Simplifying Loops and Logic

MAKING CONTROL FLOW EASY TO READ

- ➔ **Note:** Make all your conditionals, loops, and other changes to control flow as “natural” as possible—written in a way that doesn’t make the reader stop and reread your code.

The Order of Arguments in Conditionals

- Which of these two pieces of code is more readable:

```
if (length >= 10)                while (bytes_received < bytes_expected)

        Or                                         Or

 if (10 <= length)               while (bytes_expected > bytes_received)
```

Reason

Left-hand side

- The expression “being interrogated,” whose value is more in flux.

Right-hand side

- The expression being compared against, whose value is more constant.

C and C++

- if (obj = NULL)----> Bug

→ if (obj == NULL)

→ if (NULL == obj)

→ if (NULL = obj) won’t even compile

The Order of if/else Blocks

```
if (a == b) {

        // Case One ... }

 else {

        // Case Two ... }
```

```
if (a != b) {

        // Case Two ... }

else {

        // Case One ... }
```

The ?: Conditional Expression

- Here’s a case where the ternary operator is readable and compact:

```
time_str += (hour >= 12) ? "pm" : "am";
```

- Avoiding the ternary operator, you might write:

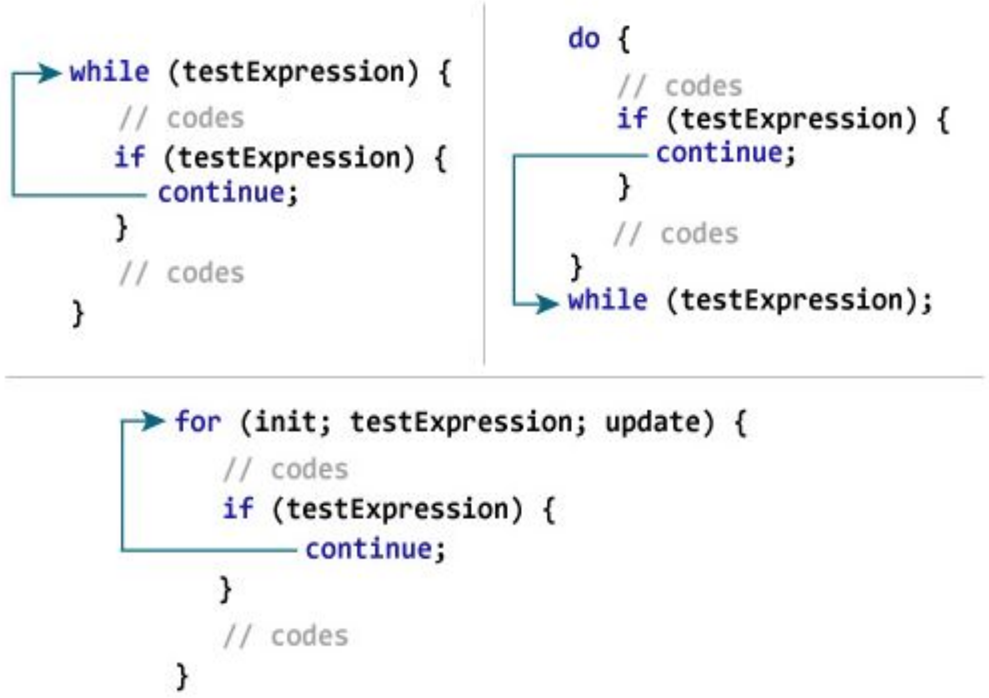
```
if (hour >= 12) {
    time_str += "pm"; }
else {
    time_str += "am"; }
```

- ➔ **Note:** Instead of minimizing the number of lines, a better metric is to minimize the time needed for someone to understand it.
- ➔ **Advice:** By default, use an if/else. The ternary ?: should be used only for the simplest cases.

Avoid do/while Loops

- A do/while loop is that a block of code may or may not be re executed based on a condition underneath it.
- The do-statement is a source of errors and confusion.

```
do {

        continue; }

while (false);
```



```
while (testExpression) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

```
do {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

Returning Early from a Function

➢ Some coders believe that functions should never have multiple return statements, this is nonsense. Returning Early from a function is perfectly fine—and often desirable.

```
public boolean Contains(String str, String substr) {
        if (str == null || substr == null) return false;
        if (substr.equals("")) return true;
                ... }
```

