frac{1}{4} x^4 + C \, [0,1] = 0.252500$

Lab 13

Function Pointers

Function to Evaluate: xsquared()

```
/*============================================================================
  FUNCTION:     xsquared()
  DESCRIPTION:  Implements function y = x^2
  PARAMETERS:   float x
  RETURNS:      float (x * x)
  REQUIREMENTS: none
============================================================================*/
float xsquared(float x)
{
        return (x * x);
}



/*---------------------------------------------------------------------------
  Evaluate y2 = Int x^2 dx over the interval 0 to 1
---------------------------------------------------------------------------*/
y2 = integral(0, 1, xsquared);
```

Lab 13

Function Pointers

```
/*============================================================================
  FUNCTION:     integral()
  DESCRIPTION:  Evaluates the integral of the function passed to it over the
                interval a to b.
  PARAMETERS:   interval end points a & b and function to integrate
  RETURNS:      integral of function f over interval a to b
  REQUIREMENTS: none
  SOURCE:       Adapted from example at:
                  http://en.wikipedia.org/wiki/Function_pointer
============================================================================*/
float integral(float a, float b, float (*f)(float))
{
    float sum = 0.0;
    float x;
    int n;
    //Evaluate integral{a,b} f(x) dx
    for (n = 0; n <= 100; n++)
    {
        x = ((n / 100.0) * (b-a)) + a;
        sum += (f(x) * (b-a)) / 101.0;
    }
    return sum;
}
```

Lab 13

Conclusions

- Function pointers, while not frequently used, can provide a very convenient mechanism for passing a function to another function
- Many other possible applications exist
  - Jump tables
  - Accommodating multiple calling conventions
  - Callback functions (used in Windows™)
  - Call different versions of a function under different circumstances

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.15
Structures

Structures

Definition

Structures are collections of variables grouped together under a common name. The variables within a structure are referred to as the structure's **members**, and may be accessed individually as needed.

- **Structures:**
  - **May contain any number of members**
  - **Members may be of any data type**
  - **Allow group of related variables to be treated as a single unit, even if different types**
  - **Ease the organization of complicated data**

Structures

How to Create a Structure Definition

Syntax

```
struct structName
{
   type1 memberName1;
   ...
   typen memberNamen;
}
```

Members are declared just like ordinary variables

Example

```
// Structure to handle complex numbers
struct complex
{
   float re;     // Real part
   float im;     // Imaginary part
}
```

Structures

How to Declare a Structure Variable (Method 1)

Syntax

```
struct structName
{
   type1 memberName1;
   . . .
   typen memberNamen;
} varName1,...,varNamen;
```

Example

```
// Structure to handle complex numbers
struct complex
{
  float re;
  float im;
} x, y;         // Declare x and y of type complex
```

Structures

How to Declare a Structure Variable (Method 2)

Syntax

If *structName* has already been defined:

```
struct structName varName1,…,varNamen;
```

Example

```
struct complex
{
  float re;
  float im;
}
...
struct complex x, y;  // Declare x and y of type complex
```

Structures

How to Use a Structure Variable

Syntax

structVariableName.memberName

Example

```
struct complex
{
  float re;
  float im;
} x, y;                // Declare x and y of type complex

int main(void)
{
  x.re = 1.25;         // Initialize real part of x
  x.im = 2.50;         // Initialize imaginary part of x
  y = x;               // Set struct y equal to struct x
  ...
```

Structures

How to Create a Structure Type with `typedef`

Syntax

```
typedef struct structTag_optional
{
  type_1 memberName_1;
  ...
  type_n memberName_n;
} typeName;
```

Example

```c
// Structure type to handle complex numbers
typedef struct
{
  float re;     // Real part
  float im;     // Imaginary part
} complex;
```

Structures

How to Declare a Structure Type Variable

Syntax

If *typeName* has already been defined:

$$\texttt{typeName varName}_1\texttt{,...,varName}_n\texttt{;}$$

The keyword `struct` is no longer required!

Example

```
typedef struct
{
  float re;
  float im;
} complex;
...
complex x, y;  // Declare x and y of type complex
```

Structures

How to Initialize a Structure Variable at Declaration

Syntax

If *typeName* or *structName* has already been defined:

$$typeName\ varName = \{const_1, \ldots, const_n\};$$

- or -

$$\texttt{struct}\ structName\ varName = \{const_1, \ldots, const_n\};$$

Example

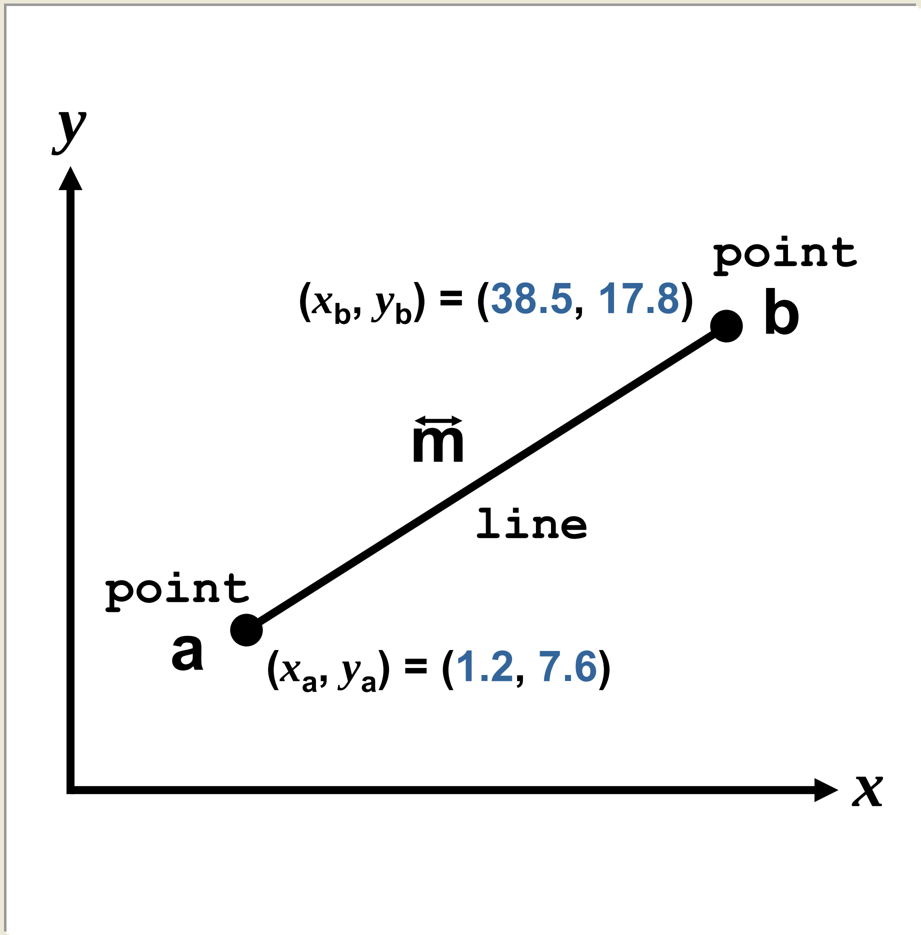
```
typedef struct
{
  float re;
  float im;
} complex;
...
complex x = {1.25, 2.50};  // x.re = 1.25, x.im = 2.50
```

