A C Primer (3): Flow of Control (ABC Chapter 4)

Ion Mandoiu
Laurent Michel
Revised by M. Khan, J. Shi and W. Wei

Flow of Control

- Statements are normally executed sequentially
- For **selective** or **repeated** execution we have all the usual suspects from Java/C++/Python
  - blocks
  - if and if-else
  - while
  - for
  - switch
  - break
  - continue

Blocks (compound statements)

- List of statements enclosed by { and }
  - Considered as a single statement
  - No semicolon after closing }
  - Can be empty
  - Can be nested (block in block)
  - Useful for branching/loop statements
  - Can define variables at the beginning of blocks
    - Can mix declarations and code in c99

```
{
    int a, b;
    a = 3;
    b = a * 10;

    // C99
    int c;
    c = a * b;
}
```

Comparison and logical operators

- Comparison operators that compare two expressions
  - Pay attention to types!

```
==      !=      >      <      >=      <=
```

- Logical operators

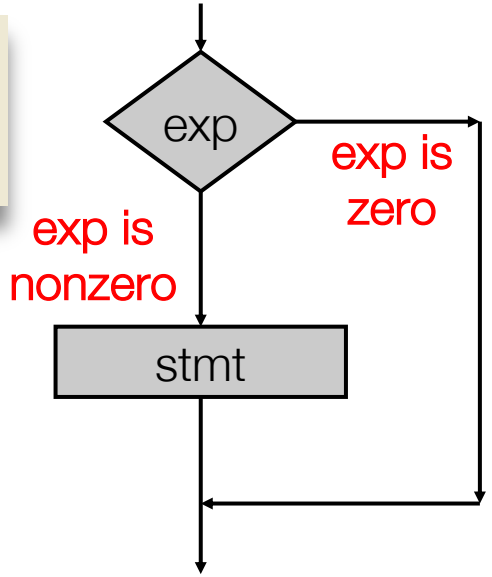
```
&&      ||      !
```

- The result is either 0 or 1 (of int type)
  - Again, 0 means false and 1 means true

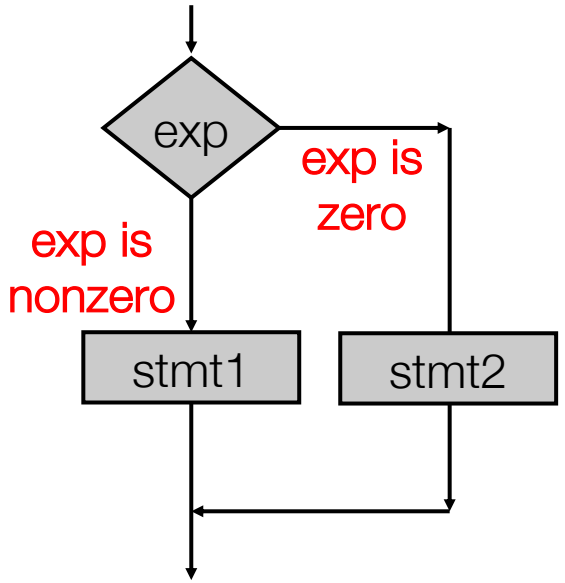
See the textbook for precedence and associativity

Branching: if and if-else

```
if (<exp>)
   <stmt>
```




```
if (<exp>)
   <stmt1>
else
   <stmt2>
```



- "exp" is typically a comparison or logical expression, but can be ANY expression (float/double, pointer, …)
- The statements can be compound statements (blocks)
- Or other if statements!
  - Beware of the dangling else ("else" matches the nearest preceding "if", **use blocks to disambiguate**)

Example: min

```
int  i, j, min;


if (i < j)
    min = i;
else
    min = j;
// Indentation is not required, above 4 lines are the same as
if (i < j) min = i; else min = j;
```

Example: if-else statement with blocks

```
int  i, j, k;


if (i < j) {
    k = i;
    printf("i is selected.\n");
}     // no ; here
else {
    k = j;
    printf("j is selected.\n");
}
```

