LECTURE 12: TYPE CONVERSION (K&R § 2.7)

IMPLICIT CONVERSION (AUTOMATIC)

When an operator has operands of different types, they are converted to a common type according to a small number of rules.

Automatic Conversion	
If an expression has operands of different types, result will be of the wider type.	
Function declarations cause automatic coercion of any arguments when the function is called.	
if there are no unsigned operands, the following informal set of rules for implicit conversions will suffice:	
if either operand is:	the other is converted to:
long double	long double
otherwise:	
double	double
otherwise:	
float	float
otherwise:	
char, short	int
then:	
long	long

Conversion rules are more complicated with unsigned operands because comparisons between signed and unsigned values are machine-dependent (because they depend on sizes of int types).

The precise conversion rules are in K&R Appendix A, § 6 (page 197).

Assignment Conversion	
The value of the right side is converted to the type of the left, which is the type of the result.	
longer int to shorter	excess high-order bits discarded value may change
shorter int to longer	new bits filled in according to implementation be careful with sign extension

int i;
char c;
i = c;
c = i;
	value of c is unchanged:
	whatever sign extension happened at the first assignment is discarded at the second.

float to int	truncation of fractional part
int to float	conversion to float

CHAR TO INT CONVERSION EXAMPLES

Common conversion as chars are small ints and can be used in arithmetic expressions.

Example: atoi (K&R, page 43)
ascii to integer

```
int atoi(char s[ ]) {
  int i, n;
  n = 0;
  /* check if each char in string is a digit */
  for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
    n = 10 * n + (s[i] - '0');
  return n;
}
```

s[i]-'0' is a char	when they are added together, the char is converted to an int
10 * n is an int	

Example: lower (K&R, page 43)
convert c to lower case (ASCII only)

```
int lower(int c) {
  if (c >= 'A' && c <= 'Z')
    return c + 'a' - 'A';
  else
    return c;
}
```

return c + 'a' - 'A';	'a' - 'A' is the offset from an uppercase letter to its lowercase
	c + 'a' converts char to int

Example: itoa (K&R, page 64)
integer to ASCII

```
void itoa(int n, char s[]) {
  int i, sign;
  if ((sign = n) < 0) /* record sign */
    n = -n; /* make n positive */
    i = 0;
  do { /* generate digits in reverse order */
    s[i++] = n % 10 + '0'; /* get next digit */
  } while ((n /= 10) > 0); /* delete it */
  if (sign < 0)
    s[i++] = '-';
  s[i] = '\0';
  reverse(s);
}
```

s[i]-'0' is a char	when they are added together, the char is converted to an int
10 * n is an int	

EXPLICIT CONVERSION

You can explicitly force conversions using a cast.

cast	(type-name) expression is as if the expression were assigned to a variable of the specified type, which is then used in place of the whole construction.

Cast Example

```
char c1 = 15;
char c2 = 15;
int j = 2379;  /*  2379 = 0x 0000094b */
float g = 12.1;
```

printf ("%d\n", c2 + j);	type : int output: 2394
c1 = j;	type of c1 : char value of c1: 75
printf ("%d\n", c1 + c2);	type : char output: 90
printf ("%d\n", (char) j + c2);	type : char output: 90
printf ("%9.1f\n", j + g);	type : float output: 2391.1
printf ("%d\n", j + (int) g);	type : int output: 2391

LECTURE 12: INITIALIZATION, INCREMENT/DECREMENT, BIT OPS, ASSIGNMENT OPS, CONDITIONAL EXP. (K&R, §§ 2.4, 2.8–2.11)

DECLARATIONS AND INITIALIZATION

	external vars
int x; int a[20];	initialized to zero
int w = 37; int v[3] = {1, 2, 3};	happens once

main () {
  [...]
  func () {
	automatic vars

int y; int b[20];	contains junk on entry
int s = 37; int t[3] = {1, 2, 3};	happens on each entry
} }	

INCREMENT/DECREMENT OPERATORS

Review:	
operator	equivalent to
++x;	x = x + 1;
	prefix increments before the variable is used
x++;	x = x + 1;
	postfix increments after the variable is used
--x;	x = x-1; (prefix)
x--;	x = x-1; (postfix)

prefix/postfix indicate temporal order of execution.
This is separate from binding order (precedence).