Structures

Nesting Structures

Example

```c
typedef struct
{
  float x;
  float y;
} point;

typedef struct
{
  point a;
  point b;
} line;

int main(void)
{
  line m;

  m.a.x = 1.2;
  m.a.y = 7.6;
  m.b.x = 38.5;
  m.b.y = 17.8;
  ...
```



Structures

Arrays and Pointers with Strings

- **Strings:**
  - May be assigned directly to `char` array member only at declaration
  - May be assigned directly to a pointer to `char` member at any time

Example: Structure

```
struct strings
{
  char a[4];
  char *b;
} str;
```

Example: Initializing Members

```
int main(void)
{
  str.a[0] = 'B';
  str.a[1] = 'a';
  str.a[2] = 'd';
  str.a[3] = '\0';

  str.b = "Good";
```

Structures

How to Declare a Pointer to a Structure

If *typeName* or *structName* has already been defined:

Syntax

```
typeName *ptrName;
```

- or -

```
struct structName *ptrName;
```

Example 1

```
typedef struct
{
  float re;
  float im;
} complex;
...
complex *p;
```

Example 2

```
struct complex
{
  float re;
  float im;
}
...
struct complex *p;
```

Structures

How to Use a Pointer to Access Structure Members

If *ptrName* has already been defined:

Syntax

```
ptrName->memberName
```

Pointer must first be initialized to point to the address of the structure itself: `ptrName = &structVariable;`

Example: Definitions

```
typedef struct
{
  float re;
  float im;
} complex;  //complex type
...
complex x;  //complex var
complex *p; //ptr to complex
```

Example: Usage

```
int main(void)
{
  p = &x;
  //Set x.re = 1.25 via p
  p->re = 1.25;
  //Set x.im = 2.50 via p
  p->im = 2.50;
}
```

Structures

Creating Arrays of Structures

If *typeName* or *structName* has already been defined:

Syntax

```
typeName arrName[n];
```

- or -

```
struct structName arrName[n];
```

Example

```
typedef struct
{
  float re;
  float im;
} complex;
...
complex a[3];
```

Structures

Initializing Arrays of Structures at Declaration

If *typeName* or *structName* has already been defined:

Syntax

```
typeName arrName[n] = {{list1},…,{listn}};
```

- or -

```
struct structName arrName[n] = {{list1},…,{listn}};
```

Example

```
typedef struct
{
  float re;
  float im;
} complex;
...
complex a[3] = {{1.2, 2.5}, {3.9, 6.5}, {7.1, 8.4}};
```

Structures

Using Arrays of Structures

If *arrName* has already been defined:

Syntax

```
arrName[n].memberName
```

Example: Definitions

```
typedef struct
{
  float re;
  float im;
} complex;
...
complex a[3];
```

Example: Usage

```
int main(void)
{
  a[0].re = 1.25;
  a[0].im = 2.50;
  ...
}
```

Structures

How to Pass Structures to Functions

Example

```
typedef struct
{
  float re;
  float im;
} complex;

void display(complex x)
{
  printf("(%f + j%f)\n", x.re, x.im);
}

int main(void)
{
  complex a = {1.2, 2.5};
  complex b = {3.7, 4.0};

  display(a);
  display(b);
}
```

Lab 14

Structures

On the CD

…\101_ECP\Lab14\Lab14.mcw

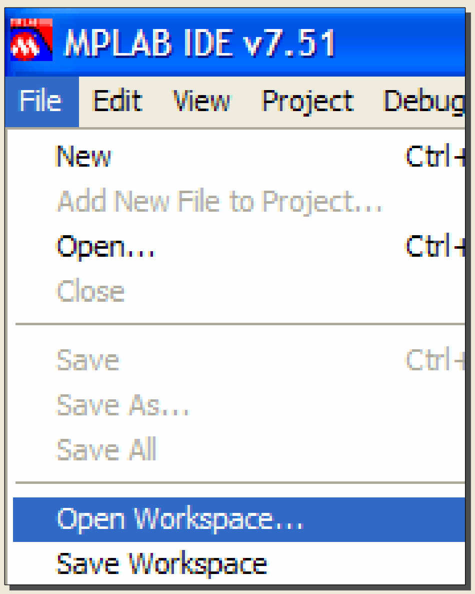
Lab 14

Structures

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab14\Lab14.mcw



1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 14

Structures

Solution: Steps 1 and 2

```
/*###############################################################################
# STEP 1: Calculate the difference between maximum and minimum power in
#         circuit 1 using the individual power structures (i.e. variables
#         PMax1 & PMin1).  Algebraic Notation:
#                                      Pdiff = (Vmax * Imax) - (Vmin * Imin)
###############################################################################*/
powerDiff1 = (PMax1.v * PMax1.i) - (PMin1.v * PMin1.i);
powerDiff2 = (PMax2.v * PMax2.i) - (PMin2.v * PMin2.i);
powerDiff3 = (PMax3.v * PMax3.i) - (PMin3.v * PMin3.i);

/*###############################################################################
# STEP 2: Calculate the difference between maximum and minimum power in
#         circuit 1 using the structure of structures (i.e. variable PRange1).
#         Algebraic Notation: Pdiff = (Vmax * Imax) - (Vmin * Imin)
###############################################################################*/
powerDiff1 = (PRange1.max.v * PRange1.max.i) - (PRange1.min.v * PRange1.min.i);
powerDiff2 = (PRange2.max.v * PRange2.max.i) - (PRange2.min.v * PRange2.min.i);
powerDiff3 = (PRange3.max.v * PRange3.max.i) - (PRange3.min.v * PRange3.min.i);
```

Lab 14

Conclusions

- **Structures make it possible to associate related variables of possibly differing types under the same name**
- **Structure members (using the dot notation) may be used anywhere an ordinary variable would be used**
- **Pointers to structures make it possible to copy one entire structure to another very easily**

