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
}
```
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;
}
```
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;
}
```
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
Pointers

Pointers and memory
Pointer/array equivalency
Pointer arithmetic
Pointers and the stack
Pointers and strings
Arrays of pointers
Pointers
How to Create a Pointer Variable

Syntax
```
(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
```
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
```
Pointers

Initialization

• To set a pointer to point to another variable, we use the & operator (address of), and the pointer variable is used without the dereference operator *:

\[ p = \& x; \]

• This assigns the address of the variable \( x \) to the pointer \( p \) (\( p \) now points to \( x \))

Note: \( p \) must be declared to point to the type of \( x \) (e.g. \( int x; int *p; \))
Pointers

Dereferencing

• When accessing the data pointed to by a pointer, we use the pointer with the dereference operator *:

\[ y = *p; \]

• This assigns to the variable \( y \), the value of what \( p \) is pointing to (\( x \) from the last slide)
• Using \( *p \), is the same as using the variable it points to (e.g. \( x \))
Pointers
Dereferencing example

Example

```c
int x = 6, *p; // int and a pointer to int
p = &x;       // Assign p the address of x
*p = 5;       // Same as x = 5;
```

- `&x` is a constant memory value
  - It represents the address of `x`
  - The address of `x` will never change

- `p` is a variable pointer to `int`
  - It can be assigned the address of any `int`
  - It may be assigned a new address any time
Pointers

Dereferencing example

Example

```c
int x = 6, *p; // int and a pointer to int
p = &x;        // Assign p the address of x
*p = 5;        // Same as x = 5;
```

- \*p represents the data pointed to by \p
  - \*p may be used anywhere you would use \x
  - \* is the dereference operator, also called the indirection operator
  - In the pointer declaration, the only significance of \* is to indicate that the variable is a pointer rather than an ordinary variable
Pointers

Another view

Contents of the Mailbox
(variable \( x \))

Address of Mailbox
(\&\( x \))

Bank of Mailboxes
(memory locations)
Pointers

Another view

Contents of the Mailbox 

\((x, *p)\)

Address of Mailbox 

\((&x, p)\)

Bank of Mailboxes 

(memory locations)

\(p = &x;\)
Pointers

Another view

Contents of the Mailbox

\((x, *p)\)

Address of Mailbox

\((&x, p)\)

\(*p = 2;\)

Bank of Mailboxes

(memory locations)
Pointers
Dereferencing non-primitives

Example

Complex x = {0.6, 1.2}, *p;
p = &x;
p->re = 5;

• p->re represents the data pointed to by p
  – p->re may be used anywhere you would use x.re
  – -> is the structure dereference operator, equivalent to (*p).re
  – In the pointer declaration, the only significance of * is to indicate that the variable is a pointer rather than an ordinary variable
void MyFunc(Complex *x)
{
    Complex t = *x;

    x->re /= t.re * t.re + t.im * t.im;
    x->im /= t.re * t.re + t.im * t.im;
}
void MyFunc(Complex *x) {
    Complex t = *x;
}
Pointers

Dereferencing non-primitives

```c
void MyFunc(Complex *x)
{
    Complex t = *x;
}
```
Pointers
How Pointers Work

Example

```c
{  
    int x, y;
    int *p;

    x = 0xDEAD;
    y = 0xBEEF;
    p = &x;

    *p = 0x0100;
    p = &y;
    *p = 0x0200;
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable at Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td>x</td>
</tr>
<tr>
<td>0x08BC</td>
<td>y</td>
</tr>
<tr>
<td>0x08C0</td>
<td>p</td>
</tr>
<tr>
<td>0x08C4</td>
<td></td>
</tr>
<tr>
<td>0x08C8</td>
<td></td>
</tr>
<tr>
<td>0x08CC</td>
<td></td>
</tr>
<tr>
<td>0x08D0</td>
<td></td>
</tr>
<tr>
<td>0x08D4</td>
<td></td>
</tr>
</tbody>
</table>
Pointers
How Pointers Work

Example