The Infamous goto

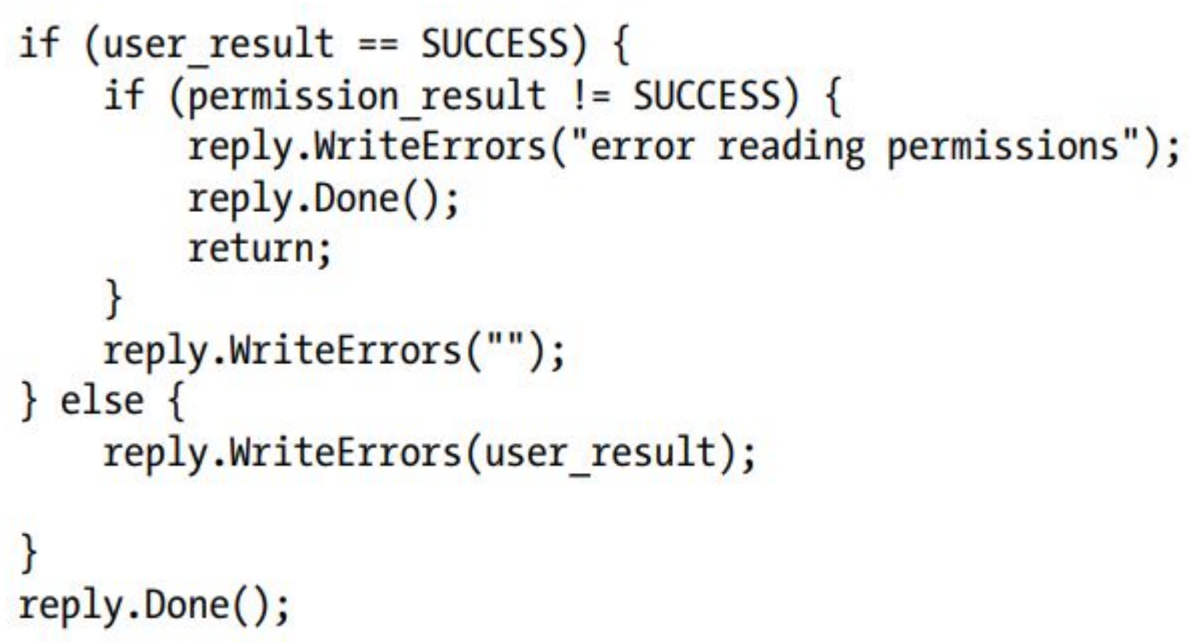
- In languages other than C, there is little need for goto because there are so many better ways to get the job done. Gotos are also notorious for getting out of hand quickly and making code difficult to follow. But you can still see goto used in various C projects—most notably the Linux kernel.

- The simplest, most innocent use of goto is with a single exit at the bottom of a function:

```
if (p == NULL) goto exit;
...
exit:
    fclose(file1);
    fclose(file2);
    ...
    return;
```

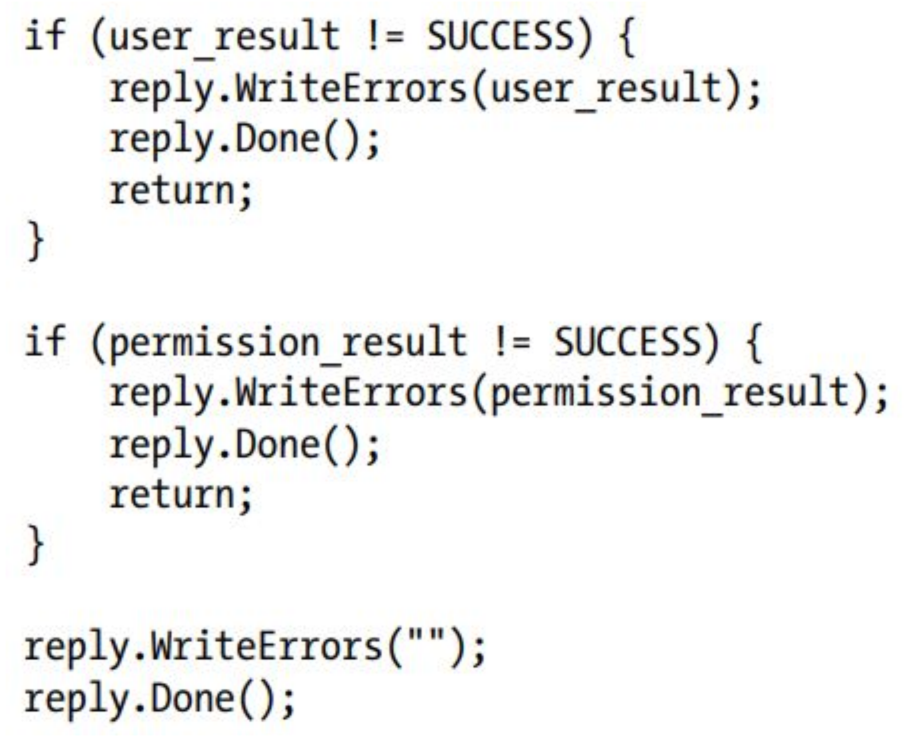
Minimize Nesting

- Deeply nested code is hard to understand.



```
if (user_result == SUCCESS) {
    if (permission_result != SUCCESS) {
        reply.WriteErrors("error reading permissions");
        reply.Done();
        return;
    }
    reply.WriteErrors("");
} else {
    reply.WriteErrors(user_result);

}
reply.Done();
```

- Improve the code of removing nesting by Returning Early.



```
if (user_result != SUCCESS) {
    reply.WriteErrors(user_result);
    reply.Done();
    return;
}

if (permission_result != SUCCESS) {
    reply.WriteErrors(permission_result);
    reply.Done();
    return;
}

reply.WriteErrors("");
reply.Done();
```

BREAKING DOWN GIANT EXPRESSIONS

- Recent research suggests that most of us can only think about three or four “things” at a time. Simply put, the larger an expression of code is the harder to understand.

Explaining Variables

- The simplest way to break down an expression is to introduce an extra variable which called explaining variable that captures a smaller subexpression.
- Here is an example:

```
if line.split(':')[0].strip() == "root":
    ...
```

- Here is the same code, now with an explaining variable:

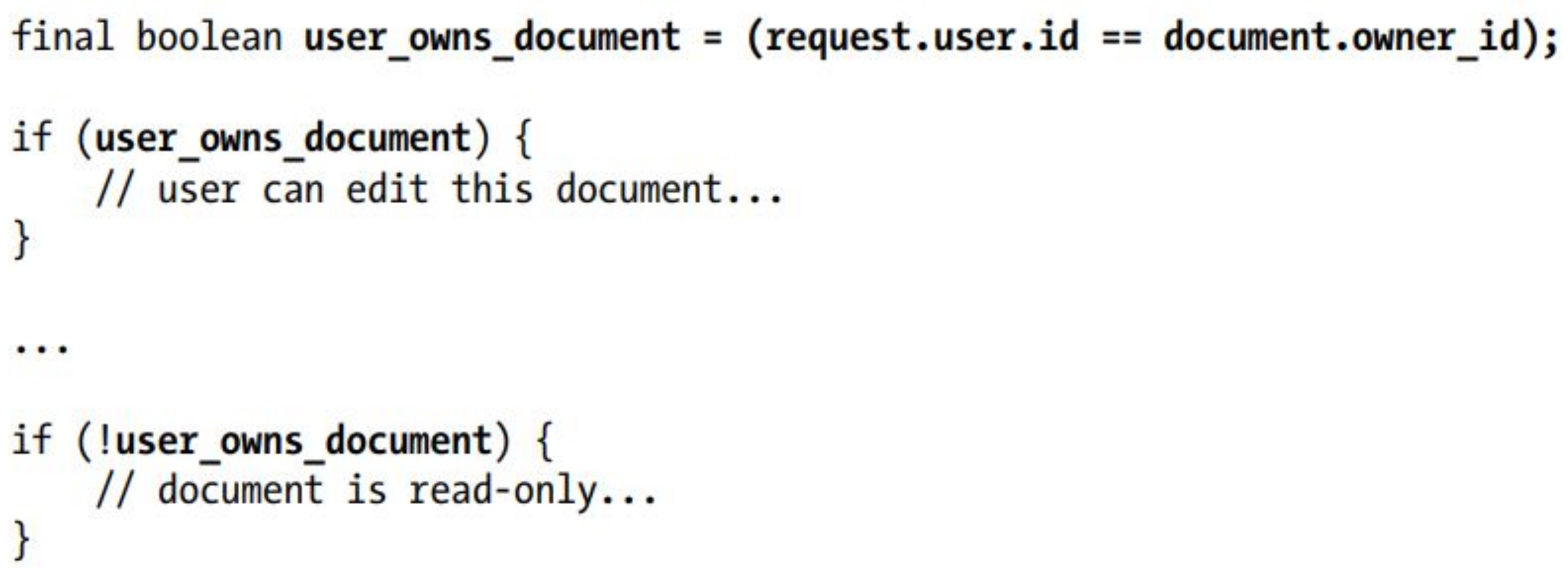
```
username = line.split(':')[0].strip()
if username == "root":
    ...
```

Summary Variables

- Example, consider the expressions in this code:

```
if (request.user.id == document.owner_id) {

        // user can edit this document...

 }

...

if (request.user.id != document.owner_id) {

        // document is read-only…

 }
```

- Adding a summary variable:



```
final boolean user_owns_document = (request.user.id == document.owner_id);

if (user_owns_document) {
    // user can edit this document...
}

...

if (!user_owns_document) {
    // document is read-only...
}
```

Using De Morgan’s Laws

➔ They are two ways to rewrite a boolean expression into an equivalent one:

- ◆ not (a or b or c) ⇔ (not a) and (not b) and (not c)
- ◆ not (a and b and c) ⇔ (not a) or (not b) or (not c)

Example: if (!(file_exists && !is_protected)) Error("Sorry, could not read file.");

➔ It can be rewritten to:

if (!file_exists || is_protected) Error("Sorry, could not read file.");

Abusing Short-Circuit Logic

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

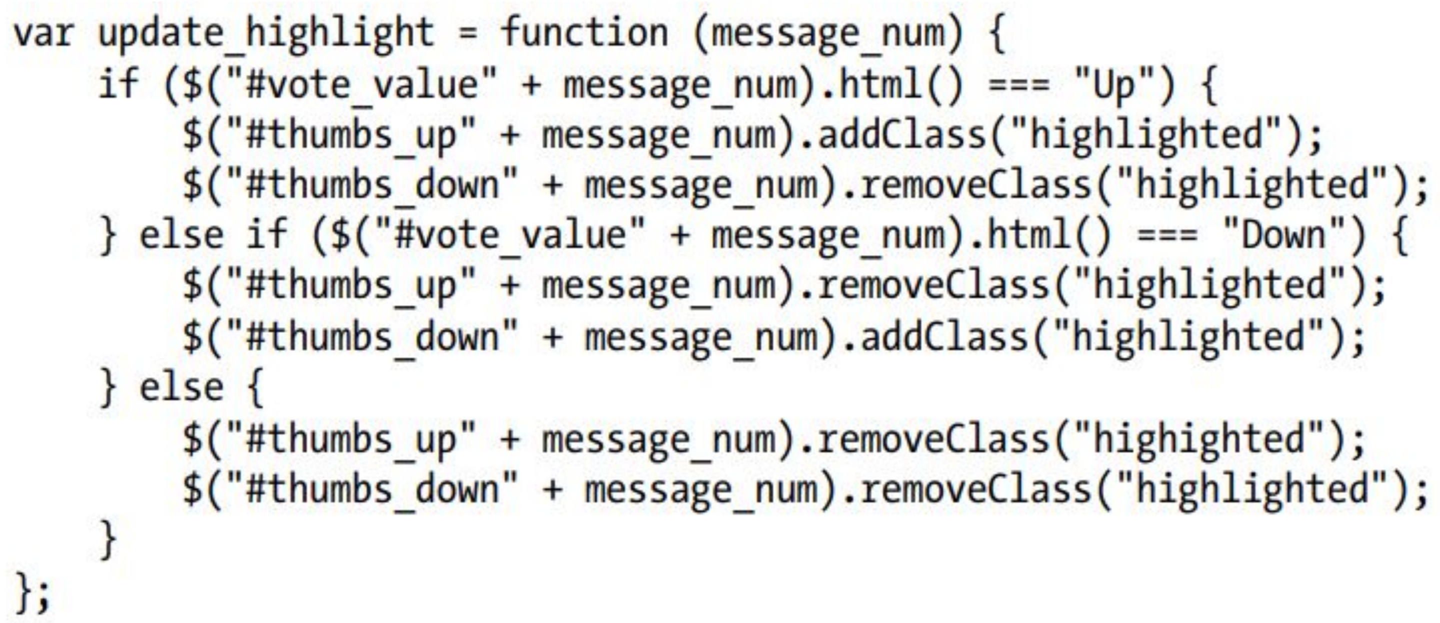
- ➔ In English, what this code is saying is, “Get the bucket for this key. If the bucket is not null, then make sure it isn’t occupied.”

```
bucket = FindBucket(key);

if (bucket != NULL) assert(!bucket->IsOccupied());
```