Lab 15

Arrays of Structures

On the CD

...\101_ECP\Lab15\Lab15.mcw

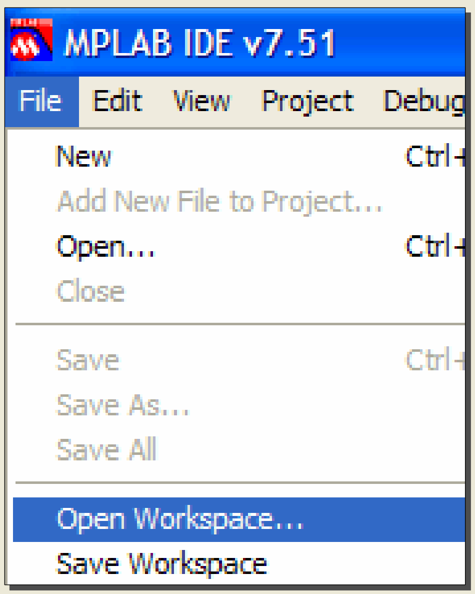
Lab 15

Arrays of Structures

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Labs\Lab15\Lab15.mcw



1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 15

Arrays of Structures

Solution: Steps 1 and 2

```
/*###############################################################################
# STEP 1: Multiply the real (re) part of each array element by 10
#         HINT: Use *=
###############################################################################*/
//Multiply re part of current array element by 10
        x[i].re *= 10;

/*###############################################################################
# STEP 2: Multiply the imaginary (im) part of each array element by 5
#         HINT: Use *=
###############################################################################*/
//Multiply im part of current array element by 5
        x[i].im *= 5;
```


Lab 15

Conclusions

- **Arrays of structures allow groups of related structures to be referenced by a common name**
- **Individual structures may be referenced by the array index**
- **Individual structure members may be referenced by the dot notation, in conjunction with the array name and index**

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.16
Unions

Unions

Definition

Unions are similar to structures but a union's members all share the same memory location. In essence a union is a variable that is capable of holding different types of data at different times.

- **Unions:**
  - **May contain any number of members**
  - **Members may be of any data type**
  - **Are as large as their largest member**
  - **Use exactly the same syntax as structures except `struct` is replaced with `union`**

Unions

How to Create a Union

Syntax

```
union unionName
{
  type1 memberName1;
  ...
  typen memberNamen;
}
```

Example

```
// Union of char, int and float
union mixedBag
{
  char a;
  int b;
  float c;
}
```

Unions

How to Create a Union Type with `typedef`

Syntax

```
typedef union unionTag_optional
{
  type_1 memberName_1;
  ...
  type_n memberName_n;
} typeName;
```

Example

```
// Union of char, int and float
typedef union
{
  char a;
  int b;
  float c;
} mixedBag;
```

Unions

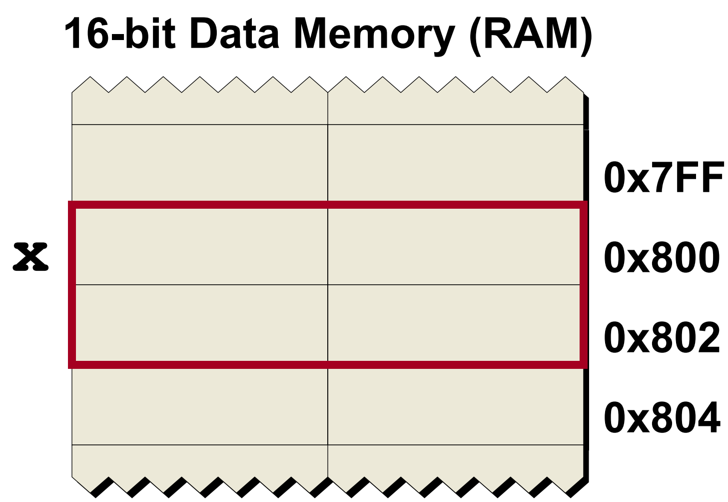
How Unions Are Stored In Memory

- Union variables may be declared exactly like structure variables
- Memory is only allocated to accommodate the union's largest member

Example

```
typedef union
{
  char a;
  int b;
  float c;
} mixedBag;

mixedBag x;
```

Space allocated for `x` is size of `float`



Unions

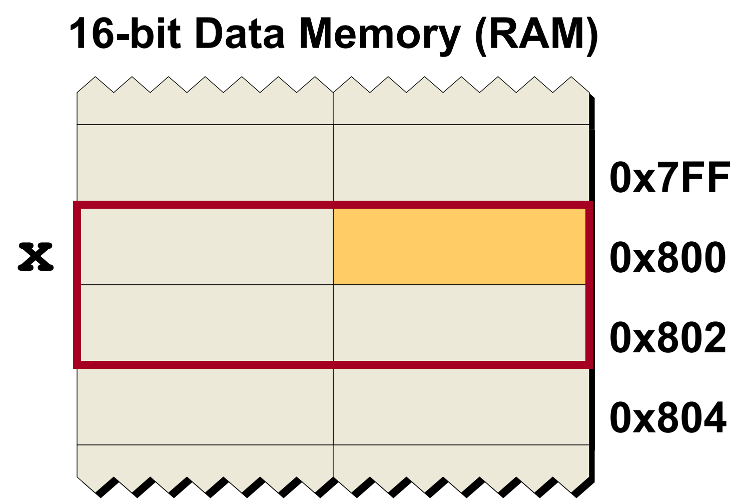
How Unions Are Stored In Memory

- **Union variables may be declared exactly like structure variables**
- **Memory is only allocated to accommodate the union's largest member**

Example

```
typedef union
{
   char a;
   int b;
   float c;
} mixedBag;

mixedBag x;
```

`x.a` only occupies the lowest byte of the union



Unions

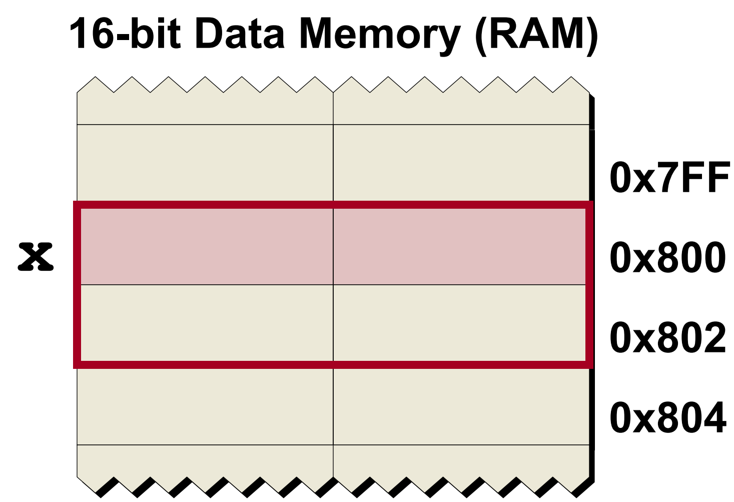
How Unions Are Stored In Memory

- Union variables may be declared exactly like structure variables
- Memory is only allocated to accommodate the union's largest member