```c
{ 
    int x, y;
    int *p;
    
x = 0xDEAD;
    y = 0xBEEF;
    p = &x;
    
    *p = 0x0100;
    p = &y;
    *p = 0x0200;
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000 0000</th>
<th>0000 DEAD</th>
<th>0000 0000</th>
<th>0000 0000</th>
<th>0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08BC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08D4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Variable at Address

x

y

p
Pointers
How Pointers Work

Example

```c
int x, y;
int *p;

x = 0xDEAD;

\text{\textcolor{red}{y = 0xBEEF;}}
p = &x;

*p = 0x0100;
p = &y;
\text{\textcolor{red}{*p = 0x0200;}}
```
Pointers
How Pointers Work

Example

```c
{  
  int x, y;  
  int *p;  
  x = 0xDEAD;  
  y = 0xBEEF;  
  p = &x;  
  *p = 0x0100;  
  p = &y;  
  *p = 0x0200;  
}
```

32-bit Data Memory (RAM)

Variable at Address

<table>
<thead>
<tr>
<th>Address</th>
<th>x</th>
<th>y</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td>0000</td>
<td>DEAD</td>
<td></td>
</tr>
<tr>
<td>0x08BC</td>
<td>0000</td>
<td>BEEF</td>
<td></td>
</tr>
<tr>
<td>0x08C0</td>
<td>0000</td>
<td></td>
<td>08BC</td>
</tr>
<tr>
<td>0x08C4</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08C8</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08CC</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08D0</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08D4</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pointers
How Pointers Work

Example

```c
int x, y;
int *p;
x = 0xDEAD;
y = 0xBEEF;
p = &x;
*x = 0x0100;
p = &y;
*p = 0x0200;
```
Pointers
How Pointers Work

Example

```c
int x, y;
int *p;

x = 0xDEAD;
y = 0xBEEF;
p = &x;

*p = 0x0100;
p = &y;
*p = 0x0200;
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08B8</td>
<td>0000 0100</td>
</tr>
<tr>
<td>0x08BC</td>
<td>0000 BEEF</td>
</tr>
<tr>
<td>0x08C0</td>
<td>0000 08C0</td>
</tr>
<tr>
<td>0x08C4</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08C8</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08CC</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08D0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x08D4</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

Maxwell James Dunne – Spring 2015
Pointers
How Pointers Work

Example

```
{ 
    int x, y;
    int *p;
    x = 0xDEAD;
    y = 0xBEEF;
    p = &x;
    *p = 0x0100;
    p = &y;
    *p = 0x0200;
}
```
Pointers and Arrays
A Quick Reminder...

- Array elements occupy consecutive memory locations

```
int x[3] = {1, 2, 3};
```

```
isn't p;
*p = x[0]
*(p + 1) = x[1]
```

<table>
<thead>
<tr>
<th>Address</th>
<th>FFFF FFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td>0000 001</td>
</tr>
<tr>
<td>0x0800</td>
<td>0000 002</td>
</tr>
<tr>
<td>0x0804</td>
<td>0000 003</td>
</tr>
<tr>
<td>0x0808</td>
<td>FFFF FFFF</td>
</tr>
<tr>
<td>0x080C</td>
<td>FFFF FFFF</td>
</tr>
</tbody>
</table>

- Pointers can provide an alternate method for accessing array elements
Pointers and Arrays

Initializing a Pointer to an Array

- The array name evaluates to the address of its first (0th) element

If we declare the following array and pointer variable:

```c
int x[5] = {1, 2, 3, 4, 5};
int *p;
```

We can initialize the pointer to point to the array using either of these methods:

```c
p = x; // Works only for arrays
p = &x[0]; // Same as the above
```
Pointers and Arrays
A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

```c
int x[3] = {1, 2, 3};
int *p;
p = x;
++p;
```

32-bit Data Memory (RAM)

