Expressions

Pedro Ribeiro

DCC/FCUP

2024/2025

(based and/or partially inspired by Pedro Vasconcelos's slides for Imperative Programming)

Expressions

- **Expressions** are made up of **variables**, **constants** and **operators**
- The C language includes **operators** for
  - arithmetic (+, -, *, /)
  - assignment
  - *pre*- and *post*-increment and decrement
  - comparisons
  - logical operations

Arithmetic operators

- **Binary** operators (on two values)

`a + b`	adition
`a - b`	subtraction
`a * b`	multiplication
`a / b`	division
`a % b`	remainder of division:

- **Unary** operators (on one value)

`-a`	unary less
`+a`	unary plus

Unary arithmetic operators

- Need only one operand; examples:

```
i = +1;
j = -i;
```

- The unary `+` operator has no effect:
  it is mainly used to signal positive constants

Binary arithmetic operators

- `+`, `-`, `*`, and `/` allow you to **mix** integer and floating-point operands.
- When **combining** `int` and `float` operands, the result is a `float`.
  - `2 + 0.5` gives `2.5`
  - `3.5 / 2` gives `1.75`
- When the operand are integers, the result of `/` is the **quotient**:
  - `3 / 4` gives `0`
  - `10 / 3` gives `3`

Binary arithmetic operators

- `i % j` is the **remainder** of the integer division of `i` by `j`:
  - `10 % 3` gives `1`
  - `12 % 4` gives `0`
- Be careful with the `%` operator
  - Both operands of `%` must be integers
  - If the right-hand side of `/` or `%` is 0, the result is *undefined* (division by zero)
  - If the right-hand side of `%` is negative, `i\j` has the same sign as `i`

Precedence of operators

- How to interpret `i + j * k`?
  - "add i with the result of multiplying j by k"
  - "add i to j and multiply the result by k"
- We can use **parentheses** to make the meaning **unambiguous**:
  - `i + (j * k)` or `(i + j) * k`
- In the absence of parenthesis: **precedence** between operators disambiguate meaning

Precedence of operators

- **Precedence** determines the order in which operators are evaluated
  - First, unary `+` and `-` are evaluated
  - Then, `*`, `/`, `%`
  - Finally, binary `+` and `-`
  - Examples:
    - ★ `i + j * k` equivalent to `i + (j*k)`
    - ★ `-i * -j` equivalent to `(-i) * (-j)`
    - ★ `+ i + j/k` equivalent to `(+i) + (j/k)`

Associativity of operators

- **Associativity** defines how to interpret two or more operators with equal precedence
  - Binary operators associate on the left
  - Unary operators associate on the right
  - Examples:
    - ★ `i - j - k` equivalent to `(i - j) - k`
    - ★ `i * j / k` equivalent to `(i * j) / k`
    - ★ `- + j` equivalent to `-(+j)`

Simple assignment

- The assignment `var = expr` calculates the value of `expr` and copies it to the variable `var`
- The right-hand side can be a constant, a variable or a complex expression

```
i = 5;            // value of i:  5
j = i;            // value of j:  5
k = 10 * i + j; // value of k:  55
```

- If the variable and expression are not of the same type, there is an implicit type conversion

```
int i;
float f;

i = 72.99; // value of i:  72
f = 136;   // value of f:  136.0
```

Assignments are expressions

- In the C language, an assignment is also an **expression**
- The result of `var = expr` is the value that was assigned
- Example:

```
int i, j, k;
i = 1;
k = 1 + (j = i); // j:  1, k:  2
```

caution: using assignments in the middle of expressions can make programs difficult to understand

- We can assign the same value to several variables:

```
i = j = k = 0;
```

- As the assignment associates to the right, this is equivalent to:

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

Left values (lvalues)

- The left-hand side of an assignment corresponds to a memory location (*lvalue*)
- A variable is an lvalue, but constants or compound expressions are not
- The compiler signs an assignment with an invalid left-hand side: (*"invalid lvalue in assigment"*)

```
i = 12;    // ok:  i is an lvalue
12 = i;    // error:  12 is not an lvalue
1+j = 12; // error:  1+j is not an lvalue
```

Compound assignment

- A variable is often assigned a new value that depends on its current value. For example:

```
i = i + 2;
```

- In these cases we can use a **compound assignment**:

```
i += 2;
```

- The compound assignment **operators** are:
  - `v += e;` add *e* from *v*, saving the result in *v*
  - `v -= e;` subtract *e* from *v*, saving the result in *v*
  - `v *= e;` multiply *e* by *v*, saving the result in *v*
  - `v /= e;` divide *v* by *e*, saving the result in *v*
  - `v %= e;` calculate the remainder of the division of *v* by *e*, saving the result in *v*

Increment and decrement

- You can often add or subtract an integer variable by one:

```
i = i + 1;
j = j - 1;
```

- Here too we can use a compound assignment:

```
i += 1;
j -= 1;
```

- Alternatively, we can use the **increment or decrement operators** They can be used prefixed (`++i` or `--i`) or postfixed (`i++` or `i--`)

```
++i; // equivalent to i = i + 1
--j; // equivalent to j = j - 1
```

Increment and decrement

- `++i` first modifies variable `i` (increments it by one) and then gives the resulting value.

```
i = 1;
printf("%d\n", ++i); // prints 2
printf("%d\n", i);   // print 2
```

- `i++` first gives the current value of `i` and then modifies the variable `i` (increments by 1 unit).

```
i = 1;
printf("%d\n", i++); // print 1
printf("%d\n", i);   // print 2
```

Increment and decrement

- The decrement operator behaves in a similar way to the increment operator:

```
i = 1;
printf("%d\n", --i); // print 0
printf("%d\n", i); // print 0
i = 1;
printf("%d\n", i--); // print 1
printf("%d\n", i); // print 0
```

- It can be difficult to follow the effect of multiple $++$ or $--$ on several variables in the same expression:

```
i = 1;
j = 2;
k = ++i + j++; // i:  2, j:  3, k:  4

i = 1;
j = 2;
k = i++ + j++; // i:  2, j:  3, k:  3
```

recommendation: avoid expressions with multiple increments

Order of evaluation

```
a = 5; c = (b = a + 2) - (a = 1);
```

- What is the final value of c?
  - If we first evaluate `b = a + 2` the result is `6`
  - If we first evaluate `a = 1` the result is `2`

  The behaviour of this program depends on the order in which the subexpressions are evaluated...

- The C language standard **does not guarantee** an order for evaluating subexpressions
  - e.g.: when calculating `(a + b) * (c - d)` we don't know if `(a + b)` or `(c - d)` will be calculated first.

- For most expressions, the order of evaluation does not affect the result. However, it can affect the result if the sub-expressions modify variables, like in the expression on the top of this slide...

Undefined behavior

- Expressions such as `c = ( b = a + 2) - (a = 1)` or `j = i * i++` have **undefined behaviour**:
  - may give different results with different compilers (or versions)
  - may not execute, terminate abruptly or give wrong results
- We should always avoid expressions with undefined behaviour
- The compiler helps detect some undefined behaviour with the `-Wall` compilation option *(all warnings)*:

```
gcc -Wall -o program program.c
```

- We can always rewrite the expression to avoid undefined behaviour.

  Instead of:

```
a = 5; c = (b = a + 2) - (a = 1);
```

  We could write:

```
a = 5; b = a + 2; a = 1; c = b - a;
```

  This way the final result of c is always 6.