#include Directive

- Three ways to use the `#include` directive:

## Syntax

```
#include <file.h>
Look for file in the compiler search path
The compiler search path usually includes the compiler's directory and all of
its subdirectories.
For example: C:\Program Files\Microchip\MPLABX\XC32\*. *
```

```
#include "file.h"
Look for file in project directory only
```

```
#include "c:\MyProject\file.h"
Use specific path to find include file
```
#include Directive

main.h Header File and main.c Source File

main.h

```c
#include <stddef.h>

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

main.c

```c
#include "main.h"

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

The contents of main.h are effectively pasted into main.c starting at the #include directive’s line.
#include Directive

Equivalent main.c File

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

```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
Header Guards
Duplicate #includes

The contents of main.h are effectively pasted twice into main.c starting at the #include directive’s line

```c
#include "main.h"
#include "main.h"

int main(void)
{
    a = 5;
    b = 2;
    c = a + b;
}
```
Header guards
Equivalent main.c File

- Duplicate declarations will occur.
- Which will give compilation errors as there cannot exist multiple declarations of the same variable in the same scope.

```
#include "main.h"

int main(void)
{
    unsigned int a;
    unsigned int b;
    unsigned int c;

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

Equivalent main.c file without #include
Header guards
Realistic example

#include "OledDriver.h"

OledDriver.h

Oled.h

main.c

#include "OledDriver.h"
#include "Oled.h"
Header guards
How do you write/use them

- Declare a macro when a header file is processed.
- Check for that macro before including the code.

```c
// Oled.h

#ifndef OLED_H
#define OLED_H

// code for header
...

#endif // OLED_H
```
Expressions

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

Examples

\[ a + b \]
\[ x = y \]
\[ \text{speed} = \text{dist} / \text{time} \]
\[ z = \text{ReadInput()} \]
\[ c \leq 7 \]
\[ x == 25 \]
\[ ++\text{count} \]
\[ d = a + 5 \]
Statements

• Cause an action to be carried out

• Three kinds of statements in C:
  – Expression Statements
  – Compound Statements
  – Control Statements
Expression Statements

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

Examples

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

Compound Statements

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

Also called Block Statements
Compound Statements

Example

```c
{ 
    float start, finish;
    start = 0.0;
    finish = 400.0;
    distance = finish - start;
    time = 55.2;
    speed = distance / time;
    printf("Speed = %f m/s", speed);
}
```
Control Statements

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

Example

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

(while syntax: `while expr statement`)
Boolean Expressions

• Boolean data type added in C99
• Boolean expressions return integers:
  – 0 expressions evaluate as false
  – non-zero expressions evaluate as true (generally 1)

```c
{ int x = 5;
  bool y, z;
  y = (x > 4); // y = true (1)
  z = (x > 6); // z = false (0)
  while (1);  // (1)
}
```
Boolean Expressions

Equivalent Expressions

• If a variable, constant, or function call is used alone as the conditional expression:
  
  \[(\text{myVar}) \text{ or } (\text{Foo}())\]

• This is the same as saying:

  \[(\text{myVar} \neq 0) \text{ or } (\text{Foo}() \neq 0)\]

• In either case, if myVar ≠ 0 or Foo() ≠ 0, then the expression evaluates as true (non-zero)
**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 Statement**

Flow Diagram

**Syntax**

```plaintext
if (expression) statement
```

Flowchart:

1. START
2. `expression`
   - `expression ≠ 0`
     - true
     - `expression = 0`
     - false
3. END
   - `statement`
if Statement

Example

```c
int x = 5;
if (x) {
    printf("x = %ld\n", x);
} while (1);
```

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

16-bit: 0
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)

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

Example: if (x == 1)

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

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

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

**Syntax**

```c
if (expression) statement_1
else statement_2
```

- `expression` is evaluated for boolean true (≠0) or false (=0)
- If true, then `statement_1` is executed
- If false, then `statement_2` is executed
if-else Statement

Syntax

\[
\text{if} \ (\text{expression}) \ \text{statement}_1 \\
\text{else} \ \text{statement}_2
\]
if-else Statement

```c
float frequency = 146.52;  // Frequency in MHz

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

Syntax

\[
\text{if } (\text{expression}_1) \text{ statement}_1 \\
\text{else if } (\text{expression}_2) \text{ statement}_2 \\
\text{else } \text{statement}_3
\]

- \( \text{expression}_1 \) is evaluated for boolean true (\( \neq 0 \)) or false (\( = 0 \))
- If true, then \( \text{statement}_1 \) is executed
- If false, then \( \text{expression}_2 \) is evaluated
- If true, then \( \text{statement}_2 \) is executed
- If false, then \( \text{statement}_3 \) is executed
if-else if Statement

Flow Diagram

Syntax