```
<table>
<thead>
<tr>
<th></th>
<th>Address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0x07FC</td>
<td>FFFF FFFF</td>
</tr>
<tr>
<td>x</td>
<td>0x0800</td>
<td>0000 0001</td>
</tr>
<tr>
<td></td>
<td>0x0804</td>
<td>0000 0002</td>
</tr>
<tr>
<td>x</td>
<td>0x0808</td>
<td>0000 0003</td>
</tr>
<tr>
<td>x</td>
<td>0x080C</td>
<td>FFFF FFFF</td>
</tr>
</tbody>
</table>
```

More on this in just a bit...
Pointers and Arrays
A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

```c
int x[3] = {1, 2, 3};
int *p;
p = x;
++p;
```

32-bit Data Memory (RAM)

- `x[0]`: 0000 0001
- `x[1]`: 0000 0002
- `x[2]`: 0000 0003

Address
- 0x07FC
- 0x0800
- 0x0804
- 0x0808
- 0x080C

More on this in just a bit...

Maxwell James Dunne – Spring 2015
Pointers and Arrays
A Preview of Pointer Arithmetic

- Incrementing a pointer will move it to the next element of the array

```c
int x[3] = {1, 2, 3};
int *p;
p = x;
++p;
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0000</td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td>0000</td>
<td>0002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td>0000</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FFFF</td>
<td>FFFF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

More on this in just a bit...
Pointer Arithmetic

Incrementing Pointers

• Incrementing or decrementing a pointer will add or subtract a multiple of the number of bytes of its base type

• If we have:

```c
float x;
float *p = &x;
++p;
```

We will the address of p incremented by 4 since a float occupies 4 bytes
**Pointer Arithmetic**

**Incrementing Pointers**

**Example**

```c
float *ptr;

ptr = &a;
++ptr;
```

Incrementing `ptr` moves it to the next sequential `float` array element (4 bytes ahead)

32-bit Data Memory Words
Pointer Arithmetic
Larger Jumps

- Adding or subtracting any other number with the pointer will change it by a multiple of the number of bytes of its type.
- If we have

```c
short int x;
short int *p = &x;
p += 3;
```

We will get the address of `p` incremented by 6 since a `short int` variable occupies 2 bytes of memory.
**Pointer Arithmetic**
Larger Jumps

**Example**

```c
float *ptr;
ptr = a;
```

Adding 6 to `ptr` moves it 6
float array elements ahead
(24 bytes ahead)

```c
ptr += 6;
```

16-bit Data Memory Words
long long x[] = {1, 2, 3};
long long *p = x;

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;

p -= 2;
*p = 0xBAD0000F00D1;

32-bit Data Memory (RAM)

0000 0000
0000 0001
0000 0000

0000 0002
0000 0000
0000 0003
0000 0000

0000 0800
Pointers

PointerType Arithmetic

Example

```c
long long x[] = {1, 2, 3};
long long *p = x;

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0xBAD0000F00D1;
```
```c
long long x[] = {1, 2, 3};
long long *p = x;

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0xBAD0000F00D1;
```
Pointers

Pointer Arithmetic

Example

```c
long long x[] = {1, 2, 3};
long long *p = x;

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0xBAD0000F00D1;
}
```

32-bit Data Memory
(RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000 0000</th>
<th>0000 0005</th>
<th>0000 0000</th>
<th>0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0804</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0808</td>
<td>0000 0000</td>
<td>1234 BEEF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x080C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0810</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0814</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0818</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

