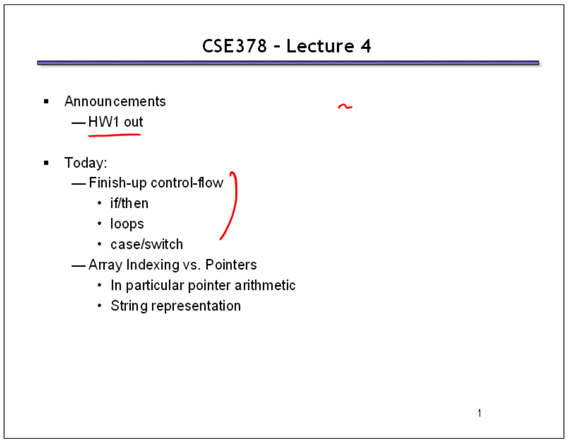
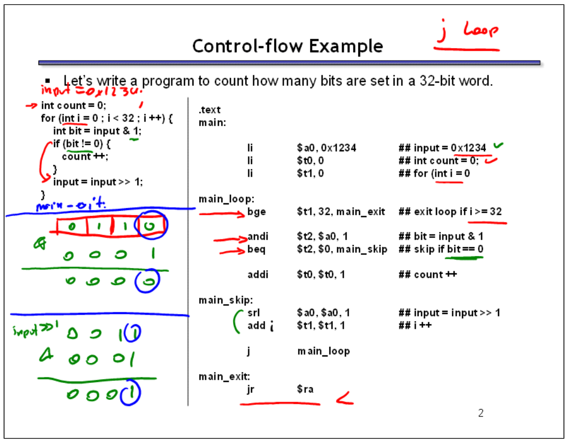


CSE378 - Lecture 4

- Announcements
  - HW1 out
- Today:
  - Finish-up control-flow
    - if/then
    - loops
    - case/switch
  - Array Indexing vs. Pointers
    - In particular pointer arithmetic
    - String representation



Control-flow Example

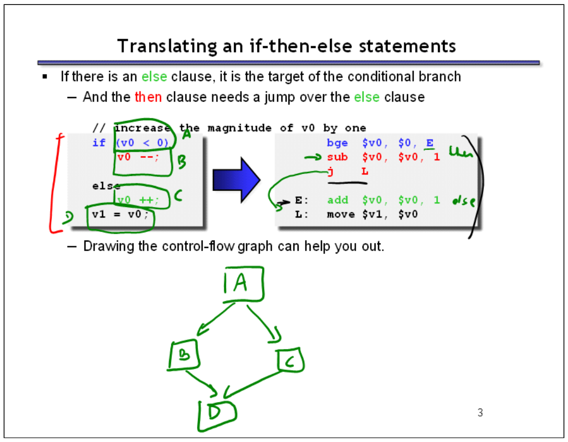
j Loop

- Let's write a program to count how many bits are set in a 32-bit word.

input = 0x1234.

```
int count = 0;
for (int i = 0 ; i < 32 ; i ++) {
   int bit = input & 1;
   if (bit != 0) {
      count ++;
   }
   input = input >> 1;
}
```

```
.text
main:

        li      $a0, 0x1234         ## input = 0x1234
        li      $t0, 0              ## int count = 0;
        li      $t1, 0              ## for (int i = 0

main_loop:
        bge     $t1, 32, main_exit  ## exit loop if i >= 32

        andi    $t2, $a0, 1         ## bit = input & 1
        beq     $t2, $0, main_skip  ## skip if bit == 0

        addi    $t0, $t0, 1         ## count ++

main_skip:
        srl     $a0, $a0, 1         ## input = input >> 1
        add     $t1, $t1, 1         ## i ++

        j       main_loop

main_exit:
        jr      $ra
```



Translating an if-then-else statements

- If there is an else clause, it is the target of the conditional branch
  - And the then clause needs a jump over the else clause