Example

```
typedef union
{
  char a;
  int b;
  float c;
} mixedBag;

mixedBag x;
```

`x.b` only occupies the lowest two bytes of the union



Unions

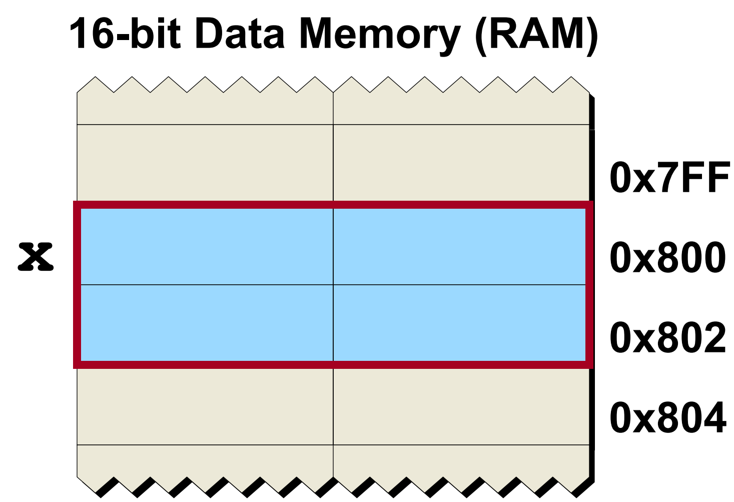
How Unions Are Stored In Memory

- Union variables may be declared exactly like structure variables
- Memory is only allocated to accommodate the union's largest member

Example

```
typedef union
{
  char a;
  int b;
  float c;
} mixedBag;

mixedBag x;
```

`x.c` occupies all four bytes of the union



Lab 16

Unions

On the CD

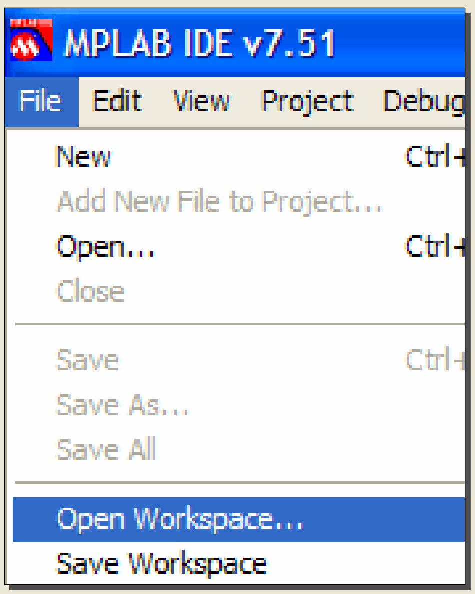
...\101_ECP\Lab16\Lab16.mcw

Lab 16
Unions

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab16\Lab16.mcw



1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 16

Unions

Solution: Steps 1 and 2

```
/*###############################################################################
# STEP 1: Set the int member of unionVar equal to 16877.
###############################################################################*/
//Set intVar = 16877
        unionVar.intVar = 16877;

/*###############################################################################
# STEP 2: Set the float member of unionVar equal to 6.02e23.
###############################################################################*/
//Set floatVar = 6.02e23
        unionVar.floatVar = 6.02e23;
```

Lab 16

Conclusions

- **Unions make it possible to store multiple variables at the same location**
- **They make it possible to access those variables in different ways**
- **They make it possible to store different variable types in the same memory location(s)**

Section 1.17

Bit Fields

Bit Fields

Definition

Bit Fields are `unsigned int` members of structures that occupy a specified number of adjacent bits from one to `sizeof(int)`. They may be used as an ordinary `int` variable in arithmetic and logical operations.

- **Bit Fields:**
  - **Are ordinary members of a structure**
  - **Have a specified bit width**
  - **Are often used in conjunction with unions to provide bit access to a variable without masking operations**

Bit Fields

How to Create a Bit Field

Syntax

```
struct structName
{
  unsigned int memberName1: bitWidth;
  ...
  unsigned int memberNamen: bitWidth;
}
```

Example

```
typedef struct
{
  unsigned int bit0: 1;
  unsigned int bit1to3: 3;
  unsigned int bit4: 1;
  unsigned int bit5: 1;
  unsigned int bit6to7: 2;
} byteBits;
```

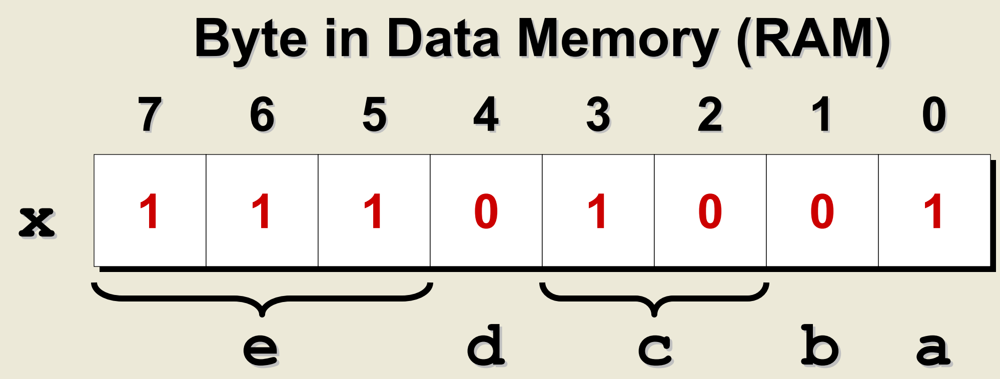
bitfield `struct` may be declared normally or as a `typedef`

Bit Fields

How to Use a Bit Field

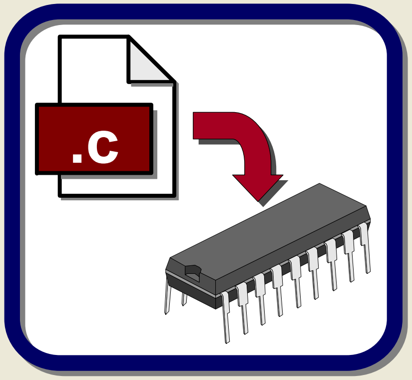
Example

```
struct byteBits
{
  unsigned a: 1;
  unsigned b: 1;
  unsigned c: 2;
  unsigned d: 1;
  unsigned e: 3;
} x;
```




```
int main(void)
{
  x.a = 1;        //x.a may contain values from 0 to 1
  x.b = 0;        //x.b may contain values from 0 to 1
  x.c = 0b10;     //x.c may contain values from 0 to 3
  x.d = 0x0;      //x.d may contain values from 0 to 1
  x.e = 7;        //x.e may contain values from 0 to 7
}
```