Maxwell James Dunne – Spring 2015
Pointers

Pointer Arithmetic

Example

```c
{ long long x[] = {1, 2, 3};
    long long *p = x;

    *p += 4;
    ++p;
    *p = 0xDEAD1234BEEF;
    ++p;
    *p = 0xF1D04321F00D;
    p -= 2;
    *p = 0xBAD0000F00D1;
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>x[0]</th>
<th>x[1]</th>
<th>x[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td>0000 0000</td>
<td>0000 0005</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x0800</td>
<td>0000 0000</td>
<td>1234 BEEF</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0804</td>
<td>0x8000 0000</td>
<td>0000 DEAD</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0808</td>
<td>0x8000 0000</td>
<td>0x8000 0000</td>
<td>0x8000 0000</td>
</tr>
<tr>
<td>0x080C</td>
<td>0x8000 0000</td>
<td>0x8000 0000</td>
<td>0x8000 0000</td>
</tr>
<tr>
<td>0x0810</td>
<td>0x8000 0000</td>
<td>0x8000 0000</td>
<td>0x0810</td>
</tr>
<tr>
<td>0x0814</td>
<td>0x8000 0000</td>
<td>0x8000 0000</td>
<td>0x8000 0000</td>
</tr>
<tr>
<td>0x0818</td>
<td>0x8000 0000</td>
<td>0x8000 0000</td>
<td>0x8000 0000</td>
</tr>
</tbody>
</table>
Example

```c
long long x[] = {1, 2, 3};
long long *p = x;

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0xBAD0000F00D1;
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FC</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x800</td>
<td>0000 0005</td>
</tr>
<tr>
<td>0x804</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x808</td>
<td></td>
</tr>
<tr>
<td>0x80C</td>
<td>1234 BEEF</td>
</tr>
<tr>
<td>0x810</td>
<td>4321 F00D</td>
</tr>
<tr>
<td>0x814</td>
<td>0000 DEAD</td>
</tr>
<tr>
<td>0x818</td>
<td>0000 F1D0</td>
</tr>
</tbody>
</table>

p = 0000 0810
Pointers

Pointer Arithmetic

Example

```c
{ 
    long long x[] = {1, 2, 3};
    long long *p = x;
    *p += 4;
    ++p;
    *p = 0xDEAD1234BEEF;
    ++p;
    *p = 0xF1D04321F00D;
    p -= 2;
    *p = 0xBA0D0000F00D1;
}
```
Pointers

Pointer Arithmetic

Example

```c
long long x[] = {1, 2, 3};
long long *p = x;

*p += 4;
++p;
*p = 0xDEAD1234BEEF;
++p;
*p = 0xF1D04321F00D;
p -= 2;
*p = 0xBAD0000F00D1;
```
# Pointers

Post-Increment/Decrement Syntax Rule

- Care must be taken with respect to operator precedence when doing pointer arithmetic:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
<th>Description by Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*p++</td>
<td>Post-Increment Pointer</td>
<td>$z = *(p++)$; is equivalent to: $z = *p$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$p = p + 1;$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* (p++)</td>
<td>Post-Increment data pointed to by Pointer</td>
<td>$z = (*p)++;$ is equivalent to: $z = *p$; $*p = *p + 1;$</td>
</tr>
</tbody>
</table>

Maxwell James Dunne – Spring 2015
Pointers

Post-Increment / Decrement Syntax

Example

```c
int x[3] = {1, 2, 3};
int y;
int *p = x;
y = 5 + *(p++);
y = 5 + (*p)++;
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0000 0001</td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td>0000 0002</td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td>0000 0003</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0000 0800</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0000 0000</td>
<td></td>
</tr>
</tbody>
</table>

Remember:
*(p++) is the same as *p++
Pointers
Post-Increment / Decrement Syntax

Example

```c
int x[3] = {1, 2, 3};
int y;
int *p = x;
y = 5 + *(p++);
y = 5 + (*p)++;
```

32-bit Data Memory (RAM)

```
<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
<th>0x0810</th>
<th>0x0814</th>
<th>0x0818</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0002</td>
<td>0000 0003</td>
<td>0000 0800</td>
<td>0000 0006</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>
```

Remember:

*(p++) is the same as *p++
Pointers
Post-Increment / Decrement Syntax

Example

