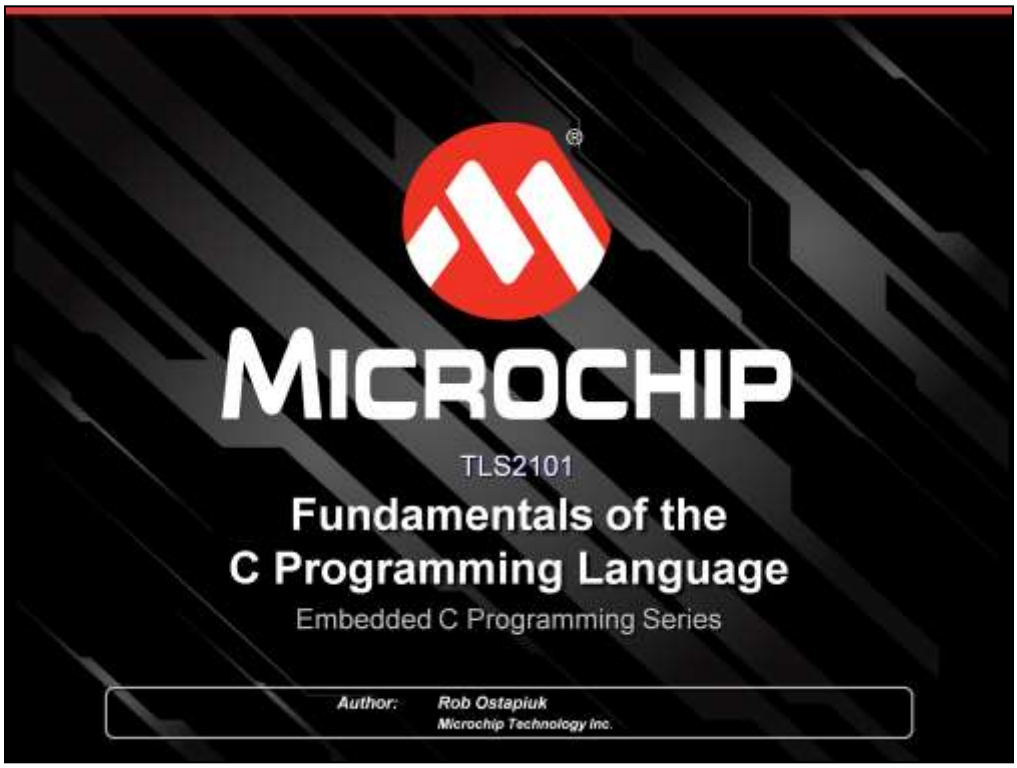


V3.2 February 2013

Class Objectives

- In the next 2 days you will learn how to read, write, and modify programs written in the C language for use in embedded systems.
- Techniques and skills learned in this class are based on the ANSI standard for the C language and are applicable to programs written for any processor using any ANSI compliant C compiler

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 2

You are going to learn the C language as defined by ANSI.. What you will learn is not specific to Microchip, but applicable to all manufacturers or compiler. Other courses are available for Microchip Uniques uses fo C

Agenda

Day One - Morning

- Using C in an Embedded Environment
- Comments
- Variables, Identifiers and Data Types
- Literal Constants
- Symbolic Constants
- printf() Library Function
- Operators

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Agenda

Day One Afternoon

- Expressions and Statements
- Making Decisions
- Loops
- Functions
- Multi-File Projects & Storage Class Specifiers

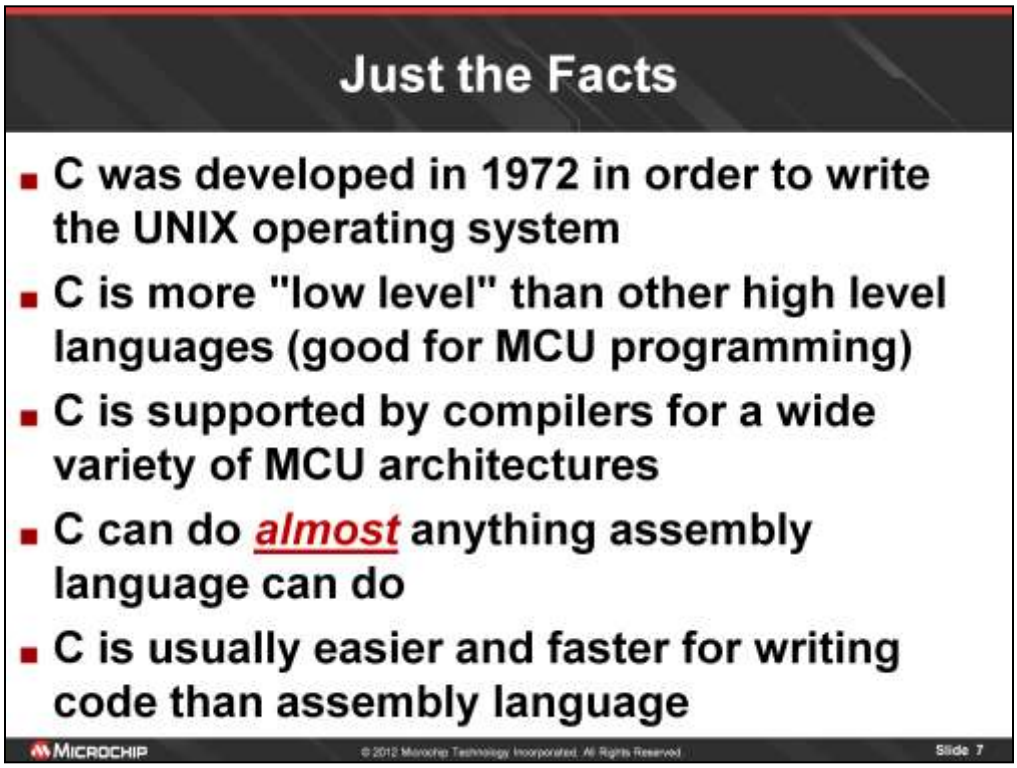
© 2012 Microchip Technology Incorporated. All Rights Reserved.

Agenda
Day Two

- Arrays
- Data Pointers
- Function Pointers
- Structures
- Unions
- Bit Fields
- Enumerations
- Macros with #define

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Using C in an Embedded Environment



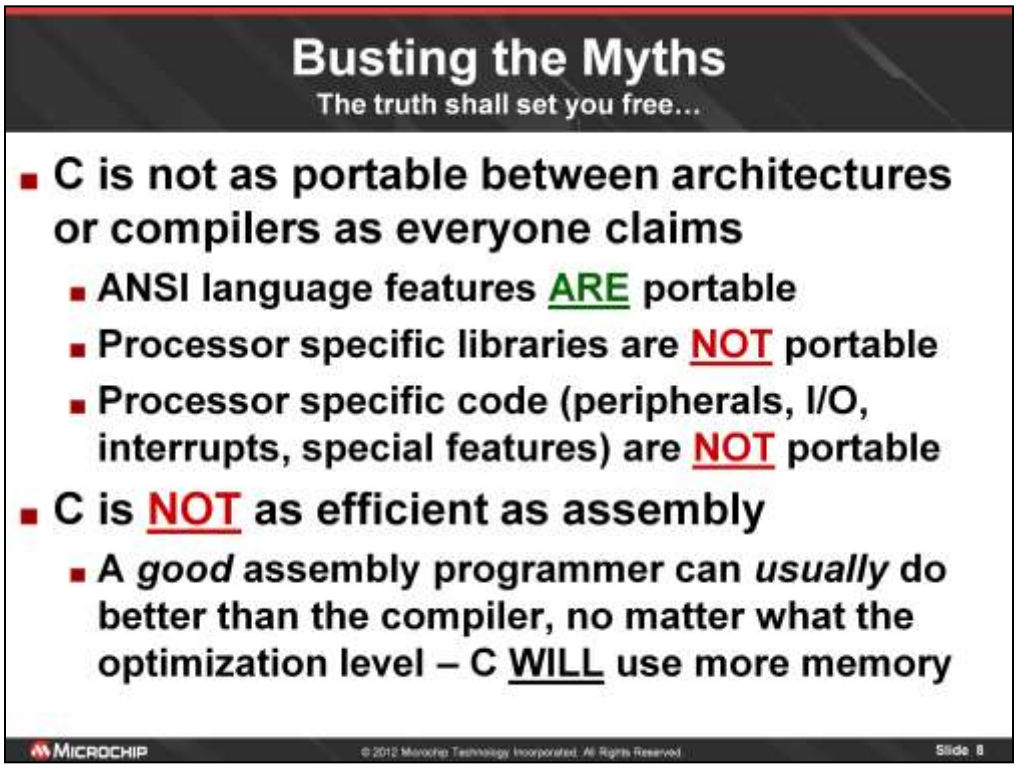
Before we get into the details of the C programming language, we first need to establish the facts about C and also bust a few of the myths that continue to be believed by programmers today.

C was originally developed at Bell Labs by Dennis Ritchie for use in developing operating systems. The first major use for C was in writing the UNIX operating system back in 1974. Due to the low level nature of operating system programming (think device drivers), C has a number of features that are a real benefit to the embedded systems developer of today. When you really think about it, every time we write code for an embedded microcontroller, we are writing an operating system for that platform.

A big advantage of C today is that there are C compilers available for virtually every microcontroller and computer platform in existence. So if you know C, you will have the programming skills needed to write code for a wide variety of devices.

Also, because of the low level features of C, it can do almost anything assembly language can do.

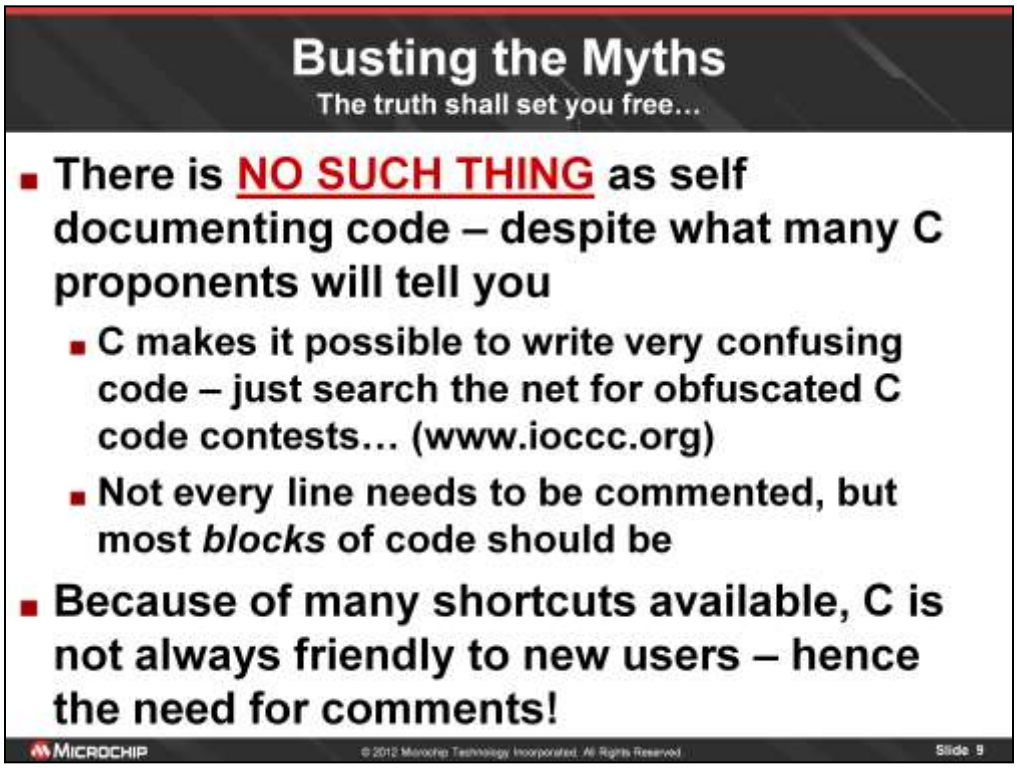
And of course, the primary reason most people use C is that for many people it is much easier to learn and can usually lead to faster development times.



Now to the downsides. Over the years there have been a lot of myths perpetuated about what C will do for you. The number one reason most people cite for using C is that it is extremely portable due to the vast array of compilers available for all the different platforms. The problem is that C really isn't portable. For example, can a program written in C for a Windows PC run on a Macintosh? Absolutely not. The way those operating systems handle the interface to their hardware, and the hardware itself are completely different. The same holds true for embedded systems, even those from different manufacturers that use the same core. The ways in which you work with memory, handle interrupts, use the on-chip peripherals are always different. Even if you try to move from one compiler to another for the exact same device you will have these problems. The ways you declare an interrupt service routine using the Hi-Tech C compiler for the PIC18 and the MPLAB-C18 compiler are completely different.

Now, the core ANSI C language itself and any algorithms you write that don't interface specifically with the hardware will be portable. But in any embedded system, you will undoubtedly make use of the on-chip peripherals and interrupts, and there is no standard way of handling those.

And, despite what some C code jocks will try to tell you, C will NEVER be as efficient as assembly. A good assembly programmer will be able to outperform the best C compiler in most cases. Due to the way C sets things up on the target processor, C will always use more memory than a well written assembly language program.



Something I have heard from a number of C gurus that strikes terror into my heart is that C is self documenting. I cannot emphasize enough that there is NO SUCH THING as self documenting code. I don't care how well structured your code is or how well you name your variables and functions. If you come back to code you wrote one year ago, you will have trouble figuring out exactly what you were doing without comments.

C is a very flexible language that enables you to write some very powerful but terse lines of code. On their own, they are not too difficult to work with, but once you combine many lines of code like this, a program can quickly grow to be totally incomprehensible. In fact, there are regular contests run by a number of organizations on writing obfuscated C code, where the most incomprehensible, yet elegant and functional code wins the prize. It is surprisingly easy to do this in C.

Now, I'm not suggesting that you should comment every single line of code. But at the very least, you should comment each functional block with a brief explanation in plain language of what it is supposed to do.

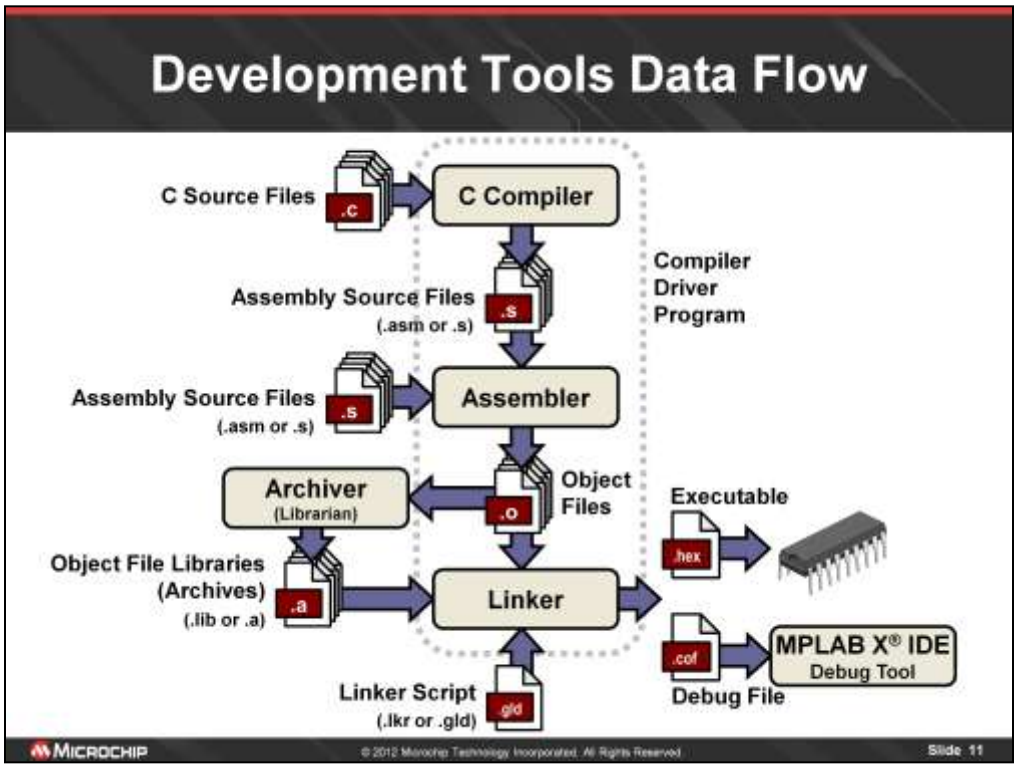
Obfuscated Code Example

- From Wikipedia entry for 'Obfuscated Code'
  - What does this code do?

```
main( _, t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,
  /#;\
#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/+k#;\
q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;#
  ){nl]!/n{n#'; \ r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
\ n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c ;;\
{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
 :   t<-50?    _==*a ?
putchar(31[a]):
```

  - Generates the lyrics for the popular song "The Twelve Days of Christmas"!

© 2012 Microchip Technology Incorporated. All Rights Reserved.



Now that we have given you some of the facts and busted a few myths, let's take a look at the process of converting C source code into something that a microcontroller can understand. Not every compiler works like this, but they will all follow a similar flow, even if some steps are omitted (This is the MPLAB-C30 process). First we start off by writing our C source file, which is what this class is all about. Once we have our code, it gets run through a compiler, which often has an additional step associated with it. We'll discuss this on the next slide. The compiler produces an assembly language file which gets run through an assembler to produce an object file. An object file is compiled code that has not been placed at any particular address. Because of this, other object files or collections of object files known as archives or libraries can easily be mixed in with our code to provide additional functionality for our programs to take advantage of. Finally our code and any other object files we add to our project are run through the linker, which decides where to place everything in memory. The linker produces several files. First is the *.hex file (for PICs) which is the machine language code that will be programmed into the microcontroller. If programming for a PC, this would instead be the *.exe file. In both cases, these files may be referred to as an executable. Other compilers may output a different format executable with a different file suffix. In addition to the executable, a *.map file is also produced, showing where all the various C objects were placed within the microcontroller's memory space. And finally, there is the *.cof file which is used by the debugging tools such as the simulator which we will be using, or things like in circuit emulators and debuggers. The linker itself is controlled by a special file known as a linker script. Many compilers give this file the extension *.lkr, but the C30 compiler, which is based on the GNU tool chain uses *.gld (GNU Linker Document). This file tells the linker how much memory a devices has, where it is allowed to place user code and variables, and where all the hardware specific features are mapped into memory so that it doesn't accidentally use one of those locations.

C Compiler

Compiles C source code into assembly language code

Assembler

Assembles assembly language code into object files

Archiver

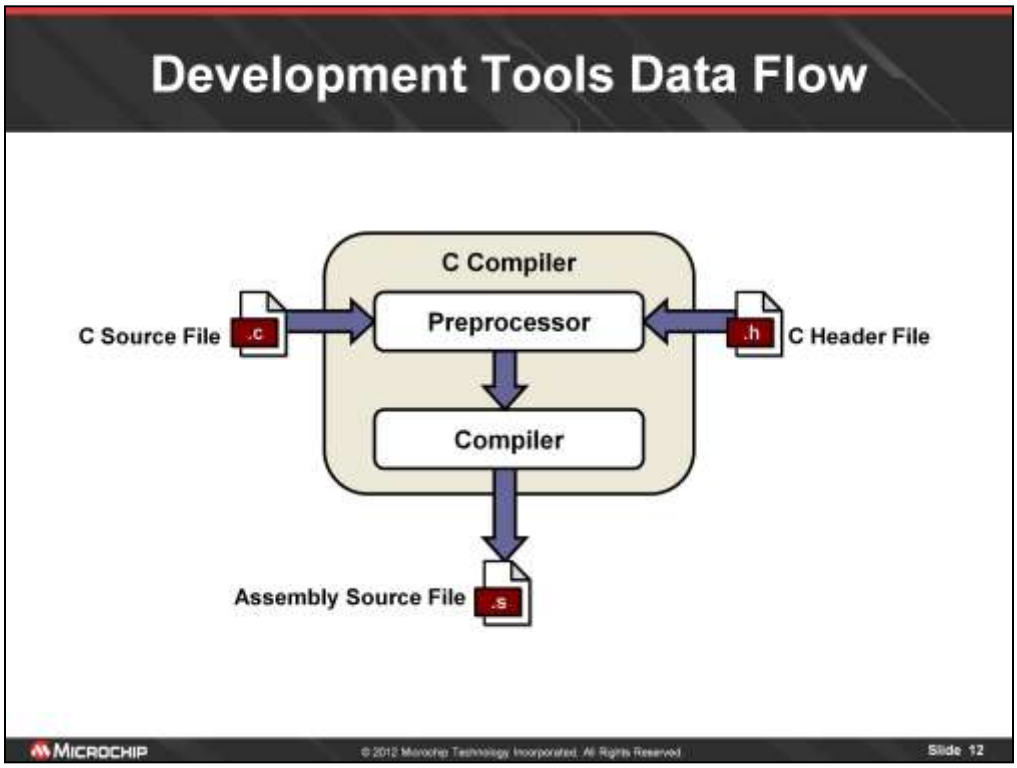
Groups precompiled object files into a library (archive) for convenient organization of reusable code

Linker

Takes relocatable code and arranges it into the microcontroller's memory map according to the guidelines provided in the linker script

Compiler Driver Program

This command line driver program allows application programs to be compiled, assembled and linked in a single step.



In most compilers, there is one step that takes place before the actual compilation of the code. A program known as a preprocessor will run through your source code and follow special instructions known as preprocessor directives that you may place in your code to trigger certain functions of the preprocessor.

One example we will see in more detail later is the inclusion of separate C files known as header files (*.h) which are used to declare things that we will make reference to in our code. There are also directives to do text substitution (#define) that make it easy to enter complicated text by replacing that text with a short label. When the preprocessor runs, every instance of that simple label would be replaced with the more complicated text before the code gets compiled.

We will see many specific examples of the preprocessor directives in action throughout this course.

C Runtime Environment

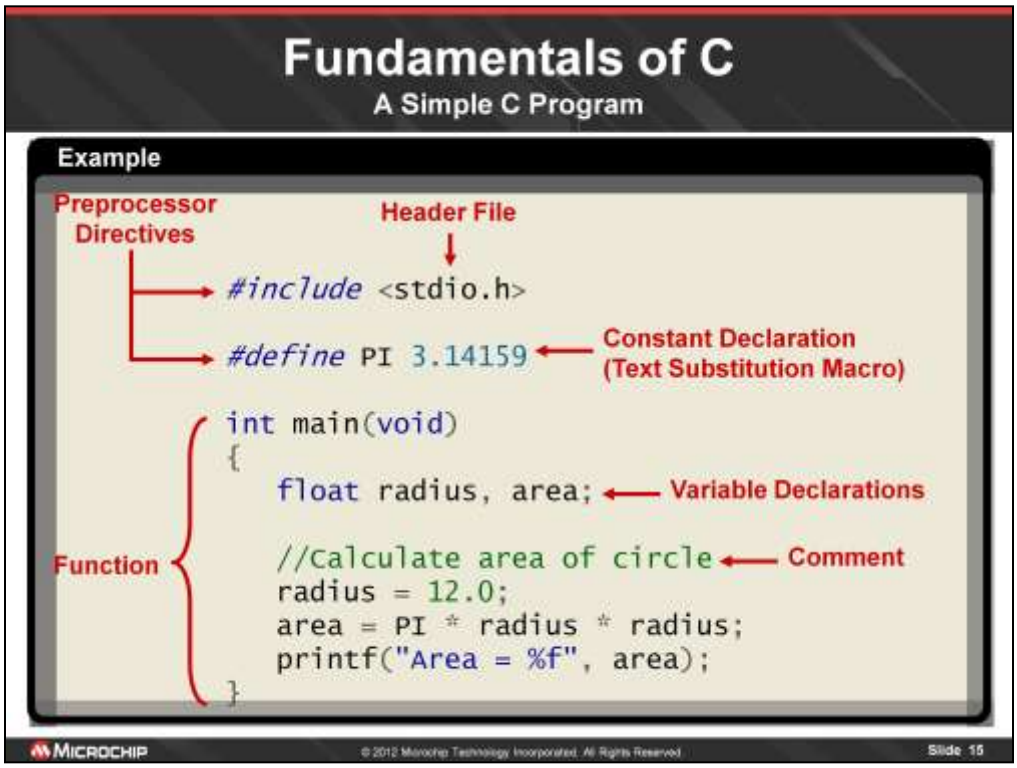
- C Compiler sets up a runtime environment
  - Allocates space for stack
  - Initialize stack pointer
  - Allocates space for heap
  - Copies values from Flash/ROM to variables in RAM that were declared with initial values
  - Clear uninitialized RAM
  - Disable all interrupts
  - Call main() function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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

C Runtime Environment

- Runtime environment setup code is automatically linked into application by most PIC® compiler suites
- *Usually* comes from either (XC16):
  - crt0.s / crt0.o (crt = C RunTime)
  - crt1.s / crt1.o
- User modifiable if absolutely necessary
- Details will be covered in compiler specific classes

© 2012 Microchip Technology Incorporated. All Rights Reserved.



To better understand the details we will be seeing in subsequent slides, it may be helpful to take a look at the big picture first. This is a very simple, yet complete C program. It doesn't use every possible feature of C, but it does illustrate some of the bare minimum of items required for a complete program.

The one item that is 100% required in every C program is a function called "main". It is possible to have any number of other functions in a C program, but they will all be called from within the main function. This is where your code will begin to execute immediately after the various system initialization and housekeeping routines get things setup as dictated by your compiler. Notice the curly braces around the function code below the header. Think of these as meaning "begin" and "end". Curly braces are used throughout C to identify blocks of code that belong together as a single unit.

At the very top of the program is a line that you will see in virtually every C program (though not necessarily in embedded C programs), and that is the inclusion of an external header file that contains declarations of various functions that are part of the standard C library and are used to implement common I/O functions that are not an inherent part of the language itself. We will see much more on the subject of header files later on.

Immediately below that, we use the #define directive to define a text substitution label called PI. When we compile our code, a special program known as the preprocessor will go over the code first and replace each instance of "PI" with the text "3.14159". When the compiler runs later, it will be correctly interpreted as a floating point number. The #define directive can play many roles in a C program, and we will see more detail on how it works later on.

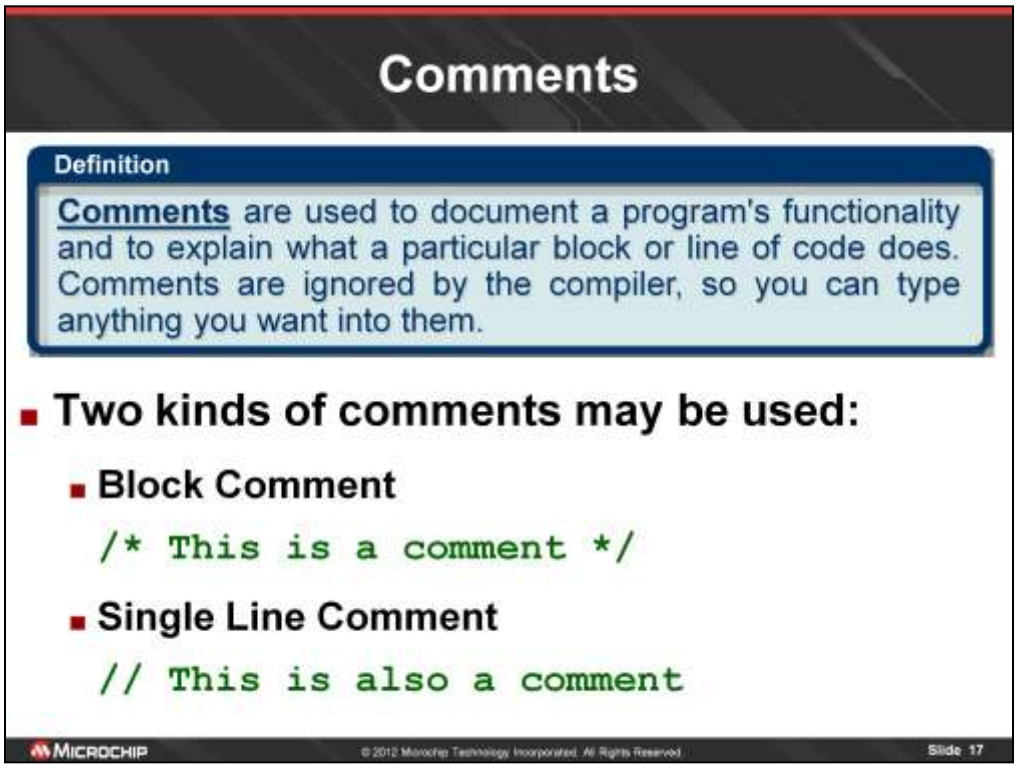
Inside the main function itself, we declare two floating point variables so that we may use them later on in the function. In C, you must declare a variable before it may be used. Notice that the two variables are associated with a data type "float". This is an important point that we will cover in detail shortly.

Below the variables is a comment line. Comments are ignored by the compiler and may contain virtually any text. The style of comment shown here starting with a double slash was actually borrowed from the C++ language. We'll see the traditional C style comments later on in the presentation. In most cases, the C++ style comments shown here are far more useful since they start with the double slash and go to the end of the line.

After the comment, we have some statements that calculate the area of a circle with a radius of 12 and outputs the results to a text window.

MICROCHIP

Comments



MAIN IDEA

•Introduce the idea of a comment

•Understand that comments do not affect the size or speed of their program (only the size of the source file)

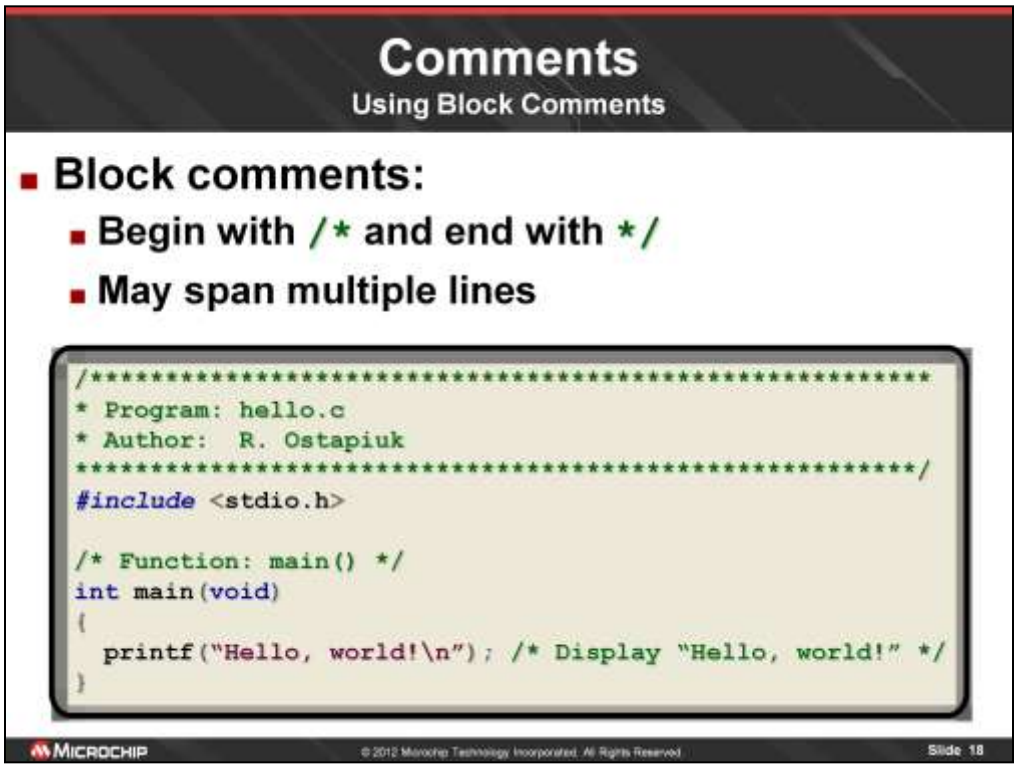
Comments: Comments are used to document a program's functionality and to explain what a particular block or line of code does. Since comments are ignored by the compiler, you can type anything you want into one. It is considered bad programming practice not to comment your code. If you don't learn this lesson today, then in the future when you look at some code you wrote six months or a year ago and can't figure out what you were thinking at the time, you will quickly become a believer in comments.

Comments don't require any memory on the target processor, they only take up space in the text file that contains your source code and sits on your PC. Since they are essentially free, use them liberally!

•**Two kinds of comments may be used**

- •**Block Comment** (Also called a delimited comment)
- •**Single Line Comment**

These two styles of comments will be discussed in detail on the next two slides.



MAIN IDEA

•Understand how to recognize and use block comments.

•Understand that comments do not affect the size or speed of their program (only the size of the source file)

•Block Comments

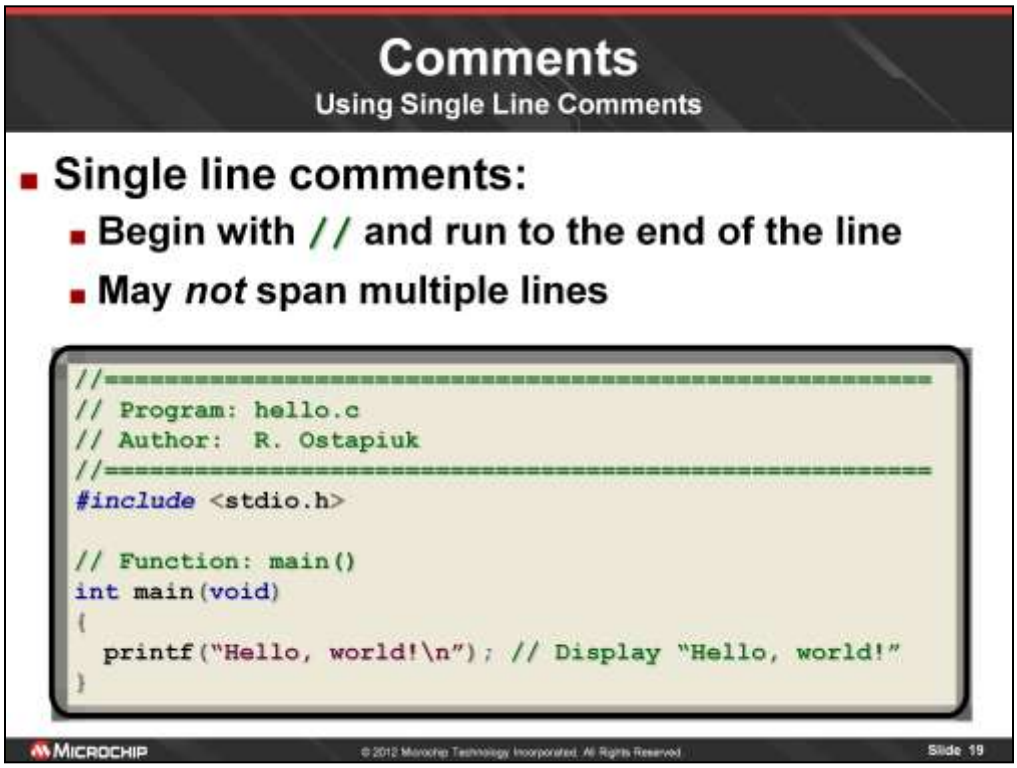
Begin with a ***/*** and end with a ****/***. Anything between these two delimiters will be completely ignored by the compiler. This is true even if the beginning delimiter ***/*** and ending delimiter ****/*** are on different lines.

CODE EXAMPLE

In the example program shown here, we use block comments in two different ways. The first four lines are an example of a comment that spans multiple lines. The opening ***/*** is at the very beginning of line 1 and the closing ****/*** is at the very end of line 4.

On the 7th and 10th lines, both block comments begin and end on their respective lines.

None of the text enclosed within the delimiters has any effect on the size or execution speed of the program.



MAIN IDEA

•Understand how to recognize and use single line comments.

•Understand that comments do not affect the size or speed of their program (only the size of the source file)

•Single Line Comments

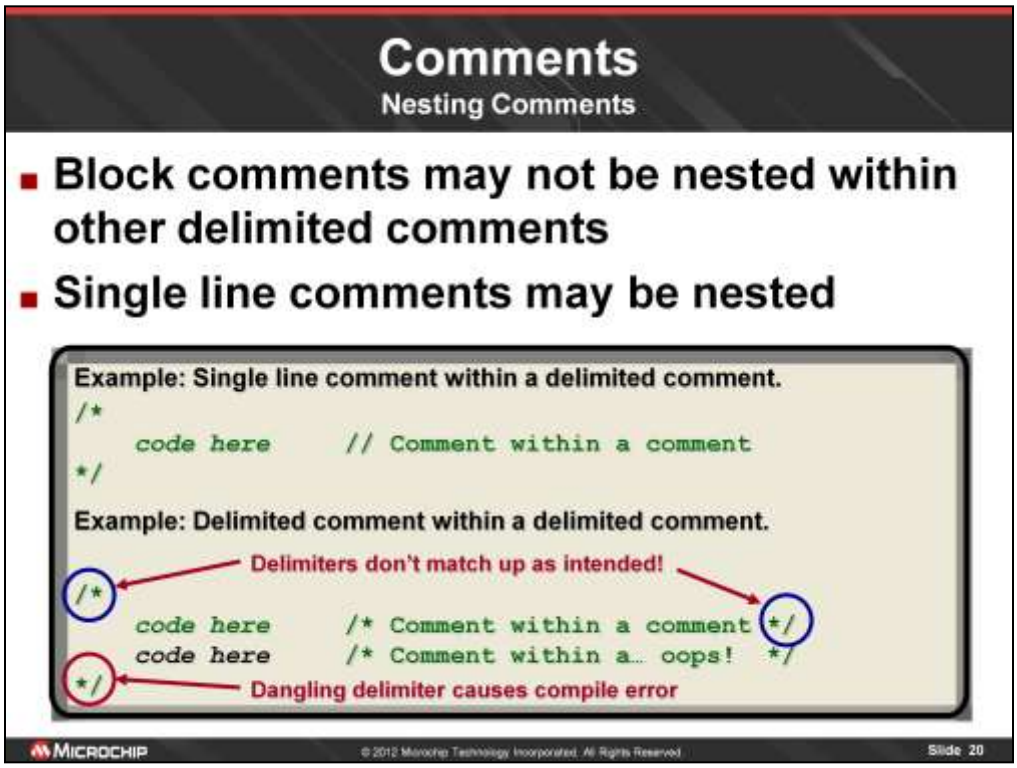
Begin with a ***//*** and run to the end of the line. Anything after the ***//*** on the current line will be completely ignored by the compiler. Since the comment ends at the end of the line, ***//*** style comments may not span multiple lines.

CODE EXAMPLE

Here we have the same code as on the last slide, but all the comments have been changed to the single line variety. By the way, it is perfectly fine to use both styles. In fact there are some big advantages to mixing the styles that we will cover shortly.

Notice that the first four lines now all have a ***//*** in front of them. Each of these lines is now a separate and distinct comment, rather than all being part of a single block comment.

The story is the same with lines 7 and 10. Both comments now start with ***//***, and anything on the line after the ***//*** will be ignored by the compiler.



MAIN IDEA

•Block comments may not be nested within other block comments.

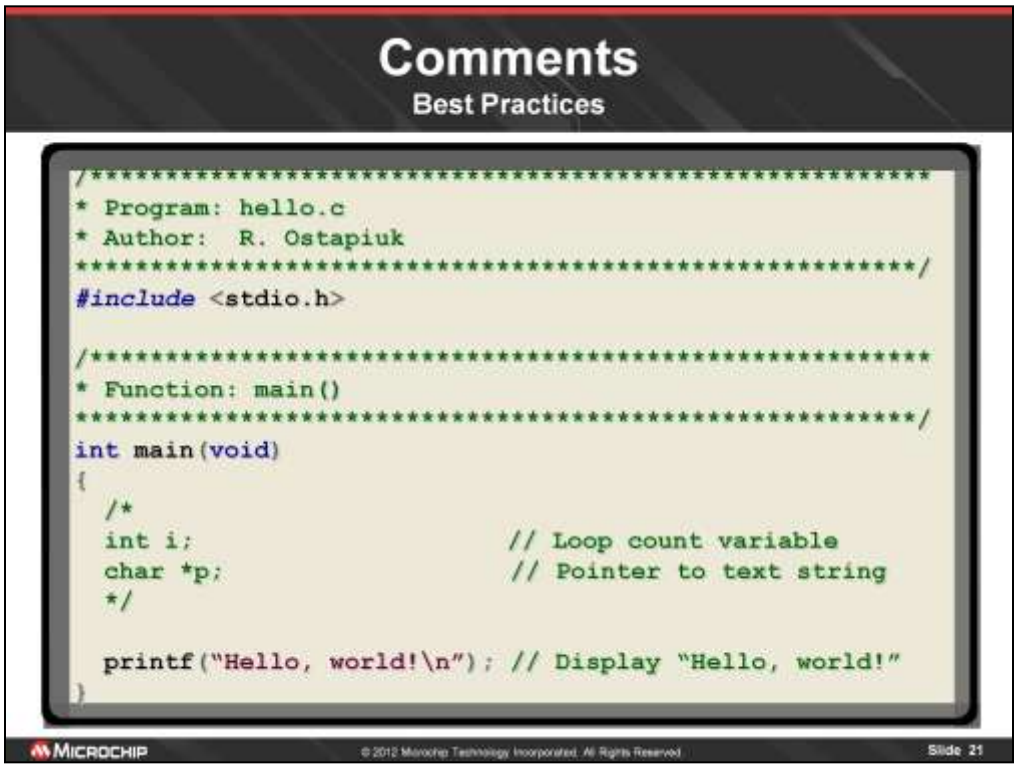
•Block comments may be nested within a single line comment, as long as the closing delimiter is on the same line, or its line is also commented out

•Single line comments may be nested within either type

CODE EXAMPLES

In the first example, we show a single line comment embedded within a block comment. This is a situation that can happen frequently during debugging sessions where you may want to comment out an entire block of code. If you have used single line comments to comment individual lines or sub-blocks, they will coexist happily with the block comments. Since the two styles of comments have completely different beginning and ending symbols, they cannot interfere with each other when used like this.

In the second example, we have two block comments in a block that we are attempting to comment out. However, because the comments on the line are also block style comments, the opening delimiter that we are using to comment out the entire block of code, will find the closing delimiter at the end of the first line's comment and will consider it a match. Because of this, the next line isn't commented out at all, and we are left with a dangling delimiter at the end of the block which will generate a compiler error.



MAIN IDEA

•Use **//** for commenting on an individual line of code

•Use **//** for commenting on sub-blocks (not shown in the example)

•Use **/* */** for block comments and headers

•Use **/* */** to comment out code when debugging

CODE EXAMPLE

How you comment your code is really a matter of personal preference. However, there are some tricks you might want to be aware of before you settle on any one way.

To make your life easier when debugging your code, it is always best to use **//** style comments within a particular code block, whether they are commenting on a single line or explaining the functionality of several lines just above the code. This is because you will then be able to use **/* */** style comments to comment out entire blocks of code while you are debugging. Inside the main function we have two variable definitions, each of which has a comment associated with it. If we aren't using these variables at the moment, we can comment out the entire list by putting a **/*** above the first one and a ***/** below the last one. The **//** comments on the individual lines do not affect the overall comment since they don't share the same syntax with the delimited comments.

NOTE TO STUDENTS: Be forewarned that despite everything I have been telling you about how great it is to comment your code, due to space restrictions, I will not be showing comments in any of the code examples for the rest of this presentation unless there is a very good reason. They will however appear in the handouts for the hands-one exercises and in the source files included on the CD.

MICROCHIP

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);
}
```

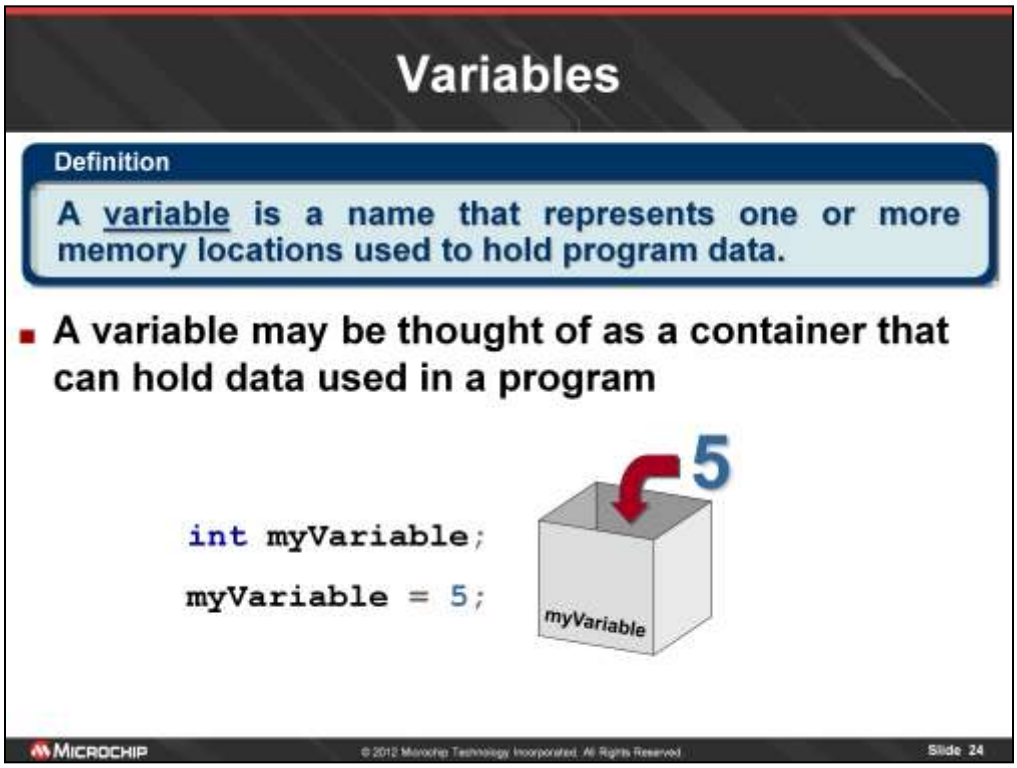
Data Types

Variable Declarations

Variables in use

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

The first aspect of our simple C program we need to discuss are variables and their associated data types. Inside the main function, we declare two variables of type float, and then use them several times throughout the function. Note that the main function itself also has a data type which we will discuss later in the section on functions.



MAIN IDEA

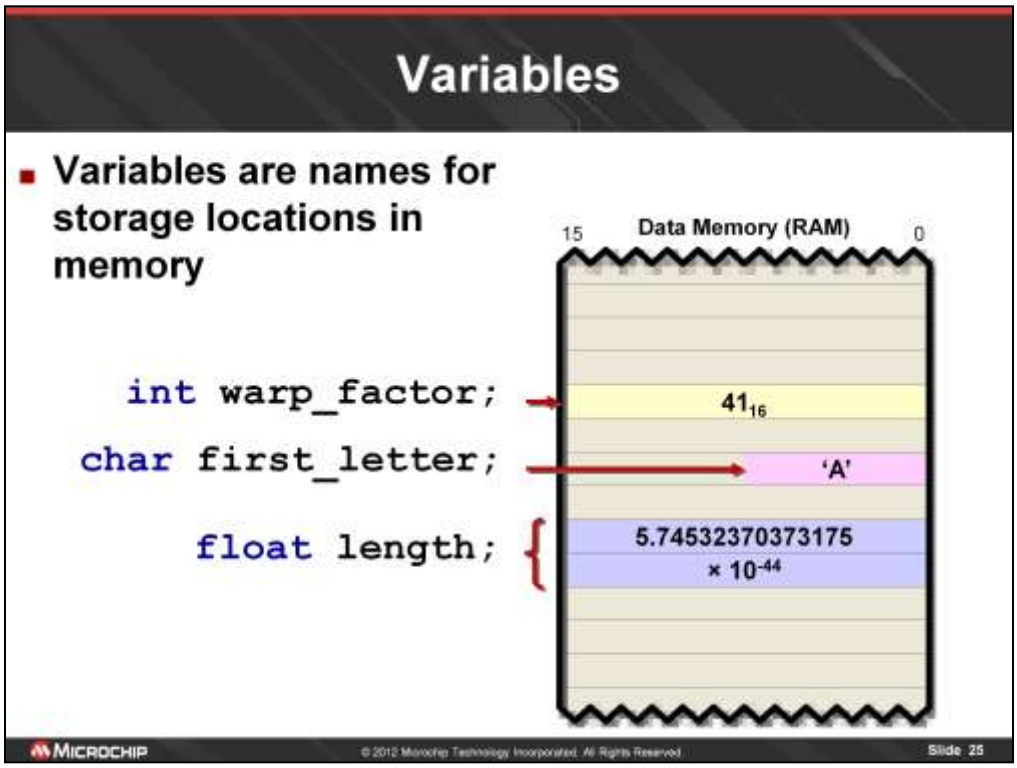
•A variable is something we use to store data

•A variable is a name that represents the memory location(s) used to store data

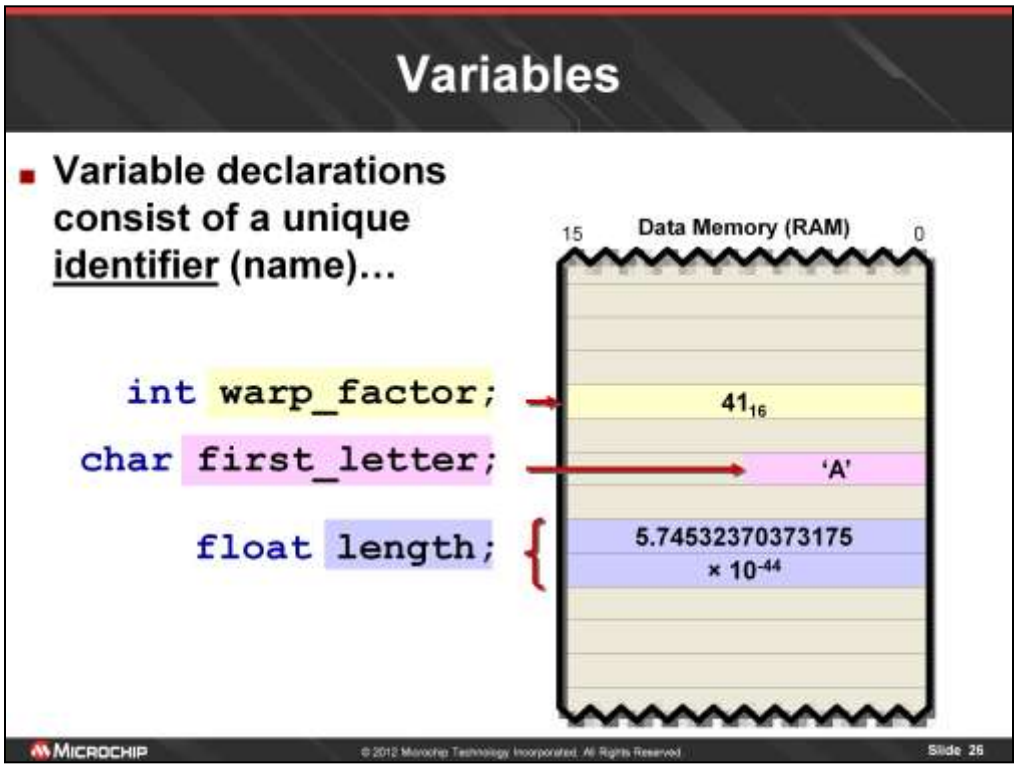
•By using a variable name, we don't care where that data resides. All we need to do is refer to its name in order to access it (store or retrieve)

The code shown here first tells the compiler that we want to create a variable named myVariable.

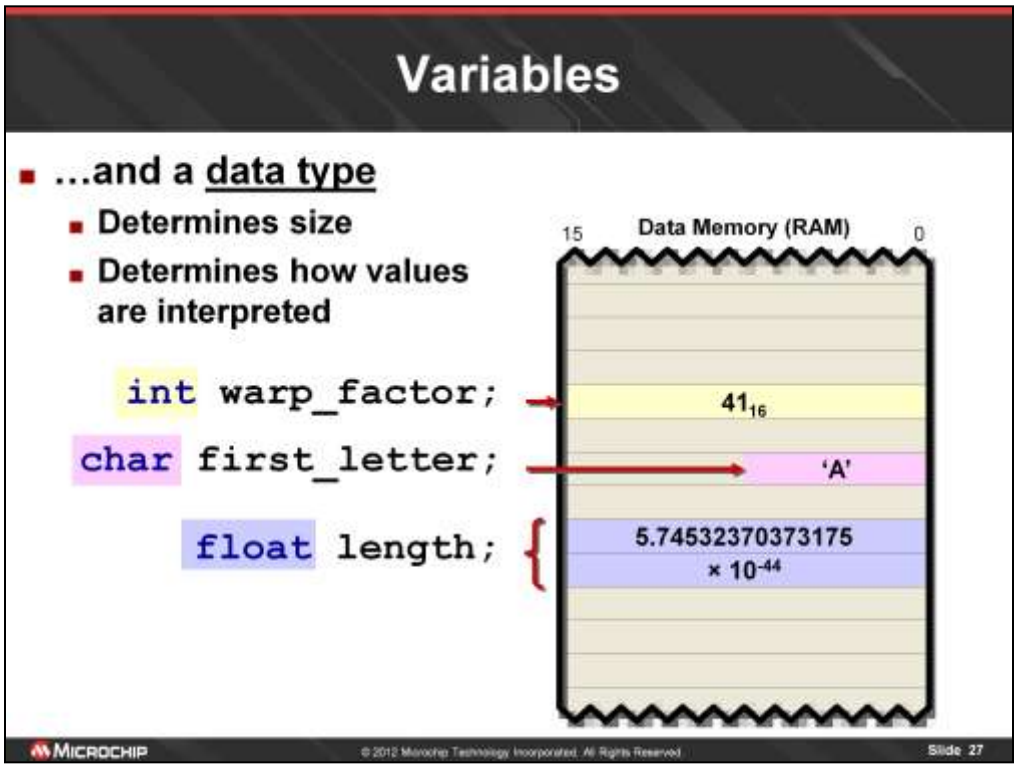
The second line places a value of 5 into the variable myVariable



As in mathematics, a variable is an identifier that is used to represent some specified type of information that can be changed. It can be thought of as a place to store a value in computer memory so that it may be retrieved for later use. More precisely, a variable binds a name (identifier) with the memory location. A value may then be stored in that memory location as a data object, which may be accessed later via the variable name.



Variables need to be declared before they can be used in your program. A variable declaration consists of two parts. The most recognizable part is its identifier (name). The name uniquely identifies a particular location in memory that we can use to store and retrieve data. A variable name must follow all the rules for identifiers that we just discussed. Traditionally, variable names are all lower case in C, though this is not a requirement.



The second part of a variable declaration is its data type. Data types are used to determine how much space is required for a particular variable (how big or small can the variable be?), as well as how the value stored there is to be interpreted. For example, is the value 41 simply the integer value 41? Is it the ASCII representation of the character 'A'? Or, is it the IEEE-754 representation of a floating point number? All of this must be known for the compiler to be able to handle variable values appropriately. Note that on this slide, all the values shown are represented by the zero padded number 41 (hexadecimal) in each of these memory locations:

The int is 0029h = 41_{16}

The char is 29h = 'A' (treated as an 8-bit integer = 41_{16})

The float is 00000029h = 5.74532370373175e-44 (in IEEE-754 floating point format)

So the type not only determines how much space a variable will occupy in memory, it also affects how it is encoded in memory and interpreted in the program.

Identifiers

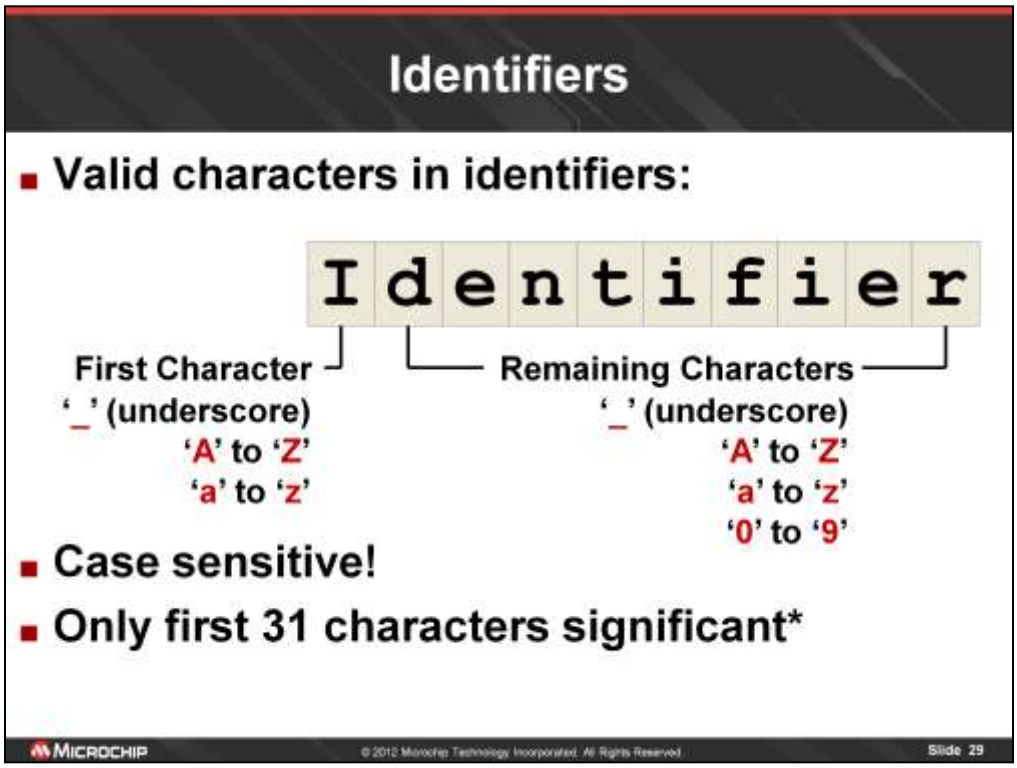
- Names given to program elements:
  - Variables, Functions, Arrays, Other elements

Example of Identifiers in a Program

```
#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);
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 28

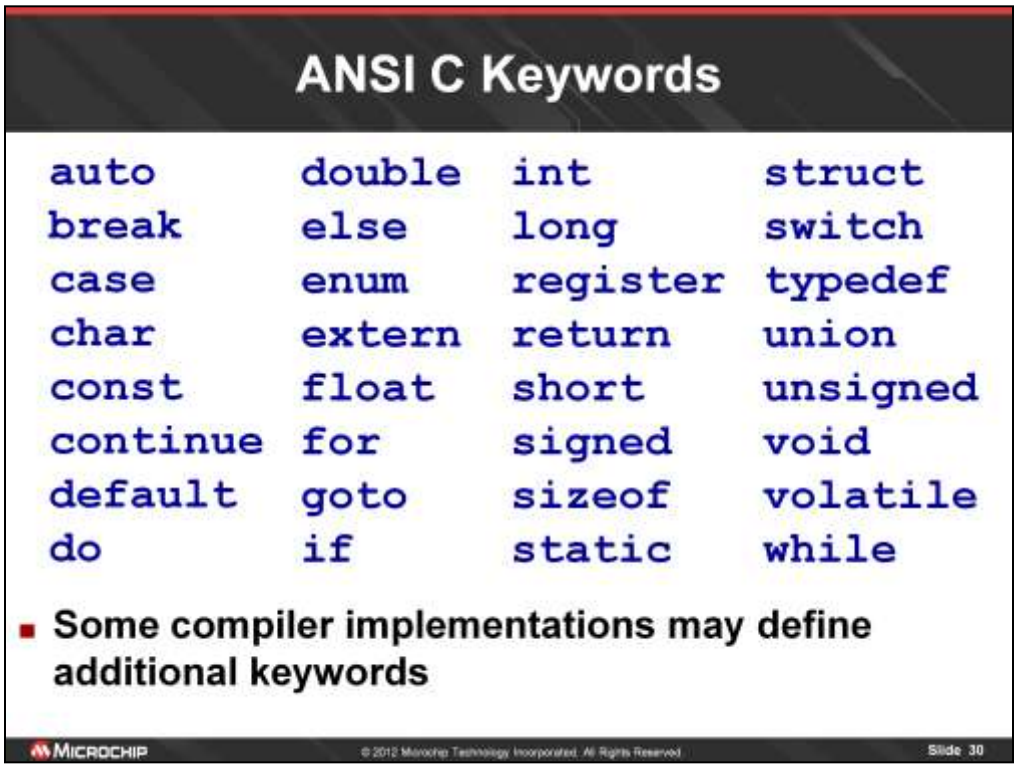
Several times we have made reference to the name of a variable. Technically, the name of something in C is known as an identifier. Identifiers are names that we give to a wide variety of program elements such as variables, functions, arrays and a whole host of other objects that we will discuss later. These names are what we will use in our code to refer to these objects whenever we want to make use of them.