```plaintext
if (expression₁) statement₁
else if (expression₂) statement₂
else statement₃
```
if-else if Statement

Example

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

Syntax

```
while (expression) statement
```

- If `expression` is **true**, `statement` will be executed and then `expression` will be re-evaluated to determine whether or not to execute `statement` again.

- It is possible that `statement` will never execute if `expression` is false when it is first evaluated.
while Loop

Flow Diagram

Syntax

while (expression) statement

START

expression?

true

statement

false

END
**while Loop**

**Example**

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

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

**Expected Output:**

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

- Primary looping mechanism
- Completely generic
- Frequently used for main loop of program

**Generic loop:**

```c
while (HaveData()) {
    PrintData();
}
```

**Main loop:**

```c
while (1) {
    ...
}
```
Functions
Program Structure

main()
{
    ...
    eat();
    ...
    drink();
    ...
}

eat()
{
    ...
    return;
}

be_merry()
{
    ...
    return;
}

drink()
{
    ...
    be_merry();
    return;
}
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 return a single value (but can export more data)
- Functions help to organize a program into logical, manageable segments
Functions

Remember Algebra Class?

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

\[ f(x) = x^2 + 4x + 3 \]

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

Definitions

Syntax

Data type of return \textit{expression}

\begin{itemize}
  \item Name \texttt{Void}
  \item \texttt{Type} \texttt{Identifier} (\texttt{Type}_1 \texttt{Arg}_1, \ldots, \texttt{Type}_n \texttt{Arg}_n)
  \item Parameter List (optional)
  \item Body \texttt{Void}
  \item Return Value (optional)
\end{itemize}

\texttt{Return expression;}

\texttt{Return \texttt{X}+2;}

Functions

Function Definitions: Syntax Examples

Example

```c
int Maximum(int x, int y)
{
    int z;
    z = (x >= y) ? x : y;
    return z;
}
```

Example – A more efficient version

```c
int Maximum(int x, int y)
{
    return ((x >= y) ? x : y);
}
```
Functions

Function Definitions: Return Data Type

Syntax

\[ \text{int } \text{identifier(} \text{type}_1 \text{ arg}_1, \ldots, \text{type}_n \text{ arg}_n \text{)} \]

\{
  \text{declarations}
  \text{statements}
  \text{return expression;}
\}

- A function's \textit{type} must match the type of data in the return \textit{expression}
Functions

Function Definitions: Return Data Type

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

Example

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

```c
// Example usage
int x = 5, y = 10;
int result = bigger(x, y);
printf("The bigger number is \%d\n", result);
```
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

```c
void identifier(type1 arg1, ..., type n arg n)
{
    declarations
    statements
    return;  // may be omitted if nothing is being returned
}
```
Questions?
return a > b;

TRUE     FALSE     FALSE     FALSE
TRUE     FALSE     FALSE     TRUE     error
Success 0
FAILURE 1

return foo(bar());
Announcements

- Lab 2 is out
- X Drive issues
Functions
Declaration and Use: Example 1

Example 1

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

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

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

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

Declaration and Use: Example 2

Example 2

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

int Maximum(int x, int y); // Function is declared with prototype before use in main()

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

    // Function is defined after it is used in main()
    int Maximum(int x, int y)
    {
        return ((x >= y) ? x : y);
    }
}
```
printf()  
Standard Library Function

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

Standard Library Function

**Syntax**

```c
printf(ControlString, arg1, ..., argn);
```

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

**Example**

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

a = 5
b = 10
```

**NOTE:** The 'd' in `%d` is the conversion character. (See next slide for details.)
`printf()`

Gotchas

- The value displayed is interpreted entirely by the formatting string:
  ```c
  printf("ASCII = %d", 'a');
  ```
  will output: `ASCII = 97`

- A more problematic string:
  ```c
  printf("Value = %d", 6.02e23);
  ```
  will output: `Value = 26366`

- Incorrect results may be displayed if the format type doesn't match the actual data type of the argument.
`printf()` 
Output buffer

- `printf()` operates on lines of text.
- Output text may not be transmit until a newline is sent.

**Example**

```c
printf("a")
```

Output:
printf()

Output buffer

- printf() operates on lines of text.
- Output stored in a buffer until a newline triggers transmission.

Example

