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.

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
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

```c
x = 2
x++, re = 1
```
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;
        y = *x;
    }
}

int main(void)
{
    int *myInt;
    MyFunc(&myInt);
}
```
 Enums
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
```
enum typeName {label_0, label_1, ..., label_n}
```
Where compiler sets $label_0 = 0$, $label_1 = 1$, $label_n = n$

Example
```
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 {label₀ = const₀, ..., labelₙ}
```

Where compiler sets `label₀ = const₀, label₁ = (const₀ + 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 \( label_0 = const_0, label_1 = (const_0 + 1), \ldots \)

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:

Rob = -4, Steve = -3, Paul = -2, Bill = -1, Gary = 0
Enumerations
How to Declare an Enumeration Type Variable

- Declared along with type:

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

- Declared independently:

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

**Example**
```
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

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

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

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

Example

```c
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

\[ \text{varName} = \text{label}_n; \]

- 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 values

Example

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

day = WED;
day = 5; // May only use values from 0 to 6
if (day == WED) {
    ...
```
Enumerations

Proper formatting

Example

typedef enum {
0|SUN,
1|MON,
2|TUE, // last
3|WED, // office hours
4|THR, // class
5|FRI,
6|SAT
} Weekday;

#define SUN 0

Weekday day = WED;

#define week [6]

array[0]
array[17]
array + 0
array + 1
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
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;
}
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;
}
```
40 Mips

115200/9 = 12,800 bytes

3,125 cycles
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

Non-blocking code
CMPE-013/L

Introduction to “C” Programming

Max Dunne

Winter 2015
Hardware Peripherals

Digital pins
Timers
ADC
Hardware Peripherals

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

SERIAL
Hardware Peripherals

Special function registers (SFRs)

- 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
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 __attribute__((__sfr__)) IFS0;
```
Hardware Peripherals

Digital pins

- Voltage
  - High \( \sim 2.3\,\text{V} \)
  - Low 0 V

- Direction
  - Input
  - Output

- Polling interface
Hardware Peripherals

Digital pins

Dedicated Port Module

RD TRISx

WR TRISx

WR LATx

RD LATx

RD PORTx

Synchronization

I/O Cell

I/O pin

CMPE-013/L: “C” Programming

Gabriel Hugh Elkaim – Winter 2015
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

I/O Cell

I/O pin

Synchronization

RD TRISx

WR TRISx

WR LATx

RD LATx

RD PORTx

WR PORTx

LATx

TRISx

ODCx

CMPE-013/L: “C” Programming

Gabriel Hugh Elkaim – Winter 2015
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

Gabriel Hugh Elkaim – Winter 2015
LAT = 1

Port = 1

1 0

Ethernet

LAT = 1;

Write to LAT

Read from Port
Hardware Peripherals

Timers

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

40MHz

2 - 5 Hz
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 -> 1kHz interrupts
Hardware Peripherals

Timers

65535
Hardware Peripherals

Timers

65535

TMRx

0
Hardware Peripherals

Timers

PRx

TMRx

event

1000
Hardware Peripherals

Timers

PRx \[\rightarrow\] TMRx \[\rightarrow\] CPU

event
Hardware Peripherals

Timers

PRx → Event

TMRx

CPU

Interrupt() → 3 timers
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

\[
\frac{3.3\,\text{V}}{1024} = 0.0032226
\]

\[
0 - 3.3\,\text{V} \rightarrow 0 - 1023
\]

1 V
Hardware Peripherals

ADC

- The input signal is continuously sampled
- Every 8th sample triggers an interrupt
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 $V_{\text{ref}} / 1024 = 0.0032V$
Hardware Peripherals

ADC

SHA
VREFH
VREFL
SAR ADC
ADC1BUF0
ADC1BUF1
ADC1BUF2
ADC1BUF
ADC1BUFE
ADC1BUFF
Hardware Peripherals
ADC

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

ADC

\[
\begin{align*}
\text{ADC1BUF0} & = 950 \\
\text{ADC1BUF1} & = 600 \\
\text{ADC1BUF2} & = 100 \\
\text{ADC1BUF3} & = 65 \\
\text{ADC1BUF4} & = 81 \\
\text{ADC1BUF5} & = 93 \\
\text{ADC1BUF6} & = 107 \\
\text{ADC1BUF7} & = 122
\end{align*}
\]
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
        }
    }
}
```
static uint8_t buttonsEvent;

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

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

Introduction to “C” Programming

Maxwell James Dunne
Switch statements
**switch Statement**

**Syntax**