- ➔ It does exactly the same thing, and even though it’s two lines of code, it’s much easier to understand.

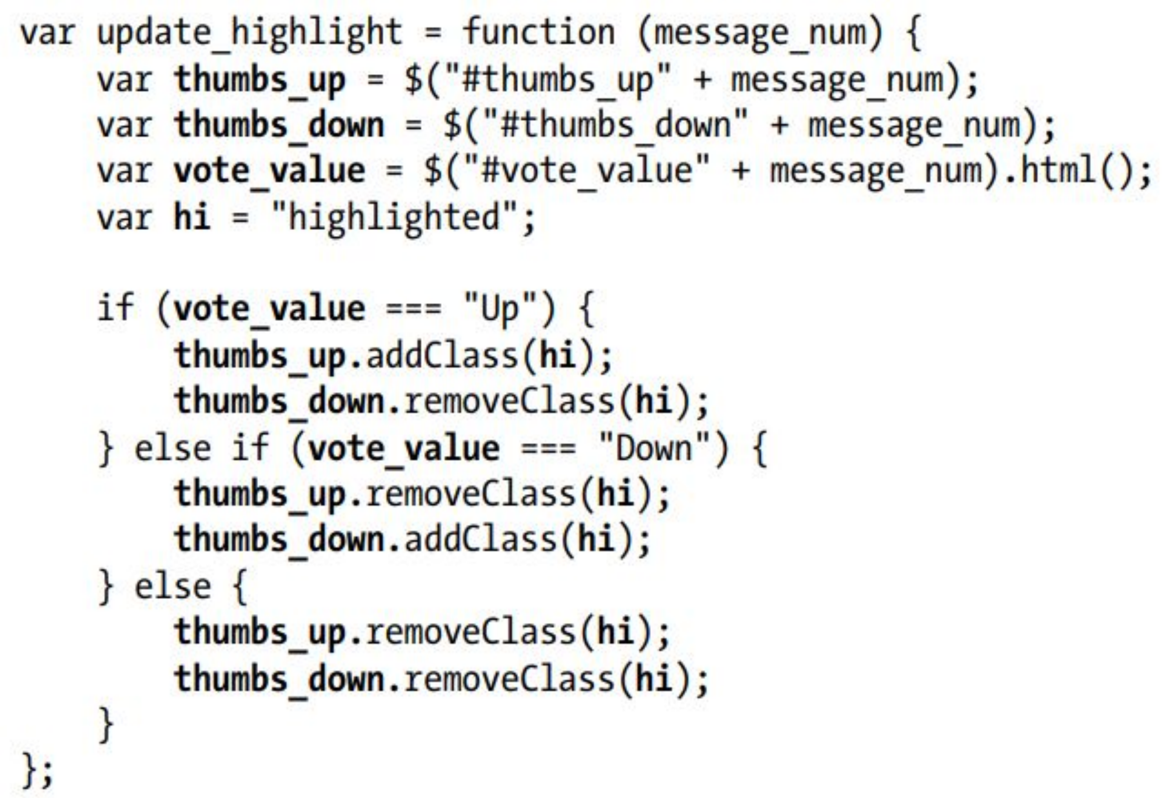
Breaking Down Giant Statements

- This chapter is about breaking down individual expressions, but the same techniques apply to breaking down larger statements as well.



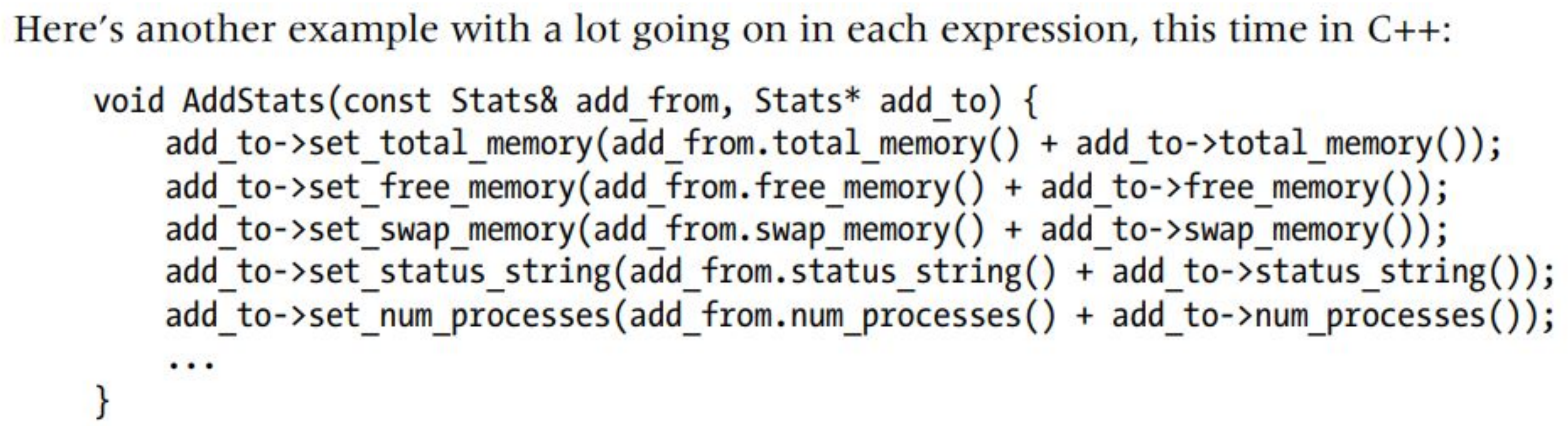
```
var update_highlight = function (message_num) {
    if ($("#vote_value" + message_num).html() === "Up") {
        $("#thumbs_up" + message_num).addClass("highlighted");
        $("#thumbs_down" + message_num).removeClass("highlighted");
    } else if ($("#vote_value" + message_num).html() === "Down") {
        $("#thumbs_up" + message_num).removeClass("highlighted");
        $("#thumbs_down" + message_num).addClass("highlighted");
    } else {
        $("#thumbs_up" + message_num).removeClass("highighted");
        $("#thumbs_down" + message_num).removeClass("highlighted");
    }
};
```

