

Lecture 1

Expressions & Statements

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, Kumar

EXPRESSIONS

A combination of constants and variables together with the operators is referred to as an *expression*.

Constants and variables by themselves are also considered expressions.

An expression that involves only constants is called a *constant* expression.

We will, for the present, restrict ourselves to the arithmetic operators and the corresponding expressions, called *arithmetic expressions*.

In an arithmetic expression, integer, character, or floating-point type of data can participate as operands.

Thus, the following are valid arithmetic expressions:

`012 'n' i -x`

`12.3 / 45.6 -(i +1)`

`i % j`

`32 + 1.8 * c`

Evaluation of an Expression

Every expression has a value that can be determined by first binding the operands to the operators and then evaluating the expression.

If an expression contains more than one operator and parentheses do not explicitly state the binding of operands, it may appear that an operand may be bound to either of the operators on its two sides.

For example,

the expression $32 + 1.8 * c$ may be interpreted as

$(32 + 1.8) * c$ or as $32 + (1.8 * c)$.

Evaluation of an Expression

C uses a *precedence and associativity rule* and a *parentheses rule* to specify the order in which operands are bound to operators.

Precedence and associativity of the arithmetic operators

Operators	Type	Associativity
+ -	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right

Every operator in C has been assigned a precedence and an associativity.

The *precedence and associativity rule* states that the operator precedence and associativity determine the order in which operands are bound to operators.

Operators receive their operands in order of decreasing precedence.

If an expression contains two or more operators of equal precedence, their associativity determines their relative precedence.

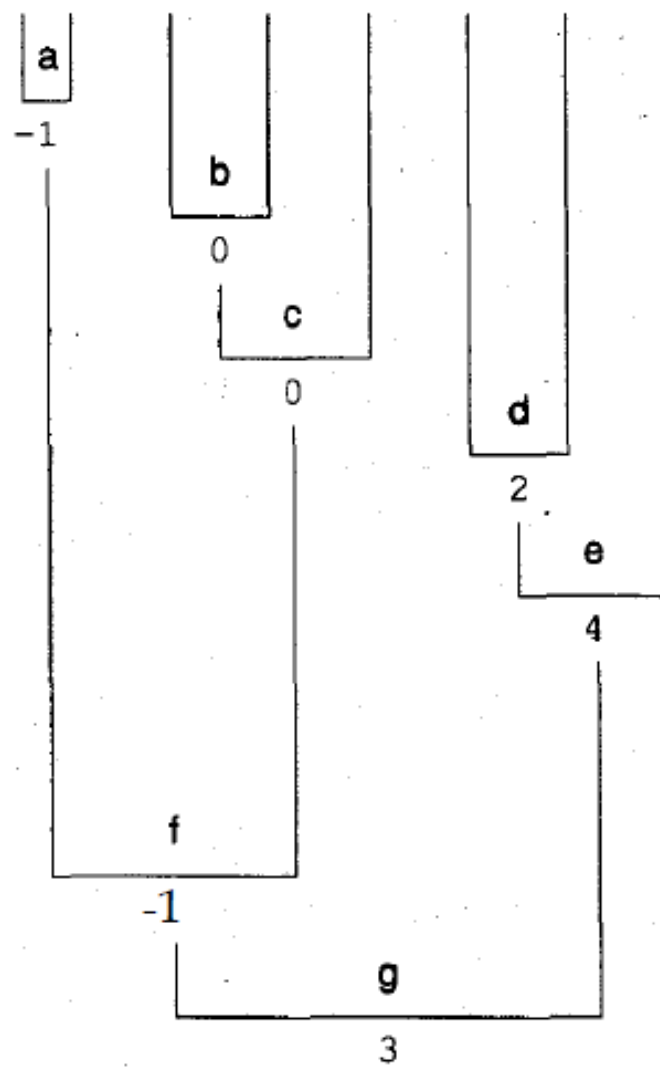
The following table shows the evaluation of some arithmetic expressions using the precedence and associativity rule:

