CMPE-013/L

Structs

Maxwell James Dunne
Tan

round 0.5 of 3.000

3.00011
**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

```
struct StructName {
  typeₙ memberNameₙ;
  ...
  type₁ memberName₁;
};
```

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

```c
struct StructName {
    type1 memberName1;
    ...
    typen memberName_n;
} varName1, ..., varName_n;
```

Example

```c
// Structure to handle complex numbers
struct Complex {
    float re;
    float im;
} x, y; // Declare x and y of type complex
```
Structures
Instantiating cont'd

Syntax

If `StructName` has already been defined:

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

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 \textbf{StructName} has already been defined:

\begin{verbatim}
struct StructName varName = \{ \texttt{const}_1, \ldots, \texttt{const}_n \};
\end{verbatim}

Example

\begin{verbatim}
struct Complex {
    float re;
    float im;
};

... 

struct Complex x = \{1.25, 2.50\};
\end{verbatim}
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}};
    ...
}
```
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**

```c
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**

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

**Example: Usage**

```c
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

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

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);
}
```
# Structures

## How to Pass Structures to Functions

### Example

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

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);
}
```
# Structures

How to Pass Structures to Functions

## Example

```c
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`)
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

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

struct Complex {
    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;
Introduction to “C” Programming

Maxwell James Dunne
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()`
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
#include <stdio.h>

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

    fgets(inputData, sizeof(inputData), stdin);
}

O  str
String Processing
String Processing

• Within `<string.h>`:
  – Examination
    • Length: `strlen()`
    • Comparing: `strcmp()`/`strncpy()`
    • 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

`strcmp()`

**Syntax**

```c
int strcmp(const char *s1, const char *s2);
```

- Ignores size of the strings, purely alphabetical comparison
- Return value is > 0 if `s1` alphabetically before `s2`, 0 if they're equal, < 0 if `s2` alphabetically before `s1`

**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

```c
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!
```

```
This is an example!
```
String Processing

strtok() Details

Example

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

```text
firstToken
s1
```

```
This is an example!
```

Maxwell James Dunne
String Processing

strtok() Details

Example

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

firstToken
```
This is \0 is \0 an example ! \0
```

s1

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, " ");
```

firstToken
s1
thirdToken

secondToken

This is \0 is \0 an \0 e x a m p l e ! \0
String Processing
strtok() Details

Example

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

```
firstToken
s1
This is an example!
thirdToken
```

```
secondToken
fourthToken
```

Maxwell James Dunne
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**

```c
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**

```c
double atof(const char *s);
```

- **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
```
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...

```c
int x;
```

Variable stored at Address

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)

Direct Access via \( x \)
\[
\text{x} = 0x0123; \quad \text{x}
\]

Indirect Access via \( \text{*p} \)
\[
\text{*p} = 0x0123; \quad \text{p}
\]
Pointers

Why would I want to do that?

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

Example: Data Buffer

```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)

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08BC</td>
<td>0123</td>
</tr>
<tr>
<td>0x08C0</td>
<td>4567</td>
</tr>
<tr>
<td>0x08C4</td>
<td>89AB</td>
</tr>
<tr>
<td>0x08C8</td>
<td>CDEF</td>
</tr>
<tr>
<td>0x08CC</td>
<td>1357</td>
</tr>
<tr>
<td>0x08D0</td>
<td>9BDF</td>
</tr>
<tr>
<td>0x08D4</td>
<td>0246</td>
</tr>
<tr>
<td>0x08D8</td>
<td>8ACE</td>
</tr>
</tbody>
</table>
Pointers
Where else are they used?

- Provide method to pass arguments \textit{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

```c
type *ptrName;
```

- In the context of a declaration, the * merely indicates that the variable is a pointer
- `type` is the type of data the pointer may point to
- Pointer usually described as “a pointer to `type`”

Example

```c
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
```