Identifiers in C must be strings of characters from the valid C character set which includes all the letters of the English alphabet (both upper and lower case), the numbers 0 through 9 and the underscore. The first character of an identifier is further restricted in that numbers may not be used. Note that under no circumstances may an identifier contain a space.

All identifiers are case sensitive, so you can have two identical identifiers with the exception of the case of the first character, and they will be considered to completely different identifiers.

Also, an identifier may be any length, but only the first 31 characters are significant. So, if you have a 32 character identifiers where all characters but the 32nd are the same, then they will be considered the same identifier. This is true according to the ANSI C standard, but most modern compilers provide many more significant characters than 31.



While we have extensive leeway with respect to what names we can use for identifiers, there are some words that cannot under any circumstances be used as identifiers. These are known as “keywords”, and they all have special meanings within the context of a C program.

These 32 keywords have specific meanings in ANSI C, and as such, may not be used for anything other than for that which they are defined. In any ANSI C implementation, these keywords will have the same meaning. However, an any compiler that deviates from ANSI C, some keywords may have slightly different meanings.

Throughout this class, we will be introduced to the special meanings of all these keywords.

MPLAB-C30 NOTE: The GCC compiler, upon which MPLAB-C30 is based, defines 12 additional keywords. These will be covered in part 2 of the C30 class.

MPLAB-C18 NOTE: The MPLAB-C18 compiler defines 5 additional keywords. These will be covered in part 2 of the C18 class.

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-XC16 `int` is 16-bits
- MPLAB-XC8 `int` is 16-bits
- MPLAB XC32 `int` is 16-bits
- CCS PCB, PCM & PCH `int` is 8-bits

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 31

When you create variables, the compiler needs to know how much memory should be used and how the values should be handled in calculations. This need is addressed by defining data types which can be associated with any variable.

There are four fundamental data types in the C programming language. The first two are the integer data types and the second two are the floating point data types.

char = character, can be used to store the ASCII value of a character, but more frequently it is used simply for an 8-bit integer value. On an 8-bit PIC, this is the most efficient data type to use since the architecture can operate directly on 8-bit values.

int = integer, is a general purpose integer value, which may be used efficiently on the 16-bit PIC devices. On most compilers, this data type is 16-bits in size, but some, like the CCS compilers for the 8-bit PIC devices define int as an 8-bit value, to match the natural size of the ALU.

float and double both work the same way, but double can obviously hold a larger number. Both of these are expressed as a base and an exponent, giving them a very large range of numbers.

There are two more data types: void and enum, which will be discussed later.

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}	$2^{31}-1$	32
`unsigned long long` *`int`*	0	$2^{64}-1$	64
`long long` *`int`*, `signed long long` *`int`*	-2^{63}	$2^{63}-1$	64

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 32

The base data types may be modified by the prefixes signed, unsigned, short and long. These modifications change either the range of values that can be represented by the data type, or they change whether or not negative numbers can be represented.

Note that wherever int is in italics and grey, it is optional. For example, when declaring a variable, you could simply define it as a long, rather than a long int. The int in all these cases is implied.

Some compilers support a “short long” which in C18 is 24-bits. It is not supported in C30.

Also short long = 24-bit

Data Type Qualifiers

Modified Floating Point Types

Qualified Type	Absolute Min	Absolute Max	Bits
`float`	$\pm \sim 10^{-44.85}$	$\pm \sim 10^{38.53}$	32
`double` [1]	$\pm \sim 10^{-44.85}$	$\pm \sim 10^{38.53}$	32
`long double`	$\pm \sim 10^{-323.3}$	$\pm \sim 10^{308.3}$	64

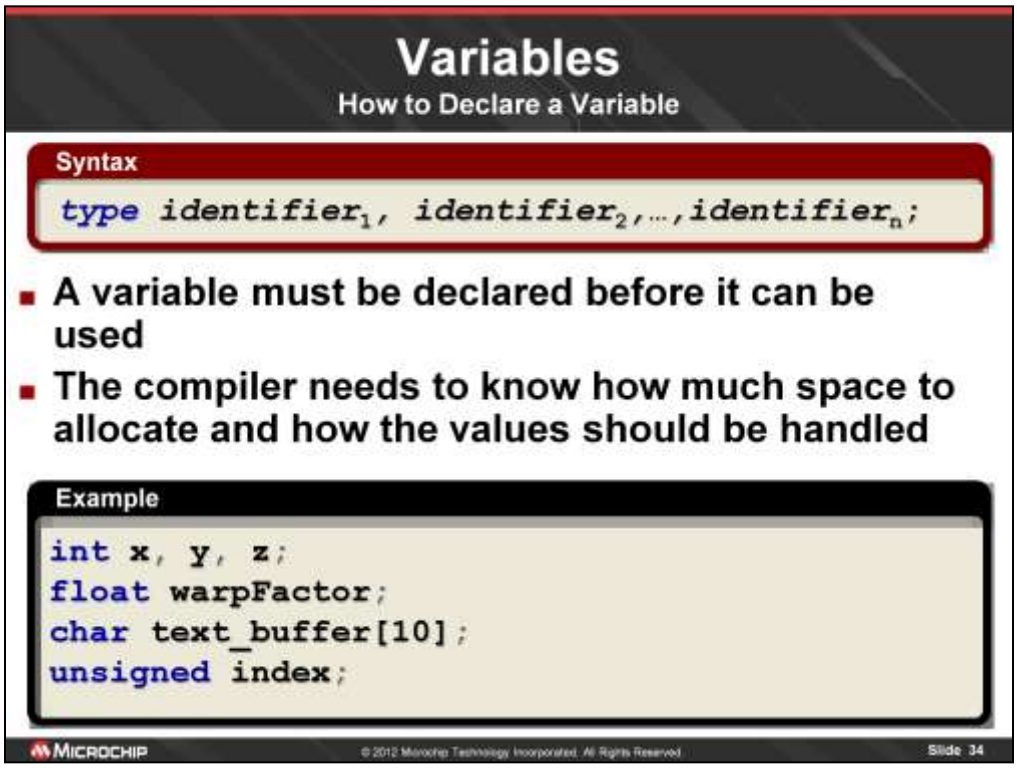
XC16: [1]`double` is equivalent to `long double` if `-fno-short-double` is used

XC16 Uses the IEEE-754 Floating Point Format

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 33

Most compilers define float and double as shown above, while some allow long to be applied to float and double. This table is shows the data type definitions for the MPLAB-C30 compiler. MPLAB-C18 doesn't allow long double, so float and double are always equivalent.

Floating point numbers are stored using the IEEE-754 format (or a modified version thereof).



MAIN IDEA

•When you declare a variable, you prefix it with its data type

•If you want to declare multiple variables of the same type, you can declare them in a comma delimited list after the type

•Variables must be declared before they can be used. If you do not declare the variable and try to use it in your code, the compiler will generate an error.

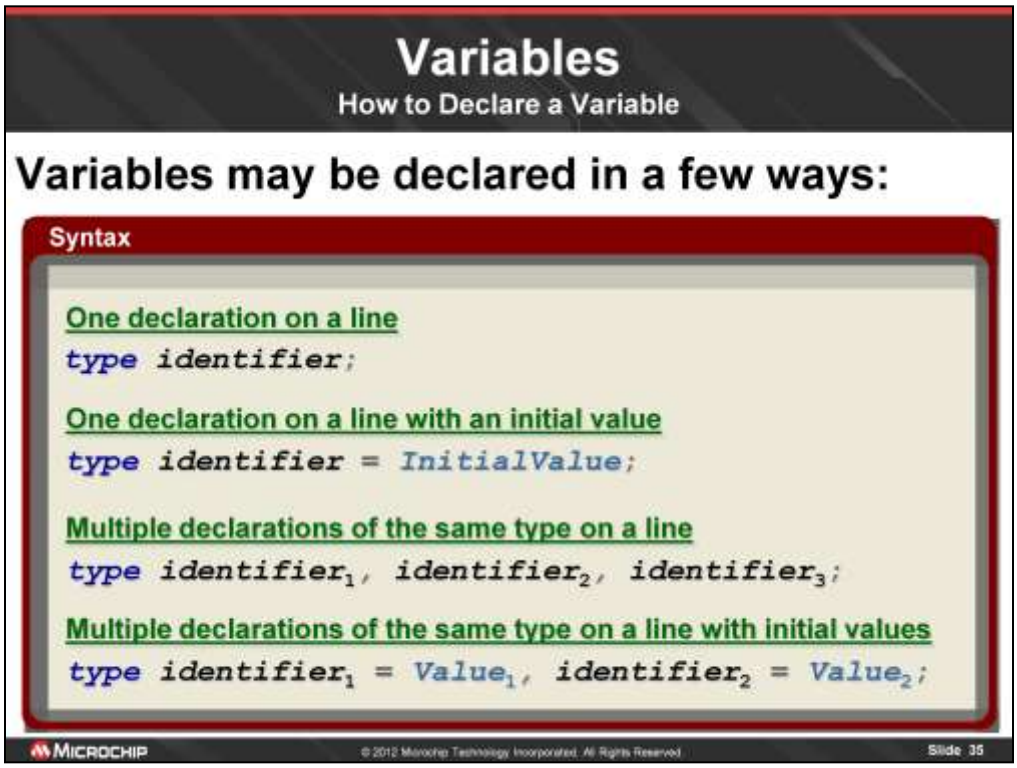
CODE EXAMPLES

•The first example declares three variables named x, y and z, all of which are type int

•The second example declares a float variable called warpFactor

•The third example declares an array with 10 elements called text_buffer (we will discuss arrays later)

•The last one declares an unsigned int called index. Remember, when declaring modified int variables, you do not need to include the keyword int. The modifier alone will suffice.



MAIN IDEA

•There are several ways to declare variables

•Variables may be initialized when they are declared

•Multiple variables may be declared on the same line

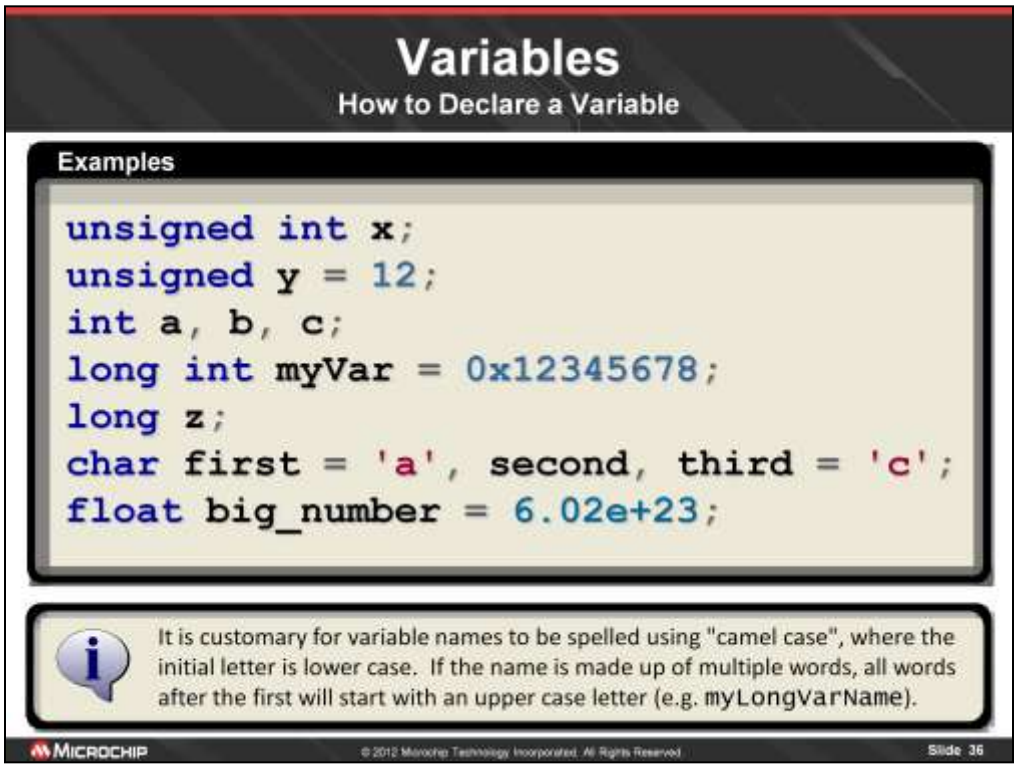
Here we can see an expanded list of the various ways you can declare variables.

The first one we saw as an example on the previous slide. It simply declares one variable.

The second one though is new. Here we are declaring a single variable as in the first one, but this time we are providing an initialization value. When our code begins execution, this variable will automatically be assigned the value that you specify.

The third is what we saw on the previous slide where multiple variables of the same type are declared on a single line.

And finally on the last line, we declare multiple variables of the same type all with their own initialization values.



Here are a few examples of the different ways variables can be declared.

The first one shows the full syntax for creating an unsigned int.

The second one also declares an unsigned int, but uses the shorter notation. It also declares that the variable should be automatically initialized with a value of 12 by the compiler.

The third one declares three ints.

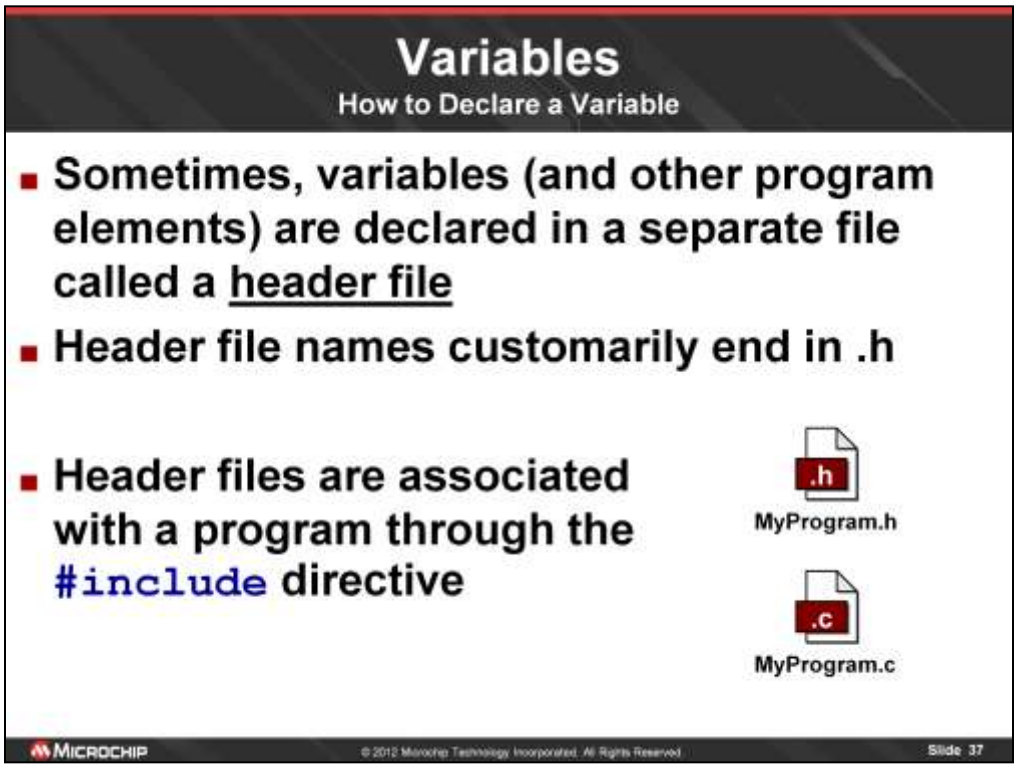
The fourth declares a long int using the full syntax, and initializes it with a value of 0x12345678

The fifth one declares a long variable by itself.

The sixth one declares three char variables and initializes two of them (the one called "second" will be initialized to 0 by the compiler)

The last one declares a float with an initial value of 6.02×10^{23}

On a side note, it is customary to use "camel case" when naming variables that include more than one word as part of their name. Traditionally, variables start with a lower case letter, but subsequent words in the name start with an upper case letter. This makes it easier to distinguish individual words in the name since spaces aren't allowed. Some programmers hate camel case, and use underscore characters in the name to separate words. They are all acceptable, so it is purely a matter of preference which one you decide to use. However, it is considered good programming practice to pick one style and stick with it.



MAIN IDEA

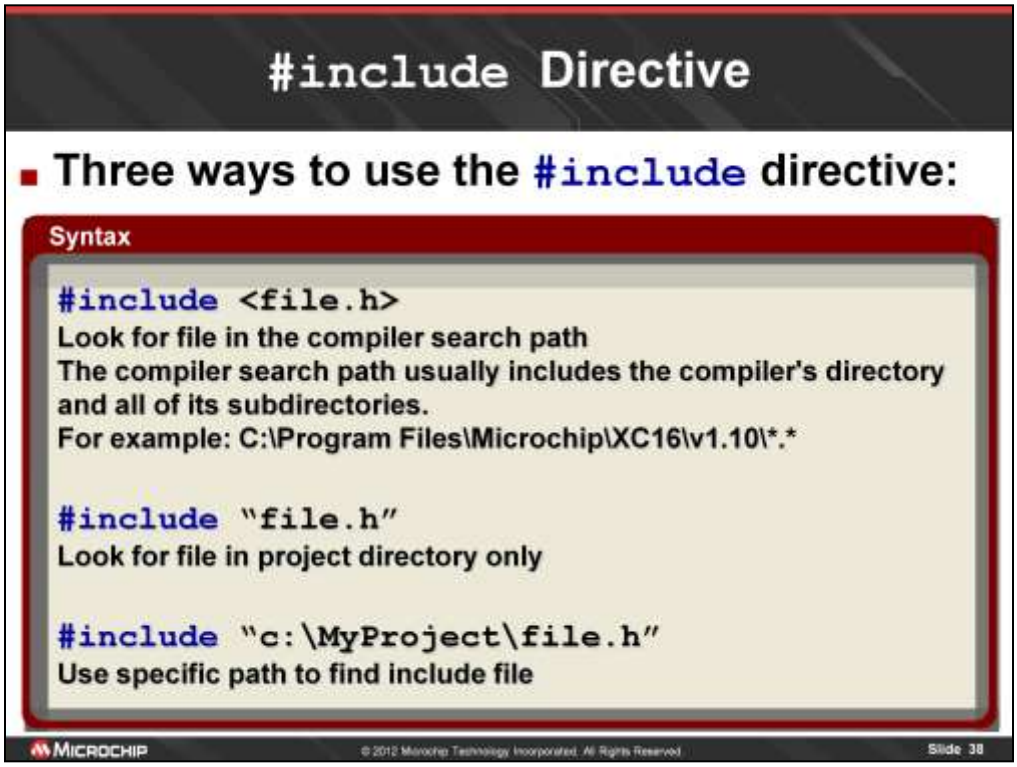
•Variables may be declared in a separate file known as a header file

•Header files are "imported" through the #include directive

•Header files are ordinary C source files, though they should only include declarations – no actual code

Oftentimes, a C program will include a large number of variables. In order to promote modularity and organization, many programmers will declare their variables in a separate file known as a header file. There is nothing special about this file, it is just like any C file, but it uses the file extension *.h. However, you should only put declarations in this file. No actual code should ever be placed in a header file.

To associate this file with your source code, you use the #include preprocessor directive in your C source file.

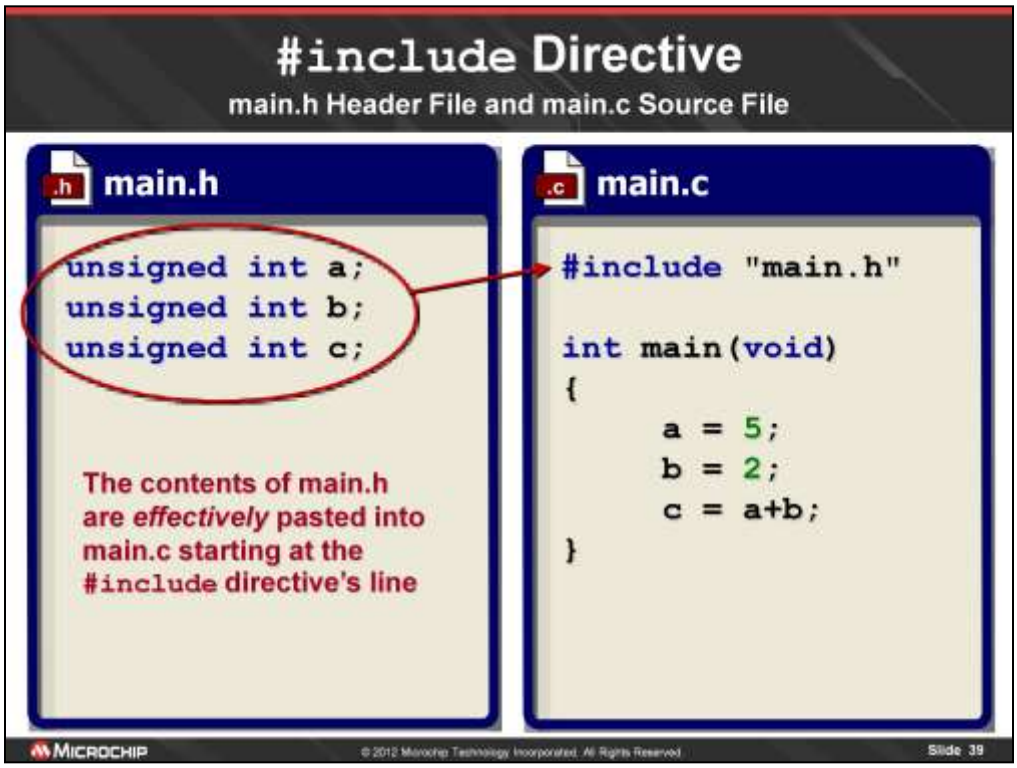


There are three ways to use #include. These are specific to the Microchip compilers (C30 and C18) but are also true for most other compilers.

The first way is to specify the file name to be included in angle brackets (less than and greater than symbols). This tells the preprocessor to look for the file in the compiler's search path.

The second method is similar to the first, but the file name is enclosed in double quotes. In this case, the preprocessor looks for the file only in the project directory.

The third method is like the second, but a full path name is specified, telling the preprocessor exactly where to look for the file.



Here we can see a typical use of an include file. We have two files, main.h and main.c. All of the variable declarations for main.c have been placed in the header file main.c. We then tell the compiler to look in main.h for the declarations by using the #include directive. Because the filename main.h is in angle brackets, the compiler will look in the compiler's search path (usually just the compiler's installation directory) until it finds the file. Once it has found main.h, it effectively copies and pastes the entire file into main.c starting at the line where it was included. The preprocessor doesn't actually copy the contents, but the end result is the same as if it did.

#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

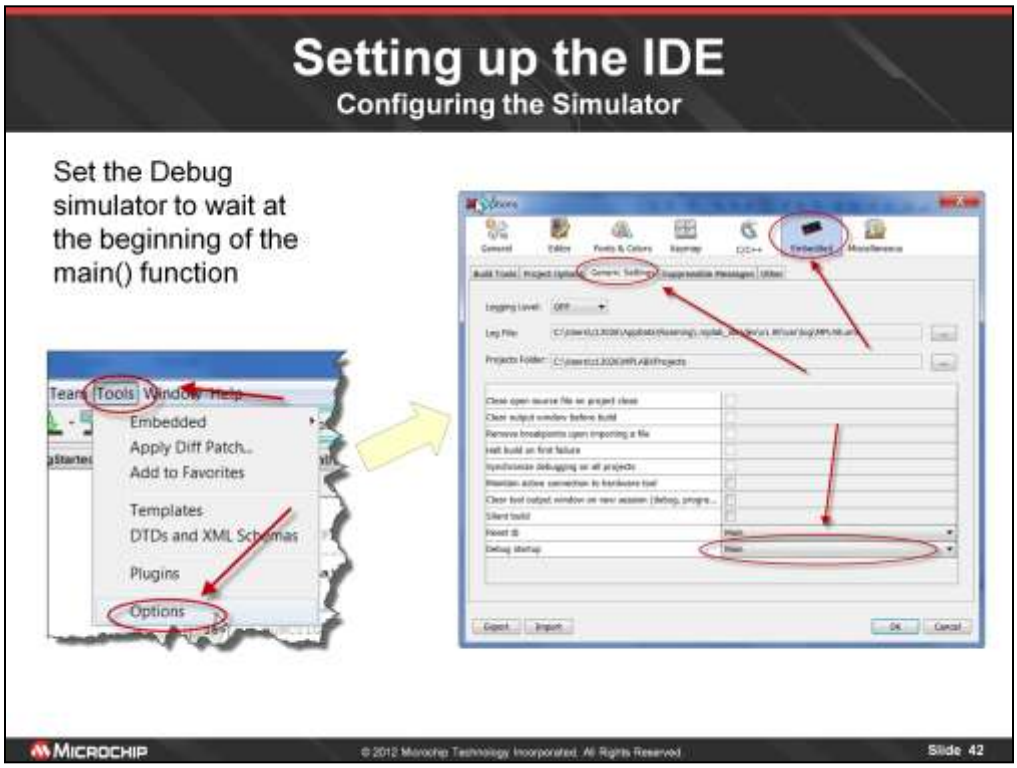
```
unsigned int a;
unsigned int b;
unsigned int c;

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

Equivalent main.c file without #include

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 40

This main.c file is the equivalent of the separate main.c and main.h files on the previous page. Both of these files will be handled in the exact same way.

MICROCHIP
Taking MPLAB® X for a Test Drive
Setting up the IDE for the Labs



The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.

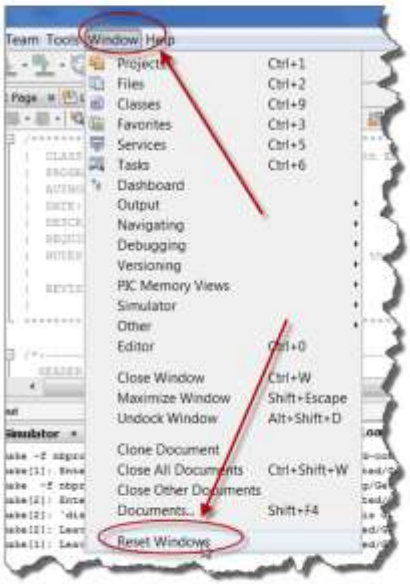
Resetting MPLAB®X windows

As you will see MPLABX has numerous adjustable windows. New MPLABX users can get a little confused about where and how the set the windows.

If you get confused

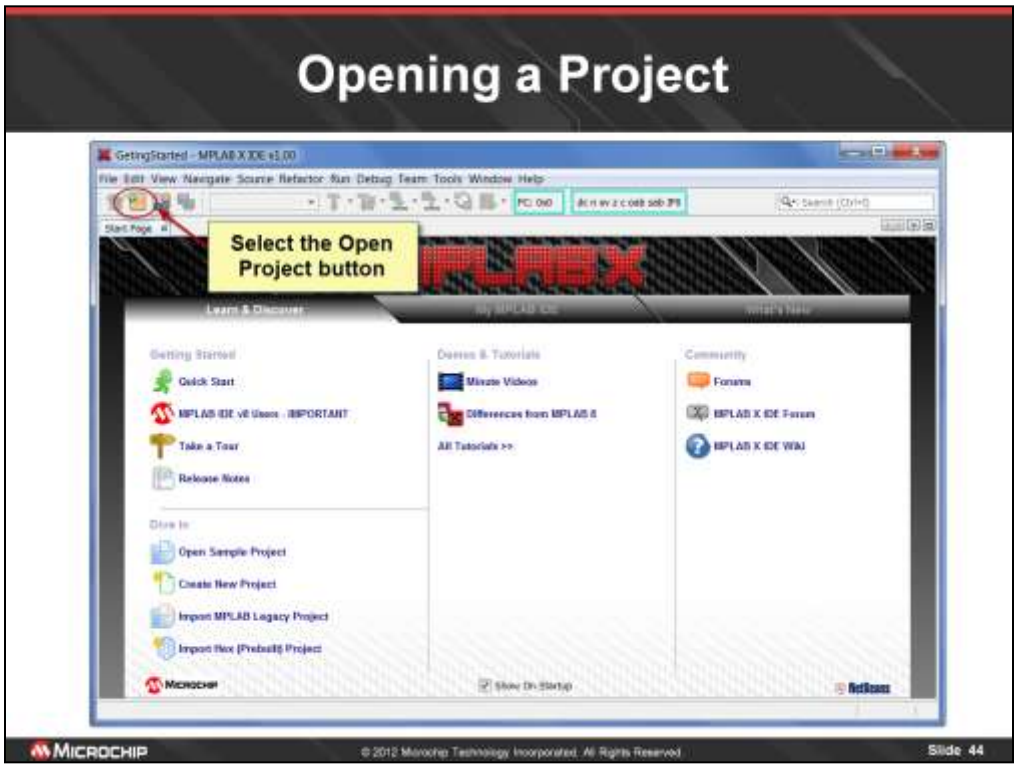
Windows -> Reset Window

Restores MPLABX Windows back to their original locations

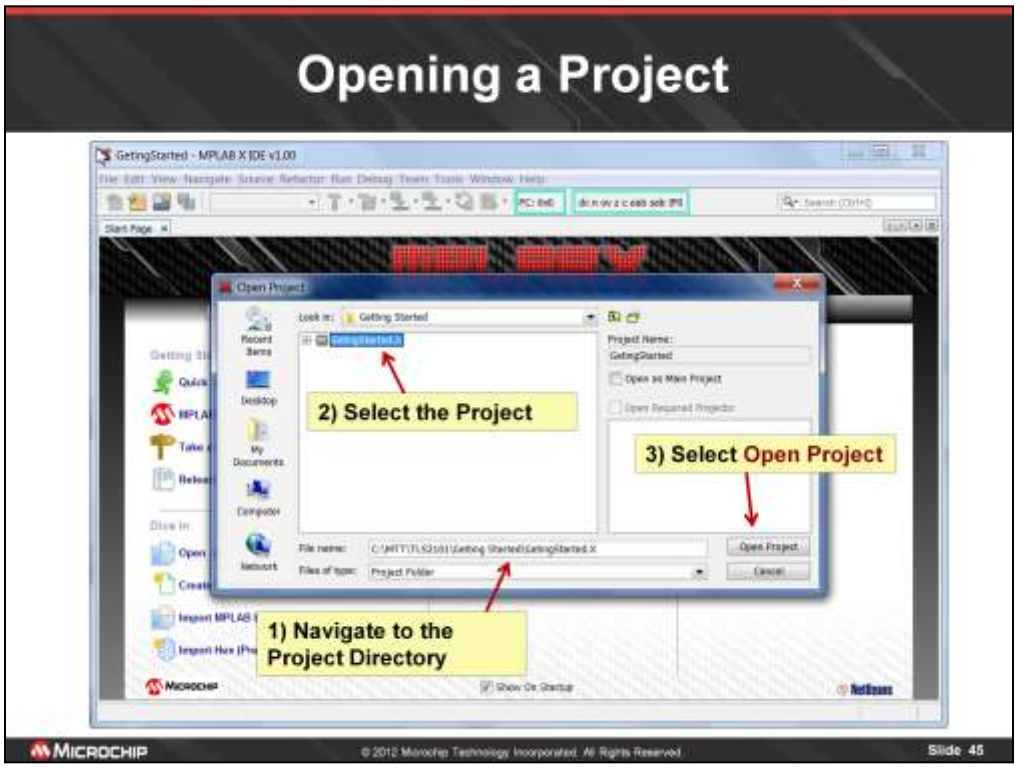


© 2012 Microchip Technology Incorporated. All Rights Reserved.

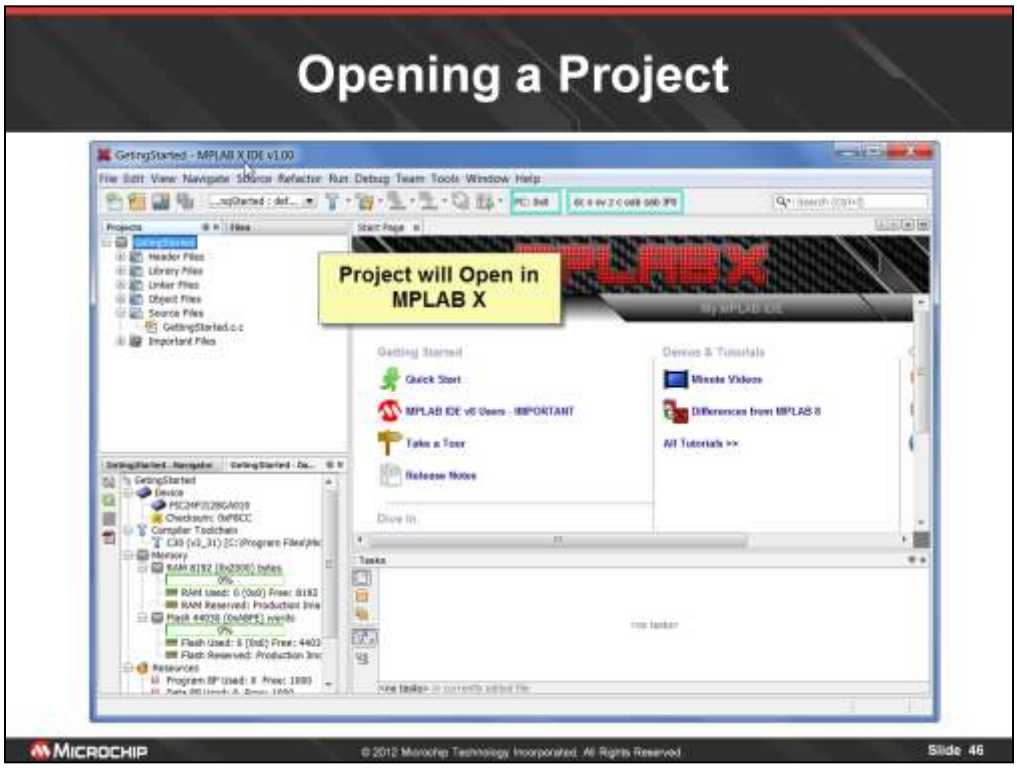
The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.



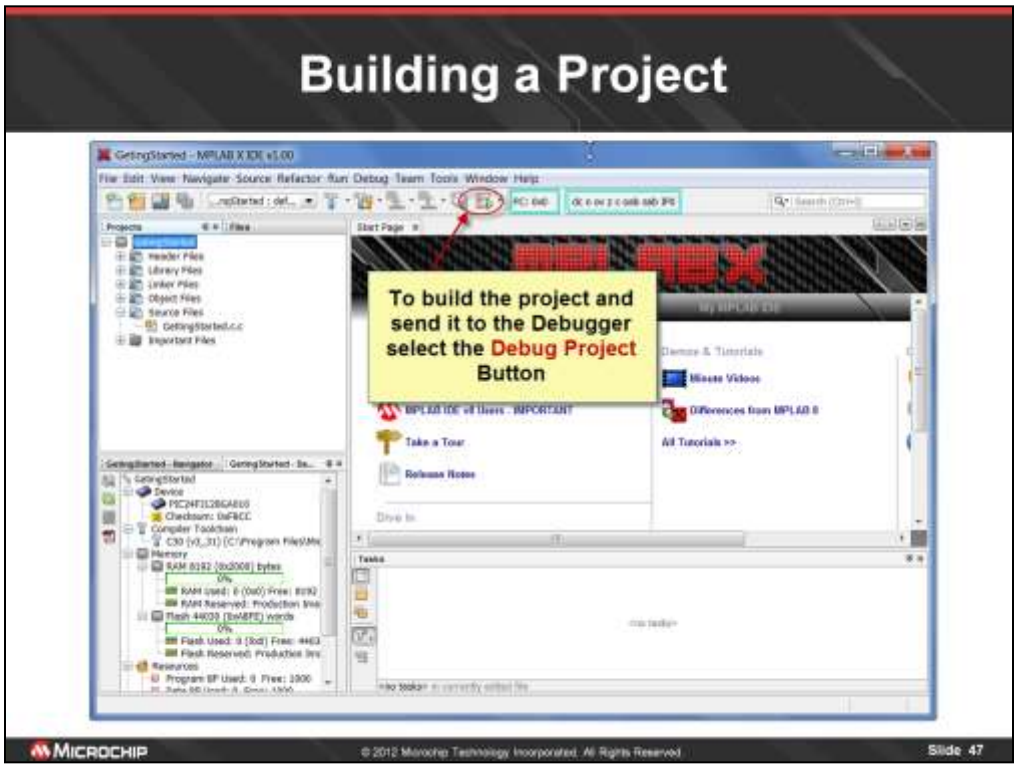
The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.



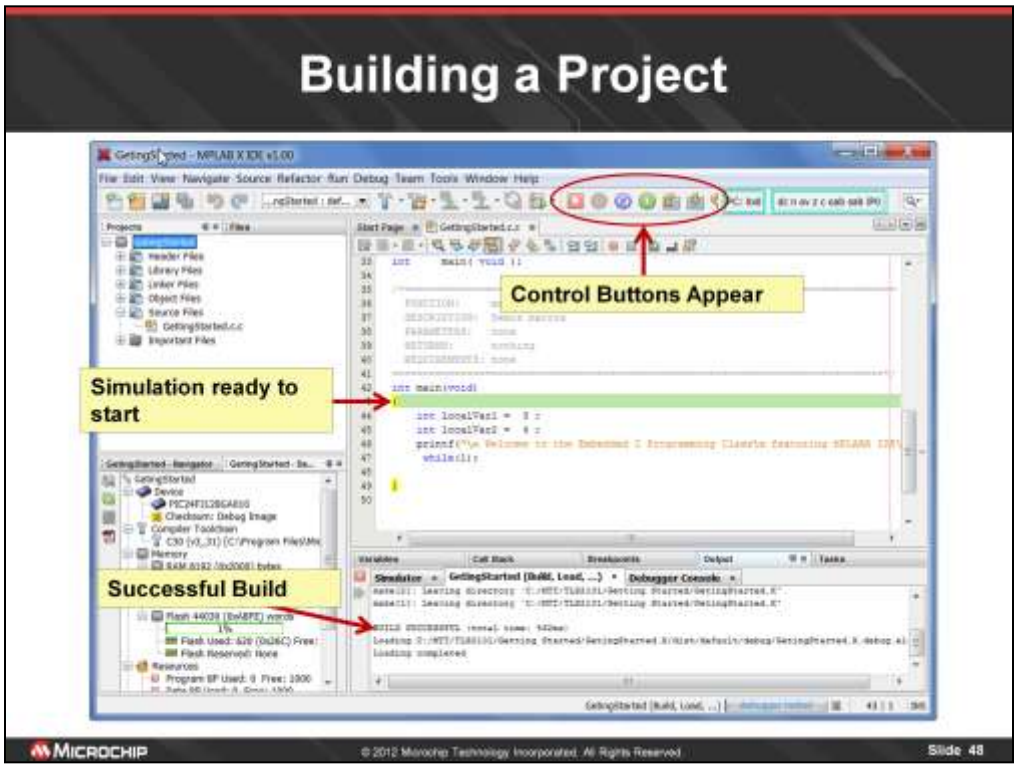
The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.



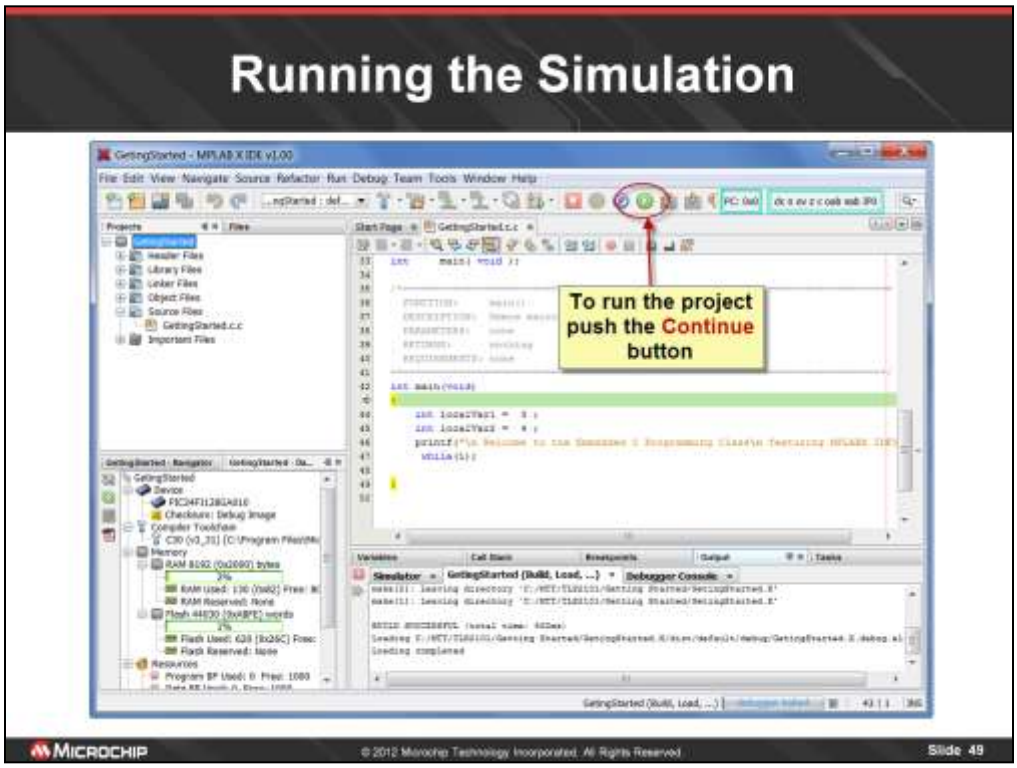
The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.



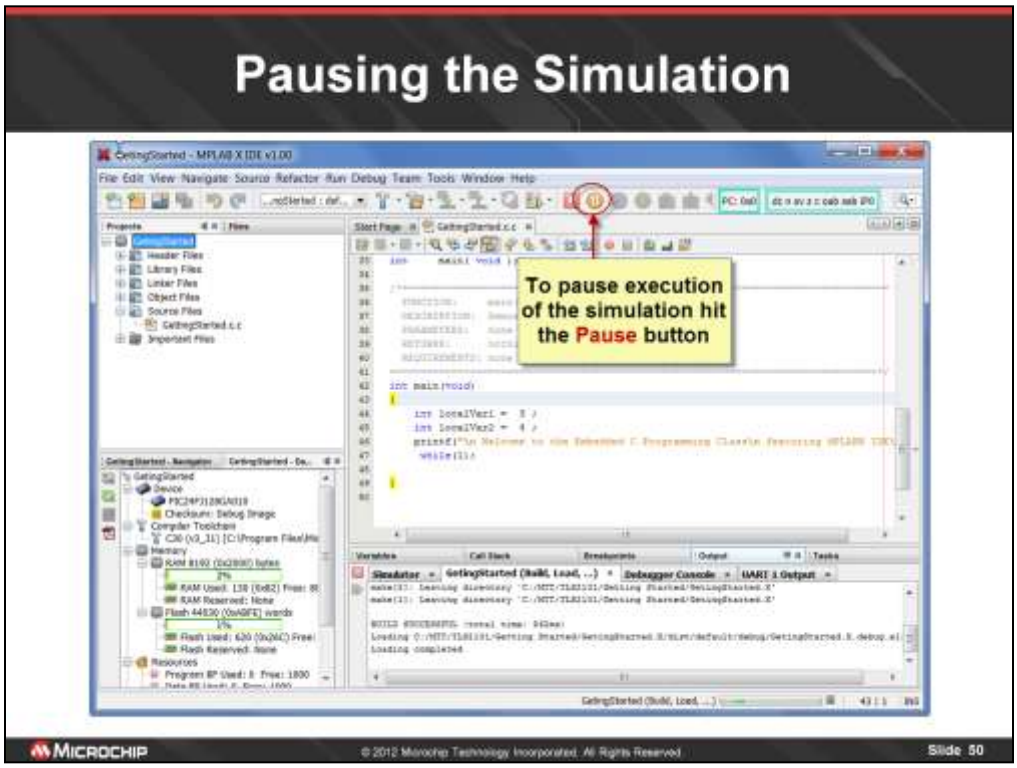
The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.



The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.



The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.



The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.

Windows used in TLS2101 Labs

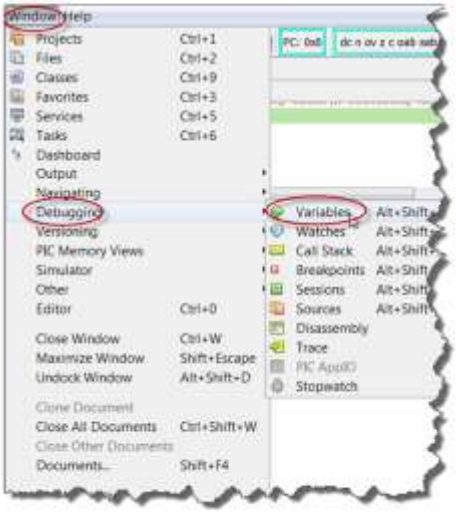
Variables Window

Variable Window displays a particular set of program variables

To Open the Variables window:

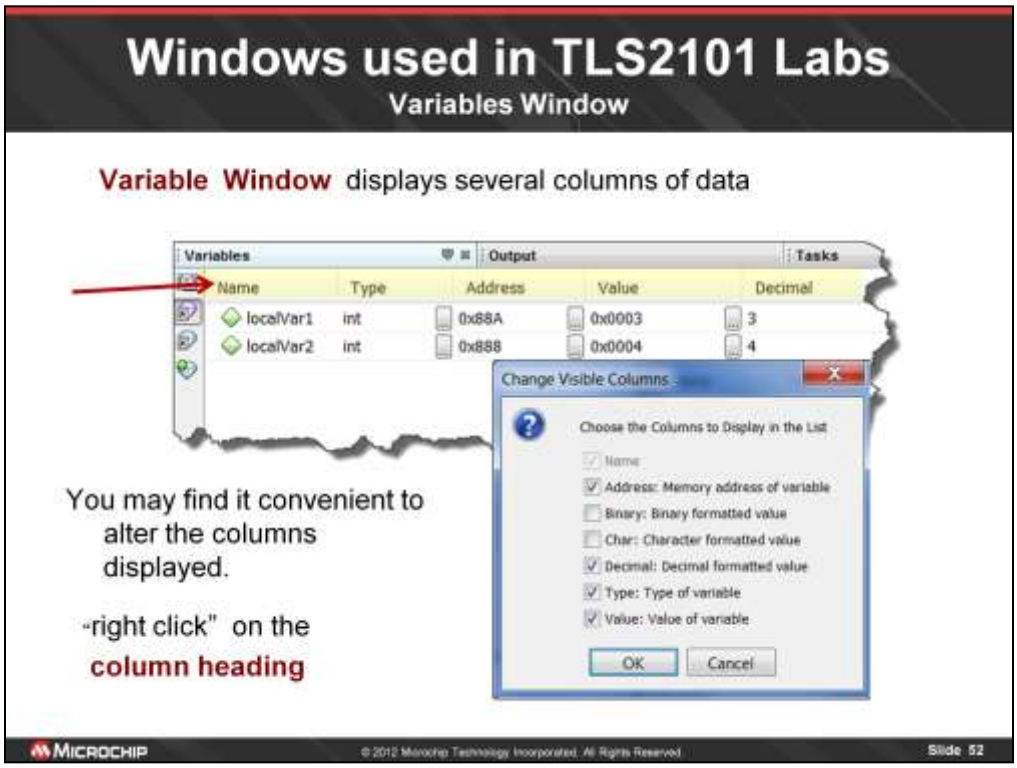
Select:

Windows->Debugging->Variables



© 2012 Microchip Technology Incorporated. All Rights Reserved.

The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.



The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.

Windows used in TLS2101 Labs

UART1 Output

UART1 Output Window prints out text from C programs

To clear this window:

Right click *inside* of the window then select Clear

© 2012 Microchip Technology Incorporated. All Rights Reserved.

The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.

Windows used in TLS2101 Labs

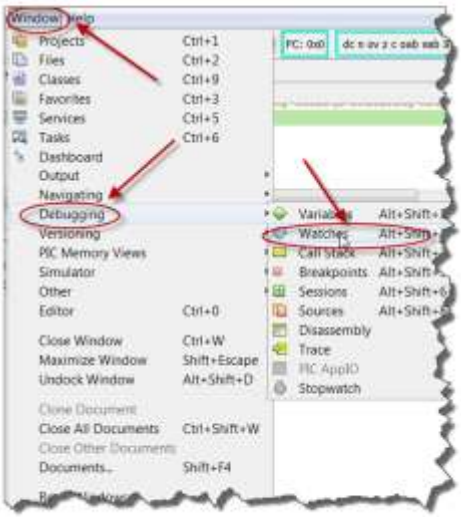
Watches Window

Watches Window is similar to the Variables window but displays a different set of data

To Open the Watches Window:

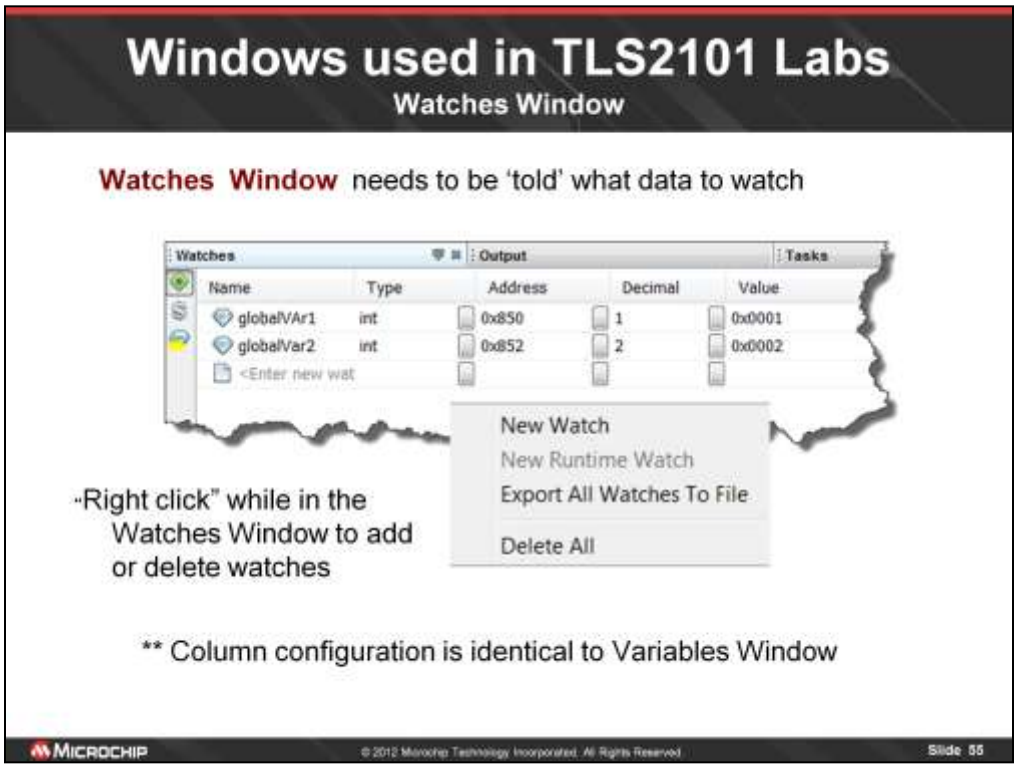
Select:

Windows->Debugging->Watches

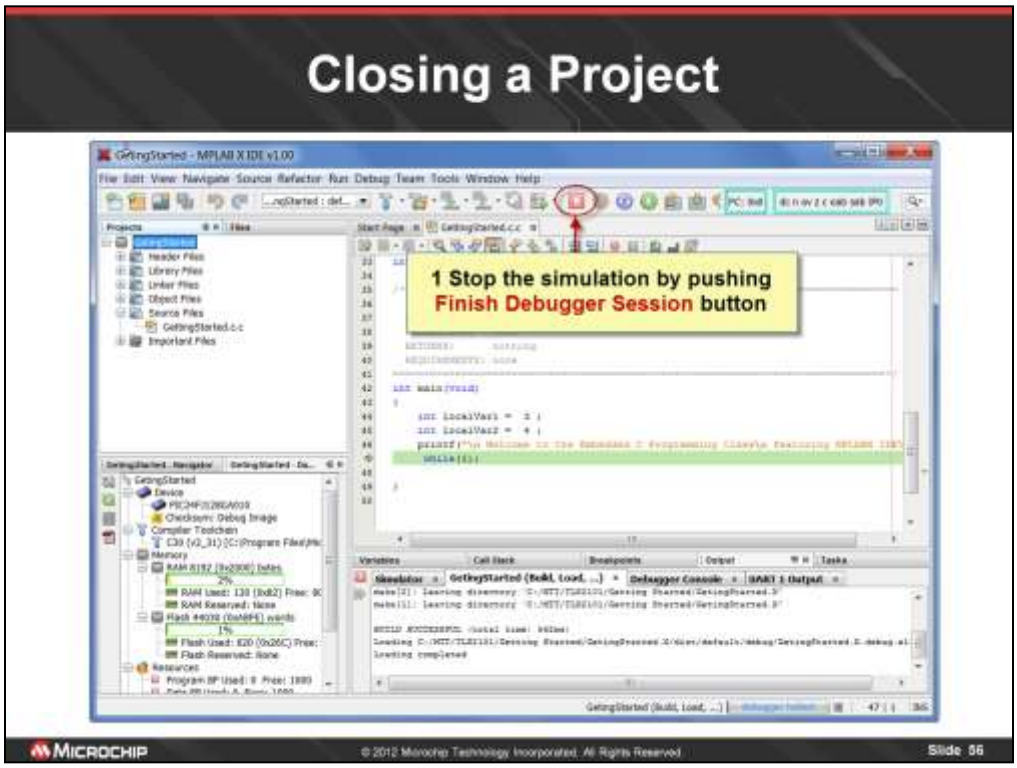


© 2012 Microchip Technology Incorporated. All Rights Reserved.

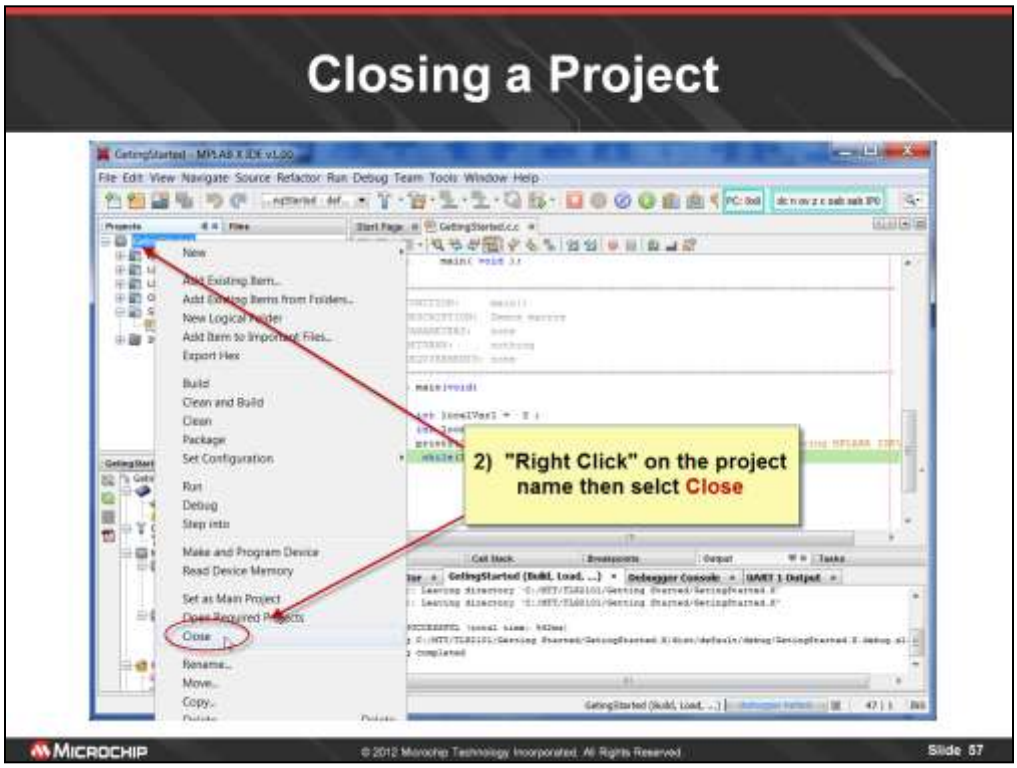
The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.



The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.

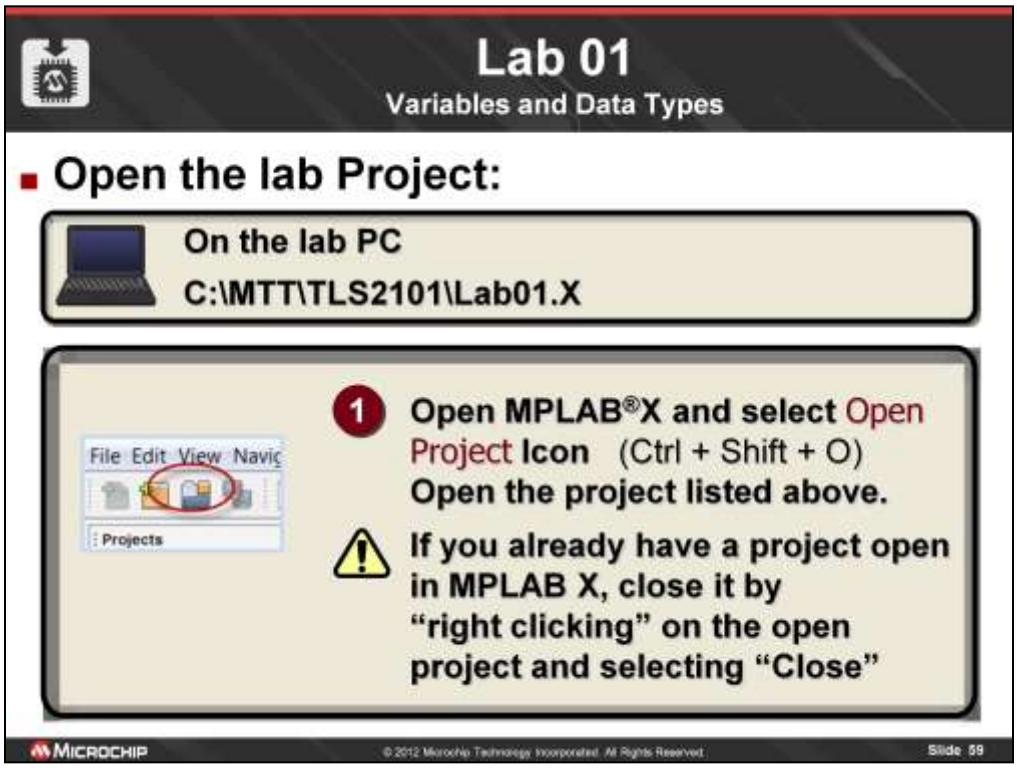


The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.

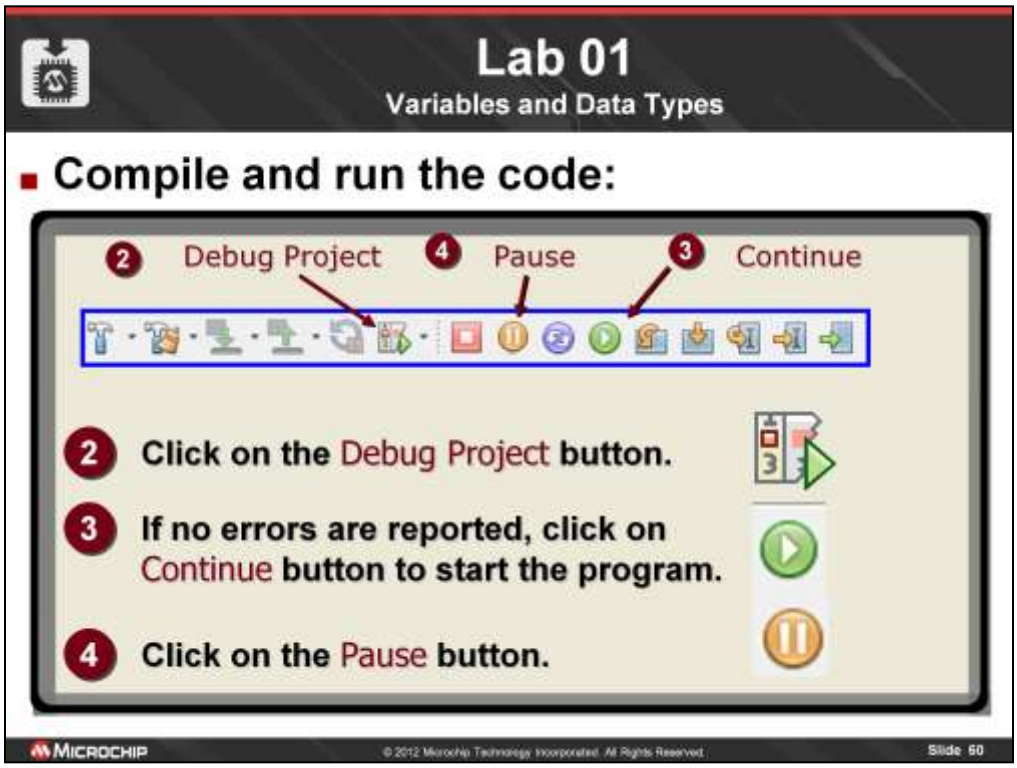


The labs in this class use MPLABX's software simulator. The simulator has several options for operation at startup. To ensure your lab experience matches the lab manual's we will set up the simulator to 'wait' at what is call the main function.

MICROCHIP
Lab Exercise 1
Variables and Data Types



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.



Next, click on the Build All button to compile the code.
If you don't get any errors (you shouldn't), then go ahead and hit the run button.
After a second or two, hit the halt button, and then let's look at the code and what it did.

Lab 01

Variables and Data Types

- **Expected Results (1):**

Simulator × | UART 1 Output × | Lab01 (Build

