CSE 3100: Systems Programming

Lecture 3: Flow of Control

Department of Computer Science and Engineering
University of Connecticut

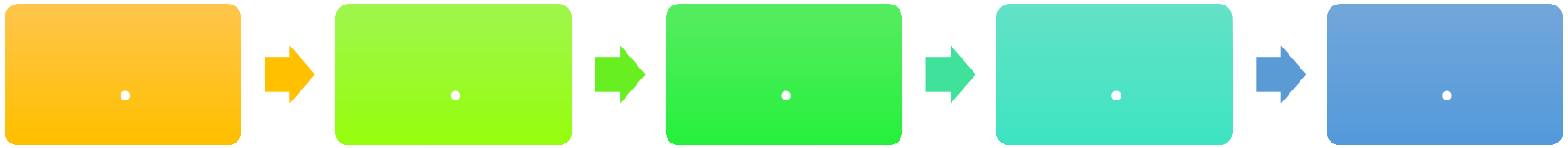
1. Control Flow and Blocks

2. If Statements

3. While/For Statements

4. Switch Statements and Common C Mistakes

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 }

```c
#include <stdio.h>
int main()
{
    //some block of code
    {
        int c;
        c = 5 * 3;
        printf("The value of c is=%d\n", c);
    } //notice the code block end doesn't need a semicolon


    return 0;
}
```

```
The value of c is=15
```

What is the purpose of block statements?

- To use local variables which cannot be accessed outside of the block:

```c
#include <stdio.h>
int main()
{
    //some block of code
    {
        int c;
        c = 5 * 3;
        printf("The value of c is=%d\n", c);
    } //notice the code block end doesn't need a semicolon
    printf("The value of c is=%d\n", c);
    return 0;
}
```

This code will actually not even compile.
“c undeclared identifier”

A small historical note:

In C89 ALL variables had to be declared BEFORE usage in a code block:

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

In C99 (the version we use) this is not the case:

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

Other Notes on Blocks

C Code:

Blocks of C Code:

- 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

~~1. Control Flow and Blocks~~

2. If Statements

3. While/For Statements

4. Switch Statements and Common C Mistakes

Comparison and logical operators

- Operators are commonly used inside of if and else statements.
- Comparison operators that compare two expressions
  - Pay attention to types!

  == != > < >= <=

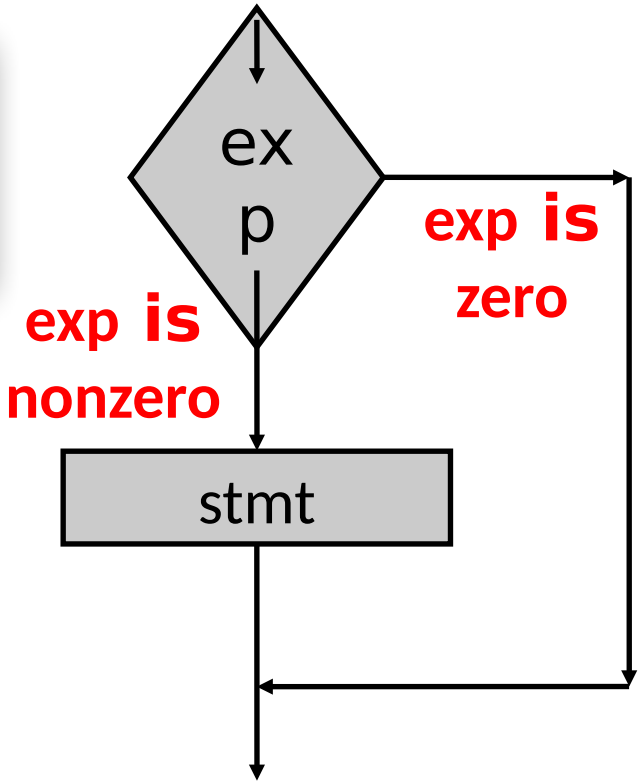
- Logical operators

  && || !