Expression	Equivalent Expression	Value
$2 - 3 + 4$	$(2 - 3) + 4$	3
$2 * 3 - 4$	$(2 * 3) - 4$	2
$2 - 3 / 4$	$2 - (3 / 4)$	2
$2 + 3 \% 4$	$2 + (3 \% 4)$	5
$2 * 3 \% 4$	$(2 * 3) \% 4$	2
$2 / 3 * 4$	$(2 / 3) * 4$	0
$2 \% 3 / 4$	$(2 \% 3) / 4$	0
$- 2 + 3$	$(-2) + 3$	1
$2 * - 3$	$2 * (-3)$	-6
$- 2 * - 3$	$(-2) * (-3)$	6

C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
() [] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of <i>type</i>) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= = <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

- 1 + 2 % 2/2 + 4/ 2*2



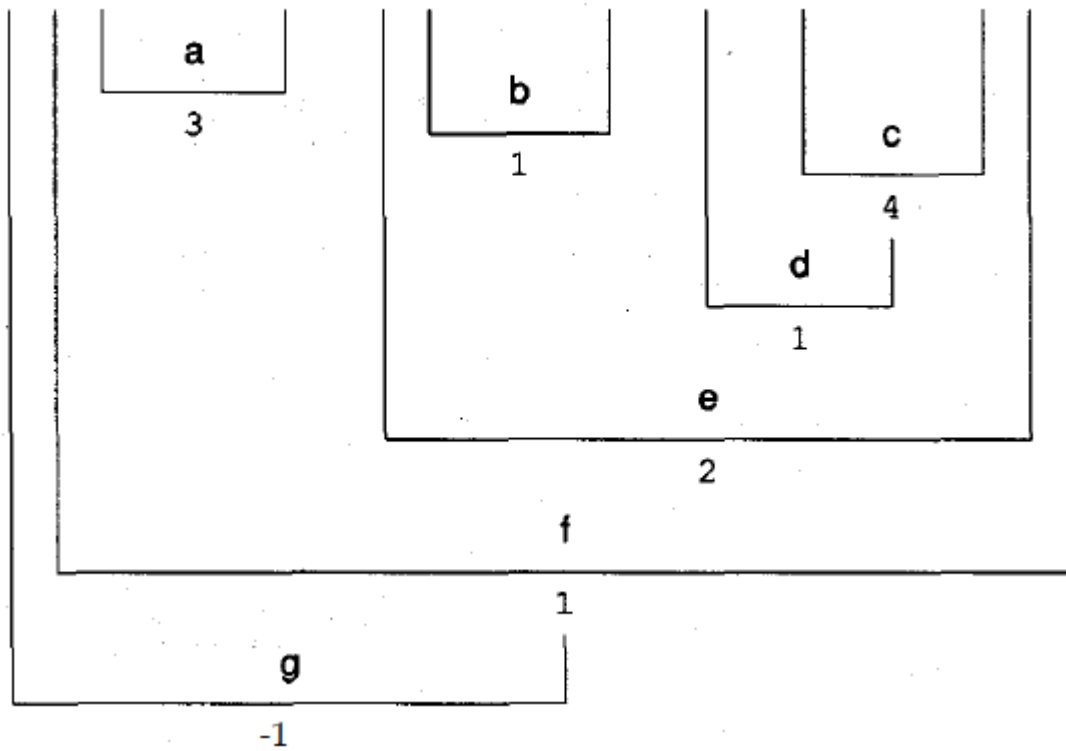
The *parentheses rule*

Parentheses can be used to alter the order of precedence; they force an operation, or set of operations, to have a higher precedence level.

The *parentheses rule* states that the operation will be performed in the innermost set of parentheses, using the precedence and associativity rule where appropriate, and then in the next outer set, etc., until all operations inside the parentheses have been performed.

The remaining operations in the expression are then carried out according to the precedence and associativity rule.

