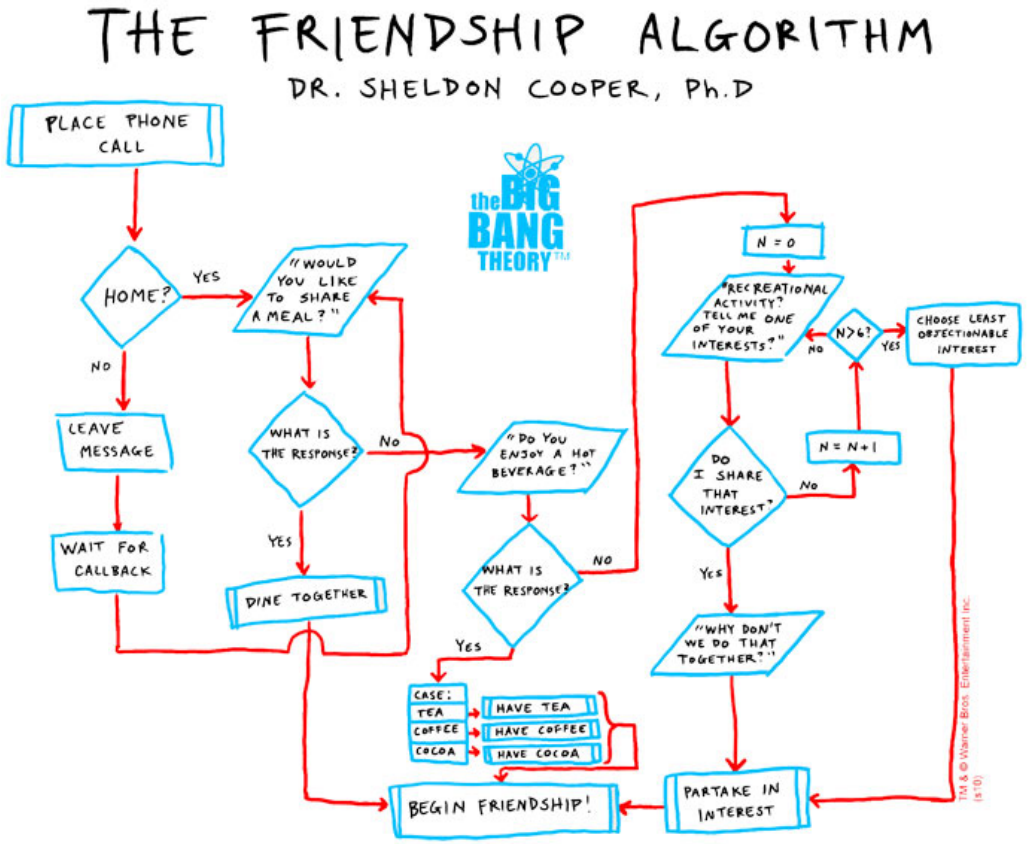
C introduction

Control structures

Contents

Back in control

Even though C is a sequential programming language, the program flow can branch. Use conditions to determine the behaviour of your program.



Let's calc

Each expression (variables or values) you type in gets evaluated.
You can use operators to combine existing expressions to new ones.

- +, -, *, / as all of you should know
- % is the modulo (remainder) operator
- *, /, % get evaluated before +, -
- Operations in ( ) are of higher precedence

```
1 int a = 5;        /* value 5 */
2 int b = a + 3;    /* value 8 */
3 char c = 120;     /* 120 */
4 char d = 2 * c;   /* why is that not 240? */
```

You see, it's not that easy

- Variables may overflow
- You shall not divide by zero
- Integer division differs from floating point division
- You can use operators between different data types
  - mixing different sizes
  - mixing integer and floating point variables

```
int i1 = 42, i2 = 23;
short s = 13;
float f = 3.14;

i1 / i2;     /* results in 1, not a real division */
i1 + s;      /* int and short, result is int */
i1 / f;      /* result is float, actual division */
```

Syntactic sugar

As you have seen, you can use any expression on the right side of the assignment operator.
This expression often contains the variable it is assigned to.

To avoid redundancy, C offers the following short forms:

```
a += 4;      /* a = a + 4; */
a -= 4;      /* a = a - 4; */
a *= b;      /* a = a * b; */
b /= 42;     /* b = b / 42; */
b %= 2;      /* b = b % 2; */
c++;         /* c = c + 1; */
++c;         /* c = c + 1; */
c--;         /* c = c - 1; */
--c;         /* c = c - 1; */
```

The truth about expressions

Expressions can also be evaluated to truth values.
If a value or a variable equals 0, its corresponding truth value is *false*.
Otherwise it's *true*.
The representations of *false* and *true* are 0 and 1.
An expression containing relational operators gets evaluated to such a truth value.

Relational operators:

- $<$, $>$, $<=$, $>=$
- $==$ for "equal to"
- $!=$ for "not equal to"

Do not get confused

Imagine the following

```
(5 < 7) == 1;    /* evaluated to 1 */
```

Why?

Do not get confused

Imagine the following

```
(5 < 7) == 1;    /* evaluated to 1 */
```

Why?

- $(5 < 7)$ is true $\rightarrow 1$

Do not get confused

Imagine the following

```
(5 < 7) == 1;    /* evaluated to 1 */
```

Why?

- $(5 < 7)$ is true $\rightarrow 1$
- $1 == 1$ is true $\rightarrow 1$

A sign meant...

Assignments are expressions that get evaluated and have a truth value, too. Consider:

```
c = 0;                  /* 0 -> false */
c = 2 * 5;              /* 2 * 5 = 10,   c = 10, true */
c = (0 < 1);            /* 0 < 1 = true, c = 1,  true */

a = (b == (c = d)); /* Wat? */
```

A sign meant...

