Operators and control structures in C

Lecture Topics

- Operators
- Conditional constructs
- Example

Lecture materials

Textbook § 12.3., 12.4, 13.1, 13.2

Homework

None

Machine problem

MP1.1 due February 2 at 5pm submitted electronically

Operators

- Operators are the language's mechanisms for manipulating values
  - Example: z = x * y;
    - Expression x * y is evaluated and
    - the resulting value is assigned to variable z
- three things to know about operators:
  - function – what it does
  - precedence – in which order are operators combined
  - associativity – in which order operators of the same precedence are executed

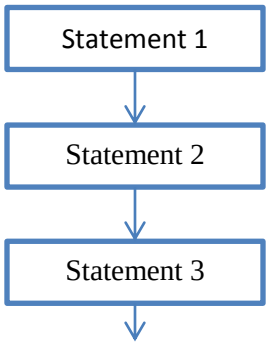
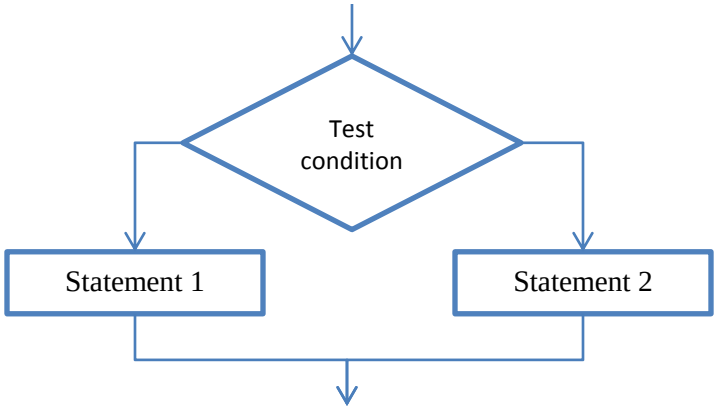
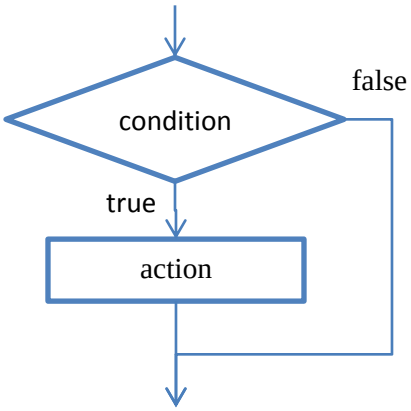
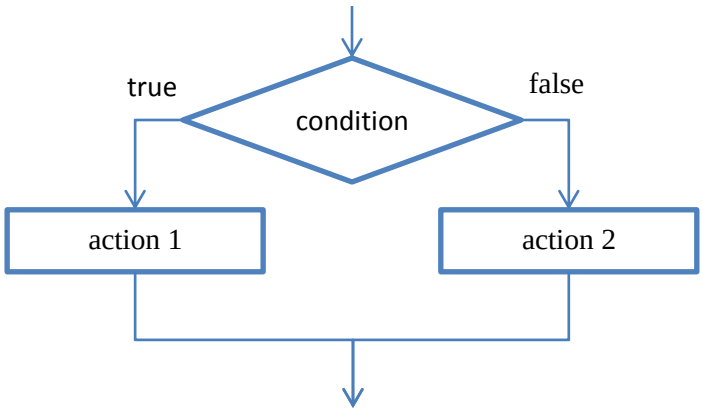
Assignment operator

