# Types, Operators and Expressions

C provides a few basic data types such as
char, int and float (and their variants)

Variables are designated to be of particular type
that determines their possible values and
admissible operations

Expressions combine variables and constants
to produce values

They are the building blocks in the C language.
This lecture deals with this topic

# Types

The type of an  object determines the values it can have and what operations can be performed on it.

The compiler uses the type information to allocate storage to variables. The type information also enables the compiler to produce efficient code. Compile time type checking allows to catch program errors that would otherwise be difficult to detect during runtime.

# Basic Data Types

char , int , float, double are the basic data types in C

The qualifiers short and long are applicable to integers:
short int , long int (int may be omitted here)

The qualifier signed or unsigned may be applied to char or any integer.

The actual size of these types is machine-dependent.
The qualifiers possibly 'stretch' or 'shrink' the size of the type involved. Only the relative sizes of the types is implied:

char < short <= int <= long

# Choice of types for variables

the choice of a type for a variable is both determined by the set of possible values the variable may take and the operations that may be performed on it.

F-C program (K & R page 12): variable fahr takes only integer values ( short int might suffice) but is declared float as floating point arithmetic is needed in this program

File copy program (K&R page 16): int c declaration.
' We must declare c to be a type big enough to hold any value that getchar returns. We can't use char since c must be big enough to to hold EOF in addition to any possible char. Therefore we use int'

Program to raise 2 to power 100 ( Exercise1- question ?) the variable that holds the result of the computation is float

# Variable names

Names are  made up of letters and digits , the first character is always a letter. Underscore "_"  is a letter. Never use names beginning with "_" as library routines use names that begin with it.

C is case sensitive i.e upper case and lower case letters are different,  so variables sum, SUM. Sum, suM,... are all different

keywords like if, else, int, float are reserved, can't be chosen as variable names.

K&R advice: choose variable names related to the purpose of the variable, avoing typo mixups; use short names for local vaiables, especially loop indices and long names for external variables

# Constants

In C any literal number, single character or a character string is known as a constant.

e.g. number 58 is a constant integer value.

'6' , 'a' , ';' are character constants ( uses single quotes)

"hello, world\n" is a string constant.

Expressions consisting entirely of constant values is a constant expression.

e.g. 123 + 6 - 98

#define is used for assigning symbolic names for program constants. Defined names do not behave the same way as variable in a program- there is no 'local' define.

e.g., #define LOWER = 0

# Integer Constants

integer constant like 1234 is an int. A long constant like 123456789L ( terminal L or l)

Unsigned constants written with terminal U and suffix UL denotes unsigned long.

floating point constants have a decimal point or exponent or both; their type is double, unless suffixed. suffix F indicates float and L long double.

integers can be specified in either octal or hexadecimal octal integers begin with 0 (zero) and hex integers with 0x (0X). Suffixes L and U used as defined in integer constants.

e.g. 0XFUL stands for unsigned long constant with value 15 decimal.

# Character Constants

A character  constant ia an integer, written as one character within single quotes, e.g., 'x'

The value of a character constant is the numeric value of the character in machine's character set. e.g., '0' has numeric value 48 ( ASCII character set), unrelated to numeric value 0.

Character constants participate in numeric operations just as any other integers. Comparison with other characters is the most frequent character operation.

'\n' , the new line character, is a single character.

Format character %c is used by printf to display a character
e.g., printf ( "%c" , 'x' ) displays x on the output.

# String Constants

A string constant (or string literal) is a sequence of zero or more characters within doube quotes. The double quote character serves to delimit the string, not part of the string.
e.g., " hello, world"  "" (empty string)

A string containing just one character is not the same as a character constant. e.g., "X" and 'X' have different data types and indeed have different storage sizes.

The type of a string constant is an array of char. The number of characters in the array is one greater than the number of characters in the string as strings terminate with the null character '\0' in their internal representation.

Same escape sequences used in characters apply in strings,
\" used in strings that contain "
String constants can be concatenated at compile time.

# Enumeration Constant

An enumeration is a list of integer values
e.g., enum boolean {NO, YES};
here the identifier boolean is called an enumeration tag
here NO has value 0 and YES has value 1

explicit values can also be introduced with the names
(see K&R page 39)

enumeration vs #define  (see K&R page 39)

# Declarations

All variables must be declared before use.

A declaration specifies a type and contains a list of one or more variables of that type.

```
e.g.,    int   lower,  upper,  step;
         char      c,  line[1000];
```

K & R recommend each variable declared in a separate line.

