CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Unit testing
Unit testing

- Testing portions of code in isolation
- Normally testing is per function
- Requires input and expected output to be known a priori

\[3 + 4 = 7\]
Unit testing

Rationale

• Find problems early
  – Before integration
• Simplify testing by only testing small, segmented portions of code
• Test functionality that may not be exposed otherwise corner cases
• Find documentation errors
Unit testing
Preparing

• The most important question:

"How am I going to test this?"

• Break code into clean functions with:
  – Clear input
  – Clear output
  – No/minimal side effects
Unit testing
Testing architecture

- Known input
  - f()
  - Actual output

- Expected output

- Output matches
  - yes: Print success
  - no: Print failure
Unit testing

Testing architecture

Example

```c
// Declare test constants
testInput ← some input
testExpOutput ← precalculated output

// Calculate result
testActOutput ← function result

// Output test results
if testActOutput equals testExpOutput
  output "Test passed"
else
  output "Test failed!"
```
# Unit testing

## Trivial example

**ExampleLib.c**

```c
int AddFive(int x) {
    return x + 5;
}
```

**main.c**

```c
#include "ExampleLib.h"

int main(void)
{
    // Declare test constants
    int test1Input = 0;
    int test1ExpOutput = 5;

    // Calculate result
    int test1ActOutput;
    test1ActOutput = AddFive(test1Input);

    // Output test results
    if (test1ActOutput == test1ExpOutput) {
        printf("Test1 passed. \n");
    } else {
        printf("Test1 failed! \n");
    }
}
```
Unit testing
Writing tests

- Write multiple tests
  - At least 1 for every group of inputs
  - Each edge case should have their own test

- Each test should check one part of the total functionality
  - One function or logical block of code at a time

Try to break the code you're testing!
Unit testing

Testing framework

• Track how many tests passed/failed
  – Per function

• Track how many functions passed/failed
  – With all tests must pass for the function to pass

• Each test cleanly separated from other tests
  – Both in code and in logic

• Output results
  – Per function/per test results
MatrixEquals(A, B)
return True;

2 tests

True
False

# define ASIZE 3
Parameter passing

Pass by value
Pass by reference
Parameter Passing
By Value

- Parameters passed to a function are generally 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 the function has a duplicate of the variable, not the original
Parameter Passing
By Value

Example

```c
int a, b, c;

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

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

The value of \(a\) is copied into \(x\).
The value of \(b\) is copied into \(y\).
The function does not change the value of \(a\) or \(b\).
Parameter Passing

By Value

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
Parameter Passing
By Reference

- Parameters can be passed to a function by reference
- Entails passing around memory address
- The original variable that is passed to a function can be modified by the function since the function knows where the data "lives" in memory
Parameter Passing

By Reference

Example function

```c
int Foo(int x[3])
{
    int z = x[2];
    x[1] = 0;
    return z;
}
```

Example main

```c
int main(void)
{
    int a[3] = {6, 19, -1};
    Foo(a);
    while (1);
}
```
Scope
Scope

Variables Declared Within a Function

- Variables declared within a code block are local to that block.

Example

```c
int x, y, z;

int Foo(int n)
{
    int a;
    ...
    a += n;
}
```

The `n` refers to the function parameter `n`

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

Variables Declared Within a Function

- Variables declared within a block are not accessible outside that block

Example

```c
int x;
int Foo(int n)
{
    int a;
    return (a += n);
}
int main(void)
{
    x = Foo(5);  // This will generate an error. a may not be accessed outside of the scope where it was declared.
    x = a;
}
```
Scope
Variables Declared Within a Function

- Variables declared within a block are not accessible outside that block

Example

```c
int x;
int main(void)
{
    {
        int a = 6;  // This will generate an error. a may not be accessed outside of the scope where it was declared.
    }

    x = Foo(5);
    x = a;
}
```
Scope
And the stack

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
**Scope**

And the stack

---

**Example function**

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

---

**Example main**

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
Scope
Global versus Local Variables

Example

```c
int x = 5;
int z;
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`
# Scope

**Parameters**

- "Overloading" variable names:

<table>
<thead>
<tr>
<th>Declared Locally and Globally</th>
<th>Declared Globally Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>int n;</td>
<td>int n;</td>
</tr>
<tr>
<td>int Foo(int n)</td>
<td>int Foo(int x)</td>
</tr>
<tr>
<td>`{</td>
<td>`{</td>
</tr>
<tr>
<td>... y += n;</td>
<td>... y += n;</td>
</tr>
<tr>
<td>... local n hides</td>
<td>...</td>
</tr>
<tr>
<td>global n</td>
<td></td>
</tr>
</tbody>
</table>

A locally defined identifier takes precedence
Scope
Parameters

Example

```c
int n;

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

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

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

Example

```c
#define x 2

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

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

Result:

```
5
5
```
Storage Class Specifiers
Scope and Lifetime of Variables

- Scope and lifetime of a variable depends on its storage class:
  - Automatic Variables
  - Static Variables
  - External Variables
  - Register Variables

- Scope refers to where in a program a variable may be accessed
- Lifetime refers to how long a variable will exist or retain its value
Storage Class Specifiers
Automatic Variables

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

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

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

**auto** Keyword with Variables

```c
int Foo(auto int x, auto int y)
{
    ...
}
```

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

Static Variables

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

```c
int x; // Global variable is always static

int main(void)
{
    ...
```
Storage Class Specifiers

**Static int**

- A variable declared as **static** inside a function retains its value between function calls (not destroyed when exiting function).
- Function parameters cannot be **static** with some compilers (XC32).

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

*a will remember its value from the last time the function was called.*

If given an initial value, it is only initialized when first created – not during each function call.
Storage Class Specifiers

External Variables

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

**External Variable Declaration Syntax:**

```
extern type identifier;
```

**External Variable Declaration Example:**

```
extern int x;
```
Storage Class Specifiers

External Variables

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

Example:

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

int a;
```
Storage Class Specifiers

External Variables

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

Register Variables

- `register` variables are placed in a processor's "hardware registers" for higher speed access than with external RAM
  - Common with loop counters
- Not as important when RAM is integrated into processor package (microcontrollers, ...)
- May be done with PIC®/dsPIC®, but it is architecture/compiler specific...
Storage Class Specifiers

Scope of Functions

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

External Functions

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

```
Main.c

int foo(void);

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

SomeFileInProject.c

int foo(void)
{
    ...
}
```
Storage Class Specifiers

Static Functions

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

```
Main.c

int foo(void);

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

SomeFileInProject.c

static int foo(void)
{
    ...
}
```
If a variable is declared as `static`, it will only be available within the file where it was declared.

```c
extern int myVar;