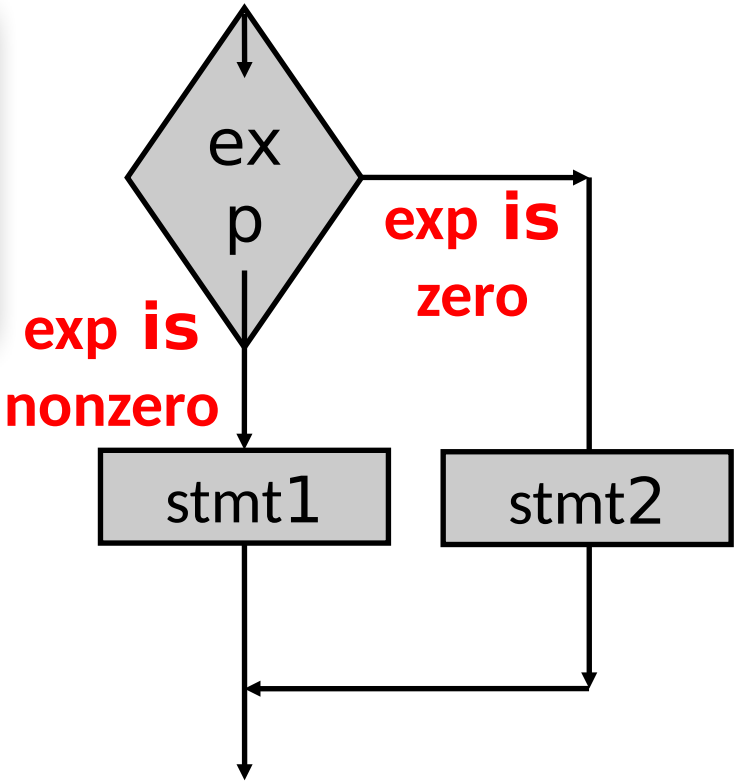
- The result is either 0 or 1 (of int type)
  - Again, 0 means false and 1 means true

Branching: if and if-else

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




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



If 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;
```

Branching: if and if-else

```
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: 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");
}
```

Bad Coding: Dangling else (1/4)

- Given the following piece of code:

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

- Assume a is true and b is false. Should we do s1++ or s2++?

Bad Coding: Dangling else (2/4)

Assume a is true and b is false. Should we do s1++ or s2++?

```c
#include <stdio.h>

int main(void){
    //starting variables
    int s1 = 0;
    int s2 = 0;
    int a = 1;
    int b = 0;
    if (a) if (b) s1++; else s2++;
    printf("s1=%d\n", s1);
    printf("s2=%d\n", s2);
```

```
kaleel@CentralCompute:~$ ./test
s1=0
s2=1
```

Bad Coding: Dangling else (3/4)

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

Instead write more understandable code:

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

if (a) {if (b) s1++; else s2++;}
```

But could we write the above in an even more readable way?

Bad Coding: Dangling else (4/4)

The even better way to write the statement:

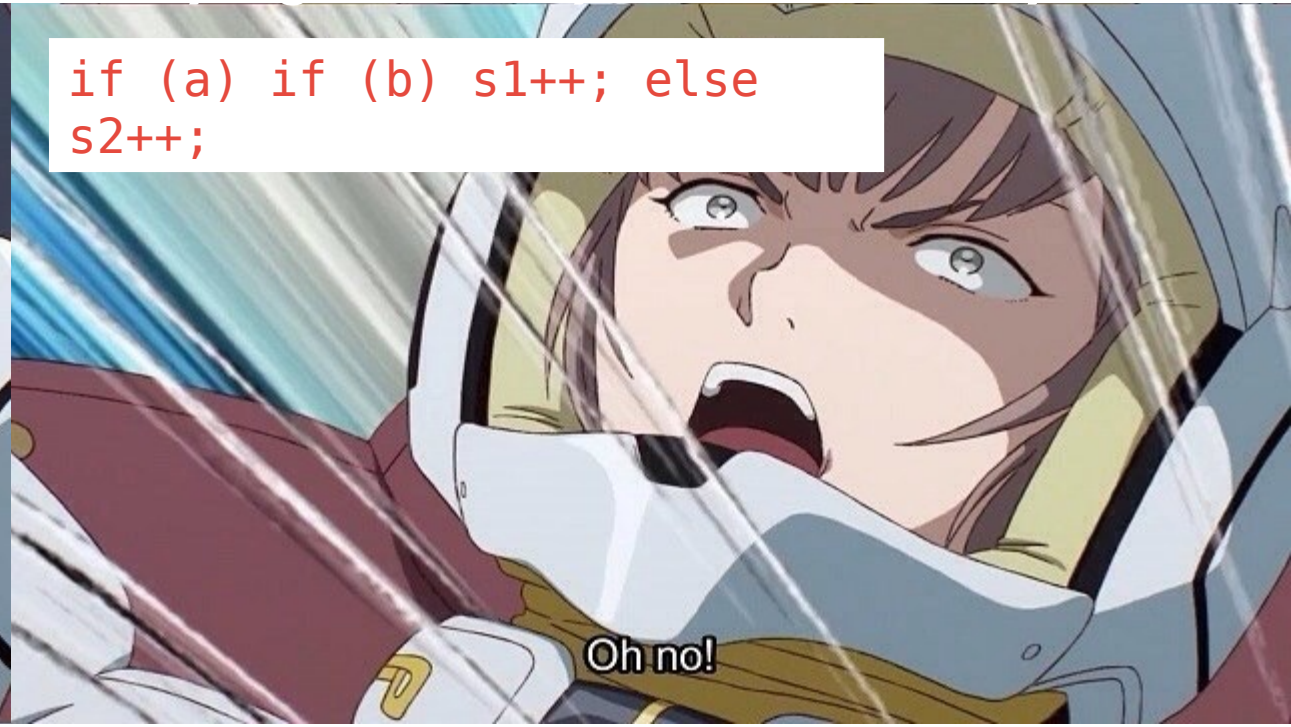
BAD

```
if (a) if (b) s1++; else s2++;
printf("s1=%d\n", s1);
printf("s2=%d\n", s2);
```

BETTER

```
if(a){
    if(b){
        s1++;
    }
    else{
        s2++;
    }
}
printf("===========\n");
printf("s1=%d\n", s1);
printf("s2=%d\n", s2);
```

Why do we care so much about code **readability** in C?



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;
```

Ternary Operator Coding Example

```c
#include <stdio.h>

int main(void){
    int i = 5;
    int j = 1;
    int min = i < j ? i : j;
    printf("The value of min is: %d\n", min);
    return 0;
}
```

```
kaleel@CentralCompute:~$ gcc hello2023.c -o test
kaleel@CentralCompute:~$ ./test
The value of min is: 1
```

Multi-way 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++;
```

