Programming I

Control

Dimitrios Michail

Dept. of Informatics and Telematics
Harokopio University of Athens

Relational and Equality Operators

A program besides arithmetic calculations also needs to take decisions.

For example a program which prints the students that passed a course needs to read the grade of each enrolled student and to check whether that grade was greater or equal to 50%.

Relational and Equality Operators

The C language does not provide any particular type which takes values TRUE or FALSE but instead uses values $\neq 0$ and 0 in order to represent those values.

Relational and Equality Operators

Equality and relational operators allow us to perform comparisons and to get as a result a value of 1 or 0 (TRUE or FALSE) which can be used in order to execute different parts of the program.

Relational and Equality Operators

operator	code in C	condition result
$>$	`x > y`	1 (TRUE) if $x > y$, otherwise 0 (FALSE)
$<$	`x < y`	1 (TRUE) if $x < y$, otherwise 0 (FALSE)
$>=$	`x >= y`	1 (TRUE) if $x \geq y$, otherwise 0 (FALSE)
$<=$	`x <= y`	1 (TRUE) if $x \leq y$, otherwise 0 (FALSE)
$==$	`x == y`	1 (TRUE) if $x = y$, otherwise 0 (FALSE)
$!=$	`x != y`	0 (FALSE) if $x = y$, otherwise 1 (TRUE)

These operators are executed from left to right and have lower precedence than the arithmetic operators.

Operators Precedence

The way that expressions are calculated depends on the precedence of the operators:

1. **parentheses:** (), `expr++`, `expr--`
   Calculated first, from left to right. Nested parentheses are calculated first.
2. **unary operators:**
   $+$, $-$, `++expr` or `--expr` (prefix) Calculated from right to left.
3. **multiplication, division and remainder:** $*$, $/$, or %
   Calculated from left to right.
4. **addition, subtraction:** $+$ or $-$
   If there are many, calculated from left to right.
5. **relational:** $<$, $>$, $<=$, $>=$
   Calculated from left to right.
6. **equality:** $==$, $!=$
   Calculated from left to right.
7. **assignment:** $=$, $+=$, $-=$, $*=$, $/=$, $\%=$
   From right to left.

Control Flow in C

The `if`

```c
int main() {
    int grade;

    grade = 80;

    if (grade >= 50)
        printf("PASS\n");
}
```

Allows to execute a set of instructions or not based on a condition which depends on the value of expressions.

Control Flow in C

The if

The if structure accepts as a condition any expression which has an integer value. If that integer is not equal to 0 then it executes the if statement, otherwise it does not.

```
int main() {
    int grade;

    grade = 80;

    if (grade >= 50) {
        printf("grade = %d ", grade);
        printf("(PASS)\n");
    }
}
```

We can execute most than one statement by forming a compound stamement using curly brackets.

Example

```c
#include <stdio.h>

int main() {
    int num1, num2;

    printf("Enter two integers\n");
    scanf("%d%d", &num1, &num2);

    if (num1 == num2)
        printf("%d is equal to %d\n", num1, num2);

    if (num1 != num2)
        printf("%d is not equal to %d\n", num1, num2);

    if (num1 < num2)
        printf("%d is less than %d\n", num1, num2);

    if (num1 > num2)
        printf("%d is greater than %d\n", num1, num2);

    if (num1 <= num2)
        printf("%d is less than or equal to %d\n", num1, num2);

    if (num1 >= num2)
        printf("%d is greater than or equal to %d\n", num1, num2);
}
```

Attention: Assignment operator and `if`

Care needs to be taken when using the equality operator, so that we do not accidentally write the same expression using the assignment operator.

```
if (x == 5) {
    /* code to be executed */
}
```

and not

```
if (x = 5) {
    /* code to be executed */
}
```

The expression `x=5` evaluates to 5 which is not equal to zero and thus TRUE. This means that the second (erroneous) program always executes the `if` statement regardless of the value of `x`.

Algorithms and Pseudocode

An algorithm is a set of instructions, typically to solve a class of problems or perform some computation.