```
A character variable requires 1 byte
A short variable requires 2 bytes
An integer variable requires 2 bytes
A long variable requires 4 bytes
A floating point variable requires 4 bytes
A double variable requires 4 bytes
```

5 **The UART 1 Output window should show the text that is output by the program, indicating the sizes of C's data types in bytes.**

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 01

Variables and Data Types

- **Expected Results (2):**

Variables | Breakpoints

Name	Type	Address	Value
charVariable	char	0x9DE	0x32
doubleVariable	float	0x9DA	50.0
floatVariable	float	0x9D6	50.0
intVariable	int	0x9D4	0x0032
longVariable	long	0x9D0	0x00000032
shortVariable	short	0x9CE	0x0032

6 **The Variables window (alt + shift + 1) shows the values and addresses of the variables**

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 01

Variables and Data Types

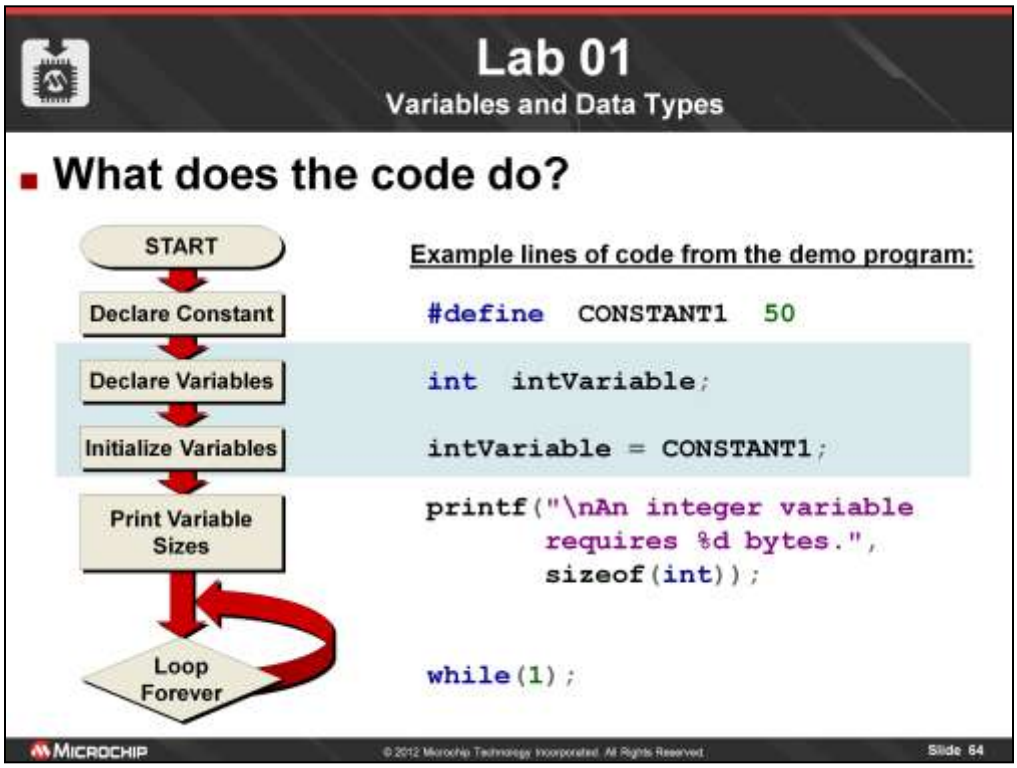
7 Variables in Memory

16-bit Data Memory

0x09CD			0x09CC
0x09CF		32	0x09CE ← short int
0x09D1	00	32	0x09D0 } long int
0x09D3	00	00	0x09D2
0x09D5	00	32	0x09D4 ← int
0x09D7	42	48	0x09D6 } float
0x09D9	00	00	0x09D8
0x09DB	42	48	0x09DA } double
0x09DD	00	00	0x09DC
0x09DF		00	0x09DE ← char
0x09E1			0x09E0

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.



The key points to remember from this demo and section is that variables must be declared before they can be used, and a variable must be declared with one of the data types so that the compiler knows how much space it requires in memory and how it should handle the variable's contents. The other items will be covered in subsequent sections.

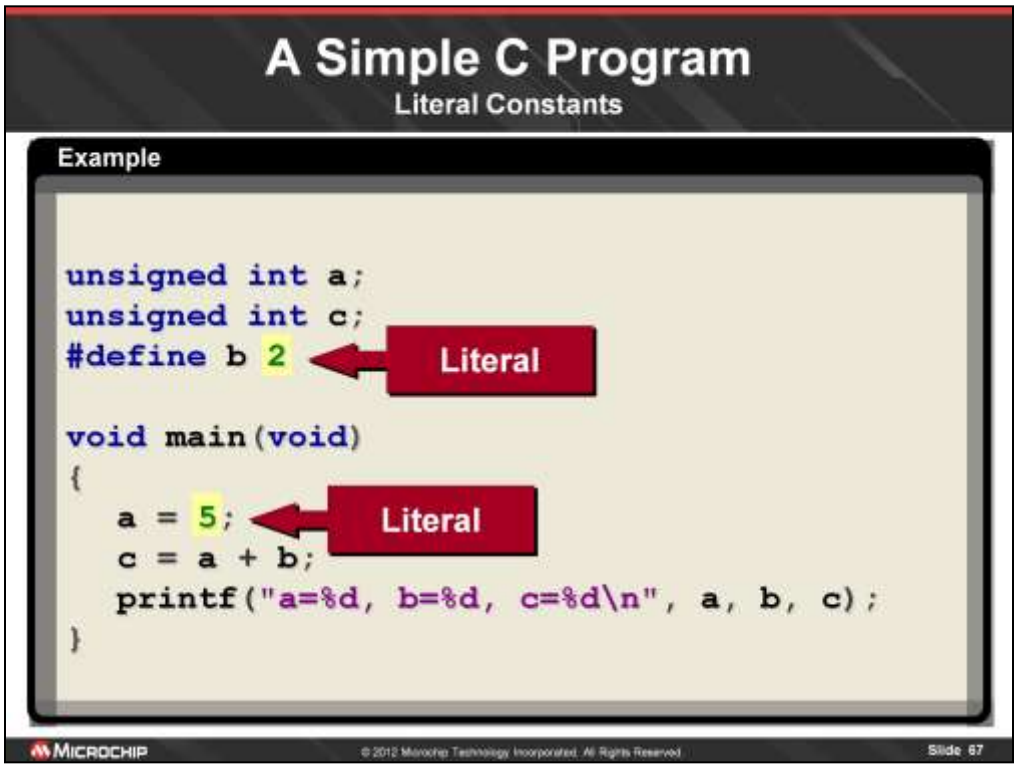
A more detailed version of this slide along with a line by line explanation of the program is included in the lab handout.

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 (if int is 16-bit)
- Don't use float/double unless you really need them

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Literal Constants



Now that we understand variables, we need to understand literals – the values we assign to variables. A literal is a constant value that we specify directly. In the case above, the literal value 5 is assigned to the variable 'a' in the main function. Above that we are defining a constant (which we will discuss soon) such that the symbol 'b' represents the literal value 2.

Literals need not be numeric. The character 'H' is a literal, as is the string "Microchip". Anytime you specify the actual value of something directly in this fashion, that is a literal.

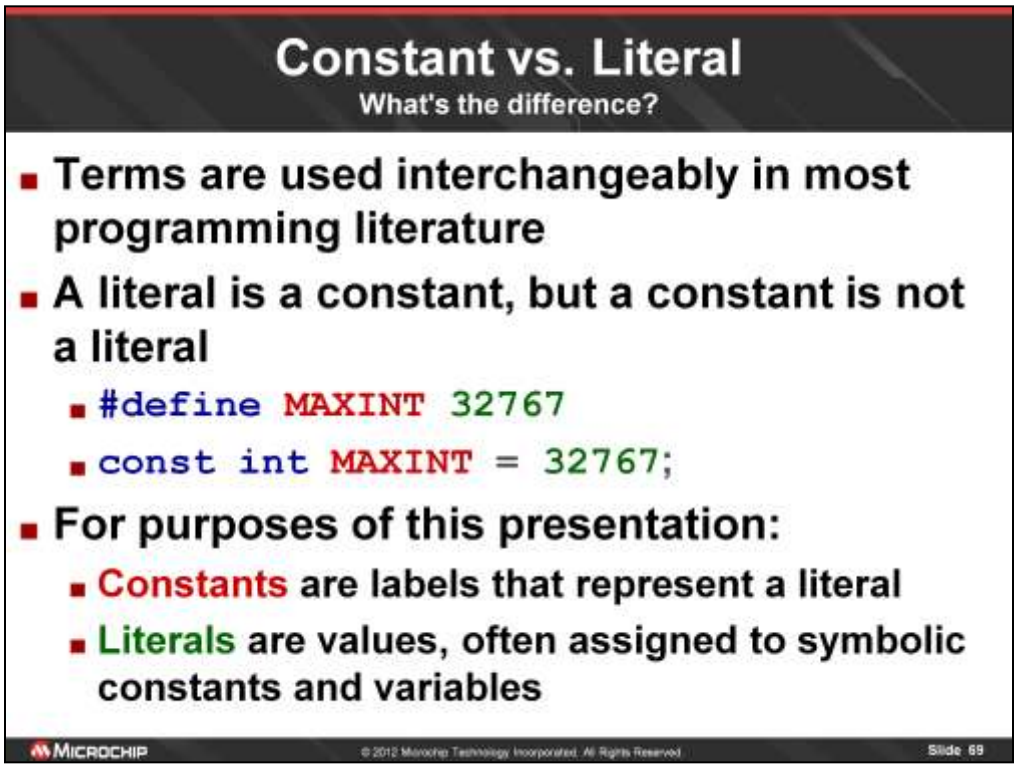
Literal Constants

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)

© 2012 Microchip Technology Incorporated. All Rights Reserved.



Before going further in the presentation, I would like to make a distinction between two terms that are often used interchangeably in programming literature: Constants and Literals. A more specific description would be Symbolic Constants and Literal Constants, which I will often use, but like others I also resort to the shortened forms. So to avoid confusion I would like to give you my definitions of the terms.

A literal is a constant, but a constant is not a literal.

A constant is a value that never changes. It may be a symbol that represents a constant value: MAXINT, or it may be a literal value like 5. However, I will almost always use it to describe symbolic constants.

A literal is simply a value, such as 5, or a string literal such as "Hello, world!". Literals are what we assign to variables and symbolic constants. Literals are values you type directly into your code wherever they are needed.

So if we have:

const int MAXINT = 32767;

or

#define MAXINT 32767

MAXINT is a constant and 32767 is a literal (which is always a constant value of 32767).

While most literature on programming uses the terms literal and constant interchangeably and fails to make any distinction, I will be using these terms to mean two distinct things as shown above. The term "constant" will always refer to a symbolic constant (text substitution label or constant variable) and the term "literal" will always refer to the actual values that we assign to symbolic constants and variables. While it is true that a literal is also a constant, I will not refer to them as such in order to reduce potential confusion in our discussions.

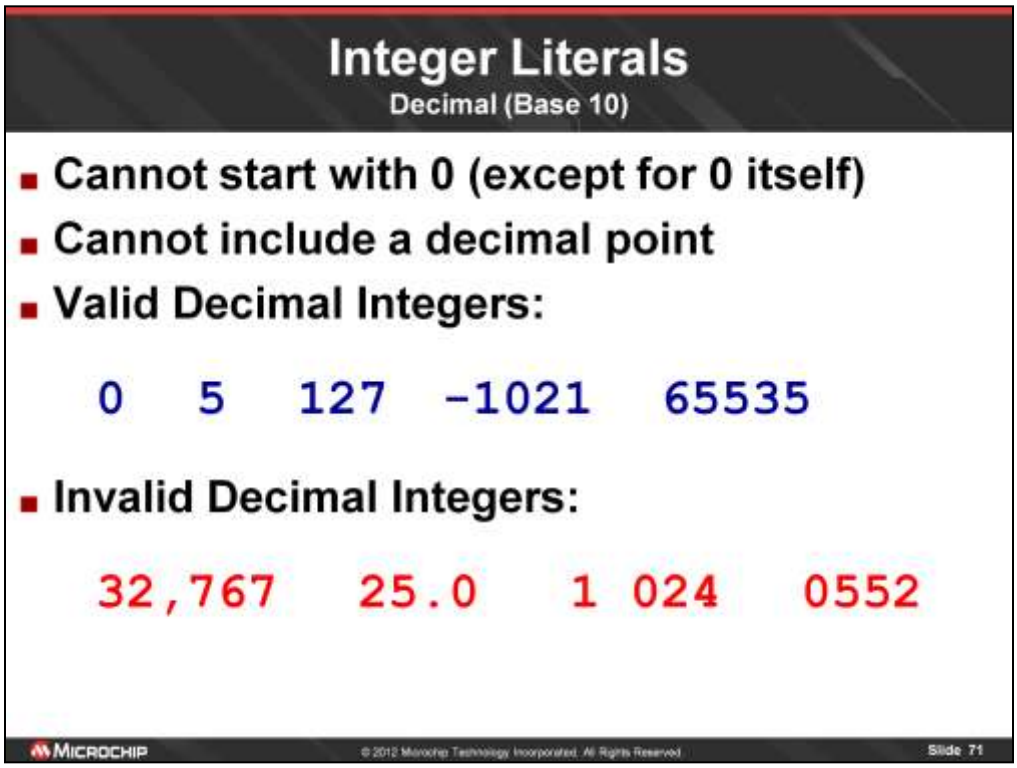
Literal Constants

- 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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 70

There are four basic types of constants in C. There is a fifth kind – enumeration constants – which will be covered later.

Note that three of the four basic types of literals match up exactly with the three basic variable types: int, float and char.



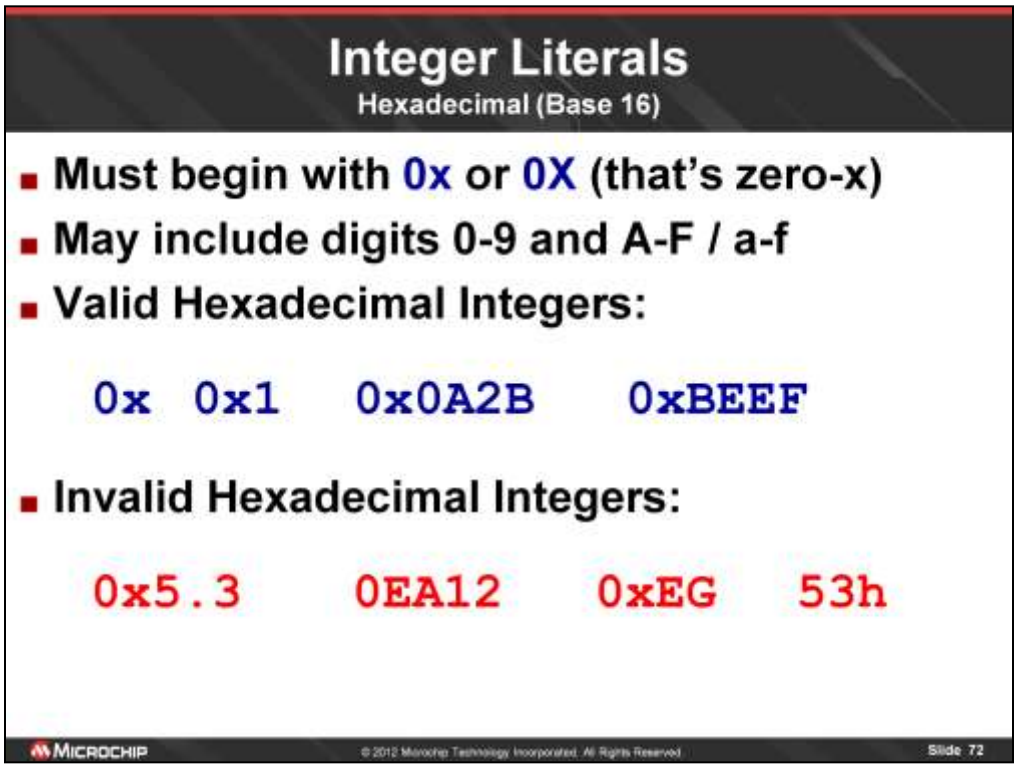
Decimal Integer constants may be as large or small as their variable counterparts (int, long, signed, unsigned). The rules for specifying an integer constant in decimal are fairly simple – only the digits 0-9 may be used. No other characters except for the minus sign at the beginning are valid.

Decimal integers other than 0 itself, cannot be preceded by a 0 because that is the syntax for an octal number (012 is 12 base 8).

Decimal integers cannot contain a decimal point, even if the fractional part is zero.

The problems with the invalid integers are as follows:

32,767	Contains a comma (,)
25.0	Contains a decimal point (.) – this is a problem even though the fractional part is 0.
1 024	Contains a space between the 1 and 0
0552	Contains a leading 0. This will be interpreted as an octal number.



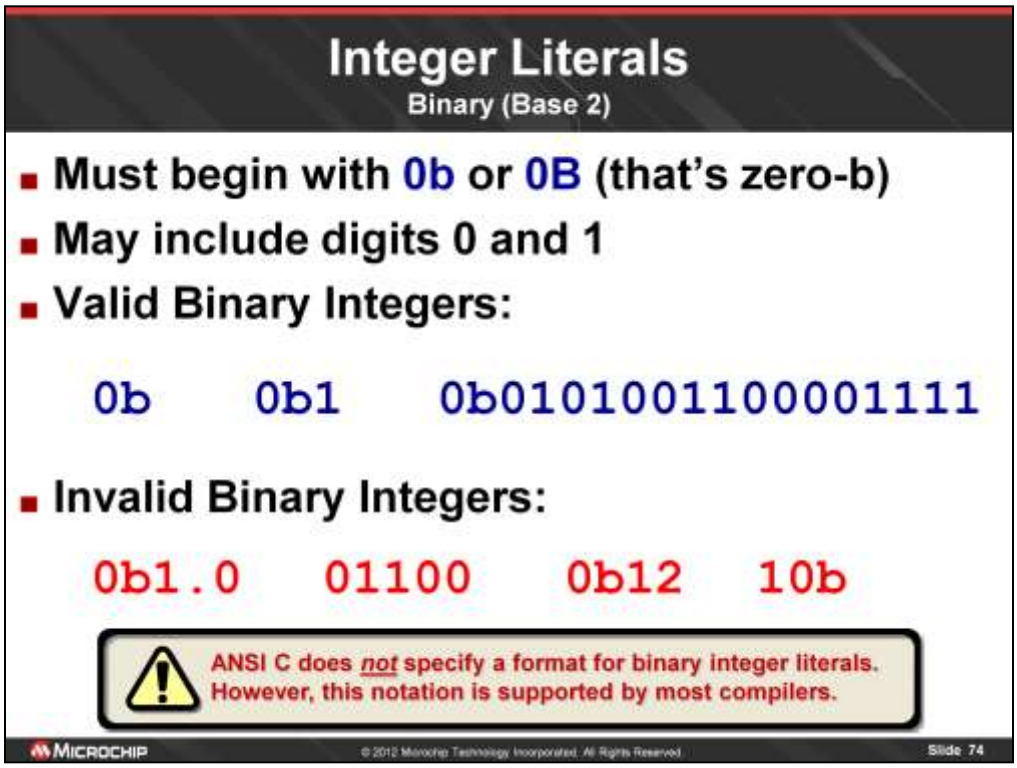
Hexadecimal integers have most of the same restrictions as decimal integers.

Hex numbers must begin with a zero x (0x). This has no value on its own, it is just a prefix that indicates that the following digits are in base 16.

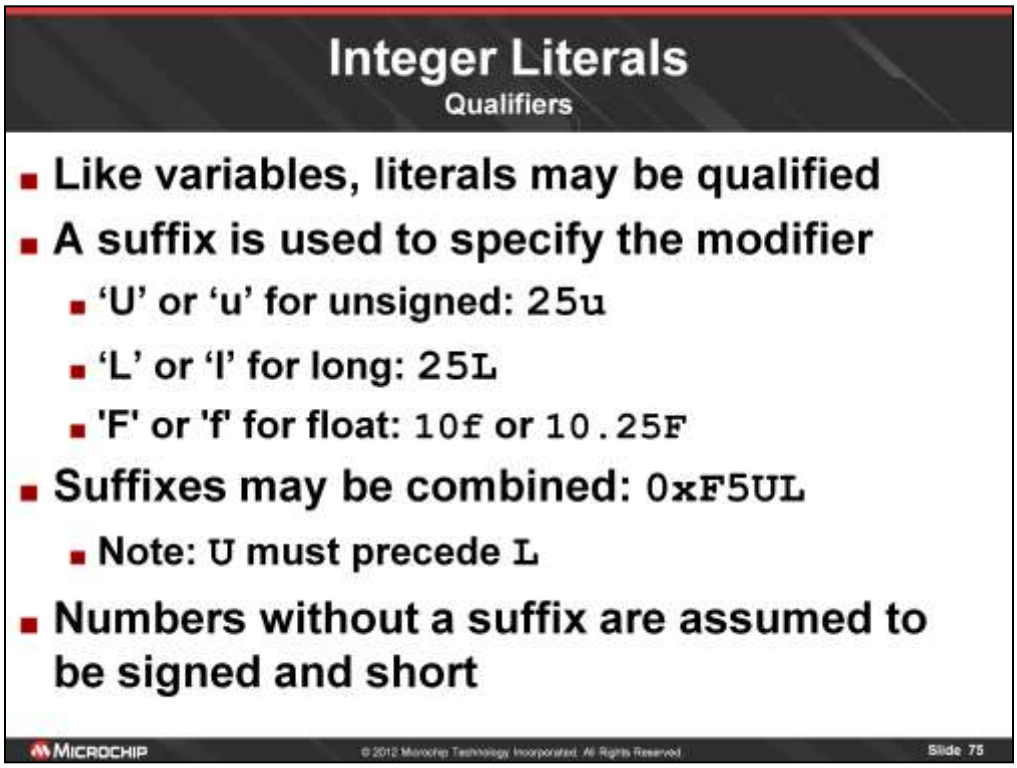
The digits themselves contain the digits 0-9 and the letters A-F, which may be upper or lower case.

Contrary to the belief of many programmers, it is not necessary to precede the hex digits A-F with a 0 when they are the first digit. For example, 0xA5 is perfectly valid. It is not necessary to write it as 0x0A5.

NOTE: Octal is supported as well, but no one uses it anymore. Octal numbers are just like decimal but are always preceded by a 0 (zero) and only the digits 0-7 are valid. e.g. 012



For whatever reason, the C language specification provides no means of directly specifying a binary integer. The expectation is that binary numbers are to be treated in groups of three (octal) or four (hexadecimal). However, in embedded systems in particular, being able to specify an integer in a binary format is extremely useful, such as when configuring a peripheral register where each bit has a specific meaning. Therefore, most compilers for embedded systems (and most compilers in general) support a syntax that is similar to that used for hexadecimal, where a 0b is used as a prefix. While it is safe to use with Microchip's compilers and most others, you may want to avoid its use if you want your code to be as portable as possible. Using this syntax violates the ANSI standard.



While many compilers will choose the appropriate qualifier based on the context of the literal, you may add a suffix to ensure that a literal is interpreted as being unsigned or long (or both). It is purely optional in most cases, but may be necessary to force the correct qualification in other cases. These qualifiers may be used with all integers in any of the supported radices.

The 'f' suffix is primarily used to distinguish a number like "10.25" as a float as opposed to a double, which would be the default type of a literal with a decimal point.

Floating Point Literals

Decimal (Base 10)

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

```
2.56e-5   10.4378   48e8   0.5   10f
```

- Invalid Floating Point Literals:

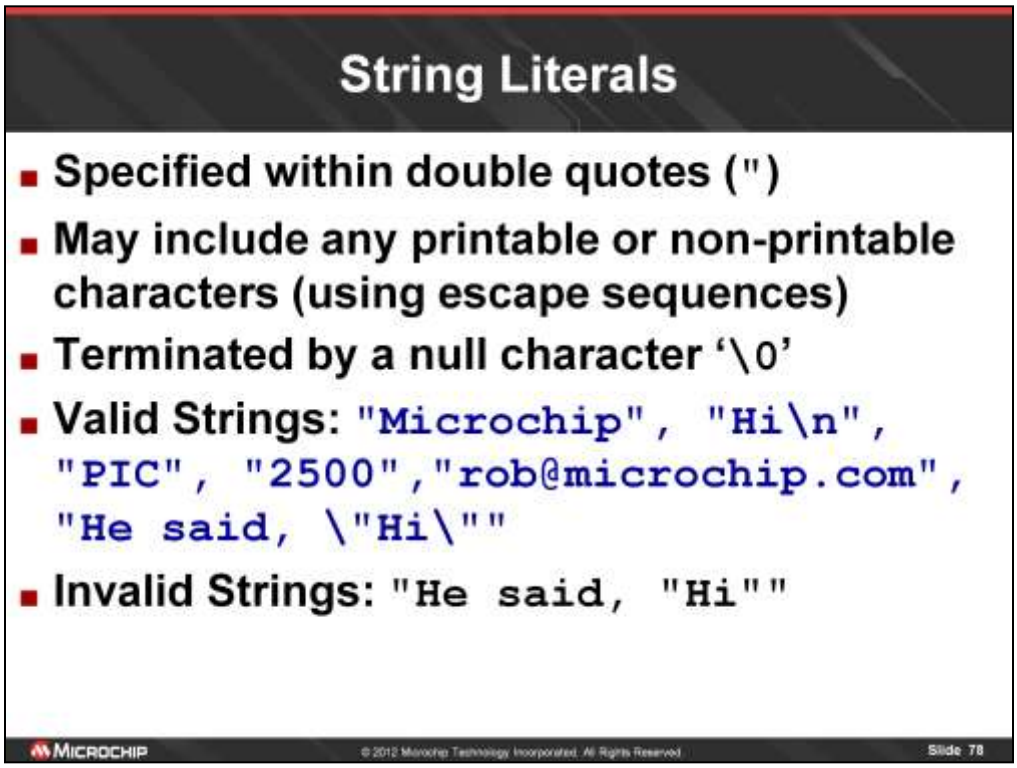
```
0x5Ae-2   02.41   F2.33
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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'` = NUL) (also called digraphs)
- Valid Characters: `'a'`, `'T'`, `'\n'`, `'5'`, `'@'`, `' '` (space)
- Invalid Characters: `'me'`, `'23'`, `'''`

© 2012 Microchip Technology Incorporated. All Rights Reserved.



Strings are something of a special case in C. You may have noticed that there is no "string" data type, so how can we have string literals?

In reality, a string is simply a group of characters. Therefore, strings are treated as arrays of type char. We will see much more of this later on. For now, let's focus on what makes a valid string literal.

First it must be enclosed in double quotes. This distinguishes a string literal from a single character literal which is in single quotes. A string can consist of all the same characters that may be specified as a character literal. However, in C, a string must also be terminated by a null character (ASCII value 0x00). More on this in a moment.

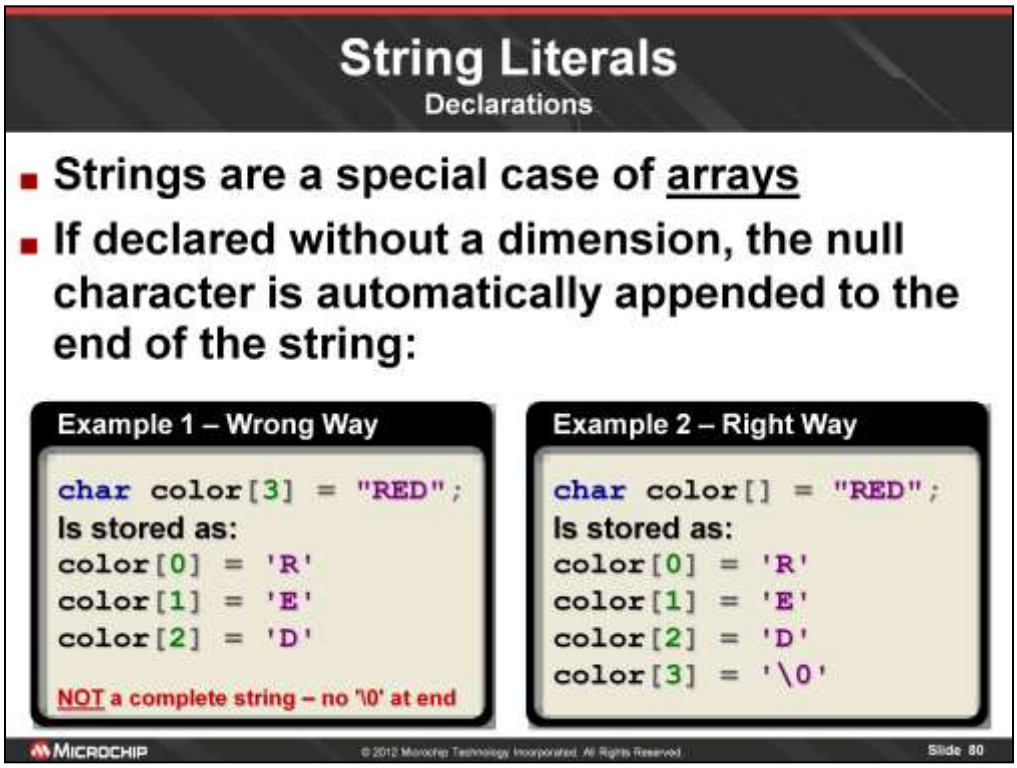
In the examples of valid and invalid strings, most of the items should be fairly obvious. Any printable or non-printable character may be included in a string. The problem is how do you specify a double quote? In the invalid example, we are trying to put quotes within a string, but the compiler will think that this line is the string "He said, ", followed by an undefined identifier 'Hi', followed by an empty string "". The double quote character needs to be specified by the escape sequence \". This is shown in the last example of valid strings.

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.



In C, all true strings must be terminated by a null character. This is required because most of the library functions that operate on strings use the null character to determine where a string ends. Therefore, it is important to remember that strings, which are nothing more than character arrays, are declared without any specified value for the dimension. This will cause the compiler to automatically append a null character to the end of the string. If for some reason you must specify a dimension, then the null character must be explicitly included as part of your string literal. (i.e. a '\0' must be at the end of your string)

TECHNICAL DETAILS FOR PRESENTER:

When declaring strings, it is important to NOT specify a dimension for the array that holds them unless you explicitly include the null character. If there is no null character terminating the string, many functions in C's standard libraries will not operate properly. As an example, here is a short program I wrote for C30:

```
char a[3] = "RED";   // [R] [E] [D]
char b[3] = "RUM";   // [R] [U] [M]
char c[] = "MCHP";   // [M] [C] [H] [P] [\0]

void main( void )
{
        printf("a = %s\n", a);          // %s indicates that the data to be printed is a string
        printf("b = %s\n", b);
        printf("c = %s\n", c);
}
```

When this program is run, the following is displayed in the UART I/O window:

a = REDRUMMCHP

b = RUMMCHP

c = MCHP

The printf function starts with the first character of the array and will print everything in the string until it encounters a null character. In this case, all three strings are stored sequentially in memory, so to the printf function, all three appear to end at the same place: c[4], which is a null character. (Remember that arrays in C are zero based – c[4] is the 5th character.) If c[] were not terminated with a null character, the printf functions could potentially have output a lot more garbage at the end of each respective string. Fortunately, at some point a null character is likely to be detected, even if it wasn't intended to be interpreted as such. In other words, as the function continues to pull characters out of memory beyond the intended string, it is highly likely that a zero value will be encountered in some memory location that may or may not be part of some character variable. Even so, this is a dangerous situation and can lead to unpredictable results.

String Literals

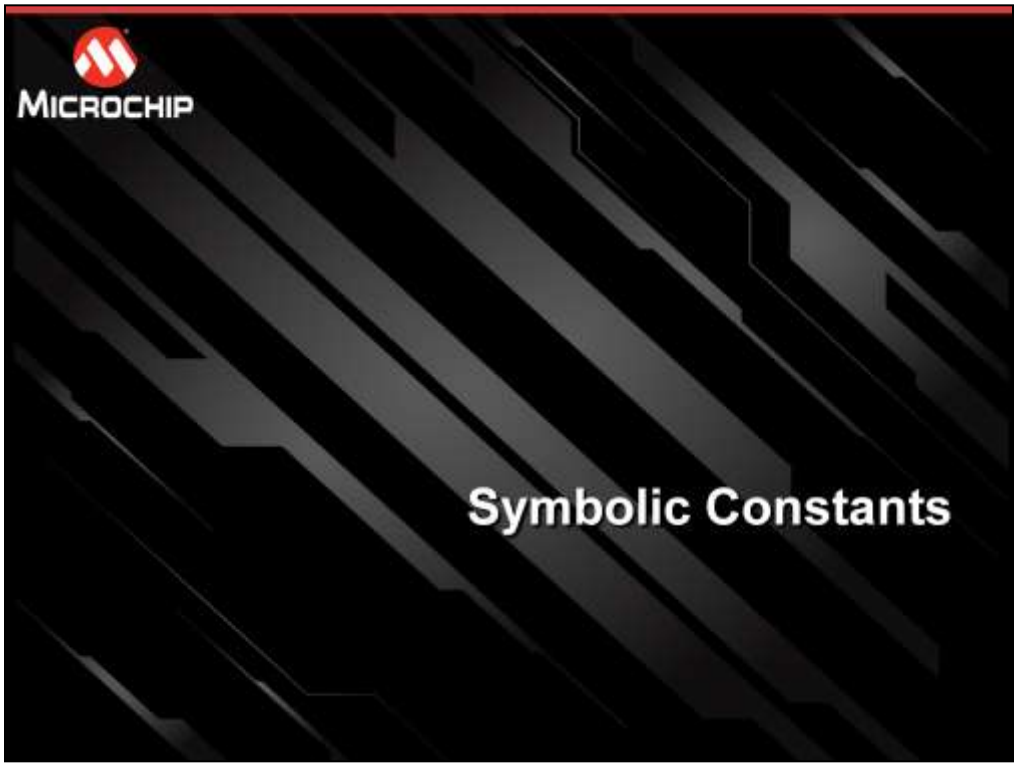
How to Include Special Characters in Strings

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.


MICROCHIP
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 (!!??)

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Symbolic Constants

Text Substitution Labels Using `#define`

- **Defines a text substitution label**

Syntax

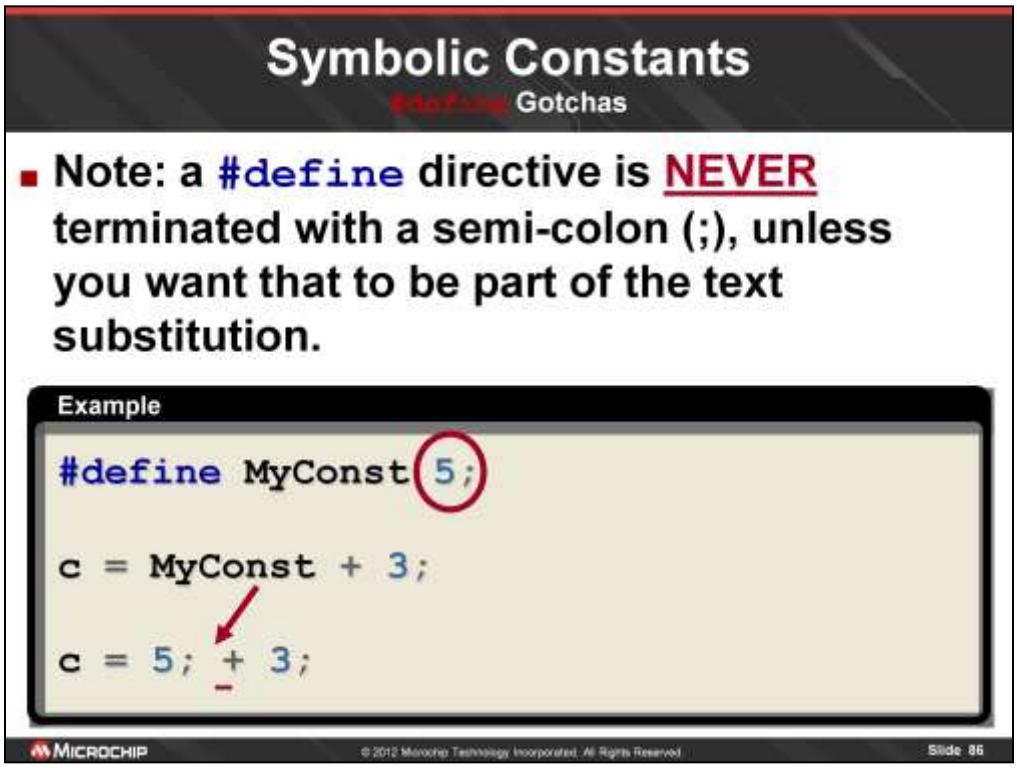
```
#define label text
```

- **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
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.



While it is important to remember to end each C statement with a semi-colon, it is equally important to recognize that directives are NOT statements and should never be terminated by a semi-colon, unless you want the semi-colon to be part of the string used for text substitution.

In the example shown, if we terminate the #define directive with a semi-colon, when MyConst is replaced by the preprocessor, it is replaced with “5;”. The resulting expression will not work as originally intended.

Note for the non-English speaking presenters: “gotcha” is a slang term for “got you” which in this case refers to a simple mistake that causes a problem – the mistake *got you* (caused you to be) in trouble. These simple mistakes are frequently referred to as “gotchas”.

Symbolic Constants

Initializing Variables When Declared

- A constant declared with `const` may not be used to initialize a global or static variable when it is declared (though it may be used to initialize local variables…)

Example

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

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 87

PERSONAL NOTES (contains info not covered at this point in class)

This is NOT allowed:

```
const int K = 5;
int x = K;                    // Compiler will generate an error here
int main(void) { … }
```

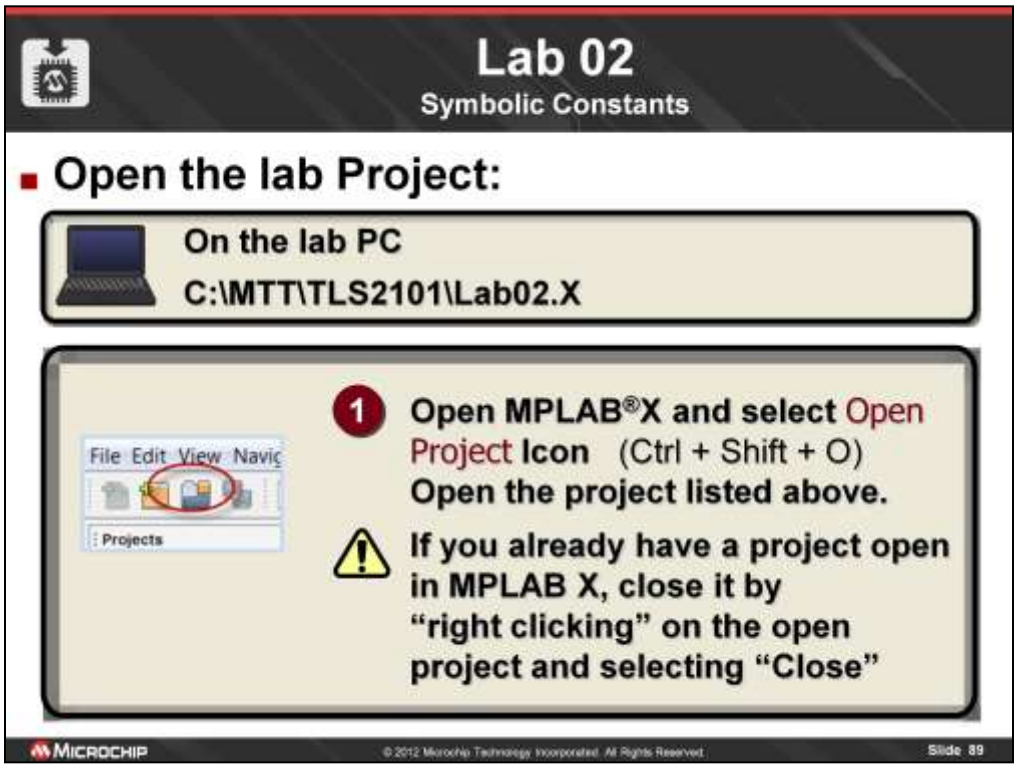
This is also NOT allowed:

```
const int K = 5;
int main(void)
{
        static int x = K;  //Compiler will generate an error here
        …
}
```

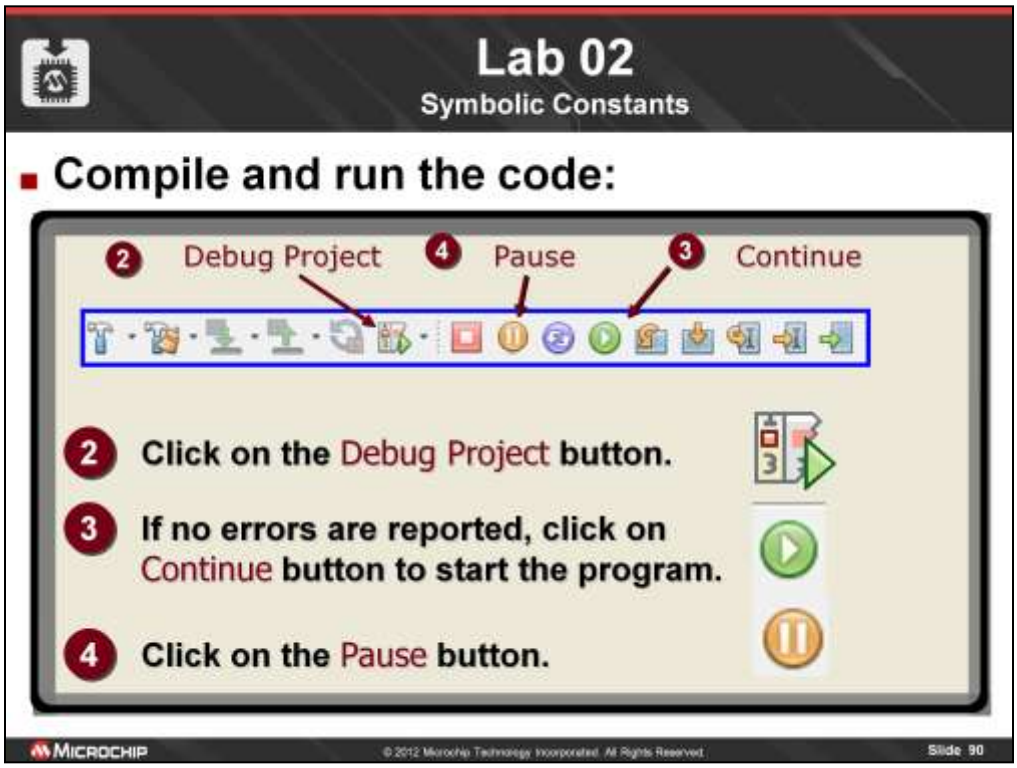
This is allowed:

```
const int K = 5;
int main(void)
{
        const int N = 10;
        int x = K;
        int y = N;
        …
}
```

MICROCHIP
Lab Exercise 2
Symbolic Constants



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.



Next, click on the Build All button to compile the code.
If you don't get any errors (you shouldn't), then go ahead and hit the run button.
After a second or two, hit the halt button, and then let's look at the code and what it did.

Lab 02

Symbolic Constants

- **Expected Results (1):**

Output

UART 1 Output × | Simulator × | Lab02 (Build,

```
The first constant is 0x33
The second constant is 0xCC
```

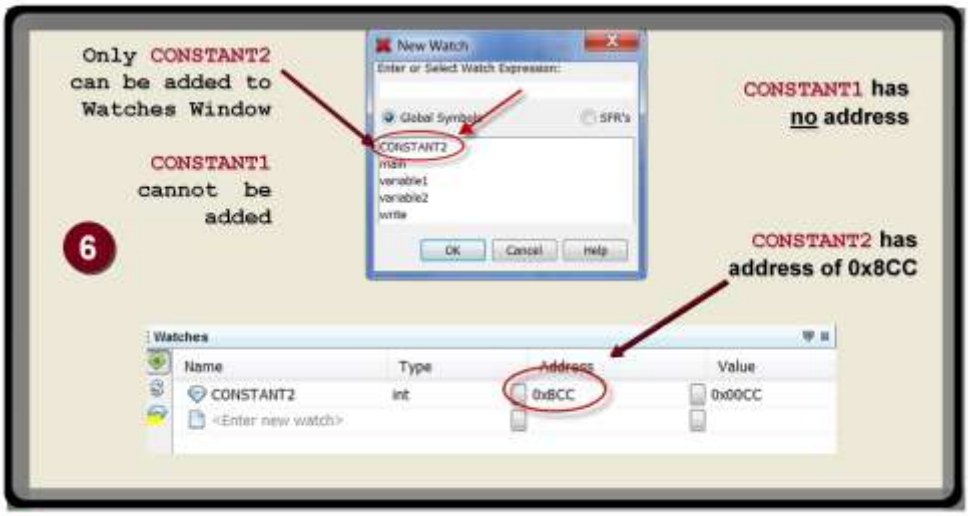
5 **The UART 1 Output window should show the text that is output by the program, indicating the values of the two symbolic constants in the code.**

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 02

Symbolic Constants

- **Expected Results (2):**



© 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 02

Symbolic Constants

- **Expected Results (3):**

Program Memory only contains CONSTANT2

CONSTANT1 not in Program Memory

Program | Output

Line	Address	Opcode	Label	DisAssy
1510	00BCA	060000		RETURN
1511	00BCC	0000CC	_CONSTANT2	NOP
1512	00BCE	006854		NOP
1513	00BD0	002065		NOP
1514	00BD2	006966		NOP
1515	00BD4	007372		NOP

7

CONSTANT2 has the address x0BCC

CONSTANT2 has a value of 0x00CC

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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…)

© 2012 Microchip Technology Incorporated. All Rights Reserved.

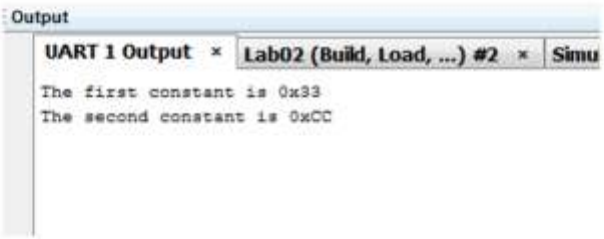
MICROCHIP

printf() Library Function

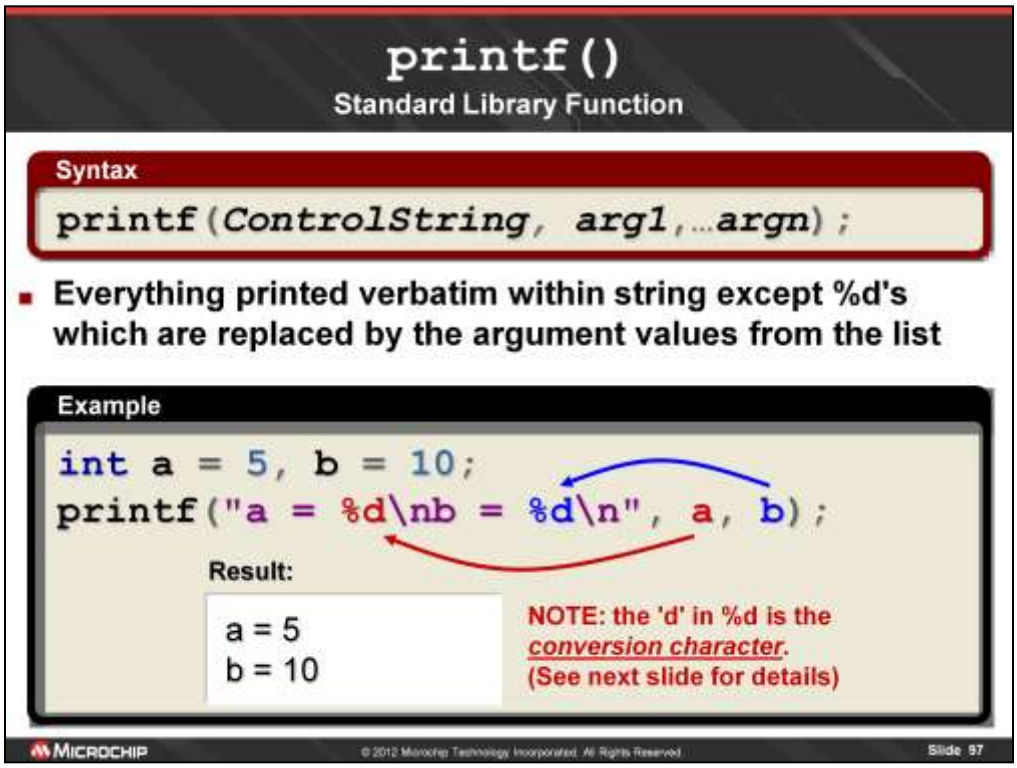
printf()

Standard Library Function

- Used to write text to the "standard output"
  - Normally a computer monitor or printer
  - Often the UART in embedded systems
  - UART1 Output window in MPLAB® X Simulator



© 2012 Microchip Technology Incorporated. All Rights Reserved.



The printf function contains two parts. The first part is the control string operand. This string contains the text you want to print along with some formatting characters that take the place of a variable, constant or expression. The second part is a list of arguments that are the variables, constants or expressions that are to be used wherever the formatting characters are used in the control string.

In the example, the first %d indicates that the variable a is to be printed as a decimal integer and the second %d indicates the same condition for the variable b. The "\n" is the new line character (escape sequence) and causes the text after it to be printed on the next line of the output window.

The % symbol indicates the beginning of a format specifier. Most of the time, the % will be followed by a single character that describes the "type" format as shown in the table on the next slide. However, additional formatting options may be added in between the % and the conversion character. Since printf() is not used frequently in embedded systems, this extra formatting information will not be covered in the class. It will be included in an appendix for those who are interested.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 98

This is a table of conversion characters that may be used with the printf() function. These are the "placeholders" that are preceded by a percent sign '%' that determine how a variable should be formatted in the string that is printed. For example:

```
double x = 6.02e23;
printf("A mol is a quantity of %g things.", x);
```

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

HIDDEN SLIDE

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

printf()

Useful Format String Examples for Debugging

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

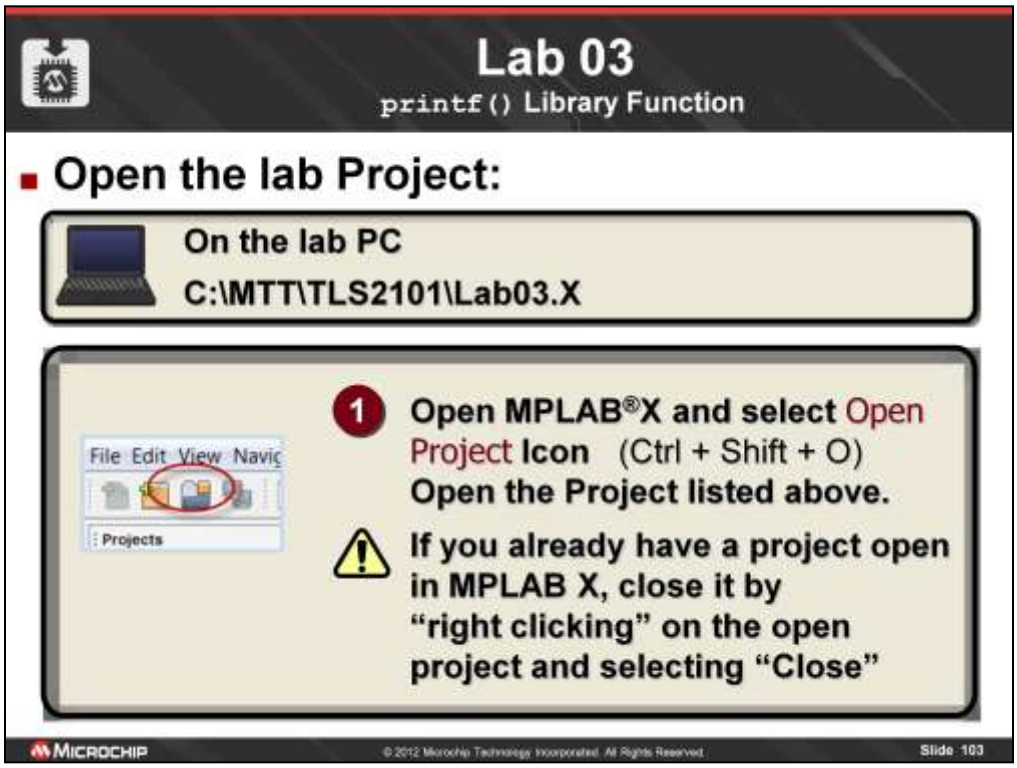
```
printf("Address of x = 0x%04X\n", x_ptr);
```

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

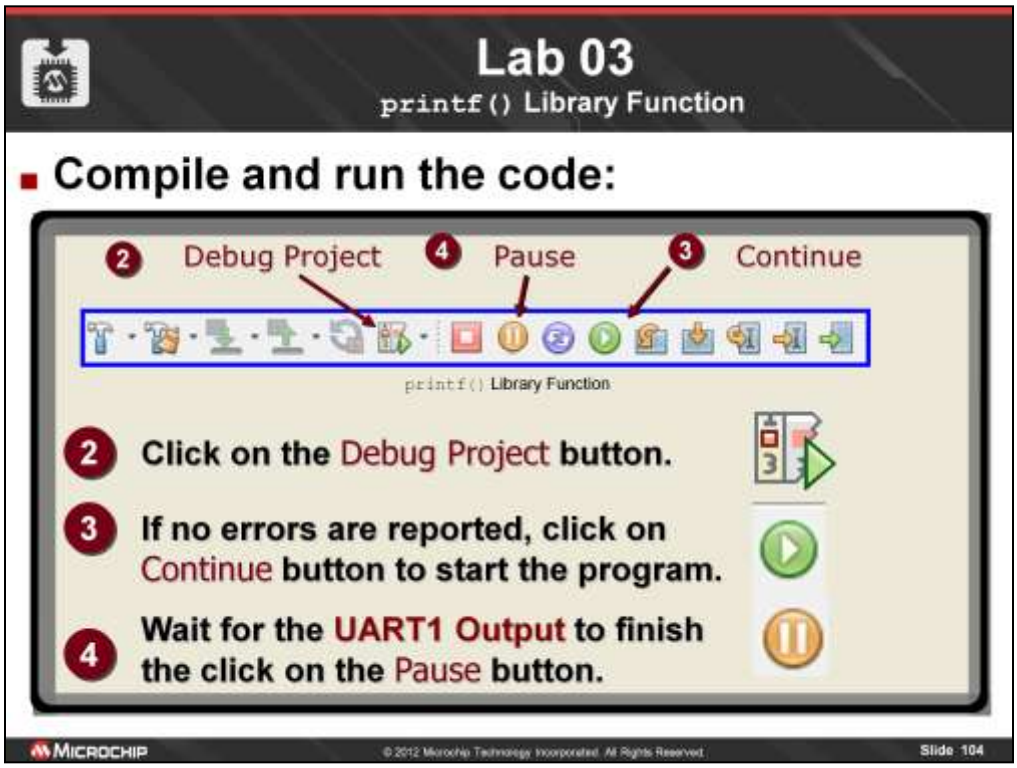
X Specifies that the output value should be expressed as a hexadecimal integer with uppercase A-F

© 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 3
printf() Library Function



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.



Next, click on the Build All button to compile the code.
If you don't get any errors (you shouldn't), then go ahead and hit the run button.
After a second or two, hit the halt button, and then let's look at the code and what it did.

Lab 03

printf() Library Function

- **Expected Results (1):**

UART 1 Output × | Lab03 (Build, Load, ...) × | Debugger Co

```
25 as decimal (d): 25
'a' as character (c): a
'a' as decimal (d): 97
2.55 as float (f): 2.550000
2.55 as decimal (d): 16419
6.02e23 as exponent (e): 6.020000e+23
6.02e23 as decimal (d): 26366
'Microchip' as string (s): Microchip
'Microchip' as decimal (d): -25790
```

5 The **UART1 Output** 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.

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

HIDDEN SLIDE

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

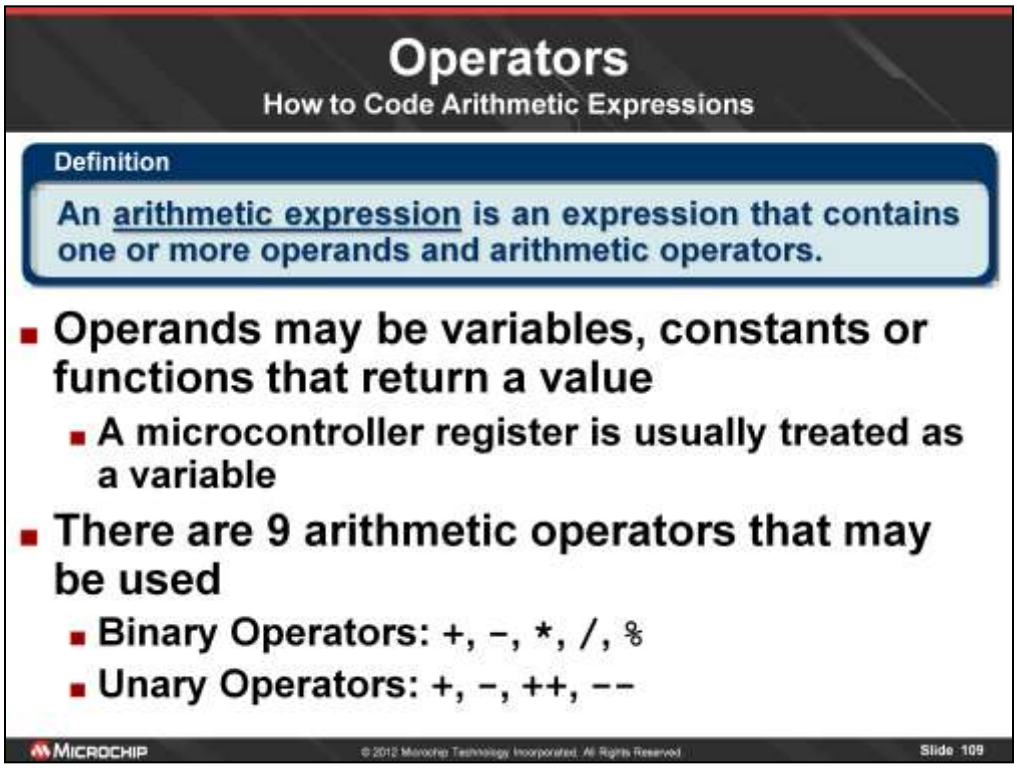
Lab 03
Conclusions

- 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®X -IDE)
  - Include the stdio.h header file

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 107

HIDDEN SLIDE

MICROCHIP
Operators



The first group of operators we will look at are those that are used in arithmetic expressions. Most of these should look familiar to you, as you have already seen them in some of the examples, plus, they for the most part use the same symbols as you would encounter in any math book.