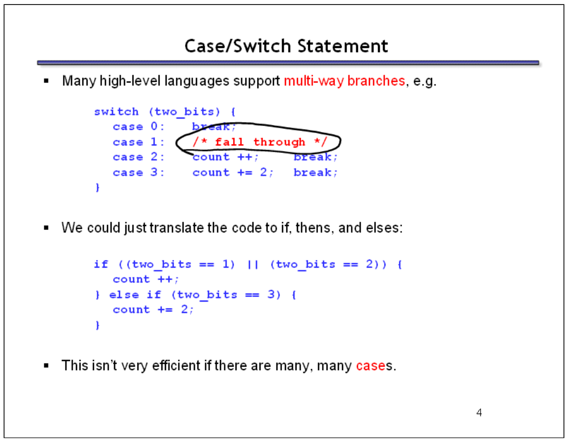
```
// increase the magnitude of v0 by one
if (v0 < 0)
      v0 --;

else
      v0 ++;
v1 = v0;
```

```
      bge   $v0, $0, E
      sub   $v0, $v0, 1
      j     L

E:    add   $v0, $v0, 1
L:    move  $v1, $v0
```

A
B
C
D
then
else

  - Drawing the control-flow graph can help you out.

A
B
C
D



Case/Switch Statement

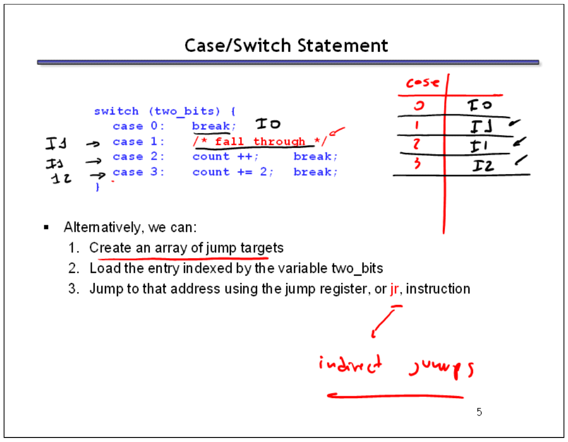
- Many high-level languages support multi-way branches, e.g.

```
switch (two_bits) {
   case 0:     break;
   case 1:     /* fall through */
   case 2:     count ++;      break;
   case 3:     count += 2;    break;
}
```

- We could just translate the code to if, thens, and elses:

```
if ((two_bits == 1) || (two_bits == 2)) {
   count ++;
} else if (two_bits == 3) {
   count += 2;
}
```

- This isn't very efficient if there are many, many cases.



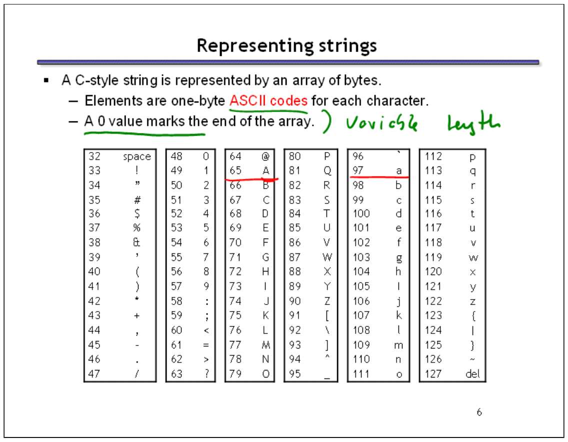
Case/Switch Statement

```
switch (two_bits) {
   case 0:     break;         I0
   case 1:     /* fall through */
   case 2:     count ++;      break;
   case 3:     count += 2;    break;
}
```

case	
0	I0
1	I1
2	I1
3	I2

- Alternatively, we can:
  1. Create an array of jump targets
  2. Load the entry indexed by the variable two_bits
  3. Jump to that address using the jump register, or jr, instruction

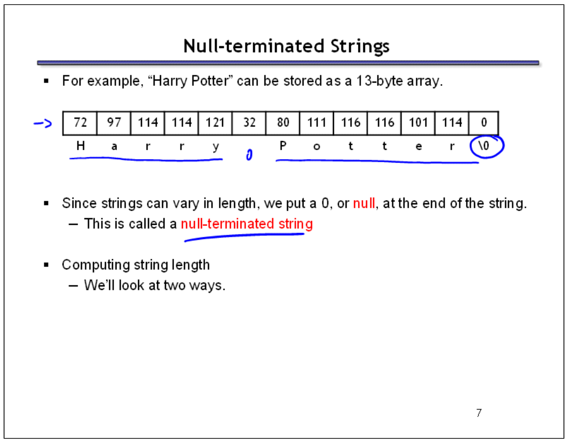
indirect jumps



Representing strings

- A C-style string is represented by an array of bytes.
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array.) variable length

