CMPE-013/L

Expressions and Control

Maxwell James Dunne
Expressions

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

Example

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

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

(while syntax: `while expr statement`)

Maxwell James Dunne
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);
}
```
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 \(\text{myVar} \neq 0\) or \(\text{Foo}() \neq 0\), then the expression evaluates as true (non-zero)
if Statement

**Syntax**

```
if (expression) statement
```

- *expression* is evaluated for boolean true(\(\neq 0\)) 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

if (expression) statement

START

expression

expression ≠ 0

true

statement

false

expression = 0

END
if Statement

Example
```c
{  
    int x = 5;

    if (x) {  
        printf("x = %d\n", x);  
        ...then print the value of x.
    }
    while (1);
}
```

- What will print if \(x = 5\)? ... if \(x = 0\)?
- ...if \(x = -82\)?
- ...if \(x = 65536\)?

\[x \mod 16 \quad \overline{05535} + 1 = 0\]
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  cp0.w  0x08c2
  011B6  320004  bra z, 0x0011c0
```

Example: if (x == 1)

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

Example

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

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

**Syntax**

\[
\begin{align*}
\textbf{if} & \ (\text{expression}) \ \textbf{statement}_1 \\
\textbf{else} & \ \textbf{statement}_2
\end{align*}
\]

- **expression** is evaluated for boolean true (≠0) or false (=0)
- If true, then **statement}_1 is executed
- If false, then **statement}_2 is executed
if-else Statement

Flow Diagram

Syntax

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

Example

```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

```
if (expression_1) statement_1
else if (expression_2) statement_2
else statement_3
```

- $expression_1$ is evaluated for boolean true ($\neq 0$) or false ($=0$)
- If true, then $statement_1$ is executed
- If false, then $expression_2$ is evaluated
- If true, then $statement_2$ is executed
- If false, then $statement_3$ is executed
if-else if Statement

Flow Diagram

Syntax

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

```
while (1)
```

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

```
do while
```
While Loop

Flow Diagram

Syntax

while (expression) statement
**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++);
    i = i + 2;  // Loop counter incremented manually inside loop
}
```

**Expected Output:**

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

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

```c
Generic loop:
while (HaveData()) {
    PrintData();
}

Main loop:
(while (1) {
    ...
})
```
Functions

Program Structure

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

eat()
{
    ... 
    return; 
}

be_merry()
{
    ... 
    return; 
}

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: $f(7) = 7^2 + 4 \times 7 + 3 = 80$
Functions
Definitions

Syntax

Data type of
return \textit{expression}

\begin{array}{c}
\text{Type} \ \text{identifier} (\text{type}_1 \ \text{arg}_1, \ldots, \text{type}_n \ \text{arg}_n) \\
\{ \\
\text{declarations} \\
\text{statements} \\
\text{return} \ \textit{expression}; \\
\}
\end{array}

Parameter List (optional)

\texttt{divisor} \ \texttt{dividend}

Header

Return Value (optional)
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

\[
\text{type identifier} (\text{type}_1 \ \text{arg}_1, \ldots, \text{type}_n \ \text{arg}_n)
\{
\text{declarations}
\text{statements}
\text{return expression;}
\}
\]

- A function's \text{type} must match the type of data in the return \text{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;
    }
}
```

Check for errors
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

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

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

Function Definitions: Parameters

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

Syntax

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

Function Definitions: Parameters

• Parameter list may mix data types
  - `int Foo(int x, float y, char z)`

• Parameters of the same type must be declared separately – in other words:
  - `int Maximum(int x, y)` will not work
  - `int Maximum(int x, int y)` is correct

Example

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

Function Definitions: Parameters

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

Example

```
type identifier(void)
{
    declarations
    statements
    return expression;
}
```
Functions
How to Call / Invoke a Function

Function Call Syntax

- No parameters and no return value
  \[ \text{Foo}() \]
  \[ \text{if} \ (\text{Foo}()) \]

- No parameters, but with a return value
  \[ x = \text{Foo}() \]

- With parameters, but no return value
  \[ \text{Foo}(a, b) \]

- With parameters and a return value
  \[ x = \text{Foo}(a, b) \]
Functions

Function Prototypes

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

Function Prototypes (Declaration)

- Function prototypes may be take on two different formats:
  - An exact copy of the function header:
    
    ```
    int Maximum(int x, int y);
    ```

  - Like the function header, but without the parameter names – only the types need be present for each parameter (bad form!):
    
    ```
    int Maximum(int, int);
    ```
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);

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

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

Function is declared with prototype before use in main()

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

integrate
import regex

Stdio.h

.py compiled

regex
def

X
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
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 Uart1/2 window in MPLAB X
typedef
Standard Library Function

```
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\nb = %d\n", a, 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  

```c
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
$\text{hypot}$

$\text{hypot}(3, 4)$

$\text{float}\ x=3,\ y=4;$

$\text{hypot}(x, y);$
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 x, 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</code> / <code>%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</code> / <code>%G</code></td>
<td>As e or f, but depends on size and precision of value</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Prints ‘%’</td>
</tr>
</tbody>
</table>
printf()  
Format String Examples

• Print a hexadecimal:

\[ \text{printf} ("0x806x\n", x); \]

0  Any unused spaces will be filled with zeros
6  Specifies that 6 characters must be output (including 0x prefix)
printf()  
Format String Examples

• Printing a double:

```
printf("f = \%06.3f\n", f);
```

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

3.3 03.300
printf()  
Format String Examples

- Printing a double:

```c
printf("%.1f88\n", percentCorrect);
```

.1  Specifies that 1 decimal place will be output

%%  Outputs a literal ‘%’
printf()  
Format String Examples

- Printing a double:

```c
printf("%.1f\n", (double)percentCorrect);
```

.1 Specifies that 1 decimal place will be output

%% Outputs a literal ‘%’

97.322 97.38
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\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

Input buffer

UART

CMPE-013/L: “C” Programming

Maxwell James Dunne
Example

```c
int a, b;
printf("Input a and b\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"
```
`scanf()`

The input buffer

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

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

```
a = 3140, b = 56
```
**scanf()**

The input buffer

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

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

Nothing!
The input buffer

```
3 1 4 0
5 6
```

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

```
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
  - `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

```c
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
12
```
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**

```c
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

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

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

Initializing Multidimensional Arrays at Declaration

Arrays may be initialized with lists within a list:

**Syntax**

```
type arrayName[size_0][...][size_n] =
    {{item,...,item},
     ...
     {item,...,item}};
```

**Example**

```
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 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 |
```

```plaintext
Row, Column
```

```plaintext
Column
```

```
```

Maxwell James Dunne
Arrays
Visualizing 3-Dimensional Arrays

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

Plane, Row, Column

```
<table>
<thead>
<tr>
<th></th>
<th>z=0</th>
<th>z=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y=0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[0][0][0] = 0;</td>
<td>a[1][0][0] = 4;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a[0][0][1] = 1;</td>
<td>a[1][0][1] = 5;</td>
</tr>
<tr>
<td>y=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[0][1][0] = 2;</td>
<td>a[1][1][0] = 6;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a[0][1][1] = 3;</td>
<td>a[1][1][1] = 7;</td>
</tr>
</tbody>
</table>
```
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);
}