- Fortunately, a lot of the expressions are the same, which means we can extract them out as summary variables at the top of the function:



```
var update_highlight = function (message_num) {
    var thumbs_up = $("#thumbs_up" + message_num);
    var thumbs_down = $("#thumbs_down" + message_num);
    var vote_value = $("#vote_value" + message_num).html();
    var hi = "highlighted";

    if (vote_value === "Up") {
        thumbs_up.addClass(hi);
        thumbs_down.removeClass(hi);
    } else if (vote_value === "Down") {
        thumbs_up.removeClass(hi);
        thumbs_down.addClass(hi);
    } else {
        thumbs_up.removeClass(hi);
        thumbs_down.removeClass(hi);
    }
};
```

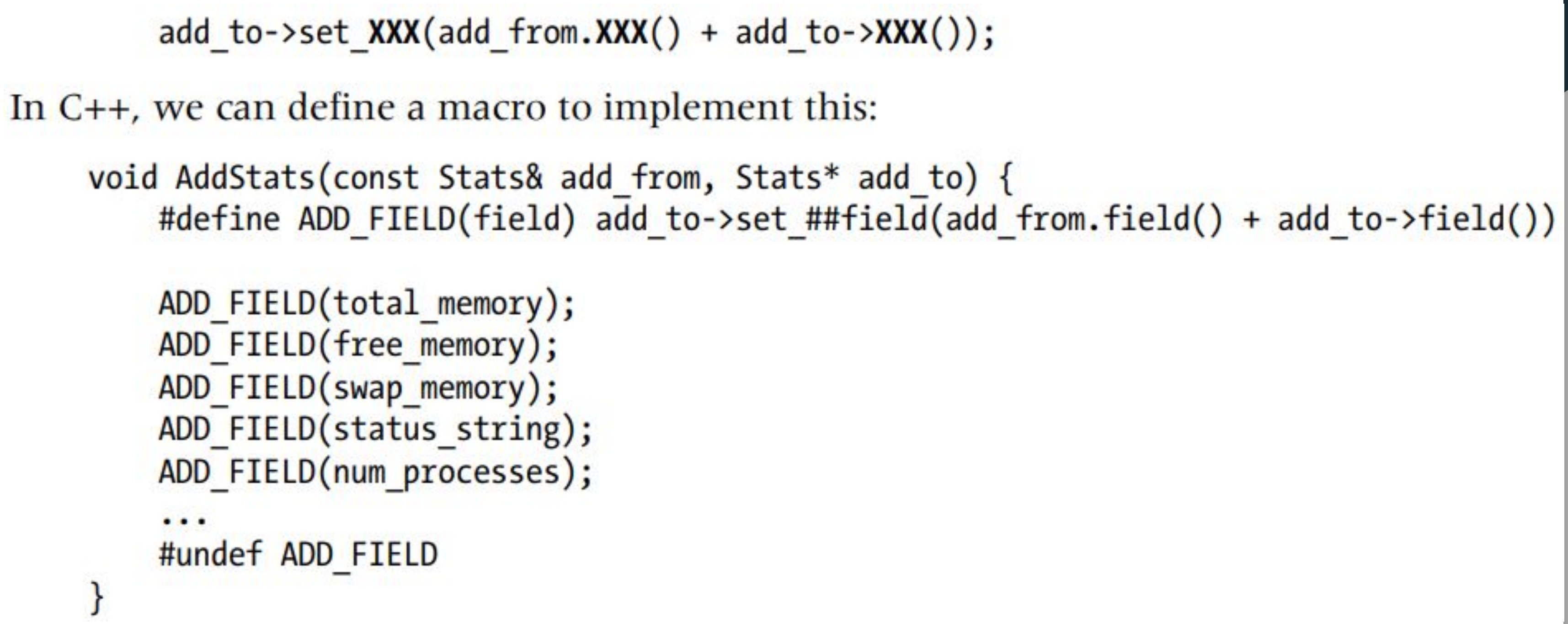
Another Creative Way to Simplify Expressions