An arithmetic expression is one that contains one or more operands connected by arithmetic operators. The operands may be variables, constants or a function that returns a value.

There are 9 arithmetic operators that may be used in an arithmetic expression. These are broken down into two categories:

-Binary Operators: These operators require an operand on either side, such as x + y to add two numbers together.

-Unary Operators: These operate on a single operand, such as –x, to return the negative value of the operand.

We will see these in detail over the next few slides.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 110

This is the list of all but two of the arithmetic operators. They include the standard operations we are used to performing in our daily lives such as addition, subtraction, multiplication and division. There are also the unary operators + and – to return the operand or the negative value of the operand.

One that you may not be familiar with is the modulo operand (%). There are many applications of this operator, but its basic function is to return the remainder of an integer division operation. We'll see more about how this can be used later in the class.

Arithmetic operands are subject to the following rules:

•% requires integer operands

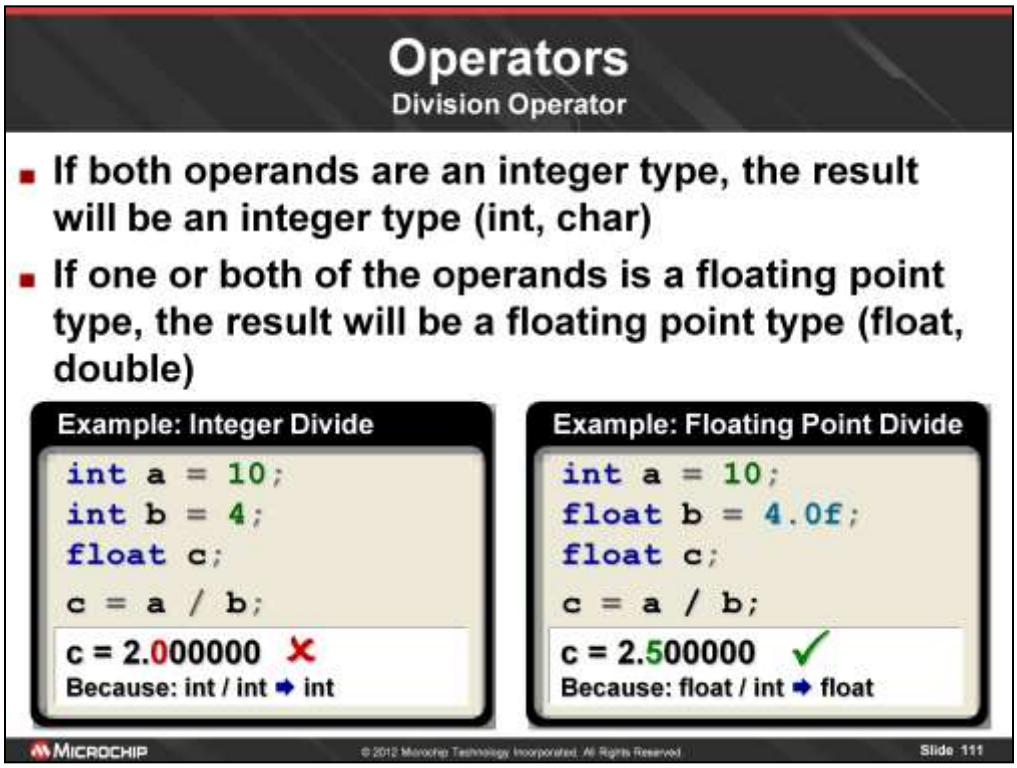
•all others may use any arithmetic type

Addition and subtraction operators may be used on pointers in the following cases:

•For addition, one operand can be an object pointer while the is an integer

•For subtraction, both operands can be pointers to objects of the same type

•For subtraction, the minuend (left hand operand) may be an object pointer while the subtrahend (right hand operand) is an integer



NOTE: both int and char are considered integer types. Remember that char is just an 8-bit integer. Also both float and double are considered floating point types.

The division operator is a bit different from the others in that it can function very differently depending on the data types of its two operands. If you divide two integer type values (int or char) the result will be an integer (can be stored in int or char depending on its size). If one or both of its operands is a floating point type (float or double) the result will be a floating point number (can be stored in float or double depending on its size). It is very important to remember this because you can easily lose the fractional part of the number if you aren't careful.

For example, even though the expression on the left is being assigned to the float variable c, the result is truncated, losing its fractional part.

Operators

Implicit Type Conversion

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

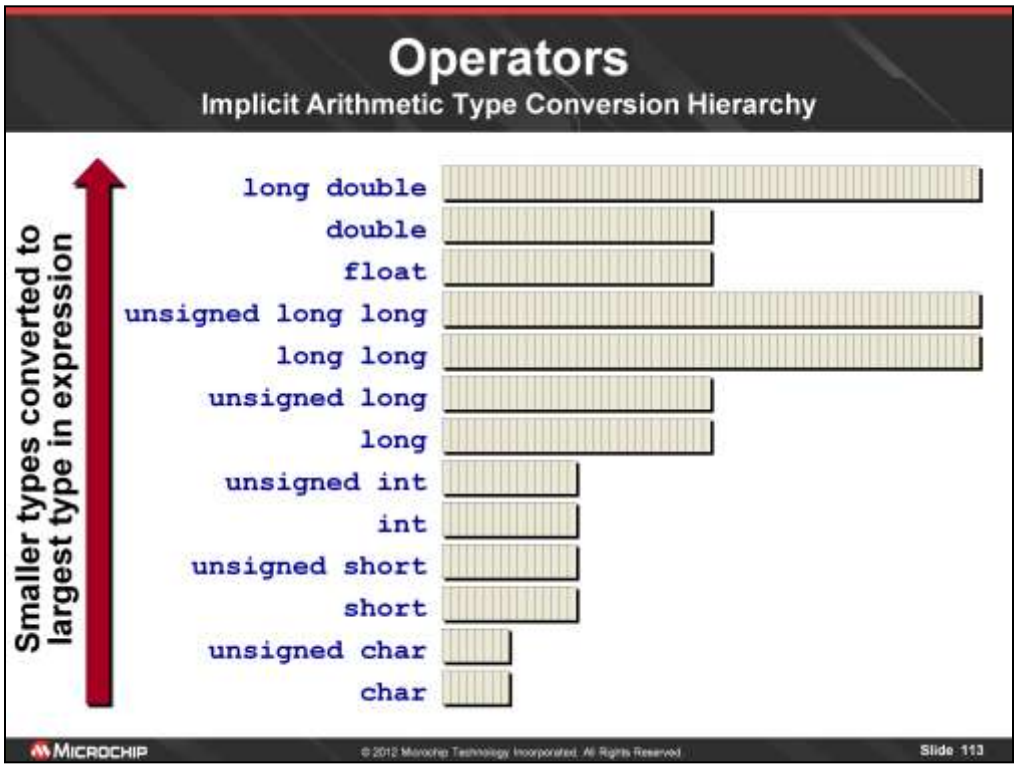
Example

```
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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 112

The previous example is just a special case of the overall concept of implicit type conversion. In virtually any arithmetic operation, the data types of the operands will be temporarily converted to the data type of the largest operand. For example, if we are multiplying an integer by a float, the integer will be promoted to a float for the duration of the evaluation of the expression. As we see here, we are multiplying an integer with a value of 10 by a float with a value of 2.0. In this case, the int is promoted to a float, so the 10 becomes 10.0. Once the expression has been evaluated giving us a float result of 20.0, the variable x goes back to being an int. This implicit type conversion helps to ensure that no precision is lost as part of the operation.



Implicit type conversion follows the hierarchy shown here, where any operand of a type below that of the largest one in an operation will be converted up to the largest type.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 114

Here are some examples of real world implicit type conversions. For each example, assume that x is defined as a short int with a value of -5.

In the first case, even though x is the only operand, it is promoted to int for the operation because int is the fundamental data type for most operations.

In the second case, x is promoted to long since the literal is explicitly specified as a long, which is larger than short.

In the third case, we are dividing an int by a short, so the short is promoted to int.

The same holds true for the fourth case.

The last one has x promoted to double because the numerator is type double (since we didn't explicitly specify the numerator as a long: 8.0L, the compiler assumes it is the largest unmodified floating point type which is double (as opposed to long double, which is a modified type).

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;
}
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 116

The final two general arithmetic operators are the increment and decrement operator. Both are unary operators, so only one operand is required, and they may be placed on either side of the operand.

If the entire statement consists of just the operand and this operator, then whether the operator is prefixed or postfixed does not matter. For example:

x++;

and

++x;

yield the same results.

However, if this operator is used as part of a larger expression, then the position of the operator becomes significant. When the operator is postfixed as in the example on the left, this means "use the value of x first when evaluating the expression, then increment x afterwards". When it is prefixed as in the example on the right, this means "increment the value of x first, then use that new value in the evaluation of the expression."

Operators

How to Code Assignment Statements

Definition

An assignment statement 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 =

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 117

The next group of operators we will be looking at are the assignment operators. You are already familiar with the simple assignment operator "=", in which it assigns the value on its right to the variable on its left. The compound assignments are a bit different, and we will look at those in a moment.

Operators
Assignment

Operator	Operation	Example	Result
=	Assignment	x = y	Assign x the value of y
+=		x += y	x = x + y
-=		x -= y	x = x - y
*=		x *= y	x = x * y
/=		x /= y	x = x / y
%=	Compound	x %= y	x = x % y
&=	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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 118

These are all the assignment operators. Again, the simple assignment should look familiar to you from some of the earlier examples, and its use should be somewhat obvious.

The compound assignment operators on the other hand often cause confusion among people new to C programming. The basic idea is that they are a shorthand notation for expressing assignments that involve the same variable on both sides of an equation, such as x = x + 5.

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):**

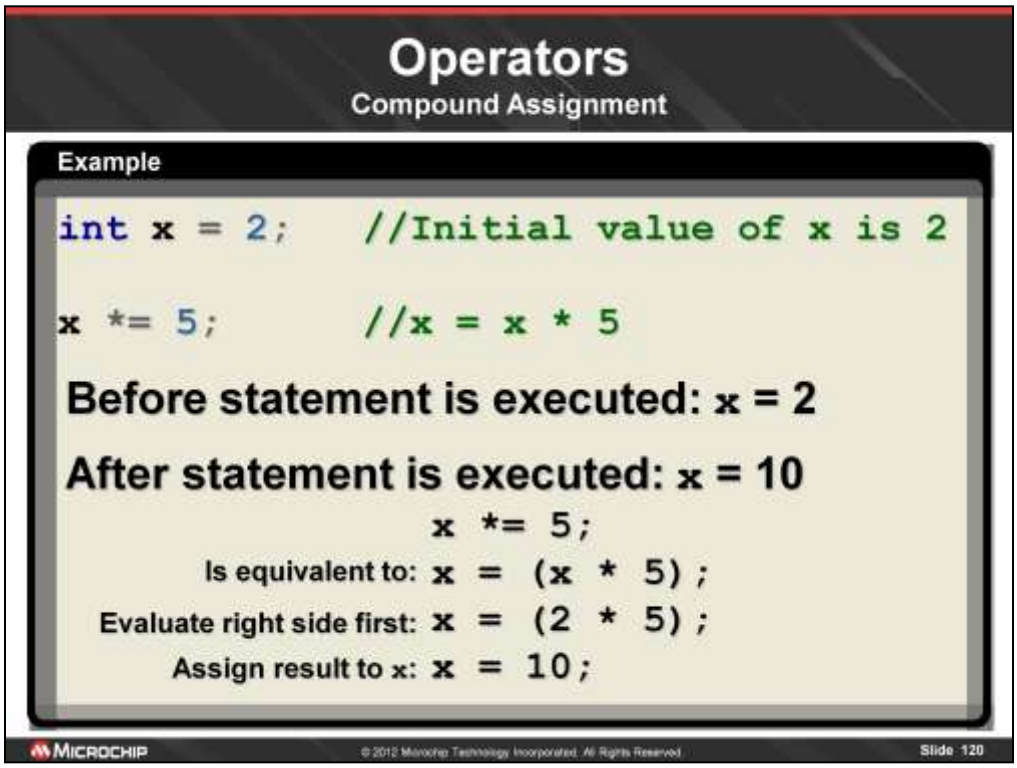
Example

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 119

When writing any kind of program, there will many instances where you will need to use a variable as part of an expression and you will want to assign the result of that expression into the same variable. One example of this kind of operation is when you want to increment a variable by some value (variable or constant). For example, if we wanted to add 5 to a variable called x, we could write it as: x = x + 5, since we want to add 5 to the value of x and store the result back in x.

Because it is extremely common to have the same variable on both sides of the expression, C provides a shortcut notation. This is implemented with one of a series of compound assignment operators. Using the same example, if we wanted to add 5 to the variable x, we could write it simply as x += 5. Similar operations may be carried out using one of the many compound operators shown on the previous slide.



As another example, here we are using the compound multiplication assignment operator *=. The expression x *= 5 may be conceptually expanded to x = x * 5. If x has a value of 2 before the statement is executed, then x will have a value of 10 after the statement has executed.

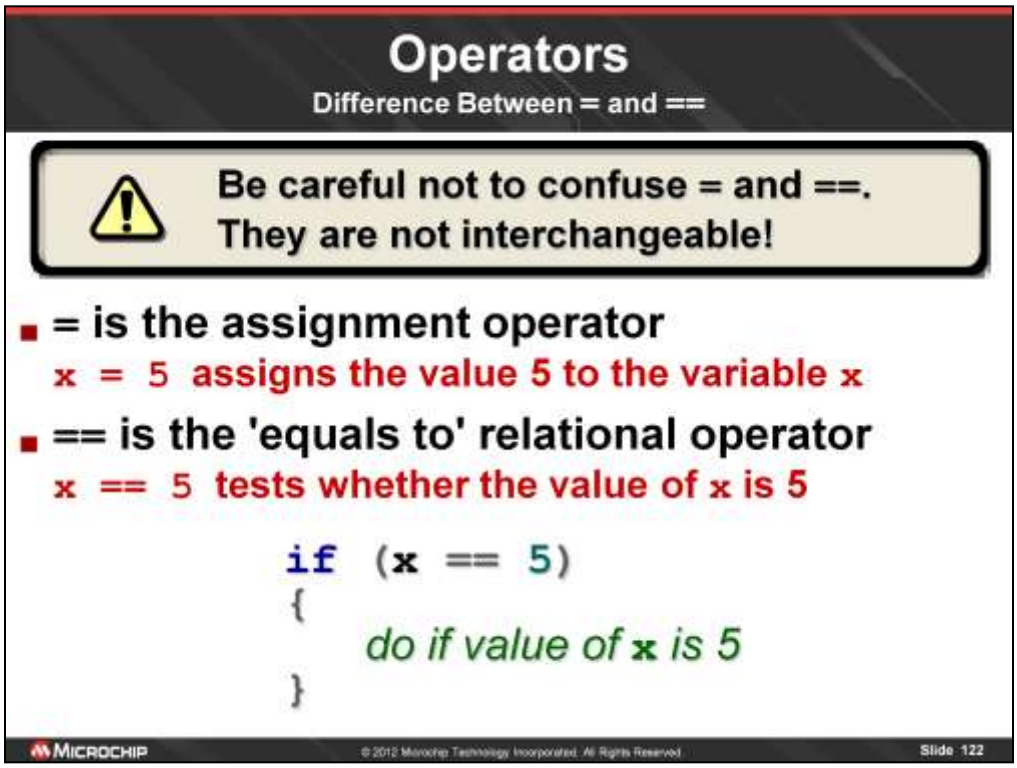
Operators
Relational

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.

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 121

The relational operators are used to compare two values and tell you whether the comparison being made is true or false. For example, in the second line in the table, we are testing to see whether x is less than or equal to y. If it is true, this expression will return a non-zero value (usually 1). If the condition is false, then a value of zero will be returned. Special attention should be paid to the == (equal to) operator – note that it is not the same as the = (assignment) operator. More on this on the next slide.



One of the biggest mistakes new C programmers make is to use = when they mean ==. It is very important to understand that these are two completely different operators.

"=" is used to assign a value to a variable.

"==" is used to compare two values for equivalence.

Here we are getting a quick preview of an if statement. We will discuss it in detail a little later, but for now, just understand that the block of code that follows the if statement will only execute if the expression in the parentheses is true (it evaluates to a non-zero value). In this case, we are checking to see whether x is equal to 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!"
  }
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 123

Before explaining this slide, first ask the class what will happen with this code.

A very common mistake new C programmers will make is to do the following:

if (x = 5) { }

This statement will always be true because x is being assigned the value of 5 (a non-zero value) rather than being compared to 5. An assignment used where a conditional expression is expected will always be evaluated as true unless the expression equals 0. If you do fall into this trap, it can be a very nasty bug to catch since the compiler won't flag it as an error. The only way you will know it is there is that the if block will always run, even in what should be a false condition.

In the example on this slide, the string "Hi!" will be printed to the output window because x is non-zero. If we changed the statement to say:

if (x = 0) { printf("Hi"); }

Then the printf() would never execute because x is always 0.

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.

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 124

The logical operators are typically used in decision making like in an "if" statement, which we will see later. They can be used to selectively execute code based on the outcome of the condition.

In the first example, if both x and y have non-zero values, then the result of the expression will be non-zero (TRUE - usually 1). If either or both x and y are zero, then the result is zero (FALSE).

We will see these in action much more in the section on decision making.

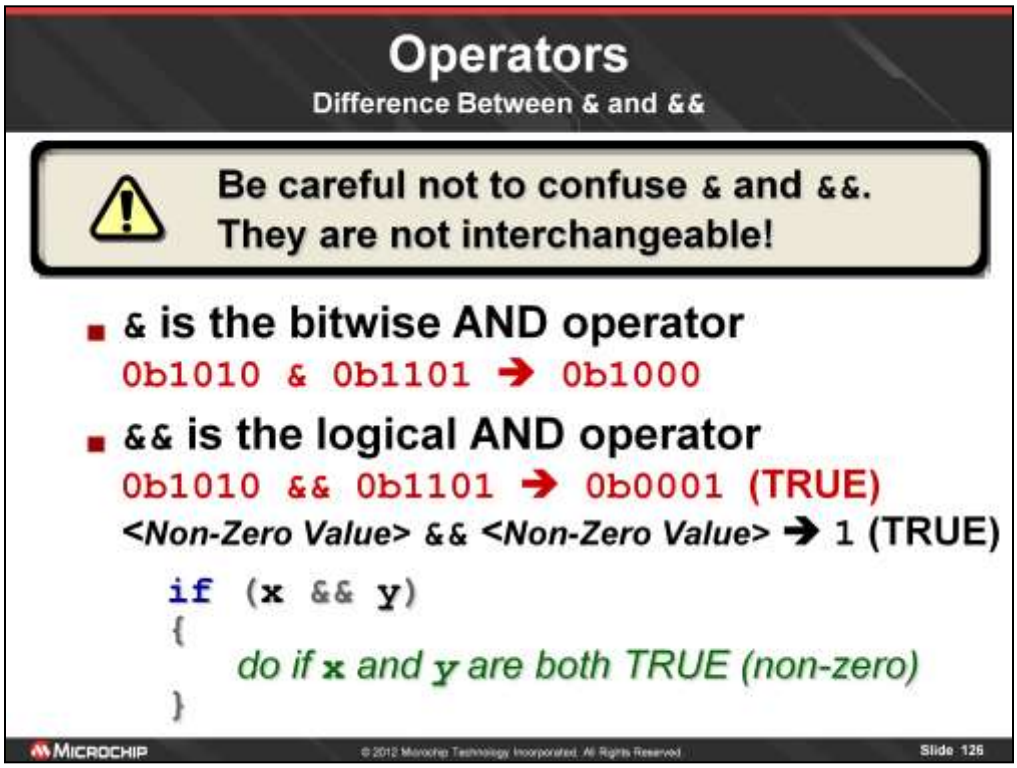
Operators
Bitwise

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**

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 125

The bitwise operators perform Boolean algebra that we all learned in our first class on digital programming. Each of these operators perform their operations on each bit of the operands. For instance, if we have two four bit binary values, the operation will be carried out between bit 0 of each value, then between bit 1 of each value and so on. In other words, these operators do a bunch of single bit operations in parallel.



Much like with the = and == operators, it can be very easy to make the mistake of swapping one of the logical operators for its bitwise counterpart (& and && for example). Logical operators and bitwise operators might sometimes work the same if you are lucky, but they are not the same, and can sometimes yield completely different results.

For example, if we perform a bitwise and "&" between the nibble 0b1010 and 0b1101, we get a result of 0b1000 (just and the bits in the same position together to get the result).

Then, if we perform a logical and "&&" between the same two nibbles, we get a different result -> TRUE, which is usually represented by a value of 1 (or 0b0001 in binary).

Now, note that in this situation, both results would evaluate to a non-zero value, so if used in a conditional expression, they would both be interpreted as TRUE. If however, you were interested in the numeric result of a bitwise AND operation, the result of using "&&" would be completely wrong.

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!");
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 127

Here is another example where you can get into trouble.

In example 1, we use a *bitwise* AND, which results in a zero value, so the text does NOT print:

0b1010
AND 0b0101

0b0000 -- This is a common bug for beginners (and sometimes experts). The evaluation of the expression has a completely different result from the desired logical AND.

In example 2, we use a *logical* AND, which results in a non-zero value, so the text does print:
0b1010 AND 0b0101 -> 0b0001 (TRUE)
Remember: with a logical AND, if both operands are non-zero, then the result is non-zero or *TRUE* (usually a value of 1).

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.

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 128

In C, when a logical operation is being evaluated, if the result is known before all sub expressions have been evaluated, then the evaluation stops, or short circuits.

The two situations where this can occur is when the first expression of a logical AND operation is FALSE (zero) or the first expression of a logical OR operation is TRUE (non-zero). In both of these cases, the result is already known. For AND, if either of the two expressions is FALSE, the result will be FALSE. For OR, if either of the two expressions is TRUE, then the result will be TRUE.

In most situations, this won't be a problem, but you need to be careful not to use the results of the second expression somewhere else in your code because there is no guarantee that it will ever be evaluated.

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.

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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)**

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 130

The shift operators will move the bits in an operand left or right by the specified number of bits.

The left shift is fairly simple and operates the same way regardless of the operand type.
The right shift operates differently, depending on whether the operand is signed or unsigned (see next slide).

With Shift Left, bits moved out on left are lost, and the right side is zero filled.
With Shift Right, if the type is unsigned, the left side is zero filled, otherwise the sign is extended. Bits moved out on the right are lost.

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
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 131

If the operand is unsigned, a shift right performs a LOGICAL shift right in which the bits are moved to the right, and the vacancies on the left are filled with zeros.

If the operand is signed, a shift right performs an ARITHMETIC shift right, in which the vacant bits are filled with whatever the value of the most significant bit was before the shift. This is done to perform a sign extend when working with signed binary values.

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

$y = x / 2^n$ → $y = x >> n$

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Operators

Power of 2 Integer Divide vs. Shift in XC-16

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
```

- 5 Assembly Instructions!
- 21 clock cycles!

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
```

- 3 Assembly Instructions!
- 3 clock cycles!

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 133

Note that the right shift routine only takes three instructions and three cycles. The divide routine takes 5 instructions, but one of those is the divide instruction preceded by a repeat instruction. The code is only slightly larger, but takes much longer to execute.

Operators

Power of 2 Integer Divide vs. Shift on PIC18 with XC8

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 134

The difference is even more dramatic when using C18.

The PIC18 doesn't have a divide instruction of any kind. Here the data is setup, and then a divide routine is called in a math library. So, the code you see on the left is only the tip of the iceberg.

Keep in mind that we are working with 16-bit values on an 8-bit machine, so that is why these routines are much longer than those for the C30 compiler.

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.

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 135

These operators are a bit too advanced to discuss yet. They will be covered in detail in the sections that describe arrays, pointers, structures and unions, where these operators are used. They have been included here for reference purposes only.

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`

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Operators

The Conditional Operator

Syntax

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

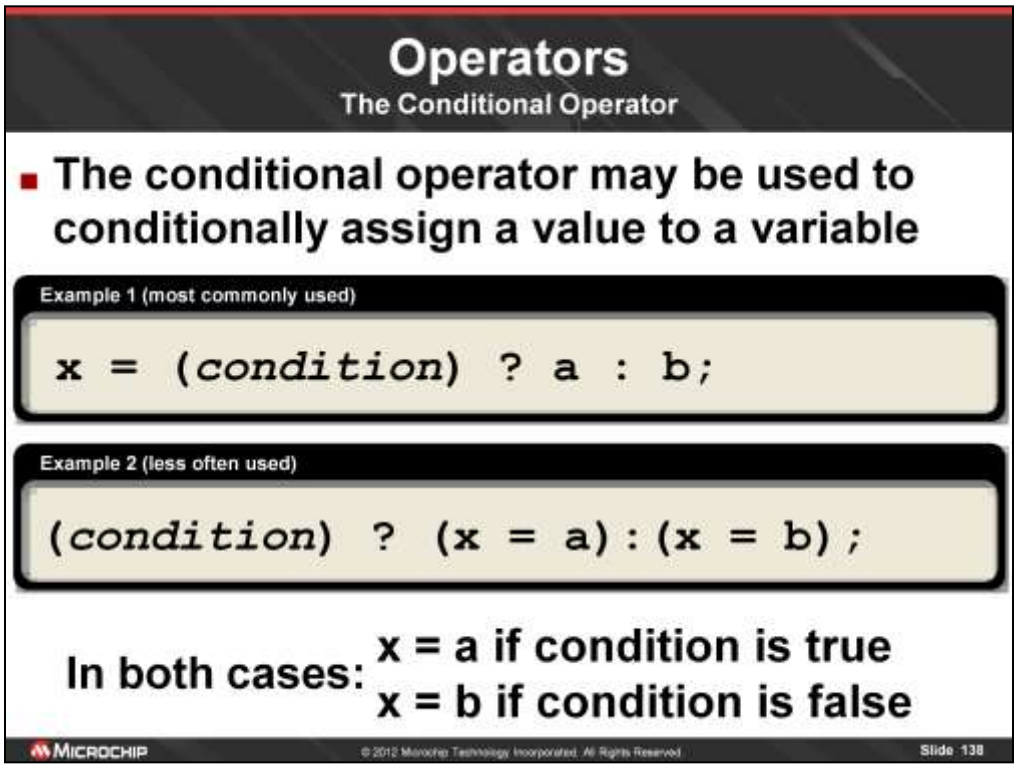
Example

```
int x = 5;

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

Result:

```
5 is odd
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.



For some reason, most students have had the tendency to use the second method in the lab that follows, not realizing that you could do it as shown in example 1. The problem with the second method is that if you forget the parentheses in the part after the '?' this will not compile. And everyone who has tried to do it the second way has left off the parentheses.

Drive home the point that in either method, this assigns one of the two values to x.

The first one requires less typing, and is therefore used more frequently. The second one actually requires one instruction less when compiled, so it is technically more efficient.

Side note:

The first version generates 10 lines of assembly code

The second version generates 9 lines of assembly code

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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...

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 140

This slide and the next two form a single table. The color bands (grey and light blue) indicate different precedence groupings. For example, the first four entries in this table (), [], ., and -> all share the same precedence. These four operators follow the rule of Left-to-Right associativity which is used as a tie breaker when two or more of these appear in the same expression. The next group of operators starting with + and ending with (type) all share the next level of precedence.

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...

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 141

On this slide, the table continues, but there are fewer operators grouped into each level of precedence. Here *, /, and % are all at the same level of precedence (which is lower than those on the previous slide), and they are evaluated from left to right when more than one occurs in an expression (more on this in a few slides).

The next level of precedence includes + and -. So the alternating color bands indicate separate levels of precedence.

Operators
Precedence

Operator	Description	Associativity
=	Assignment	
+= -=	Addition and Subtraction Assignments	
/= *=	Division and Multiplication Assignments	
%=	Modulus Assignment	Right-to-Left
<<= >>=	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**

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 142

Finally, we have another big group of operators with the same precedence – the assignment operator and all of the compound assignment operators.

The lowest level of precedence is the comma operator alone.

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.

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Operators
Associativity

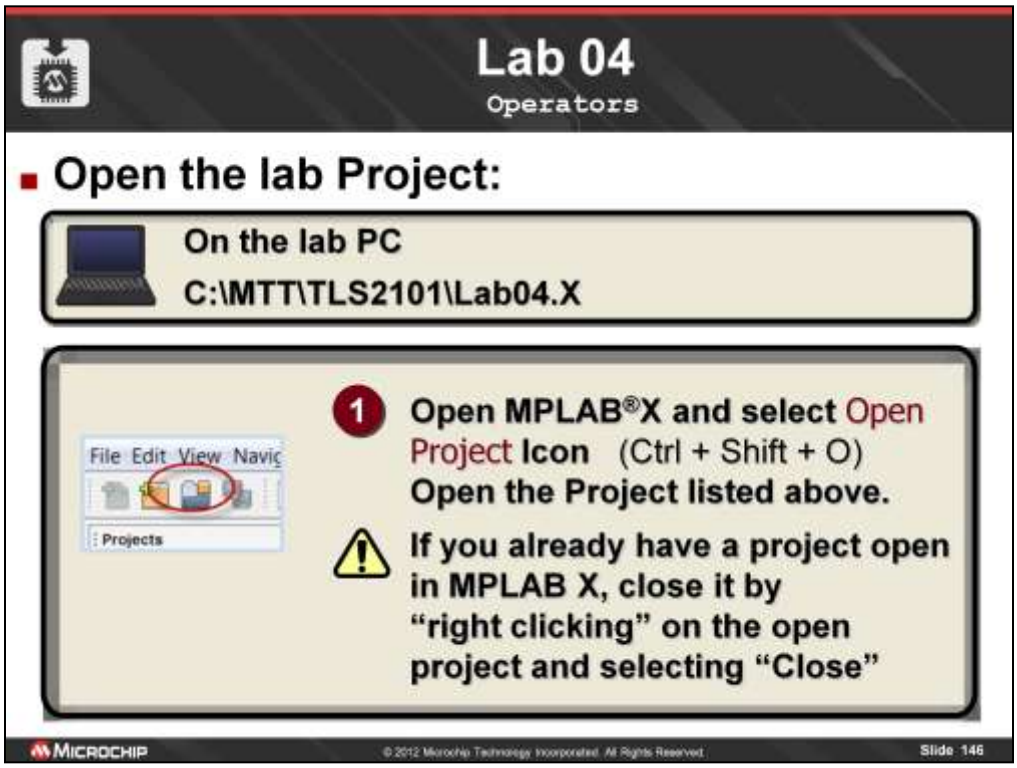
- If two operators have the same precedence, their associativity determines the order of evaluation

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 4
Operators



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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++;
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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;
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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;
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

<end of part one>

© 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
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)

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Expressions

Examples

Example

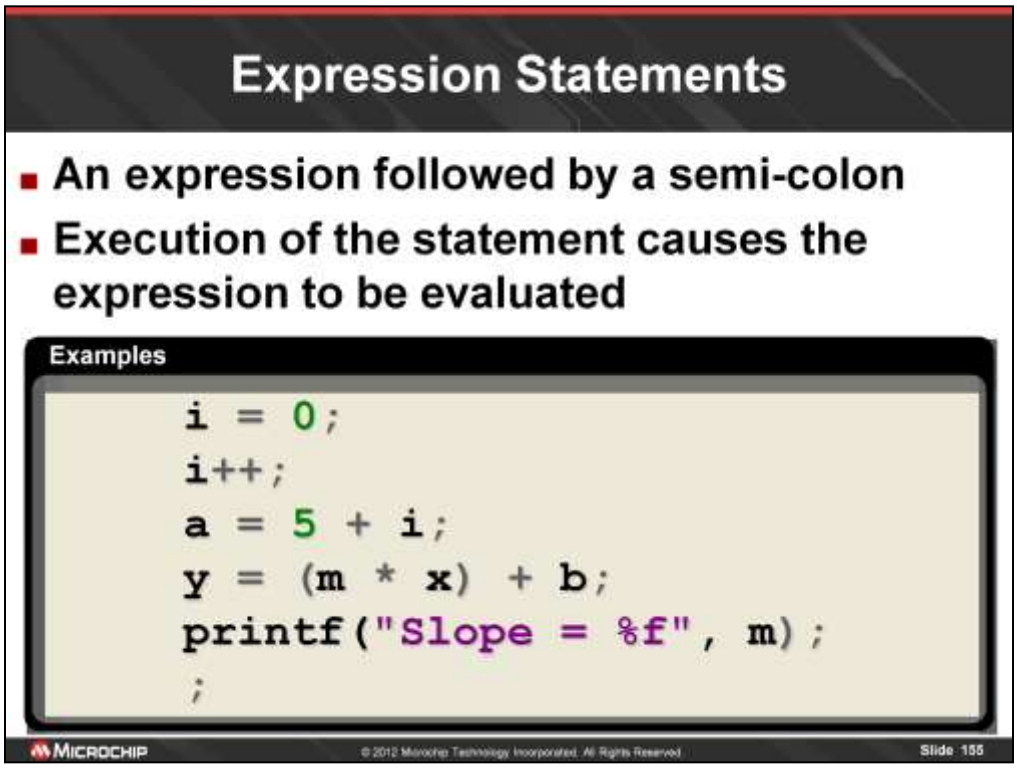
```
a + b
x = y
speed = dist/time
z = ReadInput()
c <= 7
x == 25
count++
d = a + 5
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Statements

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.



An expression statement causes the expression to be evaluated. Any valid expression may be made into an expression statement, though not all expressions make useful statements by themselves (e.g. comparison statements: a<b this makes little sense as a stand-alone expression statement). The semicolon at the end is what turns an expression into an expression statement.

Examples:

i = 0;

This is an assignment statement. It assigns the value 0 to the variable i.

i++;

This is an incrementing statement. It increments the value of i by 1.

a = 5 + i;

This is an assignment statement. It first evaluates the expression 5 + i and assigns the results to the variable a.

y = (m * x) + b;

This is yet another assignment statement, though a bit more complex. m is multiplied by x, that result is added to b and the final result is assigned to y.

printf(…)

This statement causes the printf function to be evaluated. In the example on the slide, if the value of m is 5, the result would be "Slope = 5" in the Uart1 window.

;

This is an empty statement. It does nothing and is often referred to as a null statement. It makes little sense by itself, but is very useful in other circumstances which we will see later on (for loops: for(; ;). Note that this is NOT equivalent to a nop instruction. This generates no code at all.

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

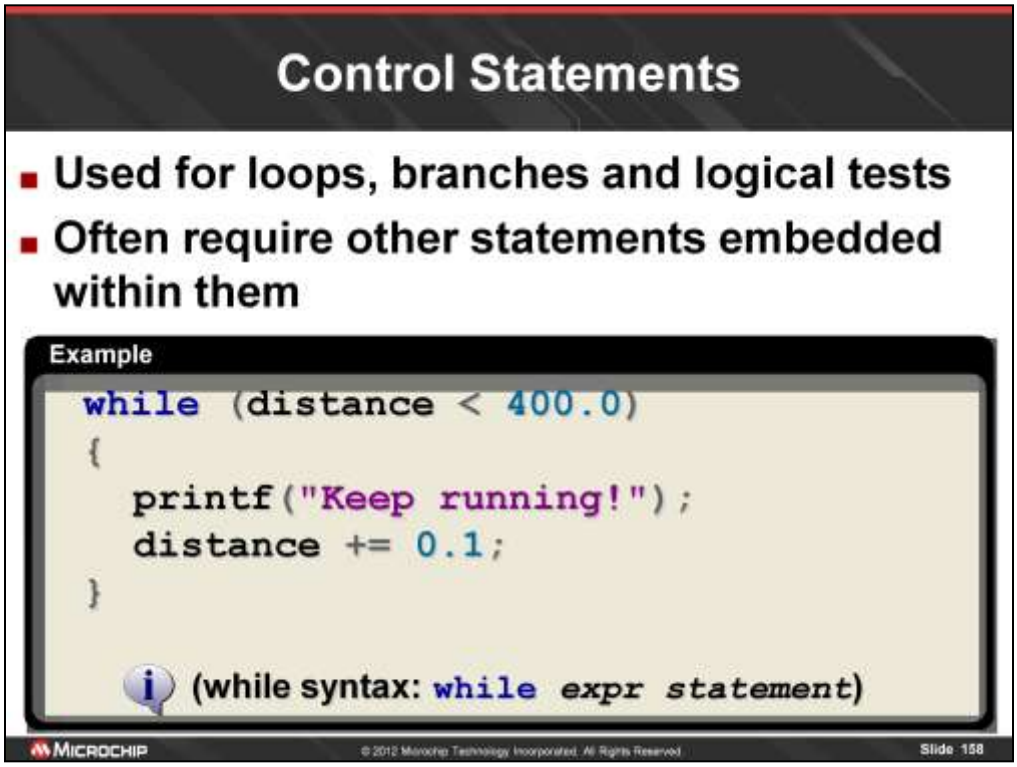
Compound Statements

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);
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.



This statement contains a compound statement which in turn contains two expression statements. The compound statement will continue to run as long as the value of distance doesn't exceed 400.0.

We will see more on the while loop later, but for now look at its syntax. After the while keyword comes an expression followed by a statement. In the example here, the expression is enclosed within parenthesis to separate it from the statement for readability. The statement itself is a compound statement and is therefore enclosed in curly braces.

As a side note, notice that a space and a carriage return are considered the same in a situation like this. The while statement could be written like this:

```
while
(distance < 400)
{ printf("Keep running!"); distance += 0.1; }
```

Or like this:

```
while (distance<400) { printf("Keep running!"); distance += 0.1; }
```

Of course, these are a bit more difficult to read, so most programmers do something like what is shown on the slide.

MICROCHIP
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);
    z = (x > 6);
    while (1);
}
```

y = 1 (TRUE)

z = 0 (FALSE)

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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)

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 161

Note to self: CHANGE THIS SLIDE

`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
}
```

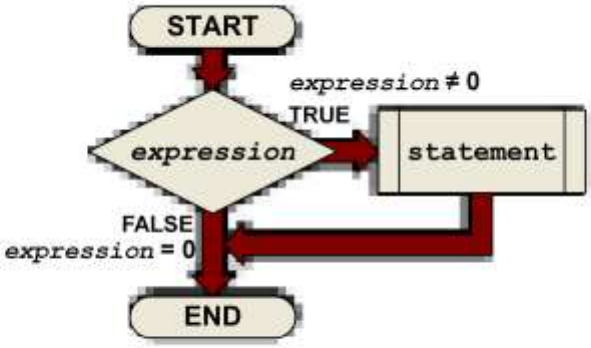

© 2012 Microchip Technology Incorporated. All Rights Reserved.

`if` Statement

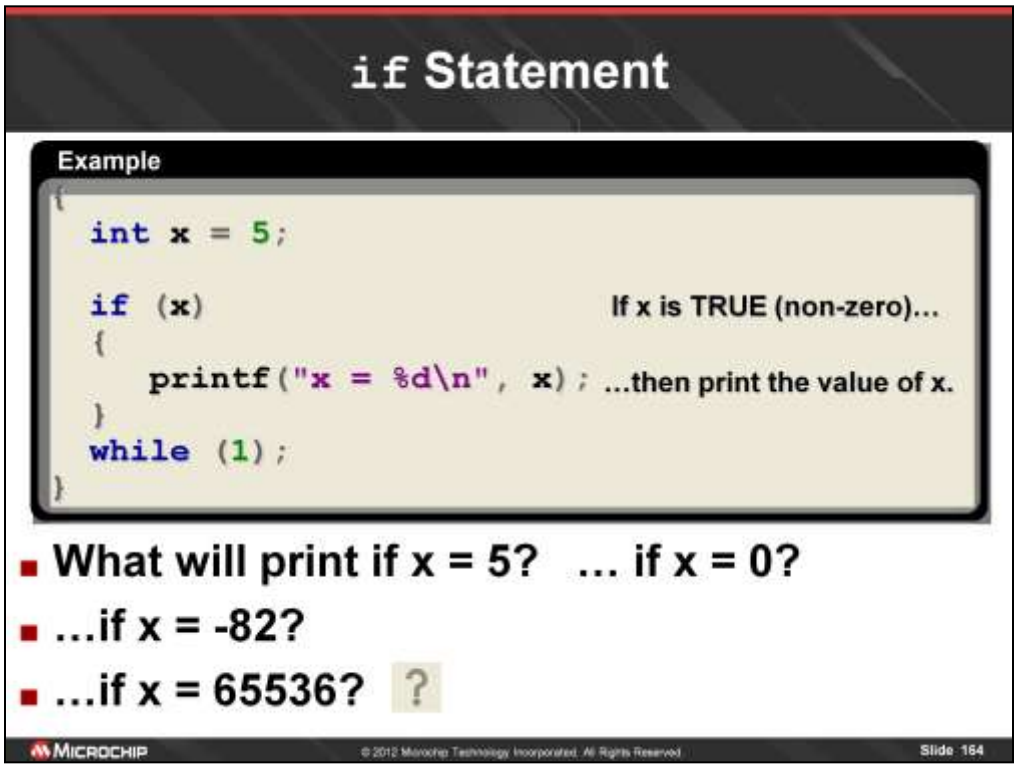
Flow Diagram

Syntax

```
if (expression) statement
```



© 2012 Microchip Technology Incorporated. All Rights Reserved.



This is about as simple as it gets. Here we are going to print the value of x if its value is anything other than 0. So, to answer the questions:

If x = 5, we will print "x = 5".

If x = 0, nothing will print.

If x = -82, we will print "x = -82"

If x = 65536, nothing will print. Trick question. An unsigned integer can only hold a value up to 2^{16}-1 = 65535 = 0xFFFF (signed integer values larger than 32767 are interpreted as negative, but still take up the same number of bits – see below for details). Therefore, we have overflowed the capacity of an integer. In binary, 2^{16} = 65536 = 0x10000 is a 1 followed by 16 zeros. Since an int can only hold 16-bits, the 17th bit is chopped off, leaving us with a value of 0x0000 or false. Using 65536 will result in 0 regardless of whether x is a signed or unsigned int.

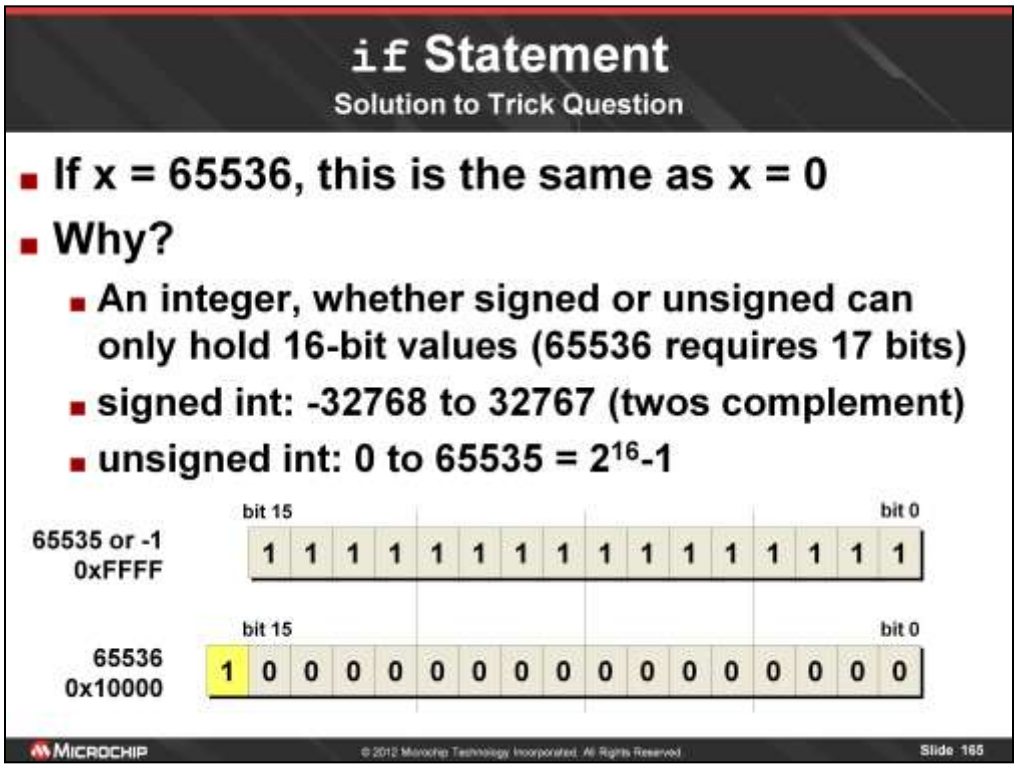
As a point of interest, if you assign a value of 2^{16}-1 = 65535 = 0xFFFF to a signed int:

int x = 65535;

The value of x will be -1, since 65535 = 0xFFFF is the twos complement of 1 = 0x0001 and it represents a value of -1 = 0xFFFF. Remember that a signed int can hold values from -32768 = 0x8000 to 32767 = 0x7FFF.

Always remember that C gives you plenty of rope to hang yourself with!

NOTE TO PRESENTER: If attendees are having difficulty understanding why assigning 65536 to an int is the same as assigning 0, click on the "?" button at the bottom right corner.



NOTE TO PRESENTER: This slide should normally remain hidden and only shown if attendees have difficulty understanding why x = 65536 is the same thing as x = 0. This slide may be accessed by clicking the '?' button at the bottom right hand corner of the previous slide.

If you assign a value of 2^{16}-1 = 65535 = 0xFFFF to a signed int:

int x = 65535;

The value of x will be -1, since 65535 = 0xFFFF is the twos complement of 1 = 0x0001 and it represents a value of -1 = 0xFFFF. Remember that a signed int can hold values from -32768 = 0x8000 to 32767 = 0x7FFF.

Assigning 65536 to a signed int will generate a compiler warning, but NOT an error. So, the code will run, but the result won't be what you indended.

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
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 166

A common mistake made by many new C programmers is to test a variable for equality with 1 to see if it is Boolean TRUE. In addition to being wrong, it also has the consequence of generating more code.

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");
      }
   }
}
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

`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

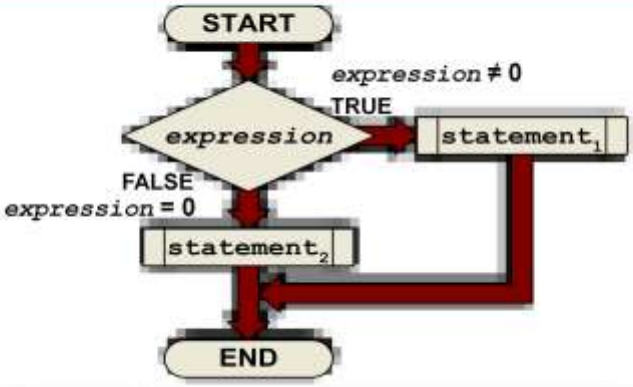
© 2012 Microchip Technology Incorporated. All Rights Reserved.

if else Statement

Flow Diagram

Syntax

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



© 2012 Microchip Technology Incorporated. All Rights Reserved.

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");
  }
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 170

In this example, not only are we seeing a real world if-else statement, but we are also seeing an example of a more complex condition expression. Here, two conditions must be met: the frequency must be greater than 144.0 and it must be less than 148.0 for the first statement to be executed. If frequency does not fall in the range of 144.0 to 148.0, then the second statement will be executed (the one in the else clause). Notice that we are using the logical AND && operator, and not the bitwise and & operator.

For those of you who are wondering, the frequency range represents the US 2m amateur (ham) radio band.

`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

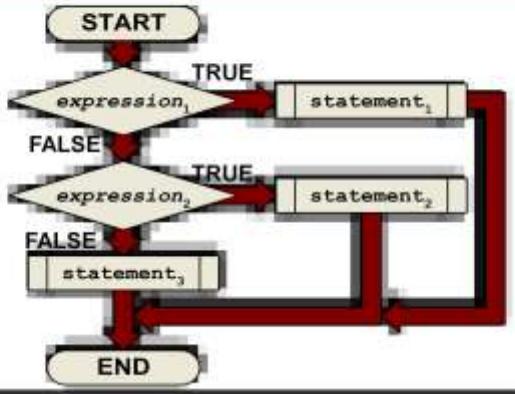
© 2012 Microchip Technology Incorporated. All Rights Reserved.

`if else if` Statement

Flow Diagram

Syntax

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



© 2012 Microchip Technology Incorporated. All Rights Reserved.

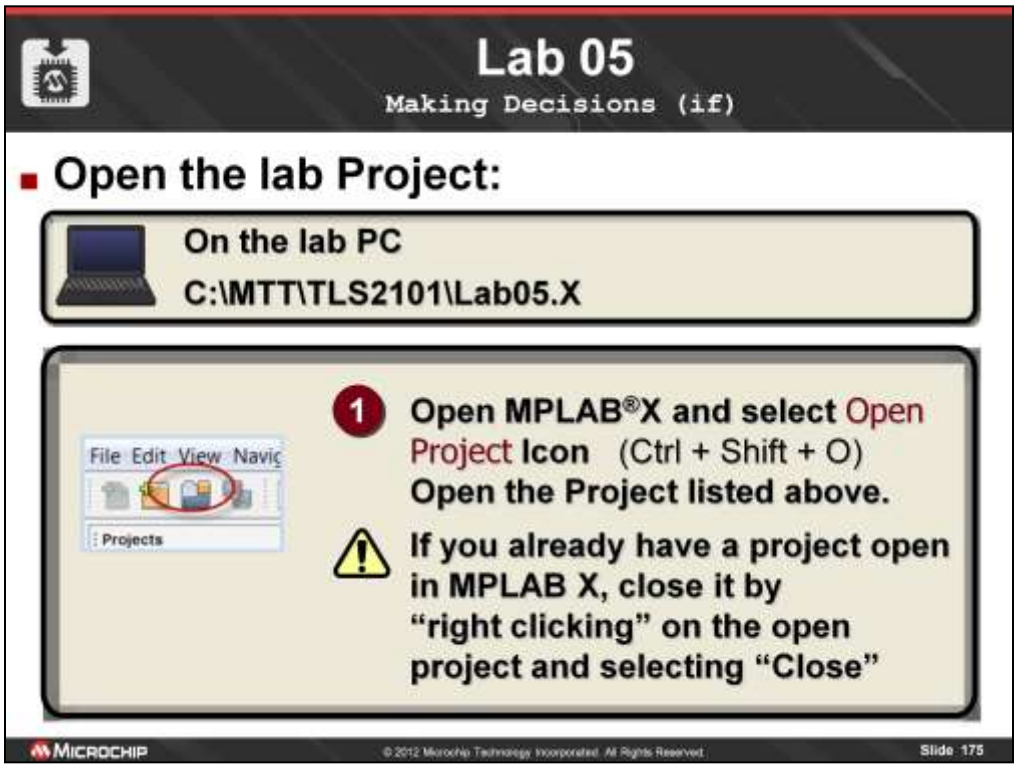
if else if Statement

Example

```
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");
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 5
Making Decisions: if Statements



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 05

Making Decisions (if)

Solution: Step 3

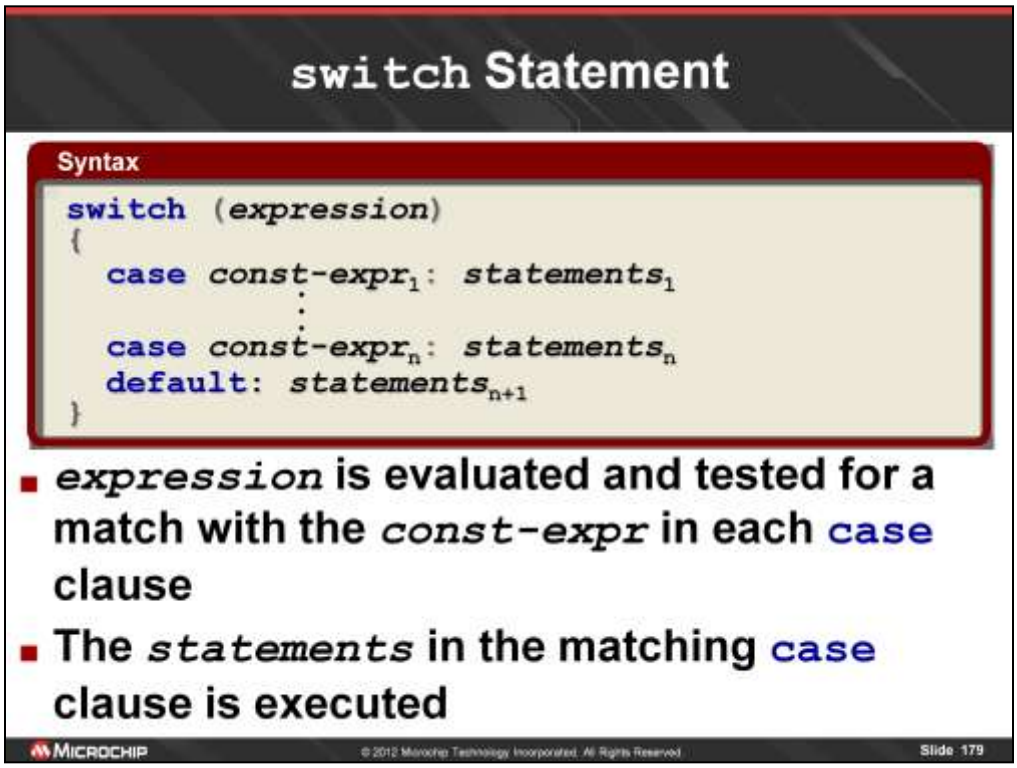
```
/*****************************************************************************
 STEP 3: If neither of the above are true, set charVariable2 equal to 1.
         (HINT: else)
*****************************************************************************/
//Write the else condition
else
{
        charVariable2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

}
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.



The switch statement is a more elegant method of handling code that would otherwise require multiple if statements. The only drawback is that the conditions must all evaluate to integer types (int or char), whereas if statements may use any data type in their conditional expressions.

The expression being evaluated may be any valid C expression, but the result must be one of the integer types (int or char).

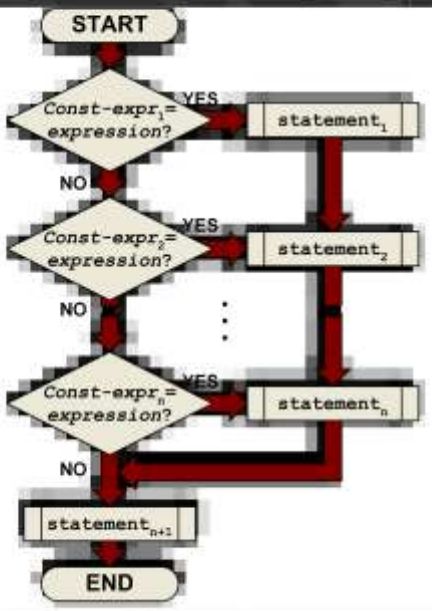
The const-expr (constant expressions) in the case clauses all must evaluate to an integer (int or char) constant – no variables may be used.

The statements may follow with or without curly braces. Each group of statements is associated with a case clause, and the block ends when the next clause begins. The statements may or may not span multiple lines within a case clause.

One of the quirks of the case statement is that the statements start executing in the first matching case clause, but then all subsequent case clauses will be executed too (this is known as fall through). This is the worst possible default behavior since fall through is required less than 1% of the time. To prevent fall through, end each clause's group of statements with a break statement. We'll see an example of this shortly.

switch Statement

Flow Diagram (default)



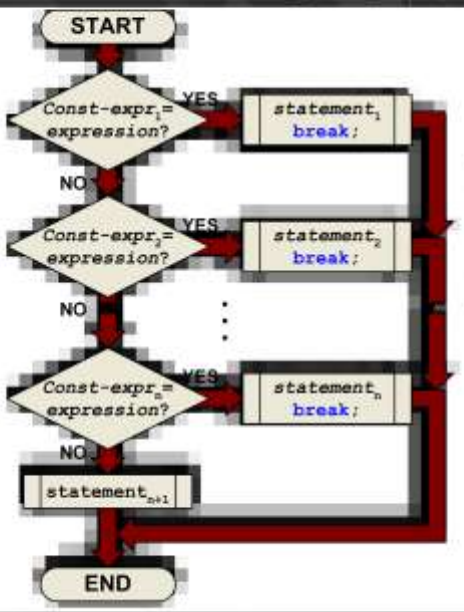
Notice that each statement falls through to the next

This is the default behavior of the `switch` statement

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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");
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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");
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 183

Notice that the code for each case may be split onto multiple lines and that it doesn't have to start on the line of the case clause itself. Again, spaces, tabs and newlines are rarely significant in C.