Ternary operator

- Takes **three** expressions as operands

```
exp1 ? exp2 : exp3
```

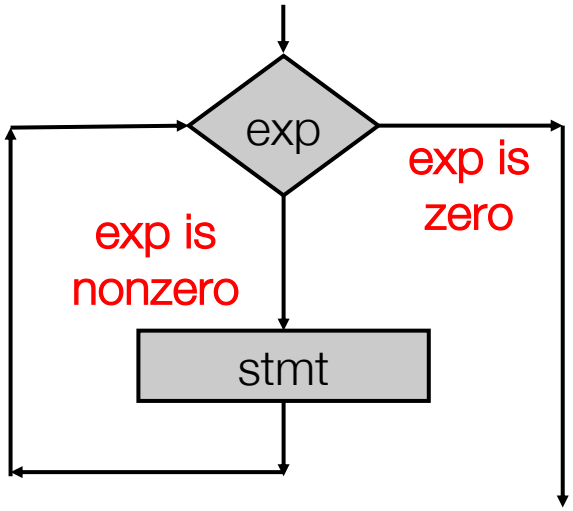
  - exp1 is evaluated first
  - If exp1 is non-zero (true), exp2 is evaluated and its value is used as the value of the ternary expression
  - If exp1 is zero (false), exp3 is evaluated and its value is used as the value of the ternary expression

- Example:

```
min = i < j ? i : j;
```

While Loop

```
while (<exp>)
    <stmt>
```



- Example: computing sum of 0..99

```
int  i = 0, sum = 0;
while (i < 100) {
    sum = sum + i;
    i++;
}
// Same as
while (i < 100)  sum += i++;
```

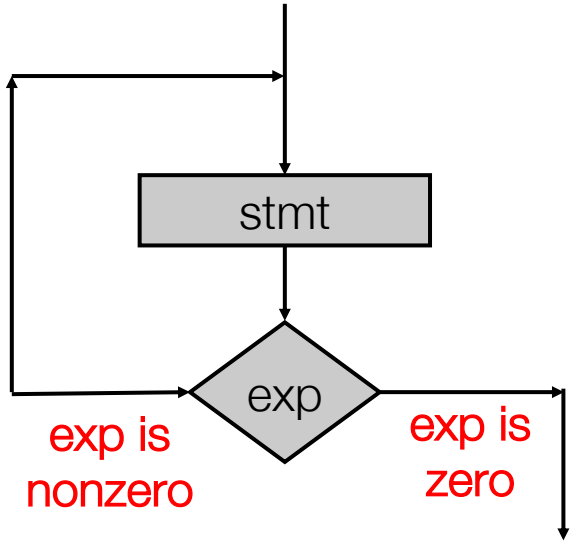
Do-While Loop

- Checks condition **after** executing loop body
  - The statement is executed at least once

```
do
   <stmt>
while (exp);
```

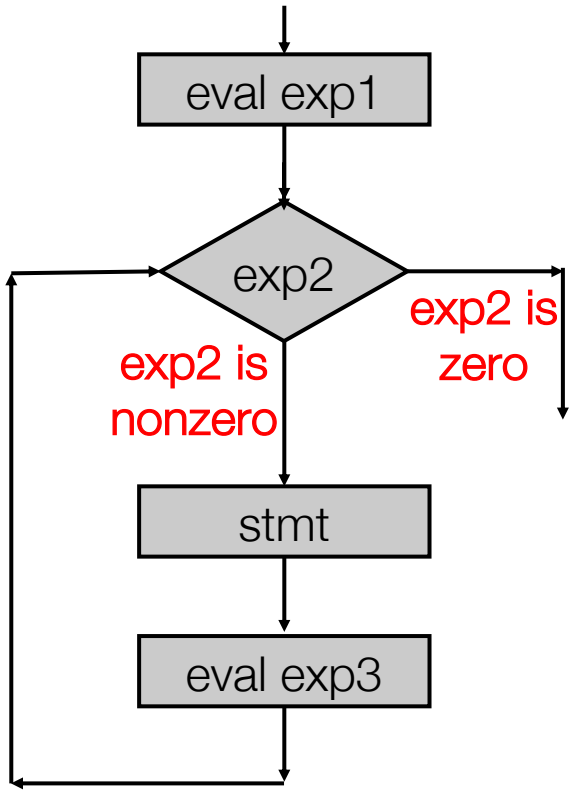
- Example: computing sum of 0..99

```
int  i = 0, sum = 0;
do {
     sum = sum + i;
     i++;
} while (i < 100);
```



For Loop

```
for( <exp1>; <exp2>; <exp3>)
    <stmt>
```



- Sometimes called “counting” loop
  - More like swiss-army knife!
- Three expressions:
  - Initialization, condition, increment
- Equivalent to

```
exp1;
while (exp2) {
  <stmt>
   exp3;
}
```