$$- ((1 + 2) \% ((2 / 2) + 4 / (2 * 2)))$$



ASSIGNMENT STATEMENTS

An *assignment expression* is of the form

$$\text{variable} = \text{expression}$$

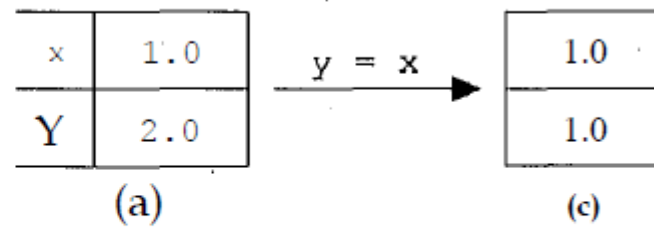
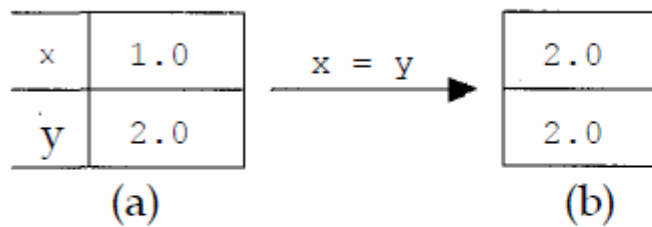
An assignment expression when followed by a semicolon becomes an *assignment statement*.

The "equals" symbol (=) in an assignment should not be interpreted in the same sense as in mathematics.

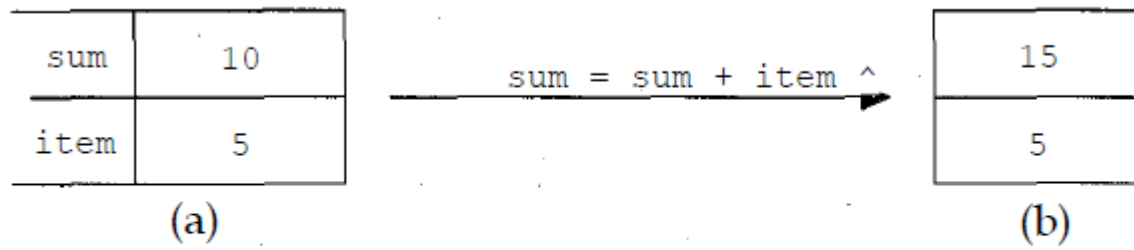
It is an operator that assigns the value of the expression on its right side to the variable on its left side. Thus,

$$x = y; \quad \text{and} \quad y = x;$$

produce very different results.



Effect of assignment statements



Effect of an assignment statement

The precedence of the assignment operator is lower than that of the arithmetic operators.

Thus,

```
sum = sum + item;
```

is interpreted as

```
sum = (sum + item);
```



and not

```
(sum = sum) + item;
```




It is mandatory that the left operand of the assignment operator be an *lvalue*.

lvalue: An expression that refers to an object that can be examined as well as altered.

A variable name is an example of an lvalue.

rvalue: An expression that permits examination but not alteration.

A constant is an example of an rvalue.

15 = n;
x + 1.0 = 2.0;  are not valid assignment statements
because 15 and x + 1.0 are not lvalues.

Note that x + 1.0 evaluates to a value, depending upon the current value of x, which can be examined and used but not altered, and hence is an rvalue.

Increment and Decrement Operators

C provides two operators, *increment* (++) and *decrement* (--), that increment or decrement the value of a variable by 1.

(++) and (--) are unary operators, and require only one operand.

They are of equal precedence and their precedence is the same as the other unary operators.

These operators can be used both as *prefix*,

++*variable* -- *variable*

and *postfix*,

variable++ *variable* --

Increment and Decrement Operators

Both in the prefix and the postfix form,

++ adds 1 to its operand

-- subtracts 1 from its operand

However,

in the prefix form, the value is incremented or decremented by 1 before it is used.

in the postfix form, the value is incremented or decremented by 1 after the use.

The statement `n = ++i;`
is equivalent to the two assignment statements

`i = i + 1;`
`n = i;`

The statement `n = i++;`
is equivalent to the two assignment statements

`n = i;`
`i = i + 1;`

The statement $n = --i;$
is equivalent to the two assignment statements

$$\begin{aligned}i &= i - 1; \\ n &= i;\end{aligned}$$

The statement $n = i--;$
is equivalent to the two assignment statements

$$\begin{aligned}n &= i; \\ i &= i - 1;\end{aligned}$$

Exercise:

if a were 5, what is the value of n in each case:

(1) $n = a++$;

(2) $n = ++a$;

(3) $n = a--$;

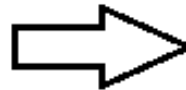
(4) $n = --a$;

(5) $n = (a++) + 6$;

(6) $n = --a + 6$;

Exercise:

Assignment	Before Values	After Values
<code>k = i ++;</code>	<code>i = 1</code>	
<code>k = ++ i;</code>	<code>i = 1</code>	
<code>k = i--;</code>	<code>i = 1</code>	
<code>k = -- i;</code>	<code>i = 1</code>	
<code>k = 5 - i++;</code>	<code>i = 1</code>	
<code>k = 5 - ++i;</code>	<code>i = 1</code>	
<code>k = 5 + i--;</code>	<code>i = 1</code>	
<code>k = 5 + --i;</code>	<code>i = 1</code>	
<code>k = i++ + --j;</code>	<code>i = 1, j = 5</code>	
<code>k = ++i + j--;</code>	<code>i = 1, j = 5</code>	



**Write the values of k,i and j
after evaluation of these
statements**

Assignment	Before Values	After Values
<code>k = i ++;</code>	<code>i = 1</code>	<code>k = 1, i = 2</code>
<code>k = ++ i;</code>	<code>i = 1</code>	<code>k = 2, i = 2</code>
<code>k = i --;</code>	<code>i = 1</code>	<code>k = 1, i = 0</code>
<code>k = -- i;</code>	<code>i = 1</code>	<code>k = 0, i = 0</code>
<code>k = 5 - i++;</code>	<code>i = 1</code>	<code>k = 5 - 1 = 4, i = 2</code>
<code>k = 5 - ++i;</code>	<code>i = 1</code>	<code>k = 5 - 2 = 3, i = 2</code>
<code>k = 5 + i--;</code>	<code>i = 1</code>	<code>k = 5 + 1 = 6, i = 0</code>
<code>k = 5 + --i;</code>	<code>i = 1</code>	<code>k = 5 + 0 = 5, i = 0</code>
<code>k = i++ + --j;</code>	<code>i = 1, j = 5</code>	<code>k = 1 + 4 = 5, i = 2, j = 4</code>
<code>k = ++i + j--;</code>	<code>i = 1, j = 5</code>	<code>k = 2 + 5 = 7, i = 2, j = 4</code>

Compound Assignment Operators

If we let $op=$ denote the compound assignment operator,
the compressed form of the assignment statement

$$variable\ op = expression;$$

is equivalent to the simple assignment statement

$$variable = variable\ op\ expression;$$

Thus, the compressed form of

<code>i = i + 1 ;</code>	is	<code>i += 1;</code>
<code>i = i - a ;</code>	is	<code>i -= a;</code>
<code>i = i * (a + 1) ;</code>	is	<code>i *= a + 1;</code>
<code>i = i / (a - b) ;</code>	is	<code>i /= a - b;</code>
<code>i = i % 101 ;</code>	is	<code>i %= 101;</code>
<code>i = i << 1 ;</code>	is	<code>i «= 1;</code>
<code>i = i » j ;</code>	is	<code>i »= j;</code>
<code>i = i & 01 ;</code>	is	<code>i &= 01;</code>
<code>i = i 0xf ;</code>	is	<code>i = 0xf;</code>

All the compound assignment operators have equal precedence, which is the same as that of the simple assignment operator (=) but lower than that of the arithmetic operators.

Thus, in the preceding example

$i *= a + 1;$ is equivalent to $i = i * (a + 1);$

and not equivalent to $i = (i * a) + 1;$

Nested Assignments

C permits multiple assignments in one statement, and the assignments are then said to be *nested*.

Assignment operators are right-associative,

Thus, the nested statement

$$i = j = k = 0;$$

is interpreted as

$$i = (j = (k = 0));$$

first assigns 0 to k, $k = 0$ evaluates to 0, this value is then assigned to j, the expression $j = k = 0$ now becomes 0, hence i also becomes 0.

