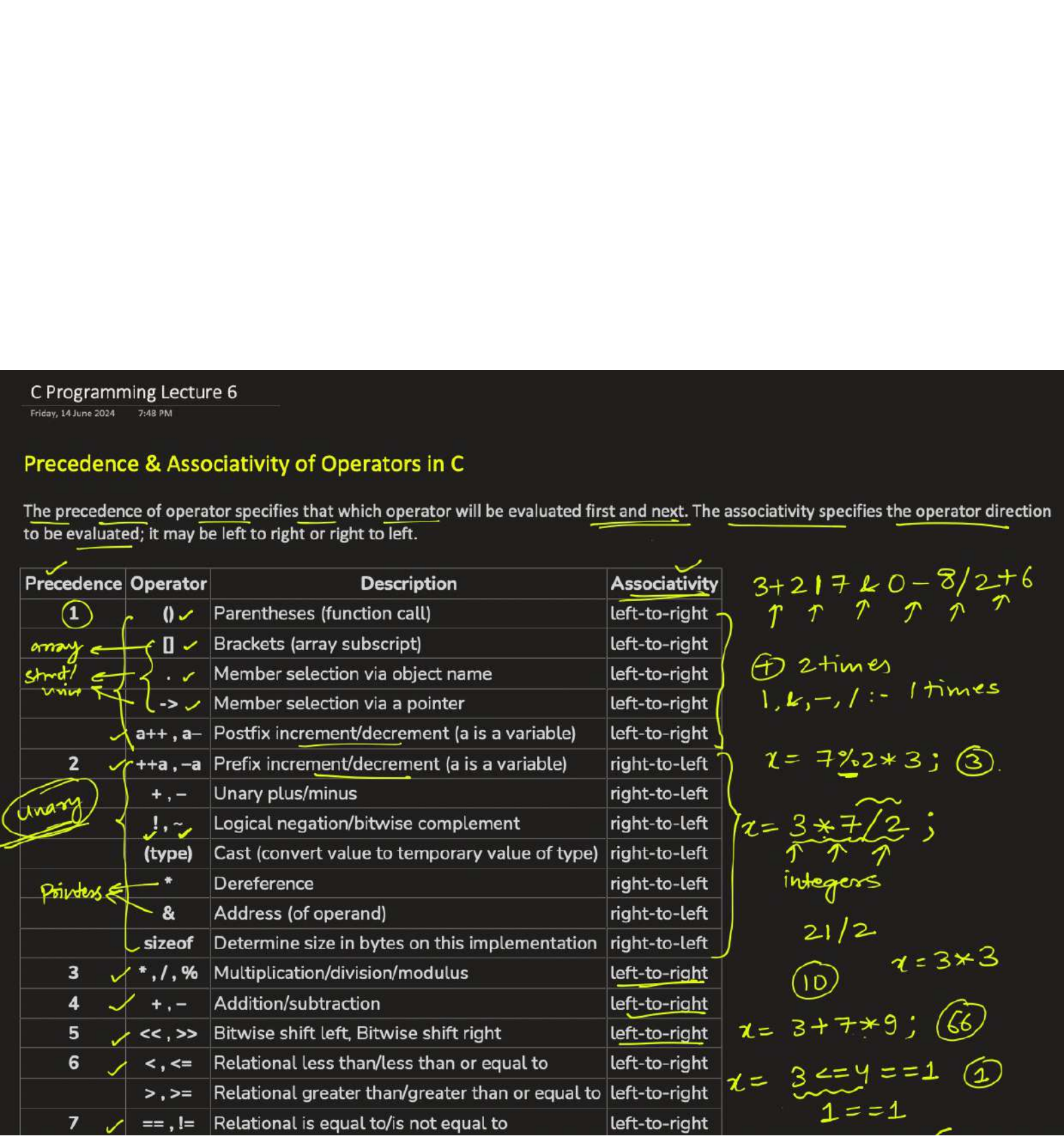
C Programming Lecture 6

Friday, 14 June 2024 7:48 PM

Precedence & Associativity of Operators in C

The precedence of operator specifies that which operator will be evaluated first and next. The associativity specifies the operator direction to be evaluated; it may be left to right or right to left.