Computing sum of 0..99 using for

```
int  i, sum;
// one way
sum = 0;
for (i = 0;  i < 100;  i++) sum = sum + i;

// another way, with all initializations inside
for (sum = i = 0;  i < 100;  i++) sum += i;

// yet another way, with empty body
for (sum = i = 0;  i < 100;  sum += i++);

// yet another, using comma operator
for (sum = 0, i = 0;  i < 100;  sum += i, i++);
```

Comma operator

- Takes two expressions

```
exp1 , exp2
```

  exp1 is evaluated first, then exp2 is evaluated

  exp2 is the result of the whole expression

- Has the lowest precedence
- Associates from left to right

```
exp1, exp2, exp3  ←→  (exp1, exp2), exp3
```

  - Order can make a difference, e.g.,

```
for (sum = 0, i = 0;  i < 100;  sum += i, i++);
```

  is not the same as

```
for (sum = 0, i = 0;  i < 100;  i++, sum += i);
```

Multiway branching using “else if” …

```
// Assume all variables are defined as int
…
if (i == 0)
    n0++;
else if (i == 1)
    n1++;
else if (i == 2)
    n2++;
else
    n_other++;
```

Switch

- Also called “selection” statement

```
switch (<integer expression>) {
    case <integer constant1>:
        <statements>
    case <integer constant>:
    case <integer constant>:
        <statements>
    default:
        <statements>
}
```

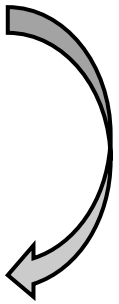
Switch example

```
// Assume all variables are defined as int
switch (i) {
    case 0:
        n0++; break; // Note the break statement
    case 1: // No break for case 1. Will continue.
    case 2:
    {   // Can put a block here and define new variables
        int a = d + 10;
        n1 = a * 10; break;    }
    default:
        n_other++;
}
```

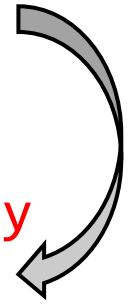
Break Statement

- Most commonly used in switch statements
  - Prevents “fall-through” to the next case
- Also works in loops (for, while, do-while)
  - Loop execution terminated immediately, control resumes at statement immediately following the loop

```
switch (a) {
  …
    break;
  …
} // end of switch
…
```



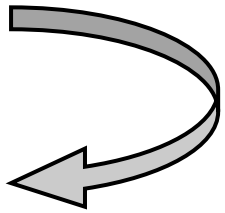
```
{ // begin loop body
  …
  break;
  …
} // end loop body
…
```



Continue Statement

- Skip the rest of current loop iteration and continue to the next one
- Can be used within for, while, and do-while loops
  - Can appear in a nested if / else
  - If used in nested loops, it applies to the “innermost” enclosing loop
  - For “for” loops, go to the evaluation of the “increment” expression

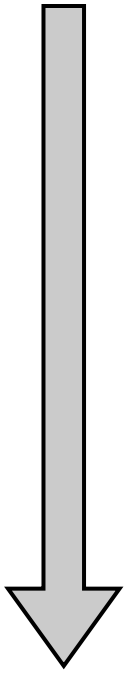
```
{ // begin loop body
  …
  continue;
  …
} // end loop body
```



- Study the remaining slides yourself

Operators so far

Most



Least

Operator precedence and associativity	
Operators	Associativity
() ++ (*postfix*) -- (*postfix*)	left to right
+ (*unary*) - (*unary*) ++ (*prefix*) -- (*prefix*)	right to left
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= *etc*	right to left
, (*comma operator*)	left to right

Shortcut evaluation of logical expressions

- For expressions that contain logical operators && and ||, the evaluation **stops** as soon as the outcome true or false is known:

```
exp1 && exp2
// Evaluate exp1. If false, exp2 is not evaluated.

exp1 || exp2
// Evaluate exp1. If true, exp2 is not evaluated.
```

- Makes it safe to write code like

```
if (a != 0 && c == b/a) … // no divide by 0 error
```

Avoid these common pitfalls

- Confusing assignments and tests for equality

```
x=8 vs. x==8
```

- Off by 1 errors in counting loops

```
for (int i=0; i <= n; i++) …
```

- Confusing logical and bitwise ops

```
x && y  vs.  x & y
```

- Forgetting the "break" statements in a switch
- Dangling else in nested if-then-else

Dangling else

```
if (a) if (b) s1++; else s2++; // Avoid this

// which 'if' is 'else' associated with?

if (a) {if (b) s1++; else s2++;} // option 1

if (a) {if (b) s1++;} else s2++; // option 2

// C chooses option 1
```

Comparing if-then in C and Python

- Indentation carries no information in C
- Block structure is *explicit,* using { }
- Condition **must be** in parentheses
- C version

```
if (n==0) return 1; else return n*fact(n-1);
```