Here's another example with a lot going on in each expression, this time in C++:

```
void AddStats(const Stats& add_from, Stats* add_to) {
    add_to->set_total_memory(add_from.total_memory() + add_to->total_memory());
    add_to->set_free_memory(add_from.free_memory() + add_to->free_memory());
    add_to->set_swap_memory(add_from.swap_memory() + add_to->swap_memory());
    add_to->set_status_string(add_from.status_string() + add_to->status_string());
    add_to->set_num_processes(add_from.num_processes() + add_to->num_processes());
    ...
}
```

You might realize that each line is doing the same thing, just to a different field each time:



```
add_to->set_XXX(add_from.XXX() + add_to->XXX());
```

In C++, we can define a macro to implement this:

```
void AddStats(const Stats& add_from, Stats* add_to) {
    #define ADD_FIELD(field) add_to->set_##field(add_from.field() + add_to->field())

    ADD_FIELD(total_memory);
    ADD_FIELD(free_memory);
    ADD_FIELD(swap_memory);
    ADD_FIELD(status_string);
    ADD_FIELD(num_processes);
    ...
    #undef ADD_FIELD
}
```

VARIABLES AND READABILITY

1. The more variables there are, the harder it is to keep track of them all.
2. The bigger a variable’s scope, the longer you have to keep track of it.
3. The more often a variable changes, the harder it is to keep track of its current value.

Eliminating Variables

➔ We’re eliminating variables that don’t improve readability. When a variable like this is removed, the new code is more concise and just as easy to understand.

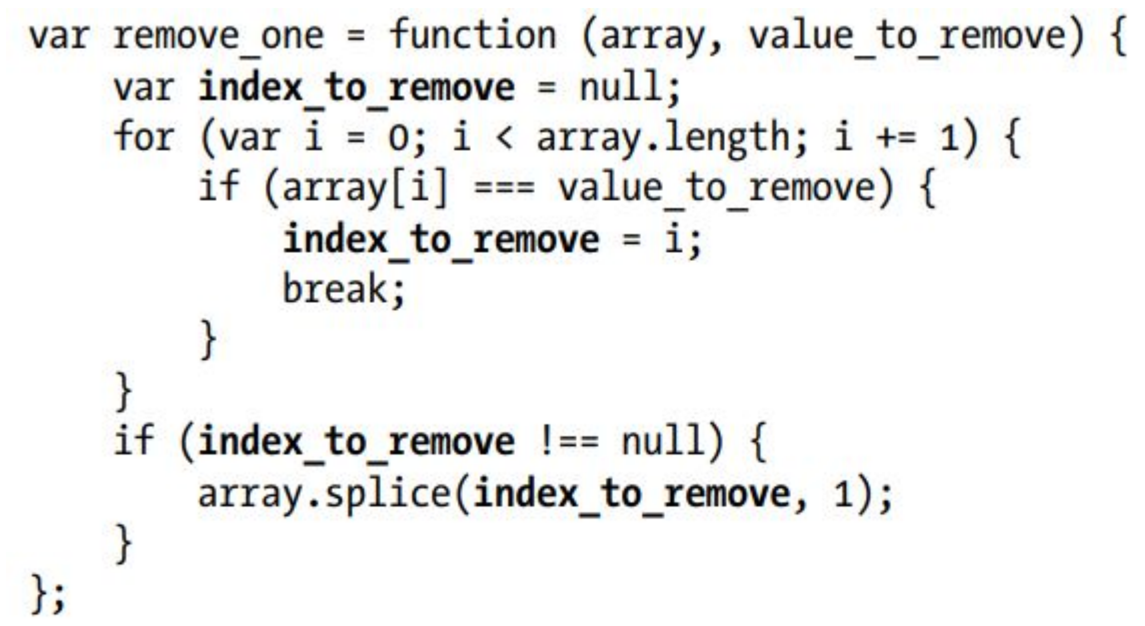
- ➢ **Remove useless Temporary Variables**

```
now = datetime.datetime.now()
root_message.last_view_time = now
```

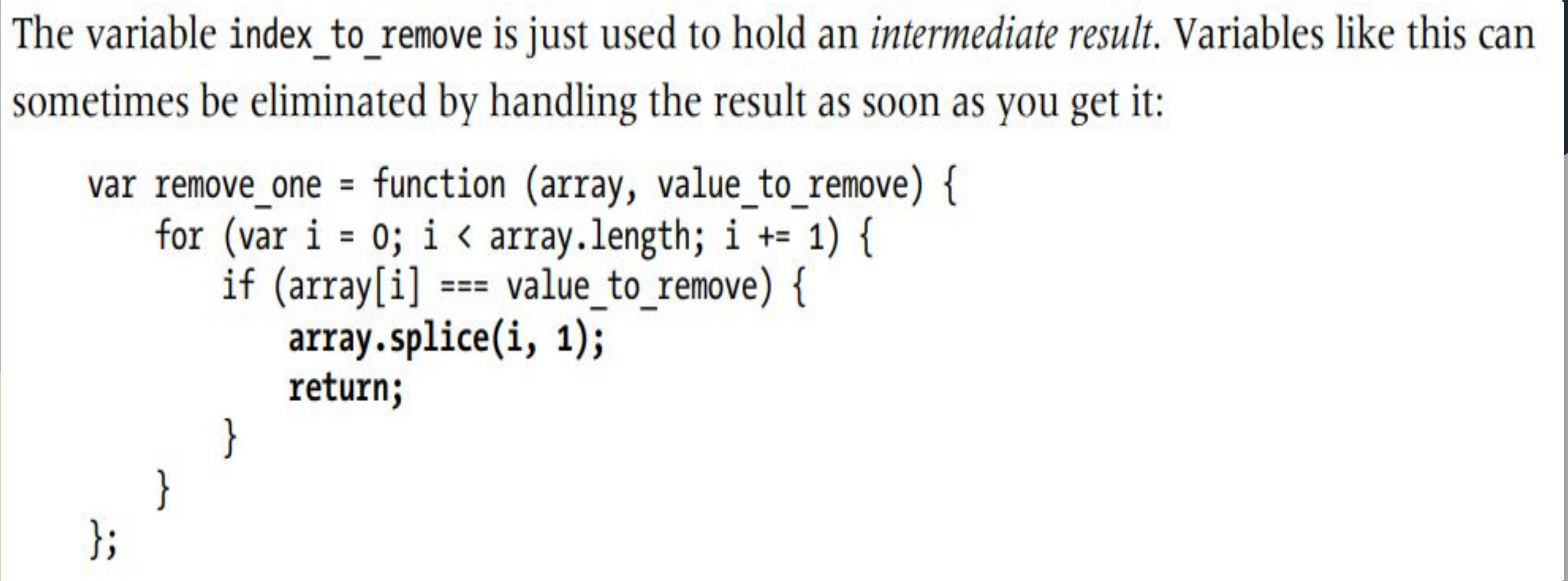
➔ Without now, the code is just as easy to understand:

```
root_message.last_view_time = datetime.datetime.now()
```