Lab 17



Bit Fields

On the CD

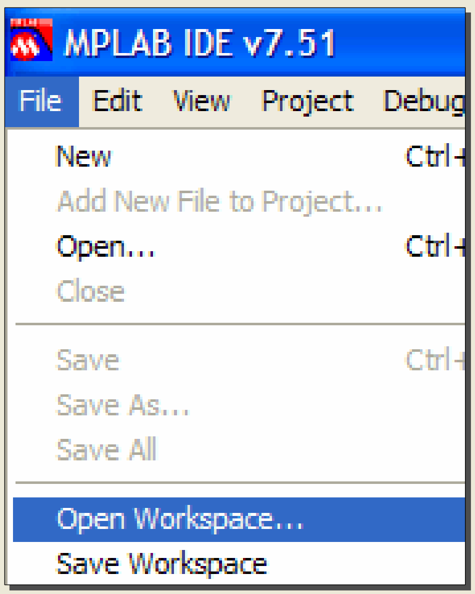
...\101_ECP\Lab17\Lab17.mcw

Lab 17
Bit Fields

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab17\Lab17.mcw

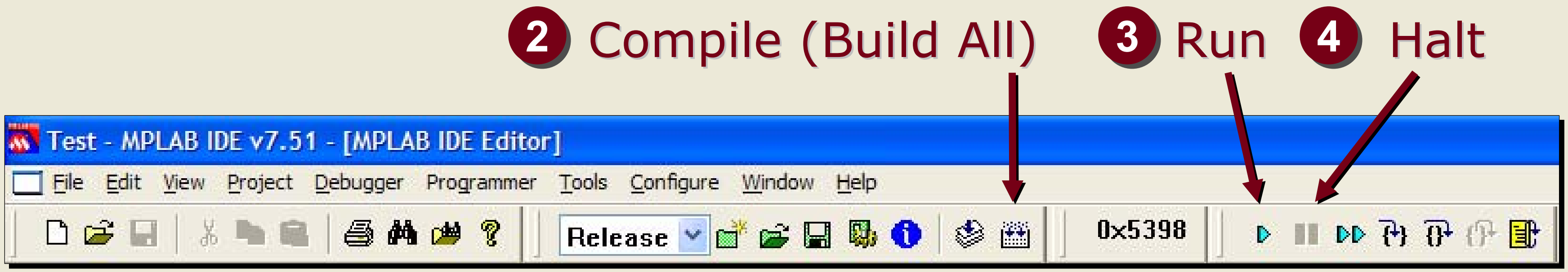


1. **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

⚠ **If you already have a project open in MPLAB IDE, close it by selecting** Close Workspace **from the** File **menu before opening a new one.**

Lab 17
Bit Fields

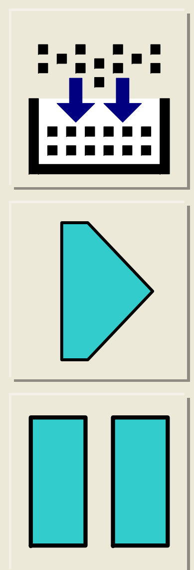
- **Compile and run the code:**



2 **Click on the Build All button.**

3 **If no errors are reported, click on the Run button.**

4 **Click on the Halt button.**



Lab 17

Bit Fields

Bit Field Definition

```
/*------------------------------------------------------------------------------
  VARIABLE DECLARATIONS
------------------------------------------------------------------------------*/
union {
        char fullByte;
        struct {
                int bit0: 1;
                int bit1: 1;
                int bit2: 1;
                int bit3: 1;
                int bit4: 1;
                int bit5: 1;
                int bit6: 1;
                int bit7: 1;
        } bitField;
} bitByte;
```

Lab 17
Bit Fields

Demo Results 1

Add SFR | AD1CHS | Add Symbol | __SP

Address	Symbol Name	Value
0800	⊟ bitByte	
0800	fullByte	0x55
0800	⊟ bitFiel	0x0055
0800	bit0	0001
0800	bit1	0000
0800	bit2	0001
0800	bit3	0000
0800	bit4	0001
0800	bit5	0000
0800	bit6	0001
0800	bit7	0000

```
bitByte.fullByte = 0x55;
```

Lab 17

Bit Fields

Demo Results 2

Add SFR | AD1CHS | Add Symbol | __SP

Address	Symbol Name	Value
0800	⊟ bitByte	
0800	fullByte	0x54
0800	⊟ bitFiel	0x0054
0800	bit0	0000
0800	bit1	0000
0800	bit2	0001
0800	bit3	0000
0800	bit4	0001
0800	bit5	0000
0800	bit6	0001
0800	bit7	0000

```
bitByte.bitField.bit0 = 0;
```

Lab 17
Bit Fields

Demo Results 3

Add SFR | AD1CHS | Add Symbol | __SP

Address	Symbol Name	Value
0800	⊟ bitByte	
0800	fullByte	0x50
0800	⊟ bitFiel	0x0050
0800	bit0	0000
0800	bit1	0000
0800	bit2	0000
0800	bit3	0000
0800	bit4	0001
0800	bit5	0000
0800	bit6	0001
0800	bit7	0000

```
bitByte.bitField.bit2 = 0;
```

Lab 17
Bit Fields

Demo Results 4

Add SFR | AD1CHS | Add Symbol | __SP

Address	Symbol Name	Value
0800	⊟ bitByte	
0800	fullByte	0xD0
0800	⊟ bitFiel	0x00D0
0800	bit0	0000
0800	bit1	0000
0800	bit2	0000
0800	bit3	0000
0800	bit4	0001
0800	bit5	0000
0800	bit6	0001
0800	bit7	0001

```
bitByte.bitField.bit7 = 1;
```

Lab 17

Conclusions

- **Bit fields provide an efficient mechanism to store Boolean values, flags and semaphores in data memory**
- **Care must be used if code size or speed is a concern**
  - **Compiler will usually make use of bit set / bit clear instructions**
  - **In some circumstances this isn't possible (comparing bit values)**

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.18
Enumerations

Enumerations

Definition

Enumerations are integer data types that you can create with a limited range of values. Each value is represented by a symbolic constant that may be used in conjunction with variables of the same enumerated type.

