Integer Decoding

- Binary to Decimal
- Unsigned = simple binary = B2U
  - 0101 = 5, 1111 = F, 1110 = E, 1001 = 9
- Signed = two's complement = B2T
  - 0 101 = positive number; same as B2U = 5
  - 1 111 = $-1*2^3 + 7 = -8 + 7 = -1$
  - 1 110 = $-1*2^3 + 6 = -8 + 6 = -2$
  - 1 001 = $-1*2^3 + 1 = -8 + 1 = -7$
  - Another way, if sign bit = 1, then it's a negative number and to get the magnitude of that number, you:
    - invert bits and add 1
  - Reminder: left-most bit is sign bit

CODE	B2U	B2T	HEX
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	**8**	**-8**	**8**
1001	9	-7	9
1010	10	-6	A
1011	11	-5	B
1100	12	-4	C
1101	13	-3	D
1110	14	-2	E
1111	15	-1	F

Converting Hexadecimal to Binary

- Suppose you are given the hexadecimal number : 0x173A
- Convert to binary format by expanding each hexadecimal digit, as follows:
  - Hexadecimal 1 7 3 A
  - Binary 0001 0111 0011 1010

- This gives the binary representation:
  - 0001011100111010
  - Putting spaces between each 4 digits for readability in this class is acceptable (and, sometimes, downright needful to keep your eyes from crossing).

Converting Binary to Hexadecimal

- Suppose you are given the binary number :
  1111001010101101
- First, split the number in to groups of 4 bits each, starting from the binary point (note, that if the total number of bits is not a multiple of 4, the leftmost group should have the one with fewer bits):
  - Binary 11 1100 1010 1101
  - Hexadecimal 3 C A D
- Then translate each 4 bit group to the corresponding hexadecimal digit.

Converting from decimal to another base

- To convert from decimal to base b, divide the decimal number by b, and write the remainders (which will be between 0 and b – 1), until the quotient is zero.
- Then, write the remainders in order from the last remainder obtained to the first remainder obtained.
- Example: convert $(221)_{10}$ to base 2 (binary) (see the next slide)

Example: Convert $(221)_{10}$ to binary

	Quotient	Remainder
2	221	1
2	110	0
2	55	1
2	27	1
2	13	1
2	6	0
2	3	1
2	1	1
	0	

Now, if we write the remainders in order from the last one obtained, to the first one obtained, we have: $(221)_{10} = (11011101)_2$

Thus, this expresses the binary representation of the original decimal number.

Binary to Hex correspondence

4 binary digits can be converted to a single hex digit, as shown below.

Binary	Hex	Binary	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Example: Convert $(743)_{10}$ to hexadecimal

Note: Since remainders can have values from 0 to b - 1, that is, 0 to 15, we use F for 15, E for 14, D for 13, C for 12, B for 11, and A for 10

	Quotient	Remainder
16	743	7
16	46	E
16	2	2
	0	

Therefore, if we write the remainders in order from the last one obtained to the first one obtained, we have:

$$(743)_{10} = (2E7)_{16}$$

To convert from base b to decimal

- We can do the conversion by multiplying the value of each digit, d_m, by d^m, for m from 0 (least significant digit) to n-1 (most significant digit) for an n digit number, and by summing all the products obtained in this way.

Example of conversion from base b to decimal

- Problem: Convert $(5377)_8$ to decimal.
- Solution:

$$
\begin{aligned}
(5377)_8 &= 5 * 8^3 + 3 * 8^2 + 7 * 8^1 + 7 * 8^0 \\
&= 5 * 512 + 3 * 64 + 7 * 8 + 7 * 1 \\
&= 2560 + 192 + 56 + 7 \\
&= 2815
\end{aligned}
$$

Number conversions

- These few slides give the basics.
- We'll get a bit more involved in the 2nd half of the semester

Logical Operators

<< left-shift
>> right-shift
& bitwise AND
| bitwise OR
^ bitwise exclusive-OR

← wait for these… later

&& logical AND ← logical operators
|| logical OR

< less than
> greater than
<= less than or equal
>= greater than or equal
== equals
!= does not equal

← relational operators

BITWISE OPERATORS:
0 is 0
1 is 1

LOGICAL OPERATORS:
Non-zero -> TRUE
-or-
0 is FALSE
Everything else is TRUE

Result: 0 if false;
1 if true

Note 1: C has no such keywords as **true** or **false**
Note 2: 'OR', 'AND' , and 'NOT': There are differences between bitwise and logical operators.