```
printf("a\n");
```

Output:

"a\n"
printf()  
The output buffer

stdio.h

output buffer

UART
printf()

Format specifiers

%[flags][width][.precision][size]type

• Flags – Special printing options
• Width – The minimum size (in chars) of the output
• Precision – Field width
• Size – Convert from base types to longer/shorter types
• Type – The base variable type
printf()  
Format specifiers

%[flags][width][.precision][size]type

- **Flags** – Special printing options
  - ‘-’ -> Left justify
  - ‘0’ -> Pad with zeros
  - ‘+’ -> Output ‘+’ for positive values
  - ‘ ‘ -> Don’t output a sign symbol
  - ‘#’ -> Prefix integer value based on output type
printf()

Format specifiers

%[flags][width][.precision][size]type

- **Width** – The minimum size (in chars) of the output
  - Output is padded
  - ‘0’ flag specifies padding with ‘0’ s instead of ‘ ‘ s
printf()  
Format specifiers  

%[flags][width][.precision][size]type  

- **Precision** – Field width  
  - For integers, minimum number of digits  
  - For floats, number of fractional digits/significant figures  
  - For strings, number of characters
printf()
Format specifiers

%[flags][width][.precision][size]type

- **Size** – Convert from base types to longer/shorter types
  - ‘h’ -> Converts to short
  - ‘l’ -> Converts to long/double
  - ‘ll’ -> Converts to long long/long double
# `printf()`

Format specifiers

<table>
<thead>
<tr>
<th>Conversion Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c</code></td>
<td>Single character</td>
</tr>
<tr>
<td><code>s</code></td>
<td>String (all characters until '\0')</td>
</tr>
<tr>
<td><code>d</code></td>
<td>Signed decimal integer</td>
</tr>
<tr>
<td><code>o</code></td>
<td>Unsigned octal integer</td>
</tr>
<tr>
<td><code>u</code></td>
<td>Unsigned decimal integer</td>
</tr>
<tr>
<td><code>x</code></td>
<td>Unsigned hexadecimal integer with lowercase digits (1a5e)</td>
</tr>
<tr>
<td><code>X</code></td>
<td>As <code>x</code>, but with uppercase digits (e.g. 1A5E)</td>
</tr>
<tr>
<td><code>f</code></td>
<td>Signed decimal value (floating point)</td>
</tr>
<tr>
<td><code>e/E</code></td>
<td>Signed decimal with exponent (e.g. 1.26e-5)</td>
</tr>
<tr>
<td><code>p</code></td>
<td>A pointer value indicating a memory address</td>
</tr>
<tr>
<td><code>g/G</code></td>
<td>As <code>e</code> or <code>f</code>, but depends on size and precision of value</td>
</tr>
<tr>
<td><code>g</code></td>
<td>Prints ‘%’</td>
</tr>
</tbody>
</table>
printf() Format String Examples

- Print a hexadecimal:

```c
printf("0x%06x\n", x);
```

0  Any unused spaces will be filled with zeros

6  Specifies that 6 characters must be output (including 0x prefix)

871587  0x0d4ca3
printf()  
Format String Examples  

- Printing a double:

```c
printf("f = 806.3f\n", f);
```

- Any unused spaces will be filled with zeros
- Specifies that 6 characters must be output
- Specifies that 3 decimal places will be output

3.3 → 03.300
printf()  
Format String Examples

- Printing a double:

```c
printf("%f\n", percentCorrect);
```

.1  Specifies that 1 decimal place will be output

%%  Outputs a literal ‘%’

97.322  
97.38
```c
printf("\8.1f\n", (double)percentCorrect);
```

- .1 Specifies that 1 decimal place will be output
- %% Outputs a literal ‘%’

97.322 \[\rightarrow\] 97.38

Maxwell James Dunne – Spring 2015
scanf()  
Standard Library Function  

- Used to read input from the "standard input"
- Normally a keyboard or file
- Often the UART in embedded systems
  - Input file in the simulator
  - Entire family of functions:
    - sscanf() reads from a string
    - fscanf() reads from a file
`scanf()`
Standard Library Function

**Syntax**

```c
int scanf(FormatString, arg1, ..., argn);
```

- The format string tells `scanf` what kind of input.
- `arg1` through `argn` are **POINTERS** to variable of the right type.

**Example**

```c
int a, b;
printf("Input a and b\n");
scanf("%d %d", &a, &b);
printf("a=%d\nb=%d", a, b);
```
### `scanf()` Gotchas

- Ignores blanks and tabs in format string
- Skips over white space (blanks, tabs, newline) as it looks for input
- Returns number of successful conversions
- Arguments **must** be pointers to variable types
- Arguments not processed in the input will be left in the input buffer.
`scanf()`

The input buffer

`stdio.h`

output buffer

UART

Input buffer

Maxwell James Dunne – Spring 2015
**scanf()**

Standard Library Function

**Example**