```c
{ 
  int x[3] = {1, 2, 3};
  int y;
  int *p = x;
  y = 5 + *(p++);
  y = 5 + (*p)++;
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000 0000</th>
<th>0000 0001</th>
<th>0000 0002</th>
<th>0000 0003</th>
<th>0000 0804</th>
<th>0000 0806</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x07FC</td>
<td>0x0800</td>
<td>0x0804</td>
<td>0x0808</td>
<td>0x0810</td>
<td>0x0818</td>
</tr>
</tbody>
</table>

Remember:
*(p++) is the same as *p++
Pointers
Post-Increment / Decrement Syntax

Example

```c
int x[3] = {1, 2, 3};
int y;
int *p = x;
y = 5 + *(p++);

y = 5 + (*p)++;
```

32-bit Data Memory (RAM)

- Address: 0x07FC
  - x[0]: 0000 0001
  - x[1]: 0000 0002
  - x[2]: 0000 0003
  - p: 0000 0804
  - y: 0000 0007

Remember: *(p++) is the same as *p++
**Pointers**

**Post-Increment / Decrement Syntax**

**Example**

```c
int x[3] = {1, 2, 3};
int y;
int *p = x;
y = 5 + *(p++);
y = 5 + (*p)++;
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000 0000</th>
<th>0000 0001</th>
<th>0000 0003</th>
<th>0000 0804</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0804</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0000 0007</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0000 0010</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember:

* (p++) is the same as *p++
Pointers

Pre-Increment/Decrement Syntax Rule

- Care must be taken with respect to operator precedence when doing pointer arithmetic:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
<th>Description by Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+++*p</td>
<td>Pre-Increment Pointer</td>
<td>$z = *(++p);$ is equivalent to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$p = p + 1;$</td>
</tr>
<tr>
<td>**(++p)</td>
<td>Pointer</td>
<td>$z = *p;$</td>
</tr>
<tr>
<td>++(*p)</td>
<td>Pre-Increment data pointed to by Pointer</td>
<td>$z = **(++p);$ is equivalent to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$*p = *p + 1;$.</td>
</tr>
</tbody>
</table>

Maxwell James Dunne – Spring 2015
Pointers
Pre-Increment / Decrement Syntax

Example

```c
{ int x[3] = {1, 2, 3};
  int y;
  int *p = x;
  y = 5 + *(++p);
  y = 5 + ++(*p);
}
```

32-bit Data Memory (RAM)

Address
0x07FC
0x0800
0x0804
0x0808
0x080C
0x0810
0x0814
0x0818

```
0000 0000
0000 0001
0000 0002
0000 0003
0000 0800
0000 0000
0000 0000
0000 0000
```

Remember:

`*(++p)` is the same as `*++p`
Pointers
Pre-Increment / Decrement Syntax

Example

```c
{  
  int x[3] = {1, 2, 3};
  int y;
  int *p = x;
  
  y = 5 + *(*(++p));
  y = 5 + ++(*p);
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
<th>0x0810</th>
<th>0x0814</th>
<th>0x0818</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0002</td>
<td>0000 0003</td>
<td>0000 0804</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>x[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember:

\[^{\text{++p}}\] \text{ is the same as } \[^{\text{+++p}}\]
Pointers
Pre-Increment / Decrement Syntax

Example

```c
{  
  int x[3] = {1, 2, 3};
  int y;
  int *p = x;
  y = 5 + *(++p);
  y = 5 + ++(*p);
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000 0000</th>
<th>0000 0001</th>
<th>0000 0002</th>
<th>0000 0003</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07FC</td>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0002</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x0800</td>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0002</td>
<td>0000 0003</td>
</tr>
<tr>
<td>0x0804</td>
<td>0000 0804</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0808</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0810</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0814</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0x0818</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

Remember:
* (++p) is the same as *+++p

Maxwell James Dunne – Spring 2015
CMPE-013/L: “C” Programming
Pointers
Pre-Increment / Decrement Syntax

Example