Also notice that now our constant expressions are characters. Remember that the char data type is really just an 8-bit integer, so it is perfectly legal to use here.

switch Statement

switch Example 3

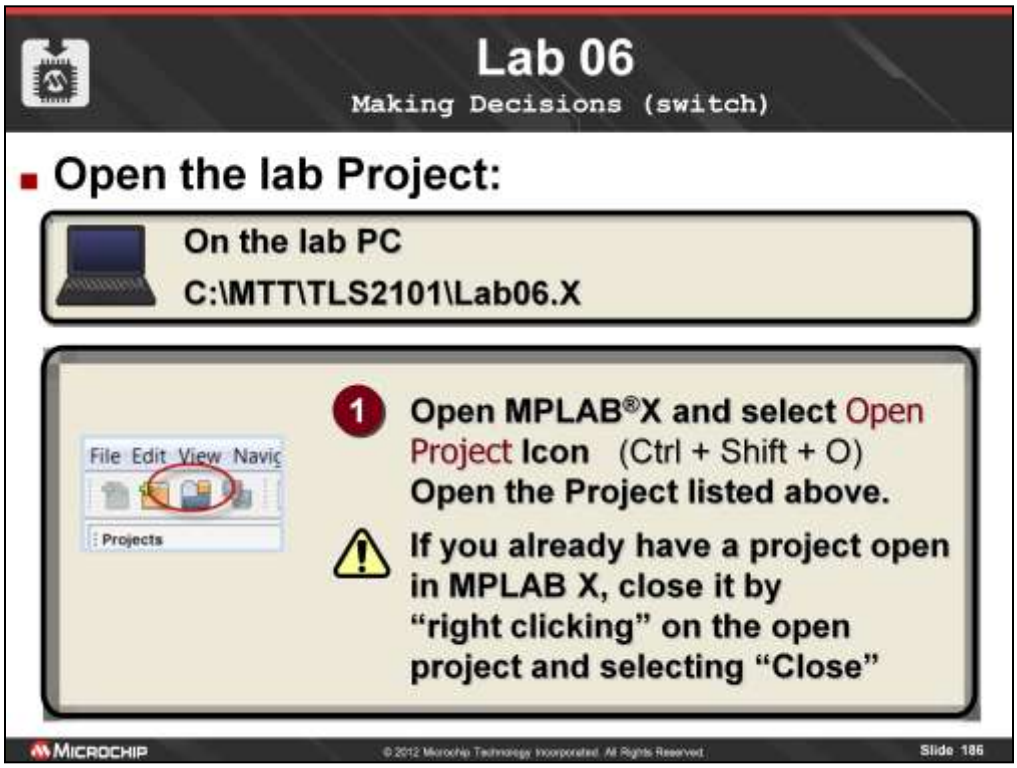
```
switch(channel)
{
      case 4 ... 7:
         printf("VHF Station\n"); break;
      case 9 ... 12:
         printf("VHF Station\n"); break;
      case 3: case 8: case 13:
         printf("Weak Signal\n"); break;
      case 14 ... 69:
         printf("UHF Station\n"); break;
      default:
         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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 6
Making Decisions: switch Statements



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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)
{
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
}
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
}
```

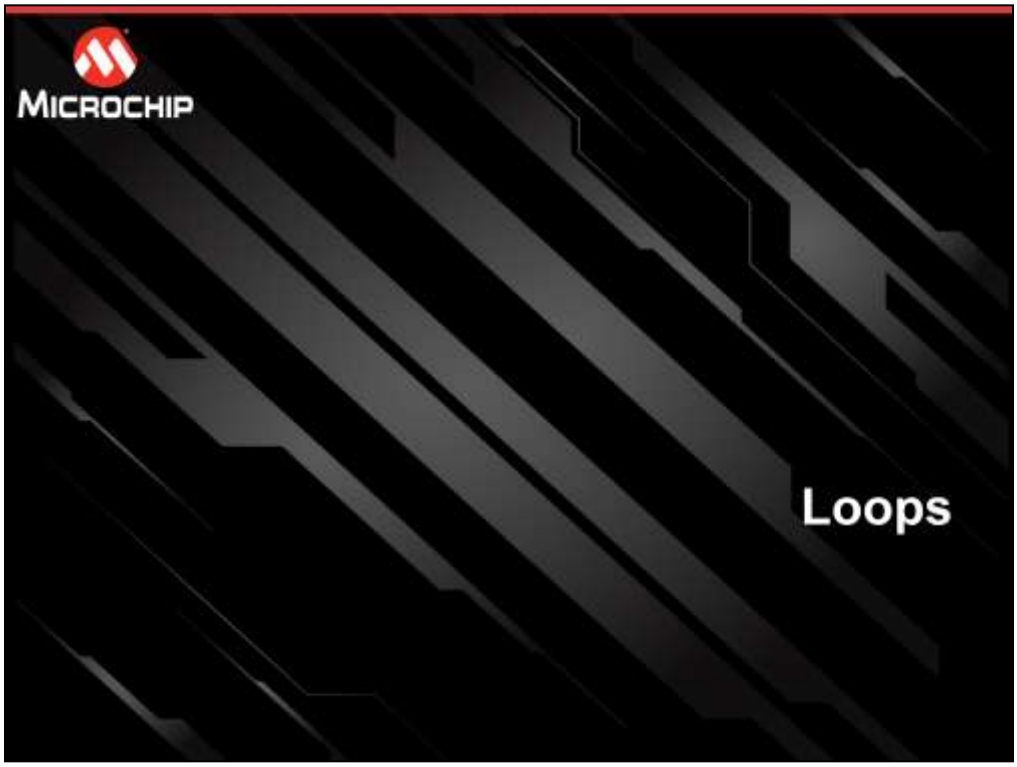

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 06

Conclusions

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.


MICROCHIP
Loops

for Loop

Syntax

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

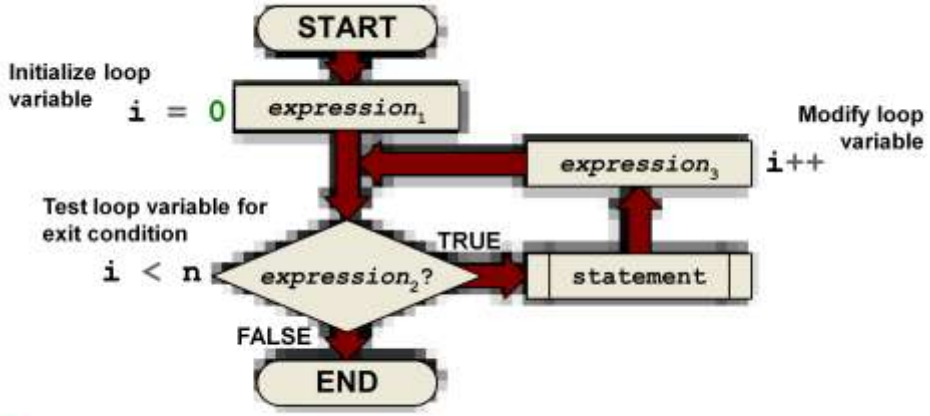
MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

for Loop

Flow Diagram

Syntax

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



© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

`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 ( ; ; )
{
    ...
}
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

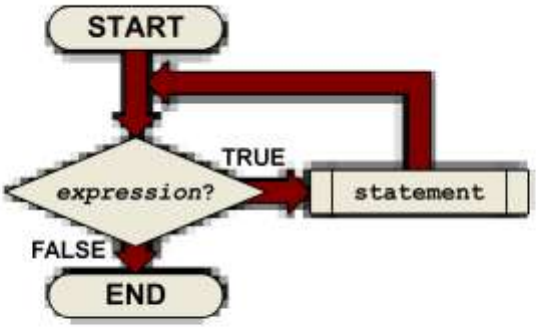
© 2012 Microchip Technology Incorporated. All Rights Reserved.

while Loop

Flow Diagram

Syntax

```
while (expression) statement
```



© 2012 Microchip Technology Incorporated. All Rights Reserved.

while Loop

Example

Example (Code Fragment)

```
int i = 0;

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

Loop counter initialized outside of loop

Condition checked at start of loop iterations

Loop counter incremented manually inside loop

Expected Output:

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


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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)
{
    ...
}
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

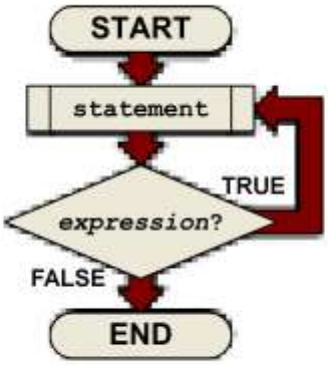
© 2012 Microchip Technology Incorporated. All Rights Reserved.

do-while Loop

Flow Diagram

Syntax

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



© 2012 Microchip Technology Incorporated. All Rights Reserved.

do-while Loop

Example

Example (Code Fragment)

```
int i = 0;

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

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
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

`break` Statement

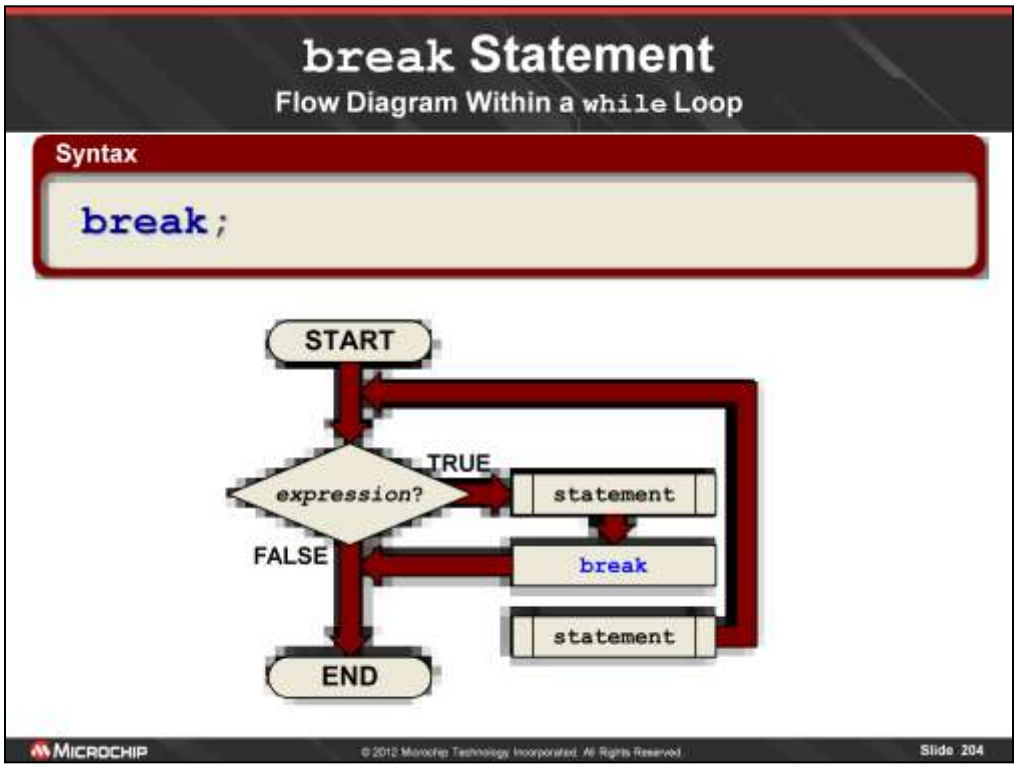
Syntax

```
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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Used for ending loops early



This uses a while loop as an example, but the concept is the same for any kind of loop. If you are in a loop, and you encounter a break statement in the middle of your code, any other code remaining within the loop will be skipped and you will exit the loop immediately.

In this example, both statement blocks in the flow chart are part of the loop's code. However, there is a break statement in the middle of these two blocks, so the second block of statements never gets executed and we exit the loop immediately.

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
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

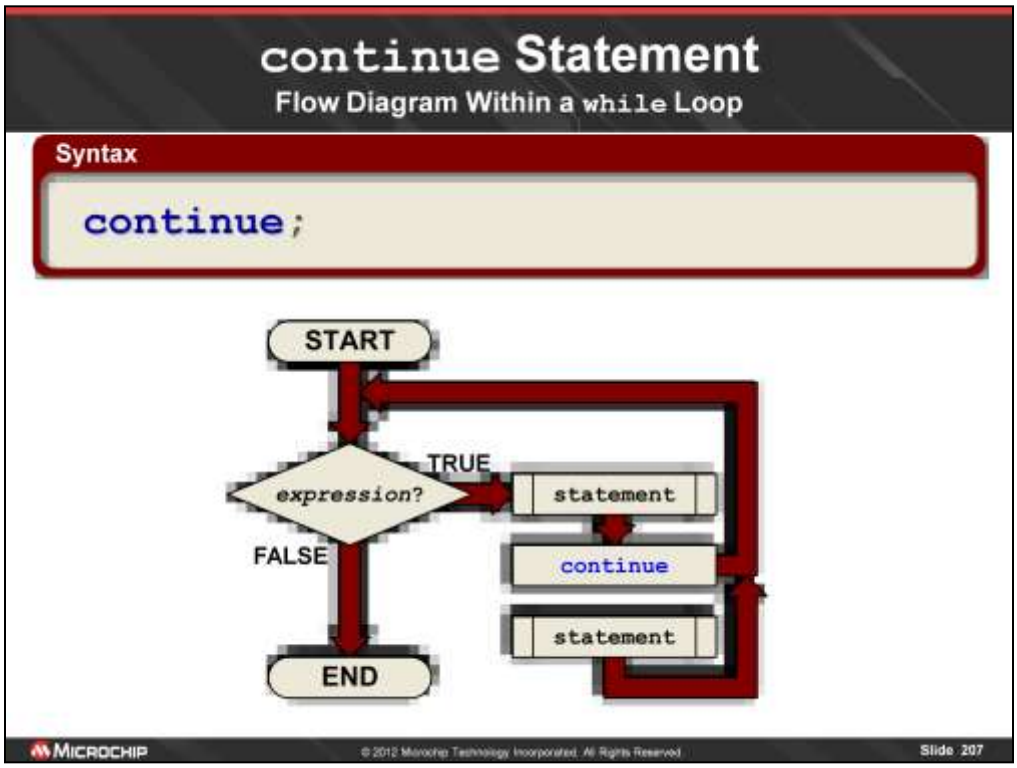
continue Statement

Syntax

```
continue;
```

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.



Similar to break, the principle holds true for all loops, not just while. Again, both statement blocks are part of the loop's code. When we encounter the continue statement, the second statement block gets skipped and we return to the top of the loop.

continue Statement

Example

Example (Code Fragment)

```
int i = 0;

while (i < 5)
{
    i++;
    if (i == 2) continue;
    printf("Loop iteration %d\n", i);
}
```

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

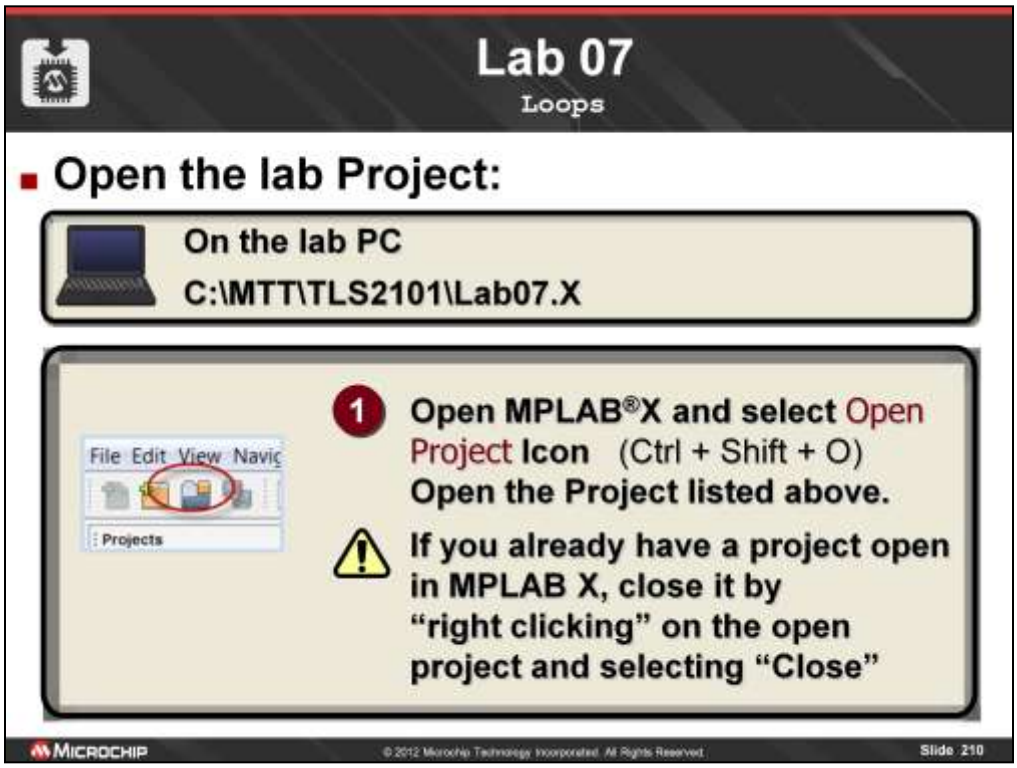
Expected Output:

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

Iteration 2 does not print

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 7
Loops



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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
```

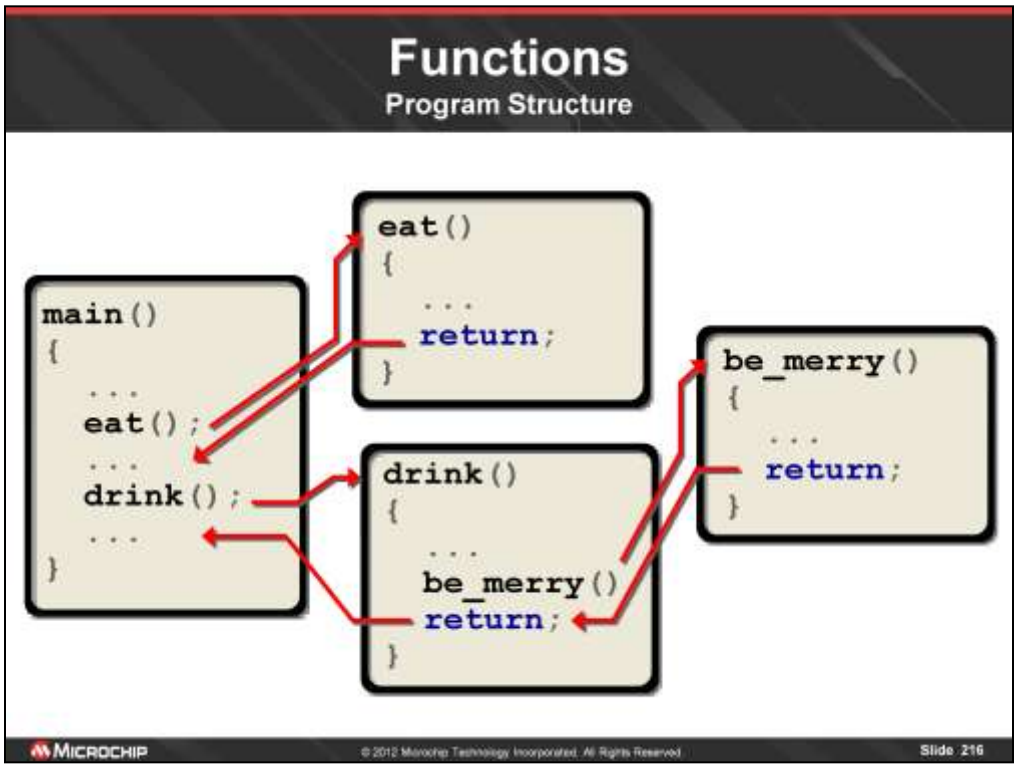
MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Functions



The idea behind functions is that they allow you to modularize your code by breaking it up into several sub-programs that may be called over and over again in the course of a program's execution. As an added benefit, well written functions may be reused in other programs as well.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 217

All functions can be thought of as small, self-contained programs that are designed to work with other functions in the context of a larger program. A well written function should perform a specific, well defined task so that it is easier to develop and debug.

The main() function is special in that it is required in every C program, and it must have the name *main*(). Other functions will be called from within the main() function.

The purpose of most functions is to process information passed to it by the part of the program that called it, though some functions take no parameters at all. In general, functions only return a single value.

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: $f(7) = 7^2 + 4*7 + 3 = 80$

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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)

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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);
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 220

Here are two versions of the same function. They both return the maximum value between x and y, but the first one is more useful for illustrating the possible components of a function. In the first one, we show how to declare a local variable (scope will be discussed shortly).

The second one is more efficient since it doesn't require the local variable z. The second one uses 6 fewer bytes of program memory than the first one – data memory usage is the same. z is temporarily allocated on the stack so no "extra" RAM is required, but more program code is needed to manipulate z.

Description:

(1) The first line of the function is known as the ***function header***. The header contains all of the information about how a program can interface with a function. It lists any parameters that the function will accept, along with their respective data types. It also defines what type of data is returned by the function. In this case, the function will accept two parameters, both of which are integers. Note that we cannot declare the parameters as a list of variables after one data type such as (int x, y) which was perfectly legal to do when declaring ordinary variables. In the header, each parameter variable must be individually listed after its data type. This function also returns an integer value.

(2) In the first function, we declare a local integer variable z to use within the scope of this function. z is only accessible within this function, as are x and y.

(3) In both functions, we use the conditional operator ?: to determine which parameter has the largest value. The first function assigns the maximum value to z first, then it returns the value of z to the program that called the function. In the second version of this function, we bypass the need to use a local variable and simply return the value of the expression directly.

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*

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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;
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 222

While a function can only return one value via the return statement, you may have multiple return statements that are conditionally executed. In this example, only one of the two return statements will be executed. If a is greater than b, then the first return statement will return the value 1. If a is less than or equal to b, then the second return statement will return the value 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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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 <u>not</u> work
  - `int maximum(int x, int y)` is correct

Example

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


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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;
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

The parameter names declared in the function header are local to the function, meaning that they are not valid anywhere outside of the function. Their names may be the same as other variables in the main program or in other functions, though they don't need to be (and usually aren't).

If a function doesn't take any parameters, the parameter list must be replaced by the keyword *void*.

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);
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Functions

Function Prototypes

- Function prototypes may take 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);
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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* <u>before</u> it is used in main()

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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 <u>before</u> use in main()

Function is *defined* <u>after</u> it is used in main()

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Functions

Passing Parameters by Value

Example

```
int a, b, c;
int foo(int x, int y);

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

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.

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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));
}
```

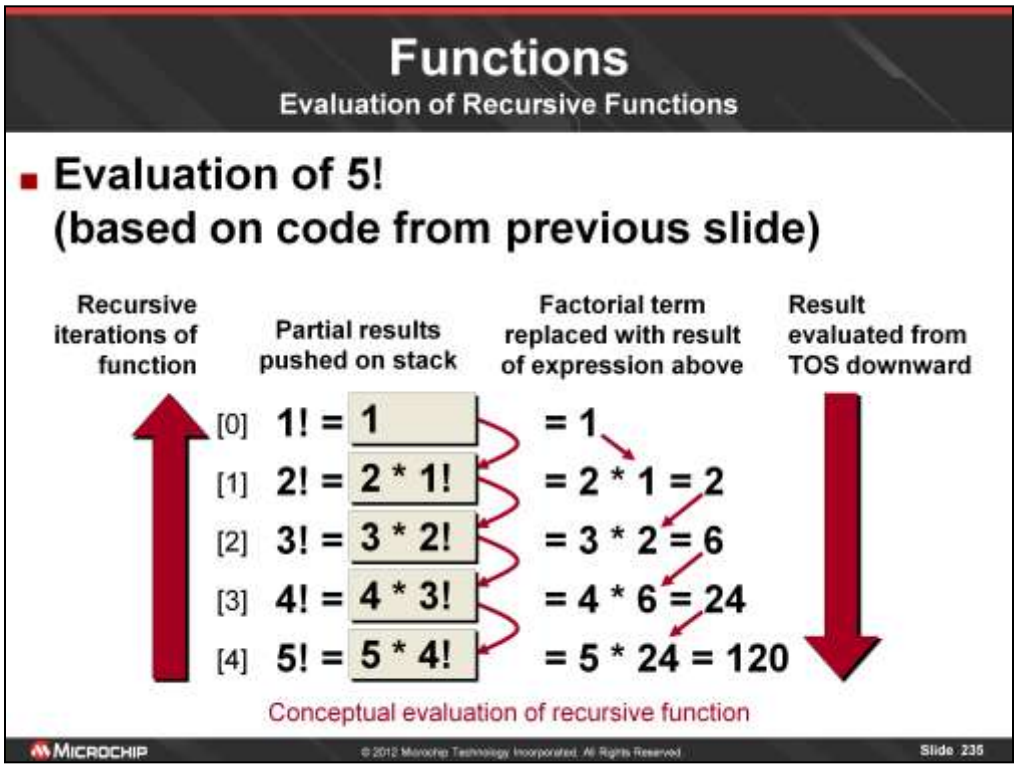
MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 234

C functions also support recursion – a concept that gives most beginning (and many experienced) programmers severe headaches as they try to wrap their minds around the execution of a recursive function.

The idea behind recursion is that a function can call itself in order to complete an iterative process where each successive step is defined in terms of the previous result. The best, and probably easiest example where a recursive function is more efficient than a non-recursive one is the calculation of factorials.

PRESENTER'S NOTE: If someone asks how to do a factorial function non-recursively, this is one way:

```
long int factorial(int n)
{
        long int f = 1;     // initialize f for the case of n=1 (1!)
        while (n > 1)       // repeat the following line until the value of n <= 1
                            f *= n--;          // f = f * n, then n = n - 1
        return f;                              // return the value of f
}
```



The factorial function on the previous slide would be evaluated like this if it were passed a value of 5. This is primarily a conceptual process for the evaluation of this function, and is not necessarily the actual way the function is evaluated. The stack shown here is not necessarily the same thing as the main stack implemented by the C compiler.

In the first iteration, the partial evaluation of 5! = 5 * 4! is pushed onto the stack. It will not be fully evaluated until 4! is calculated.

In the second iteration, the partial evaluation of 4! = 4 * 3! is pushed onto the stack.

In the third iteration, the partial evaluation of 3! = 3 * 2! is pushed onto the stack.

In the fourth iteration, the partial evaluation of 2! = 2 * 1! is pushed onto the stack.

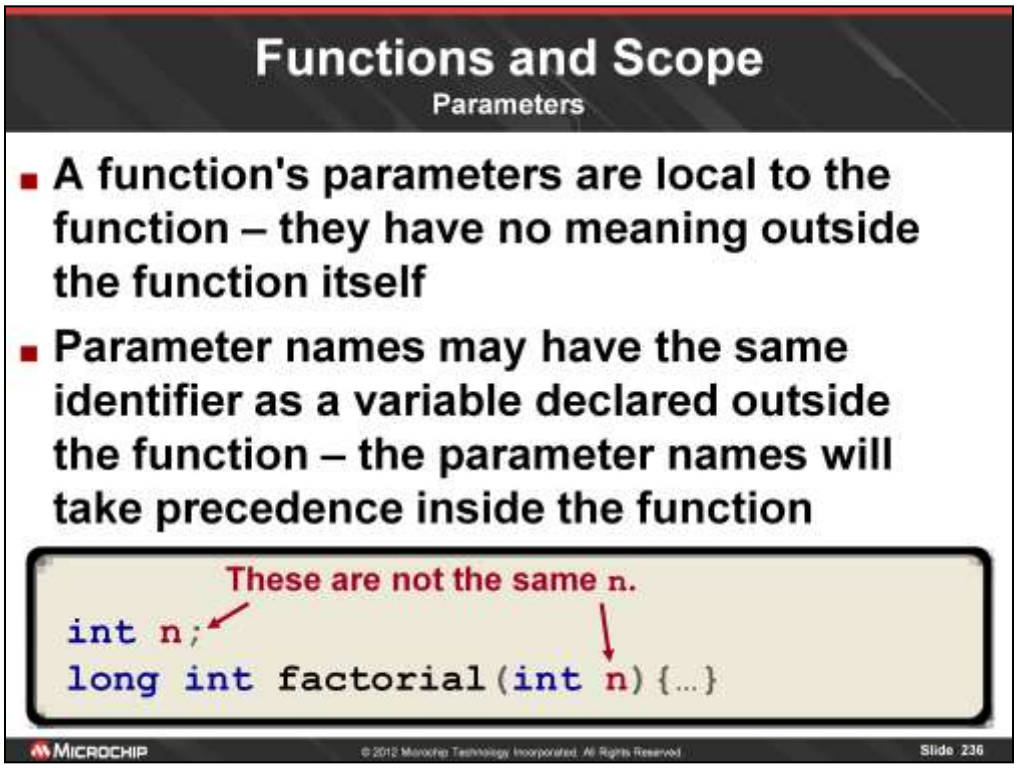
In the fifth iteration, the full evaluation of 1! = 1 is pushed onto the stack.

Now, the 1! on level [1] is replaced with the value 1 from level [0] and level [1] is evaluated to 2.

Next, the 2! on level [2] is replaced with the value 2 from level [1] and level [2] is evaluated to 6.

Next, the 3! on level [3] is replaced with the value 6 from level [2] and level [3] is evaluated to 24.

Next, the 4! on level [4] is replaced with the value 24 from level [3] and level [4] is evaluated to 120, which is the final result.



Any variables declared as a parameter to a function only exist within the context of the function. In other words, the rest of the program does not have any ability to directly read or modify parameter variables aside from passing values to them in a function call.

Because of this, parameter names may be the same as another variable name declared outside of the function or in another function. When the function does something with n from the example here, the n that it uses is the one in the parameter list, effectively making the "global" n inaccessible from within the function. This can be avoided by giving the parameter a different name from the "global" variable. Ordinarily, a function can access "global" variables just like any other part of the program.

Functions and Scope

Variables Declared Within a Function

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

Example

```
int x, y, z;

void 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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Just like with function parameters, if a variable is declared within the function body, it will be local to the function. So, if a variable of the same name is declared globally, code within the function can only access the local variable, whereas code outside the function can only access the global variable. Also, just like function parameters, the same name may be used for variable declarations in different functions. Each function will only use its version of the variable by that name.

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 = 2;
      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.

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 238

Just like with function parameters, if a variable is declared within the function body, it will be local to the function. So, if a variable of the same name is declared globally, code within the function can only access the local variable, whereas code outside the function can only access the global variable. Also, just like function parameters, the same name may be used for variable declarations in different functions. Each function will only use its version of the variable by that name.

Functions and Scope

Global versus Local Variables

Example

```
int x = 5;
int foo(int y);

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 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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Functions and Scope

Parameters

- "Overloading" variable names:

n Declared Locally and Globally

```
int n;

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

local n hides global n

n Declared Globally Only

```
int n;

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

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Functions and Scope

Parameters

Example

```
int n;

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

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

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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.

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 242

The value of a constant will be determined by the closest #define above the symbol's usage. For example:

Why? Because the closest #define for x above its use defines x = 5. The order of the code's execution doesn't matter here. The value is determined purely by its position within the source file's text. When using #define, just think of your source code as an ordinary linear text file. The label's value is valid from the line of the file where it is defined all the way to the end of the file unless it is redefined along the way.

Functions

Historical Note

- C originally defined functions like this:

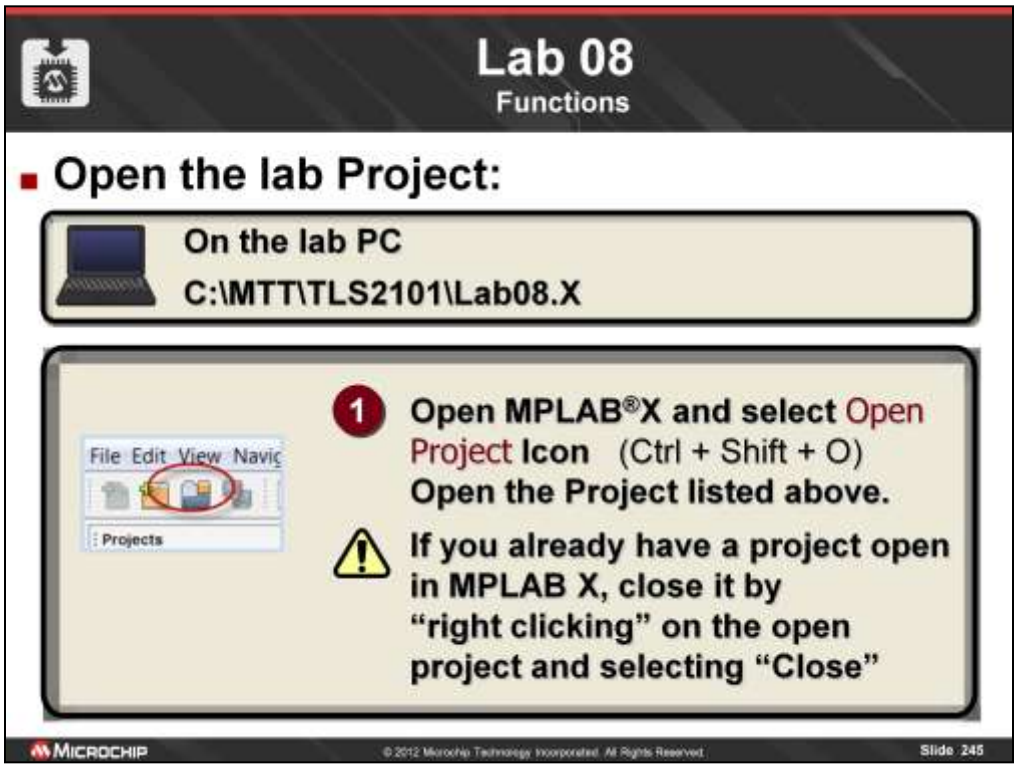
```
int maximum(x, y)
int x, int y
{
  return ((x >= y) ? x : y);
}
```

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

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


© 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 8
Functions



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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 );
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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
}
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 08
Conclusions

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

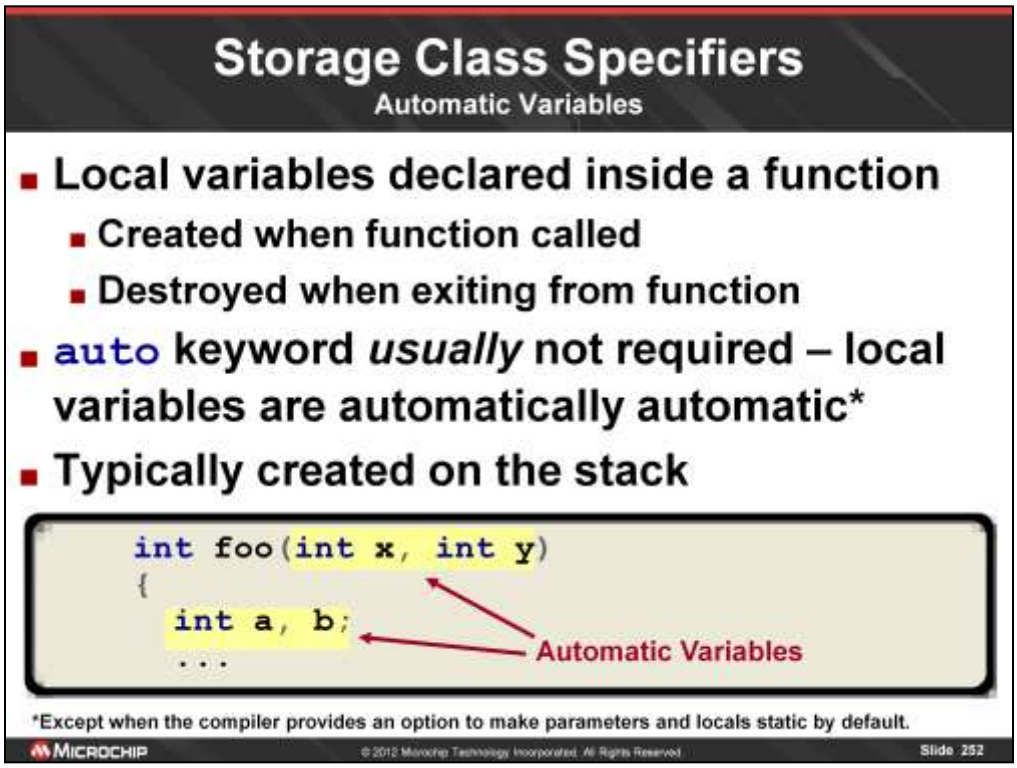
MICROCHIP
Multi-File Projects and
Storage Class Specifiers

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.



•**Local variables declared inside a function**

Any variable declared as a parameter to a function or inside the body of the function are automatic by default (as shown in the code snippet). An automatic variable is created on the stack when the function is called, and it is destroyed when exiting from the function. This means that they only exist as long as the function is active, so that if a function is called repeatedly during a program's execution, the parameters and variables will be freshly allocated, brand new entities that know nothing about their previous existence (if any). Therefore, any value they may be assigned during execution of the function will be lost when returning from the function call.

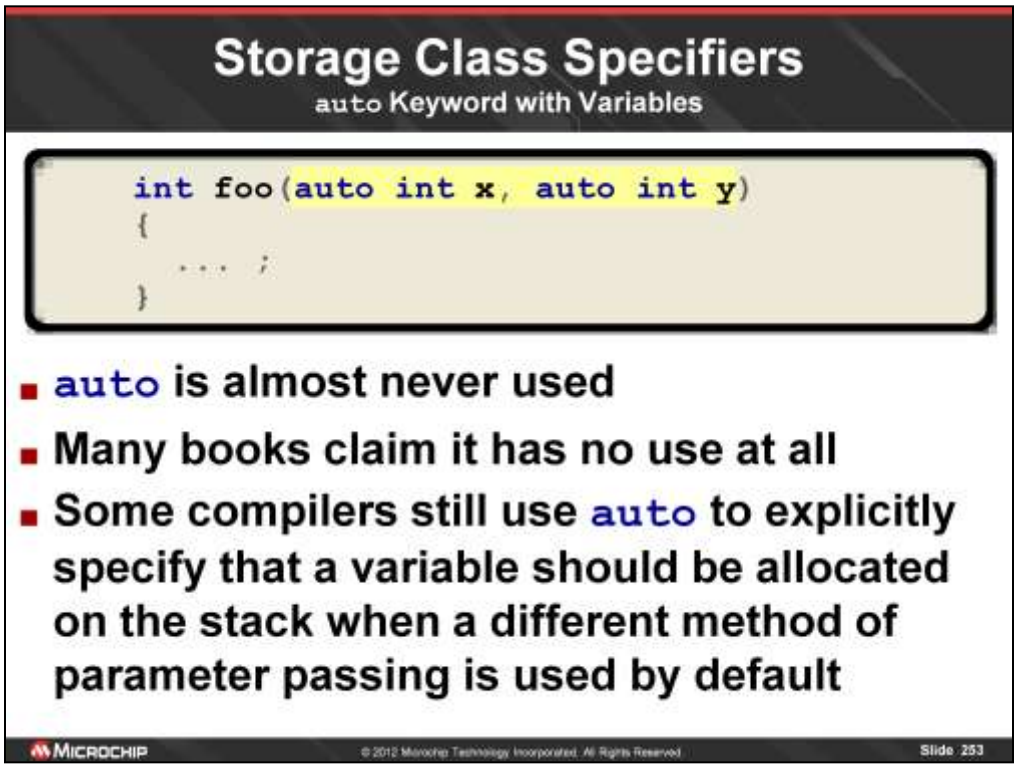
•**auto keyword *usually* not required – local variables are automatically automatic**

According to most books on C, the auto keyword serves no purpose whatsoever since it can only be used inside functions (not applicable to global variables) and those parameters and local variables are automatic by default. While this may be true in the world of the PC and mainframe, embedded systems are a little different.

Most embedded C compilers do make parameters and local variables automatic by default, they will often have an option to make them static by default. The advantage to static variables is that it takes less code space to access them. The disadvantage is that they always exist, even when the function is not in scope, and therefore take up RAM. If the default static option is selected (this is an option with MPLAB-C18), then the auto keyword may be used to override that option in cases where having an automatic variable would be advantageous.

•**Typically created on the stack**

Most compilers allocate automatic variables and parameters on the software stack. However, due to the unique architectures of some embedded processors, this is not always the case. For example, MPLAB-C30 passes function parameters in the PIC24's or dsPIC's working registers instead. If there aren't enough working registers to accommodate the parameters, then the software stack is used. (NOTE TO INSTRUCTOR: *There are a few other cases where the stack might be used instead – see the C30 User's Guide if you want details. This will be covered in greater detail in part 2 of this class which covers the compiler and architecture specific implementation details.*) MPLAB-C18 on the other hand, handles parameters in the usual way via the software stack – unless the default static option is enabled (more on this in a couple of slides).



•auto is no longer used by most compilers

Since function parameters and local variables are usually automatic by default, the auto keyword is almost never used. This is particularly true in the world of PC programming. However, in the embedded world, auto still has its uses. As mentioned on the previous slide, on some architectures, static variables (global variables with a permanent address – discussed on the next slide) are much more efficient in terms of the amount of code required to access them. Therefore, compilers often have an option to make parameters and local variables static by default. If this is the case, auto is useful to make individual variables automatic in circumstances where stack allocation is more advantageous for a particular variable, while leaving all others static.

•Many books claim it has no use at all

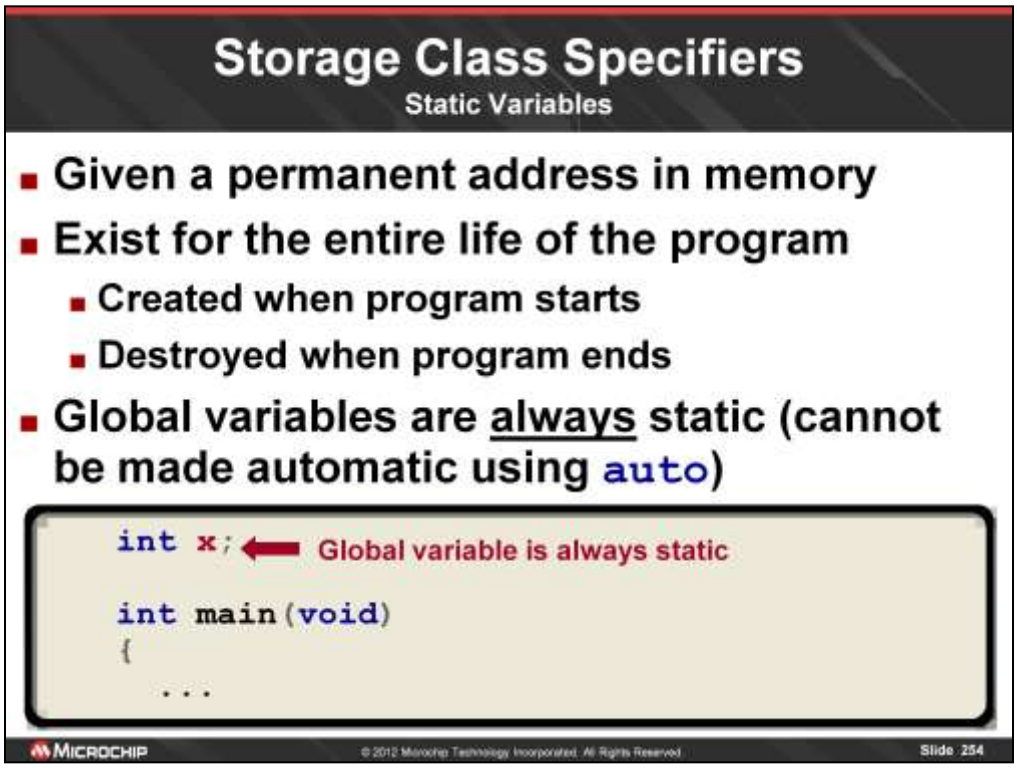
Yet another example of the great divide between PC programming and embedded systems programming. In fact, many books on C get it completely wrong. For example, C for Dummies states:

> *A keyword named **auto** is in the C language's list of keywords. It's a reserved word, so you cannot name a function or variable **auto**. What does **auto** do? Nothing.*
>
> ***auto** is a holdover from the B language, from which C was derived. **auto** was once used to describe a type of variable, but now it serves no purpose in C.*

Don't believe everything you read. The first two sentences are correct. After that it falls apart – at least from the perspective of an embedded systems programmer.

•Some compilers still use auto…

Already explained above and on the previous slide's notes.



•Given a permanent address in memory

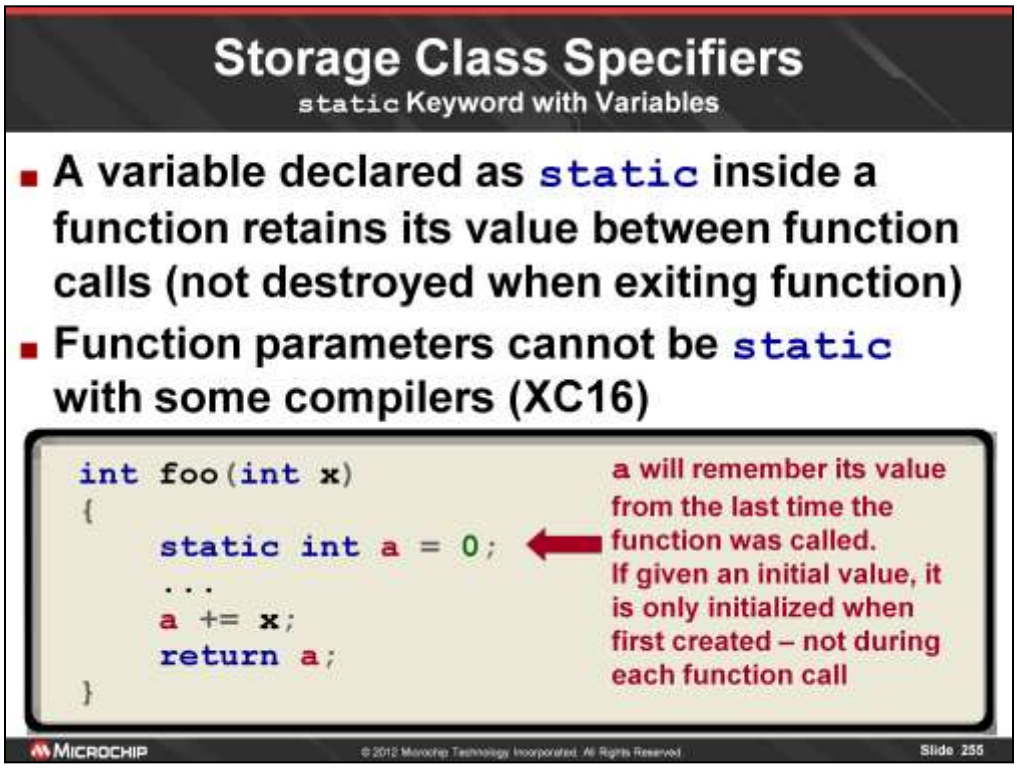
When the compiler encounters a global variable or a local variable prefixed by the **static** keyword, it will allocate space for it in RAM, giving it a permanent (static) address. This is in contrast to **auto** variables which are allocated on the stack.

•Exist for the entire life of the program

A static variable is created when the program first starts, and stays alive until the program exits. In an embedded system, this means that static variables always exist – the only time they disappear is when power is removed.

•Global variables are always static

A global variable (declared outside of any function) has no choice – it must always be static. You cannot use the auto keyword with a global variable.

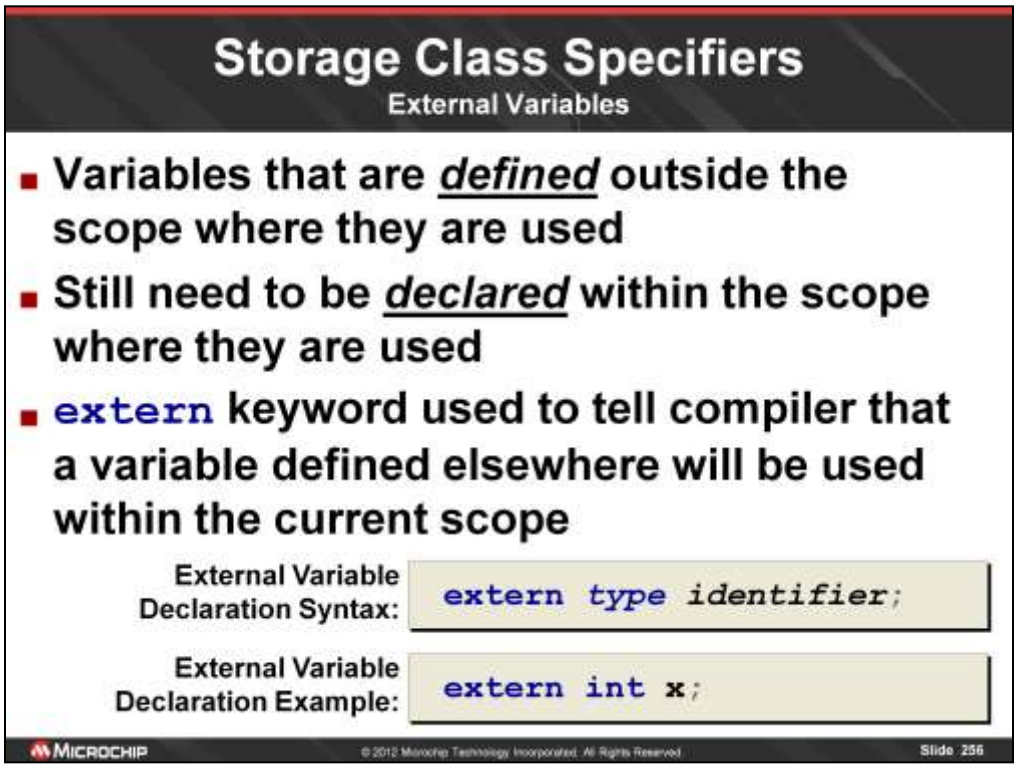


•A variable declared as static inside a function…

Static variables within functions behave much like global variables in that they have a permanent address in memory. However, if you try to access the variable outside of the function, the compiler will complain that the variable doesn't exist. A static variable inside a function may only be accessed from within the function, but it will retain its value across function calls just like a global variable. This could be used to keep track of how many times a function has been called from within the function and without the use of global variables.

•Function parameters cannot be static with some compilers

Unlike variables declared within a function, some compilers including C30, do not allow function parameters to be static under any circumstances – they must always be automatic. This isn't generally a problem since C30 passes parameters very efficiently through the working registers. However, C18 does allow static parameters, which can be accessed more efficiently than automatic variables on that architecture. This can be advantageous with the PIC18's relatively small memory size compared to other processors that are typically programmed in C.



•Variables that are *defined* outside the scope where they are used

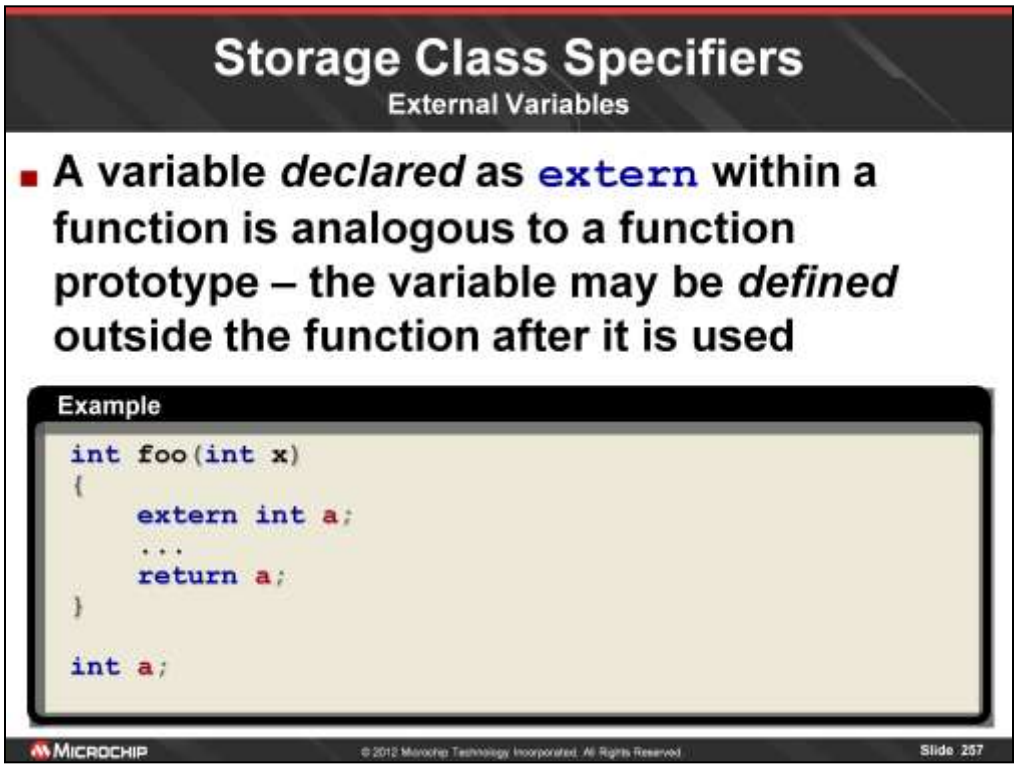
Sometimes we will want to access a variable that is *defined* someplace where we normally couldn't access it, such as in a separate file, or some place after the location in a program where we want to use the variable.

•Still need to be declared within the scope where they are used

Even though the variable is defined someplace within our project, it is still required that it be declared before it may be used. When you declare a variable, no memory is allocated. You are simply telling the compiler about its characteristics (name and type). The code will ultimately be connected to where the variable is allocated in memory during the link process.

•extern keyword used to tell compiler…

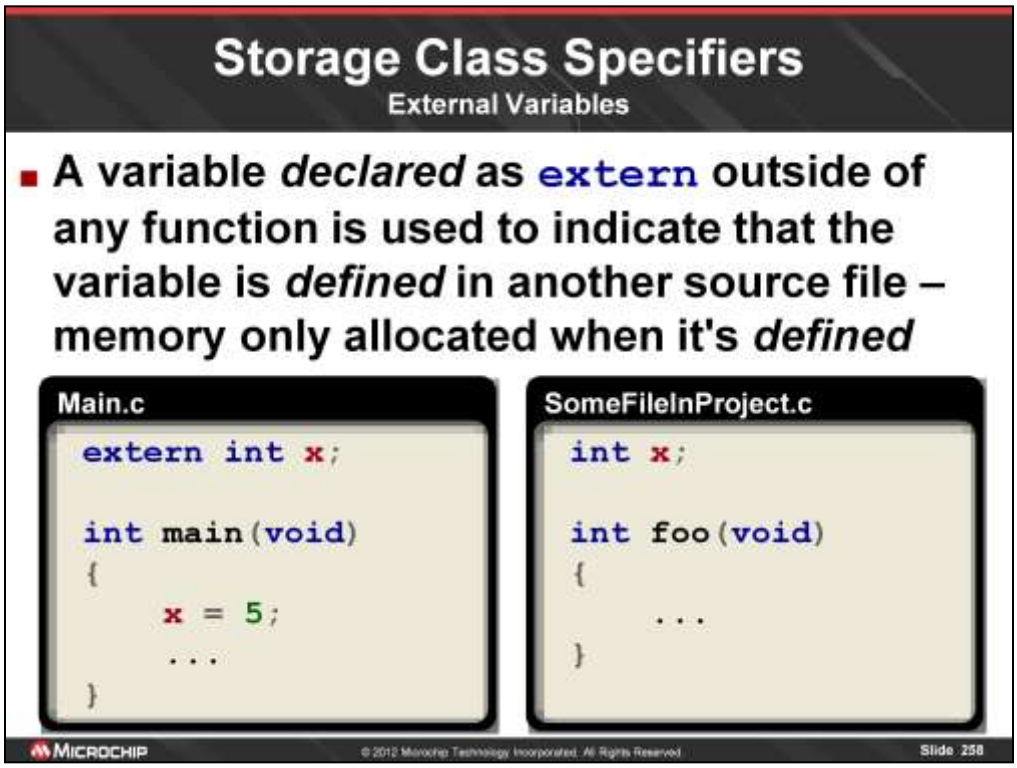
The **extern** keyword is used to declare a variable that is defined elsewhere. As with function prototypes, an **extern** variable declaration doesn't allocate memory. It merely identifies the signature of the variable so that the compiler knows how it should be handled when it is encountered in your code.



•**A variable *declared* as extern within a function…**

This is not the most common use of **extern**, but it does work and can be useful in some circumstances. When a local variable is declared as extern, that means that no space is allocated for the variable from within the function. It simply tells the compiler/linker that the variable is defined elsewhere – the memory will be allocated somewhere else. This is much like when we declare a function prototype so that the function may be called before it is defined.

The definition of the variable may be after the function in which it is used, or it may be in another file altogether. One possible way this might be useful is to create a global variable in a separate source file. Then if you declare it inside multiple functions as **extern**, it will only be accessible from within those functions unless it is declared as **extern** at the top of the source file where the functions are called. This can provide a rudimentary form of something known as data hiding, which is a fundamental rule of good programming practice in object oriented languages such as C++, C#, VisualBasic.net and Delphi (object oriented Pascal).

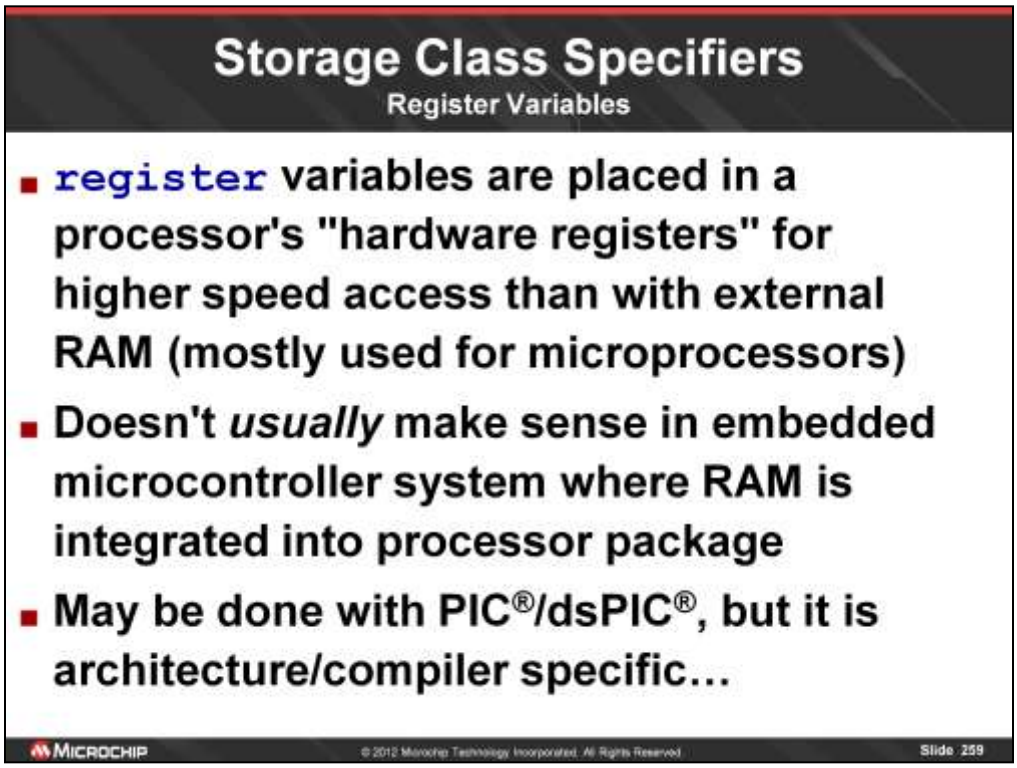


•A variable declared as extern outside of any function…

If we declare a variable as **extern** outside of any function, this means that the variable is defined in another file altogether (NOTE: This may also be true for a variable declared as **extern** within a function, but in that case, the variable may just be defined later on within the same file.) No memory is allocated where the variable is declared as **extern**. This is simply a way to tell the compiler that the variable named will be used in this file, but it will be defined, and memory will be allocated for it in another file.

In the example shown, the variable **x** is defined in the file OtherFileInProject.c. It is a global variable, and therefore is visible from anywhere in the project (not just within its own file). However, to use it in Main.c, it is necessary to declare it as **extern** so that the compiler knows that we are referring to the variable **x** that is declared in OtherFileInProject.c. Note that this is a good example of why unique variable names are good programming practice (unlike the simple single letter names we've been using). When we declare **x** as extern in Main.c, the variable a must only be defined globally in one other file in the project, otherwise the compiler won't know which variable **x** we are referring to.

Once again, this is analogous to having a function prototype. If we want to call a function in another file, we need to declare its prototype in the file that will be calling the function. This will be discussed in more detail in the section on multi-file projects.



•Variables placed in a processor's "hardware registers"…

On PCs and other microprocessor based systems where the processor and RAM are completely separate devices, the register keyword may be used to force a variable to be located in the hardware registers on board the microprocessor. This will provide much faster access to a commonly used variable since the data doesn't have to be accessed from an external chip.

•Doesn't usually make sense in embedded microcontroller system…

On a microcontroller where the CPU and memory are integrated in the same package, the access times associated with retrieving data from RAM are generally very fast, so declaring a register variable doesn't often make sense. This is especially true for the PIC18 which has only one working register. The 16-bit products with their 16 working registers can accommodate a register variable much more easily, but they are still somewhat limited with respect to a high-end microprocessor.

•May be done with PIC/dsPIC…

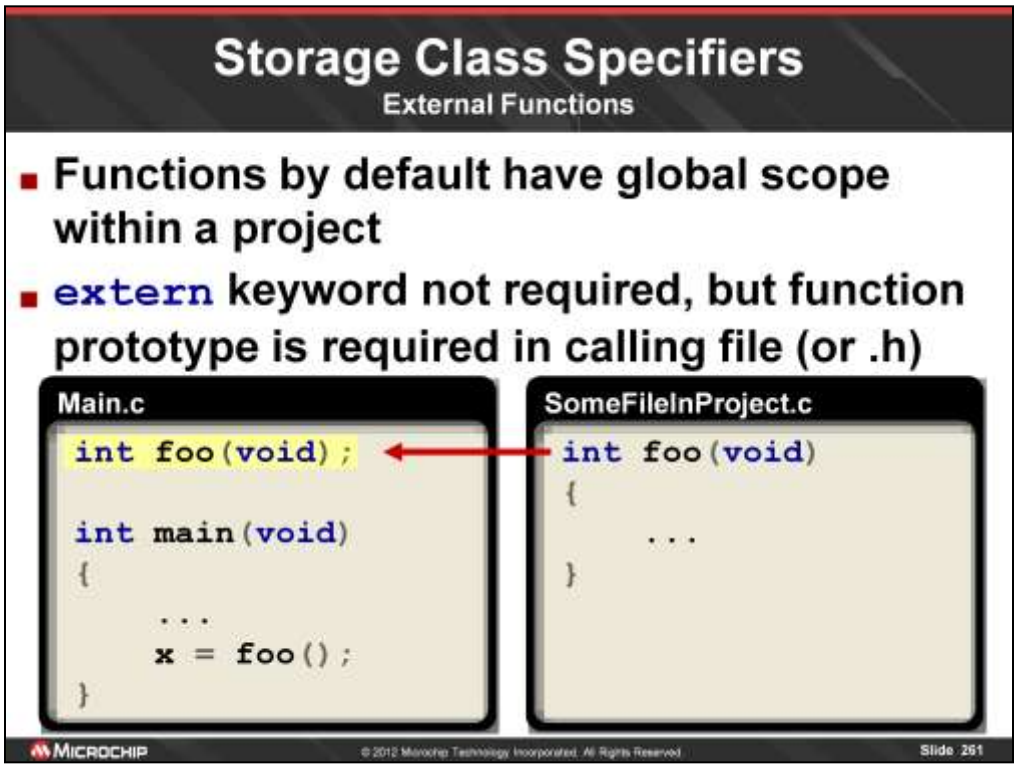
Register variables may be declared with our microcontrollers, but the syntax used and how it is actually handled are very dependent on the underlying architecture. Therefore this will be covered in part 2 of this class which covers the platform specific implementation details.

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)

© 2012 Microchip Technology Incorporated. All Rights Reserved.



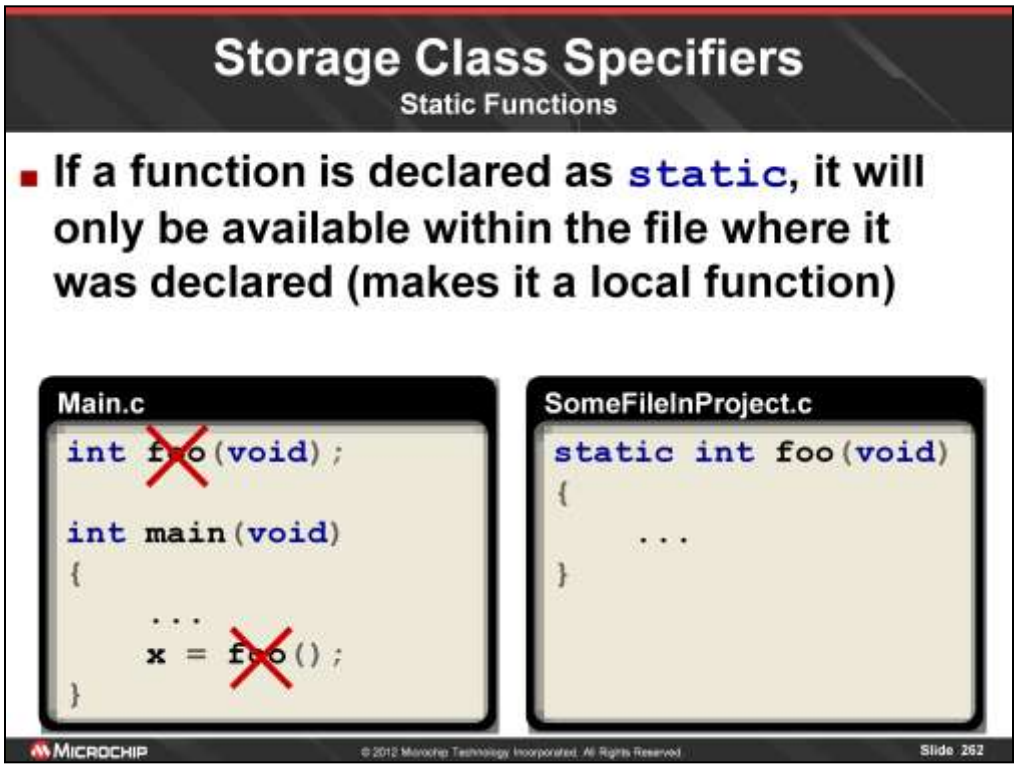
•Functions by default have global scope within a project

A function defined within any file in a project may be called from any other file.

•extern keyword not required, but function prototype is required in calling file or #included header file

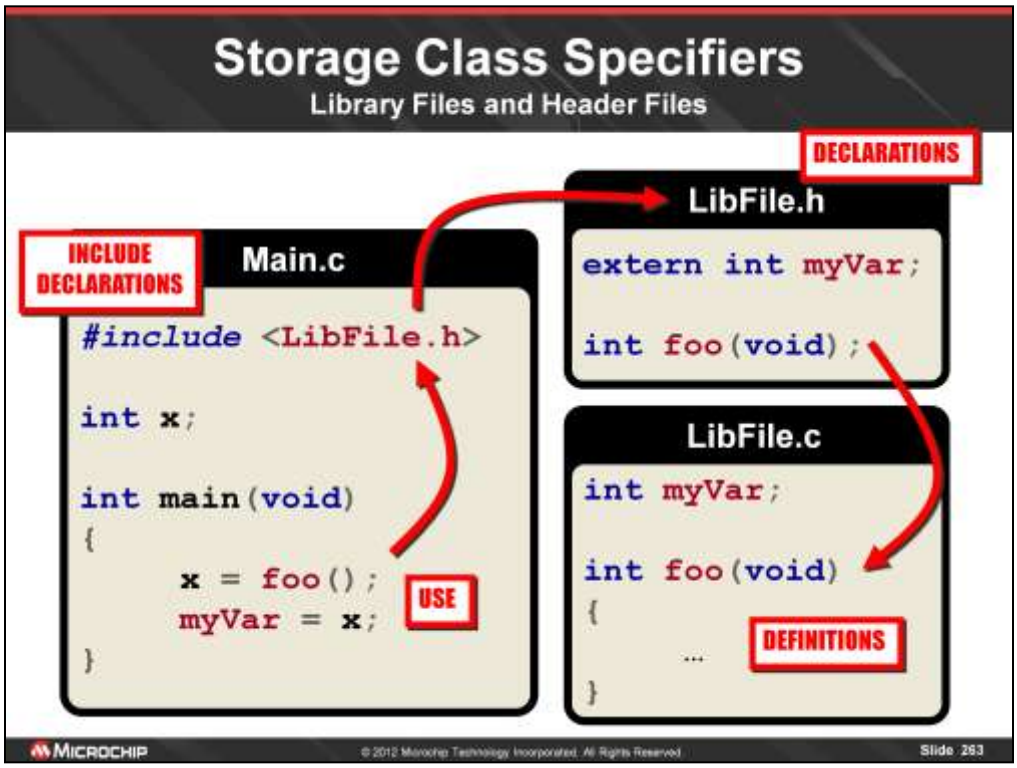
The only thing required to access the externally defined function is a function prototype in the calling file or a #included header file. The use of header files for this purpose is extremely common. The functions available in the standard C library all have associated header files that may be included in your source files. These header files contain the function prototypes for the library functions, which let the compiler know how to handle these function calls even though the definition of the function is somewhere else.

Note: it is perfectly valid to use the **extern** keyword in front of a function definition, but it isn't necessary since a function is **extern** by default.



•If a function is declared as static…

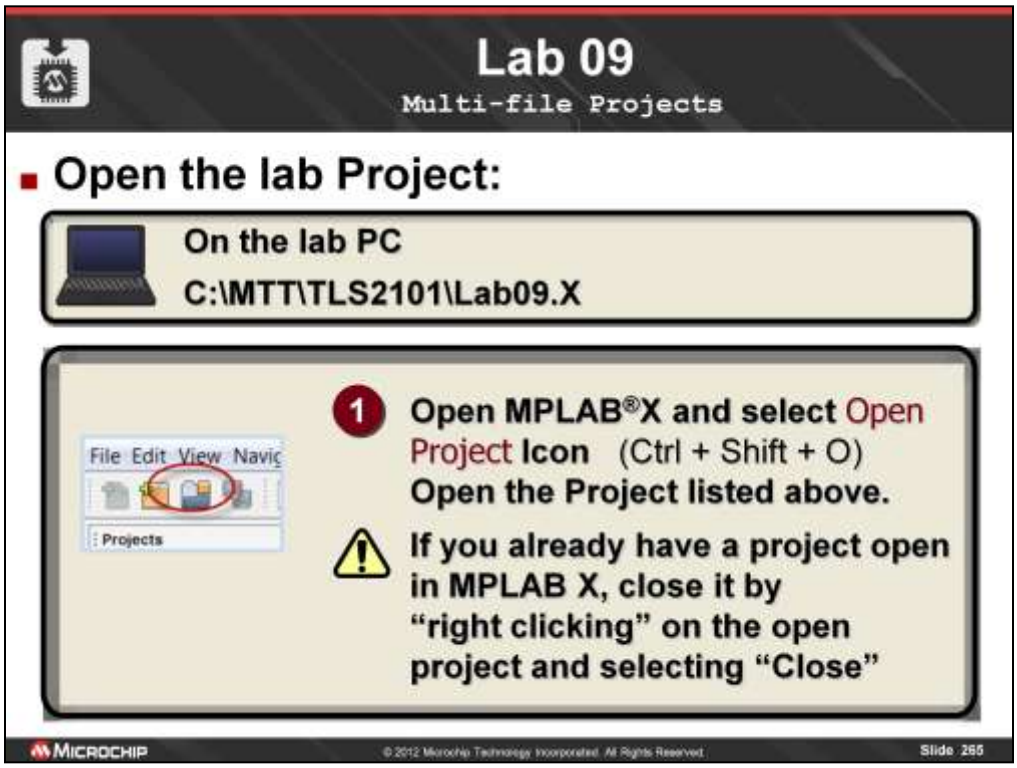
Static functions are essentially local to the file in which they are defined. They are not globally visible at all. Even if you include a function prototype in a file, it will not be able to access the static function. Only functions within the static functions file may call it.



In this example, LibFile.c could represent any library file, whether it is included with the compiler (such as ADC.c, USART.c, etc.) or one of your own making. The header file, LibFile.h, contains the extern variable declarations and function prototypes that will be required by any program that wants to access the variables and functions contained in LibFile.c. So, is we #include LibFile.h into our program Main.c, and we add the file LibFile.c to our project, Main.C will have access to any variable or function in LibFile.c that is declared in the header file LibFile.h.

Note that LibFile.c could be replaced with a pre-compiled object file such as LibFile.o, or a library (archive) file that may contain many related precompiled source files such as LibFile.lib (C18 and most non-GNU compilers) or LibFile.a (C30 and all GNU based compilers).

MICROCHIP
Lab Exercise 9
Multi-File Projects



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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);
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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 );
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

<end of part two>

© 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
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**

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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'};
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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}}};
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Arrays

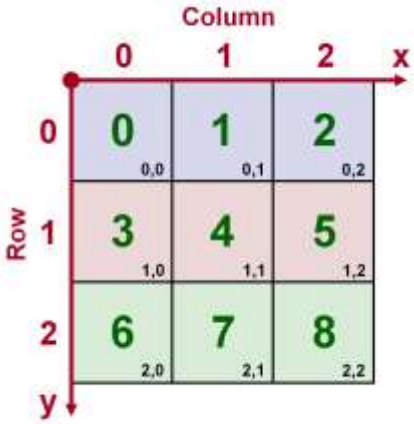
Visualizing 2-Dimensional Arrays