int main(void)
{
    ...  
    myVar = 6;
}
```

```c
static int myVar = 0;
```
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Literals & Constants

Literals

Constants
A Simple C Program

Literal Constants

Example

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

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

**Definition**

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

**Literals**

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

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

- Four basic types of literals:
  - Integer
  - Floating Point
  - Character
  - String

- Integer and Floating Point are *numeric type* constants:
  - Commas and spaces are not allowed
  - Value cannot exceed type bounds
  - May be preceded by a plus or minus sign
Integer Literals

Decimal (Base 10)

- Cannot start with 0 (except for 0 itself)
- Cannot include a decimal point
- Valid Decimal Integers:
  
  0  5  127  -1021  65535

- Invalid Decimal Integers:
  
  32,767  25.0  1,024  0552
Integer Literals

Hexadecimal (Base 16)

- Must begin with \texttt{0x} or \texttt{0X}
- May include digits 0-9 and A-F / a-f
- Valid Hexadecimal Integers:

\texttt{0x \ 0X1 \ 0xA2B \ BEEF}

- Invalid Hexadecimal Integers:

\texttt{0x5.3 \ EA12 \ EG \ 53h}
Integer Literals
Octal (Base 8)

• Must begin with \( \_0 \_ \)
• Only include digits \( 0-7 \)
• Valid Octal Integers:
  \( 0 \) \( 01 \) \( 012 \) \( 073125 \)
• Invalid Octal Integers:
  \( 05.3 \) \( 0012 \) \( 080 \) \( 530 \)
Integer Literals

Binary (Base 2)

- Must begin with \texttt{0b} or \texttt{0B}
- May include digits 0 and 1
- Valid Binary Integers:
  \begin{align*}
  \text{0b} & \quad \text{0b1} & \quad \text{0b01010011000001111} \\
  \end{align*}
- Invalid Binary Integers:
  \begin{align*}
  \text{0b1.0} & \quad \text{01100} & \quad \text{0b12} & \quad \text{10b}
  \end{align*}

\textbf{ANSI C does not specify a format for binary integer literals. However, this notation is supported by almost all compilers.}
Integer Literals

Qualifiers

- Like variables, literals may be qualified
- A suffix is used to specify the modifier
  - ‘U’ or ‘u’ for unsigned: 25u
  - ‘L’ or ‘l’ for long: 25L
  - 'F' or 'f' for float: 10f or 10.25F
- Suffixes may be combined: 0xF5UL
  - Note: U must precede L
Integer Literals

Unqualified

- Numbers without a suffix are assumed to be signed int
- Automatic promotion based on constant type/suffix:

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal</th>
<th>Octal/Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>
## Integer Literals

**Unqualified**

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal</th>
<th>Octal/Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>u/U</td>
<td>unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>I/L</td>
<td>long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>L/L</td>
<td>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
</tbody>
</table>

Floating Point Literals

Decimal (Base 10)

• Like integer literals, but the decimal point is allowed
• Exponential notation can be used: 
\( k \times 10^{\pm n} \)
• Valid Floating Point Literals:
  2.56e-5  10.4378  48e8  0.5  10f
• Invalid Floating Point Literals:
  0x5Ae-2  02.41  F2.33
Character Literals

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

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

Declarations

- Strings are a special case of arrays
- The null character is automatically appended to the end of the string:

Example 1 – Wrong Way

```c
char color[3] = "RED";
// Is stored as:
// color[0] = 'R'
// color[1] = 'E'
// color[2] = 'D'
```

**NOT** a complete string – no '\0' at end

Example 2 – Right Way

```c
char color[] = "RED";
// Is stored as:
// color[0] = 'R'
// color[1] = 'E'
// color[2] = 'D'
// color[3] = '\0'
```
String Literals
How to Include Special Characters in Strings

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\000</td>
<td>Octal number</td>
</tr>
<tr>
<td>\xhh</td>
<td>Hexadecimal</td>
</tr>
</tbody>
</table>

- Equivalent characters
  - '\0', '\x0'
  - '\41', '\x21', '!'
  - '\144', '\x64', 'd'
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 value
  – Make development changes easy
  – Eliminate the use of "magic numbers"
  – Two types of constants
    • Text Substitution Labels
    • \texttt{const} Variables
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
- Requires no memory

**Example**

```
define PI 3.14159
#define mol 6.02E23
#define MCU "PIC32MX320F128H"
#define COEF 2 * PI
```
Symbolic Constants
#define Gotchas

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

**Example**

```
#define MyConst 5;

c = MyConst + 3;

c = 5 + 3;
```
Symbolic Constants

Constant Variables Using `const`

- Declaring constants can be done with `const`:

  ```c
  const float pi = 3.141593;
  ```

- This variable is allocated in program memory, but it cannot be changed due to the `const` keyword.

- In the majority of cases, it is better to use `#define` for constants.
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

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

int variable1 = CONSTANT1;
int variable2;
// Cannot do: int variable2 = CONSTANT2
```
CMPE-013/L

Structs

Maxwell James Dunne
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 a group of related variables to be treated as a single unit, even if different types
  - Ease the organization of complicated data
Structures
Declaring

Syntax

```c
struct StructName {
    type1 memberName1;
    ...
    typen memberNameN;
};
```

Members are declared just like ordinary variables

Example

```c
// Structure to handle complex numbers
struct Complex {
    float re;    // Real part
    float im;    // Imaginary part
};
```
# Structures