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

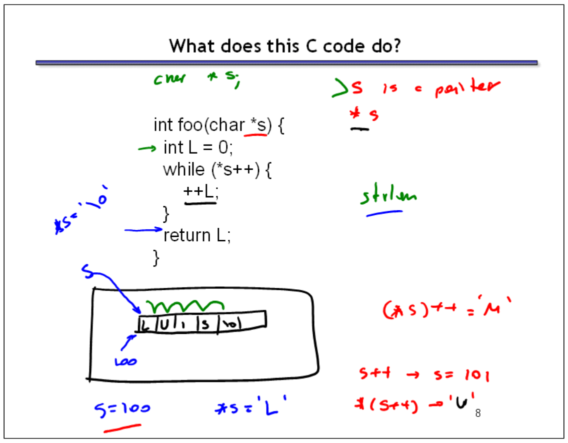


Null-terminated Strings

- For example, “Harry Potter” can be stored as a 13-byte array.

72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y	0	P	o	t	t	e	r	\0

- Since strings can vary in length, we put a 0, or null, at the end of the string.
  - This is called a null-terminated string
- Computing string length
  - We’ll look at two ways.



What does this C code do?

char *s;

```
int foo(char *s) {
  int L = 0;
  while (*s++) {
    ++L;
  }
  return L;
}
```

*s='\0'
s
L
U
i
s
\0
100

>s is a pointer

*s

strlen

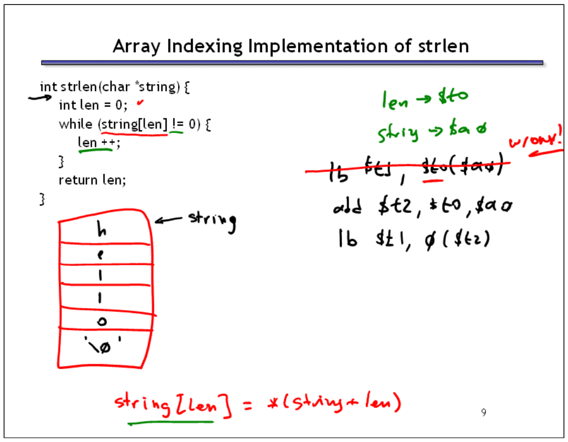
(*s)++ = 'M'

s++ → s= 101

*(s++) → 'U'

s=100

*s='L'



Array Indexing Implementation of strlen

```
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len ++;
    }
    return len;
}
```

h
e
l
l
o
'\0'
string

len → $t0

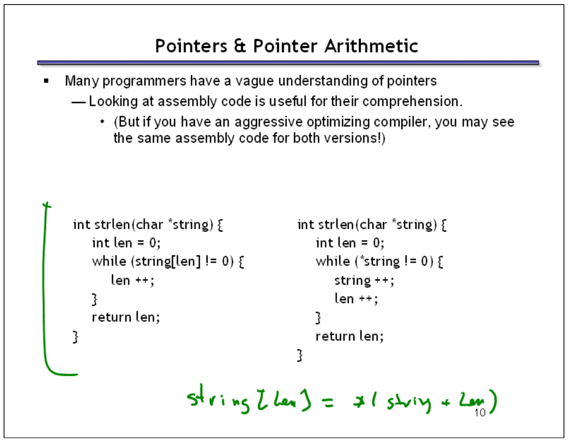
string → $a0

~~lb $t1, $t0($a0)~~ wrong!

add $t2, $t0, $a0

lb $t1, 0($t2)

string[len] = *(string + len)



