Precedence and associativity of operators

We have introduced arithmetic operators `+`, `-`, `*`, and `/`, which stand for the usual arithmetic operations of addition, subtraction, multiplication, and division, respectively. Now, we introduce the modulus operator, denoted by the percent sign (%). It is a binary arithmetic operator used to compute the remainder of an *integer division*. If `a` is the dividend and `b`$\neq$`0` is the divisor, then `a%b` satisfies

$$a = (a/b) * b + (a\%b)$$

Clearly, the sign of the remainder `a%b` is always the same as the sign of the dividend. For example, `-7%2` is -1 and `7%-2`=1. Operators have rules of *precedence* and *associativity* that are used to determine how expressions are evaluated. Consider the expression

```
2+3*4/3
```

The operators `*` and `/` are of equal precedence, and they have higher precedence than `+`. Since the operators `*` and `/` have the same precedence, the associativity rule "left to right" is used to determine how it is evaluated. Thus the above expression is equivalent to

```
2+((3*4)/3)
```

Hence, the value of the expression is `6`. On the other hand, the expression

```
2+4/3*3
```

is equivalent to

```
2+((4/3)*3)
```

which evaluates to `5`. Note that `4/3` results in `1` since integer division is performed due to both operands being integers. Since expressions inside parenthesis are evaluated first, the expression

```
(2+3)*4/3
```

is equivalent to

```
((2+3)*4)/3
```

which results in the value `6`.

Category	Operator	Associativity
Postfix operators	`( )` (parenthesis), `++` (postfix increment), `--` (postfix decrement),	left to right
Unary operators	`+` (unary plus), `-` (unary minus), `++` (prefix increment), `--` (prefix decrement), sizeof, (type) (type casting)	right to left
Multiplicative operators	`*` (multiplication), `/` (division), `%` (modulus)	left to right
Additive operators	`+` (addition), `-` (subtraction)	left to right
Assignment operators	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	right to left

The table shown here gives the rules of precedence and associativity for some operators of C. Some of them have already been described, and the rest will be described here. Note that there

are many other operators in C that we will describe later. All the operators in a given row, e.g., `*`, `/`, `%`, have equal precedence with respect to each other, but they have higher precedence than all the operators that appear on the rows below them.

Unary operators: In addition to binary plus and binary minus, we also have unary plus and unary minus. For example, the following expression

`+-a*b-+c`

has unary minus operator on the operand `a` and unary plus operator on the operand `c` and `-a`. Now, since two unary operators appear before `a`, they are evaluated from right-to-left (see table). This `+-a` is equivalent to `+(-a)`. The above expression is equivalent to

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

Now consider the problem of finding the average value `av` (a real number) of four integers `2`, `4`, `6`, and `6`. The answer is `av=4.5`. This can be accomplished using `av=(2+4+6+6)/4.0;` (note the decimal point in the denominator). However, if we write `av=(2+4+6+6)/4;`, then an integer division occurs which results in `4` which is converted to `4.0` (a real) and assigned to `av`. Suppose we want to calculate the average using identifiers as shown in the code segment below.

```
int a=2,b=4,c=6,d=6,n=4;
double av;
av=(a+b+c+d)/(double) n;
```

To make a real division, we need to convert at least one of the numerator or denominator to a real. Thus, integer `n` is converted to real by `(double) n`, which is an example of explicit type conversion, also known as explicit type casting. Note that potential data loss may occur when casting from a larger data type to a smaller one. For example, `double` to `int` will truncate the decimal part. Consider the following code segment:

```
int a,b;
a=4.789;
b=(int) 4.789;
printf("a=%d b=%d\n",a,b); // a=4 b=4 is printed
```

Here, implicit type conversion occurs in the statement `a=4.789;`, which may result in a warning during compilation. However, the statement `b=(int) 4.789;` uses explicit type conversion. Explicit type conversion provides more clarity in code.

The `sizeof` operator in C is a compile-time unary operator that returns the size, in bytes, of a data type or an expression. This operator is a very useful tool that helps programmers understand how much memory a variable or data type occupies in the computer's memory. The `sizeof` operator returns a value of type `size_t`, which is an unsigned integer and is guaranteed to be large enough to hold the size of any object that can be allocated in memory. Since `size_t` is an unsigned type, the value returned by `sizeof` will always be a non-negative number. To

print `size_t`, we use either `%zu` or `%lu`. Consider the following C program, which prints the size of typical variables on my macOS.

```
#include<stdio.h>
int main()
{
   char a;
   int b;
   float c;
   double d;
   double e[10];
   printf("Size of char: %zu byte\n",sizeof(a));
   printf("Size of int: %zu bytes\n",sizeof(b));
   printf("Size of float: %zu bytes\n",sizeof(c));
   printf("Size of double: %zu bytes\n",sizeof(d));
   printf("Size of double: %zu bytes\n",sizeof(double));
   printf("Size of array e: %zu bytes\n",sizeof(e));
   return 0;
}
```

Listing 10: C program "csize.c"

Compiling with `$ gcc csize.c ↩` and running the default executable with `$ ./a.out ↩` produce the following output:

```
Size of char: 1 byte
Size of int: 4 bytes
Size of float: 4 bytes
Size of double: 8 bytes
Size of double: 8 bytes
Size of array e: 80 bytes
```

Analysis of the output of "csize.c" program

- ☐ `printf("Size of double: %zu bytes\n",sizeof(d));` Since `d` is `double` variable, `sizeof(d)` returns the size in bytes for a double variable. The same can be done using the statement `printf("Size of double: %zu bytes\n",sizeof(double));`.
- ☐ `printf("Size of array e: %zu bytes\n",sizeof(e));` Here `e` is a one-dimensional array (will be covered later, think of it as a vector now) of dimension `10`, whose each component can hold a `double` value. Hence, its size is `10` times the size of a `double`.