## Instantiating

**Syntax**

```
struct StructName {
    type_1 memberName_1;
    ...
    type_n memberName_n;
} varName_1, ..., varName_n;
```

**Example**

```
// Structure to handle complex numbers
struct Complex {
    float re;     // x.re = 3
    float im;
} x, y;        // Declare x and y of type complex
```
If `StructName` has already been defined:

```c
struct StructName varName1, ..., varNameN;
```

Example:

```c
struct Complex {
    float re;
    float im;
}
...
struct Complex x, y; // Declare x and y of type complex
```
Structures

Accessing members

Syntax

```
structVariableName.memberName
```

Example

```c
struct Complex {
    float re;
    float im;
} x, y; // Declare x and y of type `struct complex`

int main(void)
{
    x.re = 1.25; // Initialize real part of x
    x.im = 2.50; // Initialize imaginary part of x
    y = x; // Set struct y equal to struct x
    ...
}
```
Structures
Initialization

Syntax
If $\text{StructName}$ has already been defined:

```c
struct StructName varName = \{const_1, \ldots, const_n\};
```

Example
```c
struct Complex {
    float re;
    float im;
};

... re  im

struct Complex x = \{1.25, 2.50\};
```
Structures

Nesting Structures

Example

```c
struct point {
    float x;
    float y;
};

struct line {
    struct point a;
    struct point b;
};

struct circle {
    struct point a;
    float R;
};

int main(void)
{
    struct line m = {{1.2, 7.6}, {38.5, 17.8}};
    ...
}
```

Maxwell James Dunne
Structures
Nesting Structures

Example

```c
struct point {
    float x;
    float y;
};

struct line {
    struct point a;
    struct point b;
};

int main(void)
{
    struct line m = {{1.2, 7.6}, {38.5, 17.8}};
    printf("Line (%f, %f) <-> (%f, %f)",
            m.a.x, m.a.y, m.b.x, m.b.y);
    ...
}
```
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
```c
struct Strings {
    char a[4];
    char *b;
} str = {"Bad", "Good"};
```

Example: Initializing Members
```c
int main(void)
{
    struct Strings str;
    str.a[0] = 'B';
    str.a[1] = 'a';
    str.a[2] = 'd';
    str.a[3] = '\0';
    str.b = "Good";
}
```
Structures
Creating Arrays of Structures

Syntax

If `StructName` has already been defined:

```c
struct  StructName  arrName[n];
```

Example

```c
struct  Complex  {
    float  re;
    float  im;
};
...

struct  Complex  a[3];
```
Structures

Initializing Arrays of Structures at Declaration

Syntax

If StructName has already been defined:

```c
struct StructName arrName[n] = {{list_1}, ..., {list_n}};
```

Example

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex a[3] = {{1.2, 2.5}, {3.9, 6.5}, {7.1, 8.4}};
```
Structures
Using Arrays of Structures

If `arrName` has already been defined:

**Syntax**

```
arrName[n].memberName
```

**Example: Definitions**

```c
typedef struct {
    float re;
    float im;
} Complex;
...
struct Complex a[3];
```

**Example: Usage**

```c
int main(void)
{
    a[0].re = 1.25;
    a[0].im = 2.50;
    ...
}
```
Structures
Creating a Pointer to a Structure

Syntax

If `StructName` has already been defined:

```c
struct StructName *ptrName;
```

Example

```c
struct Complex {
   float re;
   float im;
};
...
struct Complex *a;
```
Structures
How to Use a Pointer to Access Structure Members

If `ptrName` has already been defined:

**Syntax**

`ptrName->memberName`

- Pointer must first be initialized to point to the address of the structure itself: `ptrName = &structVariable;`

**Example: Definitions**

```
struct Complex {  
    float re;  
    float im;  
};  
...  
struct Complex x;  
struct Complex *p;
```

**Example: Usage**

