Decision-Making

2025 Winter APS105: Computer Fundamentals
Jon Eyolfson

Lecture 7
1.0.0

We Can Compare Characters

Recall: characters are encoded using ASCII

Encoded means converted into bytes

```
'0' < '1' < '2' < '3' < '4' < '5' < '6' < '7' < '8' < '9'
<
'A' < 'B' < 'C' < 'D' < 'E' < ... < 'W' < 'X' < 'Y' < 'Z'
<
'a' < 'b' < 'c' < 'd' < 'e' < ... 'w' < 'x' < 'y' < 'z'
```

We Can Use Arithmetic with Characters

The characters `'0'` through `'9'` are sequential, the values increase by 1

Examples:

$'0' + 2 \rightarrow '2'$

$'0' + 5 \rightarrow '5'$

We Can Use Arithmetic with Characters

The characters `'0'` through `'9'` are sequential, the values increase by 1

Examples:

```
'0' + 2 → '2'
'0' + 5 → '5'
```

The characters `'A'` through `'Z'` are sequential as well as `'a'` through `'z'`

A upper case character + `32` results in the lower case of that character

Examples:

```
'A' + 2 → 'C'
'a' + 3 → 'd'
'o' - 1 → 'n'
```

Let’s Write a Program to That Looks for a Letter

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

int main(void) {
    printf("Enter a character: ");
    char c = '\0';
    scanf("%c", &c);
    if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
        printf("You entered a letter!\n");
    }
    else {
        printf("You did not enter a letter!\n");
    }
    return EXIT_SUCCESS;
}
```

We Could Create Variables to Make Our Code More Readable

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    printf("Enter a character: ");
    char c = '\0';
    scanf("%c", &c);
    bool isUppercaseLetter = c >= 'A' && c <= 'Z';
    bool isLowercaseLetter = c >= 'a' && c <= 'z';
    if (isUppercaseLetter || isLowercaseLetter) {
        printf("You entered a letter!\n");
    }
    else {
        printf("You did not enter a letter!\n");
    }
    return EXIT_SUCCESS;
}
```

The Compiler Optimizes Logic Operators, Like "Or"

You may write: (complex condition 1) || (complex condition 2)

In the case (complex condition 1) evaluates to `true`,
the compiler will not evaluate (complex condition 2)
    Evaluate is computing the result of an expression

The Compiler Optimizes Logic Operators, Like “Or”

You may write: (complex condition 1) || (complex condition 2)

In the case (complex condition 1) evaluates to `true`,
the compiler will not evaluate (complex condition 2)
- Evaluate is computing the result of an expression

Since the left-hand side of the || operator is `true`, the final result must `true`
- The value of the right-hand side does not matter

The Compiler Also Optimizes the “And” Logic Operator

We can do a similar optimization for the `&&` operator

The Compiler Also Optimizes the “And” Logic Operator

We can do a similar optimization for the `&&` operator

You may write: (complex condition 1) `&&` (complex condition 2)

In the case (complex condition 1) evaluates to `false`,
the compiler will not evaluate (complex condition 2)
 The compiler calls this lazy evaluation

The Compiler Also Optimizes the “And” Logic Operator

We can do a similar optimization for the `&&` operator

You may write: (complex condition 1) `&&` (complex condition 2)

In the case (complex condition 1) evaluates to `false`,
the compiler will not evaluate (complex condition 2)
  The compiler calls this lazy evaluation

Since the left-hand side of the `&&` operator is `false`, the final result must `false`
  The value of the right-hand side does not matter

We Can Re-Write Logic Statements Using De Morgan’s Laws

The laws state that:

```
!(A || B) == !A && !B
!(A && B) == !A || !B
```

We Can Re-Write Logic Statements Using De Morgan's Laws

The laws state that:

```
!(A || B) == !A && !B
!(A && B) == !A || !B
```

If I wanted to only check for a character being not a letter, I might use:

```
(!(isUppercaseLetter || isLowercaseLetter))
```

We Can Re-Write Logic Statements Using De Morgan's Laws

The laws state that:

```
!(A || B) == !A && !B
!(A && B) == !A || !B
```

If I wanted to only check for a character being not a letter, I might use:

```
(!(isUppercaseLetter || isLowercaseLetter))
```

I could re-write this as:

```
(!isUppercaseLetter && !isLowercaseLetter)
```

Beware: Ensure You Use Brackets to Get What You Mean

What happens if I removed the brackets from:

```
(!(isUppercaseLetter || isLowercaseLetter))
```

So, I wrote this instead:

```
(!isUppercaseLetter || isLowercaseLetter)
```

Are these two expressions equivalent?

