Computer Programming

Decision. Assignment. Iteration

Marius Minea

marius@cs.upt.ro

8 October 2013

Review: conditional expression

condition ? *expr1* : *expr2* everything is an *expression*

expr1 or *expr2* may be conditional expression themselves
(if we need more questions to find out the answer)

$$f(x) = \begin{cases} -6 & x < -3 \\ 2 \cdot x & x \in [-3, 3] \\ 6 & x > 3 \end{cases}$$

```
double f(double x)
{
  return x < -3 ? -6     // else, we know x >= -3
                : x <= 3 ? 2*x : 6;
}
```

or: `x >= -3 ? (x <= 3 ? 2*x : 6) : -6`
if $x \geq -3$ we still need to ask $x \leq 3$?

or: `x < -3 ? -6 : (x > 3 ? 6 : 2*x)`
if x is not < -3 or > 3, it must be $x \in [-3, 3]$

Conditional expression (cont'd)

The conditional expression is an expression
⇒ may be used *anywhere* an expression is needed

Example: as an expression of type string in puts
(function that prints a string to stdout, followed by a newline)

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

int main(void)
{
  int c = getchar();  // variable initialized w/ char read
  puts(isupper(c) ? "uppercase letter"
        : islower(c) ? "lowercase letter"
        : isdigit(c) ? "digit"
        : "not letter or digit");
  return 0;
}
```

Expressions and statements

Expression: computes a result

arithmetic operations: `x + 1`

function call: `fact(5)`

Statement: executes an action

`return n + 1;`

Any *expression* followed by `;` becomes a *statement*

`n + 3;` (computes, but does not use the result)

`printf("hello!");` we do not use the *result* of `printf` but are interested in the *side effect*, printing

`printf` returns an `int`: number of characters written (rarely used)

Sequencing

Statements are written and executed in order (*sequentially*)

With *decision*, *recursion* and *sequencing* we can write any program

Compound statement: several statements between *braces* `{ }`
A *function body* is a compound statement (*block*).

```
{                         {
    statement               int c = getchar();
    ...                     printf("let's print the char: ");
    statement               putchar(c);
}                         }
```

A compound statement is considered *a single statement.*
May contain declarations: anywhere (C99/C11)/at start (ANSI C).
All other statements are *terminated* by a semicolon `;`

The *sequencing operator* is the *comma*: *expr1* `,` *expr2*
Evaluate *expr1*, ignore, the value of the expression is that of *expr2*

The conditional statement (if)

Conditional operator `? :` selects from two *expressions* to evaluate

Conditional statement selects between two *statements* to execute

Syntax:

```
if ( expression )          or      if ( expression )
  statement1                         statement1
else
  statement2
```

Effect:

If the expresson is *true* (nonzero) *statement1* is executed,
else *statement2* is executed (or nothing, if the latter is missing)

Each branch has only *one* statement. If several statements are needed, these must be grouped in a *compound statement* `{ }`

The *parantheses* ( ) around the condition are mandatory.
The `else` branch always belongs to the *closest* `if` :

```
if (x > 0) if (y > 0) printf("x+, y+"); else printf("x+, y-");
```

Example with the `if` statement

Printing roots of a quadratic equation:

```c
void printsol(double a, double b, double c)
{
  double delta = b * b - 4 *a * c;
  if (delta >= 0) {
    printf("root 1: %f\n", (-b-sqrt(delta))/2/a);
    printf("root 2: %f\n", (-b+sqrt(delta))/2/a);
  } else printf("no solution\n"); // puts("no solution");
}
```

Can rewrite the *conditional operator ?* `:` using the `if` *statement*

```c
int abs(int x)
{
  return x > 0 ? x : -x;
}
```

```c
int abs(int x)
{
  if (x > 0) return x;
  else return -x;
}
```

Example with if: printing a number

```c
#include <stdio.h>

void printnat(unsigned n) { // recursive, digit by digit
  if (n >= 10)           // if it has several digits
    printnat(n/10);      // write first part
  putchar(’0’ + n % 10); // always write last digit
}