```c
{ int x[3] = {1, 2, 3};
  int y;
  int *p = x;

  y = 5 + *(++p);
  y = 5 + ++(*p);
}
```

32-bit Data Memory (RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0000 0000</th>
<th>0000 0001</th>
<th>0000 0003</th>
<th>0000 0804</th>
<th>0000 0807</th>
<th>0000 0808</th>
<th>0000 080C</th>
<th>0000 0810</th>
<th>0000 0814</th>
<th>0000 0818</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x07FC</td>
<td>0x0800</td>
<td>0x0804</td>
<td>0x0808</td>
<td>0x0810</td>
<td>0x0814</td>
<td>0x0818</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember: 

`*(++p)` is the same as `*++p`
Pointers

Pre-Increment / Decrement Syntax

Example

```c
{  
  int x[3] = {1, 2, 3};
  int y;
  int *p = x;
  y = 5 + *(++p);
  y = 5 + ++(*p);
}
```

32-bit Data Memory
(RAM)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x07FC</th>
<th>0x0800</th>
<th>0x0804</th>
<th>0x0808</th>
<th>0x080C</th>
<th>0x0810</th>
<th>0x0814</th>
<th>0x0818</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0000 0001</td>
<td>0000 0003</td>
<td>0000 0003</td>
<td>0000 0804</td>
<td>0000 0008</td>
<td>0000 0000</td>
<td>0000 0000</td>
<td></td>
</tr>
</tbody>
</table>

Remember: *(++p) is the same as *++p
Pointers
Pre- and Post- Increment/Decrement Summary

• The parentheses determine what gets incremented/decremented:

Modify the pointer itself

\[ *\ (++p) \text{ or } +++p \text{ and } *\ (p++) \text{ or } *p++ \]

Modify the value pointed to by the pointer

\[ ++(*)p \text{ and } (*p)++ \]
Pointers

Initialization Tip

• If a pointer isn't initialized to a specific address when it is created, it is a good idea to initialize it as NULL.
• This will prevent it from unintentionally corrupting a memory location if it is accidentally used before it is initialized.

Example

```c
int *p = NULL; // Ex
```

**Note:** NULL is the character '\0' but NULL is the value of a pointer that points to nowhere.
Pointers and the Stack

Beware the stack

- Memory addresses may not always be valid
- Addresses referring to the stack have a lifetime tied to that variables scope
- Only global, static, and pointers returned by malloc() will always be valid
- You should almost never use the memory addresses of variables on the stack
Pointers and the Stack

Example function

```c
int *foo(int x, int y)
{
    int z = x + (++y);
    return &z;
}
```

Example caller

```c
int *main(void)
{
    int a = 6, b = 19;
    int *c = foo(a, b);
    printf("%d\n", *c);
}
```
Pointers and the Stack

Example function

```c
int *foo(int x, int y)
{
    int z = x + (++y);
    return &z;
}
```

Example caller

```c
int main(void)
{
    int a = 6, b = 19;
    int *c = foo(a, b);
    printf("%d\n", *c);
}
```
Pointers and the Stack

Beware the stack

Example function

```c
int *foo(int x, int y)
{
    int z = x + (;++y);
    return &z;
}
```

Example caller

```c
{
    int a = 6, b = 19;
    int *c = foo(a, b);
    puts("Hey! ");
    printf("%d\n", *c);
}
```
Pointers and Strings

- So far, we have worked with strings strictly as arrays of `char`
- Strings may be created and used with pointers much more elegantly

String declaration with a pointer:
```
char *str = "PIC32MX"; str
```

Implementation varies depending on compiler and architecture used.
Pointers and Strings

- When initialized, a pointer to a string points to the first character:

```c
char *str = "Microchip";

str += 4
```

- Increment or add an offset to the pointer to access subsequent characters
Pointers and Strings

- Pointers may also be used to access characters via an offset:

```c
char *str = "Microchip";
*str == 'M'

