EECS 2031 3.0 A Software Tools

Week 3: September 19, 2018

NB: After this week 1/4 of course has been covered.

Some C operators

- Arithmetic
  - +,-,*,/,%
- Relational - return an int (0 or not zero)
  - <, >, ==, !=, >=, <=
- Logical - return an int (0 or not zero)
  - &&, ||, !
- Binary
  - &, |, ~, <<, >>, ^

	Operator	Associativity	Precedence
()	Function call	Left-to-Right	Highest 14
[]	Array subscript		
.	Dot (Member of structure)		
->	Arrow (Member of structure)		
!	Logical NOT	Right-to-Left	13
~	One's-complement		
-	Unary minus (Negation)		
++	Increment		
--	Decrement		
&	Address-of		
*	Indirection		
(type)	Cast		
sizeof	Sizeof		
*	Multiplication	Left-to-Right	12
/	Division		
%	Modulus (Remainder)		
+	Addition	Left-to-Right	11
-	Subtraction		
<<	Left-shift	Left-to-Right	10
>>	Right-shift		
<	Less than	Left-to-Right	8
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equal to	Left-to-Right	8
!=	Not equal to		
&	Bitwise AND	Left-to-Right	7
^	Bitwise XOR	Left-to-Right	6
|	Bitwise OR	Left-to-Right	5
&&	Logical AND	Left-to-Right	4
||	Logical OR	Left-to-Right	3
? :	Conditional	Right-to-Left	2
=, += *=, etc.	Assignment operators	Right-to-Left	1
,	Comma	Left-to-Right	Lowest 0

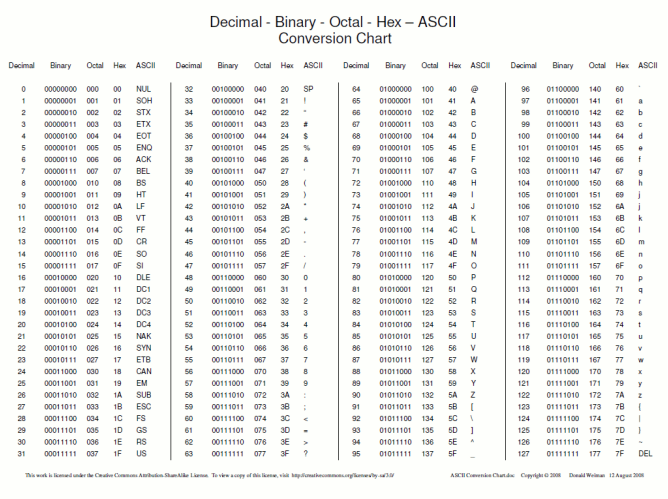
Table 5.1: Precedence and Associativity Table

What is a string?

(There is no String type in C)

In C

- A ‘string’ is an array of characters terminated by a null (0) character.
- Characters are single bytes.
  - The syntax ‘a’ defines a signed int with the value 97.
- The syntax “abcde” is a short form for an array of char’s of length 6 with the symbols ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘\0’ in it.



Decimal - Binary - Octal - Hex – ASCII
Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	j
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Gotcha’s

- Unlike Java, C does not specify order of evaluation completely
  - z = f() + g() : which is evaluated first, f() or g()?
- Conversions with unsigned numbers are often subtle
  - -1L > 1UL ?
- Use of -Wall is (almost) never wrong as it will alert you to some of these features.

Control structures

- Very similar to Java
- Repetition
  - for, while, do … while
- Selection
  - if, if else, switch
- break, continue

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i;
  for(i=0;i<10;i++)
    printf("hello world\n");
  return 0;
}
```

For loop

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i=0;
  while(i<10) {
    printf("hello world\n");
    i++;
  }
  return 0;
}
```

While loop

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i=0;
  do {
    printf("hello world\n");
    i++;
  } while(i<10);
  return 0;
}
```

Do … while

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i=0;
  if(i < 10)
    printf("hello world\n");
  return 0;
}
```

If

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i=0;
  if(i < 10)
    printf("hello world\n");
  else
    printf("goodbye world\n");
  return 0;
}
```

If else

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i=11;
  if(i < 10)
    if(i < 4)
      printf("hello world\n");
  else
    printf("foobar\n");
  return 0;
}
```

Dangling Else (don't do this)

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i=11;
  if(i < 10) {
    if(i < 4)
      printf("hello world\n");
  } else
    printf("foobar\n");
  return 0;
}
```

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i=11;
  if(i < 10) {
    if(i < 4) {
      printf("hello world\n");
    } else
      printf("foobar\n");
  }
  return 0;
}
```