int main(void)
{
  printnat(312);
  return 0;
}
```

Logical expressions in C

The *condition* in the `if` statement or the `? :` operator is usually a *relational expression*, with a *logical value*: `x != 0`, `n < 5`, etc.
The C language was conceived without a special boolean type
(since C99, `stdbool.h` has `bool`, `false` (0) and `true` (1)).

A value is considered *true* when *nonzero* and *false* when *zero*
(when used as a condition in `? :` , `if` , `while` etc.)
⇒ condition must have *scalar* type (integer, floating point, enum)

Comparison operators (`==  !=  <` etc.)
return the *integer* values 1 (for *true*) or 0 (for *false*)
⇒ suitable for direct use as conditions

Library functions often return zero or nonzero (NOT zero or one!)
only compare `if (isdigit(c))` (nonzero), don't compare to 1 !

Logical operators

With logical operators, we can write complex decisions:

expr	! *expr*
0	1
$\neq 0$	0

negation ! NOT

		e_2	
	e_1 && e_2	0	$\neq 0$
e_1	0	0	0
	$\neq 0$	0	1

conjunction && AND

		e_2	
	e_1 || e_2	0	$\neq 0$
e_1	0	0	1
	$\neq 0$	1	1

disjunction || OR

Reminder: logical operators produce 1 for *true*, 0 for *false*
An integer is interpreted as *true* if *nonzero*, and as *false* if 0

Example: leap year

A year is a leap year if

it is divisible by 4 **and**
it is **not** divisible by 100 **or** it is divisible by 400

Example: leap year

A year is a leap year if

it is divisible by 4 **and**
it is **not** divisible by 100 **or** it is divisible by 400

```
int isleap(unsigned yr)    // 1: leap year, 0: not
{
  return yr % 4 == 0 && (!(yr % 100 == 0) || yr % 400 == 0);
}
```

`!(yr % 100 == 0)` is equivalent with `(yr % 100 != 0)`

Precedence of logical operators

The *unary logical operator* `!` (logical negation): highest precedence

`if (!found)` same as `if (found == 0)` (zero is false)

`if (found)` same as `if (found != 0)` (nonzero is true)

Relational operators: lower precedence than arithmetic ones

⇒ we can naturally write `x < y + 1` for `x < (y + 1)`

Precedence: > >= < <= , then == !=

Binary logic operators: `&&` (AND) evaluated before || (OR)
have lower precedence than relational operators

⇒ can naturally write `x < y + z && y < z + x`

Short-circuit evaluation

Logical expressions are evaluated *left to right*

(in general, for other operators, evaluation order is *unspecified*)

Evaluation stops (*short-circuit*) when the result is known:

for `&&`, when the left argument is false (right is not evaluated)

for `||`, when the left argument is true

```
if (p != 0 && n % p == 0)
  printf("p divides n");

if (p != 0)             // only if nonzero
  if  (n % p == 0)      // test the remainder
    printf("p divides n");
```

⇒ Be careful when writing compound tests!

⇒ Avoid side-effects in compound tests (or place them first)

Evaluation order and precedence are different notions!

`2 * f(x) + g(x)` : multiplication before addition (precedence)

Unspecified which part of sum is *evaluated* first (`f` or `g`)

Assignment

In recursive functions we don't need to change variable values
a programming style typical for (pure) *functional languages*
Recursive calls create *new parameter instances* with *new values*.

In *imperative programming*, we use:
variables to represent objects used in solving the problem
(current character; partial result; number left to process)
assignment, to give a *new value* to a variable
(to express a computation step in the program)

Syntax: *variable* = *expression*
Everything is an *assignment expression*.

Effect: 1. The expression is evaluated
2. the value is *assigned* to the variable and becomes the value of the entire expression.
Example: `c = getchar()` `n = n-1` `r = r * n`

Assignment (cont'd)

May appear in other expressions: `if ((c = getchar()) != EOF)`
. . .

May be chained: `a = b = x+3` (`a` and `b` get the same value)

Any *expression* (function call, assignment) with `;` is a *statement*

```
printf("hello");   c = getchar();   x = x + 1;
```

A variable changes value *only by assignment*!
NOT in other expressions, or by passing as parameter!

`n + 1`   `sqr(x)`   `toupper(c)`   compute, DON'T change!
`n = n + 1`   `x = sqr(x)`   `c = toupper(c)`   *change*

WARNING!   `=` assignment   `==` comparison.

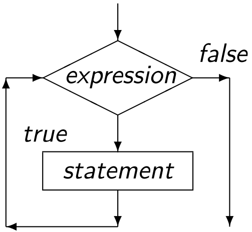
Iteration. The `while` loop (initial test)

Expresses the repetition of a statement, guarded by a condition:

Syntax:

```
while ( expression )
   statement
```

!!! Expression must be between parantheses ( )



Semantics: evaluate expression. If it is true (nonzero):

(1) execute statement (loop *body*)

(2) go back to start of `while` (evaluate expression)

Else (if condition is false/zero), don't execute anything.

⇒ body executes repeatedly, as long as (while) condition is true

Iteration and recursion

We can define iteration (the while loop) recursively:

```
while ( expression )
  statement
```

is the same as

```
if ( expression ) {
  statement
  while ( expression )
    statement
}
```

Rewriting recursion as iteration

```
unsigned fact_r(unsigned n,
              unsigned r) {
  return n > 0
      ? fact_r(n - 1, r * n)
      : r;
}
// called with fact_r(n, 1)
```

```
unsigned fact_it(unsigned n) {
  unsigned r = 1;
  while (n > 0) {
    r = r * n;
    n = n - 1;
  }
  return r;
}
```

```
int pow_r(int x, unsigned n,
                   int r) {
  return n > 0
         ? pow_r(x, n-1, x*r)
         : r;
}
// called with pow_r(x, n, 1)
```

```
int pow_it(int x, unsigned n) {
  int r = 1;
  while (n > 0) {
    r = x * r;
    n = n - 1;
  }
  return r;
}
```

Rewriting recursion as iteration

Easier if function is written by accumulating a partial result (*tail recursion*)

Stop test and initial value for the result are the same as in recursion

Recursive version creates *new instances* of parameters for each recursive call, with new values dependent on the old ones:
ex. `n * r`, `n - 1`, `x * r`, etc.

Iterative version *updates (assigns)* values to variables in each iteration, following the same rules/expressions
Ex. `r = n * r`, `n = n - 1`, `r = x * r`

Both variants return the accumulated result

!!!: Recursion and iteration both repeat a processing step
⇒ in a problem we use one or the other, rarely both

Reading a number iteratively, digit by digit

```
#include <ctype.h>    // for isdigit()
#include <stdio.h>    // for getchar(), ungetc(), stdin
unsigned readnat(void)
{
  unsigned r = 0;     // accumulates result
  int c;              // character read
  while (isdigit(c = getchar())) // while digit
    r = 10*r + c - '0';          // build number
  ungetc(c, stdin);   // put back char != digit
  return r;
}
int main(void)
{
  printf("number read: %u\n", readnat());
}
```

`ungetc(c, stdin)` puts character c back to standard input

Character will be read next time, e.g. on using `getchar()`

Reading character by character: filters

E.g. function that reads and prints up to a specified character
returns that character or EOF if reached before that char

```
int printto(int stopchar)    // up to what char ?
{
  int c;
  while ((c = getchar()) != EOF && c != stopchar)
    putchar(c);
  return c;
}
```

DON't forget ( )   `(c=getchar())!=EOF`   (assign, then compare)

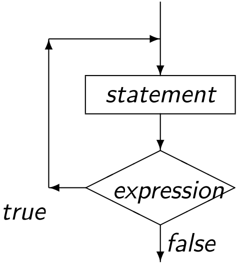
```
int skipto(int stopchar)     // ignore up to stopchar
{
  int c;
  while ((c = getchar()) != EOF && c != stopchar);
  return c;
}
```

; after `while(...)` is *empty statement* (does nothing)

DON'T use ; by mistake!

The do-while loop (final test)

```
do
  statement
while ( expression );
```



Sometimes we know that a cycle needs to be executed at least once (we read at least one character, a number has at least one digit)

Like the `while` loop, executes *statement* as long as the expression evaluates to true (nonzero)

Expression is (re)evaluated *after* every iteration

Equivalent with:

```
statement
while ( expression )
  statement
```

ERRORS with characters and loops

NO! ~~`char`~~ `c = getchar();` YES: `int c = getchar();`

If `char` is `unsigned char`, `c` will never compare equal to `EOF (-1)`

⇒ will never leave a `while (c != EOF)` loop

If `char` is `signed char`, reading byte 255 becomes `-1 (EOF)`

⇒ a valid char (code 255) will be taken as `EOF` (early stop)

NO! ~~`while (!EOF)`~~ `EOF` is a nonzero constant (`-1`)

thus the condition is always false, the loop is never entered!

YES: `while ((c = getchar()) != EOF)` and careful with the `()` !

NO! ~~`while (c = getchar() != EOF)`~~

`!=` has higher precedence, its result (`0` or `1`) is assigned to `c`

NO! ~~`int c = getchar(); if (c < 5) puts("failed exam");`~~

`c` is ASCII code, not value of a one-digit number. Need `c-'0'`

NO! ~~`while ((c = getchar()) != '\n')`~~ may loop forever!

YES: `while ((c = getchar()) != '\n' && c != EOF)` will exit!