Precedence	Operator	Description	Associativity
1	()	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	left-to-right
	.	Member selection via object name	left-to-right
	->	Member selection via a pointer	left-to-right
	a++ , a–	Postfix increment/decrement (a is a variable)	left-to-right
2	++a , –a	Prefix increment/decrement (a is a variable)	right-to-left
	+ , –	Unary plus/minus	right-to-left
	! , ~	Logical negation/bitwise complement	right-to-left
	(type)	Cast (convert value to temporary value of type)	right-to-left
	*	Dereference	right-to-left
	&	Address (of operand)	right-to-left
	sizeof	Determine size in bytes on this implementation	right-to-left
3	* , / , %	Multiplication/division/modulus	left-to-right
4	+ , –	Addition/subtraction	left-to-right
5	<< , >>	Bitwise shift left, Bitwise shift right	left-to-right
6	< , <=	Relational less than/less than or equal to	left-to-right
	> , >=	Relational greater than/greater than or equal to	left-to-right
7	== , !=	Relational is equal to/is not equal to	left-to-right



8 ✓	&	Bitwise AND	left-to-right
9 ✓	^	Bitwise XOR	left-to-right
10 ✓	|	Bitwise OR	left-to-right
11 ✓	&&	Logical AND	left-to-right
12 ✓	||	Logical OR	left-to-right
13 ✓	?:	Ternary conditional	right-to-left
14	=	Assignment	right-to-left
	+=, -=	Addition/subtraction assignment	right-to-left
	*=, /=	Multiplication/division assignment	right-to-left
	%=, &=	Modulus/bitwise AND assignment	right-to-left
	^=, |=	Bitwise exclusive/inclusive OR assignment	right-to-left
	<<=, >>=	Bitwise shift left/right assignment	right-to-left
15 ✓	,	expression separator	left-to-right

$x = 3 <= 4 >= 6$

$1 >= 6$ → 0

assignment operator returns the value it assigned

$x = (y = 3);$ → $y = 3$, $x = 3$

int x = 4, y = 5

x += y <<= 1;

x = ? y = ?

Example 1:

```
int x = 5, y = 10;
int z = x * 2 + y;
printf("\n output = %d", z);
```

20

((x++)++)+y;

post post addition

x = 6, y = 5

x 6 7

(6++) + y

(x++) + y → 6 + y → 11

x 8 7 y 5

y = y << 1

y 10

x += 10

x 14

Example 2:

```
int x = 30, y = 12;
int z = x * 2/ y;
printf("\n output = %d", z);
```

60/12

5

Example 3:

```c
#include<stdio.h>
int main(){
    int a = 5, b = 10, c = 15;
    int result = a + b * c / a - b % a;
    printf("Result: %d\n", result);
    return 0;
}
```

35

a 5 b 10 c 15

{a + [(b * c) / a]} - (b % a);

150 / 5

30

35 - 0 = 35

Example 4:

```c
#include<stdio.h>
int main(){
    int x = 1, y = 0, z = 1;
    int result = x && y || z;
    printf("Result: %d\n", result);
    return 0;
}
```

1

(x && y) || z

1 && 0

0 || 1 = 1

Example 5:

```c
#include<stdio.h>
int main(){
    int a = 4, b = 2, c = 3;
    int result = a + b * c << 2 & 8 - b;
    printf("Result: %d\n", result);
    return 0;
}
```

0

([a + (b * c)] << 2) & [8 - b]

10 << 2

40 & 6

101000

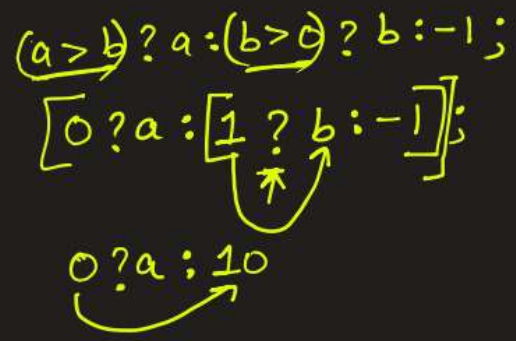
& 000110

000000

Example 6:

```c
#include<stdio.h>
int main(){
    int a = 5, b = 10;
    int result = a > b ? a : b > 0? b : -1;
    printf("Result: %d\n", result);
    return 0;
}
```