~~1. Control Flow and Blocks~~

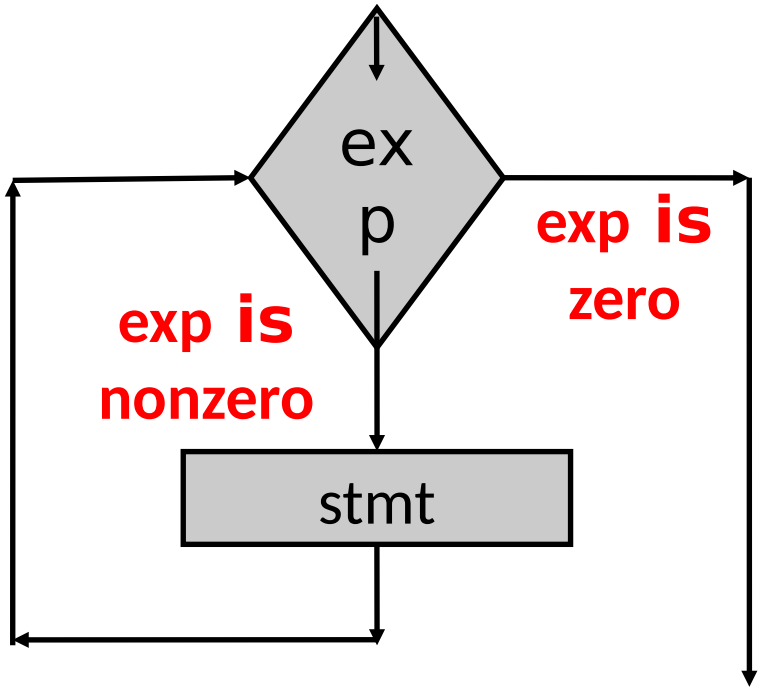
~~2. If Statements~~

3. While/For Statements

4. Switch Statements and Common C Mistakes

While Loop

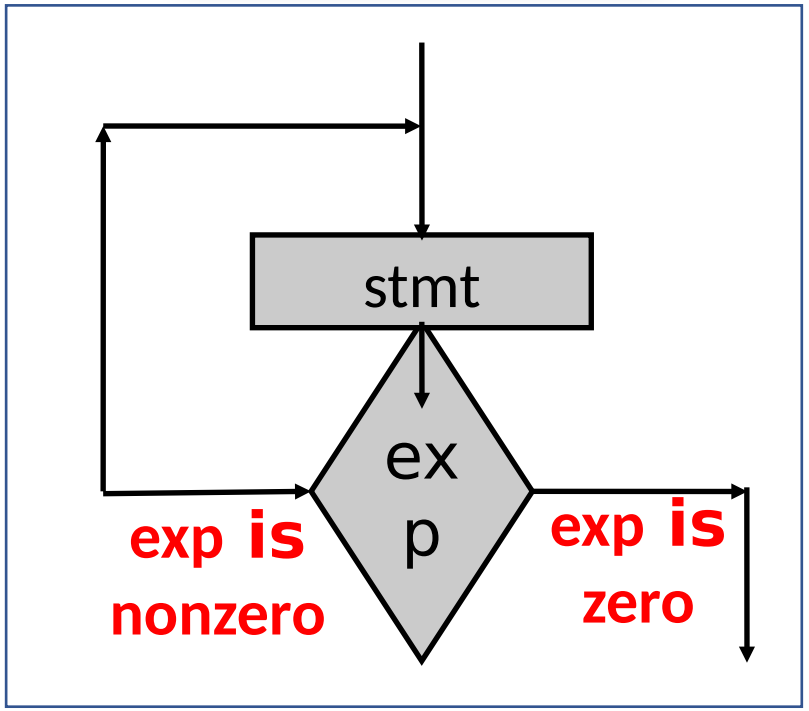
```
while (<exp>)
    <stmt>
```




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

Do-While Loop

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



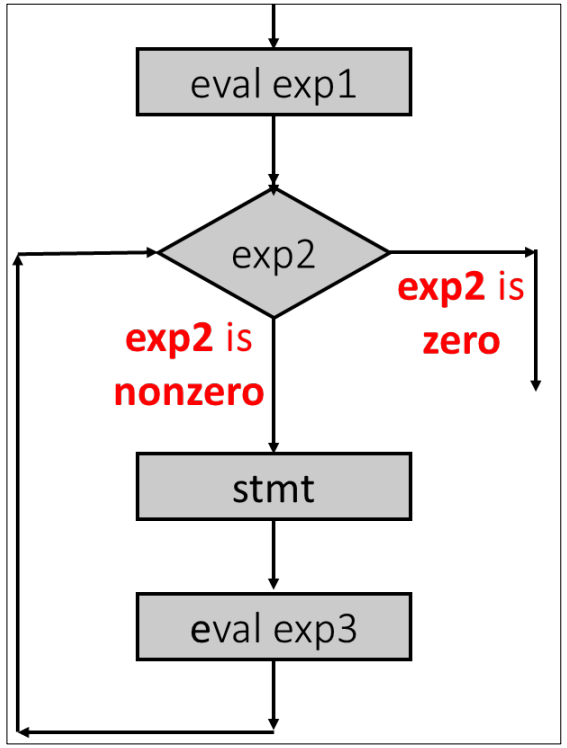
- Checks condition **after** executing loop body
  - The statement is executed at least once
- Example: computing sum of 0..99

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

While vs Do-While Loop

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++);
```

For Loop Coding Example (Compact)

```c
#include <stdio.h>

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

All variables for the “for loop” are declared with the loop.