```
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;



© 2012 Microchip Technology Incorporated. All Rights Reserved.

Arrays

Visualizing 3-Dimensional Arrays

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

Plane, Row, Column

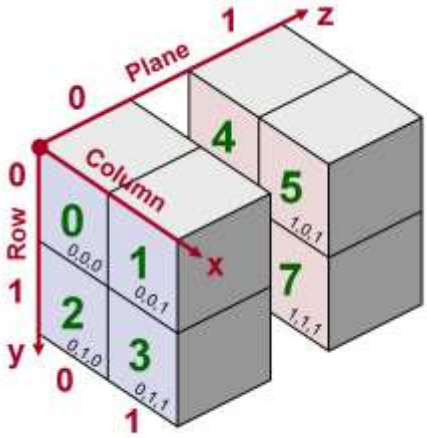
```
a[z][y][x]
```

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;
```



© 2012 Microchip Technology Incorporated. All Rights Reserved.

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);
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 278

MAIN IDEA

- Arrays are frequently processed as part of a loop since the same operation needs to be performed on each element
- Operations might include
  - Sending strings of characters to or reading strings of characters from a UART
  - Performing a mathematical transform on an array of floating point numbers (think digital filters)
- The loop count variable will often be used as the array index itself or in conjunction with the array index

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**

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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'
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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'};
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

Strings

How to Initialize a String in Code

In code, strings must be initialized element by element:

Syntax

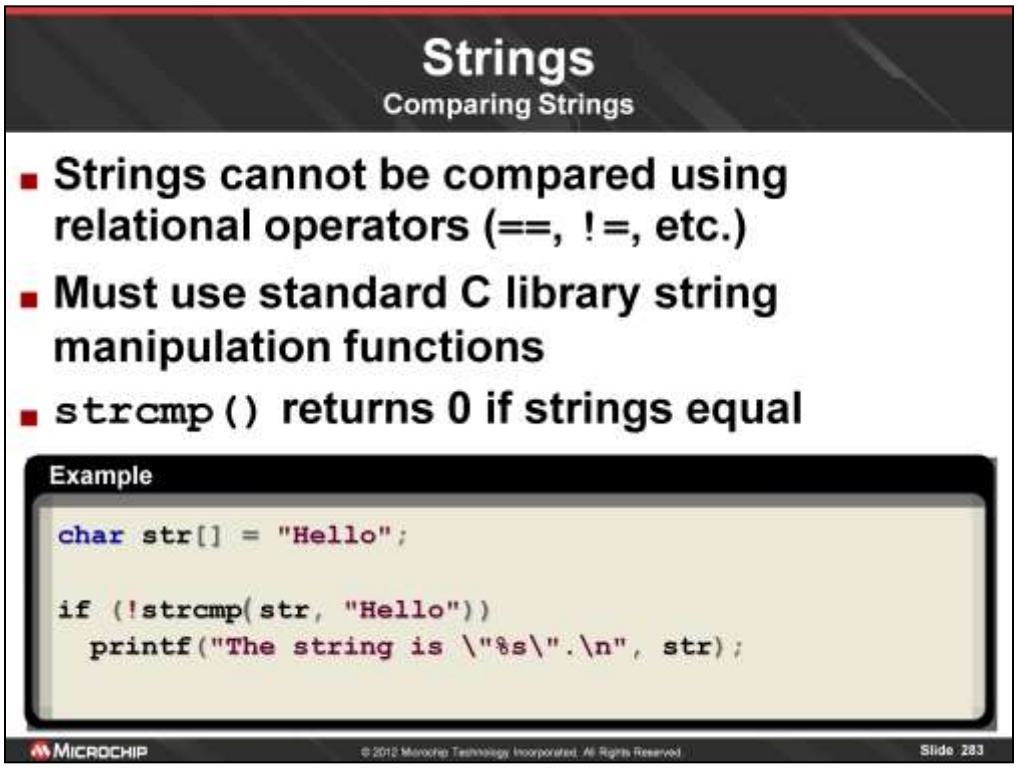
```
arrayName[0] = char1;
arrayName[1] = char2;
      ⋮
arrayName[n] = '\0';
```

- 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';
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.



MAIN IDEA

- Strings may not be compared as ordinary variables: We cannot do **if(str1 == str2)** the way we do **if (x == y)**.
- To compare two strings, it is necessary to compare them character by character.
- C provides string comparison/manipulation functions as part of the standard C library.
- When using **strcmp()** to compare two strings, it returns 0 (FALSE) when they match – so its logic must be inverted when used as a conditional expression

CODE EXAMPLE

1. Create a character array and initialize it with the string literal "Hello".
2. Test to see if the string contained in the character array **str** matches the string literal "Hello". Note that **strcmp()** may compare two array variables or compare with a string literal as we did here. e.g. **strcmp(str1, str2)** or **strcmp(str1, "This string")**. The **strcmp()** function will return 0 if the strings match, it will return < 0 if the first string is less than the second and it will return > 0 if the first string is greater than the second string.
3. If the strings match, print out the message including the value of **str**.

Due to the time constraints we are under, we will not cover all the string manipulation routines. Their functionality is detailed in just about every book on the C programming language available today. The key thing is to be aware of their existence so that you know to look for them when working with strings.

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.

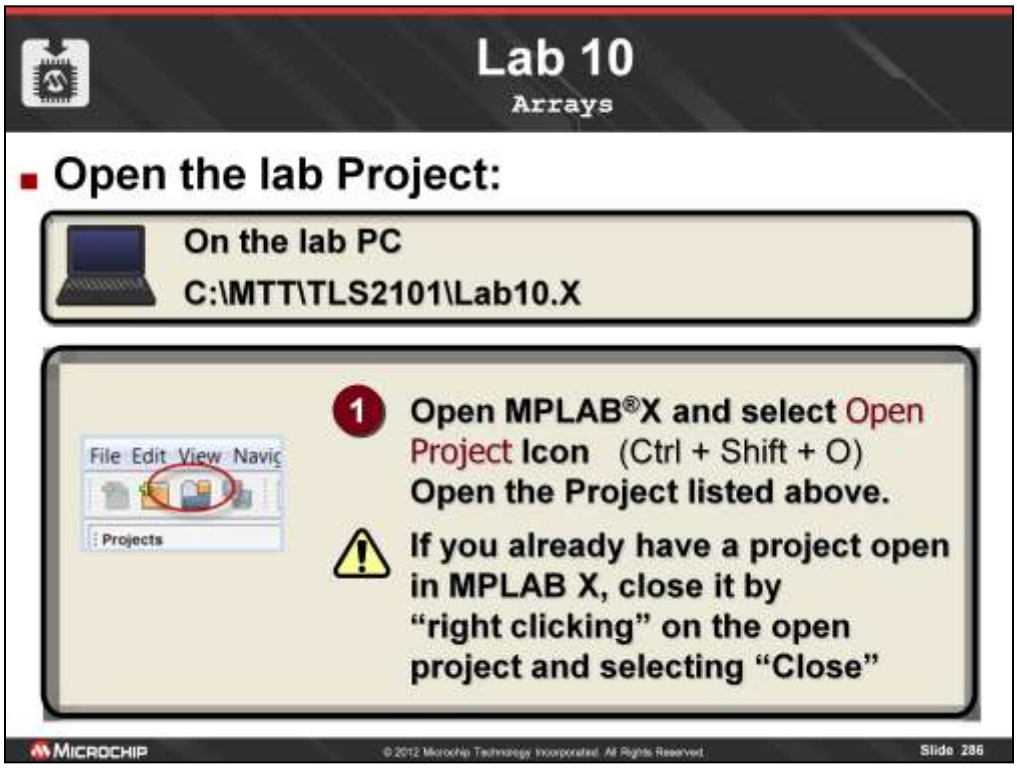
```
void WriteLCD(char *greetings){…}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 284

xxx

An advantage of using the array notation is that a human reader will immediately recognize that we are dealing with an array (character string) rather than a pointer to a single character variable.

MICROCHIP
Lab Exercise 10
Arrays



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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};
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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++;
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

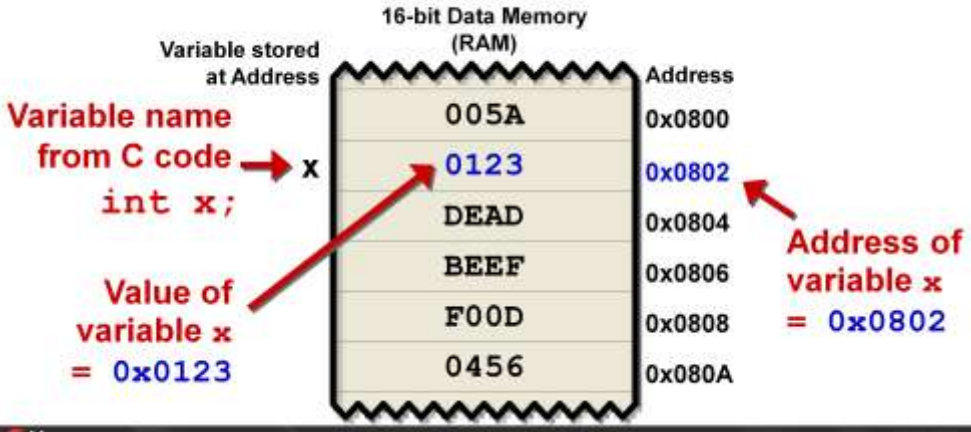
MICROCHIP

Data Pointers

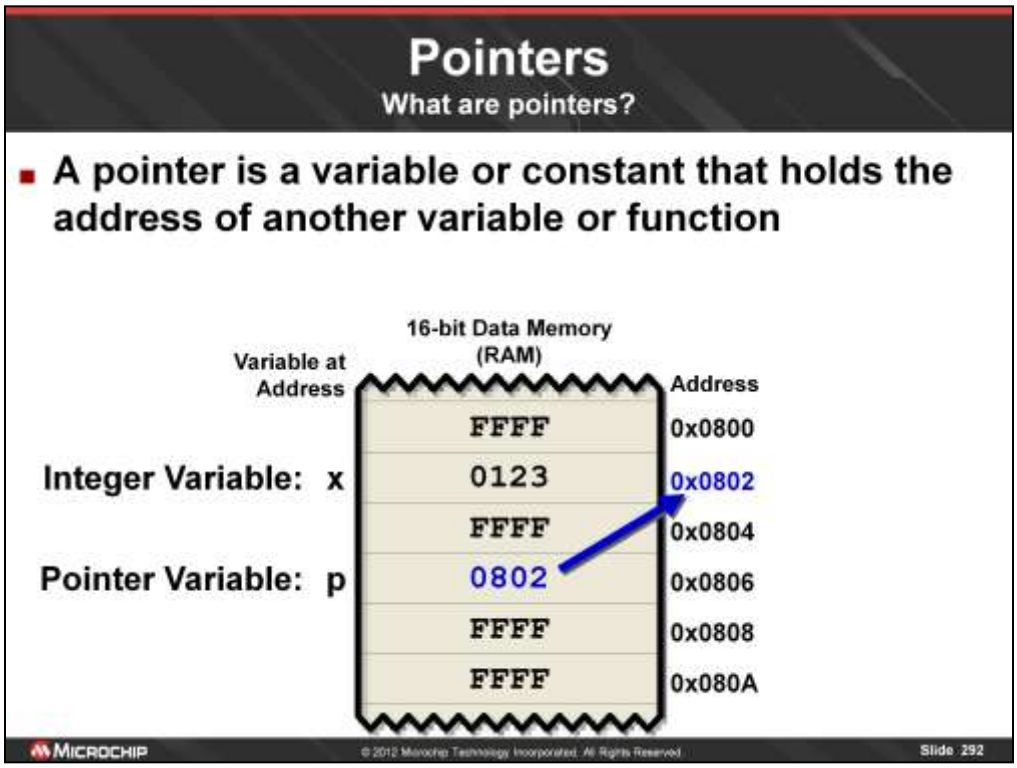
Pointers

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…



© 2012 Microchip Technology Incorporated. All Rights Reserved.

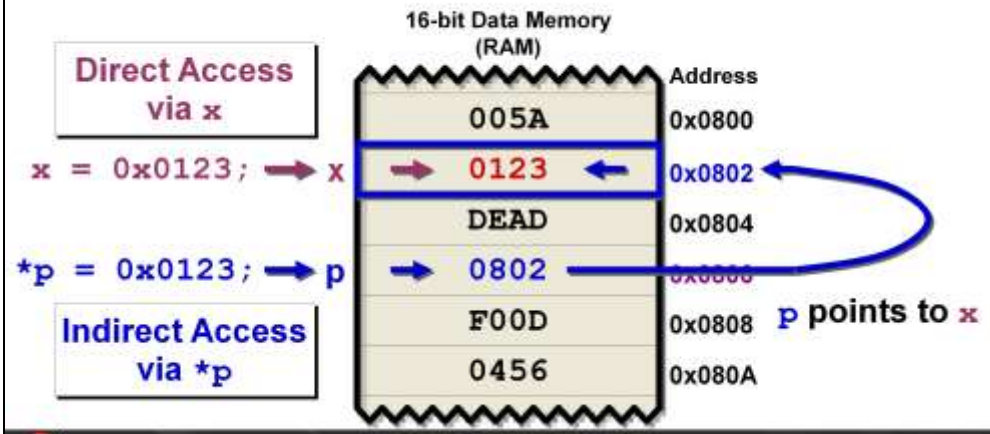


A pointer is essentially a variable that holds the address of another variable or function (we will discuss function pointers later). Pointers may be used to perform what is known in the assembly language world as indirect addressing. When you use pointers, you are able to access the variable that they point to, and you are able to change the pointer so that it points to a different variable. This simple mechanism has hundreds of uses and is one of the most powerful features of the C programming language.

Pointers

What do they do?

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



© 2012 Microchip Technology Incorporated. All Rights Reserved.

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


© 2012 Microchip Technology Incorporated. All Rights Reserved.

Pointers

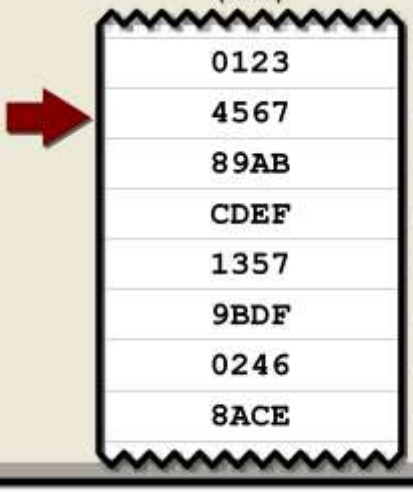
Why would I want to do that?

Example: Data Buffer

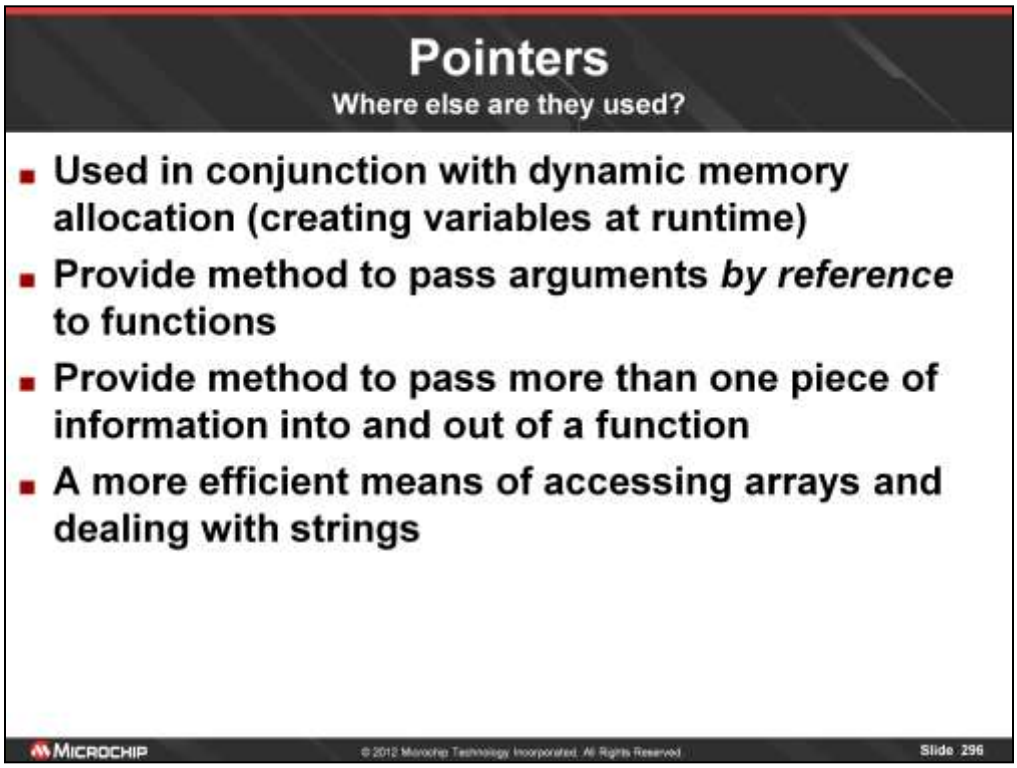
RAM buffer allocated over a range of addresses (perhaps an array)

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)



MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.



Pointers may be used in many ways. Some of the more common include being used in conjunction with dynamic memory allocation. A pointer may point to a location in the heap where a variable is created at runtime since the variable doesn't have a name in the same way as variables we explicitly declare before compilation.

More frequently, you will see pointers used as a mechanism to pass multiple parameters to or from a function, or as a way to pass a value BY REFERENCE to a function. Since C ordinarily passes variables to functions BY VALUE, the original variable is never changed. In some cases we want to modify the original variable as a part of the function's operation. Therefore, we can pass a pointer to the function, and it will then be able to modify the original variable (much more on this later). Pointers also provide an alternate means of accessing arrays, which can be more efficient, especially when 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
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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;`)

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 299

To initialize a pointer, we need to set it equal to the address of another variable. We can achieve this with the & (address of) operator. Because we are setting the value of the pointer itself, we do not use the * operator here. We simply use the pointer's name, followed by an equals sign and the name of the variable whose address we want prefixed by the & operator. In this example, we are setting the pointer p to the address of x, so now p points to x. Of course this assumes that both p and x are declared as the same type (e.g. int). For the assembly language programmers in the room, this is similar to loading a register's address into the FSR register. p is like the FSR register itself, and x is like the name of a general purpose register.

Pointers
Usage

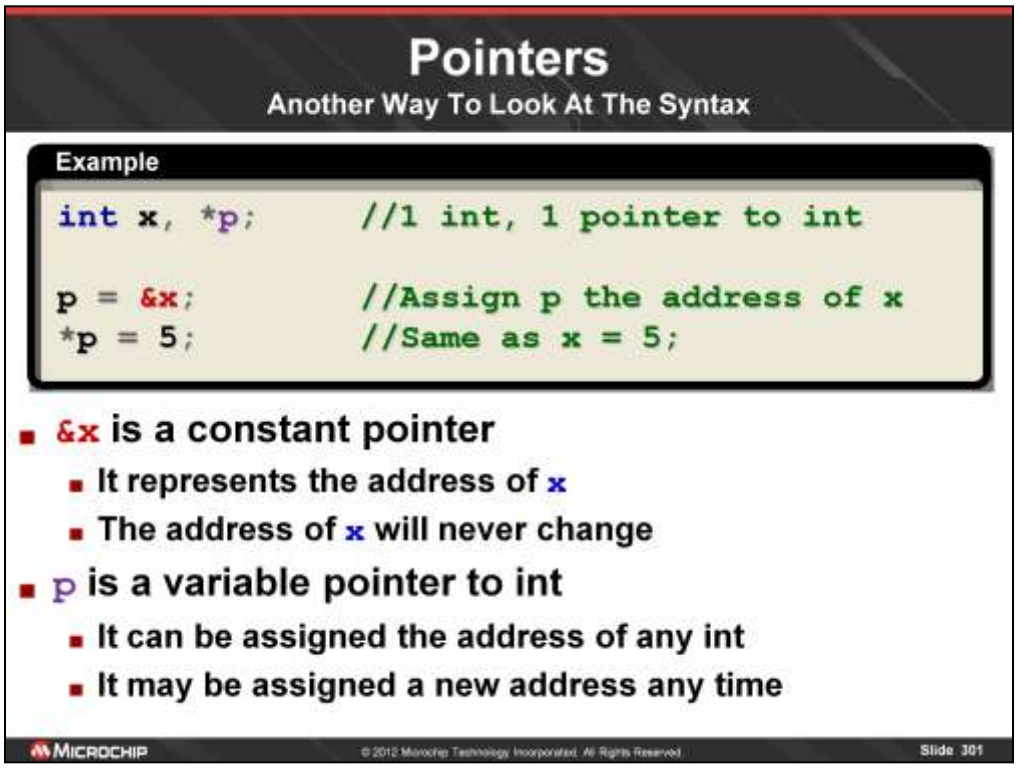
- When accessing the variable pointed to by a pointer, we use the pointer <u>with</u> the dereference operator `*`:

```
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`)

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 300

Now that the pointer has been initialized , it may be used as if it were the variable it points to. To do this though, we need to prefix the pointer variable with the * (dereference) operator. Unlike in the pointer declaration, here the * means "use the value of the variable that p points to – not the value of p itself". Since we set p to point to x on the previous slide, *p means the same thing as x in our code.



Another way to look at the syntax that might help it make more sense is to consider the concept of a constant pointer and a variable pointer.

First, understand that the terms ***address*** and ***pointer*** are synonymous. An address **<u>is</u>** a pointer.

So, in the code above, **&x** is a pointer to the variable **x** because **&** means "address of". Since **x** will always be at the same address (once it is assigned by the linker/compiler – it never moves), **&x** may be thought of as a constant pointer. Moreover, the value of **&x** is known at link time, so no computation need be done by the microcontroller. In every respect, this is a constant that happens to be a pointer (address).

The real power of pointers comes from the ability to have a variable pointer, where the address it points to may be changed any time. In the code above, **p** is a variable pointer. On the second line, we assign **p**, the constant value **&x** so that **p** now points to **x**. We can modify the address that **p** points to in this fashion anywhere in our code (or we can perform pointer arithmetic which will be discussed shortly).

Pointers

Another Way To Look At The Syntax

Example

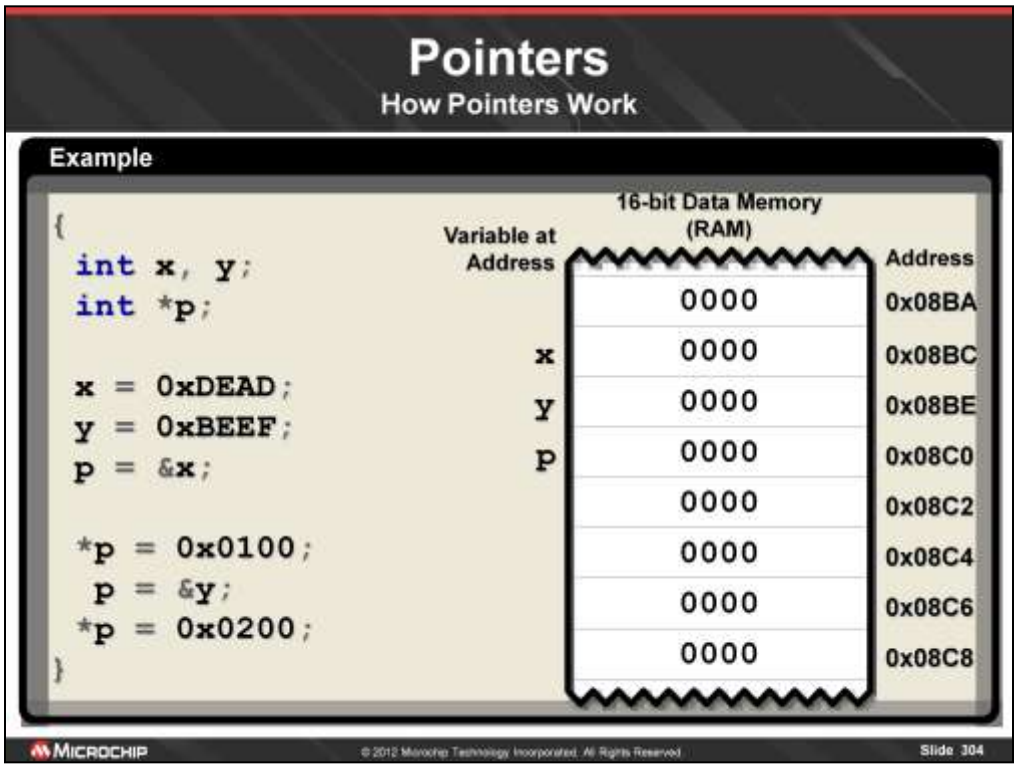
```
int x, *p;      //1 int, 1 pointer to int

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 302

After we have declared the pointer and given it an address to point to, we can now use it to access the data that resides at that address. To do this, we use the dereference operator *****. Since **p** was assigned the address of **x**, anywhere we use ***p**, it is the same as if we used **x** directly. Therefore, on the last line, **y = *p** does the exact same thing as **y = x**.

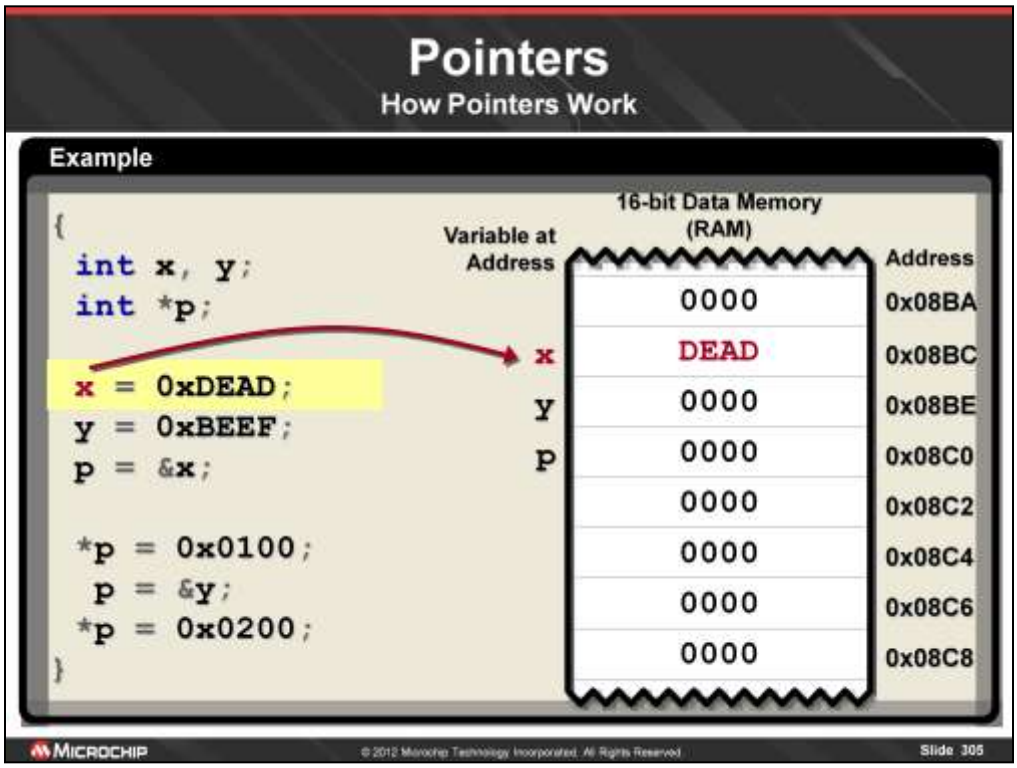
Note that in the context of the pointer's declaration **int *p**, the ***** operator is only used to indicate that the variable **p** is a pointer to int rather than an ordinary variable of type int. It does not indicate indirection/dereference as it does when used in code.



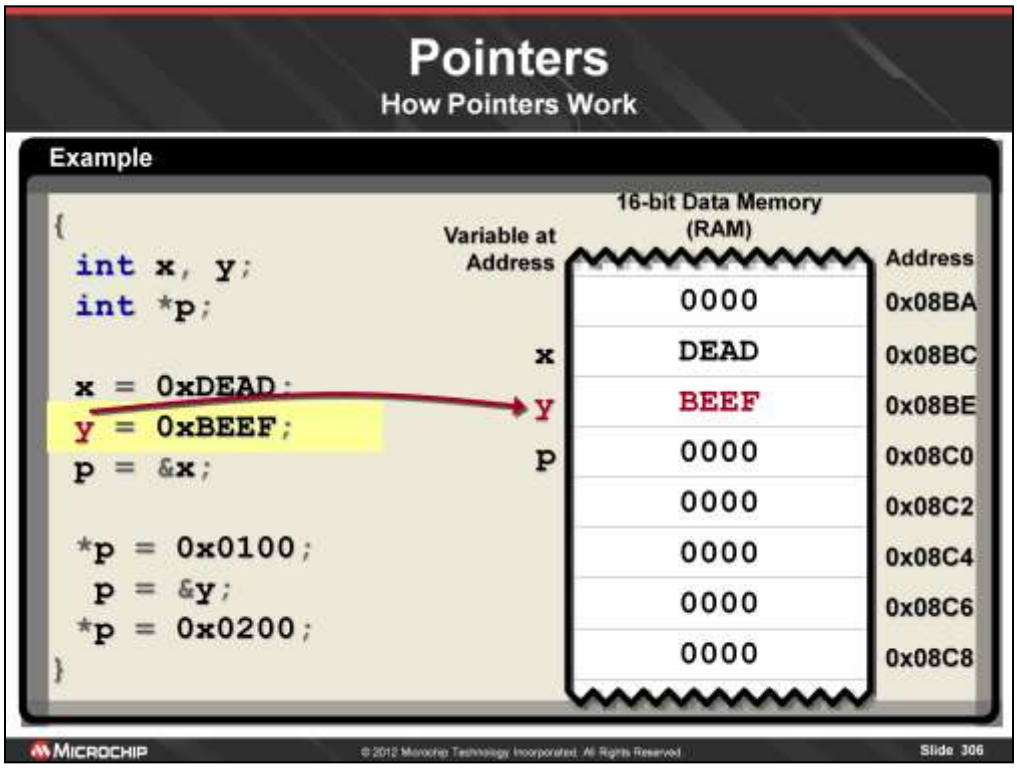
NOTE TO PRESENTER: *This example isn't exactly a practical use of pointers, but it does clearly illustrate the various syntactical elements of pointers. Because some students have trouble accepting impractical examples, please emphasize up front that this example is only to illustrate the pointer mechanism and the syntax used to accomplish particular ends when using pointers. These are just the building blocks used to take advantage of pointers in more sophisticated, real world applications. Without a firm understanding of what happens when we use a particular syntax, more complicated examples will be very difficult to understand.*

Let's take a look at a very simple example of how pointers work. In this program fragment, we have declared two ordinary integer variables, and one pointer to an integer.

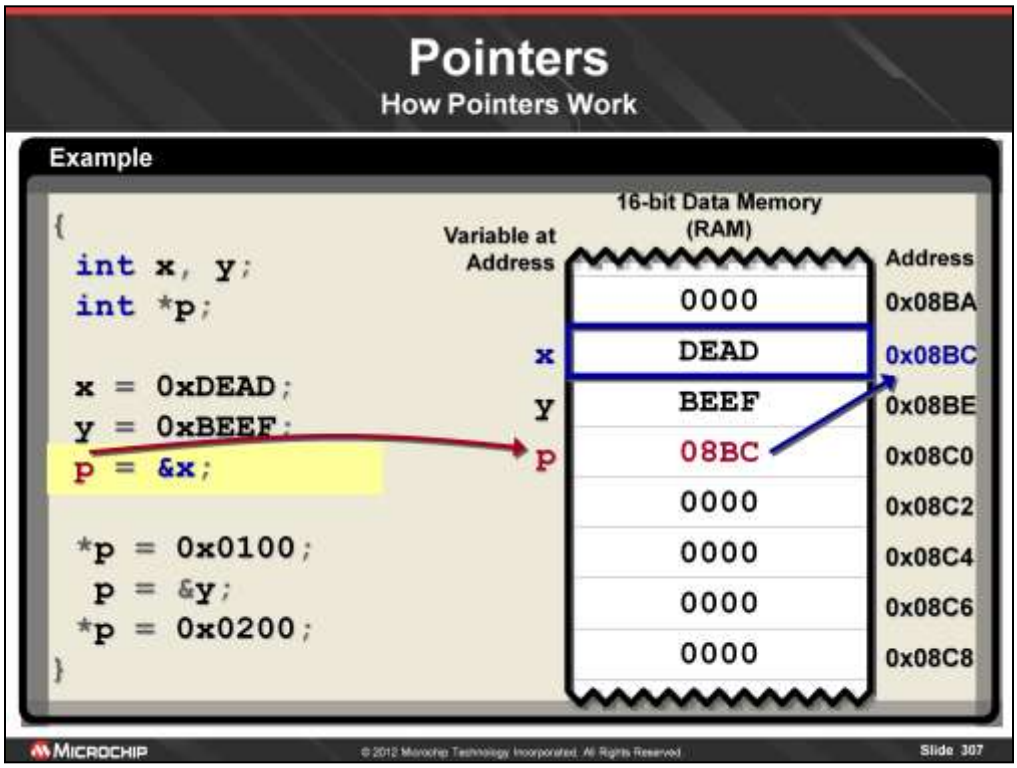
I would like to highlight one important thing about C syntax here. In the pointer declaration, the * operator here is used only to indicate that the variable p is a pointer to an int, not an int itself. Anywhere else in code, this notation will mean something else, as we will see in a moment.



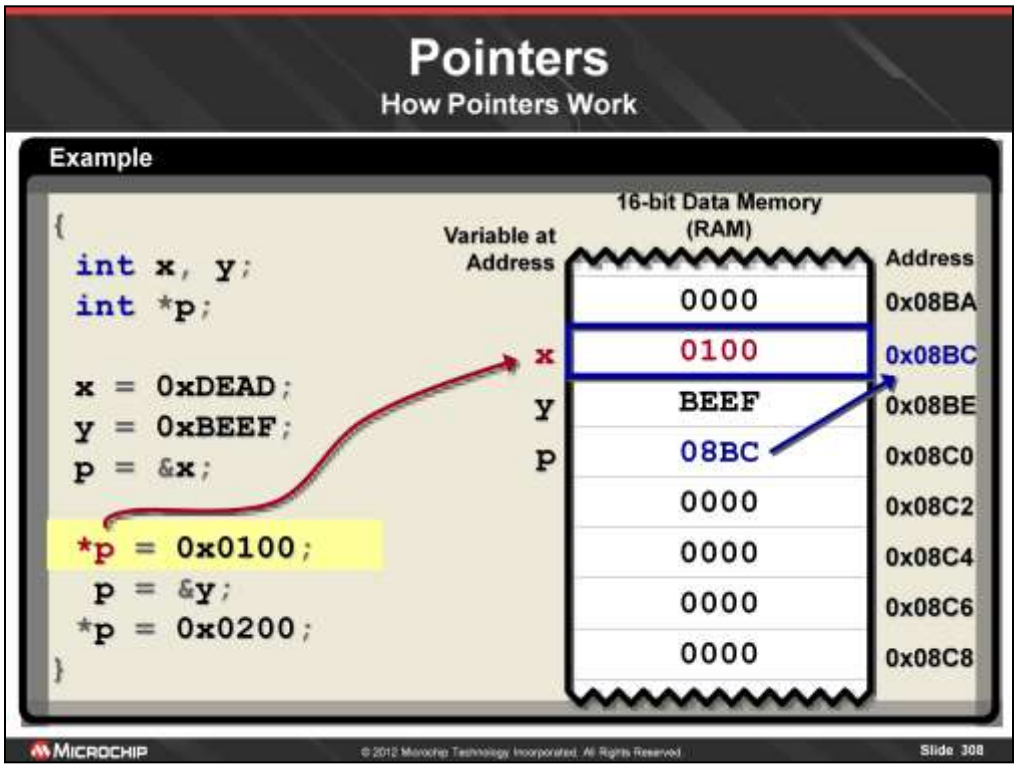
The first line of code assigns the hex value 0xDEAD to the variable x, as we have seen in previous examples. When this code is compiled, we can look at the map file to see where x was located by the linker. Every time you compile a program, a map file will be generated by the linker to document where it placed the various elements of your program. When I compiled this example on my system, I found that x was placed at 0x08BC.



Next, we assign the hex value 0xBEEF to y. Using the map file as I did with x, I found that y was located at address 0x08BE.

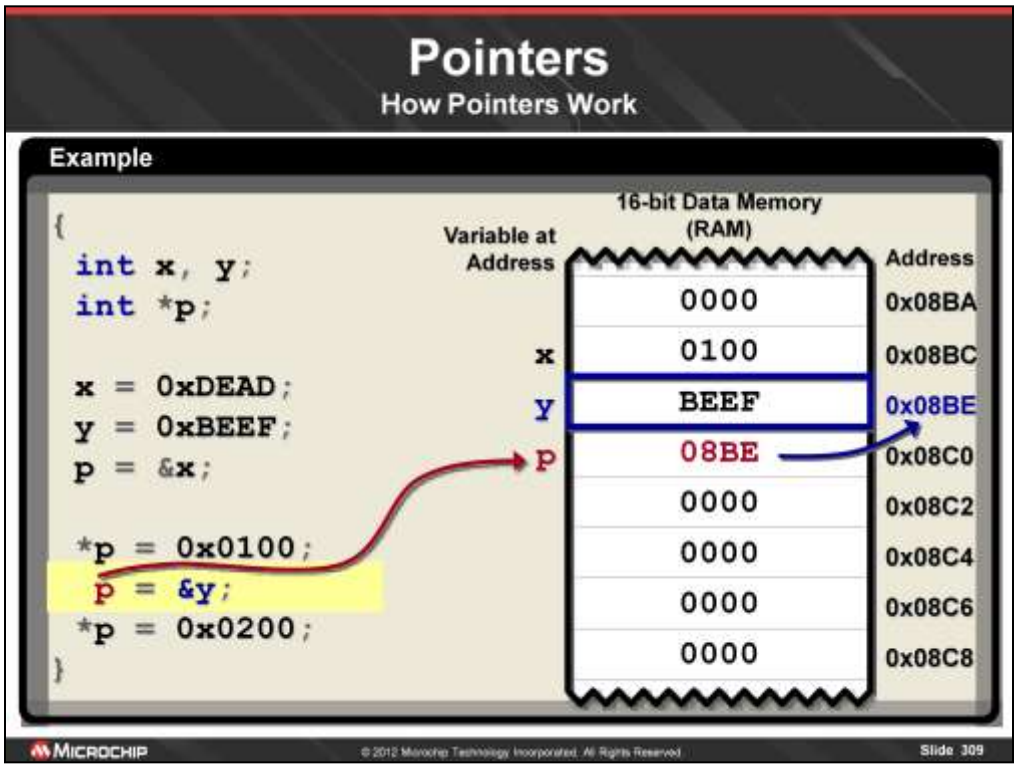


And now, we start working with the pointer variable p. On this line, we assign the address of x to p by using the unary & (address of) operator in front of x. Note that the value stored in p is 0x08BC – this is the address of x (that we can verify with the map file). Now we can say that p points to x.

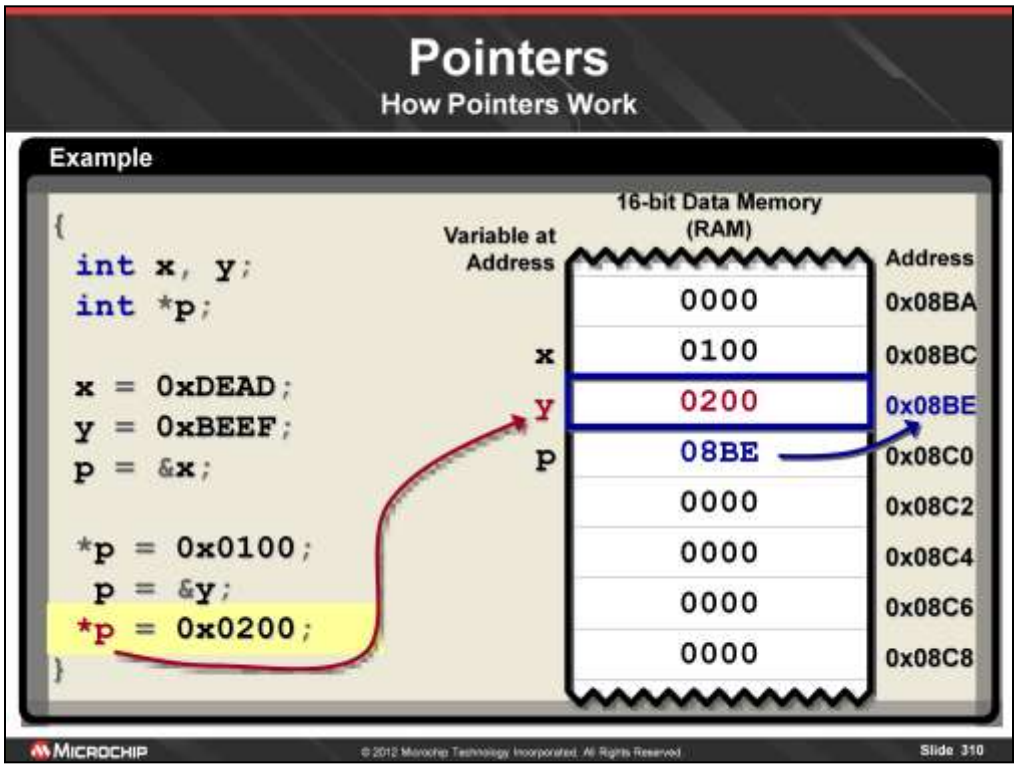


Now that we have initialized the pointer variable p, we can use it to access the data it points to. This line of code basically says, assign the value 0x0100 to the variable pointed to by p. Also note that now, the * in front of p indicates that we want to use the location that p points to – not the location of p itself. Although the syntax of *p is the same as what we saw in its declaration, it has a different meaning: use the variable that is pointed to by p.

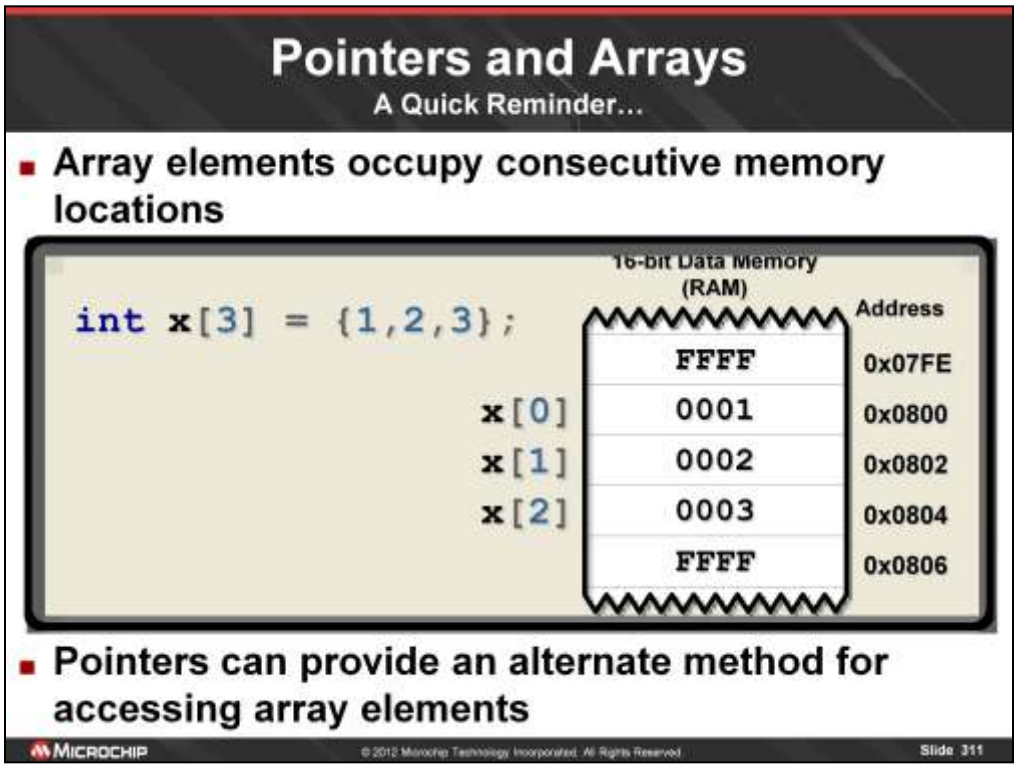
The net result of this operation is that the value of x is changed to 0x100. Since p contains the address of x, using *p is the same as using x.



On this line, we change the value of the pointer p to contain the address of y. So now, p points to y.



Finally, we write the value 0x0200 to the variable pointed to by p, which is y since that is what we just set it to. The net effect is that the value of y was changed to 0x0200. After changing p in the previous line, any time we use *p now, will be the same as if we used y.



Let's take a quick look at arrays again. Recall that array elements occupy consecutive memory locations. Because of this, if you know the address of the first element of an array, you can easily move through the array by incrementing the address or adding an offset to that address. While we could do this with the ordinary array syntax by incrementing or adding a value to the array offset, we could also do this with pointers.

Pointers and Arrays

Initializing a Pointer to an Array

- The array name is the same thing as the address of its first (0th) 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
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 312

Creating a pointer to an array is just like creating a pointer to any other variable. However, because of the unique nature of arrays, there are additional ways you can initialize pointers to arrays.

The first method is to simply assign the array variable to the pointer variable. By definition, an array variable without the square brackets and index represents the address of the array, which is also the address of the first element. So, we can simply say p = x; without the & (address of) operator. Personally, I don't like this syntax because of the confusion it might cause. Despite the aforementioned rule, it just doesn't look like an address is being assigned due to the missing & operator.

The method I prefer to use is the second one, where we use the same syntax as when we assign an ordinary variable's address to a pointer. This syntax is much easier to remember since it works for both variables or arrays.

