CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Dynamic Memory

malloc()
free()
Dynamic Memory

Memory leaks

- If pointers returned by `malloc()` are lost, that memory is then "lost"
- Easy to do because this may not crash your program, possibly only causing errors over long periods of time

Example

```c
void MyFunc(void)
{
    Complex *x = malloc(sizeof(Complex));
    ...
}
```
Dynamic Memory

Memory leaks

- So for every pointer obtained from `malloc()`, there should be an equivalent `free()` at some point

```c
void MyFunc(void)
{
    Complex *x = malloc(sizeof(Complex));
    ...
    free(x);
}
```
Dynamic Memory

When to use the Heap

- For **unknown amounts of data** rare
  - Arrays are always fixed-length at compile time
- When data needs to be accessible outside of the scope it was created in
  - Pointers need to be passed around
Pointers

Pointers to pointers
Pointers

Pointers to pointers

• Since pointers can point to any valid datatype, they can also point to other pointers

• No limit on levels of indirection
Pointers

Pointers to pointers

Example

```c
{  
    int x = 6;  
    int *y = &x;  
    int **z = &y;  
    printf("%d\n", **z);  
}
```

Output

6
Pointers
Passing by reference, again

• Passing by reference only allows persistently changing the value 1 level of indirection from the pointer and further
  – If a pointer is passed to a function, the data it points to can be altered
  – If a pointer-to-a-pointer is passed, the pointer it points to and the data that pointer points to can be altered
Pointers
Passing by reference, again

Example interrupt

```c
void MyFunc(int *x)
{
    *x = 6;
}

int main(void)
{
    int myInt;
    int *myIntPtr = &myInt;
    MyFunc(&myIntPtr);
}
```
Pointers
Passing by reference, again

Example interrupt

```c
void MyFunc(int **x)
{
    *x = malloc(sizeof(int));
    if (*x) {
        **x = 6;
    }
}

int main(void)
{
    int *myInt;
    MyFunc(&myInt);
}
```
Enums
Enumerations

Definition

Enumerations are integer data types that you can create with a limited range of values. Each value is represented by a symbolic constant that may be used in conjunction with variables of the same enumerated type.

- Enumerations:
  - Are unique integer data types
  - May only contain a specified list of values
  - Values are specified as symbolic constants
Enumerations
How to Create an Enumeration Type

• Creates an ordered list of constants
• If unspecified, each label’s value is one greater than the previous label

Syntax

```c
enum typeName {label_0, label_1, ..., label_n}
```

Where compiler sets \( label_0 = 0 \), \( label_1 = 1 \), \( label_n = n \)

Example

```c
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT};
```

Label Values:

\[
\begin{align*}
SUN &= 0, & MON &= 1, & TUE &= 2, & WED &= 3, & THR &= 4, & FRI &= 5, & SAT &= 6
\end{align*}
\]
Enumerations
How to Create an Enumeration Type

- Any label may be assigned a specific value
- The following labels will increment from that value

Syntax

```c
enum typeName {label_0 = const_0, ..., label_n}
```
Where compiler sets `label_0 = const_0, label_1 = (const_0 + 1), ...`

Example

```c
enum people {Rob, Steve, Paul = 7, Bill, Gary};
```

Label Values:

- Rob = 0, Steve = 1, Paul = 7, Bill = 8, Gary = 9
Enumerations
How to Create an Enumeration Type

• Any label may be assigned a specific value
• The following labels will increment from that value

Syntax

```
enum typeName {label_0 = const_0, ..., label_n}
```

Where compiler sets label_0 = const_0, label_1 = (const_0 + 1), ...

Example

```
enum people {Rob = 'a', Steve, Paul, Bill, Gary};
```

Label Values:
Rob = 'a', Steve = 'b', Paul = 'c', Bill = 'd', Gary = 'e'
Enumerations
How to Create an Enumeration Type

- Any label may be assigned a specific value
- The following labels will increment from that value

**Syntax**

```c
enum typeName {label₀ = const₀, ..., labelₙ}
```

Where compiler sets `label₀ = const₀, label₁ = (const₀ + 1), ...`

**Example**

```c
enum people {Rob = -4, Steve, Paul, Bill, Gary};
```

Label Values:

Rob = -4, Steve = -3, Paul = -2, Bill = -1, Gary = 0
Enumerations

How to Declare an Enumeration Type Variable

• Declared along with type:

```c
enum typeName {const-list} varname1,...;
```

• Declared independently:

```c
enum typeName varName1,...,varName_n;
```

Example

```c
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT} today;
enum weekday day; // day is a variable of type weekday
```
Enumerations
How to Declare a ‘Tagless’ Enumeration Variable

- No type name specified:

Syntax

```
enum {const-list} varName_1, ..., varName_n;
```

- Only variables specified as part of the `enum` declaration may be of that type

- No type name is available to declare additional variables of the `enum` type later in code

Example

```
enum {SUN, MON, TUE, WED, THR, FRI, SAT} Today;
```
Enumerations

How to Declare an Enumeration Type with `typedef`

- Variables may be declared as type `typeName` without needing the `enum` keyword

**Syntax**

```
typedef enum {const-list} typeName;
```

- The enumeration may now be used as an ordinary data type (compatible with `int`)

**Example**

```
typedef enum {SUN, MON, TUE, WED, THR, FRI, SAT} Weekday;

Weekday day;       // Variable of type weekday
```
Enumerations

How to Use an Enumeration Type Variable

If enumeration and variable have already been defined:

**Syntax**

```c
varName = label_n;
```

- The labels may be used as any other symbolic constant
- Variables defined as enumeration types must be used in conjunction with the type’s labels or equivalent integer

**Example**

```c
enum weekday {SUN, MON, TUE, WED, THR, FRI, SAT};
enum weekday day;

day = WED;
day = 6;  // May only use values from 0 to 6
if (day == WED) {
    ...
```
Enumerations

Proper formatting

Example

typedef enum {
    SUN, // sunday
    MON,
    TUE,
    WED,
    THR,
    FRI,
    SAT
} Weekday;

Weekday day = WED;
Enumerations

Proper formatting

Example

typedef enum {
   SUN,
   MON,
   TUE,
   WED,
   THR,
   FRI,
   SAT
} Weekday;

Weekday day = 3; // No compilation warning/error
Enumerations
Datatype usage

Example

typedef enum {
    SUN,
    MON,
    TUE,
    WED,
    THR,
    FRI,
    SAT
} Weekday;

void PrintDayName(Weekday d)
{
    if (d == SUN) {
        printf("Sun\n");
    } ...
}

PrintDayName(WED); // No compilation warning/error
Enumerations

Why enumerations?

- Enumerations are a proper datatype as well as the possible values for them
- Some compile-time checking
- Doesn't do text replacement, done during the compiler stage
- Use for a group of related values
Interrupts
Interrupts

- High-priority alerts that an event requires immediate attention
- Generally interrupts can be assigned priorities
- Event is handled by an Interrupt Service Routine (ISR)

7 priority level
3 subpriority
Interrupts

- ISR is a special function that is written by the developer, but called directly by the processor
- ISRs have no inputs or outputs
  - All processing through global/module-level variables
- ISRs are written a specific way and the processor is told they have been implemented by the compiler/developer
Interrupts

Traps

• Software interrupts are generally referred to as exceptions or traps

• Examples:
  – Division by zero
  – Invalid address dereference
  – Debugging breakpoint
  – Stack overflow
Example interrupt

```c
void _ISR IsrName(void)
{
    // Process data from the interrupt
    // Store results in global/module variable
    // Clear interrupt flag
}
```
void _ISR Uart1TxInterrupt(void)
{
    // Stall until transmission finishes
    while (!U1STAbits.TRMT);

    // Continue transmitting next batch of data
    Uart1StartTransmission();

    // Clear interrupt flag
    IFS0bits.U1TXIF = 0;
}
Interrupts

Calling

Example program

```c
int main(void)
{
    int x = 20;
    int y;
    y = x / 2;
}
```

Interrupt: UART1 Post-transmission

```c
void _ISR_U1TXInt(void)
{
    IFS0bits.U1TXIF = 0;
}
```
# Interrupts

## Calling

### Example program

```c
int main(void)
{
    int x = 20;
    int y;
    _U1TXInt();
    y = x / 2;
}
```

### Interrupt: UART1 Post-transmission

```c
void _ISR _U1TXInt(void)
{
    IFS0bits.U1TXIF = 0;
}
```
Interrupts

- Interrupts are important events that happen in real-time
- ISRs are the functions that handle these events
- ISRs are called outside of regular program execution order
Hardware Peripherals