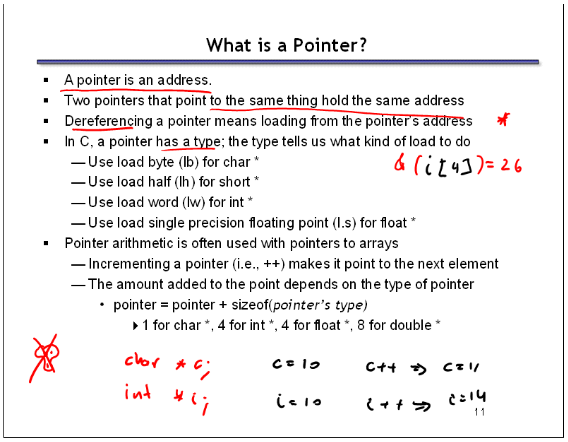
Pointers & Pointer Arithmetic

- Many programmers have a vague understanding of pointers
  - Looking at assembly code is useful for their comprehension.
    - (But if you have an aggressive optimizing compiler, you may see the same assembly code for both versions!)

```
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len ++;
    }
    return len;
}
```

```
int strlen(char *string) {
    int len = 0;
    while (*string != 0) {
        string ++;
        len ++;
    }
    return len;
}
```

string[len] = *(string + len)



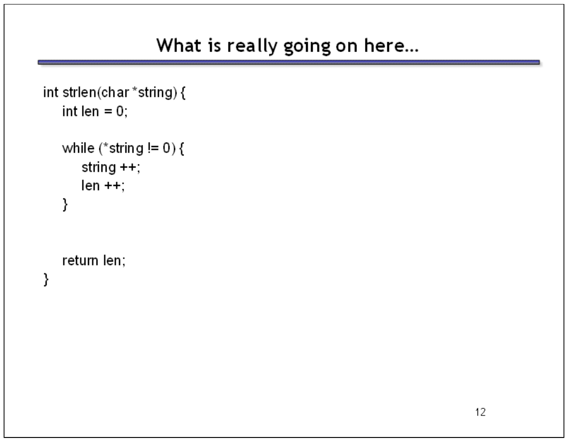
What is a Pointer?

- A pointer is an address.
- Two pointers that point to the same thing hold the same address
- Dereferencing a pointer means loading from the pointer’s address *
- In C, a pointer has a type; the type tells us what kind of load to do
  - Use load byte (lb) for char *
  - Use load half (lh) for short *
  - Use load word (lw) for int *
  - Use load single precision floating point (l.s) for float *
- Pointer arithmetic is often used with pointers to arrays
  - Incrementing a pointer (i.e., ++) makes it point to the next element
  - The amount added to the point depends on the type of pointer
    - pointer = pointer + sizeof(*pointer’s type)*
      - 1 for char *, 4 for int *, 4 for float *, 8 for double *

&(i[4])=26

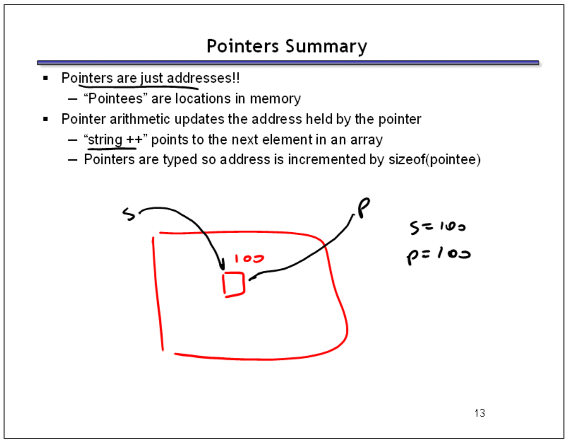
char *c; c=10 c++ ⇒ c=11

int *i; i=10 i++ ⇒ i=14



What is really going on here…

```
int strlen(char *string) {
    int len = 0;

    while (*string != 0) {
        string ++;
        len ++;
    }


    return len;
}
```



Pointers Summary

- Pointers are just addresses!!
  - "Pointees" are locations in memory
- Pointer arithmetic updates the address held by the pointer
  - "string ++" points to the next element in an array
  - Pointers are typed so address is incremented by sizeof(pointee)

13