Assignments are expressions that get evaluated and have a truth value, too. Consider:

```
c = 0;                  /* 0 -> false */
c = 2 * 5;              /* 2 * 5 = 10,   c = 10, true */
c = (0 < 1);            /* 0 < 1 = true, c = 1,  true */

a = (b == (c = d)); /* Wat? */
```

c++ expressions are evaluated before the increment while *++c* increments first (the same applies on *c−−* and *−−c*):

```
int c = 42;
int a = c++;    /* evaluates to c,     a is 42, c is 43 */
int b = ++c;    /* evaluates to c + 1, b is 44, c is 44 */
```

Boolean arithmetic

Truth values can be connected by boolean operators resulting in a new truth value.

- && for AND (results in 1 if both operands are true, else 0)
- || for OR (results in 1 if at least one operator is true, else 0)
- ! for NOT (results in 1 if the operand is false, else 0)

Precedence order:

$$! \; > \; \&\& \; > \; ||$$

Seems logical

Caution: a condition will only be evaluated until its result is definitive.

- *a* && *b*: *b* will only be evaluated if *a* is true.
- *a* || *b*: *b* will only be evaluated if *a* is false.

```
/* safely check if a is divisible by b */
if ((b != 0) && (a % b == 0)) {
    ...
}
```

Seems logical

Caution: a condition will only be evaluated until its result is definitive.

- *a* && *b*: *b* will only be evaluated if *a* is true.
- *a* || *b*: *b* will only be evaluated if *a* is false.

```
/* safely check if a is divisible by b */
if ((b != 0) && (a % b == 0)) {
    ...
}
```

- How do you get NAND, NOR and XOR?

Seems logical

Caution: a condition will only be evaluated until its result is definitive.

- *a* && *b*: *b* will only be evaluated if *a* is true.
- *a* || *b*: *b* will only be evaluated if *a* is false.

```
/* safely check if a is divisible by b */
if ((b != 0) && (a % b == 0)) {
    ...
}
```

- How do you get NAND, NOR and XOR?

```
!(a && b);                    /* NAND */
!(a || b);                    /* NOR */
(a && !b) || (!a && b);       /* XOR */
```

if...else

To make decisions during run time, you can use the truth value of an expression:

```
if (condition)
    statement1;
else
    statement2;
```

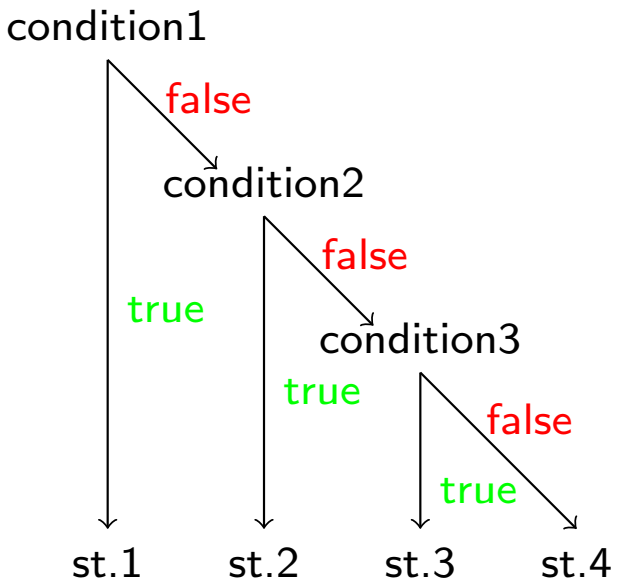
Now statement1 is only executed if the truth value of condition is *true*. Otherwise statement2 is executed. The *else* part is optional.

For multiple statements in the *if* or *else body*, use braces:

```
if (condition) {
    statement1;
    statement2;
}
```

else if

To differentiate between more than two cases, you can use the if condition as a statement in the else body:




```
if (condition1)
    statement1;
else if (condition2)
    statement2;
else if (condition3)
    statement3;
else
    statement4;
```

switch

If you have to check one variable for many constant values, *switch case* is your friend:

```
switch (variable) {
    case option1: statement1; break;
    case option2: statement2; break;
    case option3: statement3; break;
    default: statement4; break;
}
```

- *case option* defines a jump label
- More than one statement after it possible without braces
- All statements until the next *break;* will be executed

Scopes

You begin a block with a '{' and end it with a '}':

```
1 int i = 4;
2 {
3     i = 3;          /* valid */
4     int j = 5;
5 }
6 j = 2;              /* invalid, j is not in scope */
```

- Program area in which an identifier may be used
- Referring to it anywhere else causes compilation errors
- Starts at the line of declaration
- Ends at the end of the block, in which the variable was declared

Shadowing identifiers

When redeclaring identifiers inside a block, they refer to a new variable:

```
1 int i = 3;          /* Defining i with value 3 */
2 {
3     i = 2;          /* "Outside" i is now 2 */
4     int i = 4;      /* New "inside" i */
5     i = 7;          /* Changes "inside" i only */
6 }
7                     /* "Outside" i is still 2 */
```

Style: When nesting blocks, indent every inner block by one additional tab!

A few words on style

- Typing if (cond) instead of if(cond) helps people to differentiate between control structures and function calls faster
- When starting a new block, you should type) { rather than){
- Do not start a new block for a single statement
- Do not put statements and conditions on the same line

```
if(cond){ statement; }   /* bad style */

if (cond) {              /* looks better, still bad style */
    statement;
}

if (cond)
    statement;           /* looks way clearer */
```

More words on style

- if you use a block anywhere in an if ... else structure, put all blocks of this structure in braces

```
if (cond)          /* bad style, inconsistent */
    statement;
else {
    statement;
    statement;
}

if (cond) {        /* way better style */
    statement;
} else {
    statement;
    statement;
}
```