Say what you mean

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i=11;
  switch(i) {
  case 0:
  case 1:
    printf("0 or 1\n");
    break;
  case 2:
    printf ("2\n");
    break;
  case 3:
    printf("3\n");
  case 4:
    printf("4\n");
    break;
  default:
    printf("default\n");
  }
  return 0;
}
```

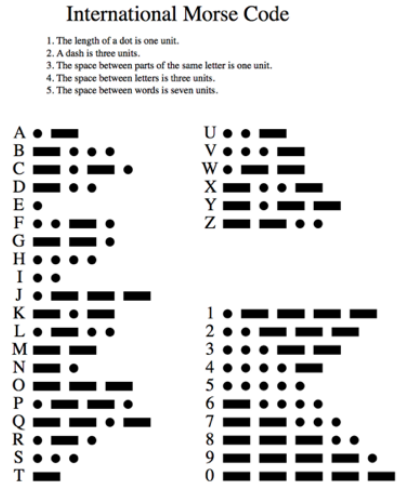
switch/case/default

break, continue

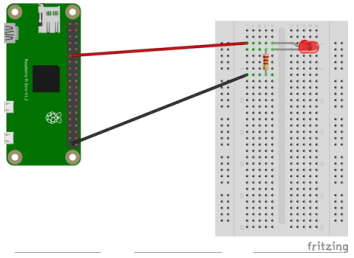
- In loops
  - Break - break out of the loop
  - Continue - continue with next loop iteration
- In switch
  - Break out of the switch

Lab 03

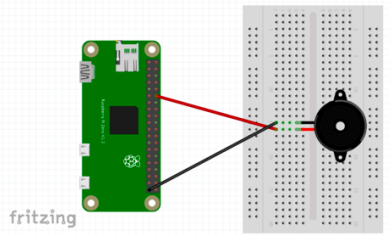
- Morse Code
- Multiple source files
- Use of GitHub to distribute files to you
- Use of Makefiles to automate program building



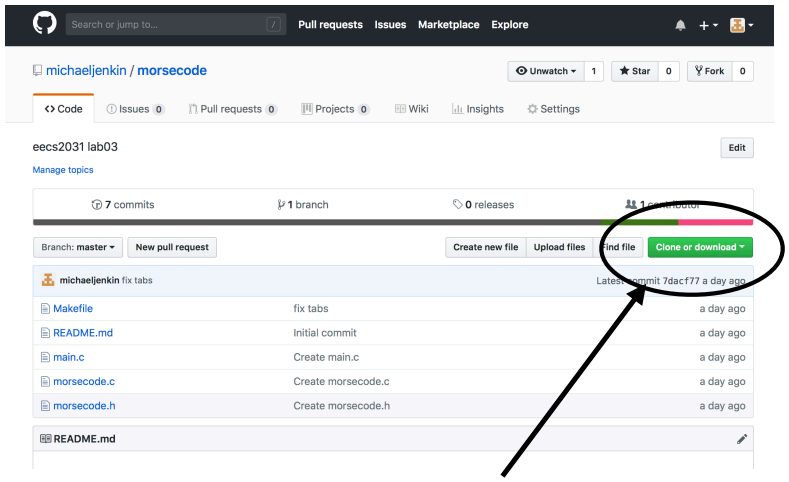
Program takes arguments and outputs them in Morse code



Light a LED



Make a sound



Obtain a URL of a clone of the git repository

```
wandererreecsyorkuca:t jenkin$ git clone https://github.com/michaeljenkin/morsecode.git
Cloning into 'morsecode'...
remote: Counting objects: 21, done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 21 (delta 6), reused 5 (delta 2), pack-reused 0
Unpacking objects: 100% (21/21), done.
wandererreecsyorkuca:t jenkin$
```

Clone(s) the archive into current directory

```
wandererreecsyorkuca:t jenkin$ clear

wandererreecsyorkuca:t jenkin$ pwd
wandererreecsyorkuca:t jenkin$ git clone https://github.com/michaeljenkin/morsecode.git
Cloning into 'morsecode'...
remote: Counting objects: 21, done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 21 (delta 6), reused 5 (delta 2), pack-reused 0
Unpacking objects: 100% (21/21), done.
wandererreecsyorkuca:t jenkin$ ls
morsecode
wandererreecsyorkuca:t jenkin$ cd morsecode
wandererreecsyorkuca:morsecode jenkin$ ls
Makefile        README.md       main.c          morsecode.c     morsecode.h
wandererreecsyorkuca:morsecode jenkin$
```

We will do more about git (and GitHub) in the following weeks
But for now — a tool to download code associated with the lab