```c
switch (expression) {
    case const-expr\_1: statements\_1
    :
    case const-expr\_n: statements\_n
    default: statements\_n+1
}
```

- **expression** is evaluated and tested for a match with the **const-expr** in each **case** clause.
- The **statements** in the matching **case** clause is executed.
**switch Statement**

Flow Diagram (default)

Start

1. `Const-expr_1 = expression?`
   - YES: `statement_1`
   - NO: `Const-expr_2 = expression?`
2. `Const-expr_2 = expression?`
   - NO: `(...)`
   - YES: `statement_2`
3. `Const-expr_n = expression?`
   - NO: `statement_n`
   - YES: `statement_n`

This is the default behavior of the `switch` statement.

Notice that each statement falls through to the next.
switch Statement

Flow Diagram (modified)

START

Const-expr_1 = expression?

YES

statement_1
break;

NO

Const-expr_2 = expression?

YES

statement_2
break;

NO

...  

Const-expr_n = expression?

YES

statement_n
break;

NO

statement_{n+1}

END

Adding a break statement to each statement block will eliminate fall through, allowing only one case clause's statement block to be executed.
```c
switch (channel) {
    case 2:  puts("WBBM Chicago"); break;
    case 3:  puts("DVD Player"); break;
    case 4:  puts("WTMJ Milwaukee"); break;
    case 5:  puts("WMAQ Chicago"); break;
    case 6:  puts("WITI Milwaukee"); break;
    case 7:  puts("WLS Chicago"); break;
    case 9:  puts("WGN Chicago"); break;
    case 10: puts("WMVS Milwaukee"); break;
    case 11: puts("WTTW Chicago"); break;
    case 12: puts("WISN Milwaukee"); break;
    default: puts("No Signal Available");
}
```
switch Statement

Styling

switch Example 1

```c
switch (channel) {
    case 2:
        puts("WBBM Chicago");
        break;
    case 3:
        puts("DVD Player");
        break;
    case 4:
        puts("WTMJ Milwaukee");
        break;
    ...
}
```
switch (letter) {
  case 'a':
    puts("Letter 'a' found.");
    break;
  case 'b':
    puts("Letter 'b' found.");
    break;
  case 'c':
    puts("Letter 'c' found.");
    break;
  default:
    puts("Letter not in list.");
}
switch Statement

Fall-through

switch Example 3

```c
switch (channel) {
    case 4:
    case 5:
    case 6:
    case 7:
        puts("VHF Station");
        break;
    case 9:
    case 10:
    case 11:
    case 12:
        puts("VHF Station");
        break;
    default:
        puts("No Signal Available");
}
```
switch Example 3

```c
switch (channel) {
    case 4 ... 7:
        puts("VHF Station");
        break;
    case 9 ... 12:
        puts("VHF Station");
        break;
    default:
        puts("No Signal Available");
}
```
isHex (char in) TF

switch (in) {
    case '0' ... '9':
    case 'a' ... 'f':
    case 'A' ... 'F':
        return TRUE;
    default: break;
        return FALSE;
}
switch Statement
Real-world example

switch Example 2

bool IsHex(char character)
{
    switch (character) {
    case 'a' ... 'f':
    case 'A' ... 'F':
    case '0' ... '9':
        return true;
    default:
        return false;
    }
}
CMPE-013/L

Linked Lists

Maxwell James Dunne
Linked List
Theory

a → b → c → d → e

Dovel
typedef struct ListItem {
    struct ListItem *previousItem;
    struct ListItem *nextItem;
    char *data;
} ListItem;
Linked List

ListItem *ListNew(char *data);

```
ListItem *x = malloc(sizeof(ListItem));
if (x) {
    x->data = data;
    return x;
}
```
Linked List

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

\[ \text{item} \rightarrow N = x; \]
\[ x \rightarrow r = \text{item}; \]
List Item

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

`Z → N = item → N;
Z → N → p = Z;
item → N = Z;
Z → p = item;`
Linked List

char *LinkedListRemove(ListItem *item);
Linked List

char *LinkedListRemove(ListItem *item);

item \rightarrow p \Rightarrow N = NULL
free(item)
Linked List

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

```
0 ← item → p → n = item → n;
item → N → p = item → p;
```
Linked List

char *LinkedListRemove(ListItem *item);
Linked List

char *LinkedListRemove(ListItem *item);

Linked List

ListItem *LinkedListGetFirst(ListItem *list);

item → p == NULL

NULL
for ( ; item = item -> p )
Linked List

```c
int LinkedListSize(ListItem *list);
```

```
get first
```

```c
int LinkedListPrint(ListItem *list);
```

```c
d = NULL

printf("\n", d);

strcmp(
```
\[ + = q \rightarrow data + a; \]
\[ a \rightarrow data + a = b \rightarrow data + a; \]
\[ b \rightarrow data + e; \]
\[ d = 0 \]
for i = 0 to length(A) - 2 do
  for j = i + 1 to length(A) - 1 do
      [# swap A[j] and A[i] #]
    end if
  end for
end for

for (get first; NULL; item = item \rightarrow n)
Heap size

8 1 4 2

struct item

(node, cat, cow, cow, dog)

strncpy(item->data)

set size

print

remove