```
int main(void)
{
    p = &x;  
    // Set x.re = 1.25 via p  
    p->re = 1.25;  
    // Set x.im = 2.50 via p  
    p->im = 2.50;
}
```
Structures
How to Pass Structures to Functions

Example

typedef struct {
    float re;
    float im;
} Complex;

void Display(struct Complex x)
{
    printf("(%f + j%f)\n", x.re, x.im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(a);
    Display(b);
}
typedef struct {
    float re;
    float im;
} Complex;

void Display(struct Complex *x)
{
    printf("(%f + j%f)\n", x->re, x->im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(&a);
    Display(&b);
}
typedef struct {
    float re;
    float im;
} Complex;

void Display(const struct Complex *x)
{
    printf("(%f + j%f)\n", x->re, x->im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(&a);
    Display(&b);
}
typedef
typedef

- Assign new names to existing datatypes
- Interpreted by the compiler (unlike `#define`)

**Syntax**

```c
typedef datatype typeName;
```

- `typedef int Length;`
- `typedef float single;`
typedef

How to Create a Structure Type with `typedef`

**Syntax**

```c
typedef struct StructTag_{optional} {
    type_1 memberName_1;
    ...
    type_n memberName_n;
} TypeName;
```

**Example**

```c
// Structure type to handle complex numbers
typedef struct {
    float re;    // Real part
    float im;    // Imaginary part
} Complex;

Complex x;
```
typedef

Declaring structs

Example

```
struct Complex {
    float re;
    float im;
};

struct {
    float re;
    float im;
} Complex;

struct Complex {
    float re;
    float im;
} Complex;

typedef struct {
    float re;
    float im;
} Complex;
```
typedef

Declaring structs

Example

typedef struct Complex {
    float re;
    float im;
} Complex;
Text I/O
Text I/O

• Within `<stdio.h>`:
  - Formatted text: `scanf()`/`printf()`
  - Characters: `getchar()`/`putchar()`
  - Strings/Lines: `fgets()`/`puts()`
  • NEVER EVER EVER USE `gets()`
Text I/O
fgets()

Syntax
char *fgets(char *str, int count, FILE *stream);

- str is where received data is stored
  - Needs to be an array
- count is how many characters to process
  - Stops when \n or (count-1) chars are received
- stream is stdin
Text I/O
fgets() example

Example

```c
#include <stdio.h>

int main(void)
{
    // Create enough memory for a 50 char string
    char inputData[50 + 1];

    fgets(inputData, sizeof(inputData), stdin);
}
```
String Processing
String Processing

- Within `<string.h>`:
  - Examination
    - Length: `strlen()`
    - Comparing: `strcmp()`/`strncmp()`
    - Splitting: `strtok()`
  - Manipulation
    - Copying: `strncpy()` *(Don't use `strcpy()`!)*
    - Appending: `strncat()`
String Processing

`strlen()`

**Syntax**

```c
size_t strlen(const char *str);
```

- `str` is the string to calculate the length of
- `size_t` can be treated as an `int`

**Examples**

```c
int x = strlen("My string"); // x = 9

char str[] = "asdf";
int y = strlen(str); // y = 4
```
String Processing

\textbf{strcmp()}

**Syntax**

\begin{verbatim}
int strcmp(const char *s1, const char *s2);
\end{verbatim}

- Ignores size of the strings, purely alphabetical comparison
- Return value is > 0 if \(s_1\) alphabetically before \(s_2\), 0 if they're equal, < 0 if \(s_2\) alphabetically before \(s_1\)

**Examples**

```c
char *s1 = "apple", *s2 = "zed";
int cmpResult = strcmp(s1, s2);
if (cmpResult > 0) {
    printf("apple > zed\n");
} else if (cmpResult == 0) {
    printf("apple == zed\n");
} else {
    printf("apple < zed\n");
}
```
# String Processing

## `strtok()`

### Syntax

```
char *strtok(char *s1, const char *s2);
```

- `s1` (input/output), string to be tokenized
  - Will be modified!
- `s2` (input) – Delimiters

### Examples

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " "); // firstToken = "This"
char *secondToken = strtok(NULL, " "); // secondToken = "is"
char *thirdToken = strtok(NULL, " "); // thirdToken = "an"
char *fourthToken = strtok(NULL, " "); // fourthToken = "example!"
```
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
```