Digital pins
Timers
ADC
Hardware Peripherals

- Communications
  - Serial
  - SPI
  - I²C

- Pin change notification

- DMA
  - Direct Memory Access

- Output compare
  - PWM

- Input capture
  - Timing
  - 1 ns
  - 1.25 ns

- Digital pins
  - 40

- Timers

- ADC
  - 300
Hardware Peripherals

Special function registers

- Peripherals are controlled by hardware registers
  - Referred to as Special Function Registers (SFRs)
- Memory-mapped unsigned 16-bit integers
- Accessible as global variables
  - Included from the `<xc.h>` header

PIC24

m.5s4k
Hardware Peripherals

Special function registers

- Declaration of Interrupt FlagS 0 register
- **volatile** qualifier indicates value can change outside of the code in this program
- **__attribute__** is a compiler directive to specify additional compiler parameters
  - **__sfr__** indicates that it's a memory-mapped SFR

SFR: IFS0

```c
extern volatile unsigned int IFS0 __attribute__((__sfr__));
```
Hardware Peripherals

Digital pins

- **Voltage**
  - High: 3.3V
  - Low: 0

- **Direction**
  - Input: 0 - 3.3V
  - Output: 0, 1

- **Polling interface**
Hardware Peripherals

Digital pins

- TRIS – TRIState register. Sets pin direction.
  - Pin is an output when corresponding bit is 0, input when corresponding bit is 1
- LAT – LATch register. Sets pin value/gets pin's desired value
  - Desired output value of the pin
- PORT – PORT register. Sets pin value/gets pin's actual value
  - Actual value of the pin
Hardware Peripherals

Digital pins

Dedicated Port Module

RD TRISx

WR TRISx

WR LATx
WR PORTx
RD LATx
RD PORTx

I/O Cell

I/O pin

Synchronization
Hardware Peripherals

Digital pins

Dedicated Port Module

I/O Cell

I/O pin

Synchronization

RD TRISx

WR TRISx

WR LATx

RD LATx

WR PORTx

RD PORTx

D

Q

CK

EN

Q

ODCx

TRISx

LATx
Hardware Peripherals

Digital pins
Hardware Peripherals

Timers

• Multiple 16-bit timers
  – 5 total
• Interrupt-based
  – ISR is called every X seconds
• Configurable periodicity
  – Range from 20MHz to 305Hz
Hardware Peripherals

Timer SFRs

- **TMRx** – Timer counter
  - uint16
  - Ticks every instruction clock cycle (20MHz)
- **PRx** – Timer x prescaler
  - Limit for when to trigger the timer interrupt.
  - Valid values are [1, INT16_MAX]
  - 0 is a special value, disables peripheral.
Hardware Peripherals

Timers

- To modify timer interrupt period, set PRx register.
- To set a period of the timer interrupt:
  - $20\text{MHz} / PRx = \text{periodicity}$
- PRx of 20000 $\rightarrow$ 1kHz interrupts
Hardware Peripherals

Timers
Hardware Peripherals

Timers

65535

TMRx
Hardware Peripherals

Timers

PRx

TMRx

event
Hardware Peripherals

Timers

PRx → CPU

event
Hardware Peripherals

Timers

PRx → CPU

Interrupt()

event
Hardware Peripherals

ADC

- Analog to Digital Converter
- Measures the voltage of a processor pin
- Used to read analog sensors
  - Temperature
  - Power
  - Battery levels
Hardware Peripherals

ADC SFRs

- ADCxBUFy: Buffer for holding samples
  - x is the ADC
  - y is the sample [0, 7]
  - 16-bit unsigned value
    - Only lowest 10-bits matter
Hardware Peripherals

ADC

- The input signal is continuously sampled
- Every $8^{th}$ sample triggers an interrupt
Hardware Peripherals

ADC

- Voltage range from $V_{ref^-}$ to $V_{ref^+}$
  - 0V to 3.3V
- Values are unsigned 10-bits, from [0, 1023]
- Units are in $V_{ref} / 1023 = 0.0032V$
Hardware Peripherals

