C: A High Level Programming Language

• **Gives symbolic names to values**
  – Don’t need to know which register or memory location

• **Provides abstraction of underlying hardware**
  – operations do not depend on instruction set
  – example: can write “a = b * c”, even though underlying hardware may not have a multiply instruction
C: A High Level Programming Language

- Provides expressiveness
  - use meaningful symbols that convey meaning
  - simple expressions for common control patterns (if-then-else)
- Enhances code readability
- Safeguards against bugs
  - can enforce rules or conditions at compile-time or run-time
Compilation vs. Interpretation

- Different ways of translating high-level language

- Interpretation
  - interpreter = program that executes program statements
  - generally one line/command at a time
  - limited processing
  - easy to debug, make changes, view intermediate results
  - languages: BASIC, LISP, Perl, Java, Matlab, Python

\[ PYC \]
Compilation vs. Interpretation

- **Compilation**
  - translates statements into machine language
  - does not execute, but creates executable program
  - performs optimization over multiple statements
  - change requires recompilation
    - can be harder to debug, since executed code may be different
  - languages: C, C++, Fortran, Pascal, Ada
Compilation vs. Interpretation

- Consider the following algorithm:
  
  Get W from the keyboard.
  
  \[
  \begin{align*}
  X &= W + W \\
  Y &= X + X \\
  Z &= Y + Y \\
  \end{align*}
  \]
  
  Print Z to screen.

- If interpreting, how many arithmetic operations occur? 3

- If compiling, we can analyze the entire program and possibly reduce the number of operations. Can we simplify the above algorithm to use a single arithmetic operation?

\[
\begin{align*}
  x &= 2w \\
  y &= 2x \\
  z &= 2y = 8w
\end{align*}
\]
Compilation

• **Source Code Analysis**
  - “front end”
  - parses programs to identify its pieces
  - variables, expressions, statements, functions, etc.
  - depends on language (not on target machine)

• **Code Generation**
  - “back end”
  - generates machine code from analyzed source
  - may optimize machine code to make it run more efficiently
  - very dependent on target machine

• **Symbol Table**
  - map between symbolic names and items
  - like assembler, but more kinds of information
“Hello World”

- The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:
  
  ```
  Print the words
  hello, world
  ```

- This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went.

- With these mechanical details mastered, everything else is comparatively easy.
```c
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n"); // Uses the I/O library to print
    return 0;
}

q = foo(x)
q = main(x)
```
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");
    while (1); // Loop forever and never return
}

foo: L foo
int main(void)
{
    while (1) {
        // Read inputs
        // Perform calculations
        // Update outputs
    }
}
Setting up the IDE
Configuring the Simulator

Set the Debug simulator to wait at the beginning of 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
Opening a Project

Select the Open Project button
Opening a Project

1) Navigate to the Project Directory
2) Select the Project
3) Select Open Project
Opening a Project

Project will Open in MPLAB X
To build the project and send it to the Debugger select the **Debug Project Button**.
Building a Project

- Control Buttons Appear
- Simulation ready to start
- Successful Build

`int main( void )
{
  int localVar1 = 3;
  int localVar2 = 4;
  printf("Welcome to the Embedded C Programming Class\n when using MPLAB IDE.
```
while(1);

BUILD SUCCESSFUL (total time: 542ms)
Loading C:\/MIT\/TSL2101\/Getting Started\/GettingStarted.X\dist/default/debug/GettingStarted.X.debug.xml
Loading completed

Maxwell James Dunne
CMPE-013/L: “C” Programming
Running the Simulation

To run the project, push the Continue button.
Pausing the Simulation

To pause execution of the simulation hit the Pause button.
Windows used in Examples

Variables Window

**Variable Window** displays a particular set of program variables.

To Open the Variables window:

Select:
Windows->Debugging->Variables
Windows used in Examples

Variables Window

Variable Window displays several columns of data

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>localVar1</td>
<td>int</td>
<td>0x88A</td>
<td>0x0003</td>
<td>3</td>
</tr>
<tr>
<td>localVar2</td>
<td>int</td>
<td>0x888</td>
<td>0x0004</td>
<td>4</td>
</tr>
</tbody>
</table>

You may find it convenient to alter the columns displayed.

“right click” on the column heading
Windows used in Examples

UART1 Output

**UART1 Output Window** prints out text from C programs

To clear this window:

Right click *inside* of the window then select **Clear**
Windows used in Examples

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

To Open the Watches Window:

Select:
Windows->Debugging->Watches
**Windows used in Examples**

**Watches Window** needs to be ‘told’ what data to watch.

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Decimal</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>globalVar1</td>
<td>int</td>
<td>0x850</td>
<td>1</td>
<td>0x0001</td>
</tr>
<tr>
<td>globalVar2</td>
<td>int</td>
<td>0x852</td>
<td>2</td>
<td>0x0002</td>
</tr>
</tbody>
</table>
```

“Right click” while in the Watches Window to add or delete watches.

