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));
    ...
    free(x);
}
```

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

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

Example interrupt

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

int main(void)
{
    int myInt;
    int *myIntPtr = &myInt;
    MyFunc(&myIntPtr);
}
```
Example interrupt

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

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

Nothing to do with the compiler...

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**

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

**Example**

```plaintext
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**

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

**Example**

```plaintext
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**

```
enum typeName {label\_0 = const\_0, \ldots, label\_n}
```

Where compiler sets label\_0 = const\_0, label\_1 = (const\_0 + 1), ...

**Example**

```cpp
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:

**Syntax**

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

- Declared independently:

**Syntax**

```
enum typeName varName\_1, \ldots, varName\_n;
```

**Example**

```cpp
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

```
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**

```c
typedef enum {
    SUN,
    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

Interrupts

Standard form

```
Example interrupt

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

Real-world example

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;

    return;
}
```

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

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