Logical versus Bitwise Operators example

	int a = 10; int c = 0;	int a = 10; int c = 1;	int a = 10; int c = 3;
a & c Bitwise	a=10=1010b c= 0=0000b ---------- 0000b 0	a=10=1010b c= 1=0001b ---------- 0000b 0	a=10=1010b c= 3=0011b ---------- 0010b 2
a && c Logical	a=True c=False ---------- False	a=True c=True ---------- True	a=True c=True ---------- True

Bit shifting

- When we represent values in binary, we can do what is called “shifting” bits either to the right or to the left.

- Left shift example:
  - Binary value: 0111 0101
  - Left shift 2 places: 1101 0100 (fill with 0’s)

Bit Shifting

- Shifting to the right has 2 options:
  - Arithmetic shift
  - Logical shift
- Shift Right Arithmetic
  - Fills in from the left with copy of Most Significant Bit
  - Example: Binary value: 1111 0101
    - Shift Right Arithmetic 1 bit: 1111 1010
    - Shift Right Arithmetic 2 bits: 1111 1101
- Shift Right Logical
  - Fills in from the left with 0's
  - Example:Binary value: 1111 0101
    - Shift Right Logical 1 bit: 0111 1010
    - Shift Right Logical 2 bits: 0011 1101

Relational Operators

- Used to compare two values
  - < <= > >= (Higher precedence than == and !=)
  - == != (Higher precedence than bitwise, logical, conditional, and comma operators)
- Precedence order is given above, L-R associativity
- Arithmetic operators have higher precedence than relational operators
- A true evaluates to a nonzero number (generally 1). A false statement evaluates to zero.
  - For example, the expression (0 == 2) evaluates to 0.
  - while the expression (2 == 2) evaluates to a 1
    - (non-zero technically, but usually 1).

Boolean Operators

- ANSI C does not have a distinct Boolean type
  - int is used instead (*usually*, but other types are possible)
- 0 is treated as FALSE
- Non-zero is treated as TRUE

```
i = 0;
while (i - 10) {
    ...
    }
```

  - As long as (i-10) ≠ 0 it is considered true, and the body of the while loop will execute.

(Later versions of C have Boolean type)

Boolean Operators (cont)

- **Short-Circuit Evaluation**: Relational statements stop evaluating once a statement's value is definitive
  - In (x && y), if x == 0 evaluates to true (i.e. 1st condition evaluates to false), evaluation stops
  - It does not matter what the outcome of y==0 is. y is not evaluated or compared with 0
  - Same for OR if first condition evaluates to 1 (true).
- **This can cause buggy code (or not!)**
  - This is a valid way to write code
  - There are many arguments made that it can be a correct and expedient way to write ***some*** code
  - Be very cautious

Boolean Operators (cont)

- **Short-Circuit Evaluation**:

```
func1(float a, float b){
Float func_result = 0;
If ((b !=0) && (a/b< 0.5)){
        printf(" The result of func1 is %f.4\n", a*b + a/b);
}
return;
```

In this example, short-circuit evaluation saves your bacon!
Without short-circuit, this code will seg fault when b=0.

Boolean Operators (cont)

- **Short-Circuit Evaluation**:

```
func1(float a, float b){
Float func_result = 0;
If ((b ==0) && ((func_result = (++a*b+3)))){
        printf(" The result of func1 is %f.4\n",
                                                a*func_result);
}
return;
```

In this example, short-circuit evaluation might cause you problems.