- Python version

```
if n==0:
   return 1
else:
   return n * fact(n-1)
```

```
// Easier to read
if (n == 0)
    return 1;
else
    return n * fact(n - 1);
```

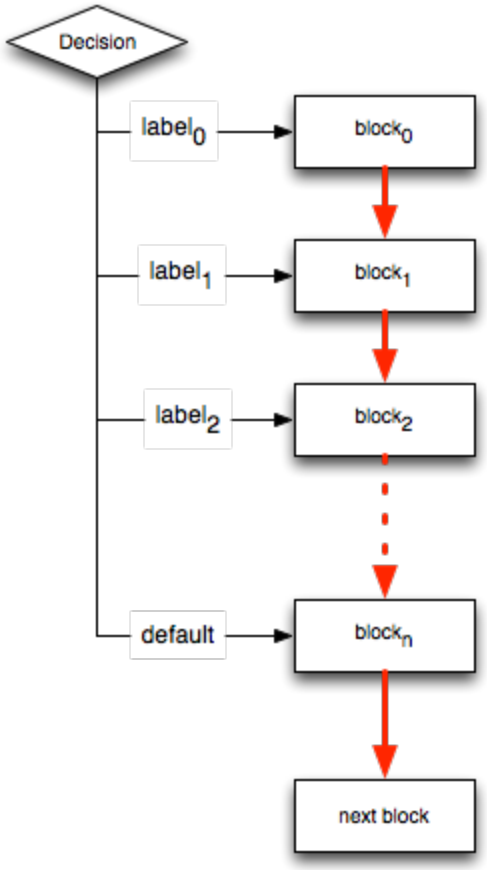
Declaring variables in “for” (C99)

```
// C99 allows variable declarations in init expression
// i lives in the loop. It is undefined after the loop


int  sum = 0;
for (int i = 0; i < 100; i ++)
    sum += i++;
```

Example: computing the sum of odd integers

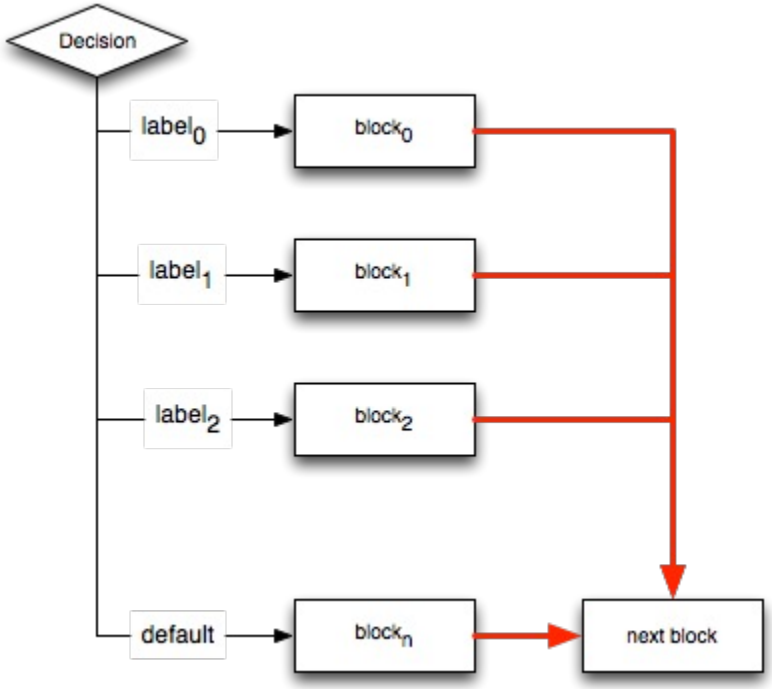
```
int  i, sum;


sum = 0;
for (i = 1; i < 100; i++)
    if (i % 2) sum += i;


// (i % 2) is the same as (i % 2 != 0)
// the condition can also be (i & 1)
```

Switching




```
switch(expression) {
   case label0: <block0>
   case label1: <block1>
   …
   default: <blockn>
}
```

- This is ***usually*** what you mean




```
switch(expression) {
   case label0: <block0>;break;
   case label1: <block1>;break;
   …
   default: <blockn>
}
```

Don't forget the breaks

The forbidden One: goto

```
   goto <label>;
…
label:
   <block>
```

- Forget you ever knew goto existed
- This is a terrible control primitive
- Don’t ever use it.
- Really.
- Seriously.
- Nothing good will come out of it