- **Enumerations:**
  - **Are unique integer data types**
  - **May only contain a specified list of values**
  - **Values are specified as symbolic constants**

Enumerations

How to Create an Enumeration Type

- Creates an ordered list of constants
- Each label's value is one greater than the previous label

Syntax

```
enum typeName {label0, label1,…,labeln}
```

Where compiler sets $label_0$ = 0, $label_1$ = 1, $label_n$ = n

Example

```
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT};
```

Label Values:

SUN = 0, MON = 1, TUE = 2, WED = 3, THR = 4 , FRI = 5, SAT = 6

Enumerations

How to Create an Enumeration Type

- Any label may be assigned a specific value
- The following labels will increment from that value

Syntax

```
enum typeName {label0 = const0,…,labeln}
```

Where compiler sets $label_0 = const_0$, $label_1 = (const_0 + 1)$, ...

Example

```
enum people {Rob, Steve, Paul = 7, Bill, Gary};
```

Label Values:

Rob = 0, Steve = 1, Paul = 7, Bill = 8, Gary = 9

Enumerations

How to Declare an Enumeration Type Variable

- **Declared along with type:**

Syntax

```
enum typeName {const-list} varname1,…;
```

- **Declared independently:**

Syntax

```
enum typeName varName1,…,varNamen;
```

Example

```
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT} today;

enum weekday someday;  //day is a variable of type weekday
```

Enumerations

How to Declare a 'Tagless' Enumeration Variable

- **No type name specified:**

Syntax

```
enum {const-list} varName1,…,varNamen;
```

- **Only variables specified as part of the `enum` declaration may be of that type**
- **No type name is available to declare additional variables of the `enum` type later in code**

Example

```
enum {SUN, MON, TUE, WED, THR, FRI, SAT} today;
```

Enumerations

How to Declare an Enumeration Type with `typedef`

- Variables may be declared as type ***typeName*** without needing the `enum` keyword

Syntax

```
typedef enum {const-list} typeName;
```

- The enumeration may now be used as an ordinary data type (compatible with `int`)

Example

```
typedef enum {SUN, MON, TUE, WED, THR, FRI, SAT} weekday;

weekday day;      //Variable of type weekday
```

Enumerations

How to Use an Enumeration Type Variable

If enumeration and variable have already been defined:

Syntax

$varName = label_n;$

- The labels may be used as any other symbolic constant
- Variables defined as enumeration types must be used in conjunction with the type's labels or equivalent integer

Example

```
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT};
enum weekday day;

day = WED;
day = 6;                    //May only use values from 0 to 6
if (day == WED)
{ …
```

Lab 18

Enumerations

On the CD

…\101_ECP\Lab18\Lab18.mcw

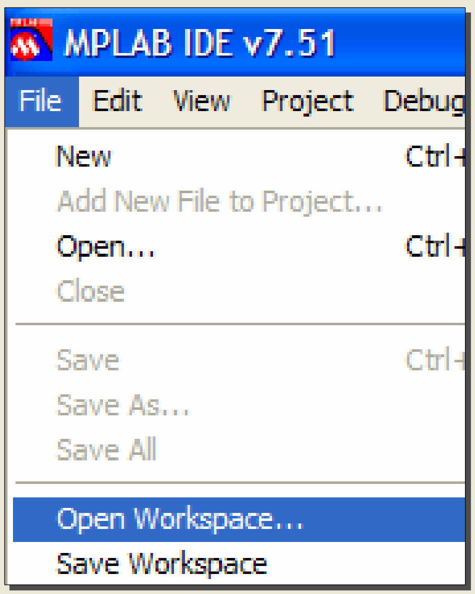
Lab 18

Enumerations

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab18\Lab18.mcw



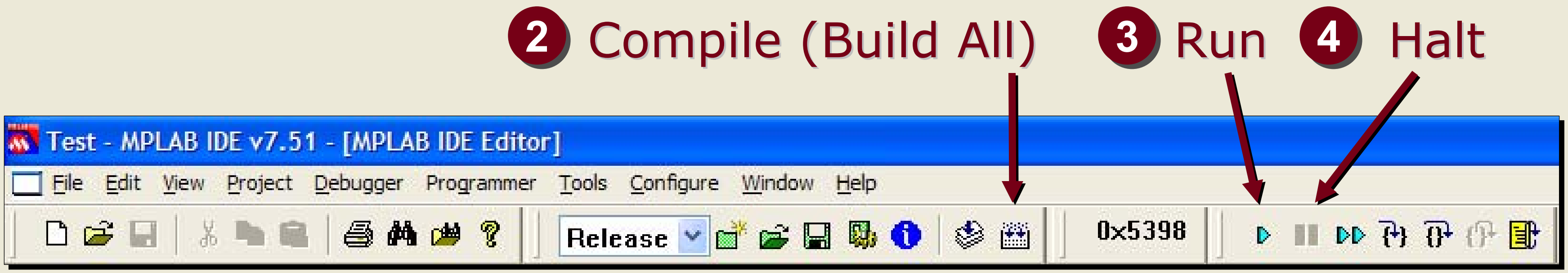
1 **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

If you already have a project open in MPLAB IDE, close it by selecting Close Workspace **from the** File **menu before opening a new one.**

Lab 18

Enumerations

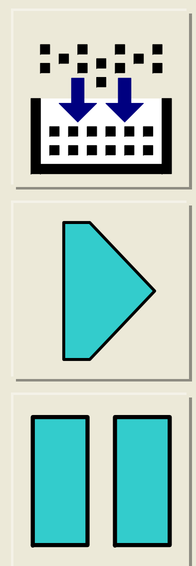
- **Compile and run the code:**



2 Click on the Build All button.

3 If no errors are reported, click on the Run button.

4 Click on the Halt button.



Lab 18

Enumerations

Enum Definition and Use

```
typedef enum {BANDSTOP, LOWPASS, HIGHPASS, BANDPASS} filterTypes;

filterTypes filter;

/*============================================================================
  FUNCTION:     main()
============================================================================*/
int main(void)
{
   filter = BANDPASS;

   switch (filter)
   {
    case BANDSTOP: BandStopFilter(); break;
    case LOWPASS:  LowPassFilter();  break;
    case HIGHPASS: HighPassFilter(); break;
    case BANDPASS: BandPassFilter(); break;
        }

    while(1);
}
```

Lab 18

Conclusions

- Enumerations provide a means of associating a list of constants with one or more variables
- Make code easier to read and maintain
- Variables declared as `enum` are essentially still `int` types

YOU + MICROCHIP ENGINEERING THE FUTURE TOGETHER

Section 1.19
Macros with #define

Macros with `#define`

Definition

Macros are text replacements created with #define that insert code into your program. Macros may take parameters like a function, but the macro code and parameters are always inserted into code by text substitution.

