

Introduction to C, C++, and Unix/ Linux

CS 60

Lecture 6: Control Flow

Today

→ C control flow

- Reading for next time K&R ch. 1 – 4 & 7.1-7.4

Note: Lexical elements of C

- **Keywords**
 - Reserved words that may not be used for anything else
 - **Identifiers**
 - Variable names, function names...
 - **Constants**
 - E.g., the number 5
 - **String constants**
 - E.g. “Hello, world\n”
 - **Operators**
 - E.g., +, -, =, ++
 - **Punctuators**
 - E.g., {} () ; ,
- These are the basic tokens that the compiler cares about

Control flow

- Control flow affects the order in which statements are executed

if, if-else, if-else if

switch

for, while, do

break, continue, goto

Expressions and statements

- An expression has a value

42

x+1

myfunc(a, b)

printf("Hi")

x = 7

i=0, j=1, k=2

- A statement does not have a value

42;

x+1;

myfunc(a, b);

printf("Hi");

x = 7;

i=0, j=1, k=2;

;

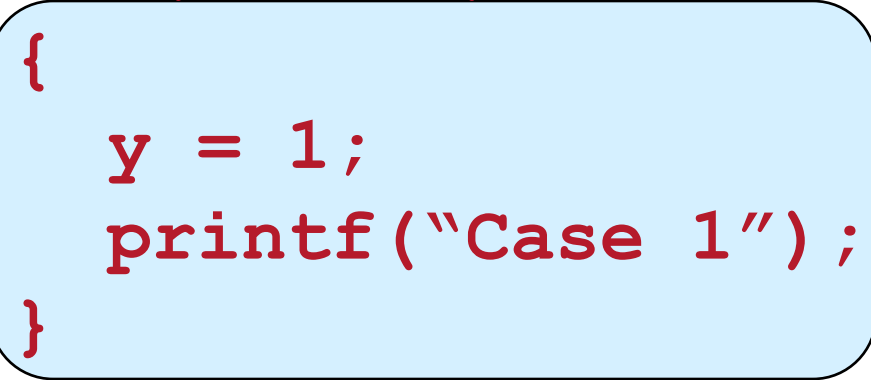


Compound Statement

- A **compound statement** or **block** groups zero or more statements surrounded by braces { }

```
if (x == 0)
  y = 1;
z = x + y;
```

```
if (x == 0)
{
  y = 1;
  printf("Case 1");
}
z = x + y;
```



If

if (expr)

statement

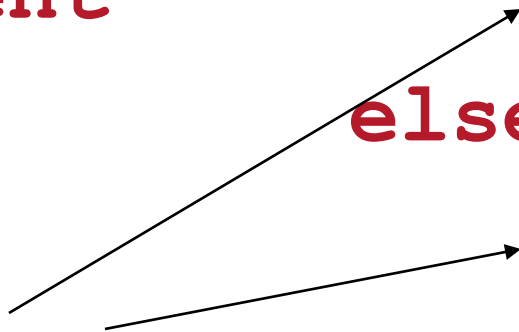


if (expr)

statement₁

else

statement₂



Single or compound statement { }

```
if (x) z = y/x;
if (x==12) fct(x);
if (!happy) smile();
if (a+1 && b && c)
    x = a*b*c;
if (!strcmp(str1, "-h"))
    help = 1;
else
    help = 0;
```

What's wrong here?

```
if (a && b && c) {
```

```
    x = a*b*c;
```

```
    y = a+b+c;
```

```
};
```

← Compiler error

```
else
```

```
    x = y = 0;
```

If-else-if

```
if (expr1)
    statement1
else if (expr2)
    statement2
else if (expr3)
    statement3
else
    statement4
```

```
if (x == 1)
    return(0);
else if (x == 2)
    return(1);
else
    return(
        fib(x-1) +
        fib(x-2)
    );
```

Same as this:

```
if (expr1)
    statement1
else
    if (expr2)
        statement2
    else
        if (expr3)
            statement3
        else
            statement4
```

Matching *ifs* to *elses*

```
if (a == 1)
if (b == 2)
printf ("***\n");
else
printf ("###\n");
```

?

```
if (a == 1)
if (b == 2)
printf("***\n");
else
printf("###\n");
```

```
if (a == 1)
{
    if (b == 2)
        printf("**\n");
    else
        printf("##\n");
}
```

```
if (a == 1)
{
    if (b == 2)
        printf("**
\n");
}
else
printf("##\n");
```



Use braces { } to clarify for readability!

Formatting danger

- Warning: C doesn't understand pretty formatting (proper indentation)
 - Remember, white space is mostly ignored
- So this won't help:

```
if (a == 1)
    if (b == 2)
        printf("***\n");
else
    printf("###\n");
```

```
double x=4.2, y=42, z;
```

```
z = x - y/10;
```

```
printf("%f\n", z);
```

0.000000

```
if (z == 0.0)
```

```
    printf("ZERO");
```

```
else
```

```
    printf("NON-ZERO");
```



Comparing floating point numbers

- Tiny rounding errors can make seemingly equal floating point numbers different

z is 0.0000000000000000001778091563...