The final syntax is virtually never used, even though it is perfectly correct to use and is the only syntax that makes it perfectly obvious what is going on. But it requires more typing, and C programmers hate to type more than necessary…

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…

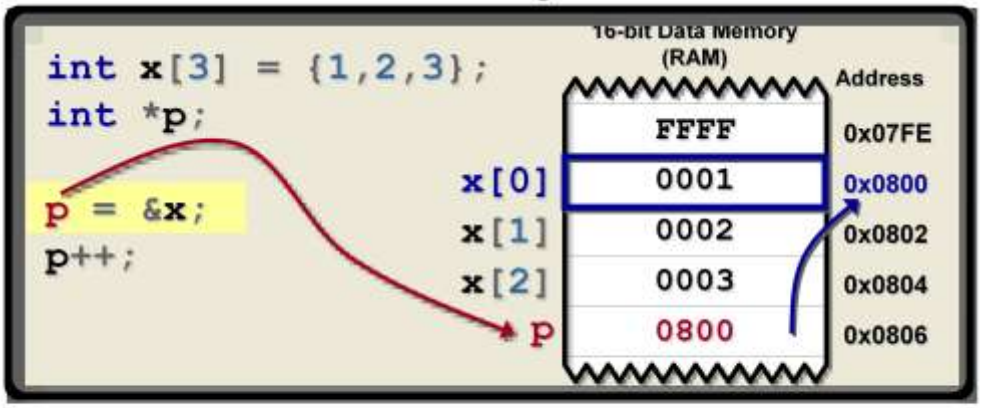
MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Here is an example of an array and a pointer being used to identify a particular element. One of the most powerful features of using pointers with arrays is that incrementing a pointer always moves it to the next element in the array, regardless of the number of bytes that each element occupies. We will see much more about this in just a moment…

Pointers and Arrays

A Preview of Pointer Arithmetic

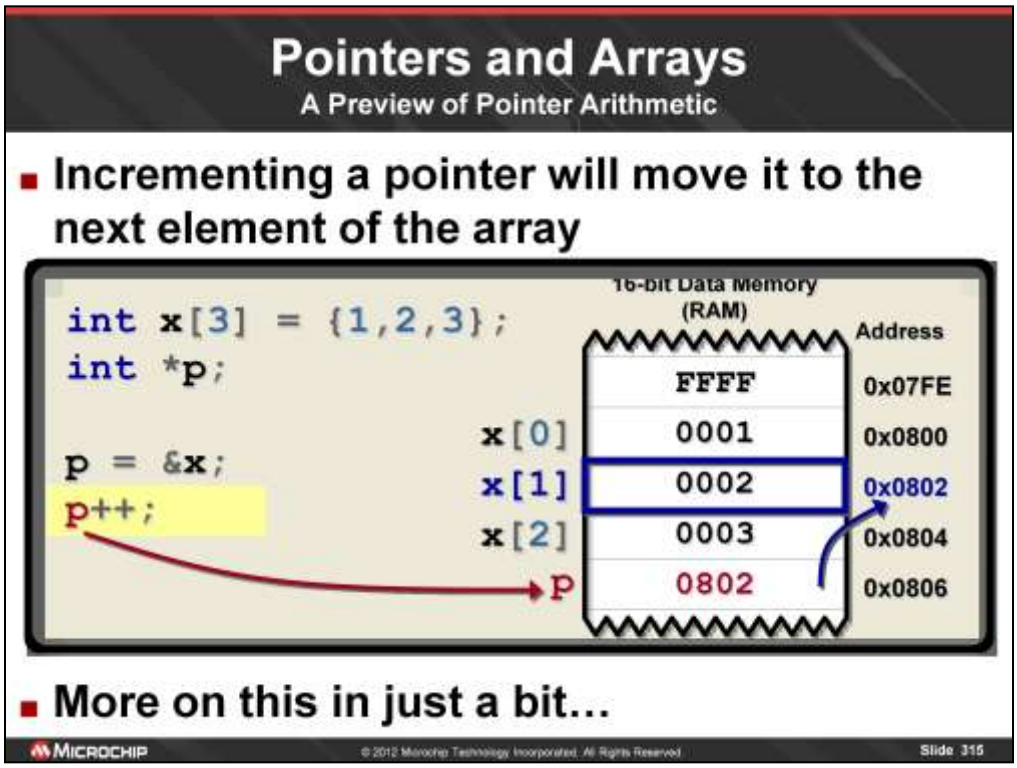
- Incrementing a pointer will move it to the next element of the array



- More on this in just a bit…

© 2012 Microchip Technology Incorporated. All Rights Reserved.

First, we set the pointer to the address of the first element (which is the address of the array iteslf)



Next, we increment the pointer so that it now points to the next element in the array.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 316

When you increment or decrement a pointer, it will always increment by the number of bytes occupied by the type it points to.

For example, if we have a pointer to float, incrementing the pointer will increment the address it contains by 4, since float variables occupy 4 bytes of memory. The compiler does this automatically for you, eliminating the need to keep track of variable sizes yourself in these situations.

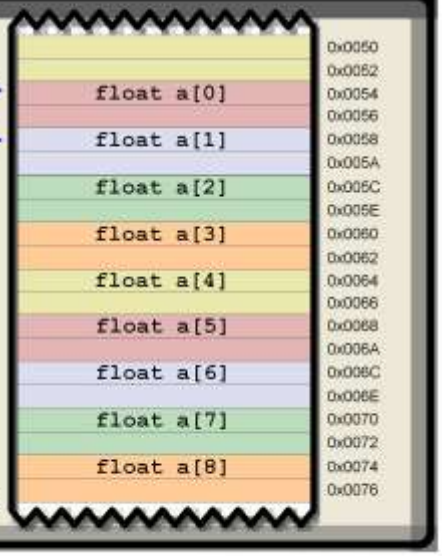
Pointer Arithmetic

Incrementing Pointers

Example

```
float *ptr;
ptr = &a;
ptr++;
```

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



16-bit Data Memory Words

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 318

Similarly, if we use the compound assignment operator to add or subtract a larger number for a longer jump, the pointer will be modified by a multiple of the number of bytes occupied by the type it points to.

For example, if we have a pointer to int, adding 3 to the pointer will add 3*2 = 6 to the address it contains. This is because int variables occupy 2 bytes.

Similarly, if we had a pointer to float, adding 3 to the pointer will add 3*4 = 12 to the address it contains. This is because float variables occupy 4 bytes.

Pointer Arithmetic

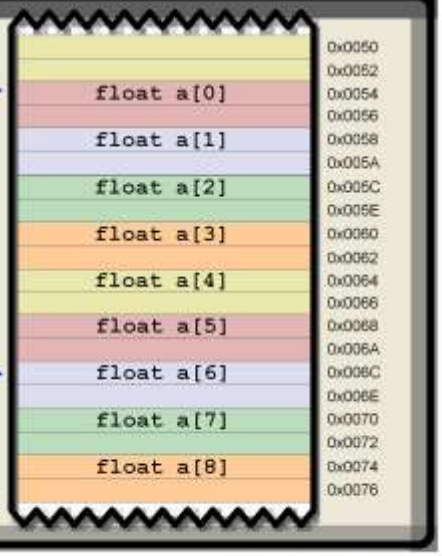
Larger Jumps

Example

```
float *ptr;
ptr = &a;
```

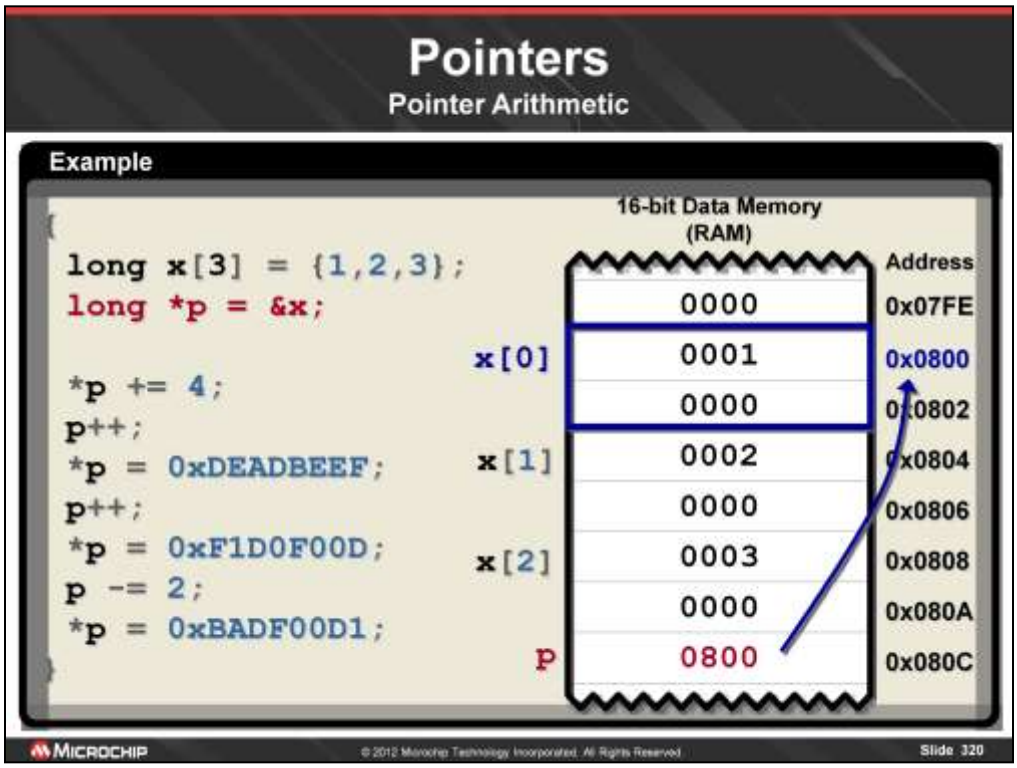
Adding 6 to `ptr` moves it 6 `float` array elements ahead (24 bytes ahead)

```
ptr += 6;
```



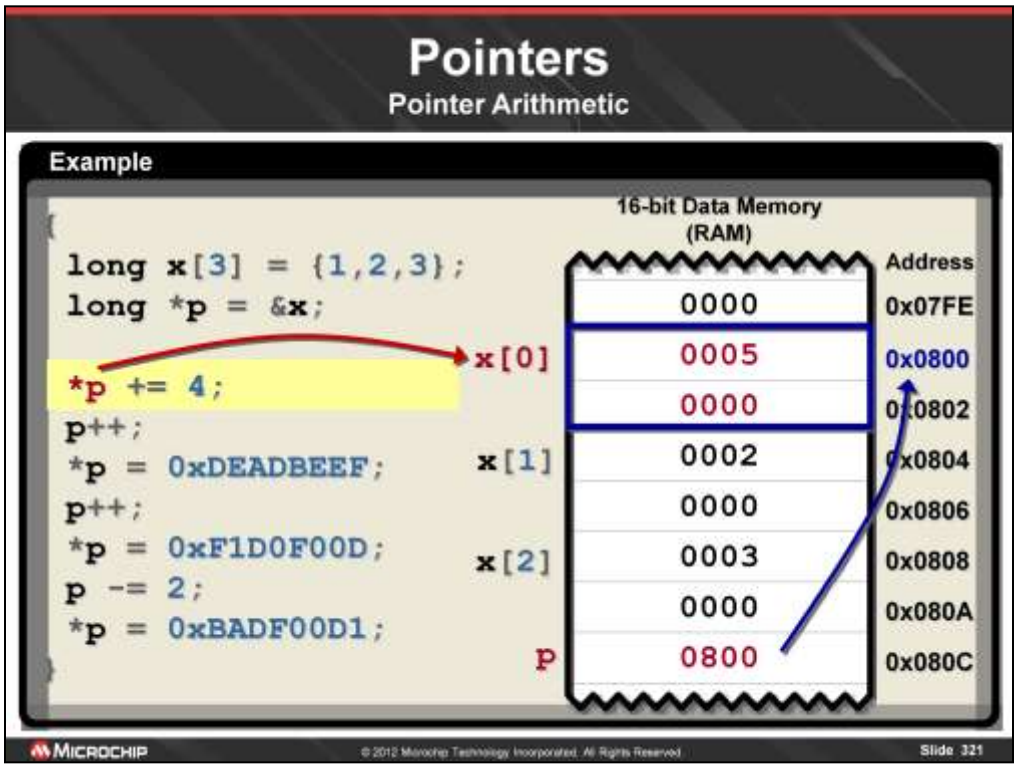
16-bit Data Memory Words

© 2012 Microchip Technology Incorporated. All Rights Reserved.



Here we have a simple program to illustrate the concepts of both incrementing a pointer and using a compound assignment operator to jump more than one element in the array.

First, we initialize the pointer p to point to the array x. In this case, we are initializing the pointer as it is declared. The syntax is exactly the same as doing this for an ordinary variable.



This first line adds 4 to the element pointed to by p, which in this case is x[0]. So, we started off with x[0] = 1:

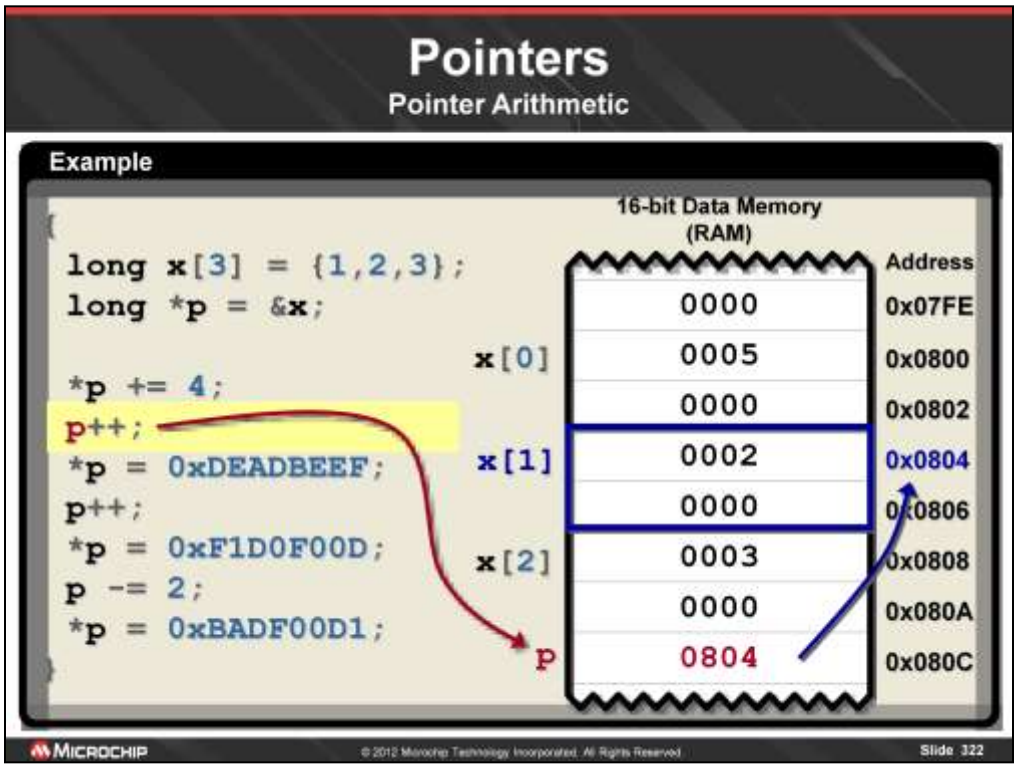
*p += 4

x[0] += 4

x[0] = x[0] + 4

x[0] = 1 + 4

x[0] = 5



Now we increment the pointer. Since we are pointing to a long, the pointer will be incremented by 4 bytes:

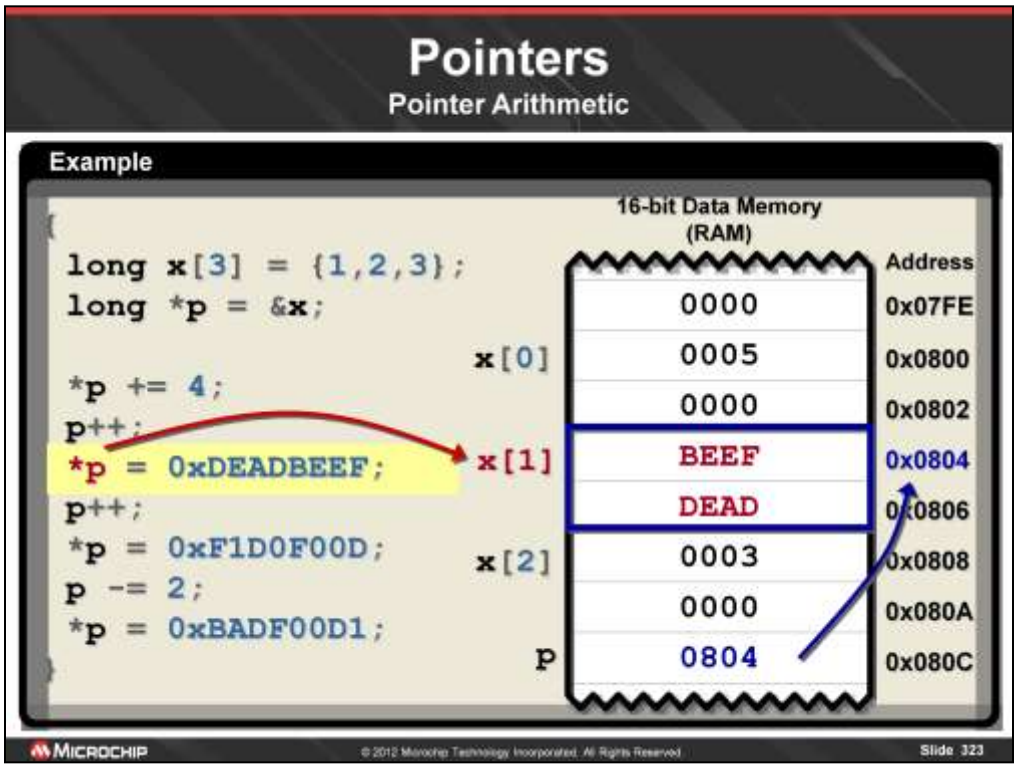
p = 0800

p++

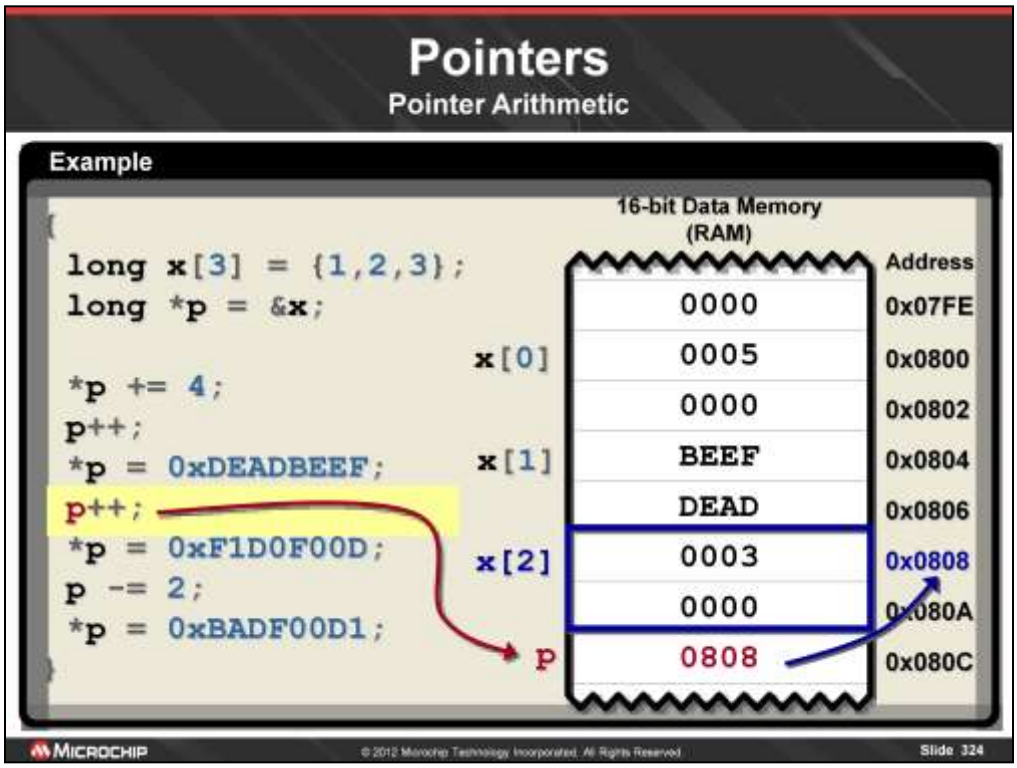
p = p + (1 * 4)

p = 0800 + 4

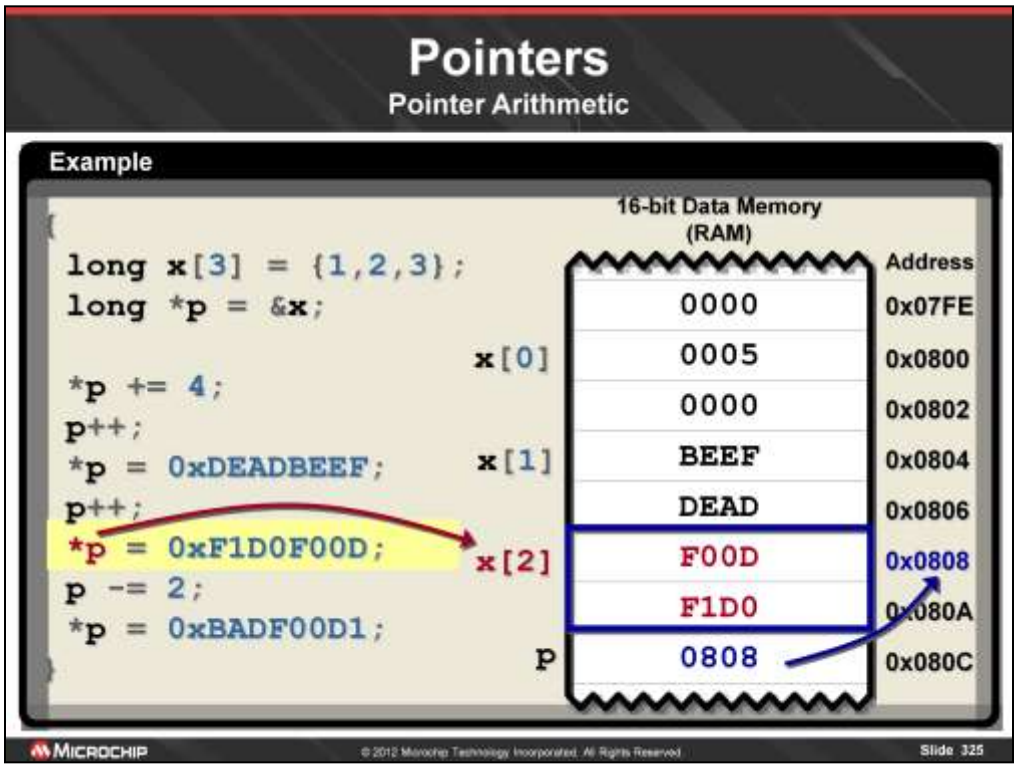
p = 0804



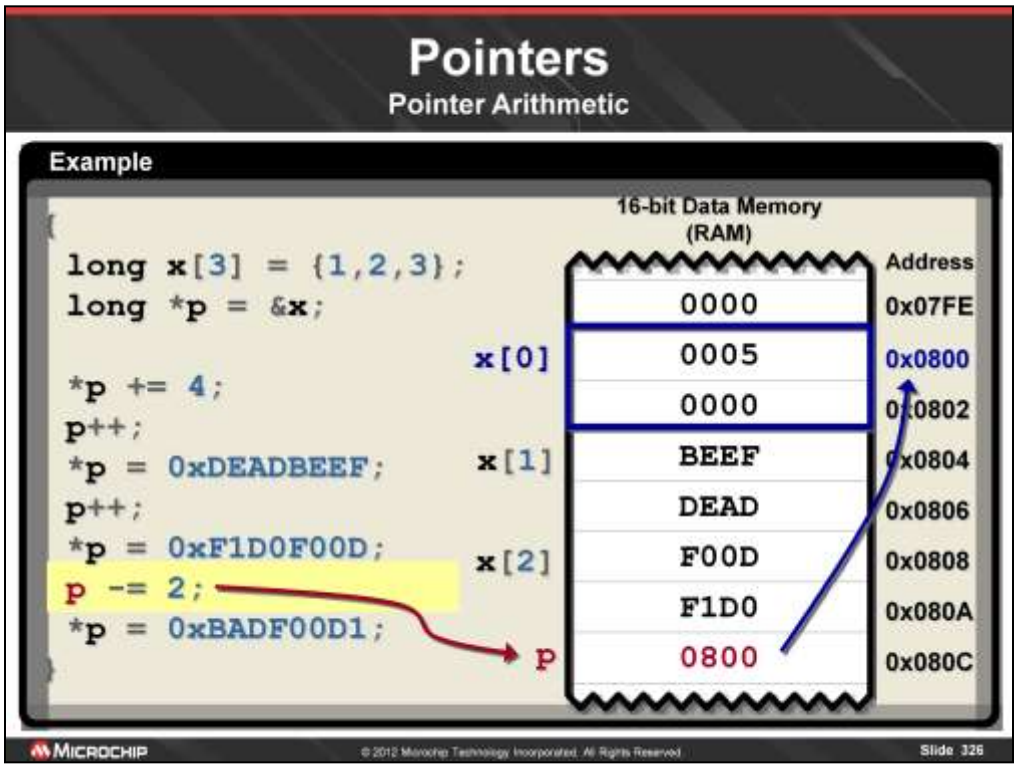
Now, we assign the value 0xDEADBEEF to x[1] via the pointer, which was just incremented.



Increment the pointer again (by 4 bytes)



Assign the value 0xF1D0F00D to x[2] via the pointer.



Now we want to decrement the pointer by 2 elements. Since a long is 4 bytes, we will decrement p by 2 * 4 = 8 bytes.

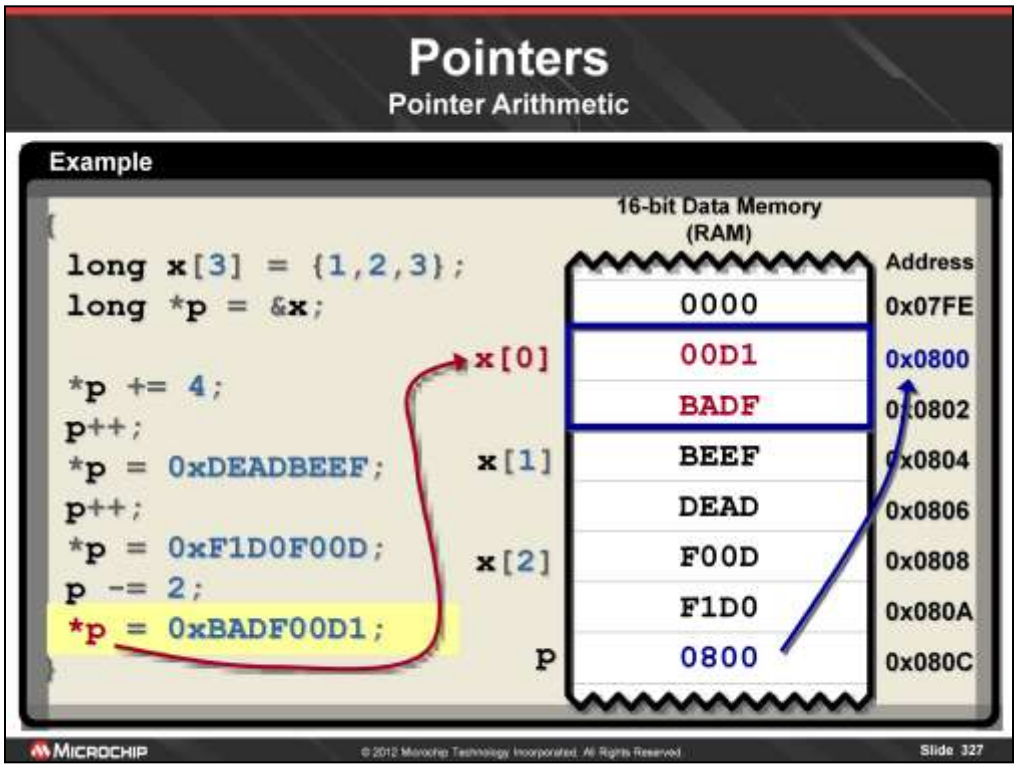
p = 0808

p -= 2

p = p – (2 * 4)

p = 0808 – 8

p = 0800



And finally we overwrite the previous value in x[0] via the pointer with 0xFABF00D1.

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++`
`*(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;`

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 328

Short Explanation: The item inside the parentheses is what gets incremented at the end of the operation.

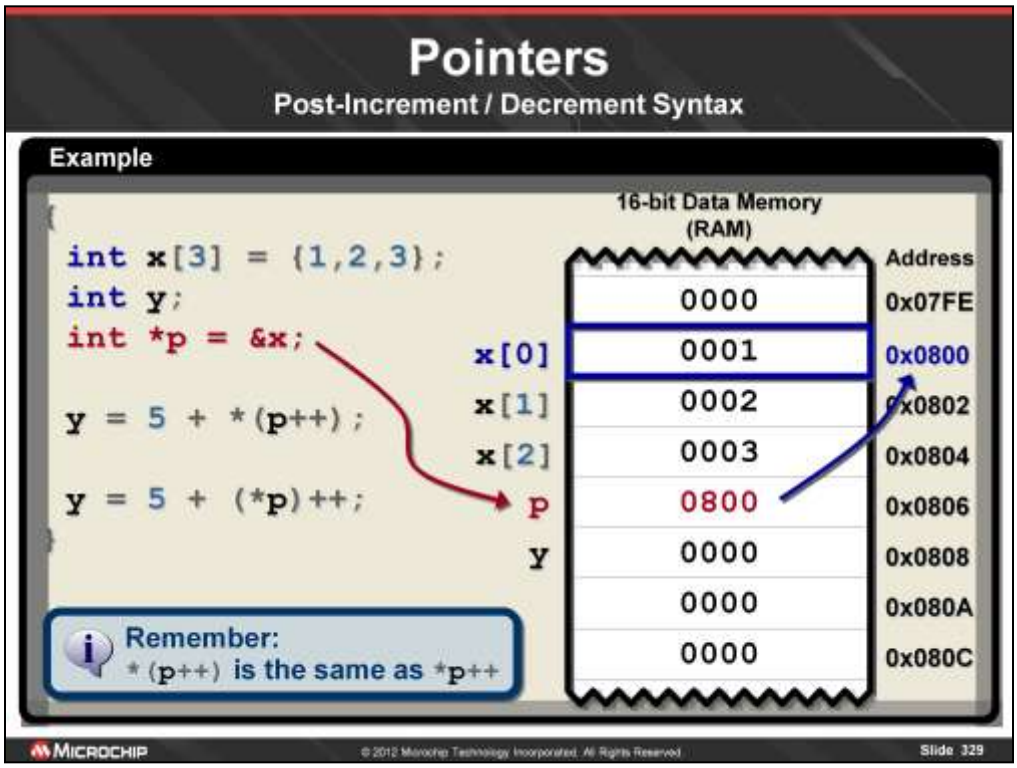
The association of the * operator and ++ operator on pointers is problematic because both * and ++ have the same precedence, but they have right to left associativity, which effectively gives the postfix ++ operator higher precedence than *. Many beginners assume that *p++ will increment the data pointed to by p, when in fact it has no effect on the data, but does increment the pointer variable itself. To be explicit, the operation is carried out as *(p++). When used in an expression, the value of the data pointed to by p will be extracted first, then the pointer will be incremented (postfix ++ uses the variable first, then increments it).

To operate on the data pointed to by the pointer, it is necessary to enclose the dereference operator and the pointer variable in parentheses, essentially showing that the two tokens are inseparable and must be considered together first. So, (*p)++ is the same as x++ if p points to x. The increment operation will be carried out on the variable pointed to by p, while the pointer itself remains unchanged.

Now, you could get really crazy and do something like (*p++)++. This will first take the value pointed to by p and use it in the expression. It then increments both the value pointed to by p and the pointer variable itself. Basically it is the same as doing:

(*p)++;

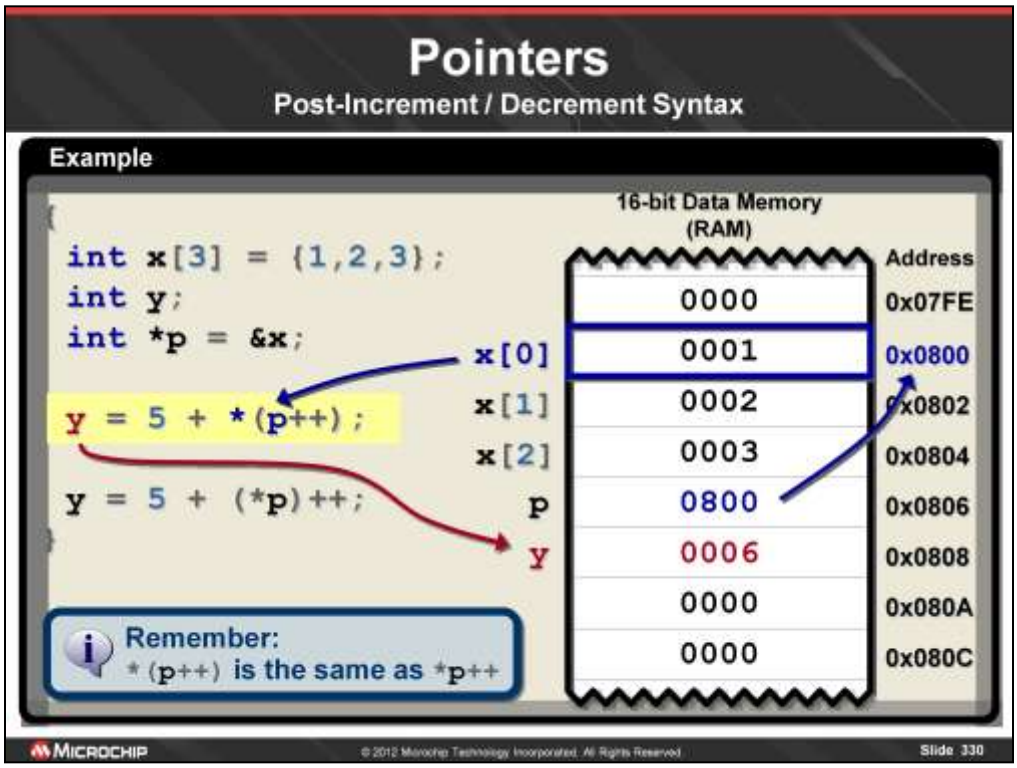
p++;



Lets take a look at how the two different operations work based on the syntax used in each one.

Here we have declared and initialized an array of integers called x, and a regular integer called y. Then, we declare a pointer p, and initialize it with the address of x. (*Remember that the address of the array x is the same as the address of its first element*)

There are only two lines of code, but each line performs two distinct operations.



On this line, we first use the value pointed to by p, which is x[0] to perform the addition operation and store the result of 6 in the variable y.

In other words:

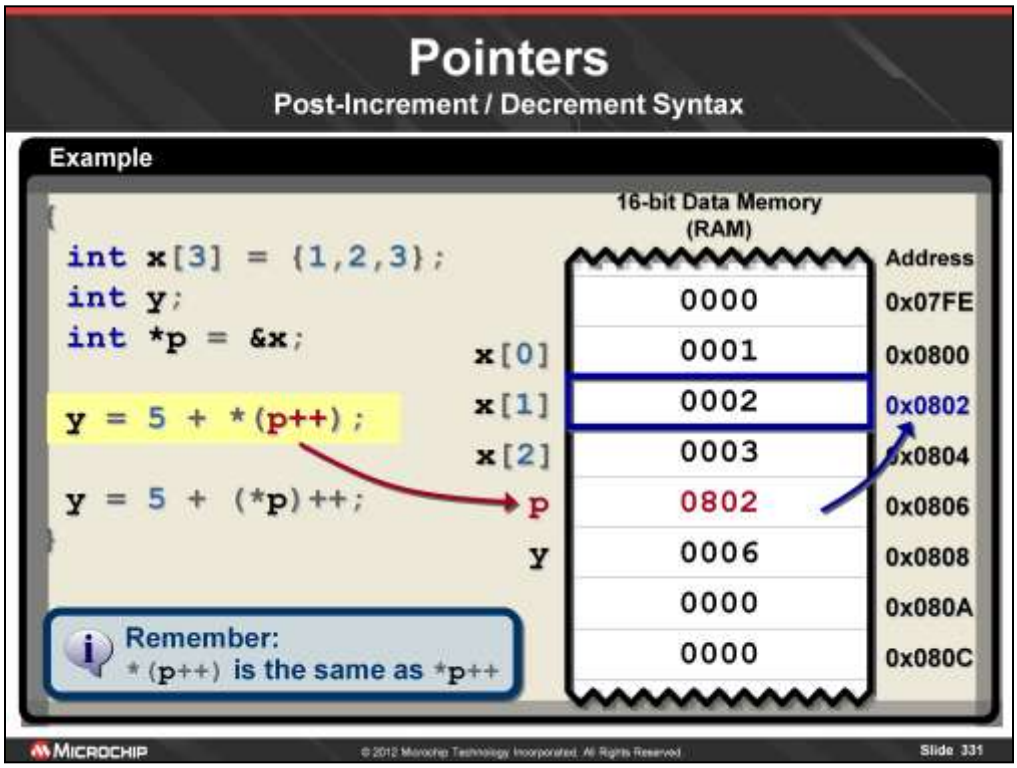
y = 5 + *p

y = 5 + x[0]

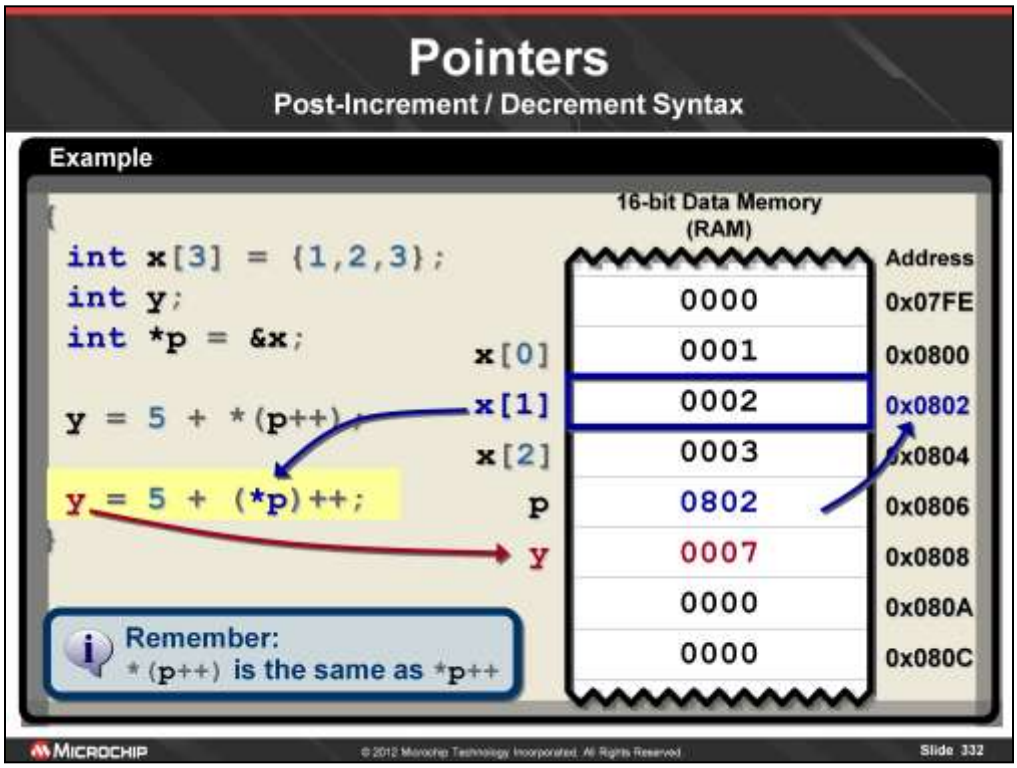
y = 5 + 1

y = 6

*p is the same as using x[0] because we initialized p with the address of the array x. The address of the array x is the same as the address of its first element x[0].



After the addition is performed, the pointer itself is incremented to point to the next integer in the array. Remember, that incrementing a pointer will increment it by the multiple of bytes that the type it points to occupies in memory. Since int occupies two bytes, incrementing p will actually increment it by 2.



On this next line, just like before, we first use the value pointed to by p, which is now x[1] = 2, to perform the addition. The result of 7 is stored in y.

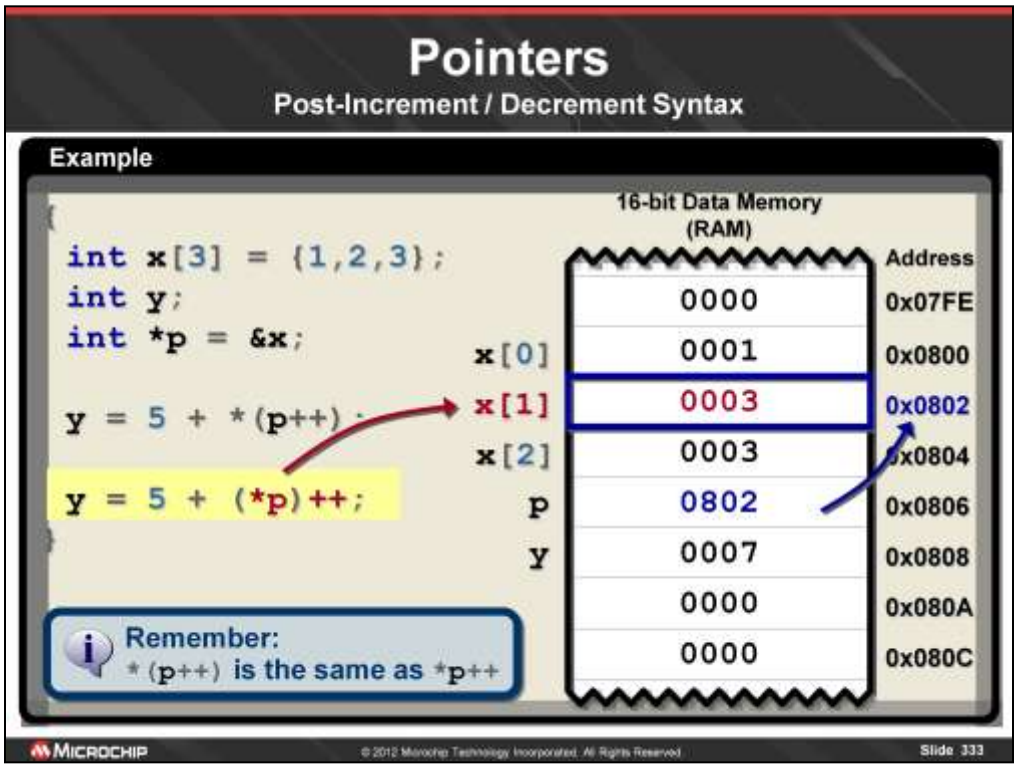
In other words:

y = 5 + *p

y = 5 + x[1]

y = 5 + 2

y = 7



After the addition has been performed, we now increment the value pointed to by p, so x[1] is incremented from 2 to 3.

On both of these lines of code, the increment took place after the addition operation because the increment operator followed the pointer variable (postfix). If the increment operator were prefixed to the pointer variable, the increment operation would have occurred first, then the value pointed to would be used.

The key thing to understand here, is that the parenthesis have nothing to do with the order of operations. They are used to determine what gets incremented – the pointer itself or the value pointed to by the pointer. In both cases, the increment takes place *after* the pointer is used to retrieve the data it points to.

Pointers

Pre-Increment/Decrement Syntax Rule

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

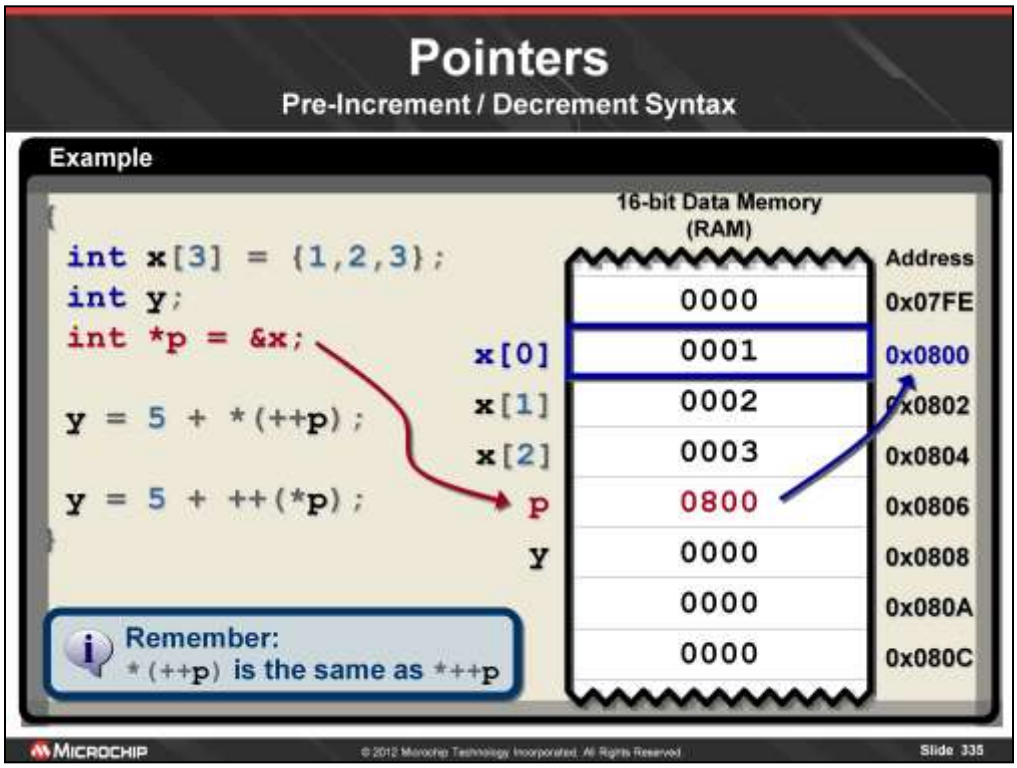
Syntax	Operation	Description by Example

`++p`
`++*p`
`*(++p)`
	Pre-Increment Pointer	`z = *(++p);`
		is equivalent to:
		`p = p + 1;`
		`z = *p;`

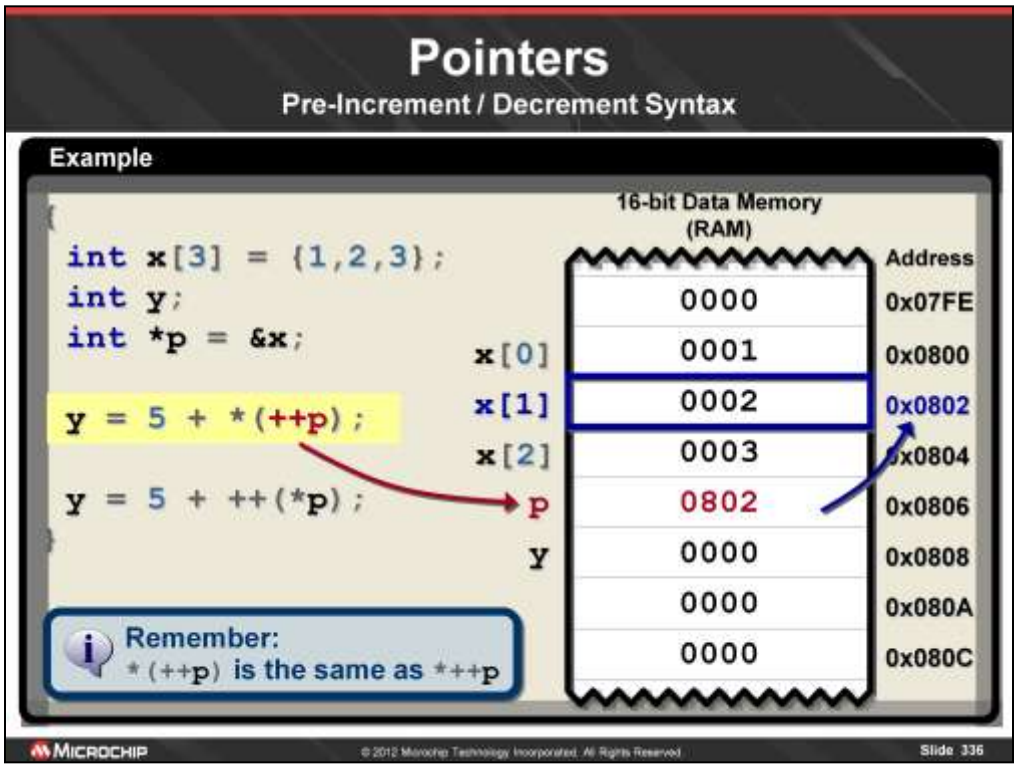
`++(*p)`	Pre-Increment data pointed to by Pointer	`z = ++(*p);`
		is equivalent to:
		`*p = *p + 1;`
		`z = *p;`

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 334

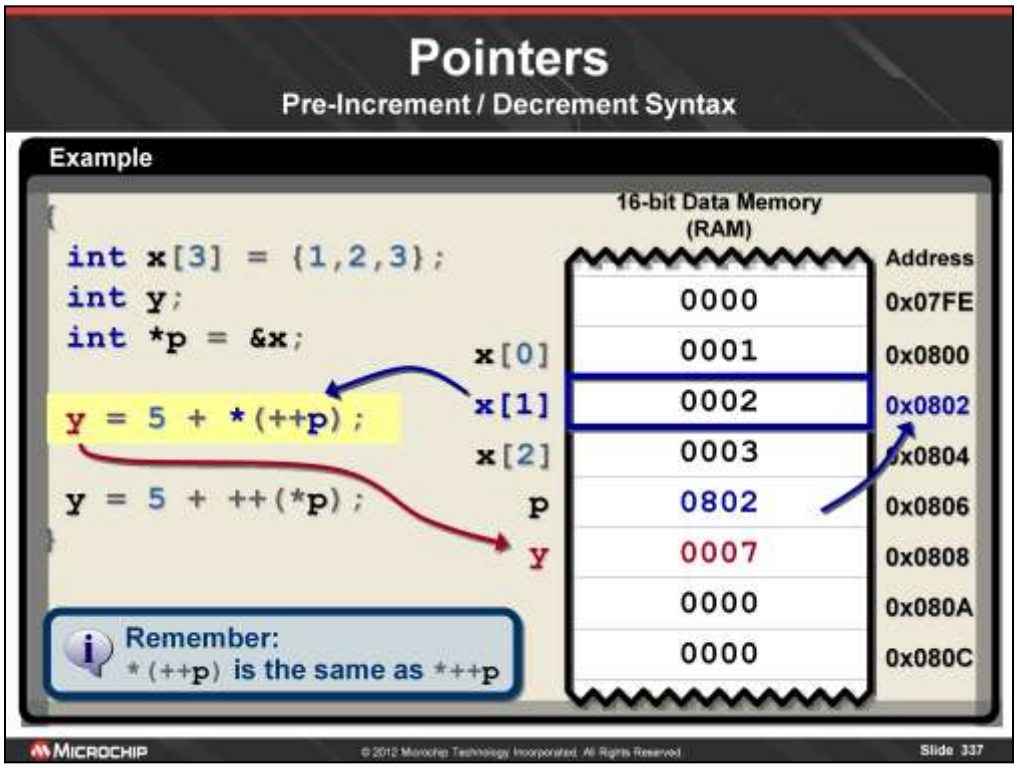
When doing a pre-increment operation, it works just like the post-increment operation, but now the increment operation will take place before the pointer is used. Otherwise the same rules apply.



With this program, the prefix increment means that it will be used before the pointer is used. So here, we first increment the pointer, so that it now points to x[1] instead of x[0] as it did originally.



With this program, the prefix increment means that it will be used before the pointer is used. So here, we first increment the pointer, so that it now points to x[1] instead of x[0] as it did originally.



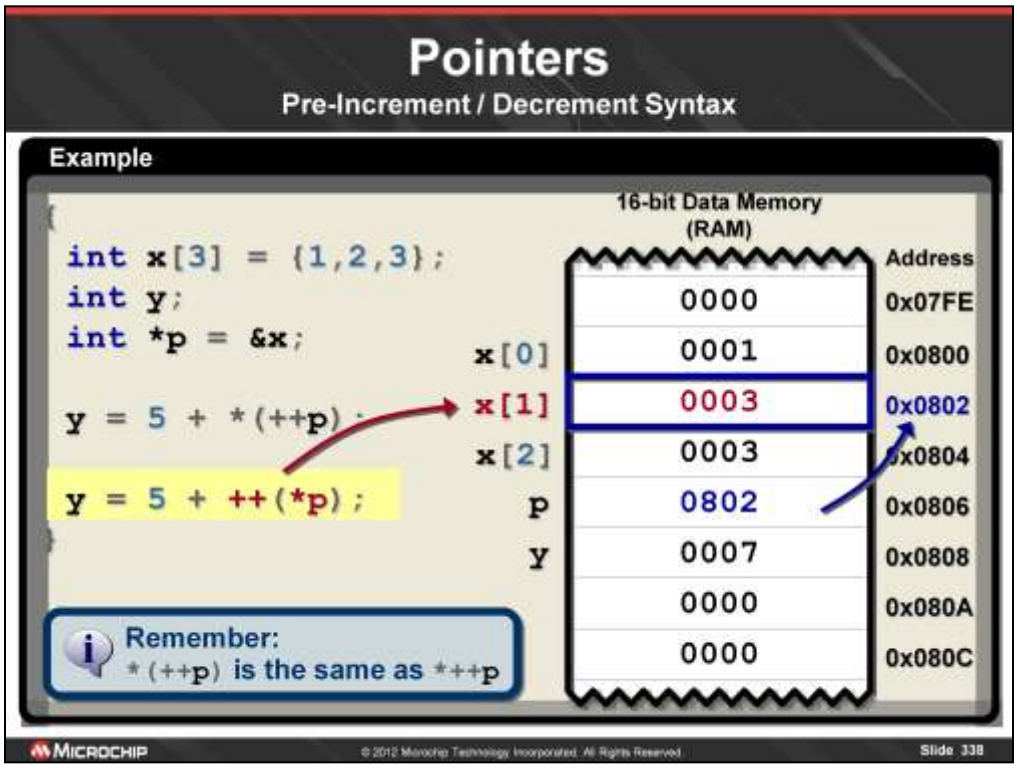
Now that the pointer has been incremented, we will use the value it points to in the addition operation. So:

y = 5 + *p

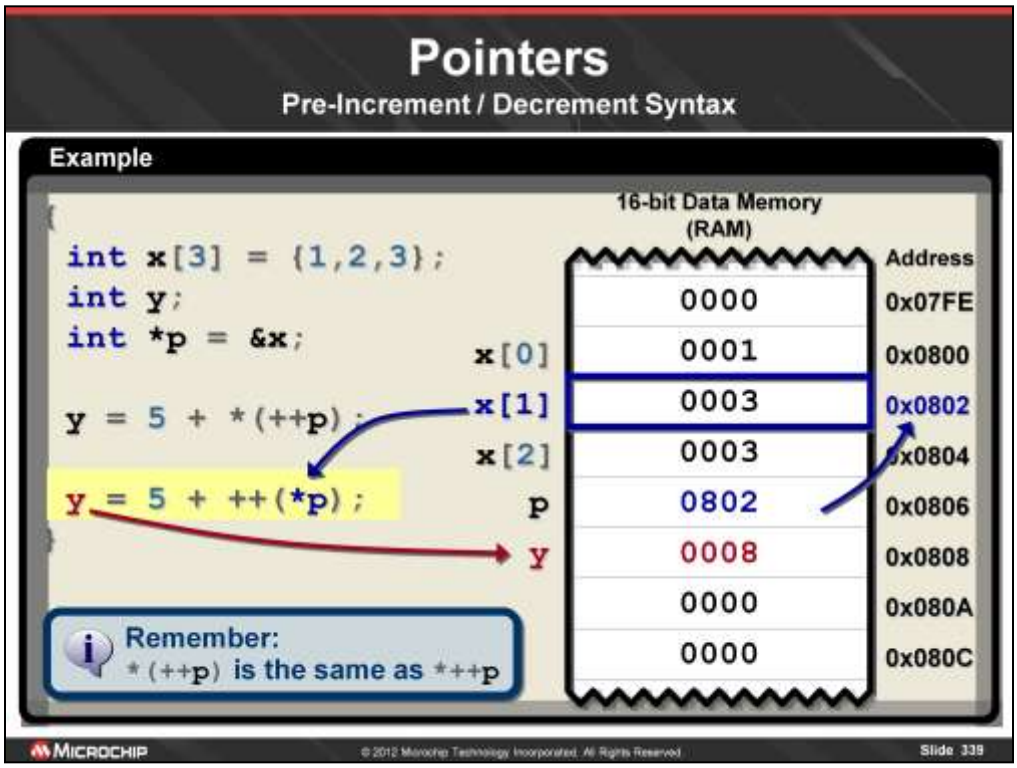
y = 5 + x[1]

y = 5 + 2

y = 7



Now on this line, we first increment the data pointed to by p. So x[1] is incremented from 2 to 3.



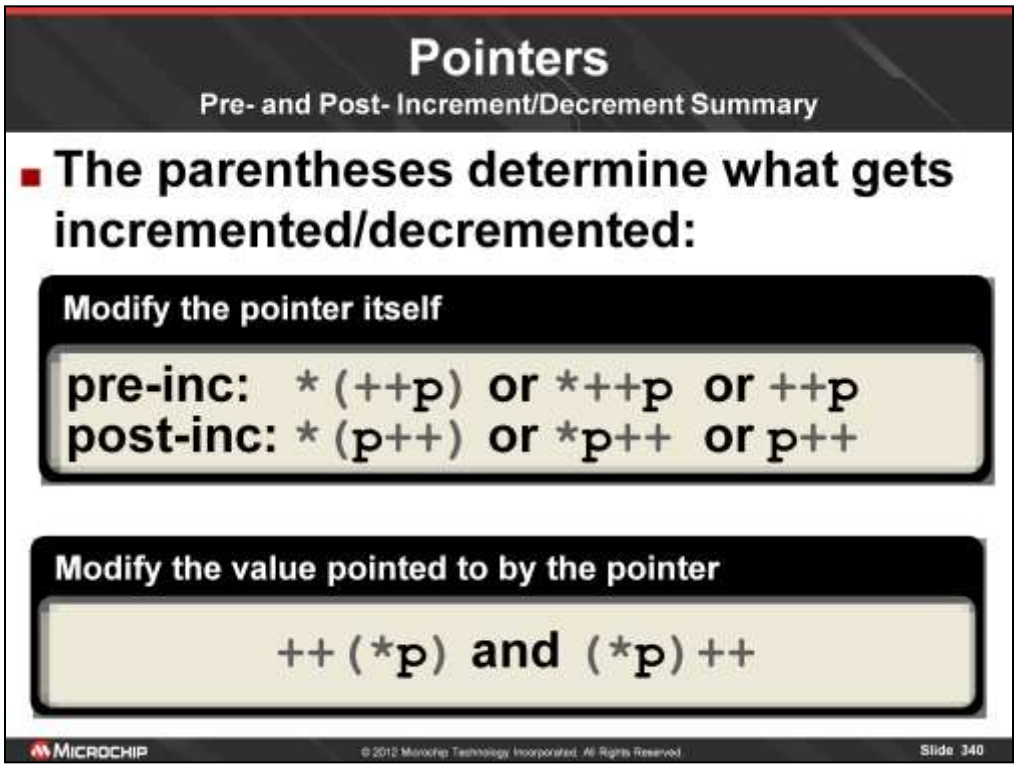
And now the value pointed to by p is used in the addition operation:

y = 5 + *p

y = 5 + x[1]

y = 5 + 3

y = 8

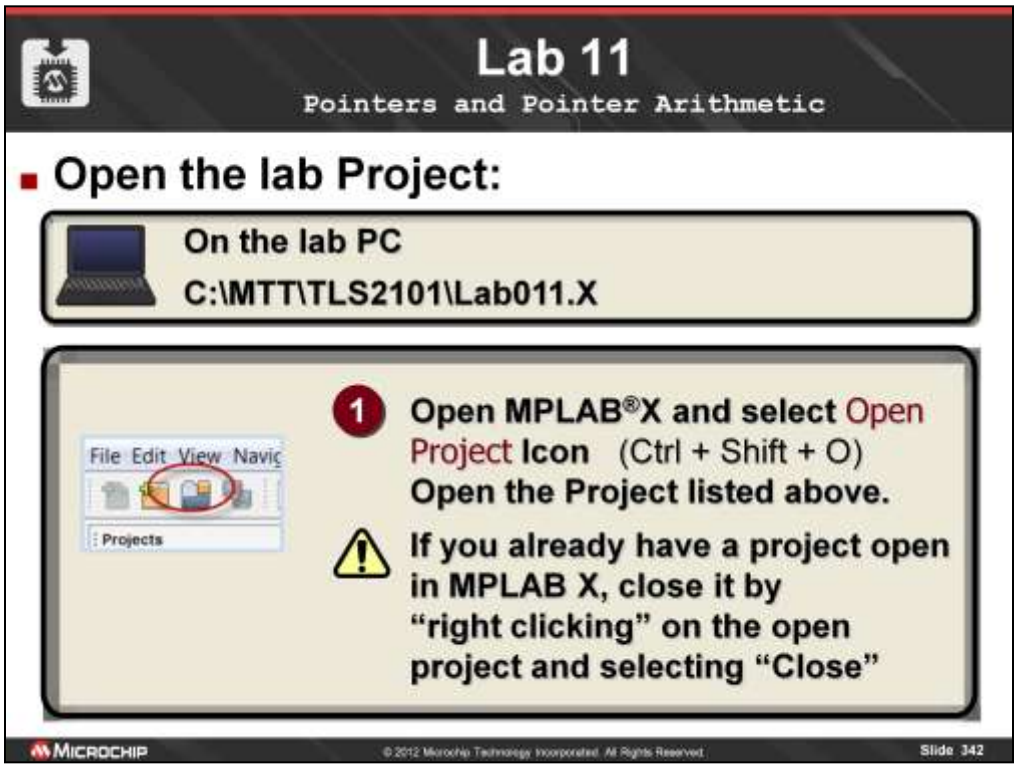


So to summarize, the parentheses around the pointer don't affect the order of operations (i.e. whether to increment before or after using the pointer). Only the position of the increment or decrement operator affects the order: prefix makes the increment happen before the pointer is used and postfix makes the increment happen after the pointer is used.