An Example:

```
#include <stdio.h>
main()
{
    int z, a=10, b=0, c=3;
    z = (a > c) ||  (++a > b);
    printf("z = %d  a = %d\n", z , a);
    c = 30;
    z = (a > c) ||  (++a > b);
    printf("z = %d  a = %d\n", z, a);
    a = 10;
    c=3;
    z = ++a * c + a++;
    printf("z = %d  a = %d",z,a);
return 0;
}
```

Results

```
#include <stdio.h>
main()
{
    int z, a=10, b=0, c=3;
    z = (a > c) ||  (++a > b);
    printf("z = %d  a = %d\n", z, a);  /* z=1(true)    a=10 */
   c = 30;
    z = (a > c) ||  (++a > b);
    printf("z = %d  a = %d\n", z, a); /* z= 1(true)   a=11 */
    a = 10;
    c=3;
    z = ++a * c + a++;
    printf("z = %d  a = %d",z,a);  /* z= 44   a=12 */
return 0;
}
```

Arithmetic Type Issues

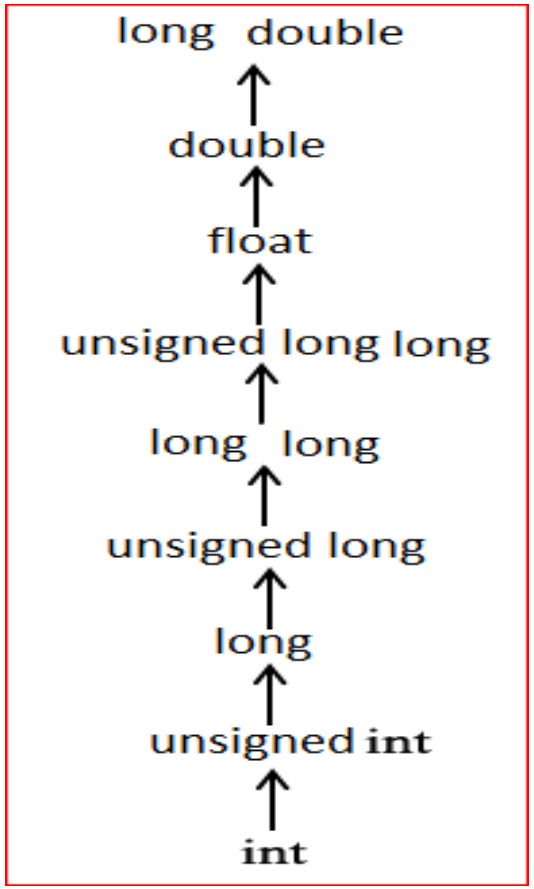
- Type casting – EXPLICIT
  - Purposely converting a variable from one data type to another data type
  - Syntax: (type-name) variable
- Type combination and promotion - IMPLICIT
  - (‘a’ – 32) = 97 – 32 = 65 (if used as a char = ‘A’)
  - Smaller type (char) is “promoted” to be the same size as the larger type (int)
  - Determined at compile time – type of the whole expression is based purely on the types of the values in the expressions
  - **Does not lose information** – convert from type to compatible *larger* type
- Whether the casting is implicit or explicit, the compiler will create separate storage for the cast value, and any operands that are necessary to determine it. [See next slide for example]

C type-casting

The **usual arithmetic conversions** are implicitly performed to cast values of distinct types to a common type.

-Compiler first performs *integer promotion* (promotion of char to int)

-If operands still have different types, then any variables or constants in operand expressions are converted to the type that appears highest in the following hierarchy (except any variables that were already of that type; for those, no conversion is necessary)



Arithmetic Expressions and Casting

Following code is supposed to scale a homework score in the range 0-20 to be in the range 0-100.

```
cnvt_score()
{
   int score;
   /* score gets set in the range 0..20 */
   score = (score / 20) * 100; /*convert to percentage*/
   return(score);
}
```

Does this work?

Arithmetic Expressions and Casting

This does not work! Unfortunately, score will almost always be set to 0 for this code because the integer division in the expression (score/20) will be 0 for every value of score less than 20.

- The fix is to force the quotient to be computed as a floating point number...

```
score = ((double)score / 20) * 100; /*OK - double floating
  point division with explicit cast */

score = (score / 20.0) * 100; /*OK - double floating point
  division with implicit casting because float (double)
  constant 20.0 */

score = (int)(score / 20.0) * 100; /*NO -- the (int)cast
  truncates the floating quotient back to 0 if score < 20 */
```