** Column configuration is identical to Variables Window **
Closing a Project

1. Stop the simulation by pushing the Finish Debugger Session button.
2) "Right Click" on the project name then select **Close**
# Fundamentals of C

A Simple C Program

### Example

```c
#include <stdio.h>

#define PI 3.14159

int main(void)
{
    float radius, area;

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

Comments

Definition

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

- Two kinds of comments may be used:
  - Block Comment
    
    ```
    /* This is a comment */
    ```
  - Single Line Comment
    
    ```
    // This is also a comment
    ```
Quiz on Wednesday due to snafu with classrooms

No Class on Monday

MLK
Comments
Using Block Comments

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

```c
#include <stdio.h>

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

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

```c
#include <stdio.h>

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

• Block comments may not be nested within other delimited comments
• Single line comments may be nested

Example: Single line comment within a delimited comment.

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

Example: Delimited comment within a delimited comment.

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

Delimiter don't match up as intended!
Dangling delimiter causes compile error
/**
 * @file
 * @author R. Ostapiuk
 * @section DESCRIPTION
 * This is an example Hello World program
 */
#include <stdio.h>

/**
 * Main, the entrypoint for this C program.
 * @return A success code, where non-zero values indicate failure
 */
int main(void)
{
    int i;    // Loop counter variable
    char *p;  // Pointer to text string

    // Display greeting
    printf(“Hello, world!\n”);
}
Variables and Data Types
A Simple C Program

```c
#include <stdio.h>

#define PI 3.14159

int main(void)
{
    float radius, area;

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

Definition

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

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

```c
int myVariable;
myVariable = 5;
```
Variables

- Variables are names for storage locations in memory

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

- Variable declarations consist of a unique identifier (name)...

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

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

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

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

Example of Identifiers in a Program

```c
#include <stdio.h>

#define PI 3.14159

int main(void)
{
    float radius, area;

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

- Valid characters in identifiers:
  - First Character:
    - `_` (underscore)
    - `A` to `Z`
    - `a` to `z`
  - Remaining Characters:
    - `_` (underscore)
    - `A` to `Z`
    - `a` to `z`
    - `0` to `9`

- Case sensitive!

- Only first 31 characters significant*
Name mangling

400

400

31 character.

4MB
ANSI C Keywords

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

- Some compiler implementations may define additional keywords
  __asm ("NOP")

Maxwell James Dunne
## Data Types

### Fundamental Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>single character</td>
<td>8</td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
<td>16</td>
</tr>
<tr>
<td>float</td>
<td>single precision floating point number</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>double precision floating point number</td>
<td>64</td>
</tr>
</tbody>
</table>

The size of an `int` varies from compiler to compiler.
- XC16 `int` as 16-bits
- XC32 defines `int` as 32-bits

If you need precise length variable types, use stdint.h
- `uint8_t` is unsigned 8 bits
- `int16_t` is signed 16bits, etc.
# Data Type Qualifiers

## Modified Integer Types

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

<table>
<thead>
<tr>
<th>Qualified Type</th>
<th>Min</th>
<th>Max</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsigned char</code></td>
<td>0</td>
<td>255</td>
<td>8</td>
</tr>
<tr>
<td><code>char, signed char</code></td>
<td>-128</td>
<td>127</td>
<td>8</td>
</tr>
<tr>
<td><code>unsigned short int</code></td>
<td>0</td>
<td>65535</td>
<td>16</td>
</tr>
<tr>
<td><code>short int, signed short int</code></td>
<td>-32768</td>
<td>32767</td>
<td>16</td>
</tr>
<tr>
<td><code>unsigned int</code></td>
<td>0</td>
<td>65535</td>
<td>16</td>
</tr>
<tr>
<td><code>int, signed int</code></td>
<td>-32768</td>
<td>32767</td>
<td>16</td>
</tr>
<tr>
<td><code>unsigned long int</code></td>
<td>0</td>
<td>$2^{32}-1$</td>
<td>32</td>
</tr>
<tr>
<td><code>long int, signed long int</code></td>
<td>$-2^{31}$</td>
<td>$2^{31}-1$</td>
<td>32</td>
</tr>
<tr>
<td><code>unsigned long long int</code></td>
<td>0</td>
<td>$2^{64}-1$</td>
<td>64</td>
</tr>
<tr>
<td><code>long long int, signed long long int</code></td>
<td>$-2^{63}$</td>
<td>$2^{63}-1$</td>
<td>64</td>
</tr>
</tbody>
</table>
# Data Type Qualifiers

## Modified Floating Point Types

<table>
<thead>
<tr>
<th>Qualified Type</th>
<th>Absolute Min</th>
<th>Absolute Max</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>$\pm \sim 10^{-44.85}$</td>
<td>$\pm \sim 10^{38.53}$</td>
<td>32</td>
</tr>
<tr>
<td>double$^{(1)}$</td>
<td>$\pm \sim 10^{-44.85}$</td>
<td>$\pm \sim 10^{38.53}$</td>
<td>32</td>
</tr>
<tr>
<td>long double</td>
<td>$\pm \sim 10^{-323.3}$</td>
<td>$\pm \sim 10^{308.3}$</td>
<td>64</td>
</tr>
</tbody>
</table>

MPLAB-X XC32 Uses the IEEE-754 Floating Point Format

---

Maxwell James Dunne

CMPE-013/L: “C” Programming
Variables
How to Declare a Variable

**Syntax**