- **=** is the assignment operator
  - The value on the right of the assignment operator will be assigned to the variable whose name is provided on the left side of the operator
    - The value actually will be copied to the memory associated with the variable name on the left of the assignment operator
  - Example: a = b + c;
- If the type of the value on the right of the assignment operator is not the same as the type of the variable on the left of the assignment operator, data type conversion will take place
- Example:
  - ```
    int x, a;
    double y, z;
    z = x + y;
    ```
    <- value of x will be converted to double type before the expression is evaluated
    ```
    a = x + y;
    ```
    <- once evaluated, the sum will be converted to int type before assigning it to variable a

Arithmetic operators

- Defined for **int**, **float**, and **char** data types
- *, +, -, /, % (modulus)
- Precedence: *, /, % are executed first, followed by + and –
  - 2+3x4 = 2+(3x4)
- Associativity: operators with the same precedence are executed sequentially
  - 2+3-4+5=((2+3)-4)+5
- Parenthesis can be used to override the evaluation order
- While floating-point addition and multiplication are both commutative (a + b = b + a and a*b = b*a), they are not necessarily associative. That is, (a + b) + c is not necessarily equal to a + (b + c).

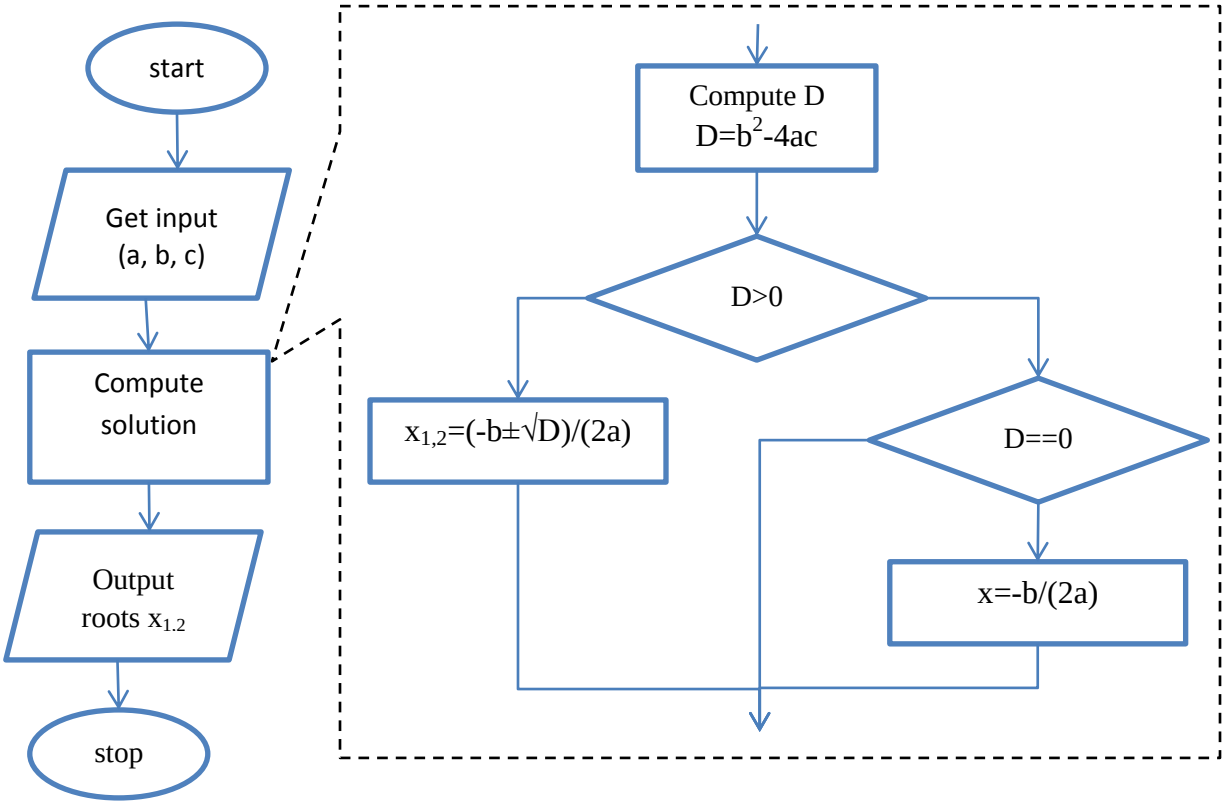
(Post/pre-) in-/de-crement

- Borrowed from assembly
- ++, --
    ```

- Examples
  - x = 4; y = x++; /* post increment: y = 4, x = 5 */
  - z = ++x; /* pre-increment: z = 6, x = 6 */

Bitwise operators

- normally used with **unsigned int** and **unsigned char** data types, they really just manipulate bits without regards to the numerical value/meaning
- ~ bitwise NOT: ~x
- & bitwise AND: y & x
- | bitwise OR: x | y
- ^ bitwise XOR: x ^ y
- << left shift: x << y (for positive int same as $x*2^y$)
- >> right shift: x >> y (for positive int same as $x/2^y$)
- Examples
  - h = f & g;
  - h = g << 4;
  - h = ~f | ~g;

Relational operators

- > less
- >= less or equal
- < more
- <= more or equal
- == equal
- != not equal
- Examples
  - q = (321 == 83); /*q = 0 */
  - q = (x == y); /* q is one if x == y, otherwise it is 0 */
  - h = f <= g; /* h is 1 when f is less of equal g */

Logical operators

- value of 0 is referred to as logically false
- value of ~0 is referred to as logically true
- ! - logical NOT
- && logical AND
- || logical OR
- Example: `y = ( 10 <= x) && (x <= 20)` /* true if 10 <=x<=20, otherwise false */

Expressions with multiple operators

- Example: y = x & z + 3 || 9 – w % 6;
- Applying operators precedence rules, this is equal to
  - y = (x & (z + 3)) || (9 – (w%6));

- Table 12.5 from the textbook lists operators precedence sequence
- To be sure the code does what you want it to do, just use parenthesis!

Special operator: conditional

- x = y ? z : t;
- if y is TRUE, x = z, otherwise, x = t
    - works like a MUX
- example: x = (y > 0) ? y : -y; /* x = |y| */

Compound-assignment operators

- *var1* **operator=** *var2*; in C is the same as *var1* = *var1* **operator** *var2*;
- += addition assignment, **a += b** same as **a = a + b**
- -= subtraction assignment
- *= multiplication assignment
- /= b division assignment
- %= b modulo assignment
- &= b bitwise AND assignment
- |= b bitwise OR assignment
- ^= b bitwise XOR assignment
- <<= b bitwise left shift assignment
- >>= b bitwise right shift assignment

Conditional constructs

- There are three basic programing constructs: sequential, conditional, iterative
- Sequential construct means that C program instructions (statements) are executed sequentially, one after another:



- Conditional construct means that one or another statement will be executed, but not both, depending on some condition:



- In C, conditional constructs can be implemented using **if**, **if-else**, or **switch** statements

if statement