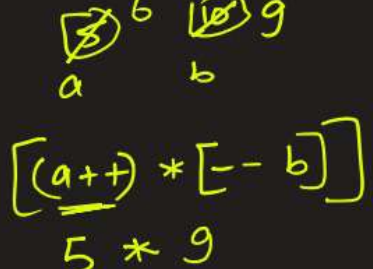
10



Example 7:

```c
#include<stdio.h>
int main(){
    int a = 5, b = 10;
    int result = a++ * --b;
    printf("Result: %d\n", result);
    return 0;
}
```

45



Example 8:

```c
#include<stdio.h>
int main(){
    int a = 6, b = 3, c = 4, d = 2;
    int result = a++ + --b * c & d | a-- || b++ && --c * d++;
    printf("Result: %d\n", result);
    return 0;
}
```

(1)

((([(a++) + [--b] * c] & d) | (a--)) || ((b++) && [--c] * (d++)))

a 6 → 7 → 6

b 3 → 2 → 3

c 4 → 3

d 2 → 3

[((6 + 3 * c) & d) | 7] || [3 && 3 * 2]

9

15 & 3 | 7

t

1

|| 1

t + 6

3 && 6

1

true (1)

Constants in C

In programming languages like ***C, constants*** give you a mechanism to store ***unchanging values*** that hold true throughout the course of the program. These numbers may be used for several things, such as ***creating mathematical constants*** or ***giving variables set values***.

A ***constant in C*** is a value that doesn't change as the program runs. ***Integers, floating-point numbers, characters,*** and ***strings*** are just a few of the several types of constants that may be employed. When a constant has a value, it cannot be changed, unlike variables. They may be utilized in various operations and computations and serve as the program's representation of fixed values.

2 ways to define constant in C ✓

There are two ways to define constant in C programming.

1. const keyword ✓
2. #define preprocessor ✓

const keyword

```
const data_type var_name = value;
```

const int num = 3;

const int num; allowed but NOT suggested

One thing to note here is that we have to initialize the constant variables at declaration. Otherwise, the variable will store some garbage value and we won't be able to change it.

```c
// C program to illustrate constant variable definition
#include<stdio.h>

int main() {

    // defining integer constant using const keyword
    const int int_const = 100;

    // defining character constant using const keyword
    const char char_const = 'J';

    // defining float constant using const keyword
    const float float_const = 3.14;

    printf("Printing value of Integer Constant: %d\n",
           int_const);
    printf("Printing value of Character Constant: %c\n",
           char_const);
    printf("Printing value of Float Constant: %f",
           float_const);

    return 0;
}
```

#define preprocessor

define num 30

```
#define const_name value
```

```c
// C Program to define a constant using #define
#include<stdio.h>
#define pi 3.14

int main() {

    printf("The value of pi: %.2f", pi);
    return 0;
}
```

3.14

String constants

The string constant in C is a sequence of characters enclosed in double quotes (" "). It can represent text, including regular strings, alphanumeric characters, space characters, and special characters. It is a character array that ends with the null character \0.

```c
#include<stdio.h>

int main(){
    char string[] = "Hello, World!";
    printf("The string constant is: %s\n", string);
    return 0;
}
```

Special symbols in C

Some special characters are used in C, and they have a special meaning which cannot be used for another purpose.

- **Square brackets [ ]:** The opening and closing brackets represent the single and multidimensional subscripts. → arrays
- **Simple brackets ( ):** It is used in function declaration and function calling. For example, printf() is a pre-defined function.
- **Curly braces { }:** It is used in the opening and closing of the code. It is used in the opening and closing of the loops.
- **Comma (,):** It is used for separating for more than one statement and for example, separating function parameters in a function call, separating the variable when printing the value of more than one variable using a single printf statement.
- **Hash/pre-processor (#):** It is used for pre-processor directive. It basically denotes that we are using the header file.
- **Asterisk (*):** This symbol is used to represent pointers and also used as an operator for multiplication.
- **Tilde (~):** It is used as a destructor to free memory.
- **Period (.):** It is used to access a member of a structure or a union.