Microchip

*(str + 4) == 'o'
```

- Pointer always points to "base address"

Offsets used to access subsequent chars
Pointers and Strings

Pointer versus Array: Initialization at Declaration

- Depending on variable type, part of the variable is constant

<table>
<thead>
<tr>
<th>Example: Pointer to String Constant</th>
<th>Example: Character array</th>
</tr>
</thead>
</table>
| ```
char *str = "PIC";
``` | ```
char str[] = "PIC";
``` |

The NULL character '\0' is automatically appended to strings in both cases (array must be large enough).
Pointers and Strings
Pointer versus Array: Initialization at Declaration

Example: Pointer Variable

```c
char *str1 = "PIC";
char str2[] = "PIC";
```

<table>
<thead>
<tr>
<th>RAM</th>
<th>ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9D0008C0</td>
<td>0x9D0008C0</td>
</tr>
<tr>
<td>0xA0000FB0</td>
<td>0xA0000FB0</td>
</tr>
<tr>
<td>0xA0000FB4</td>
<td>0xA0000FB4</td>
</tr>
<tr>
<td>0xA0000FB5</td>
<td>0xA0000FB5</td>
</tr>
<tr>
<td>0xA0000FB6</td>
<td>0xA0000FB6</td>
</tr>
<tr>
<td>\0</td>
<td>0xA0000FB7</td>
</tr>
</tbody>
</table>

```
str1 str2
```
Pointers and Strings

Pointer versus Array: Assignment in Code

- An entire string may be assigned to a pointer
- A character array must be assigned character by character

Example: Pointer Variable

```c
char *str;
str = "PIC";
```

Example: Array Variable

```c
char str[4];
str[0] = 'P';
str[1] = 'I';
str[2] = 'C';
str[3] = '\0';
```

Must explicitly add NUL character '\0' to array.
Pointers and Strings

Comparing Strings

str = "Microchip",

- If you want to test a string for equivalence, the natural thing to do is:
  
  ```c
  if (str == "Microchip")
  ```

- This is not correct, though it might appear to work sometimes.

- This compares the address in `str` to the address of the string literal "Microchip"

- The correct way is to use the `strcmp()` function in the standard library which compares strings character by character.
Arrays of Pointers

Declaration

• An array of pointers is an ordinary array variable whose elements happen to all be pointers.

```
char *p[4];
```

• This creates an array of 4 pointers to `char`
  – The array `p[]` itself is like any other array
  – The elements of `p[]`, such as `p[1]`, are pointers to `char`
Arrays of Pointers

Array Elements are Pointers Themselves

32-bit Data Memory (RAM)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p[0]</td>
<td>9D00</td>
<td>3FC0</td>
<td>On</td>
</tr>
<tr>
<td>p[1]</td>
<td>9D00</td>
<td>3FC3</td>
<td>Off</td>
</tr>
<tr>
<td>p[2]</td>
<td>9D00</td>
<td>3FC7</td>
<td>Main</td>
</tr>
<tr>
<td>p[3]</td>
<td>9D00</td>
<td>3FCC</td>
<td>Aux</td>
</tr>
</tbody>
</table>

Maxwell James Dunne – Spring 2015
CMPE-013/L: "C" Programming
Arrays of Pointers

Initialization

• A pointer array element may be initialized just like its ordinary variable counterpart:

```
p[0] = &x;
```

• Or, when working with strings:

```
p[0] = "My string";
```
Arrays of Pointers
Different from Two-dimensional Array