Similarly, the statement

$$i += j = k; \quad \text{is treated as} \quad i += (j = k);$$

Exercise:

```
int i = 9, j = 3, k = 1;
```

Assignment

Equivalent Statement

Values After Assignments

```
i /= j = k+1;  
i = j /= k+1;  
i /= j /= k+1;  
i /= j /= k++;  
i /= j /= ++k;
```



Write the equivalent statements and the values of i,j,k after evaluation of these stmts.

```
int i = 9, j = 3, k = 1;
```

Assignment	Equivalent Statement	Values After Assignments
$i /= j = k+1;$	$i /= (j = k+1);$	$k = 1, j = 1+1 = 2, i = 9/2 = 4$
$i = j /= k+1;$	$i = (j /= k+1);$	$k = 1, j = 3/(1+1) = 1, i = 1$
$i /= j /= k+1;$	$i /= (j /= k+1);$	$k = 1, j = 3/2 = 1, i = 9/1 = 9$
$i /= j /= k++;$	$i /= (j /= k++);$	$k = 2, j = 3/1 = 3, i = 9/3 = 3$
$i /= j /= ++k;$	$i /= (j /= ++k);$	$k = 2, j = 3/2 = 1, i = 9/1 = 9$

INPUT AND OUTPUT

C does not provide language constructs for input/output operations.

However, ANSI C has defined a rich *standard I/O library*,

a set of functions designed to provide a standard I/O system for C programs.

We now present some features of only two of these functions:

`printf`, which is used for output operations, and
`scanf`, which is used for input operations.

A program using these functions must

include in it the standard header file `<stdio.h>` using the directive

```
#include <stdio.h>
```

For example, the `printf` statement

```
printf ( "%c", 'C' ) ;
```

results in

```
C
```

The printf statement

```
int i = 1;  
printf("%d\n", 2 * i);
```

results in

2

being displayed, and then the cursor moves to the next line.

The printf statement

```
float r = 100.0;  
printf("\n%f\t%e", r, 100.0);
```

first moves the cursor to the next line and then displays

100.000000 1.000000e+002

on the new line.

The printf statement

```
float c = -11.428572;  
printf("%f Centigrade = %f %s\n",  
       c, 1.8*c+32, "Fahrenheit");
```

displays

```
-11.428572 Centigrade = 11.428571 Fahrenheit
```

The blank characters in the control string are significant. Thus,

```
printf("1 2 3 4 5                end\n");
```

displays

```
1 2 3 4 5                end
```

d, i The integer argument is converted to decimal notation of the form *[-]ddd*.

f The `float` or `double` argument is converted to decimal notation of the form *[-]ddd.dddddd*. By default, six digits are printed after the decimal point.

e The `float` or `double` argument is converted to decimal notation of the form *[-]d.ddddde[±]ddd*. There is one digit before the decimal point, and the exponent contains at least two digits. By default, six digits are printed after the decimal point.

c The argument is taken to be a single character.

s The argument is taken to be a string.

scanf Function

The `scanf` function is the input analog of the `printf` function.

```
scanf (control string, arg1, arg2, ...) ;
```

The *control string* contains conversion specifications

The `scanf` function reads one data item from the input corresponding to each argument other than the control string, skipping whitespaces including newlines to find the next data item, and returns as function value the total number of arguments successfully read.

It returns EOF when the end of input is reached.

As in the case of `printf`, a conversion specification consists of the character `%` followed by a *conversion control character*. Conversion specifications may optionally be separated by whitespace characters, which are ignored. Some of the conversion control characters and their effects are as follows:

- d, i A decimal integer is expected in the input. The corresponding argument should be a pointer to an `int`.
 - f, e A floating-point number is expected in the input. The corresponding argument should be a pointer to a `float`. The input can be in the standard decimal form or in the exponential form.
 - c A single character is expected in the input. The corresponding argument should be a pointer to a `char`. Only in this case, the normal skip over the whitespaces in input is suppressed.
-

For example, given the declarations

```
int i;  
float f1, f2;  
char c1, c2;
```

and the input data

```
10 1.0e1 10.0pc
```

the statement

```
scanf("%d %f %e %c %c", &i, &f1, &f2, &c1, &c2);
```

results in i, f1, f2, c1, and c2

being assigned 10, 10.000000, 10.000000, p, and c respectively.

TYPE CONVERSIONS

An expression may contain variables and constants of different types. We discuss in this section how such expressions are evaluated.