- ```
  if (condition)
      action;
  ```



- examples
  - ```
    if (x < 0)
        x  = -x;                    /* simple statement */
    ```
  - ```
    if (x > 5 && x< 25) {
        y = x * x +5;               /* compound statement */
        printf("y=%d\n", y);
    }
    ```
- action statement can be simple, as in fist example, or compound, as in second example

if-else statement

- ```
  if (condition)
      action_when_condition_is_true;
  else
      action_when_condition_is_false;
  ```



- examples
  - ```
    if (x < 0)
        x  = -x;
    else
        x = x * 2;
    ```
  - ```
    if (x > 5 && x< 25) {
        y = x * x +5;
        printf("y=%d\n", y);
    }
    else
        printf("x=%f\n", x);
    ```
- associating **if**s with **else**s
  - in a cascaded **if-else** statement, an **else** is associated with the closest **if**
    - that is, when not using braces, which is not a good practice

``if (x != 0)
  if (y > 3)
  z = z / 2;
  else
  z = z + 2;```
	same as	```if (x != 0) {
		  if (y > 3)
		  z = z / 2;
		  else
		  z = z + 2;
		}```

  - if we really want to associate **else** with the first **if**, then we should use braces:
    - ```
      if (x != 0) {
          if (y > 3)
              z = z / 2;
      }
      else
          z = z + 2;
      ```
  - use braces to write clear and readable code!
- common programming errors
  - if (x = 2) using assignment operator instead of ==
      ```

Example

Computing solution of a quadratic equation $ax^2+bx+c=0$

- **Algorithm:**
  - $D = b^2 - 4ac$
  - If D equals 0, there is one real root: $x = -b/(2a)$
  - If D is positive, there are two roots: $x_{1,2} = (-b \pm \sqrt{D})/(2a)$
  - If D is negative, no real roots exist
- Problem decomposition into separate steps using a flowchart
  - Get input
  - Compute solution according to the above algorithm
  - Print output



- Once we have the problem decomposed into individual steps, translating it into C is straightforward:

```
/* solution of the quadratic equation ax^2+bx+c=0
 */

#include <stdio.h>        /* needed for printf and scanf */
#include <math.h>         /* needed for sqrtf */

int main()
```

```
{
    float a, b, c;   /* quadratic equation coefficients */
    float D;         /* determinant */
    float x1, x2;    /* solution(s) */

    /* get equation coefficients */
    printf("Enter a, b, and c: ");
    scanf("%f %f %f", &a, &b, &c);
    printf("Solving equation %fx^2+%fx+%f=0\n", a, b, c);

    /* compute solution */
    D = b * b - 4 * a * c;       /* compute determinant */

    if (D > 0)                   /* two real roots exist */
    {
        x1 = (-b + sqrtf(D)) / (2 * a);
        x2 = (-b - sqrtf(D)) / (2 * a);
    }
    else if (D == 0)             /* only one root exists */
        x1 = -b / (2 * a);

    /* print results */
    if (D > 0)
        printf("x1=%f, x2=%f\n", x1, x2);
    else if (D == 0)
        printf("x=%f\n", x1);
    else
        printf("No real roots exist\n");

    return 0;
}
```

- To compile, we will need to link the code with additional library (libm.a) using **-lm** compiler flag
- Examples:
  - $x^2+2x-8=0$: $x_1 = 2$, $x_2 = -4$
  - $x^2-10x+25=0$: $x = 5$
  - $5x^2-2x+2=0$: no real roots