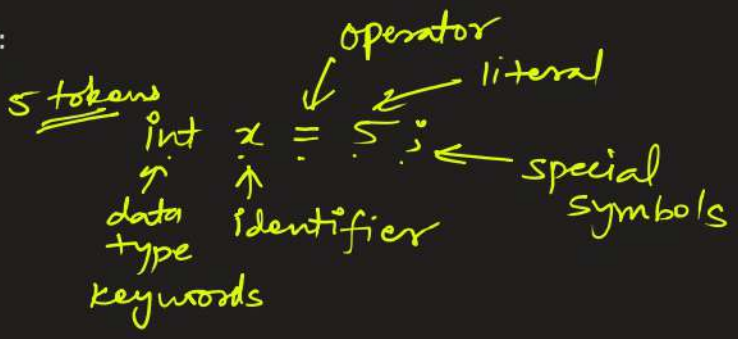
(&) → pointers & address

Tokens in C

Tokens in C is the most important element to be used in creating a program in C. We can define the token as the smallest individual element in C. For `example, we cannot create a sentence without using words; similarly, we cannot create a program in C without using tokens in C.

Tokens in C language can be divided into the following categories:

- Keywords ✓
- Identifiers ✓
- Strings ✓
- Operators ✓
- Constants ✓
- Special Characters ✓



Programming Errors in C

Errors are the problems or the faults that occur in the program, which makes the behavior of the program abnormal, and experienced developers can also make these faults. Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as **debugging**.

There are 5 types of error in C:

- Syntax Errors
- Runtime Errors
- Logical Errors
- Linked Errors
- Semantic Errors

Syntax Errors (Compile)

These are also referred to as compile-time errors. These errors have occurred when the rule of C writing techniques or syntaxes has been broken. These types of errors are typically flagged by the compiler prior to compilation.

```c
#include<stdio.h>

int main() {
  // missing semicolon
  printf("Hello World!")
  return 0;
}
```

Runtime Errors

This type of error occurs while the program is running. Because this is not a compilation error, the compilation will be completed successfully. These errors occur due to segmentation fault when a number is divided by division operator or modulo division operator.

```c
#include<stdio.h>
int main() {
    int a=2;
    int b=2/0;
    printf("The value of b is : %d", b);
    return 0;
}
```

Logical Errors

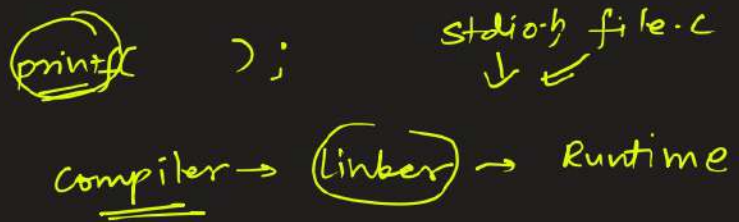
Even if the syntax and other factors are correct, we may not get the desired results due to logical issues. These are referred to as logical errors. We sometimes put a semicolon after a loop, which is syntactically correct but results in one blank loop.

```c
#include<stdio.h>

int main() {
  for(int i = 0; i <= 5; i++);
    printf("Hello World!");
  return 0;
}
```

Linker Errors (linker)

When the program is successfully compiled and attempting to link the different object files with the main object file, errors will occur. When this error occurs, the executable is not generated. This could be due to incorrect function prototyping, an incorrect header file, or other factors. If main() is written as Main(), a linked error will be generated.

```c
#include<stdio.h>

int Main() {
  printf("Hello World!");
  return 0;
}
```



Semantic Errors (compile)

When a sentence is syntactically correct but has no meaning, semantic errors occur. This is similar to grammatical errors. If an expression is entered on the left side of the assignment operator, a semantic error may occur.

```c
#include<stdio.h>

int main(){
  int x = 10, y = 20, result;
  x + y = result;
  printf("%d", result);
  return 0;
}
```