Example:

```
n = 5;
m = n-- + 7;
```

Precedence rules dictate that binding order of the second statement is: m = ((n--) + 7);

Postfix says we decrement after the value has been used and the entire expression has been evaluated.

So, despite binding order, we use the value 5.

Example:

```
int arr[4] = {1, 2, 3, 4};
int n = 2;
printf("%d\n", arr[n++]);   output: 3
printf("%d\n", arr[--n]);   output: 3
printf("%d\n", arr[++n]);   output: 4
```

Note: The use of increment/decrement operators can yield undefined results.

Example 1:
```
n = 3;
n = (n++) * (n++);
```
Result is implementation-defined.
Avoid this type of statement.

Example 2:
```
printf("%d %d\n", ++n, n);
```
Again, we're not sure which expression is evaluated first. Avoid.

MORE ON BITWISE OPERATORS

Distinguishing between & and &&	
&	bitwise and
&&	logical operator and

Example:

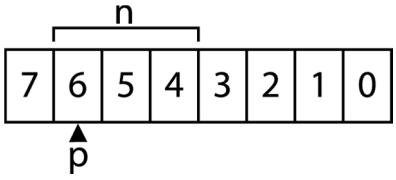
```
int a, b, c, d;
a = 0xf;          0000 1111
b = 0xab;         1010 1011
c = a & b;        0000 1011 (0xb)
d = a && b;       0000 0001
                  a is true AND b is true
                  d is true (0x1)
```

Masking	
&	turn OFF bits where mask bit is 0.
	e.g., n = n & 0xff (sets all except 8 LSBs to 0)
|	turn ON bits where mask bit is 1.
	e.g., n = n | 0xff (sets 8 LSBs to 1)

Other tricks with bitwise operators	
^	can be used to zero any value
	n = n ^ n sets value of n to 0.
~	can be used to get the negative of any value
	n = ~n + 1 sets value of n to -n
	this is the two's complement operation

getbits (K&R, page 49)

Goal: Given unsigned int x
We want the right-adjusted n-bit field that starts at position p.




```
unsigned getbits(unsigned x, int p, int n) {
  return (x >> (p + 1 - n)) & ~(~0 << n);
}
```

Example: x = 94,
p = 6,
n = 3

binary representation of 94: [24 0s] 0101 1110

```
1. (00...01011110 >> (6 + 1 - 3)) & ~(~0 << 3);
2. (00...01011110 >> 4) & ~(11...11111111 << 3);
3. (000000...0101)      & ~(11...11111000);
4. (000000...0101)      &  (00...00000111);
5. 00000000 00000000 00000000 00000101
```

ASSIGNMENT OPERATORS

equivalent: $expression_1$ op= $expression_2$
$expression_1$ = ($expression_1$) op ($expression_2$)

op= is called an assignment operator.
Most binary operators have a corresponding assignment operator.

op=	where op is one of:				
	+	-	*	/	%
	<<	>>	&	^	|

Example:
x *= y + 1 is equivalent to: x = x * (y + 1)
Note the parentheses to maintain $expression_2$.

Example:	
int i = 3;	
i += 3;	i = i + 3 = 6
i <<= 2;	i = i * (2^2) = 24
i |= 0x02;	24 = 0x18 0x18 | 0x02 = 0x1a

CONDITIONAL EXPRESSIONS / TERNARY OPERATOR

we have seen this form:

```
if (expr1)
  expr2
else
  expr3
```

The conditional expression provides another way write this and similar constructions:

$expr_1$? $expr_2$: $expr_3$

$expr_1$ is evaluated first.

Only one of the other 2 is evaluated.
- $expr_1$ is true : evaluate $expr_2$
- $expr_1$ is false: evaluate $expr_3$

Conditionals can be nested.

Examples:

```
z = max(a, b)        z = (a > b) ? a : b;

print EOL at:        for (i = 0; i < n; i++)
  -every 10th          printf("%6d%c", a[i],
   value                 (i % 10 == 9 || i == n-1) ? '\n' : ' ');
  -at end
```