Automatic Type Conversion

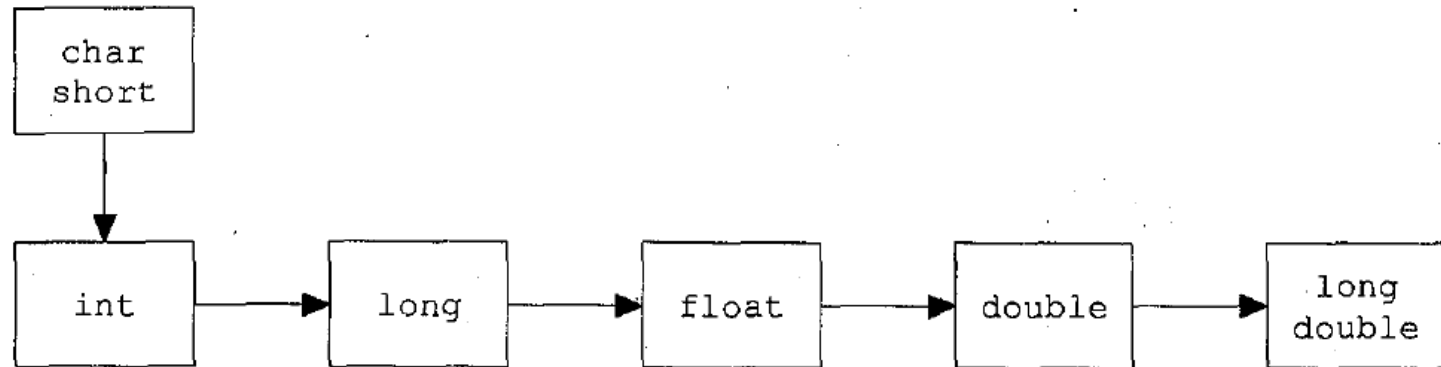
ANSI C performs all arithmetic operations with just six data types:

`int`, `unsigned int`, `long int`, `float`, `double`, **and** `long double`.

Any operand of the type `char` or `short` is implicitly converted to `int` before the operation.

Conversions of `char` **and** `short` to `int` are called *automatic unary conversions*.

Automatic binary conversions apply to both operands of the binary operators and are carried out after automatic unary conversions. In general, if a binary operator has operands of different types, the "lower" type is promoted to the "higher" type before the operation proceeds. The result is of the higher type. More precisely, for each arithmetic operator, automatic binary conversions are carried out according to the following sequence:



Automatic conversion rules (no unsigned operands)

Consider, for example, the evaluation of the expression

$$(c / u - 1) + s * f$$

where the types of c , u , 1 , s , and f are

char , unsigned int , long , short ,
and float respectively.

Expression	Conversion	Operand 1	Operand 2	Result
c	unary	char		int
c / u	binary	int	unsigned int	unsigned int
$c / u - 1$	binary	unsigned int	long int	long int
s	unary	short int		int
$s * f$	binary	int	float	float
$(c / u - 1) + s * f$	binary	long int	float	float

Explicit Type Conversion

C provides a special construct called a *cast* for this purpose.

The general form of a cast is

(cast-type) expression

where *cast-type* is one of the C data types.

For example,

(int) 12.8

casts 12.8 to an int, which is 12 after truncation.

(int) 12.8 * 3.1

casts only 12.8, and not the whole expression 12.8 * 3.1 to int, yielding 37.2 as the value of the expression.

A typical use of a cast is in forcing division to return a real number when both operands are of the type `int`.

Thus,

```
(float) sum / n
```

casts `sum` to a `float`, and hence causes the division to be carried out as a floating-point division.

Without the cast, truncated integer division is performed, if both `sum` and `n` are of type `int`.

Type Conversion in Assignments

When variables of different type are mixed in an assignment expression, the type of the value of the expression on the right side of the assignment operator is automatically converted to the type of the variable on the left side of the operator. The type of the resultant expression value is that of the left operand.

Simple Macros

```
#include <stdio.h>

int main(void)
{
    float radius;

    scanf("%f", &radius);
    printf("surface area = %f\n",
           4 * 3.14 * radius * radius);
    printf("volume = %f\n",
           4 * 3.14 * radius * radius * radius / 3);
    return 0;
}
```

```
#include <stdio.h>
```

```
#define PI 3.14
```



```
int main(void)
```

```
{
```

```
    float radius;
```

```
    scanf("%f", &radius);
```

```
    printf("surface area = %f\n",
```

```
           4 * PI * radius * radius);
```

```
    printf("volume = %f\n",
```

```
           4 * PI * radius * radius * radius / 3);
```

```
    return 0;
```

```
}
```