Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Spring 2023

Lexical Elements & Operators

C Compiler

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

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

C Program

- A sequence of characters that will be converted by C compiler to object code
- Compiler first collects the characters of the C program into tokens

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

```c
#include <stdio.h>

int main(void)
{
    int i, sum = 0;

    for (i = 1; i <= 5; i++)
        sum = sum + i;

    printf("sum = %d\n", sum);
    return 0;
}
```

Characters used in a C Program

- Lowercase letters
- Uppercase letters
- Digits
- Other characters

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

- White space characters

blank(`' '`), newline(`'\n'`), tab(`'\t'`), etc.

Comments

- Arbitrary strings of symbols placed between /* and */
- The compiler changes each comment into a single blank character
- Used by programmer as a documentation aid
  - How the program works
  - How it is to be used
- Most compilers support C++ single-line comments with //

```
/* comment */

/** another comment **/

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

```
int sum; // This is a comment
```

Keywords

- Reserved words
  - Have a strict meaning as individual tokens in C
  - Cannot be redefined or used in other contexts

`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 (1)

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

Examples

```
k
_id
progpractice2022
SNU_cse_pp
```

Wrong examples

```
not#m2
101_south
-plus
x.32
```

Identifiers (2)

- Give unique names to objects in a program (e.g., variables, functions, ...)
- Keywords can be thought of as identifiers that are reserved to have special meaning
- The identifier `main` is special
- Choose names that are meaningful!

```
a = b * c;
```

```
tax = price * tax_rate;
```

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

Numeric Constants

- Integer constants
  - Decimal representation: `0` `12` `100`
  - Binary representation: `0b0` `0b1100` `0b01100100`
  - Octal representation: `0` `014` `0144`
  - Hexadecimal representation: `0x0` `0xc` `0x64`
  - What about `-12`?

- Floating-point constants
  - Decimal representation: `0.0` `3.14159` `-2.7`
  - Exponential representation: `0e0` `314159e-5` `-0.0027E3`

Character Constants

- Written between single quotes
  - `'a'`, `'b'`, `'c'`, `'!'`
  - Each character corresponds to an integer (ASCII)

- Special character constants
  - `'\n'`, `'\t'`, `'\''`, etc.
  - Backslash is the escape character ("escaping the usual meaning of n")

```c
#include <stdio.h>

int main(void)
{
    char a = 'a';
    char b = 98;
    printf("%c %c\n", a, b);
}
```

String Constants

- A sequence of characters enclosed in a pair of double-quote marks
  - `"abc" "def"` ⇔ `"abcdef"`
  - 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"
```

Wrong examples

```
/*"this is not a string"*/
"and
 neither is this"
'nope!'
\"what about this?\"
```

Operators

- 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;`

Punctuators

- Parentheses, braces, commas, and semicolons
- Operators and punctuators, along with white space, serve to separate language elements
- Some special characters are used in many different contexts
  - `a + b`
  - `++a`
  - `a += b`

Increment and Decrement Operators (1)

- `++` and `--` are unary operators, and can be applied to variables but not to constants or expressions

Examples

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

Wrong examples

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

Increment and Decrement Operators (2)

- Difference between `++i` and `i++`
  - The expression `++i` causes the stored value of `i` to be incremented first, 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 printed */
```

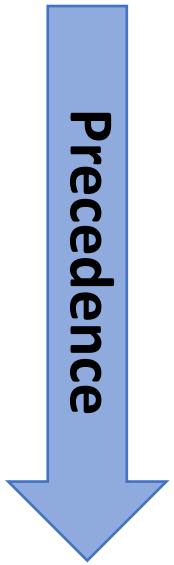
Increment and Decrement Operators (3)

- `++` and `--` cause the value of a variable in memory to be changed (side effect)

- Other operators do NOT do this (e.g., `a + b`)

- All three statements are equivalent:
  - `++i;`
  - `i++;`
  - `i = i + 1;`

Precedence and Associativity (1)



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
((- a) * b) - c
```

Precedence and Associativity (2)

- Parentheses can be used to clarify or change the order in which operators are performed
  - `1 + 2 * 3` ⇔ `1 + (2 * 3)`
  - `(1 + 2) * 3`

- Binary operators `+` and `–` have the same precedence, the associativity rule "left to right" is used
  - `1 + 2 – 3 + 4 - 5` ⇔ `(((1 + 2) – 3) + 4) - 5`

Exercise:

Declarations and initializations		
`int a = 1, b = 2, c = 3, d = 4;`		
Expression	**Equivalent expression**	**Value**
`a * b / c`	`(a * b) / c`	0
`a * b % c + 1`	`((a * b) % c) + 1`	3
`++a * b - c--`	`((++a) * b) - (c--)`	1
`7 - - b * ++d`	`7 - ((- b) * (++d))`	17

Assignment Operators (1)

- Assignment expression: *variable* = *right_side*
  - = is treated as an operator
  - *right_side* is itself expression
  - The value of *right_side* is assigned to variable

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

  - "right to left" associativity:

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

Assignment Operators (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;
```

Expression	Equivalent expression	Value
`i += j + k`	`i = i + (j + k)`	6
`j *= k = m + 5`	`j = j * (k = (m + 5))`	18