Structured Programming

In structured programming we use the following mechanisms in order to implement algorithms:

1. sequence
2. selection
3. iteration
4. functions

Structured Programming

In structured programming we use the following mechanisms in order to implement algorithms:

1. sequence
   in C all statements are executed one after another in the order that they appear in the source code.
2. selection
3. iteration
4. functions

Structured Programming

In structured programming we use the following mechanisms in order to implement algorithms:

1. sequence
2. selection
   one or a number of statements is executed depending on the state of the program. This is implemented using **if-then-else**.
3. iteration
4. functions

Structured Programming

In structured programming we use the following mechanisms in order to implement algorithms:

1. sequence
2. selection
3. iteration
   a statement or a block of statements can be executed multiple times until the program reaches a certain state. C provides several structures for loops like **for, do..while**.
4. functions

Structured Programming

In structured programming we use the following mechanisms in order to implement algorithms:

1. sequence
2. selection
3. iteration
4. functions
   a sequence of program instructions that performs a specific task, packaged as a unit. We have a separate chapter for functions in C.

if/then/else

```
if (/* boolean expression */) {
    /* code to be executed if true */
}
else {
    /* code to be executed if false */
}
```

The else is optional...

if/then/else Example

```
int main() {
    float grade = 0.65;

    if (grade >= 0.50) {
        printf("PASSED!");
    } else {
        printf("FAILED!");
    }
}
```

Nested if/then/else

Control structures can be nested, something that allows us to have greater control on the exact statements which will be executed.

```
int main() {
    float grade = 0.65;

    if (grade >= 0.50) {
        printf("PASSED!");
        if (grade >= 0.85) {
            printf(" with distinction!!");
        }
        printf("\n");
    } else {
        printf("FAILED!\n");
    }

    return 0;
}
```

?:

ternary if (aka conditional operator)

The only ternary operator in C is closely related to the if/then/else structure.

```
boolean expression ? value if true : value if false
```

It has 3 operands:

1. a condition
2. a value for the expression if the condition is TRUE
3. a value for the expression if the condition is FALSE

?:

ternary if (aka conditional operator)

```c
int main() {
    float grade;

    printf("Enter your grade in [0,1])\n");
    scanf("%f", &grade);

    printf("%s\n", grade >= 0.5? "Passed" : "Failed");
}
```

or

```c
int main() {
    float grade;

    printf("Enter your grade in [0,1]\n");
    scanf("%f", &grade);

    if (grade >= 5.0)
       printf("Passed\n");
    else
       printf("Failed\n");
}
```

`while` Loop

The `while` loop allows a programmer to execute certain instructions repeatedly based on a certain boolean condition being TRUE.

```
while(condition) {
    /* code to be executed if condition is true */
}
```

`while` Examples

Print integers from 0 to 9

We would like to print all integers from 0 up to 9 (inclusively), one integer per row.

```c
#include <stdio.h>

int main() {
    int i;

    i = 0;
    while(i < 10) {
        printf("%d\n", i);
        i = i + 1;
    }

    return 0;
}
```

Infinite Loop

```c
#include <stdio.h>

int main() {
    int i;

    i = 0;
    while(i < 10) {
        printf("%d\n", i);
    }
}
```

What will the above program do?

Infinite Loop

```c
#include <stdio.h>

int main() {
    int i;

    i = 0;
    while(i < 10) {
        printf("%d\n", i);
    }
}
```

What will the above program do?
If you happen to run it, use CTRL-C to stop it.

do/while Loop

The do/while loop is similar to the while loop but it guaranties to be executed at least once.

```
do {
    /* code to be executed */
} while(condition);
```

Be careful with the semicolon at the end, it is mandatory!

do/while Loop

Example

```c
#include <stdio.h>

int main() {
    int i;

    i = 0;
    do {
        printf("%d\n", i);
    } while(++i < 10);
}
```

do/while Loop

Example

```c
#include <stdio.h>

int main() {
    const int SECRETCODE = 3313;
    int code;

    do {
        printf("Type the secret code to enter.\n");
        scanf("%d", &code);
    } while (code!=SECRETCODE);

    printf("Well done, you can now enter\n");

    return 0;
}
```

do/while Loop

Example

```c
#include <stdio.h>

int main() {
    const int SECRETCODE = 3313;
    int code;

    do {
        printf("Type the secret code to enter.\n");
        scanf("%d", &code);
    } while (code!=SECRETCODE);

    printf("Well done, you can now enter\n");

    return 0;
}
```

Disclaimer: This is only meant as an example. Never store plain-text passwords!

`for` Loop

The `for` loop helps writing iterations which counters, by providing special places for common patterns such as initialization and incrementing.

```
for(initializations; test conditions; increment value) {
    /* block of code to be repeated */
}
```

for Loop

Example

```
for(i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

for Loop

Example

The condition can be independent from the variables in the initialization or iteration.