- **Macros**
  - **Are evaluated by the preprocessor**
  - **Are not executable code themselves**
  - **Can control the generation of code before the compilation process**
  - **Provide shortcuts**

Macros with #define

Simple Macros

- Text substitution as seen earlier

Syntax

```
#define label text
```

- Every instance of *label* in the current file will be replaced by *text*
- *text* can be anything you can type into your editor
- Arithmetic expressions evaluated at compile time

Example

```
#define Fosc 4000000
#define Tcy (0.25 * (1/Fosc))
#define Setup InitSystem(Fosc, 250, 0x5A)
```

Macros with `#define`

Argument Macros

- Create a function-like macro

Syntax

```
#define label(arg1,…,argn) code
```

- The *code* must fit on a single line or use '\' to split lines
- Text substitution used to insert arguments into *code*
- Each instance of `label()` will be expanded into *code*
- This is not the same as a C function!

Example

```
#define min(x, y) ((x)<(y)?(x):(y))
#define square(x) ((x)*(x))
#define swap(x, y) { x ^= y; y ^= x; x ^= y; }
```

Macros with #define

Argument Macros – Side Effects

Example

```
#define square(a) ((a)*(a))
```

Extreme care must be exercised when using macros. Consider the following use of the above macro:

```
i = 5;
x = square(i++);
```

Results:

x = 30 ✗
i = 7 ✗

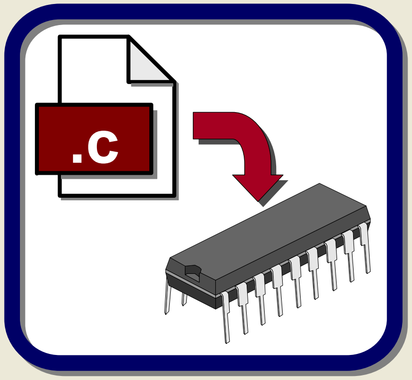
Wrong Answers!

```
x = square(i++);
```

expands to:

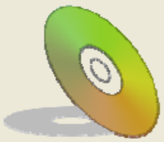
```
x = ((i++)*(i++));
```

So i gets incremented twice, not once at the end as expected.



Lab 19

#define Macros



On the CD

...\101_ECP\Lab19\Lab19.mcw

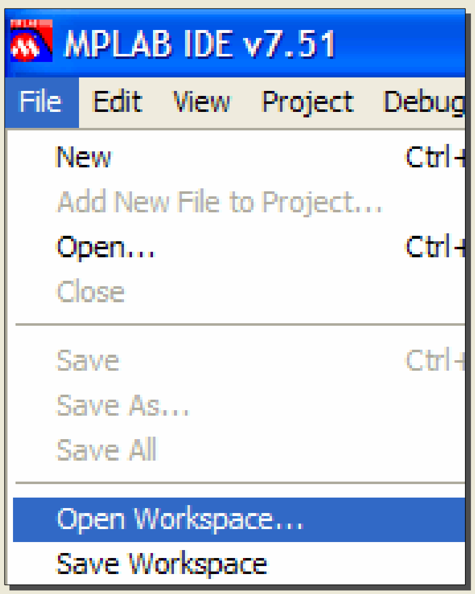
Lab 19

#define Macros

- **Open the project's workspace:**

On the lab PC

C:\RTC\101_ECP\Lab19\Lab19.mcw



1. **Open MPLAB® IDE and select** Open Workspace... **from the** File **menu. Open the file listed above.**

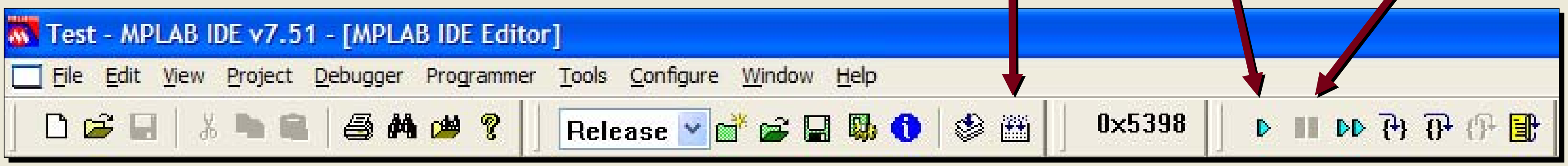
⚠ **If you already have a project open in MPLAB IDE close it by selecting** Close Workspace **from the** File **menu before opening a new one.**

Lab 19

`#define` Macros

- **Compile and run the code:**

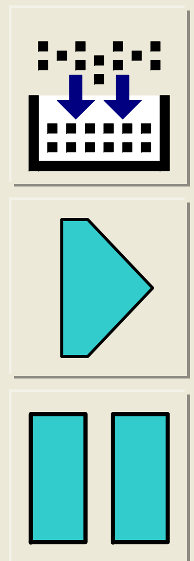
2 Compile (Build All) 3 Run 4 Halt



2 **Click on the Build All button.**

3 **If no errors are reported, click on the Run button.**

4 **Click on the Halt button.**



Lab 19

#define Macros

#define Macro Definition and Use

```c
/*----------------------------------------------------------------------------
  MACROS
----------------------------------------------------------------------------*/
#define square(m)  ((m) * (m))
#define BaudRate(DesiredBR, FoscMHz)  ((((FoscMHz * 1000000)/DesiredBR)/64)-1)




/*============================================================================
  FUNCTION:     main()
============================================================================*/
int main(void)
{
        x = square(3);
        printf("x = %d\n", x);

        SPBRG = BaudRate(9600, 16);
        printf("SPBRG = %d\n", SPBRG);
}
```

Lab 19

Conclusions

- #define macros can dramatically simplify your code and make it easier to maintain
- Extreme care must be taken when crafting a macro due to the way they are substituted within the text of your code

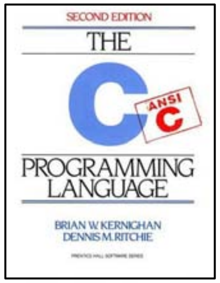
Resources

A Selection of C Compilers

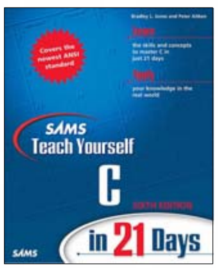
- **Microchip Technology MPLAB® C30 and MPLAB® C18 (Free 'student' versions available)**
  http://www.microchip.com
- **Hi-Tech PICC™, PICC-18™, C for dsPIC®/PIC24**
  http://www.htsoft.com
- **Custom Computer Services Inc. (CCS) C Compilers**
  http://www.ccsinfo.com
- **ByteCraft Ltd. MPC**
  http://www.bytecraft.com
- **IAR Systems Embedded Workbench**
  http://www.iar.com
- **Small Device C Compiler (Free)**
  http://sourceforge.net/projects/sdcc/
- **SourceBoost BoostC™**
  http://www.sourceboost.com/