➢ Eliminating Intermediate Results



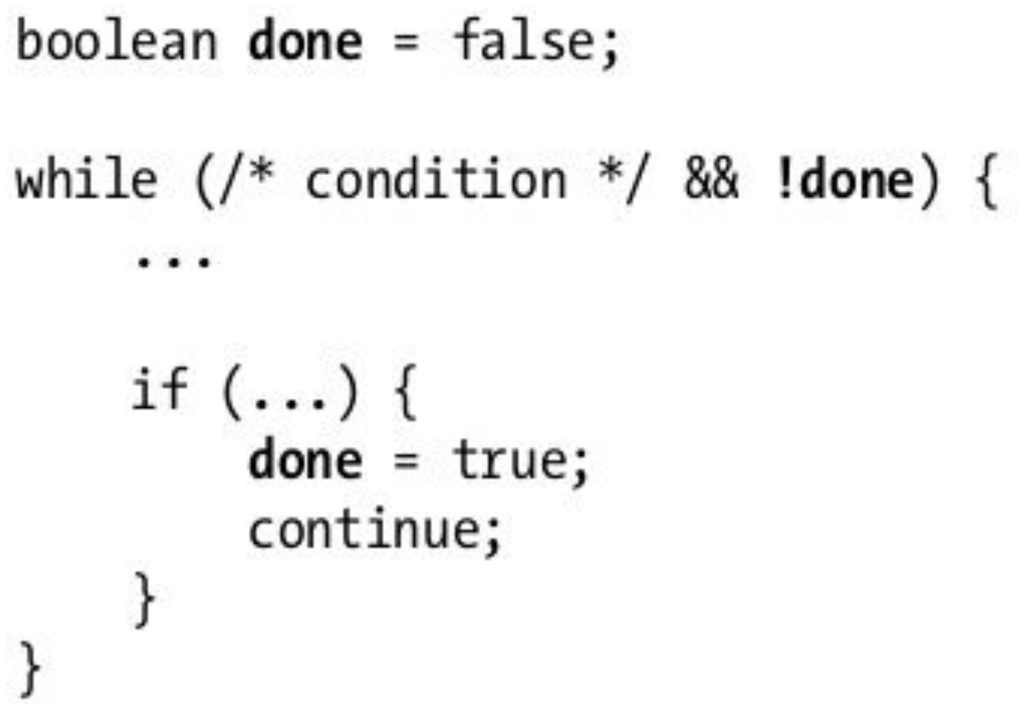
```
var remove_one = function (array, value_to_remove) {
    var index_to_remove = null;
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === value_to_remove) {
            index_to_remove = i;
            break;
        }
    }
    if (index_to_remove !== null) {
        array.splice(index_to_remove, 1);
    }
};
```



The variable `index_to_remove` is just used to hold an *intermediate result*. Variables like this can sometimes be eliminated by handling the result as soon as you get it:

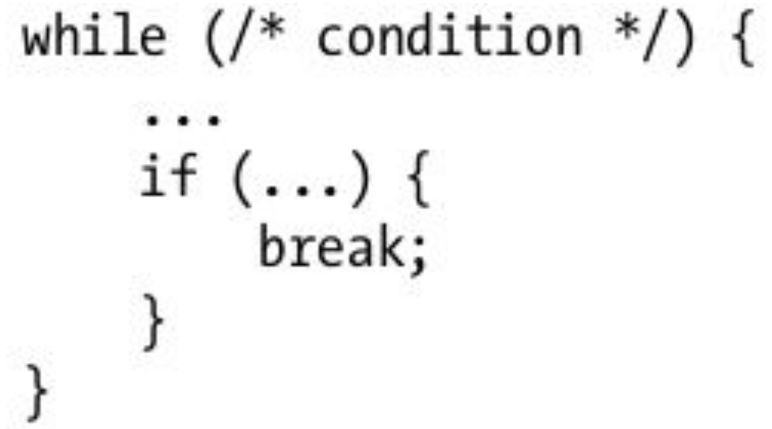
```
var remove_one = function (array, value_to_remove) {
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === value_to_remove) {
            array.splice(i, 1);
            return;
        }
    }
};
```

➢ Eliminating Control Flow Variables



```
boolean done = false;

while (/* condition */ && !done) {
    ...

    if (...) {
        done = true;
        continue;
    }
}
```