```
This is an example!
```

```
s1
```

```
| T | h | i | s | i | s | a | n | e | x | a | m | p | l | e | ! | \0 |
```
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
```

First token:
```
This is an example!
```

Array s1:
```
This is \0is an example! \0
```

Pointer firstToken:
```
This is \0is an example! \0
```
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
```

```
firstToken
s1

T h i s \ 0 i s \ 0 a n \ 0 e x a m p l e \ 0
```

secondToken
String Processing
strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
char *thirdToken = strtok(NULL, " ");
```

```
This is \0 is \0 an \0 example \0
```

firstToken
s1
thirdToken

secondToken
String Processing

strtok() Details

Example

char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
char *thirdToken = strtok(NULL, " ");
char *fourthToken = strtok(NULL, " ");

↑

firstToken
s1
thirdToken

This is \0 is \0 an \0 example !\0

↑

secondToken

↑

fourthToken
String Processing

strncpy()

Syntax

```c
char *strncpy(char *s1, const char *s2, size_t n);
```

- **s1** (output) – where the string will be copied to
- **s2** (input) - the string that to be copied
- **n** - how many characters can be copied
- Undefined if s1 and s2 overlap!

Examples

```c
char s1[50];
strncpy(s1, "asdf", 4); // s1 = "asdf\0"
strncpy(s1 + strlen(s1), "asdf", 4); // s1 = "asdfasdf\0"
```
String Processing

strncat()

Syntax

char *strncat(char *s1, const char *s2, size_t n);

- **s1** (input/output) - is the base string
- **s2** (input) - the string that will be appended
- **n** - how many characters can be appended
- Undefined if s1 and s2 overlap!

Examples

```c
char s1[50] = "This is an example!";
strncat(s1, "asdf", 4);
```
String Processing

• Within `<stdlib.h>`:
  – Conversion
    • Integer: `atoi()`, `xtoi()`
    • Floats: `atof()`

• Within `<stdio.h>`:
  – Conversion
    • Any: `sscanf()`
String Processing

atof()

Syntax

double atof(const char *s);

- \textbf{s} (input) – The string to parse
- Returns the converted value or 0.0

Examples

```c
char s1[] = "1.03";
double x = atof(s); // y = 1.03

char s2[] = "efg";
double y = atof(s); // y = 0.0
```
// (-3.3) (--) +
len ↑

4 + 9.11 + 3.14 / *

c char * token = strdup("in");

} token = strtol(NULL) 3

}
Pointers

Pointers and memory
Pointer/array equivalency
Pointer arithmetic
Pointers and the stack
Pointers and strings
Arrays of pointers
Pointers
Address versus value

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

```
int x;
```

Value of variable \( x = 0x0123 \)

Address of variable \( x = 0x0804 \)
Pointers

What are pointers?

- A pointer holds the address of another variable or function
Pointers
What do they do?

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

Why would I want to do that?

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

```c
// 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++;
}
```
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)

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08BC</td>
</tr>
<tr>
<td>0x08C0</td>
</tr>
<tr>
<td>0x08C4</td>
</tr>
<tr>
<td>0x08C8</td>
</tr>
<tr>
<td>0x08CC</td>
</tr>
<tr>
<td>0x08D0</td>
</tr>
<tr>
<td>0x08D4</td>
</tr>
<tr>
<td>0x08D8</td>
</tr>
</tbody>
</table>

0000 0123
0000 4567
0000 89AB
0000 CDEF
0000 1357
0000 9BDF
0000 0246
0000 8ACE
Pointers
Where else are they used?

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

Syntax

\[ \text{type} \ \ast \text{ptrName}; \]

- In the context of a declaration, the \( \ast \) merely indicates that the variable is a pointer
- \text{type} is the type of data the pointer may point to
- Pointer usually described as “a pointer to \text{type}”

Example

\begin{verbatim}
int *iPtr;       // Create a pointer to int
int *iPtr, x;   // Create a pointer to int and an int
float *fPtr1, *fPtr2;  // Create 2 float pointers
\end{verbatim}