```
#include <stdio.h>
#include <stdlib.h>
#include "morsecode.h"

int main(int argc, char *argv[])
{
  char *p;
  if(argc != 2) {
    fprintf(stderr,"usage: %s text\n", argv[0]);
    exit(1);
  }
  init_morse();
  play_buzzer(500);
  wait_word();
  p = argv[1];
  while(*p) {
    if(*p == ' ')
      wait_word();
    else {
      char *v = char2morse(*p);
      printf("character %c translates to %s\n", *p, v);
      while(*v) {
        switch(*v) {
        case '*' : send_dot(); break;
        case '=' : send_dash(); break;
        default:
          /*NOTREACHED*/
          fprintf(stderr,"internal logic error\n");
          exit(1);
        }
        if(*(v+1) != '\0')
          wait_dot();
        else if(*(p+1) == ' ')
          wait_word();
        else if(*(p+1) != '\0')
          wait_letter();
        v++;
      }
    }
    p++;
  }
  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include "morsecode.h"

int main(int argc, char *argv[])
{
  char *p;
  if(argc != 2) {
    fprintf(stderr,"usage: %s text\n", argv[0]);
    exit(1);
  }
  init_morse();
  play_buzzer(500);
  wait_word();
  p = argv[1];
  while(*p) {
    if(*p == ' ')
      wait_word();
    else {
      char *v = char2morse(*p);
      printf("character %c translates to %s\n", *p, v);
      while(*v) {
        switch(*v) {
        case '*' : send_dot(); break;
        case '=' : send_dash(); break;
        default:
          /*NOTREACHED*/
          fprintf(stderr,"internal logic error\n");
          exit(1);
        }
        if(*(v+1) != '\0')
          wait_dot();
        else if(*(p+1) == ' ')
          wait_word();
        else if(*(p+1) != '\0')
          wait_letter();
        v++;
      }
    }
    p++;
  }
  return 0;
}
```

Deal with command line arguments

```c
#include <stdio.h>
#include <stdlib.h>
#include "morsecode.h"

int main(int argc, char *argv[])
{
  char *p;
  if(argc != 2) {
    fprintf(stderr,"usage: %s text\n", argv[0]);
    exit(1);
  }
  init_morse();
  play_buzzer(500);
  wait_word();
  p = argv[1];
  while(*p) {
    if(*p == ' ')
      wait_word();
    else {
      char *v = char2morse(*p);
      printf("character %c translates to %s\n", *p, v);
      while(*v) {
        switch(*v) {
        case '*' : send_dot(); break;
        case '=' : send_dash(); break;
        default:
          /*NOTREACHED*/
          fprintf(stderr,"internal logic error\n");
          exit(1);
        }
        if(*(v+1) != '\0')
          wait_dot();
        else if(*(p+1) == ' ')
          wait_word();
        else if(*(p+1) != '\0')
          wait_letter();
        v++;
      }
    }
    p++;
  }
  return 0;
}
```

Some initialization

```c
#include <stdio.h>
#include <stdlib.h>
#include "morsecode.h"

int main(int argc, char *argv[])
{
  char *p;
  if(argc != 2) {
    fprintf(stderr,"usage: %s text\n", argv[0]);
    exit(1);
  }
  init_morse();
  play_buzzer(500);
  wait_word();
  p = argv[1];
  while(*p) {
    if(*p == ' ')
      wait_word();
    else {
      char *v = char2morse(*p);
      printf("character %c translates to %s\n", *p, v);
      while(*v) {
        switch(*v) {
        case '*' : send_dot(); break;
        case '=' : send_dash(); break;
        default:
          /*NOTREACHED*/
          fprintf(stderr,"internal logic error\n");
          exit(1);
        }
        if(*(v+1) != '\0')
          wait_dot();
        else if(*(p+1) == ' ')
          wait_word();
        else if(*(p+1) != '\0')
          wait_letter();
        v++;
      }
    }
    p++;
  }
  return 0;
}
```

For every character in the argument