```
char p[4][] = {
    "On",
    "Off",
    "Main",
    "Aux"
};
```

- This creates an two-dimensional array of chars
  - Amount of memory for every string the same
Arrays of Pointers

Array Elements are Sequential

32-bit Data Memory (RAM)

- p[0]: 0000 6E4F
- p[1]: 6666 4F00
- p[2]: 614D 0000
- p[3]: 4100 6E69

Locations:
- 3FA4-3FA8: On
- 3FA9-3FAD: Off
- 3FAE-3FB2: Main
- 3FB3-3FB7: Aux
Arrays of Pointers

Dereferencing

• To use the value pointed to by a pointer array element, just dereference it like you would an ordinary variable:

\[ y = *p[0]; \]

• Using \( *p[0] \) is the same as using the object it points to, such as \( x \) or the string literal "My String" from the previous slide.
Arrays of Pointers

Accessing Strings

Example

```c
int i = 0;
char *str[] = {"Zero", "One", "Two", "Three", "Four", \0};

int main(void)
{
    while(*str[i] != \0) {
        printf("%s\n", str[i++]);
    }
    while(1);
}
```
Dynamic Memory

malloc()
free()
Dynamic Memory

Rationale

- Memory needs not known at compile time
- Memory needs to persist outside of current scope
Dynamic Memory

`malloc()`

**Syntax**

```c
void *malloc(size_t size);
```

- Request memory of `size` bytes
  - Usually returned by `sizeof` operator
- Returns valid pointer or NULL

**Example**

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

Complex *x = malloc(sizeof(Complex));
```
Dynamic Memory

malloc()d memory

Example

typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
printf("Complex{re:%f im:%f}\n",
        x->re, x->im);
Dynamic Memory

malloc()d memory

Example

typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
x->re = 0.0;
x->im = 0.0;
printf("Complex{re:%f im:%f}\n", 
x->re, x->im);
typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
Dynamic Memory

The Heap

Example

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

Complex *x = malloc(sizeof(Complex));
Complex *x = malloc(sizeof(Complex));
Complex *x = malloc(sizeof(Complex));
Complex *x = malloc(sizeof(Complex));
Complex *x = malloc(sizeof(Complex));
Complex *x = malloc(sizeof(Complex));
```

Heap (top)

NULL
typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
x->re = 0.0;
x->im = 0.0;
printf("Complex\{re:%f im:%f\}\n", x->re, x->im);
Dynamic Memory

NULL pointers

Example

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

Complex *x = malloc(sizeof(Complex));
Complex y = *x;
```
typedef struct {
    float re;
    float im;
} Complex;

Complex *x = malloc(sizeof(Complex));
if (x) {
    x->re = 0.0;
}

x->im = 0.0;
printf("Complex{re:%f im:%f}\n", x->re, x->im);
Dynamic Memory

NULL pointers

Example

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

Complex *x = malloc(sizeof(Complex));
if (x) {
    x->re = 0.0;
    x->im = 0.0;
    printf("Complex\{re:%f \ im:%f\}\n", x->re, x->im);
}
```
Dynamic Memory

`free()`

**Syntax**

```c
void free(void *ptr);
```

- Frees memory pointed to by `ptr`
  - **Must** have been returned by `malloc()`

**Example**

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

Complex *x = malloc(sizeof(Complex));
free(x);
```
Dynamic Memory

Invalid free()ing

Example

```c
// Non-initialized pointers
Complex *x;
free(x); // Invalid!

// NULL pointers
Complex *y = NULL;
free(y); // Invalid!

// Non-heap pointers
char *z = "Hey!";
free(z); // Invalid!

// Heap pointers not returned by malloc()
Complex *w = malloc(sizeof(Complex));
free(&w->re); // Invalid!
```
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);
```

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 3F54</td>
</tr>
<tr>
<td>0x3F58</td>
<td>0000 3F58</td>
</tr>
<tr>
<td>0x3F5C</td>
<td>0000 0000</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)
{
    *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 unspecifed, 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

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

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

Example

```
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**

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

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

**Example**

```c
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, ..., label_n}
```

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

**Example**

```
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} varname₁,…;
  ```

- Declared independently:
  
  Syntax
  
  ```
  enum typeName varName₁,…,varNameₙ;
  ```

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:

  Syntax:

  ```c
  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:

```c
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

```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,
    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)