```c
#include <stdio.h>

int main() {
    int i, j;

    j = 0;
    for(i = 0; j < 10; i += 2) {
        printf("i = %d, j = %d\n", i, j);
        j++;
    }
}
```

for Loop

Example

You might also have multiple initializations, etc.

```c
#include <stdio.h>

int main() {
    int i, j;

    for(i = 0, j = 0; j < 10; i += 2, j++)
        printf("i = %d, j = %d\n", i, j);
}
```

break

Sometimes we want to stop a loop prematurely.

```c
#include <stdio.h>

int main() {
    int i = 0;

    while(i < 100) {
        printf("%d\n", i);
        if (i >= 9)
            break;
        i++;
    }

    printf("Due to break i should be 9: %s\n",(i==9)?"YES":"NO");
}
```

break can be used in all kind of loops.

continue

Other times we want to skip the remaining part of the loop, but execute more iterations.

```
#include <stdio.h>

int main() {
    int i,j;

    for(i = 0, j = 0; i < 10; i++, j++) {
        printf("j = %d\n", j);
        if (i == 5)
            continue;
        printf("i = %d\n", i);
    }
}
```

Nested Loops

Several times we need multiple nested loops.

```
int main() {
    int i,j;
    for(i = 0; i < 10; i++) {
        for(j = 0; j < 5; j++) {
            printf("(%d,%d) ", i, j);
        }
        printf("\n");
    }
}
```

What does the code above print?

`switch` Statement

There are cases where `if`/`then`/`else` and the two cases are not enough.

```
switch(expression) {
case value1:
    /* code if value1 */
    break;
case value2:
    /* code if value2 */
    ...
    break;
default:
    /* execute default action */
    break;
}
```

- the value of the expression must be an integer
- the value at `case` must be a constant

`switch` Statement

Example

```c
#include <stdio.h>

int main() {
    int acount = 0, bcount = 0;
    char c;

    while ((c = getchar()) != EOF) {
        switch(c) {
        case 'a':
        case 'A':
            acount++;
            break;
        case 'b':
        case 'B':
            bcount++;
            break;
        default:
            break;
        }
    }
    printf("number of a: %d, number of b: %d\n", acount, bcount);
}
```

Boolean Conditions (AND OR NOT)

C provides the boolean operators `&&`, `||` and `!` in order to write more complicated conditions in flow control statements.

operator	meaning	example
`&&`	boolean AND	`x > 10 && x < 20`
`||`	boolean OR	`x <= 10 || x >= 20`
`!`	boolean NOT	`!(x>10 && x < 20)`

Example

```c
/*
 * A program to count letters in input.
 */
#include <stdio.h>

int main() {
    int c ;
    int count = 0;

    while ((c = getchar()) != EOF) {
        if ((c >= 'A') && (c <= 'Z') ||
            (c >= 'a') && (c <= 'z')) {
            count++;
        }
    }

    printf("%d letters\n" , count);

    return 0;
}
```

Operators Precedence

The way that expressions are calculated depends on the precedence of the operators:

1. **parentheses:** (), `expr++`, `expr--`
   Calculated first, from left to right. Nested parentheses are calculated first.
2. **unary operators:**
   $+$, $-$, `++expr` or `--expr` (prefix) Calculated from right to left.
3. **multiplication, division and remainder:** $*$, $/$, or $\%$
   Calculated from left to right.
4. **addition, subtraction:** $+$ or $-$
   If there are many, calculated from left to right.
5. **relational:** $<$, $>$, $<=$, $>=$
   Calculated from left to right.
6. **equality:** $==$, $!=$
   Calculated from left to right.
7. **boolean AND:** `&&` Calculated from left to right.
8. **boolean OR:** `||` Calculated from left to right.
9. **assignment:** $=$, $+=$, $-=$, $*=$, $/=$, $\%=$
   From right to left.

goto

The C language also supports `goto` statements which allow the program counter to jump to any location in memory. This usually leads to "spaggeti code", meaning code which is difficult to follow.

```c
#include <stdio.h>

int main() {
    int a = 1;

    if (a == 0)
        goto test0;
    else
        goto test1;

test0:
    printf("a = 0\n");
    return 0;

test1:
    printf("a = 1\n");
    return 0;

}
```

goto

Whatever can be implemented with `goto` statements can also be implemented using only structured programming.

The use of `goto` is prohibited

http://xkcd.com/292

For more information read the following article by E. W. Dijkstra.