```c
#include <stdio.h>
#include <stdlib.h>
#include "morsecode.h"

int main(int argc, char *argv[])
{
  char *p;
  if(argc != 2) {
    fprintf(stderr,"usage: %s text\n", argv[0]);
    exit(1);
  }
  init_morse();
  play_buzzer(500);
  wait_word();
  p = argv[1];
  while(*p) {
    if(*p == ' ')
      wait_word();
    else {
      char *v = char2morse(*p);
      printf("character %c translates to %s\n", *p, v);
      while(*v) {
        switch(*v) {
        case '*' : send_dot(); break;
        case '=' : send_dash(); break;
        default:
          /*NOTREACHED*/
          fprintf(stderr,"internal logic error\n");
          exit(1);
        }
        if(*(v+1) != '\0')
          wait_dot();
        else if(*(p+1) == ' ')
          wait_word();
        else if(*(p+1) != '\0')
          wait_letter();
        v++;
      }
    }
    p++;
  }
  return 0;
}
```

Its a blank, wait

```c
#include <stdio.h>
#include <stdlib.h>
#include "morsecode.h"

int main(int argc, char *argv[])
{
  char *p;
  if(argc != 2) {
    fprintf(stderr,"usage: %s text\n", argv[0]);
    exit(1);
  }
  init_morse();
  play_buzzer(500);
  wait_word();
  p = argv[1];
  while(*p) {
    if(*p == ' ')
      wait_word();
    else {
      char *v = char2morse(*p);
      printf("character %c translates to %s\n", *p, v);
      while(*v) {
        switch(*v) {
        case '*' : send_dot(); break;
        case '=' : send_dash(); break;
        default:
          /*NOTREACHED*/
          fprintf(stderr,"internal logic error\n");
          exit(1);
        }
        if(*(v+1) != '\0')
          wait_dot();
        else if(*(p+1) == ' ')
          wait_word();
        else if(*(p+1) != '\0')
          wait_letter();
        v++;
      }
    }
    p++;
  }
  return 0;
}
```

Not a blank

```c
#include <stdio.h>
#include <stdlib.h>
#include "morsecode.h"

int main(int argc, char *argv[])
{
  char *p;
  if(argc != 2) {
    fprintf(stderr,"usage: %s text\n", argv[0]);
    exit(1);
  }
  init_morse();
  play_buzzer(500);
  wait_word();
  p = argv[1];
  while(*p) {
    if(*p == ' ')
      wait_word();
    else {
      char *v = char2morse(*p);
      printf("character %c translates to %s\n", *p, v);
      while(*v) {
        switch(*v) {
        case '*' : send_dot(); break;
        case '=' : send_dash(); break;
        default:
          /*NOTREACHED*/
          fprintf(stderr,"internal logic error\n");
          exit(1);
        }
        if(*(v+1) != '\0')
          wait_dot();
        else if(*(p+1) == ' ')
          wait_word();
        else if(*(p+1) != '\0')
          wait_letter();
        v++;
      }
    }
    p++;
  }
  return 0;
}
```

Obtain Morse code

*** <- dot**

= <- dash

```c
#include <stdio.h>
#include <stdlib.h>
#include "morsecode.h"

int main(int argc, char *argv[])
{
  char *p;
  if(argc != 2) {
    fprintf(stderr,"usage: %s text\n", argv[0]);
    exit(1);
  }
  init_morse();
  play_buzzer(500);
  wait_word();
  p = argv[1];
  while(*p) {
    if(*p == ' ')
      wait_word();
    else {
      char *v = char2morse(*p);
      printf("character %c translates to %s\n", *p, v);
      while(*v) {
        switch(*v) {
        case '*' : send_dot(); break;
        case '=' : send_dash(); break;
        default:
          /*NOTREACHED*/
          fprintf(stderr,"internal logic error\n");
          exit(1);
        }
        if(*(v+1) != '\0')
          wait_dot();
        else if(*(p+1) == ' ')
          wait_word();
        else if(*(p+1) != '\0')
          wait_letter();
        v++;
      }
    }
    p++;
  }
  return 0;
}
```

Process character

Most complex art of the logic is due to the blanks and their delay

