Dynamic Memory

malloc()
free()
00000
↑
6
One line method
while (\_\_\_?)
8 + 4 ? \(
\begin{array}{c}
8 + 4 = 12 \\
12 / 7 = 1.71 \\
1.71
\end{array}
\)
14 + 64 - x 8

1 + 4 = 5

6 - 4 = 2

5 * 2 = 10

10 / 8 = 1.25
struct stack X;
stack = stack * 4;
stack = init(x);
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));
    ...

    free(x);  // Free the allocated memory
}
```
Dynamic Memory

Memory leaks

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

Example

```c
void MyFunc(void)
{
    Complex *x = malloc(sizeof(Complex));
    ...
    free(x);
}
```
Dynamic Memory
When to use the Heap

- For unknown amounts of data
  - 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

Over bound 60 180
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);
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3F50</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x3F54</td>
<td>0000 0006</td>
</tr>
<tr>
<td>0x3F58</td>
<td>0000 3F54</td>
</tr>
<tr>
<td>0x3F5C</td>
<td>0000 3F58</td>
</tr>
<tr>
<td>0x3F60</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x3F64</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x3F68</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x3F6C</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

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)
{
    struct
    *
    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:

```
SUN = 0, MON = 1, TUE = 2, WED = 3, THR = 4, FRI = 5, SAT = 6
```
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 { label_0 = const_0, ..., label_n }
```

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

**Example**

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

Label Values:

<table>
<thead>
<tr>
<th>Label</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rob</td>
<td>-4</td>
</tr>
<tr>
<td>Steve</td>
<td>-3</td>
</tr>
<tr>
<td>Paul</td>
<td>-2</td>
</tr>
<tr>
<td>Bill</td>
<td>-1</td>
</tr>
<tr>
<td>Gary</td>
<td>0</td>
</tr>
</tbody>
</table>
Enumerations

How to Declare an Enumeration Type Variable

- Declared along with type:

  Syntax

  ```
  enum typeName {const-list} varname\_1,\ldots;
  ```

- Declared independently:

  Syntax

  ```
  enum typeName varName\_1,\ldots,\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:

```
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

```
enum {SUN, MON, TUE, WED, THR, FRI, SAT} Today;
```
Enumerations
How to Declare an Enumeration Type with \texttt{typedef}

- Variables may be declared as type \texttt{typeName} without needing the \texttt{enum} keyword

\begin{verbatim}
typedef enum \{const-list\} typeName;
\end{verbatim}

- The enumeration may now be used as an ordinary data type (compatible with \texttt{int})

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

Weekday day;  // Variable of type weekday
\end{verbatim}
Enumerations
How to Use an Enumeration Type Variable

If enumeration and variable have already been defined:

**Syntax**

```
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, // comments
    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)
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

Null Pointer
Example interrupt

```c
void ISR IsrName(void)
{
    // Process data from the interrupt
    // Store results in global/module variable
    // Clear interrupt flag
}
```
Example interrupt

```c
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

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
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne

Spring 2016
Hardware Peripherals

- Digital pins
- Timers
- ADC
Hardware Peripherals

- Communications
- Pin change notification
- DMA
- Output compare
- Input capture
- Digital pins
- Timers
- ADC
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

Pic32
Pic24
Pic16
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

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

Digital pins

• Voltage
  – High T 3.3 - 2.7
  – Low F 0 - .8

• Direction
  – Input
  – Output

• Polling interface
Hardware Peripherals

Digital pins

Dedicated Port Module

I/O Cell

I/O pin

Synchronization
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

Synchronization

I/O Cell

I/O pin

CMPE-013/L: “C” Programming

Maxwell James Dunne – Spring 2016
Hardware Peripherals

Timers

• Multiple 16-bit timers
  – 5 total

• Interrupt-based
  – ISR is called every X seconds

• Configurable periodicity
  – Range from 20MHz to 305Hz

\[ 2^{16} = 32 \]

\[ \text{1ms} \]

\[ 16 - \text{bit} + 29 \text{ 0s} \]

\[ 32 - \text{bit} = 30 \text{days} \]
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:
  – $20\text{MHz} / \text{PRx} = \text{periodicity}$
• PRx of 20000 $\rightarrow$ 1kHz interrupts

$20\text{MHz} / 2, 8, 16, 32, 64$
Hardware Peripherals

Timers
Hardware Peripherals

Timers

\[ \begin{array}{c}
65535 \\
0
\end{array} \]

TMRx
Hardware Peripherals

Timers

65K

PRx

TMRx

event
Hardware Peripherals

Timers

PRx → CPU

event
Hardware Peripherals

Timers

\[ SFTI = 1 \]

PRx → CPU

TMRx

event

Interrupt()
Hardware Peripherals

ADC

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

0-1023
Hardware Peripherals

ADC

- The input signal is continuously sampled
- Every 8th sample triggers an interrupt

average
Hardware Peripherals

ADC

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

ADC

SHAH

VREFH

VREFL

SAR ADC

ADC1BUF0

ADC1BUF1

ADC1BUF2

ADC1BUFE

ADC1BUFF
Hardware Peripherals

ADC

ADC1BUF0 = 2
ADC1BUF1 = 146
ADC1BUF2 = 288
ADC1BUF3 = 420
ADC1BUF4 = 563
ADC1BUF5 = 691
ADC1BUF6 = 829
ADC1BUF7 = 987
Hardware Peripherals

ADC

event

<table>
<thead>
<tr>
<th>ADC1BUF0</th>
<th>ADC1BUF1</th>
<th>ADC1BUF2</th>
<th>ADC1BUF3</th>
<th>ADC1BUF4</th>
<th>ADC1BUF5</th>
<th>ADC1BUF6</th>
<th>ADC1BUF7</th>
</tr>
</thead>
<tbody>
<tr>
<td>950</td>
<td>600</td>
<td>100</td>
<td>65</td>
<td>81</td>
<td>93</td>
<td>107</td>
<td>122</td>
</tr>
</tbody>
</table>
// struct x = 0, False

x = 0, initialized = false;

struct x;

struct *y = &x;

stack init(cx);

struct *y = &x;
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
    }
}
```
Event-driven Programming

Real-world example

```c
{
    while (1) {
        if (buttonsEvent) {
            // Update fixed LED mask
        }
        if (adcEvent) {
            // Update OLED
        }
        if (timerEvent) {
            // Update bouncing LED mask
        }
        if (ledEvent) {
            // Update LEDs
        }
    }
}
```
Event-driven Programming

Real-world example

```c
static uint8_t buttonsEvent;

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

void _ISR Timer1Int(void)
{
    buttonsEvent = ButtonsCheckEvents();
    IFS0 &= ~(1 << 3);
}
```
Bit manipulation

Bit masking
Bit flags
Bit fields
Bit manipulation

Bit packing

- Data is commonly packed into larger unsigned integers on embedded systems
- Generally a tie in to hardware or when space is critical
  - Hardware
  - Storage
  - Binary formats
Bit manipulation

Bit packing

C1CTRL1 – dsPIC33EP256MC502