
CS240: Programming in C

Lecture 4: Operators and Expressions. Control Flow.

Example

```
#include<stdio.h>

int main() {
    int fahr, celsius;
    const int lower = 10, upper = 300, step = 10;

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5* (fahr - 32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }

    return 0;
}
```

Example

```
#include<stdio.h>
```

```
int main() {  
    int fahr, celsius;  
    const int lower = 10, upper = 300, step = 10;  
  
    fahr = lower;  
    while (fahr <= upper) {  
        celsius = 5* (fahr - 32) / 9;  
        printf("%d\t%d\n", fahr, celsius);  
        fahr = fahr + step;  
    }  
  
    return 0;  
}
```

10	-12
20	-6
30	-1
40	4
50	10
60	15
70	21
80	26
90	32
100	37
110	43
120	48
130	54
140	60
150	65
160	71
170	76
180	82
190	87
200	93
210	98
220	104
230	110
240	115
250	121
260	126
270	132
280	137
290	143
300	148

Operators

- Arithmetic: $+$, $-$, $*$, $/$, $\%$
- Relational: $<$, $>$, $<=$, $>=$, $!=$, $==$
- Logical: $||$, $\&\&$, $!$
- Increment/decrement: $++$, $--$
- Bitwise: $|$, $\&$, $>>$, $<<$, \wedge , \sim
- Assignment: $=$, $+=$, $-=$, $*=$, $/=$, $\%=$, $<<=$, $>>=$, $\&=$, $\wedge=$, $|=$

Bit operators example

```
int bitcount(unsigned x) {
    int b;

    for (b=0; x != 0; x >>= 1)
        if (x & 01)
            b++;

    return (b);
}
```

what does bitcount(32) return?

bitcount(15)?

bitcount(34)?

Conditional expressions

```
if (a > b)
```

```
    z = a;
```

```
else
```

```
    z = b;
```

```
z = (a > b) ? a : b;
```

expression₁ ? expression₂ : expression₃

expression₁ evaluated first, then

if true expression₂ is evaluated

if false expression₃ is evaluated

If else

```
if (a > b)
    max = a;
else
    max = b;
```

```
if (expression)
    statement1
else
    statement2
```

If expression is true (non-zero) statement₁ is executed
Otherwise statement₂ is executed

Nested if-else: when things go wrong

```
if ( n >=0)
    printf (...);
    some_function(...);
    if(n % 2 == 0) {
        ....
    }
else /* you mean n is
      negative*/
```

*why does this
ambiguity arise?*

what is a parse?


OOPS! The compiler will associate the else with
the closest if (n%2)

Use braces to fix it!

Nested if-else

- To avoid ambiguity the else is associated with the closest else-less if

```
if (n > 0)
  if (a > b)
    z = a;
  else
    z = b;
```



Always safer to use braces if you're not sure!

Else-if

```
if (expression1)
    statement1
else if (expression2)
    statement2
else if (expression3)
    statement3
else if (expression4)
    statement4
else
    statement5
```

Precedence and associativity

Operators	Associativity
<code>() [] -> .</code>	Left to right
<code> ~ ++ -- + - * & (type) sizeof</code>	Right to left
<code>* / %</code>	Left to right
<code>+ -</code>	Left to right
<code><< >></code>	Left to right
<code>< <= > >=</code>	Left to right
<code>== !=</code>	Left to right
<code>&</code>	Left to right
<code>^</code>	Left to right
<code> </code>	Left to right
<code>&&</code>	Left to right
<code> </code>	Left to right
<code>?:</code>	Right to left
<code>= += -= *= /= %= &= ^= = <<= >>=</code>	Right to left
<code>,</code>	Left to right

Unpredictable results

- `s[i] = i++;`
- **Is the subscript the old one or the new one?**
- It is unspecified so different compilers treat this issue differently

$$x = f() + g()$$

It is not specified which is evaluated first -- f() or g()

Order of evaluation

- Referential transparency revisited
- When does this matter?
- Is this a good or bad design feature?
 - Implications for implementations

Example else-if

```
int binsearch (int x, int v[], int n) {
    int low, high, mid;

    low = 0;
    high = n-1;
    while (low <= high) {
        mid = (low+high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

Switch

```
switch (expression) {  
    case const_expr: statements  
    case const_expr: statements  
    case const_expr: statements  
    default: statements  
}
```

Statements often ends with a **break** statement which causes the exit from **switch**.

Example switch

```
switch (day) {  
    case MONDAY:  
    case WEDNESDAY:  
        prepare_class();  
        break;  
    default:  
        other_stuff();  
        break;  
}
```

How do we generate the symbols MONDAY and WEDNESDAY?

Break and continue

- **break**: provides early exit from a **for**, **while**, **do**, and **switch**
- **continue**: continues the next iteration for a **for**, **while** or **do** to begin

```
for (i=0; i<n; i++) {  
    if(a[i]<0)  
        continue /* skips negative elements*/  
    ...  
}
```

Go to

- `goto label`: makes the program jump to the `label`

```
for ()
    for ()
        ...
        if(failure)
            goto error;
error:
    clean up the mess
```

Goto Considered Harmful

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce.

...

our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Edsger W. Dijkstra

1968

Readings for this lecture

K&R Chapter 2 and 3