```c
#include <stdio.h>
#include <stdlib.h>
#include "morsecode.h"

int main(int argc, char *argv[])
{
  char *p;
  if(argc != 2) {
    fprintf(stderr,"usage: %s text\n", argv[0]);
    exit(1);
  }
  init_morse();
  play_buzzer(500);
  wait_word();
  p = argv[1];
  while(*p) {
    if(*p == ' ')
      wait_word();
    else {
      char *v = char2morse(*p);
      printf("character %c translates to %s\n", *p, v);
      while(*v) {
        switch(*v) {
        case '*' : send_dot(); break;
        case '=' : send_dash(); break;
        default:
          /*NOTREACHED*/
          fprintf(stderr,"internal logic error\n");
          exit(1);
        }
        if(*(v+1) != '\0')
          wait_dot();
        else if(*(p+1) == ' ')
          wait_word();
        else if(*(p+1) != '\0')
          wait_letter();
        v++;
      }
    }
    p++;
  }
  return 0;
}
```

Code defined elsewhere

```c
#include <stdio.h>
#include <wiringPi.h>
#include "morsecode.h"

#define TIME_UNIT 250
#define DOT_TIME (TIME_UNIT)
#define DASH_TIME (TIME_UNIT*3)
#define LETTER_SPACE_TIME (TIME_UNIT*3)
#define WORD_SPACE_TIME (TIME_UNIT*7)


void init_morse(void)
{
}

void send_dot(void)
{
}

void send_dash(void)
{
}

void wait_letter(void)
{
  printf("wait for letter\n");
}

void wait_dot(void)
{
  printf("wait for dot\n");
}

void wait_word(void)
{
  printf("wait for word\n");
}

void play_buzzer(int msec)
{
}

char *char2morse(char c)
{
  return "***===***";
}
```

morsecode.c

```c
void init_morse(void);
void send_dot(void);
void send_dash(void);
void wait_letter(void);
void wait_word(void);
void wait_dot(void);
void play_buzzer(int msec);
char *char2morse(char c);
```

morsecode.h

Makefile's

- Input to the make command
- There are two primary kinds of statements in a make file
  - A macro definition (CC=gcc)
  - A dependency rule

hello: hello.c hello.h
cc hello -o hello

```
OBJS=main.o morsecode.o
LDFLAGS=-lwiringPi
CC=gcc
CFLAGS=-Wall -pedantic

%.o:  %.c morsecode.h
  $(CC) $(CFLAGS) -c -ansi $<


main: ${OBJS}
  $(CC) $(CFLAGS) ${OBJS} -o main $(LDFLAGS)
```

Makefile

(make will make it happen)

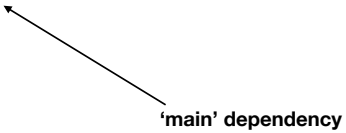
```
OBJS=main.o morsecode.o
LDFLAGS=-lwiringPi
CC=gcc
CFLAGS=-Wall -pedantic

%.o:  %.c morsecode.h
  $(CC) $(CFLAGS) -c -ansi $<


main: ${OBJS}
  $(CC) $(CFLAGS) ${OBJS} -o main $(LDFLAGS)
```

Macros

```
OBJS=main.o morsecode.o
LDFLAGS=-lwiringPi
CC=gcc
CFLAGS=-Wall -pedantic

%.o:  %.c morsecode.h
  $(CC) $(CFLAGS) -c -ansi $<


main: ${OBJS}
  $(CC) $(CFLAGS) ${OBJS} -o main $(LDFLAGS)
```

A dependency

```
OBJS=main.o morsecode.o
LDFLAGS=-lwiringPi
CC=gcc
CFLAGS=-Wall -pedantic

%.o:  %.c morsecode.h
  $(CC) $(CFLAGS) -c -ansi $<

main: ${OBJS}
  $(CC) $(CFLAGS) ${OBJS} -o main $(LDFLAGS)
```



'main' dependency

Makefile

- You have to encode the dependencies
  - Unlike other languages (e.g., Java) you have to apply the smarts
- You can write incomprehensible Makefiles. Or not.

```
OBJS=main.o morsecode.o
LDFLAGS=-lwiringPi
CC=gcc
CFLAGS=-Wall -pedantic

%.o:  %.c morsecode.h
  $(CC) $(CFLAGS) -c -ansi $<

main: ${OBJS}
  $(CC) $(CFLAGS) ${OBJS} -o main $(LDFLAGS)
```

To make a file foo.o, this depends on foo.c and identifiers.h and g_identifiers.h
This dependency can be addressed by $(CC) $(CFLAGS) -c ansi foo.c
$@ expands to LHS, $< to first prerequisite (here foo.c)

Makefiles

- Are often built (quasi) automatically in large applications as certain definitions are system dependent
  - ./configure
  - make
- Is a common build sequence.

Summary

- Keep up with the reading
- Do the lab
- Starting next week we move on to IoT with your Raspberry Pi
- Think about your user name