```
type identifier_1, identifier_2,...,identifier_n;
```

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

**Example**

```
int x, y, z;
float warpFactor;
char text_buffer[10];
unsigned index;
```
Variables
How to Declare a Variable

Variables may be declared in a few ways:

Syntax

One declaration on a line
```
type identifier;
```

One declaration on a line with an initial value
```
type identifier = InitialValue;
```

Multiple declarations of the same type on a line
```
type identifier₁, identifier₂, identifier₃;
```

Multiple declarations of the same type on a line with initial values
```
type identifier₁ = Value₁, identifier₂ = Value₂;
```
Variables
How to Declare a Variable

Examples

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

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

How to Declare a Variable

- Sometimes, variables (and other program elements) are declared in a separate file called a **header file**
- Header file names customarily end in `.h`

- Header files are associated with a program through the `#include` directive
#include Directive

- Three ways to use the #include directive:

```
#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\XC16\*.*

#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

```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

main.h

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

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

main.c

```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.

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

int main(void)
{
    ...
}

Equivalent main.c file without #include
```
Header guards

Realistic example

```c
#include "OledDriver.h"

#include "OledDriver.h"
#include "Oled.h"

main.c
```
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
#ifdef OLED_H
#define OLED_H

#include "OledDriver.h"

...

#endif // OLED_H
```
Questions?
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

\[ a + b \]
\[ x = y \]
\[ \text{speed} = \text{dist} / \text{time} \]
\[ z = \text{ReadInput}() \]
\[ c \leq 7 \]
\[ x = 25 \]
\[ \text{++count} \]
\[ d = a + 5 \]
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++;  // Corrected
a = 5 + i;
y = (m * x) + b;
printf("Slope = %f", m);
;  // Corrected
```

Maxwell James Dunne
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);
```

Maxwell James Dunne
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!");
    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);
```
Boolean Expressions

Equivalent Expressions

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

• This is the same as saying:
  \((myVar \neq 0) \text{ or } (\text{Foo}() \neq 0)\)

• In either case, if \(myVar \neq 0 \text{ or } \text{Foo}() \neq 0\), 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
```

Flow Diagram:
- **START**
- **expression**
  - true
    - **statement**
  - false
    - **expression = 0**
- **END**
**if Statement**

Example

```c
{
    int x = 5;
    if (x) {
        printf("x = %d\n", x);  // If x is TRUE (non-zero)...
    }
    while (1);
}
```

- What will print if `x = 5`?  ... if `x = 0`?
- ...if `x = -82`?
- ...if `x = 65536`?
if Statement
Testing for TRUE

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

Example: if (x)

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

Example: if (x == 1)

```
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

```c
#include <stdio.h>

int main() {
    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");
            }
        }
    }

    return 0;
}
```

if-else Statement

Syntax

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

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

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

- $expression₁$ is evaluated for boolean true ($\neq 0$) or false ($=0$)
- If true, then $statement₁$ is executed
- If false, then $expression₂$ is evaluated
- If true, then $statement₂$ is executed
- If false, then $statement₃$ is executed
if-else if Statement

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

```plaintext
while (expression) statement
```

![Flow Diagram]

```plaintext
while(1);
```
**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:

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

Main loop:

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

Definitions

Syntax

Data type of

return expression

Name

definition (type_1 arg_1, ..., type_n arg_n)

Parameter List
(optional)

Body

declarations

statements

return expression;

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

```c
type identifier(type1 arg1, ..., type_n arg_n)
{
    declarations
    statements
    return expression;
}
```

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

Function Definitions: Return Data Type

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

Example

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

Function Definitions: Return Data Type

- The function type is **void** if:
  - The **return** statement has no **expression**
  - The **return** statement is not present at all

- This is sometimes called a **procedure function** since nothing is returned

---

**Example**

```c
void identifier(type_1 arg_1, ..., type_n arg_n)
{
    declarations
    statements
    return;           // return; may be omitted if nothing is being returned
}
```
320

340
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

```c

typedef identifier (type₁ arg₁,...,typeₙ argₙ)
{
    declarations       Function Parameters
    statements
    return expression;
}
```

Maxwell James Dunne
Functions

Function Definitions: Parameters

• Parameter list may mix data types
  \[-\text{int } \text{Foo}(\text{int } x, \text{ float } y, \text{ char } z)\]

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

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

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

Function Call Syntax

- No parameters and no return value
  \[ \text{Foo()}; \]
- 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
Functions
Function Prototypes

- 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

```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()
void foo();

#include "foo.h"

main.c

#include <foo.h>

void fooie() { }
fooie.c

fooie.o

main.c

main.o

fooie()