Resources

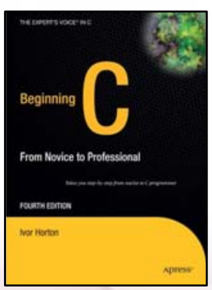
Books – General C Language



- **The C Programming Language**
  2nd Edition (March 22, 1988)
  Brian W. Kernighan & Dennis Ritchie
  ISBN-10: 0131103628
  ISBN-13: 978-0131103627



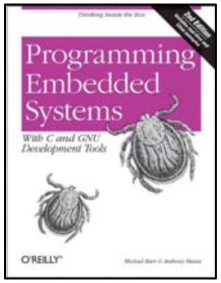
- **SAMS Teach Yourself C in 21 Days**
  6th Edition (September 25, 2002)
  Bradley L. Jones & Peter Aitken
  ISBN-10: 0672324482
  ISBN-13: 978-0672324482



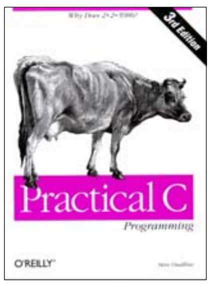
- **Beginning C From Novice to Professional**
  4th Edition (October 19, 2006)
  Ivor Horton
  ISBN-10: 1590597354
  ISBN-13: 978-1590597354

Resources

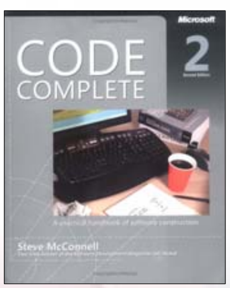
Books – General C Language



- **<u>Programming Embedded Systems</u>**
  <u>with C and GNU Development Tools</u>
  2nd Edition (October 1, 2006)
  Michael Barr & Anthony Massa
  ISBN-10: 0596009836
  ISBN-13: 978-0596009830



- **<u>Practical C Programming</u>**
  3rd Edition (August 1, 1997)
  Steve Oualline
  ISBN-10: 1565923065
  ISBN-13: 978-1565923065

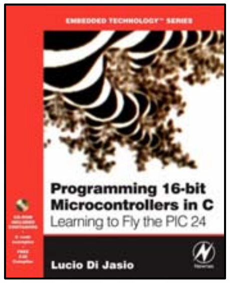


- **<u>Code Complete</u>**
  2nd Edition (June 2004)
  Steve McConnell
  ISBN-10: 0735619670
  ISBN-13: 978-0735619678

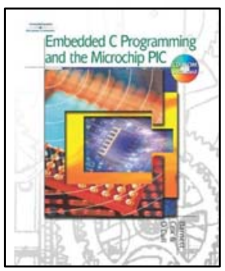
Not about C specifically, but a must read for all software engineers

Resources

Books – PIC® MCU Specific



- **Programming 16-Bit PIC Microcontrollers in C**
  Learning to Fly the PIC24
  1st Edition (March 16, 2007)
  Lucio Di Jasio
  ISBN-10: 0750682922
  ISBN-13: 978-0750682923

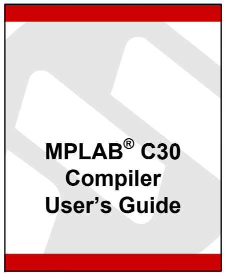


- **Embedded C Programming and the Microchip PIC**
  1st Edition (November 3, 2003)
  Richard H. Barnett, Sarah Cox, Larry O'Cull
  ISBN-10: 1401837484
  ISBN-13: 978-1401837488

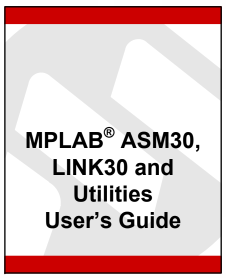
- **PICmicro MCU C:**
  An Introduction to Programming the Microchip PIC in CCS C
  2nd Edition (August 19, 2002)
  Nigel Gardner
  ISBN-10: 0972418105
  ISBN-13: 978-0972418102

Resources

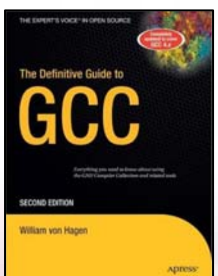
Books – Compiler Specific



- **MPLAB® C30 C Compiler User's Guide**
  Current Edition (PDF)
  Microchip Technology
  DS51284F
  http://www.microchip.com



- **MPLAB® ASM30 LINK30 and Utilities User's Guide**
  Current Edition (PDF)
  Microchip Technology
  DS51317F
  http://www.microchip.com

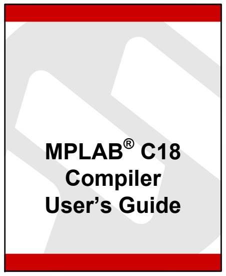


- **The Definitive Guide to GCC**
  2nd Edition (August 11, 2006)
  William von Hagen
  ISBN-10: 1590595858
  ISBN-13: 978-1590595855

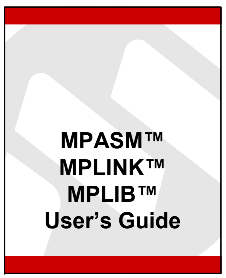
MPLAB® C30 is based on the GCC tool chain

Resources

Books – Compiler Specific



- **MPLAB® C18 C Compiler User's Guide**
  Current Edition (PDF)
  Microchip Technology
  DS51288J
  http://www.microchip.com



- **MPASM™ MPLINK™ and MPLIB™ User's Guide**
  Current Edition (PDF)
  Microchip Technology
  DS33014J
  http://www.microchip.com

> The older books on C are much more relevant to embedded C programming since they were written back when PCs and other computers had limited resources and programmers had to manage them carefully.

Thank you!

Trademarks

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KeeLoq, KeeLoq logo, MPLAB, PIC, PICmicro, PICSTART, PRO MATE, rfPIC and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, Linear Active Thermistor, MXDEV, MXLAB, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, PICkit, PICDEM, PICDEM.net, PICtail, PIC32 logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rfLAB, Select Mode, Total Endurance, UNI/O, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.