Increment and decrement operators: The increment operator `++` and decrement operator `--` are unary operators. They are interesting in the sense that they can be used as both prefix and postfix operators. Suppose that `i` is a `int` variable. Then both `++i` and `i++` are valid expressions. The operator `++` in `++i` and `i++` acts as a prefix operator and a postfix operator, respectively. Both `++` and `--` can be applied to variables but not to constants and expressions. For example, for variables `a` and `b`, the expression `(a+b)++` is not valid. Similarly, the expression `++2` is also not valid.

Each of the expressions `++a` and `a++` has a value, and each of them causes the stored value of `a` in the memory to be incremented by 1. The expression `++a` causes the stored value of `a` to be incremented first, and the expression then takes as its value the new stored value of `a`. On the other hand, the expression `a++` has as its value the current value of `a`; then the expression causes the stored value of `a` to be incremented. The following code segment explains this point.

```
int a=2,b=3,c;
c=a++*--b;
printf("a=%d b=%d c=%d\n",a,b,c);  //a=3 b=2 c=4 is printed
```

Due to the precedence of the operators, the expression in the second line is equivalent to
`c=(a++)*(--b);`
The expression `a++` takes the current value of `a`, which is `2`, and then this expression causes the value of `a` to be incremented to `3`. The expression `--b` takes the decremented value of `b`, which is 2. Thus, the right side evaluates to `4`, which is assigned to `c`. Hence, the output `a=3 b=2 c=4` is printed. Later, we will come across control loops where we can use either `++i` or `i++`, and both give the same result. Note that each of the statements `i++;` and `++i;` is equivalent to `i=i+1`. Note that `++` and `--` cause the value of a variable in memory to be changed, which is not true for other operators. For example, an expression such as `a*b` leaves the values of `a` and `b` unchanged. These ideas are expressed by saying that the operators `++` and `--` have a side effect: not only do they yield a value, but they change the stored value of a variable in memory.
Assignment operators: To change the value of a variable, we make use of an assignment statement. For example, consider the following code segment:

```
int a,b,c;
a=5;
b=2;
c=a*b;
printf("%d\n",c);  // 10 is printed
```

Unlike other languages, C treats `=` as an operator. Its precedence is lower than all those operators that we have discussed until now, and its associativity is from right to left. A simple assignment expression is of the form

`variable=right_side`

where `right_side` is an expression. When we place a semicolon (`;`) at the end, then this becomes an assignment statement. We have three assignment statements in the above code segment. The assignment operator `=` has the two operands: a `variable` and a `right_side`. The value of `right_side` is assigned to `variable`, and that value becomes the value of the assignment expression as a whole. To see this, consider the code segment below:

```
int a,b,c;
a=5;
b=2;
printf("%d\n",c=a*b);  // 10 is printed
```

Here, we print the value of the expression `c=a*b` and due to the reason mentioned above, it takes the value of the `right_side` expression. Note that due to this peculiar behaviour, the above code can be written as follows:

```
int a,b,c;
c=(a=5)*(b=2);
printf("%d\n",c);  // 10 is printed
```

The assignment expression `a=5` assigns the value `5` to `a`, and the value of the expression `(a=5)` becomes `5`. Similarly, the expression `b=2` assigns `2` to `b`, and the value of the expression `(b=2)` becomes `2`. Finally, the two assignment expressions are multiplied, and the resulting value `10` is assigned to `c`.

Now consider the following code segment in which `a`, `b` and `c` are of `int` and we assign value `0` to each of them:

```
int a,b,c;
a=0;
b=0;
c=0;
```

The same can be accomplished using the following code segment:

```
int a,b,c;
a=b=c=0;
```

Since the associativity of operator `=` is from right to left, the statement `a=b=c=0;` is equivalent to `a=(b=(c=0));`. Hence, first `c` is assigned the value `0`, and the value of the expression `(c=0)` also becomes `0`. Hence, `b=(c=0)` assigns value `0` to `b` and the values of the expression `(b=(c=0))` becomes `0`, which is finally assigned to `a`.

In addition to the assignment operator `=`, there are other assignment operators as well (see the table). For example, the statement

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

can be written as

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

Here, `+=` is the assignment operator. Similarly, there are other assignment operators like `-=`, `*=`, `/=`, etc. The statement `a/=b+c;` is equivalent to `a=a/(b+c);`.

Return to Lecture Topics