LEXICAL ELEMENTS & OPERATORS

C Compiler

- Syntax of the language
  - Rules for putting together words and punctuation to make correct

- Compiler
  - A program that checks on the legality of C code
  - If errors, compiler prints error msg.s and stops
  - If NO errors, compiler translates the code into object code

C Program

- C program
  - A sequence of characters that will be converted by a C compiler to object code
  - Compilers first collects the characters of the C program into <u>tokens</u>

  - 6 kinds of tokens
    - Keywords
    - Identifiers
    - Constants
    - String constants
    - Operators
    - Punctuators

Characters used in a C Prog.

- Lowercase letters
- Uppercase letters
- Digits
- Other characters
- White space characters

+ - * / = ( ) { } [ ] < > ' "
! # % & _ | ^ ~ \ . , ; : ?

blank, newline, tab, etc.

Comments

- Arbitrary strings of symbols placed btwn /* and */

```
/*  comment */ /*** another comment ***/ /*****/

/*
 * a comment can be written in this fashion
* to set it off from the surrounding code
*/

/****************************
*     If you wish, you can      *
*     put commas in a box.      *
****************************/
```

Comments

- Used by programmer as a documentation aid
- Aim of documentation
  - To explain clearly
    1) how the program works
    2) how it is to be used
- Should be written SIMULTANEOUSLY with program text
  - 2 problems with inserting comments as a LAST step
    1) Once the prog. is running, the tendency is to either omit or abbreviate comments
    2) Ideally, the comments should serve as running commentary.

    They can not do this if they are inserted after finishing the coding

Keywords

- Reserved words
  - have a strict meaning as individual tokens in C
  - can not be redefined or used in other contexts

Keywords				
auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

Identifiers

- Identifiers
  - A token composed of a sequence of letters, digits, and the special character _ (*underscore*)
  - A letter or underscore must be the first char. of an identifier
  - Lower- and uppercase are distinct

< Examples >

k

_id

iamanidentifier2

so_am_i

< NOT Examples>

not#m2

101_south

-plus

Identifiers

- Give unique names to objects in a prog.
- Keywords can be thought of as identifiers that are reserved to have special meaning
  - e.g. **printf**
- The identifier **main** is special.
- Choose names that are meaningful!!

```
tax = price * tax_rate
```

- Identifier beginning with an underscore
  - Usually used for system names. (e.g. _iob)
  - Please do NOT begin with an undescore!

Constants

- Integer constants
  0, 17
- Floating constants
  1.0, 3.14159
- Character constants
  - Written btwn single quotes
    - **'a', 'b', 'c'**
    - closely related to integers
  - Special character constants
    - **\n** (newline)
    - Backslash is the escape character ("escaping the usual meaning of n")

Constants

- Integer constants
  - Decimal integers **0, 17**
  - Octal integers **017**
  - Hexadecimal integers **0x17**

  - How about **-49** ? Constant expression

String Constants

- A sequence of characters enclosed in a pair of double-quote marks
  - **"abc"**
  - collected as a single token
  - **'a'** and **"a"** are NOT the same.

<Examples >

"a string of text"

""

" "

"/* this is not a comment */"

"a string with double quotes \" within"

"a single backslash \\ is in this string"

<NOT Examples>

/*"this is not a string"*/

"and

neither is this"

Operators & Punctuators

- Arithmetic Operators

  + , - , * , / ,%

  e.g. 5%3 has the value 2.

- Operators can be used to separate identifiers

  a+b (or, **a + b**) /*an expression*/

  a_b /* a 3-character identifier*/

- Some symbols have meanings that depend on context

  printf("%d", a);

  a = b % 7;

Operators & Punctuators

- Punctuators
  - parentheses, braces, commas, and semicolons
- Operators and punctuators, along with white space, serve to separate language elements

```
int main(void)
{
    int a, b = 2, c = 3;
    a = 17 * (b + c);
    ……
```

✓The parentheses following **main** are treated as an operator.

✓The symbols “{“,”,”,”;”,”(“ and “)” are punctuators

- Some special char.s are used in many different contexts

```
a + b        ++a        a += b
```

Precedence and Associativity of Operators

- Precedence: 연산의 우선순위
- Associativity: 연산의 방향

- Parentheses can be used to clarify or change the order in which operators are performed.

  1 + 2 * 3 ⇔ 1 + (2 * 3)

  (1 + 2) * 3

  1 + 2 – 3 +4 – 5 ⇔ (((1+2) – 3) + 4) -5

- Binary operators + and – have the same precedence, the associativity rule "left to right" is used.

Precedence and Associativity of Operators

Operator precedence and associativity	
Operator	Associativity
() ++ *(postfix)* -- *(postfix)*	left to right
+ (*unary*) - (*unary*) ++ (*prefix*) -- (*prefix*)	right to left
* / %	left to right
+ -	left to right
= += -= *= /= etc.	right to left

- a * b – c unary minus sign, binary subtraction

((- a) * b) – c

Increment and Decrement Operators

- ++ and -- are unary operators

<Examples>

```
++i
cnt--
```

<NOT Examples>

```
777++
++(a * b -1)
```

Increment and Decrement Operators

- Difference btwn **++i** and **i++**
  - The expression **++i** causes the stored value of **i** to be incremented first, with the expression then taking as its value the new stored value of **i**.
  - The expression **i++** has as its value the current value of **i**; then the expression causes the stored value of **i** to be incremented.

```
int        a, b, c = 0;
a = ++c;
b = c++;
printf("%d %d %d\n", a, b, ++c);   /* 1 1 3 is printed */
```

Increment and Decrement Operators

- ++ and -- cause the value of a variable in memory to be changed (<u>side effect</u>)
- Other operators do NOT do this

```
a + b

++i;

i++;

i = i + 1;
```

✓**All three statements are equivalent.**

Declarations and Initializations		
int a = 1, b=2, c=3, d=4;		
Expression	Equivalent expression	Value
a * b / c	(a * b) / c	
a * b % c +1	(((a * b) % c) +1	
++ a * b - c --	((++ a) * b) -(c --)	
7 - - b * ++ d	7 - ((- b) * (++ d))	

Assignment Operators

```
a = b + c;     /* assignment statement */
```

- = is treated as an operator
  - Its precedence is lower than all others
  - "right to left" associativity
- Assignment expression
  - *variable* = *right_side*
  - *right_side* is itself expression
  - The value of *right_side* is assigned to *variable*, and <u>that value becomes the value of the assignment expression.</u>

Assignment Operators

```
b = 2;
c = 3;          ⇔      a = (b = 2) + (c = 3);
a = b + c;

a = b = c = 0;   ⇔   a = (b = (c = 0));

k = k +2;        ⇔      k += 2;
```

Assignment operators										
=	+=	-=	*=	/=	%=	>>=	<<=	&=	^=	|=

variable op= expression ⇔ *variable = variable op* (*expression*)

```
j *= k + 3;    ⇔   j = j * (k+3);          /* NOT j = j * k+3; */

int i = 1. j = 2, k = 3, m = 4;
i += j + k;      ⇔  i += (j + k);        ⇔   i = i + (j + k);    /* 6 */
j *= k = m + 5;  ⇔  j *= (k = (m + 5));  ⇔  j = j * (k = (m + 5));  /*18*/
```