A variable may also be intialized in its declaration.

```
e.g.,    char   esc  =  '\ \';
         int    limit  =  MAXLINE  +  1;
```

Automatic variables, not initialized will have undefined values.

A variable declaration with const qualifier specifies that its value will not be changed.

```
e.g.,  const    double e  =  2.7182818;
       const    char  msg[]  =  "warning ";
```

# Arithmetic Operators

The binary arithmetic operators are +, -, *, / and  %  (modulus).

Integer divison truncates the fractional part.
x % y is x mod y. % operator is applicable only to integer operands.

Arithmetic operators associate left to right.

# Relational/Logical Operators

relational operators:
of same precedence : > >= < <=
of just lower precedence: = = !=
(equality operators)

logical operators: && ||
short-circuit evaluation for && and || i.e, the second operator evaluated only if necessary
e.g. while ( i >= 0 && x != A[i] ) i = i - 2 ;

&& has higher precedence than || and both are lower than relational and equality operators.

Numeric value of a relational or logical expression is 1 if the relation is true, and 0 if the relation is false.

unary negation operator !
e.g. if ( !valid ) is the same as if (valid == 0)

# Type Conversions

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

Automatic conversions are those that convert a 'narrower' operand to a 'wider' one (no loss of information)
e.g. conversion of int to float in expression f + i

Expressions that might lose information, like assigning a floating point type to an integer, may draw a warning but not illegal.

A char is a small integer, so chars may be freely used in arithmetic expressions.

Implicit arithmetic conversions work along expected lines ;
the lower type is converted to higher type before the operation proceeds; the result is the higher type.

# Type Casting

Explicit type conversions can be forced ("coerced") in any expression with a unary operator called cast.

(type-name) expression
converts expression to the named type via conversion rules.

e.g., if n is an integer,
sqrt ((double) n ) converts the value of n to double before it is passed to sqrt. The cast produces the value of n in the proper type, n itself is not altered.

# Increment / Decrement Operators

In C the increment operator is ++ and the decrement operator is --
Both ++ and -- can be used either as prefix or postfix operators.

e.g., prefix ++n , postfix n++
The effect in both cases is to increment n. while ++n increments n before the value is used whereas n++ increments n after its value has been used.

e.g. If n is 5 then

```
x = n++;    /* sets x to 5 */
```

but

```
x = ++n;    /* sets x to 6 */
```

These operators can only be applied to variables.
Anexpression such as (i+j)++ is illegal.

# Bitwise Operators

C provides 6 operators for bit manipulation. These may be applied to integral operands , char, short, int and long (signed / unsigned)

| | |
|---|---|
| & | bitwise AND |
| \| | bitwise OR |
| ^ | bitwise exclusive OR |
| << | left shift |
| >> | right shift |
| ~ | one's complement |

e.g. n = n & 0177;

It is important to distinguish between bitwise operators & and | from the logical operators && and | |.
e.g. If x is 1 and y is 2 , then x & y is 0 while x && y is 1

# Assignment Operators and Expressions

Assignment is an expression.

e.g. `n = 0;` `m = n = 0;` `i = (j = 1) + (k = 2);`

The third example above is legal but quite difficult to read.

e.g., `while ((c = getchar()) != EOF)`

`i += 2;` is the same as `i = i + 2;`

Generally if expr_1 and expr_2 are expressions

```
expr_1 op= expr_2 ;
```

is equivalent to

```
expr_1 = (expr_1) op (expr_2);
```

here op can be any of

`+ - * / % << >> & | ^`

In all such expressions type of the assignment expression is the type of its left operand and the value is the value after the assignment.

# Conditional Expressions

expr_1 ? expr_2 : expr_3

e.g., z = (a > b) ? a : b; /* z = max(a, b) */
This is equivalent to the statements

```
if (a > b)
    z = a;
else
    z = b;
```

Conditional expression can be used wherever any other expression can be used.

Type conversion rules apply here if the expr_2 and expr_3 are of different type.
e.g., if f is a float and n is an int
(n > 0) ? f : n
is of type float regardless of the value of n.

# Precedence and Order of Evaluation

Precedence rules are summarized in K & R page 53

e.g., `x = f() + g();`
can evaluate f() and g() in either order with possibly different results if they have a common variable.

e.g., `printf("%d %d \n", ++n, power(2, n));`
better to write

```
++n;
printf("%d %d \n", n, power(2, n));
```

Writing code that depends on the order of evaluation is a bad programming practice.