ADC
Hardware Peripherals

ADC

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC1BUF0</td>
<td>2</td>
</tr>
<tr>
<td>ADC1BUF1</td>
<td>146</td>
</tr>
<tr>
<td>ADC1BUF2</td>
<td>288</td>
</tr>
<tr>
<td>ADC1BUF3</td>
<td>420</td>
</tr>
<tr>
<td>ADC1BUF4</td>
<td>563</td>
</tr>
<tr>
<td>ADC1BUF5</td>
<td>691</td>
</tr>
<tr>
<td>ADC1BUF6</td>
<td>829</td>
</tr>
<tr>
<td>ADC1BUF7</td>
<td>987</td>
</tr>
</tbody>
</table>
Hardware Peripherals

ADC

- ADC1BUF0 = 950
- ADC1BUF1 = 600
- ADC1BUF2 = 100
- ADC1BUF3 = 65
- ADC1BUF4 = 81
- ADC1BUF5 = 93
- ADC1BUF6 = 107
- ADC1BUF7 = 122
Event-driven Programming

Events
Event loop
Event-driven Programming

- Real-time programming paradigm
- Build around the concept of events
- Events are then handled by specific event handlers
- Works well with systems with multiple inputs that need to be handled in a timely manner
  - Real-time system
- Integrates well with interrupts
Event-driven Programming

Events

- Any temporally-short sensor occurrence
- Usually the derivative of a signal
  - Button was pressed down
  - The mouse was clicked
  - This sensor value changed
  - This interrupt triggered
Event-driven Programming

The event loop

• A continual loop that checks for and processes events
• The core of an event-driven program
Event-driven Programming

The event loop

```c
{
  while (1) {
    // Check for events
    // Process events
  }
}
```
Event-driven Programming

The event loop

```c
{
    while (1) {
        // Check for event 1
        // Check for event 2
        ...
        // Check for event n

        // Process event 1
        // Process event 2
        ...
        // Process event n
    }
}
```
Event-driven Programming

Event priorities

```c
{
    while (1) {
        // Check for event 3
        // Process event 3

        // Check for event 1
        // Process event 1

        // Check for event 2
        // Process event 2
    }
}
```
{ 
    while (1) { 
        if (buttonsEvent) { 
            // Update fixed LED mask 
        } 
        if (adcEvent) { 
            // Update OLED 
        } 
        if (timerEvent) { 
            // Update bouncing LED mask 
        } 
        if (ledEvent) { 
            // Update LEDs 
        } 
    } 
}
static uint8_t buttonsEvent;

void main()
{
    while (1) {
        if (buttonsEvent) {
            // Event loop
        }
    }
}

void _ISR Timer1Int(void)
{
    buttonsEvent = ButtonsCheckEvents();
    IFS0 &= ~(1 << 3);
}
CMPE-013/L

Linked Lists

Maxwell James Dunne
typedef struct ListItem {
    struct ListItem *previousItem;
    struct ListItem *nextItem;
    char *data;
} ListItem;
Linked List

ListItem *LinkedListNew(char *data);
Linked List

ListItem *LinkedListCreateAfter(ListItem *item, char *data);
LinkedList

ListItem *LinkedListCreateAfter(ListItem *item, char *data);

I → N → p = G;

I → N

I → N = G.
Linked List

char *LinkedListRemove(ListItem *item);
Linked List

```c
char *LinkedListRemove(ListItem *item);
```

\[ I \rightarrow p \rightarrow N = I \rightarrow N; \]

\[ I \rightarrow N \rightarrow p = I \rightarrow p; \]
Linked List

char *LinkedListRemove(ListItem *item);

\[
\begin{array}{c}
\text{I} \rightarrow \text{P} \\
\text{N} = 0;
\end{array}
\]

free();
Linked List

```c
char *LinkedListRemove(ListItem *item);
```

```
temp = item->data;
item->next = p = 0;
free(item);
```
Linked List

char *LinkedListRemove(ListItem *item);

\[ \Rightarrow I \Rightarrow p \Rightarrow N = I \Rightarrow N; \]
\[ \Rightarrow I \Rightarrow N \Rightarrow p = I \Rightarrow p; \]
Linked List

`ListItem *LinkedListGetFirst(ListItem *list);`

Loop while `L->p != 0`

```
3
```

`L = L->p;
if (L = 0)
3
Linked List

int LinkedListSize(ListItem *list);

int LinkedListPrint(ListItem *list);

L = LLGetFirst(L);

Loop while L != 0

L = L->N;

count++; print
Heap size = 0
It won't work
reset
Sort

null strings

strcmp strlen

-> 0

strcmp 0 +
cat cow
strcpy("cat", "cow")
cat bird
↑
2ebra