```c
int a, b;
printf("Input a and b\n\n");
scanf("%d %d", &a, &b);
printf("a=%d\nb=%d", a, b); ← "3140 56\n"
scanf("%d %d", &a, &b);
printf("a=%d\nb=%d", a, b); ← "77 -3\n"
```
\textbf{scanf()}

The input buffer

\begin{verbatim}
  3 1 4 0 5 6 \n
  scanf("\%d \%d", \&a, \&b)
\end{verbatim}

\begin{align*}
a &= 3140, \quad b &= 56
\end{align*}
`scanf()`

The input buffer

```
\n  7  7   -  3  \n\n```

```
scanf("%d %d", \&a, \&b)
```

Nothing!
`scanf()`

The input buffer

```
3 1 4 0 5 6 \n
Enter
```

```
scanf("%d %d%c", &a, &b, &c)
```

```
3 1 4 0 5 6 \n
a = 3140, b = 56
```
`scanf()`

Format specifiers

`%[*][width][modifier]type`

- * – Ignores this field
- Width – The maximum number of characters to match
- Modifier – Convert from base types to longer/shorter types
- Type – The base variable type
scanf()  
Examples  
• Read input line with date in the format:  
  – 25/12/2012  

```c
scanf("%d/%d/%d", &day, &month, &year);
```

- **day**: int, &day is pointer to day  
- **month**: int, &month is pointer to month  
- **year**: int, &year is pointer to year
scanf()

Examples

- Read input line with date in the format:
  - 25 Dec 2012

```
scanf("%d %s %d", &day, month, &year);
```

day int, &day is pointer to day

month char[20], is a string for putting the month into, does not need "&" because name of array is already a pointer

year int, &year is pointer to year
scanf()  

Return value

Example

```c
int a, b;
char c;
while (scanf("%d %d%c", &a, &b, &c) != 3) {
    printf("Please enter an integer pair!\n");
}
```

```
1
1 2
1 2
```

Maxwell James Dunne – Spring 2015
Arrays

Definition

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

Arrays:

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

How to Create an Array

Arrays are declared much like ordinary variables:

Syntax

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

- `size` refers to the number of elements
- `size` can be a constant OR specified at runtime (c99)

Example

```
int a[10];

char s[25];

char str[x];
```
Arrays
How to Initialize an Array at Declaration

Arrays may be initialized with a list when declared:

```
Syntax

type arrayName[size] = {item_1, ..., item_n};
```

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

Example

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

char b[5] = {'a', 'b', 'c', 'd', 'e'};
```
Arrays
How to Use an Array

Arrays are accessed like variables, but with an index:

**Syntax**

```
arrayName[index]
```

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

**Example**

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

for(i = 0; i < 10; i++) {
    a[i] = 0;  // Initialize all array elements to 0
}

a[4] = 42;  // Set fifth element to 42
```
Arrays
Creating Multidimensional Arrays

Add additional dimensions to an array declaration:

Syntax

type arrayName[size_1]...[size_n];

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

Example

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

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

Initializing Multidimensional Arrays at Declaration

Arrays may be initialized with lists within a list:

**Syntax**

```c
type arrayName[size0][size1][...][sizen] =
{
    {item, ..., item},
    ...
    {item, ..., item}
};
```

**Example**

```c
char a[3][3] = {
    {'X', '0', 'X'},
    {'0', '0', 'X'},
    {'X', 'X', '0'}
};

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

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

Row, Column

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

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

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

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

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

Visualizing 3-Dimensional Arrays

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

Plane, Row, Column

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

<table>
<thead>
<tr>
<th>Plane 0</th>
<th>Plane 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0][0][0] = 0;</td>
<td>a[1][0][0] = 4;</td>
</tr>
<tr>
<td>a[0][0][1] = 1;</td>
<td>a[1][0][1] = 5;</td>
</tr>
<tr>
<td>a[0][1][0] = 2;</td>
<td>a[1][1][0] = 6;</td>
</tr>
<tr>
<td>a[0][1][1] = 3;</td>
<td>a[1][1][1] = 7;</td>
</tr>
</tbody>
</table>

Maxwell James Dunne – Spring 2015
Arrays
Example of Array Processing

//navbar
* Print out 0 to 90 in increments of 10
/navbar
*/

int main(void)
{
    int i = 0;
    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    while (i < 10) {
        a[i] *= 10;
        printf("%d\n", a[i]);
        ++i;
    }
    while (1);
}
Strings
Character Arrays and Strings

Definition

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

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

Creating a String Character Array

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

**Syntax**

```c
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**

```c
char str1[10]; // Holds 9 characters plus '\0'
char str2[6]; // Holds 5 characters plus '\0'
```
Strings

How to Initialize a String at Declaration

Character arrays may be initialized with string literals:

```
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
char str3[] = {'P', 'I', 'C', '\0'};
```