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);
    ```

~~1. Control Flow and Blocks~~

~~2. If Statements~~

~~3. While/For Statements~~

4. Switch Statements and Common C Mistakes

Switch Statements

- 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 ++;
}
```

Where would we need to use switch?

- Consider the scenario where we need to test the output of some operation with a lot of different outcomes.
- We could write it using if and else if…

```c
#include <stdio.h>
int main()
{
    int n = 0x123ABC;

    //We want to print out the last digit of the hex number n
    //Below is the first implementation using if else statements

    if(n % 16 == 0)
        printf("The last digit of the hex number %0x is 0.\n", n);
    else if(n % 16 == 1)
            printf("The last digit of the hex number %0x is 1.\n", n);
    else if(n % 16 == 2)
            printf("The last digit of the hex number %0x is 2.\n", n);
        else if(n % 16 == 3)
            printf("The last digit of the hex number %0x is 3.\n", n);
        else if(n % 16 == 4)
            printf("The last digit of the hex number %0x is 4.\n", n);
        else if(n % 16 == 5)
            printf("The last digit of the hex number %0x is 5.\n", n);
        else if(n % 16 == 6)
            printf("The last digit of the hex number %0x is 6.\n", n);
        else if(n % 16 == 7)
            printf("The last digit of the hex number %0x is 7.\n", n);
        else if(n % 16 == 8)
            printf("The last digit of the hex number %0x is 8.\n", n);
        else if(n % 16 == 9)
            printf("The last digit of the hex number %0x is 9.\n", n);
        else if(n % 16 == 10)
            printf("The last digit of the hex number %0x is A.\n", n);
        else if(n % 16 == 11)
            printf("The last digit of the hex number %0x is B.\n", n);
```

Where would we need to use switch?

```c
#include <stdio.h>
int main()
{
    int n = 17;
    switch (n % 16)
    {
        case 0:
            printf("The last digit of the hex number %0x is 0.\n", n);
            break;
        case 1:
            printf("The last digit of the hex number %0x is 1.\n", n);
            break;
        case 2:
            printf("The last digit of the hex number %0x is 2.\n", n);
            break;
        case 3:
            printf("The last digit of the hex number %0x is 3.\n", n);
            break;
        case 4:
            printf("The last digit of the hex number %0x is 4.\n", n);
            break;
        case 5:
            printf("The last digit of the hex number %0x is 5.\n", n);
            break;
```

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
```

Common Mistakes in C coding 1

- Confusing assignments and tests for equality

  `x=8 vs. x==8`

- Confusing logical and bitwise ops

  `x && y vs. x & y`

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

Common Mistakes in C coding 2

- Looping the right amount of times. Let's say we want to loop 10 times:

```c
#include <stdio.h>
int main()
{
    //loop 10 times
    for (int i = 0; i <= 10; i++) {
        printf("%d\n", i);
    }
}
```

```
0
1
2
3
4
5
6
7
8
9
10
```

Figure Sources

- https://preview.redd.it/rvir9ttjg1a11.png?auto=webp&s=69a4976a0df629d77c43497b9358d9a3a918f571
- https://static.wikia.nocookie.net/tomandjerry/images/2/2b/Tom-de-brazos-cruzados11578470925.jpg/revision/latest/scale-to-width-down/1200?cb=20210224013420
- https://preview.redd.it/sllzna67f6k61.jpg?auto=webp&s=edfe83e6b7fd4c9f0dd51c3c72d9be2facd310b3
- https://i.imgflip.com/3efwu7.jpg?a470208
- https://preview.redd.it/6wksqjmmyw321.jpg?auto=webp&s=aceebdeb23af98598c2508b42f77debbcd36cf4b