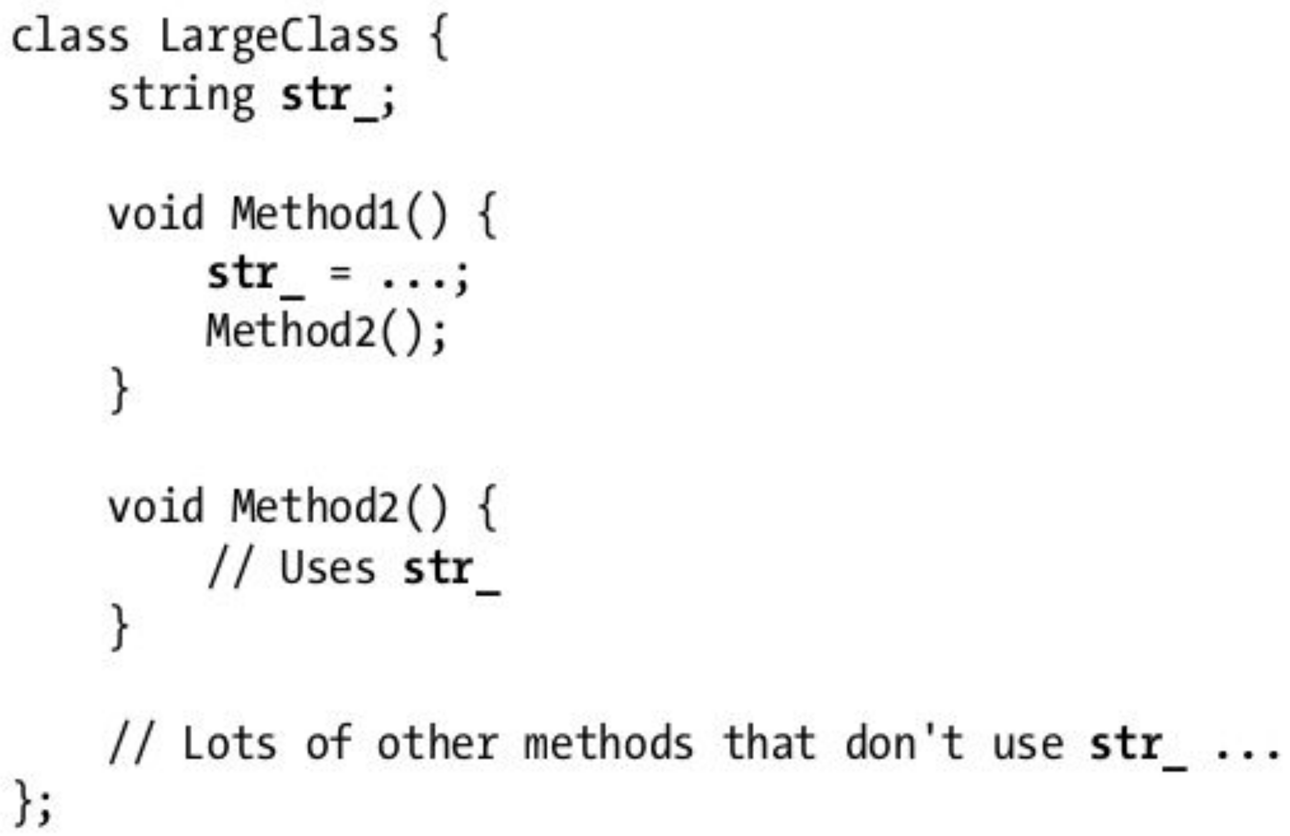
➔ Variables like done are what we call “control flow variables”. In our experience, control flow variables can often be eliminated by making better use of structured programming:



```
while (/* condition */) {
    ...
    if (...) {
        break;
    }
}
```

Shrink the Scope of Your Variables

- We’ve all heard the advice to “avoid global variables.” This is good advice, because it’s hard to keep track of where and how all those global variables are being used.
- In fact, it’s a good idea to “shrink the scope” of all your variables, not just the global ones.



```
class LargeClass {
    string str_;

    void Method1() {
        str_ = ...;
        Method2();
    }

    void Method2() {
        // Uses str_
    }

    // Lots of other methods that don't use str_ ...
};
```

For this case, it may make sense to “demote” str_ to be a local variable:

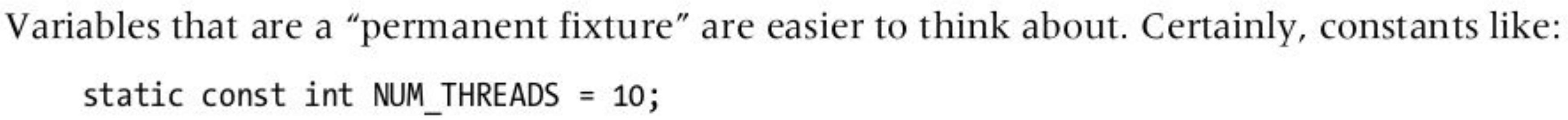
```
class LargeClass {
    void Method1() {
        string str = ...;
        Method2(str);
    }

    void Method2(string str) {
        // Uses str
    }

    // Now other methods can't see str.
};
```

- To restrict access to class members is to **make as many methods static as possible**. Static methods are a great way to let the reader know “these lines of code are isolated from those variables.”
- Or another approach is to **break the large class into smaller classes**. This approach is helpful only if the smaller classes are in fact isolated from each other.

★ **But different languages have different rules for what exactly constitutes a scope.**

Prefer Write-Once Variables

- ➢ The more places a variable is manipulated, the harder it is to reason about its current value.



Variables that are a "permanent fixture" are easier to think about. Certainly, constants like:

```
    static const int NUM_THREADS = 10;
```