- Instead of **(x == y)**, define a small value such as

eps = 1.0e-10

and use

if (fabs(x-y) < eps) ...

Floating-point absolute value



Switch

- Multi-way conditional statement. All statements are executed beginning with the first match (not the same as “if-else if-else if-...”)

```
Integer → switch (expr) {
                Constant integers → case (c1): statement1
                                     case (c2): statement2
                                     ...
Optional → default: statementn
                }
```

However **break** statements prevent “falling through” to subsequent statements

All statements are executed beginning with the first match (not the same as “if-else if-else if-...”)

```
switch (expr) {  
    case (c1): statement1; break;  
    case (c2): statement2; break;  
    case (c3): statement3; break;  
    case (c4): statement4; break;  
    default: statement5 ; break;  
}
```

Match →

statement₂
statement₃
statement₄
statement₅

Switch example

```
switch (c) {  
    case 'a': lcase_a++;  
    case 'A': ++a_cnt;  
        break;  
    case 'b': lcase_b++;  
    case 'B': ++b_cnt;  
        break;  
    default: ++other_cnt;  
        break;  
}
```

← Why this **break**?

Fibonacci example

```
switch (x) {  
  case 0: val = 0;  
          break;  
  case 1: val = 1;  
          break;  
  default:  
          val = fib(x-1)  
              + fib(x-2);  
          break;  
}
```

Common `switch` mistake

```
switch (x) {  
    case 0: val = 0; ← Forgot break  
    case 1: val = 1;  
             break;  
    default:  
             val = fib(x-1)  
                 + fib(x-2);  
             break;  
}
```

While

```
while (expr)  
    statement
```

- Executes **statement** (which may be a compound statement) over and over, as long as **expr** remains non-zero (TRUE)
- **expr** is checked first, and if it is zero, the **statement** is never executed

```
while (x == 0)
    ;
```

```
while (x==0) ;
```

```
while (x == 0)
{ }
```

But not:

```
while (x == 0)
{ };
```

Two statements. Actually valid.

```
while (i++ < 100)
    sum += i;
```

```
while (1) ;
```

```
while ((c = getchar()) != EOF)
{ /* read chars from file */ }
```

```
while (i--)
{ ... }
```

Problem?

```
while (i-- > 0)
{ ... }
```

What if **i** starts at **-3**?

What if the **while** loop modifies the value of **i**?

Do

Good form to always use braces
around the statement here



```
do  
    statement  
while (expr)
```

- Executes **statement** (which may be a compound statement) over and over, as long as **expr** remains non-zero (TRUE)
- **expr** is checked after the **statement** is executed, so **statement** will be executed at least once

For

```
for (expr1; expr2; expr3)  
    statement
```

This is equivalent to:

```
expr1;  
while (expr2) {  
    statement  
    expr3;  
}
```

```
expr1;  
if (expr2)  
    do statement  
    expr3;  
while (expr2)
```

```
for (i=0; i<ncols; i++)  
    for (j=0; j<nrows; j++)  
        image(i, j) = 0;
```

```
for ( ;; );
```

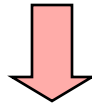
Same as...

```
for ( ; 1; );
```

```
int factorial(int n)  
{  
    int i, fact;  
    for (i=1, fact=1; i<=n; fact*=i, i++) ;  
    return(fact);  
}
```

What is the value of `npixels`? **1**

```
int npixels=0;
for (i=0; i<ncols; i++)
    for (j=0; j<nrows; j++)
        image[i][j] = 0;
        npixels++;
```



```
int npixels=0;
for (i=0; i<ncols; i++)
    for (j=0; j<nrows; j++)
        image[i][j] = 0;
npixels++;
```

Break and continue

- These interrupt the normal flow of control
 - **break** forces an exit from the innermost control loop
 - ◆ Quits the current for, while, do, or switch
 - **continue** causes the “next iteration” of the enclosing loop to begin
 - ◆ Skips the rest of the current loop
- Don't use these if you can avoid them
 - Can be awkward and confusing (except in **switch**)
 - But sometimes useful

```
while (c=getchar()) {  
    if (c == EOF) break;  
    printf("%c", c);  
}
```

```
for (i=0; i<filesize; i++) {  
    if (c=getchar() == '\n')  
        continue;  
    c = capitalize(c);  
    putchar(c);  
}
```

Better to use **if-else**

```
// When img[.][.][.] >= 0
```

```
for (i=0; i<rows; i++)
```

```
for (j=0; j<cols; j++)
```

```
for (k=0; k<colors; k++)
```

```
{
```

```
    if (!img[i][j][k])
```

```
        continue;
```

```
    if (img[i][j][k] <= -1)
```

```
        break;
```

```
    AddHist(img[i][j][k]);
```

```
}
```

i	j	k
0	0	0
0	0	1
0	0	2
0	0	3
0	1	0
0	1	1
0	1	2
0	1	3
0	2	0
0	2	1
.	.	.
.	.	.
.	.	.

Goto

- Read about it
- Don't use it (at least not in CS60) !
- Rare reasonable usage: To jump out of a deeply nested inner loop if some test fails