CSE 220: Systems Programming

Conditionals and Control Flow

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Conditionals in C

We have previously discussed true and false in C:

- `0` is false
- anything else is true

However, boolean expressions and `true` and `false` are less unpredictable:

- `true` and true results are exactly 1
- `false` and false results are exactly `0`

Control Flow

We have discussed only the for loop in C.

Required readings in K&R have covered other control flow.

We will look at `if`, `switch`, and their implementations.

There are other control flow statements (discussed in K&R), but they behave similarly to one of these.

Boolean Operators

C uses the following Boolean operators:

- `!`: Logical not; inverts the following expression
- `&&`: Logical and; true iff the LHS and RHS are both true
- `||`: Logical or; true if either the RHS or LHS is true

Do not confuse these with the similarly-named bitwise operators!
(We will discuss those later.)

Boolean Logic in C

C uses a short circuit evaluation for Boolean logic.

This means that evaluation of a Boolean sentence stops as soon as its final truth value is known.

For example:

```
x && y
```

If x is false, then this sentence is false.

In that case, y will never be evaluated.

Short Circuit Consequences

The consequences of short-circuit evaluation can be surprising.

If terms in the sentence have side effects, those side effects may not run.

This can be very useful, but also surprising!

```
if (i < len && array[i] == SOMEVAL) {
    /* Useful!  If array[i] is past the end of the
       array, the illegal access never happens. */
}
```

Equality Operators

There are two equality operators:

- `==`: Compares value equality, returns true if equal
- `!=`: Compares value equality, returns false if equal

Note that these operators compare values, not logical truth!

In particular, note that many values are "true", but `true` is 1!

This means that two logically true values may compare unequal.

Truthiness

```
bool x = true;
int y = 2;

if (x)
    printf("x is true\n");
if (y)
    printf("y is true\n");
if (x == y)
    printf("x and y are equal\n");
```

Truthiness

```
bool x = true;
int y = 2;

if (x)
    printf("x is true\n");
if (y)
    printf("y is true\n");
if (x == y)
    printf("x and y are equal\n");
```

Output:

```
x is true
y is true
```

stdbool

The header `#include <stdbool.h>` defines some useful things.

- The type `bool`, which holds only 0 or 1
- The values `true` and `false`

Before C99, these things didn't exist in the standard, but were widely defined in programs.

Therefore they were standardized to require a header.

```
bool b = 2;
printf("%d\n", b);
```

Output:

```
1
```

Control Flow

Control flow is the path that execution takes through a program.

The C model is linear flow by default.

Control flow statements can change the order of execution.

This is how our programs make decisions.

We will examine how this flow is achieved.

The `if` Statement

The simplest control statement in C is `if`.

Its syntax is:

```
if (condition) {
    body;
}
```

If the expression `condition` evaluates to any true value, `body` runs.

Otherwise, `body` is skipped.

Implementing `if`

The `if` statement must be compiled to machine instructions.

Those machine instructions must encode the condition check and jump.

This is normally implemented as a conditional branch instruction.

You don't have to learn assembly for this course, but we will look at some machine instruction concepts.

A Simple Condition — C

```
int main(int argc, char *argv[])
{
    if (argc == 2 && argv[1][0] == '-') {
        puts("negative");
    }
    return 0;
}
```

A Simple Condition — Assembly

```
      cmpl $2, %edi             ; compare argc to 2
      je   .L8                  ; jump to .L8 if ==
.L4:
      xorl %eax, %eax           ; set up return value
      ret                       ; return 0
.L8:
      movq 8(%rsi), %rax        ; load argv[2][0] into %rax
      cmpb $45, (%rax)          ; compare %rax to 45 ('-')
      jne  .L4                  ; jump to .L4 if !=
      leaq .LC0(%rip), %rdi     ; load "negative" into %rdi
      subq $8, %rsp             ; make room on stack
      call puts@PLT             ; call puts("negative")
                                ; another return 0 goes here
```

Conditional Instruction Flow

Note that the structure of the program was lost.

One of the advantages of high-level languages is structure.

The computer can generally only:

- Make simple comparisons (sometimes only to zero!)
- Jump to a program location

Anything more complicated is a software construction.

The `else` Clause

The `else` clause is simply either:

- The next instruction after a jump
- The jump destination (with the `if` body being the next instruction)

Which layout the compiler uses depends on the code and architecture.

else Gotchas

I have previously advocated always using blocks.

Here is a place where it really matters:

```
if (modify_x)
    if (negate)
        x = x * -1;
else
    y = -x;
```

else Gotchas

I have previously advocated always using blocks.

What this actually means is:

```
if (modify_x)
    if (negate)
        x = x * -1;
    else
        y = -x;
```

else Gotchas

I have previously advocated always using blocks.

What you should use is:

```
if (modify_x) {
    if (negate) {
        x = x * -1;
    }
} else {
        y = -x;
}
```

Understanding `else if`

Unlike some languages, C does not have an else if statement.

Instead, it uses `else if`.

This is because `if` is a statement that forms the `else` body.

Therefore, `else if (...)` is actually `else { if (...)}`!

Languages using `elif`, `Elsie`, *etc.*often have syntax reasons. Consider Python:

- `else if` is missing `:`
- `else: if` has invalid indentation.

The switch Statement

C provides a convenient multi-case condition statement: switch.

It compares an integer with a set of values.

The first matching integer value begins execution.

```
switch (integer) {
case value1:
    body_for_value1;
    break;
case value2:
    body_for_value2;
    break;
default:
    else_body;
}
```

`switch` Gotchas

The `break` keyword is never implied.

switch Gotchas

The break keyword is never implied.

```
int i = 0, value = 1;
switch (value) {
case 1:
    i++;
case 2:
    i++;
default:
    i++;
}
printf("%d\n", i);
```

switch Gotchas

The break keyword is never implied.

```
int i = 0, value = 1;
switch (value) {
case 1:
    i++;
case 2:
    i++;
default:
    i++;
}
printf("%d\n", i);
```

Output:

```
3
```

switch Machine Instructions

```
        cmpl $1, %eax        ; Is i 1?
        je   .L3             ; Jump to L3 if so
        cmpl $2, %eax        ; Is i 2?
        je   .L4             ; Jump to L4 if so
        jmp  .L2             ; Jump to L2
.L3:
        addl $1, -4(%rbp)    ; Add 1 to i
.L4:
        addl $1, -4(%rbp)    ; Add 1 to i
.L2:
        addl $1, -4(%rbp)    ; Add 1 to i
```

Looping Control Flow

Consider looping using machine instructions.

Looping Control Flow

Consider looping using machine instructions.

1. Evaluate the loop condition
2. Jump after the loop body if not met
3. Execute the loop body
4. Jump to the loop condition

Summary

- All nonzero values are true conditions in C.
- All Boolean expressions use 1 for true.
- The `bool` keyword holds only 0 or 1.
- C uses short-circuit evaluation of Boolean logic.
- `if` and `switch` implement conditionals.
- Use blocks for if and else!
- Control flow is implemented with comparisons and jumps.

Next Time …

- Addresses and Pointers

References I

Optional Readings

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Chapter 3. Prentice Hall, 1988.

License