The parentheses determine whether the pointer or the item it points to is what gets incremented. The rule is : if the * (dereference operator) is outside the parentheses or no parentheses are used at all, the pointer is what will be incremented. If the * is inside the parentheses, the value pointed to by the pointer is what will be incremented.

This can be very confusing to someone new to C as well as to some experts. Just try to remember – it's just syntax. It might be a little ugly, but it does work.

MICROCHIP
Lab Exercise 11
Pointers and Pointer Arithmetic



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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;
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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++;
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP

Data Pointers

(continued)

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 347

Normally in C, function parameters are passed by value. This means that if you pass a variable as a parameter to a function, the function makes a copy of that value to operate on and it leaves the original variable alone. As we can see in this example, we pass the variable x to the function square. x is initialized to 2, and is still 2 after the function call. The value of x is copied into n and it is n that is squared and returned from the function to the variable y.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

In many situations, it is desirable to let a function modify the actual variable passed to it. In order to do this, we use a pointer in the function parameter (pass by reference).

In order to use a function like this, you need to pass an address or a pointer to the function when it is called.

So basically, you are telling the function the address of the variable that you want it to modify.

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`

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Pointers and Functions

Passing Parameters By Reference

Example – Part 1

Swap function definition:

We know where you live!

```
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.

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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…

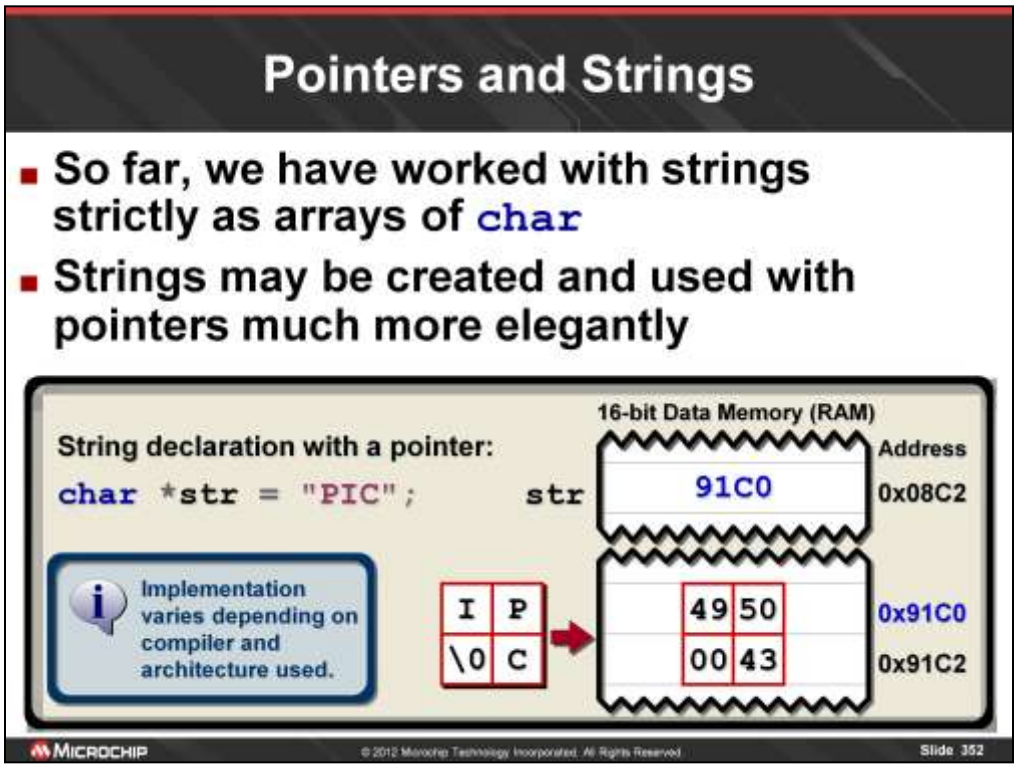
```
n1 = &x
n2 = p
```

After running program:

x = 10

y = 5

© 2012 Microchip Technology Incorporated. All Rights Reserved.



A string may be declared using a pointer just like it was with a char array, but now we use a pointer variable (no square brackets) instead of an array variable. The string may be initialized when it is declared, or it may be assigned later. The string itself will be stored in memory, and the pointer will be given the address of the first character of the string.

At this point, it is only important to understand that the string is "somewhere" in memory and that the pointer points to the first character. Specific implementation details will be covered in the compiler/architecture specific classes since the underlying hardware architecture and the specific compiler determine exactly how this will be implemented.

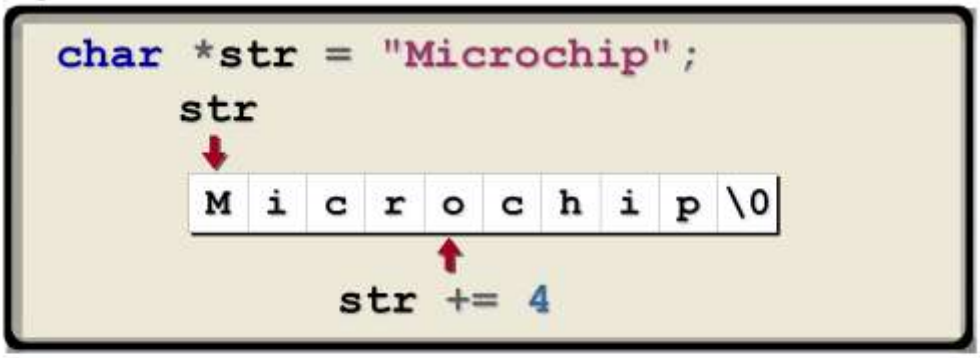
In case someone asks about implementation details and demands an immediate answer during this class:

In general, with most architectures, the string literal is stored in program memory and the pointer will be to a program memory address. This is how the PIC18 and C18 handle it. However, on 16-bit PICs, the string will be stored in program memory and accessed via the PSV window. The address stored in the pointer will be 8000h larger than the actual program memory address. For example, **str** is at 0x08C2 and contains the address 0x91C0. The actual string starts at 0x11C0 in *program* memory even though it appears to be in data memory. Because of the PSV window, all C30 pointers are only 16-bits in size and they never point to program memory – the PSV window will always be used (so 16-bits is plenty). With C18, there are data pointers (16-bit) and program pointers (24-bit). Program pointers in C18 make use of the table read mechanism to read data from program memory.

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Pointers and Strings

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

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

M	i	c	r	o	c	h	i	p	\0

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

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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).

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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.

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 357

When I first started working with strings in C, I fell back on what I knew from other high-level languages which have string data types. In those cases the natural thing to do is test a string variable for equivalence with a string literal. However, a string pointer in C is not even close to a string data type. If we do a comparison like if (str == "Microchip"), the compiler will compare the address contained in the pointer str with the address of the string literal "Microchip" (the literal is stored in program memory and accessed via the PSV on the 16-bit devices). This might work if the compiler is smart enough to recognize that the literal in the comparison is the same one used to initialize str and use the same literal in memory. However, the string in the comparison might be allocated as a separate distinct variable that happens to have the same characters by some compilers. If this happens, the address in str will not match the address of the string literal and the comparison will fail.

The correct way to compare strings is to use the standard library function strcmp() which will compare the 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\XC16\v1.10\include\string.h`

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 358

Because strings in C are just arrays of characters (or simply a string of characters stored in sequential memory locations) terminated by a null character, the only way to compare them is to do it character by character.

C makes up for its lack of a string data type by including numerous string handling functions in its standard library. The strcmp() function is used to compare two strings. It takes two pointers as parameters. Note that they are both declared as const, which means the function cannot change the contents of the strings themselves – though it can change the value of the pointers, which it does in order to move from character to character. This is one of the legitimate uses for the const keyword that I alluded to when we discussed #define versus const. Here, we are not creating a constant, but rather restricting the function from changing the value of the parameters passed to it (the strings pointed to by s1 and s2), even though they are passed by reference.

If the strings are equal, it will return 0 (which unfortunately is interpreted as "FALSE" in Boolean type operations – so you must test the function in an expression to see if it is == 0). If the strings are not equal, it will return a non-zero value.

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);
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 359

We can use strcmp() in a number of different ways. In this example, we compare the string pointer str with a string literal "Microchip". This works because the string literal is stored in program memory and has an address. When used in this fashion, the address of the string literal is what gets passed to the function. When this code is compiled, the string "Microchip" will exist in two distinct locations – one from where it is initially assigned to str, and a second for its use in the strcmp() function. These two strings are two distinct entities with their own addresses. (The compiler might optimize this so that both instances of the string literal point to the same memory location)

The strcmp() function will compare the two strings character by character. Since they are the same, the function will return 0, which makes the expression in the if statement true.

Arrays of Pointers

Declaration

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

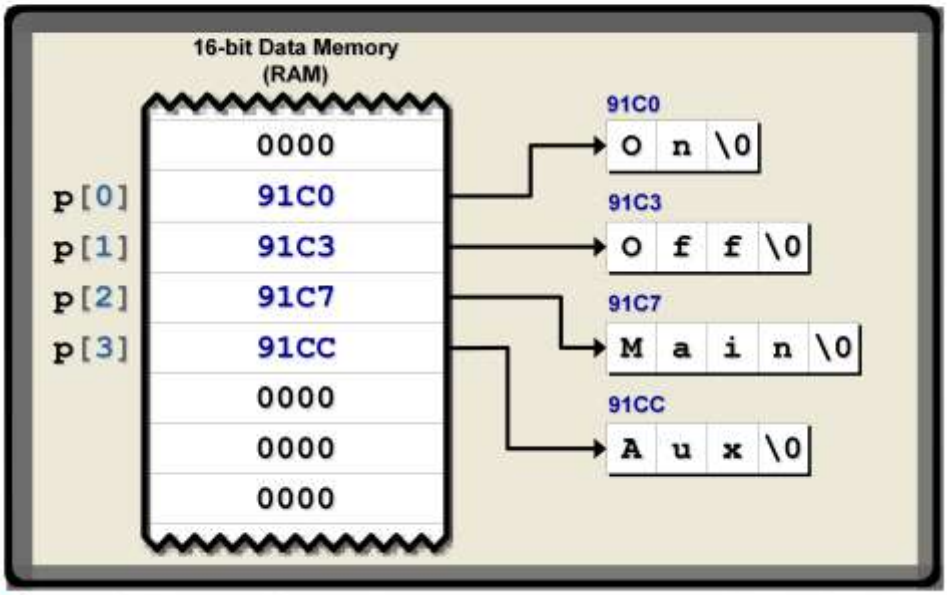
```
char *p[4];
```

- 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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Arrays of Pointers

Array Elements are Pointers Themselves



MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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";
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Initializing an element of an array of pointers is just like initializing ordinary pointers, but we must include the array index. The first example assigns the address of x to the pointer in element 0 of the array. The second example assigns the address of the first character in the string to the pointer in element 0 of the array. This second example is how you would populate the array of pointers shown on the previous slide. More explicitly:

p[0] = "On";

p[1] = "Off";

p[2] = "Main";

p[3] = "Aux";

This would create the strings in memory and initialize each of the pointers in the array to point to a particular 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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

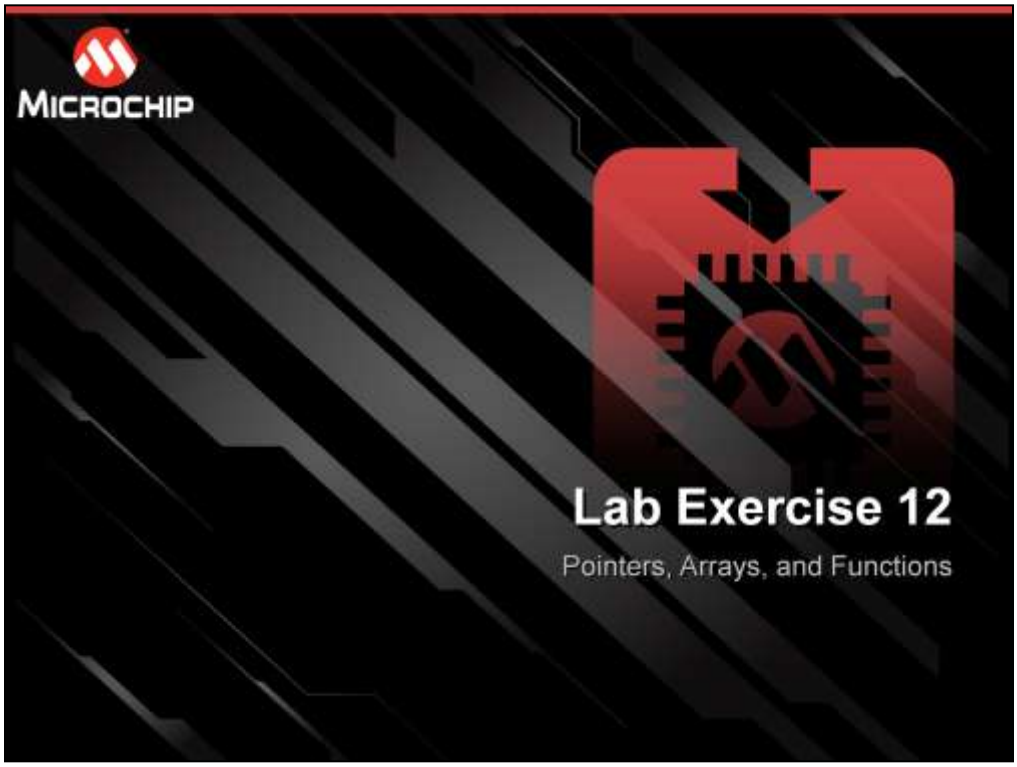
Arrays of Pointers

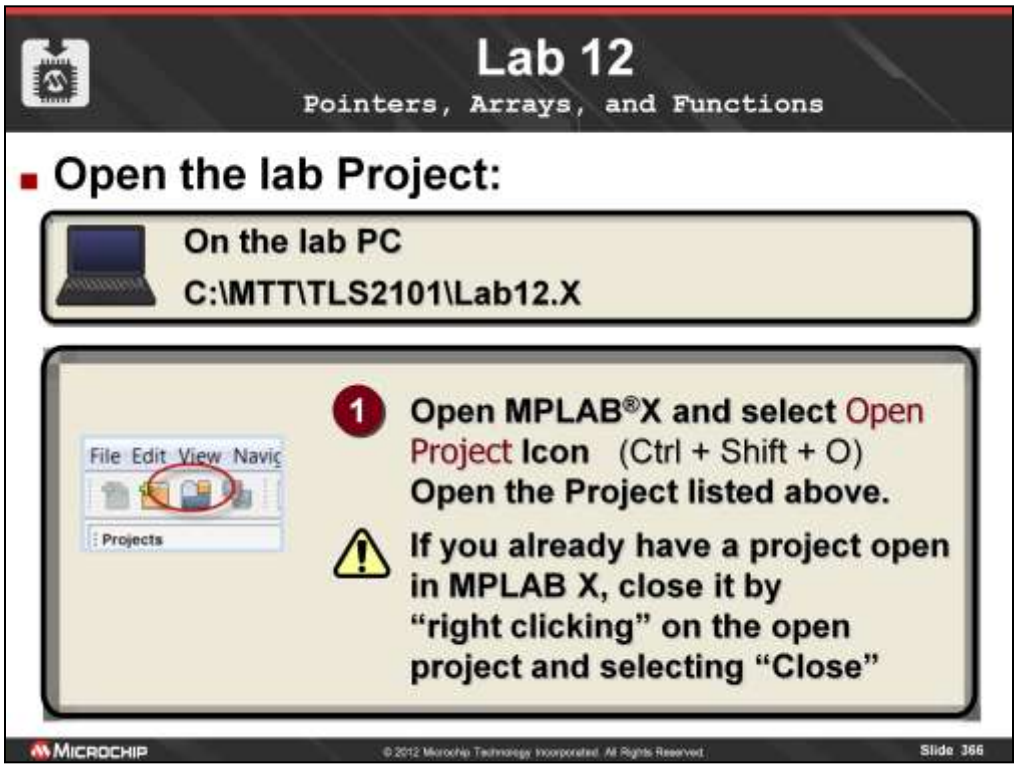
Accessing Strings

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);
}
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.


MICROCHIP
Lab Exercise 12
Pointers, Arrays, and Functions



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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 reversel().  Use the constant
         ARRAY_SIZE for the second parameter.
         See definition of function reversel() below.
*****************************************************************************/
/Reverse order of elements by passing array
        reversel(a, ARRAY_SIZE);
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

<end of part three>

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

NOTE: Parentheses are required to create a pointer to a function. If we left off the parentheses:

int *fn(int x);

We would have declared a function that returns a pointer to type int. Functions that return pointers will be covered when we discuss the heap and dynamic memory allocation.

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
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 373

NOTE: The address of operator & is not required. When using the function name alone, it is like using an array name alone. The function name by itself represents the address (starting point) of the function in program memory.

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);
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

NOTE: No dereference operator is used when calling the function via its pointer.

Function Pointers

Passing a Function to a Function

Example 1: Understanding the Mechanism

```
int x,y;
int add(int a, int b);        //Function prototype
int sub(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, &add);  //Pass address of add
   y = foobar(5, 12, &sub);  //Pass address of sub
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 375

www.function-pointer.org provides some great examples of function pointer applications.

Function pointers may be used to call functions using either of these formats:
fp(a,b)
(*fp)(a,b)

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

function to be evaluated

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

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

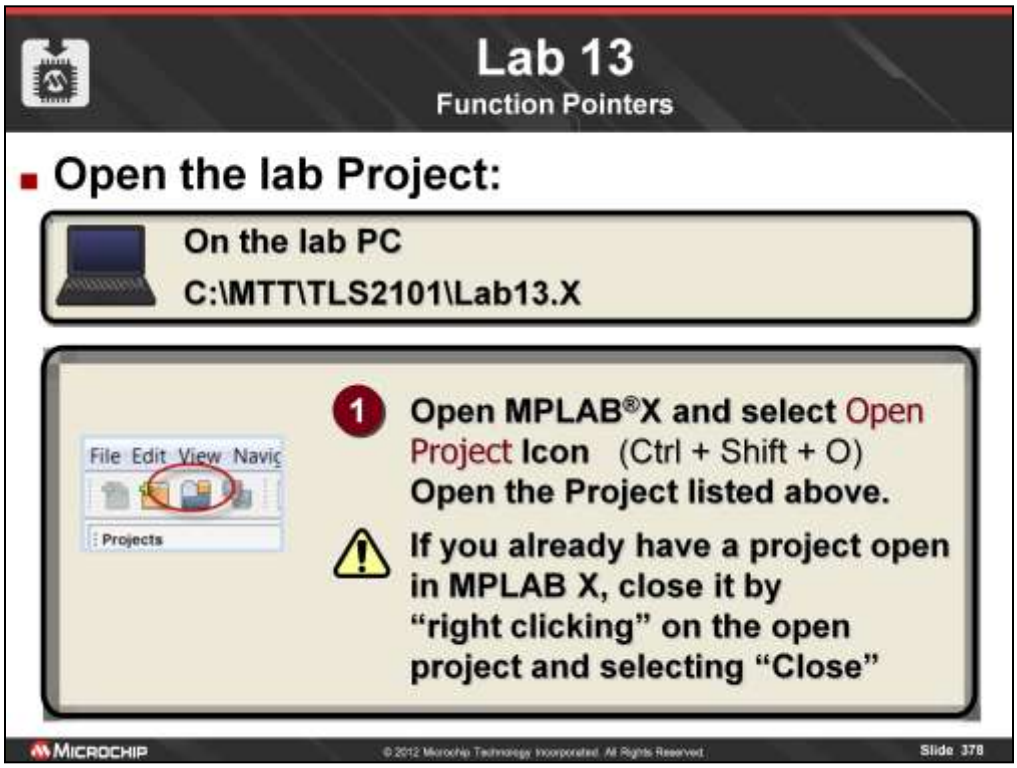
MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

This is a modified version of the example from: http://en.wikipedia.org/wiki/Function_pointer

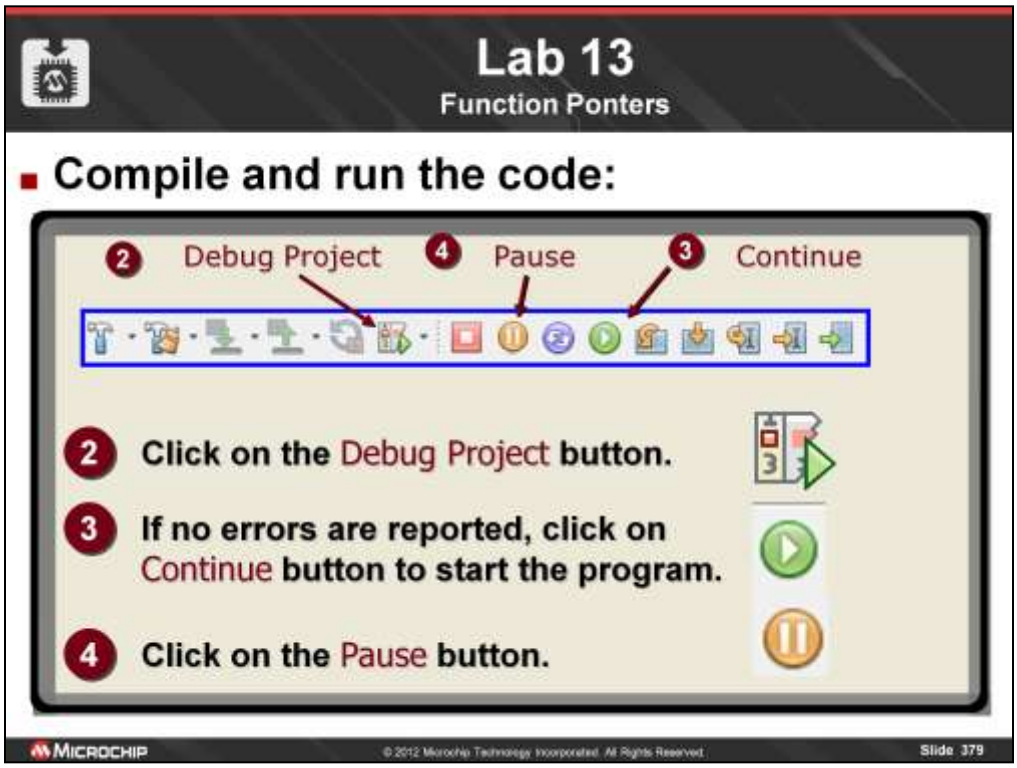
(1) Changed types to float from double

(2) Corrected function parameter syntax (removed x from function parameter definition)

MICROCHIP
Lab Exercise 13
Function Pointers



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.



Next, click on the Build All button to compile the code.
If you don't get any errors (you shouldn't), then go ahead and hit the run button.
After a second or two, hit the halt button, and then let's look at the code and what it did.

Lab 13
Function Pointers

- **Results**

UART 1 Output × | Lab13 (Build, Load, ...) × | Debugger Cons

```
y1 = integral of x dx over 0 to 1 = 0.500000
y2 = integral of x^2 dx over 0 to 1 = 0.335000
y3 = integral of x^3 dx over 0 to 1 = 0.252500
```

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

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

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

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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);
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 13

Function Pointers

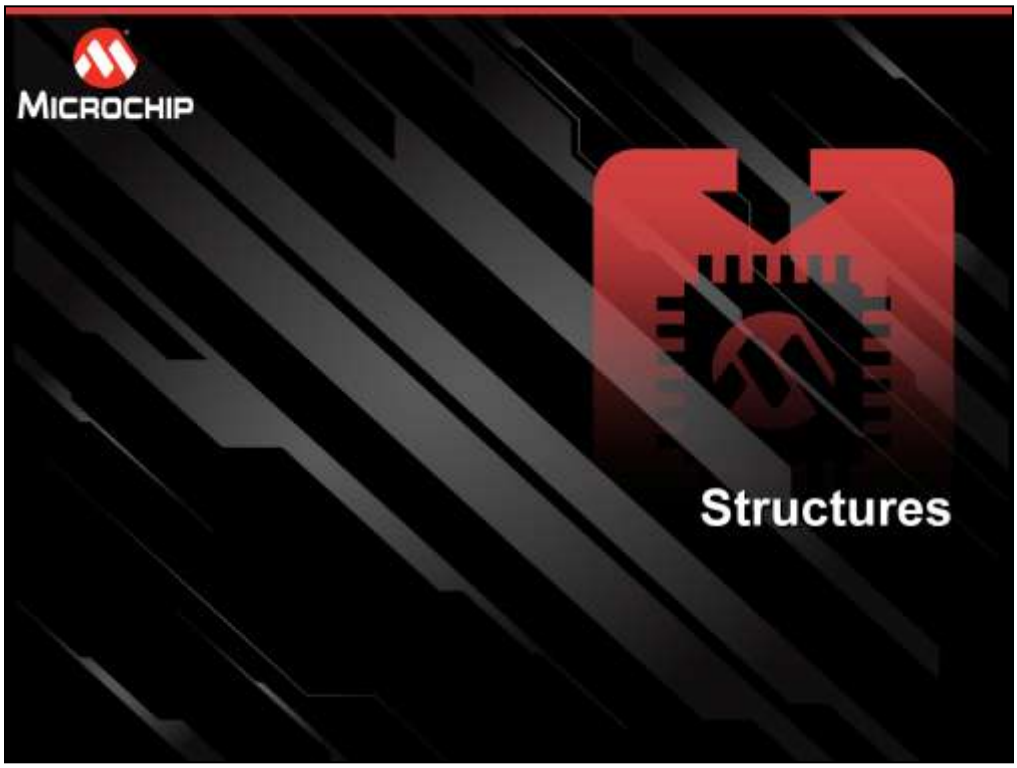
```
/*==========================================================================
 FUNCTION:      integral()
 DESCRIPTION:   Evaluates the integral of the function passed to it over the
                interval a to b.
 PARAMETERS:    interval end points a & b and function to integrate
 RETURNS:       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;
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.


MICROCHIP
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**

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
}
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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
  ...
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

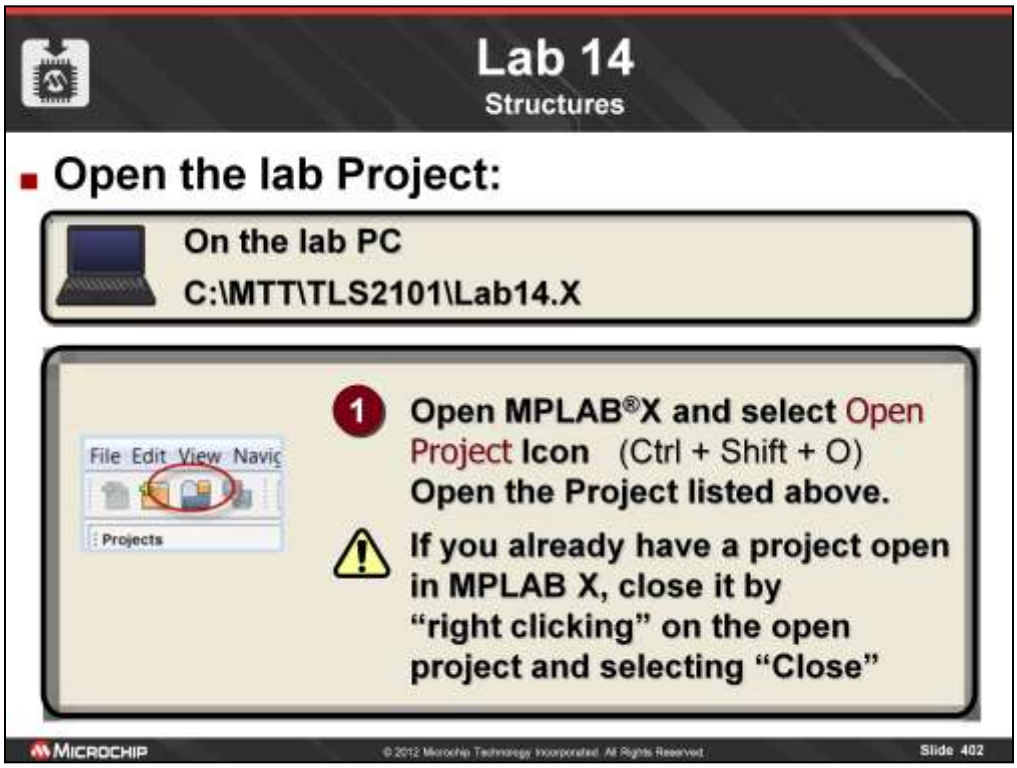
```
struct strings
{
  char a[4];
  char *b;
} str;
```

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";
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 14
Structures



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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);
```

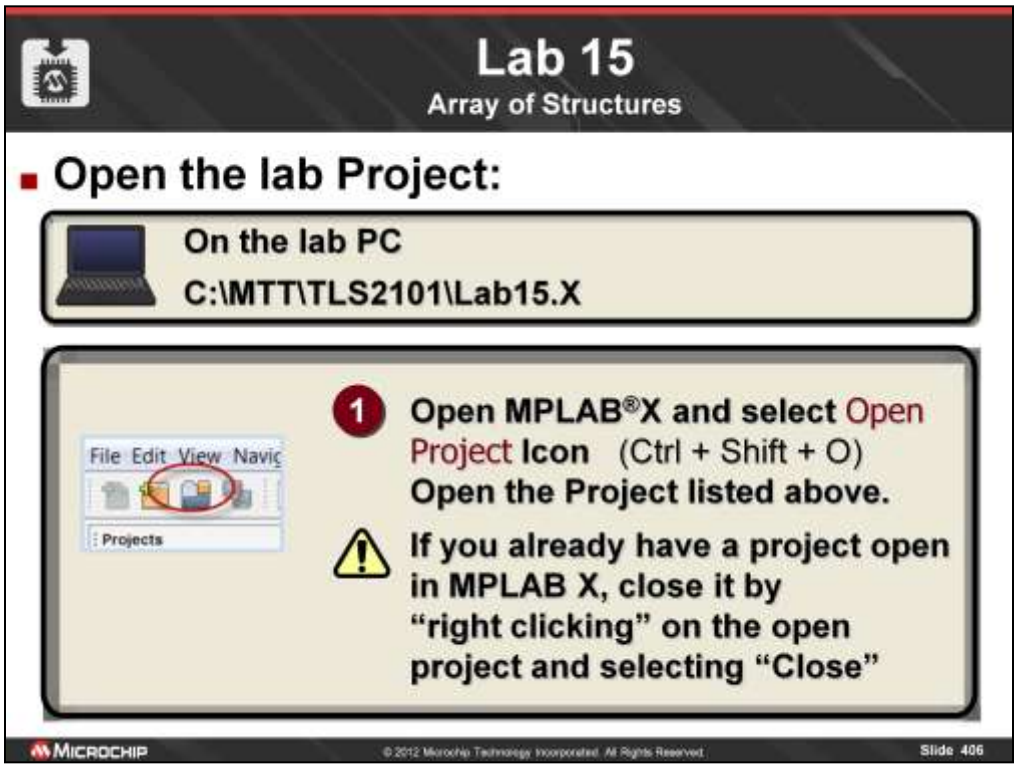
MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 15
Arrays of Structures



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

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;
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP

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**

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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`

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Bit Fields

How to Use a Bit Field

Example

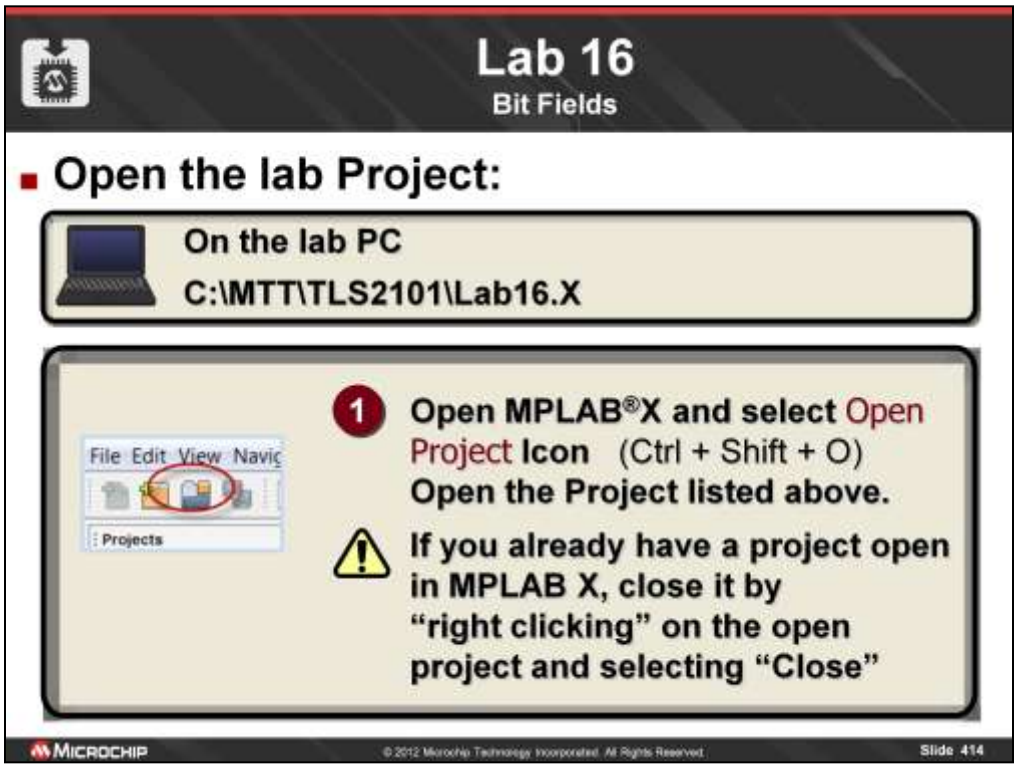
```
struct byteBits
{
  unsigned a: 1;
  unsigned b: 1;
  unsigned c: 2;
  unsigned d: 1;
  unsigned e: 3;
} x;

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
}
```

Byte in Data Memory (RAM)

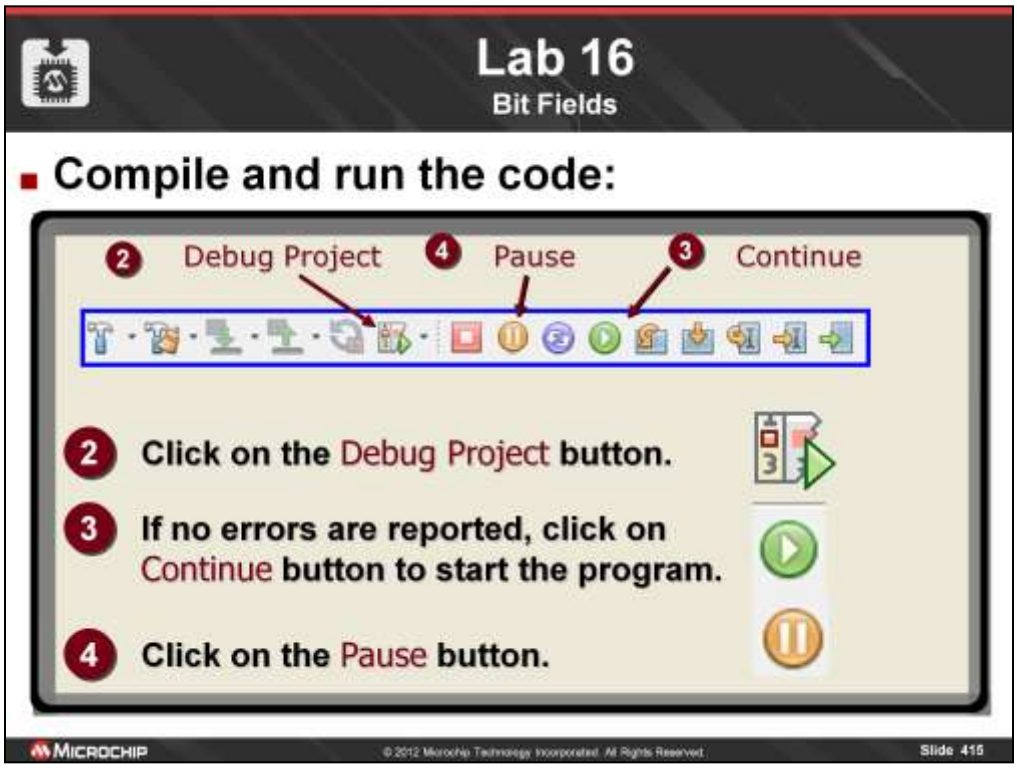
	7	6	5	4	3	2	1	0
x	1	1	1	0	1	0	0	1
	e			d	c		b	a

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 16
Bit Fields



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.



Next, click on the Build All button to compile the code.
If you don't get any errors (you shouldn't), then go ahead and hit the run button.
After a second or two, hit the halt button, and then let's look at the code and what it did.

Lab 16

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;
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 16

Bit Fields

Demo Results 1

Variables | Output

Name	Type	Address	Value
bitByte	(2) Bytes	0x856	
fullByte	char	0x856	0x55
bitField	struct	0x856	
bit0	int	0x856	0x01
bit1	int	0x856	0x00
bit2	int	0x856	0x01
bit3	int	0x856	0x00
bit4	int	0x856	0x01
bit5	int	0x856	0x00
bit6	int	0x856	0x01
bit7	int	0x856	0x00

```
bitByte.fullByte = 0x55;
```

(ctrl + shift + 1) to open Variables Window

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 16

Bit Fields

Demo Results 2

Variables | Output

Name	Type	Address	Value
bitByte	(2) Bytes	0x856	
fullByte	char	0x856	0x54
bitField	struct	0x856	
bit0	int	0x856	0x00
bit1	int	0x856	0x00
bit2	int	0x856	0x01
bit3	int	0x856	0x00
bit4	int	0x856	0x01
bit5	int	0x856	0x00
bit6	int	0x856	0x01
bit7	int	0x856	0x00

```
bitByte.bitField.bit0 = 0;
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 16

Bit Fields

Demo Results 3

Variables | Output

Name	Type	Address	Value
bitByte	(2) Bytes	0x856	
fullByte	char	0x856	0x50
bitField	struct	0x856	
bit0	int	0x856	0x00
bit1	int	0x856	0x00
bit2	int	0x856	0x00
bit3	int	0x856	0x00
bit4	int	0x856	0x01
bit5	int	0x856	0x00
bit6	int	0x856	0x01
bit7	int	0x856	0x00

```
bitByte.bitField.bit2 = 0;
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 16

Bit Fields

Demo Results 4

Variables | Output

Name	Type	Address	Value
bitByte	(2) Bytes	0x856	
fullByte	char	0x856	0xD0
bitField	struct	0x856	
bit0	int	0x856	0x00
bit1	int	0x856	0x00
bit2	int	0x856	0x00
bit3	int	0x856	0x00
bit4	int	0x856	0x01
bit5	int	0x856	0x00
bit6	int	0x856	0x01
bit7	int	0x856	0x01

```
bitByte.bitField.bit7 = 1;
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 16
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)

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
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`**

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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;
}
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

Unions

What's the purpose of a union?

- Unions are similar to structures EXCEPT
  - `struct` by itself outside of a union assigns a new memory location upon declaration

Example

```
Struct
{
  char Byte1:8;
  char Byte2:8;
}myVar;

myVar.Byte1 = 0xF0;    // myVar = 0000000011110000
myVar.Byte2 = 0xF0;    // myVar = 1111000011110000
```

  - So how do we assign a value to `myVar` as a single word?

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Unions

What's the purpose of a union?

- `union` allows the same memory location to be viewed and manipulated as different data types

Example

```
union
  int Word;
  struct
  {
    char Byte1:8;
    char Byte2:8;
  };
}myVar;

myVar.Word  = 0xFFFF;    // myVar = 1111111111111111
myVar.Byte1 = 0xF0;      // myVar = 0000000011110000
myVar.Byte2 = 0xF0;      // myVar = 1111000011110000
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

Unions
example from typical .h file

SR, STATUS Register

-	-	-	-	-	-	-	DC	IPL<2:0>		RA	N	OV	Z	C
15														0

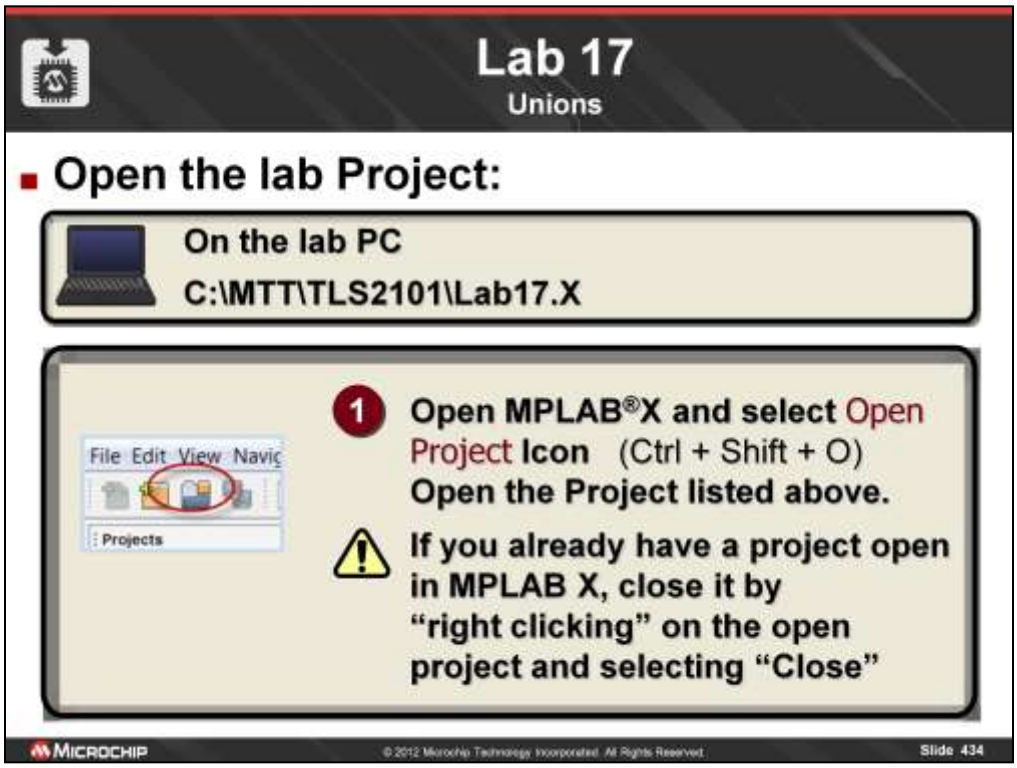
PIC24FJ128GA010.h

```
extern volatile unsigned int SR __attribute__((__sfr__));
union {
    struct {
      unsigned C:1;
      unsigned Z:1;
      unsigned OV:1;
      unsigned N:1;
      unsigned RA:1;
      unsigned IPL:3;
      unsigned DC:1;
    };
    struct {
      unsigned :5;
      unsigned IPL0:1;
      unsigned IPL1:1;
      unsigned IPL2:1;
    };
} SRBITS;
extern volatile SRBITS SRbits __attribute__((__sfr__));
```

Note:
Three bit field
Single bit fields

© 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 17
Unions



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.

Lab 17
Unions

Solution: Step 3

```
/*****************************************************************************
 STEP 3:   Declare a variable named AccB of type AccumulatorB
           Follow the instructions below to clear AccB, then assign MCHP to each
           byte of AccB in sequence.
           Set a breakpoint at the Nop(); and single step through while watching
           the effect of the updates in the Variables window
*****************************************************************************/

  AccumulatorB AccB;                    // *** Declare variable AccB of type AccumulatorB

  AccB.Acc32 = 0;                       // *** Clear AccB
  Nop();                                // set  breakpoint here

  AccB.AccByte.AccByteL  = 'M';         // *** Assign 'M' to AccB low byte
  AccB.AccByte.AccByteH  = 'C';         // *** Assign 'C' to AccB high byte
  AccB.AccByte.AccByteU  = 'H';         // *** Assign 'H' to AccB upper byte
  AccB.AccByte.AccByteUU = 'P';         // *** Assign 'P' to AccB upper upper byte
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

Lab 17
Conclusions

- Unions allow the same memory location to be viewed and manipulated as different data types
- They make it possible to store different variable types in the same memory location(s)

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP

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**

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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 day;  //day is a variable of type weekday
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

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;
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

Enumerations

How to Use an Enumeration Type Variable

If enumeration and variable have already been defined:

Syntax

```
varName = label_n;
```

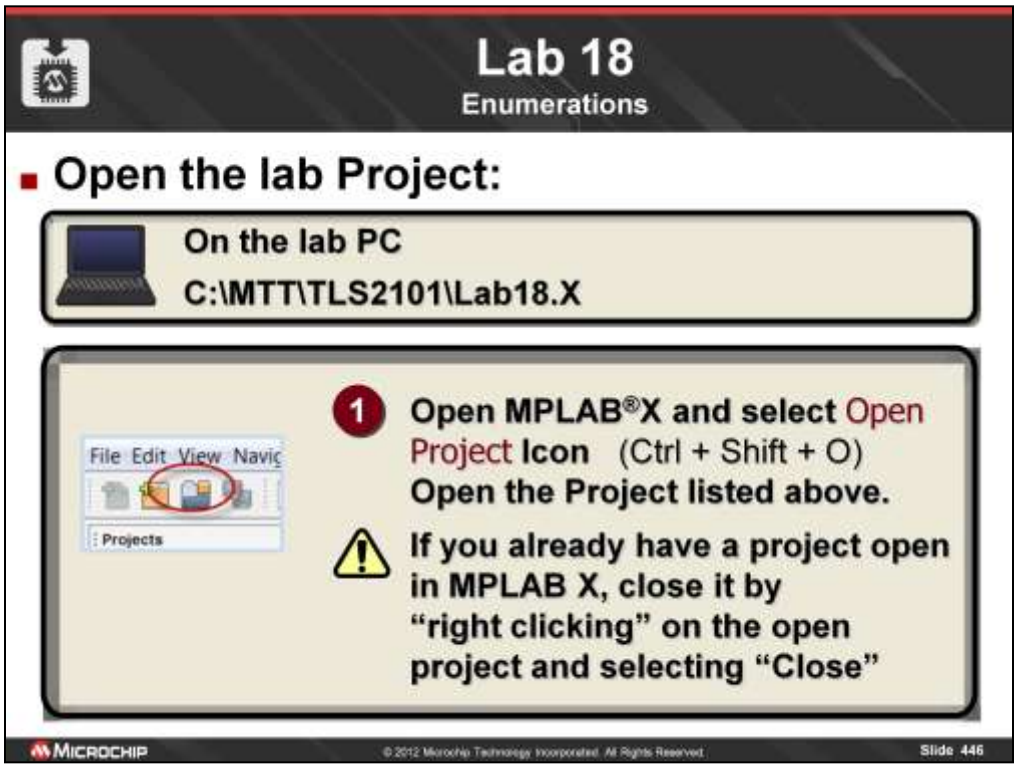
- The labels may be used as any other symbolic constant
- Variables defined as enumeration types must be used in conjunction with the type's labels or equivalent integer

Example

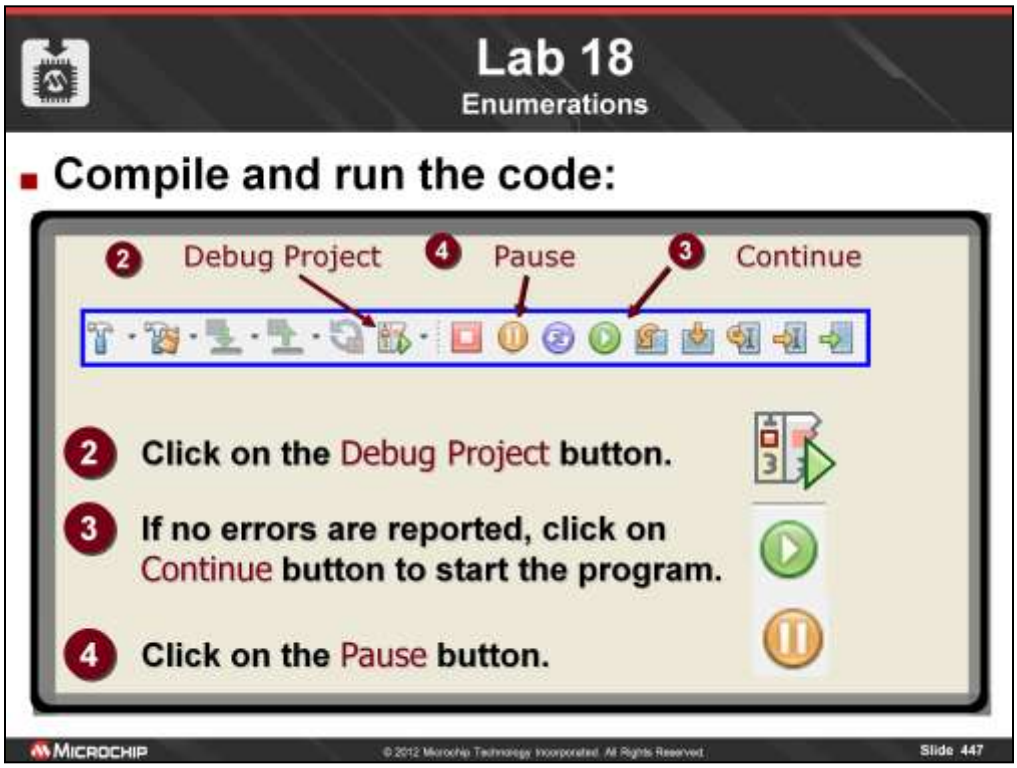
```
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT};
enum weekday day;

day = WED;
day = 6;                //May only use values from 0 to 6
if (day == WED)
{ …
```


© 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Lab Exercise 18
Enumerations



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.



Next, click on the Build All button to compile the code.
If you don't get any errors (you shouldn't), then go ahead and hit the run button.
After a second or two, hit the halt button, and then let's look at the code and what it did.

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
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**

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 451

HIDDEN SLIDE

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 (4 * (1/Fosc))
#define Setup InitSystem(Fosc, 250, 0x5A)
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 452

HIDDEN SLIDE

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; }
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved. Slide 453

HIDDEN SLIDE

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:

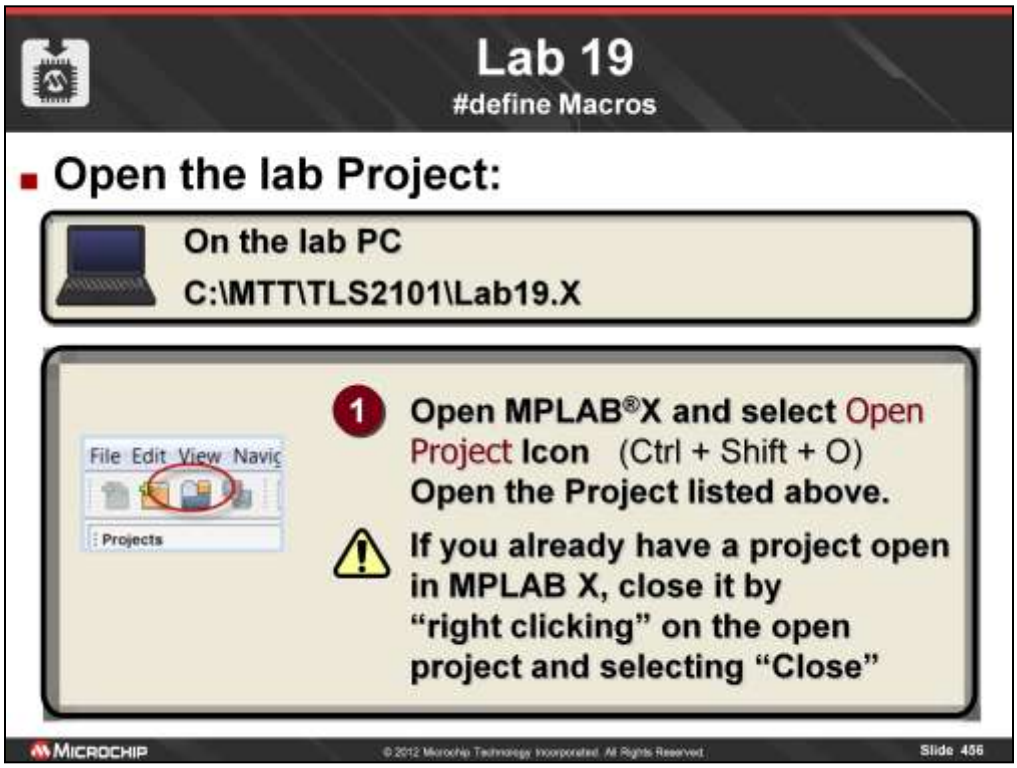
```
x = ((i++)*(i++));
```

So i gets incremented twice, not once at the end as expected.

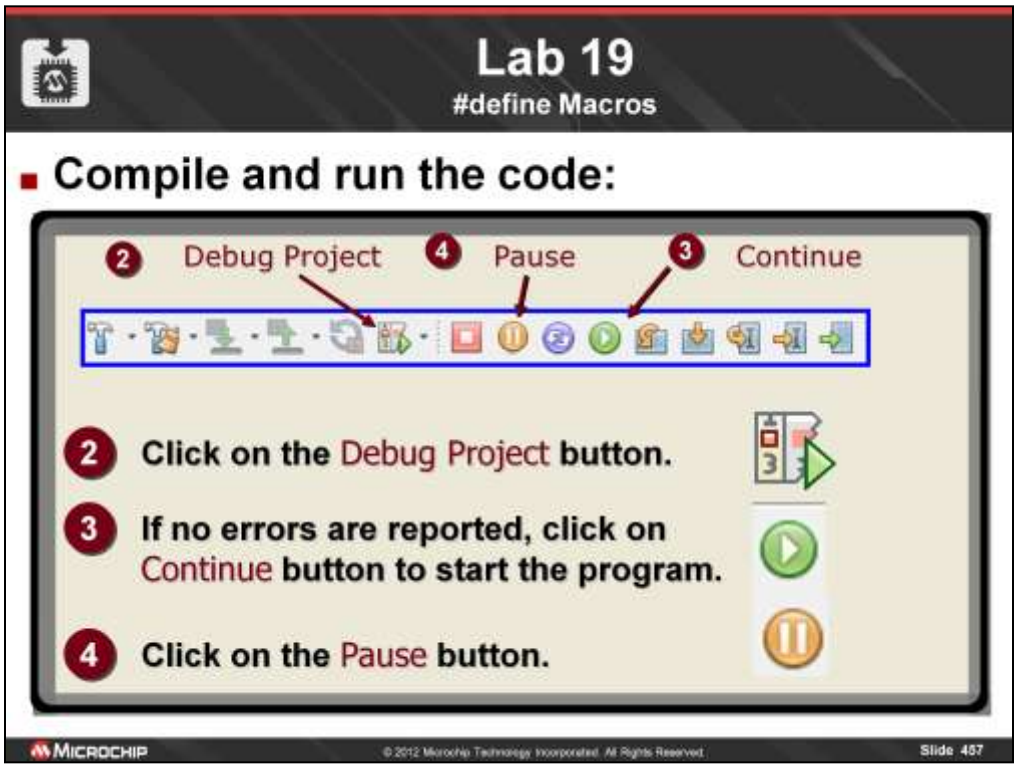
MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

HIDDEN SLIDE

MICROCHIP
Lab Exercise 19
Macros with #define



If you are not familiar with using MPLAB, simply follow the instructions in the handout, or follow along as I go through the demo. First, if you already have a project open, please close it by selecting File> Close Workspace. Now, we need to open the lab 1 workspace, so select Open Workspace… from the Fle menu and navigate to the class directory and workspace file shown in the box at the top of the slide.



Next, click on the Build All button to compile the code.
If you don't get any errors (you shouldn't), then go ahead and hit the run button.
After a second or two, hit the halt button, and then let's look at the code and what it did.

Lab 19

#define Macros

#define Macro Definition and Use

```
/*-----------------------------------------------------------------------
 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);
}
```

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

HIDDEN SLIDE

Lab 19
Conclusions

- #define macros can dramatically simplify your code and make it easier to maintain
- Extreme care must be taking when crafting a macro due to the way they are substituted within the text of your code

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

HIDDEN SLIDE

MICROCHIP

Summary

Summary

- During this class, we have covered:
  - Using C in an Embedded Environment
  - Variables, Identifiers and Data Types
  - Literal & Symbolic Constants
  - Symbolic Constants
  - printf() Library Function as debugging tool
  - Operators
  - Expressions and Statements
  - Decision Making and Looping

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Summary

- Loops
- Functions
- Multi-File Projects & Storage Class Specifiers
- Arrays
- Data Pointers (Data and Function)
- Function Pointers
- Structures, Unions & Bit Fields
- Enumerations
- Macros with #define based macros

MICROCHIP © 2012 Microchip Technology Incorporated. All Rights Reserved.

Resources

A Selection of C Compilers

- Microchip Technology XC8/XC16/XC32 compilers (Free 'student' versions available)
  http://www.microchip.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/

© 2012 Microchip Technology Incorporated. All Rights Reserved.

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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Resources

Books – General C Language

- **Programming Embedded Systems**
  with C and GNU Development Tools
  2nd Edition (October 1, 2006)
  Michael Barr & Anthony Massa
  ISBN-10: 0596009836
  ISBN-13: 978-0596009830

- **Practical C Programming**
  3rd Edition (August 1, 1997)
  Steve Oualline
  ISBN-10: 1565923065
  ISBN-13: 978-1565923065

- **Code Complete**
  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

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Resources

Books – PIC® 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

MICROCHIP

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Slide 466

Resources

Books – PIC® Specific

- **Beginner's Guide to Embedded C Cprogramming**
  Chuck Hellebucyk
  - Available from www.elproducts.com

- **Beginner's Guide (Volume 2)**
  Chuck Hellebucyk

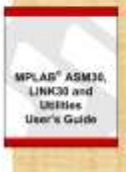
- **Beginner's Guide (Volume 3)**
  Chuck Hellebucyk

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Resources

Books – Compiler Specific

- **MPLAB® XC16 C Compiler User's Guide**
  Current Edition (PDF)
  Microchip Technology
  http://www.microchip.com



- **MPLAB® XC16 Link30 and Utilities User's Guide**
  Current Edition (PDF)
  Microchip Technology
  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® XC16 is based on the GCC tool chain

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Resources

Books – Compiler Specific

- **MPLAB® XC8 C Compiler User's Guide**
  Current Edition (PDF)
  Microchip Technology
  http://www.microchip.com

- **MPASM™ MPLINK™ and MPLIB™ User's Guide**
  Current Edition (PDF)
  Microchip Technology
  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.

© 2012 Microchip Technology Incorporated. All Rights Reserved.

Questions?

© 2012 Microchip Technology Incorporated. All Rights Reserved.

MICROCHIP
Thank You!

Trademarks

The Microchip name and logo, the Microchip logo, dsPIC, KeeLoq, KeeLoq logo, MPLAB, MPLAB X, PIC, PICmicro, PICSTART, PIC^{32} logo, rfPIC and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, Hampshire, HI-TECH C, Linear Active Thermistor, MXDEV, MXLAB, SEEVAL 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, HI-TIDE, In-Circuit Serial Programming, ICSP, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, Octopus, Omniscient Code Generation, PICC, PICC-18, PICDEM, PICDEM.net, PICkit, PICtail, REAL ICE, rfLAB, Select Mode, Total Endurance, TSHARC, UniWinDriver, 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.

© 2011, Microchip Technology Incorporated, All Rights Reserved.