Beware: Ensure You Use Brackets to Get What You Mean

What happens if I removed the brackets from:

```
(!(isUppercaseLetter || isLowercaseLetter))
```

So, I wrote this instead:

```
(!isUppercaseLetter || isLowercaseLetter)
```

Are these two expressions equivalent?

No, the second is the same as:

```
((!isUppercaseLetter) || isLowercaseLetter)
```

Remember, unary operators have higher precedence!

Beware: ; is a Statement

You may write something like:

```
if (isUppercaseLetter || isLowercaseLetter); {
    printf("You entered a letter!\n");
}
```

When you run this, no matter what, it always prints you entered a letter

Beware: ; is a Statement

You may write something like:

```
if (isUppercaseLetter || isLowercaseLetter); {
    printf("You entered a letter!\n");
}
```

When you run this, no matter what, it always prints you entered a letter

This is because ; by itself is an empty statement that does nothing
  When the condition is true, it does nothing

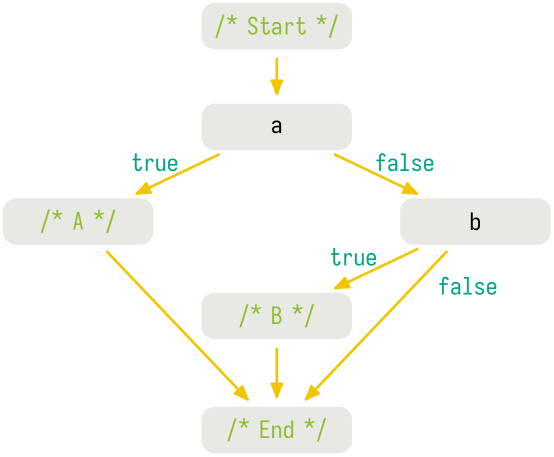
We either do nothing then run `printf`, or jump to `printf`

We Can Chain If Statements Together

You can write:

```
/* Start */
if (a) {
    /* A */
    /* This only runs if a is true. */
}
else if (b) {
    /* B */
    /* This only runs if a is false and b is true. */
}
/* End */
```

The Flow of the Previous Program



We Can Write Nested If Statements

```
if (a) {
    if (b) {
        /* Statements */
    }
}
```

We can put an `if` statement inside an `if` statement

- Each time we begin an if, we add another level of indentation

What Should Try to Be as Concise as Possible

Instead of writing:

```
if (a) {
    if (b) {
        /* Statements */
    }
}
```

We should write:

```
if (a && b) {
    /* Statements */
}
```

In general, the fewer levels of indentation you have, the easier it is to read

Let’s Write a Program to Find the Maximum of 3 Integers

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

int main(void) {
    printf("Enter 3 integers: ");
    int x = 0, y = 0, z = 0;
    scanf("%d%d%d", &x, &y, &z);
    /* TODO */
    int max;
    printf("Maximum: %d\n", max);
    return EXIT_SUCCESS;
}
```

I'll Only Write the Code After the `scanf` (to Save Space)

```c
int main(void) {
    int max;
    if (x >= y) {
        if (x >= z) { max = x; }
        else        { max = z; }
    }
    else if (y >= x) {
        if (y >= z) { max = y; }
        else        { max = z; }
    }
    else {
        max = z;
    }
    printf("Maximum: %d\n", max);
    return EXIT_SUCCESS;
}
```

Can We Get Rid of the Nested Ifs?

The structure looks similar to:

```
if (a) {
    if (b) {
        /* Statements */
    }
}
```

Except there's an `else`, however all the `else` statements are the same

We Can Get Rid of the Nested Ifs

```c
int main(void) {
    int max;
    if (x >= y && x >= z) {
        max = x;
    }
    else if (y >= x && y >= z) {
        max = y;
    }
    else {
        max = z;
    }
    printf("Maximum: %d\n", max);
    return EXIT_SUCCESS;
}
```

In Fact, We Can Get Rid of the `else`

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

int main(void) {
    printf("Enter 3 integers: ");
    int x = 0, y = 0, z = 0;
    scanf("%d%d%d", &x, &y, &z);
    int max = z;
    if (x >= y && x >= z) {
        max = x;
    }
    else if (y >= x && y >= z) {
        max = y;
    }
    printf("Maximum: %d\n", max);
    return EXIT_SUCCESS;
}
```