A Unary Postfix Operator for Boolean Inversion in C++

Document #: P3883
Date: 2025-10-18
Project: Programming Language C++
Library Evolution Group
Reply-to: Muhammad Taaha
<taaha2004@gmail.com>

1 Introduction

This paper proposes introducing a new **postfix Boolean flip operator** in C++, written as `flag~`, to enable concise, in-place logical inversion of boolean variables.

```
bool flag = true;
flag~;  // equivalent to: flag = !flag;
```

This operator performs a toggle on a modifiable boolean lvalue when used as a statement. Unlike the prefix `~` operator, which already represents bitwise NOT and undergoes integral promotion, the postfix form is **syntactically new**, avoiding any conflict or backward compatibility issues.

2 Motivation and Scope

2.1 The Problem

C++ has no concise shorthand for in-place boolean inversion. The expression:

```
flag = !flag;
```

is both repetitive and inconsistent with mutating operators like `++flag` and `flag++`.

2.2 Real-World Use Cases

Boolean toggles appear frequently across multiple domains:

GUI Development:

```
// Current
isVisible = !isVisible;
isEnabled = !isEnabled;

// Proposed
isVisible~;
isEnabled~;
```

Game Development:

```
// Toggling debug overlays, pause states
debugMode~;
isPaused~;
showFPS~;
```

Embedded Systems:

```
// Hardware pin toggling
motorState~;
ledStatus~;
pumpActive~;
```

State Machines:

```
class Connection {
    bool connected = false;
public:
    void onToggle() { connected~; }
    void onDisconnect() { if (connected) connected~; }
};
```

Configuration Management:

```
struct Settings {
    bool darkMode = false;
    bool notifications = true;

    void toggleDarkMode() { darkMode~; }
    void toggleNotifications() { notifications~; }
};
```

2.3 Why a Postfix Operator?

The existing prefix ~ operator cannot be repurposed without breaking compatibility, as it already applies to all integral types and performs bitwise inversion.

However, postfix `flag~` is currently **invalid syntax in C++**, making it a safe extension point for a new boolean toggle feature.

Key advantages of the postfix position:

- **Consistency with mutation operators**: Like `flag++` and `flag--`, the postfix position clearly signals an in-place modification
- **No syntactic ambiguity**: Distinct from prefix bitwise NOT
- **Clear intent**: The postfix form emphasizes the mutation aspect

3 Design Overview

Aspect	**Specification**
Operator	Postfix ~
Operand	`bool` lvalue
Behavior	Toggles the boolean in-place (`flag = !flag;`)
Return type	`void` (prevents chaining and misuse)
Scope	Boolean lvalues only; rvalues and temporaries are disallowed
Backward compatibility	Fully safe; postfix ~ syntax is currently unused

4 Impact on the Standard

4.1 Backward Compatibility

- Prefix ~ behavior remains completely unchanged.
- All existing valid programs remain valid and unaffected.
- The new operator introduces a fresh postfix grammar rule under `[expr.post]`.
- No existing code uses `flag~` syntax (currently a syntax error).

4.2 Implementation Complexity

- Requires adding a postfix operator rule to `[expr.post]`.
- No standard library changes required.
- No ABI implications.
- Minimal compiler implementation effort (similar to adding any postfix operator).
- Straightforward semantic analysis (simpler than context-dependent alternatives).

5 Example Usage

5.1 Basic Flip

```
bool flag = true;
flag~;  // flag becomes false
flag~;  // flag becomes true
```

5.2 GUI Toggle Example

```
class Widget {
    bool isVisible = true;
public:
    void onClick() { isVisible~; }
    bool visible() const { return isVisible; }
};
```

5.3 Embedded Example

```
bool motor = false;

void loop() {
    if (buttonPressed()) {
        motor~;  // Toggle motor state
        updateHardware(motor);
    }
}
```

5.4 Event Handler Example

```
class AudioPlayer {
    bool muted = false;
public:
    void onMuteButton() { muted~; }
    bool isMuted() const { return muted; }
};
```

6 Proposed Wording

Add the following new production to [expr.post]:

Postfix boolean inversion postfix-expression: postfix-expression `~`

The operand of postfix `~` shall be a modifiable lvalue of type `bool`.

The expression `E~` shall be equivalent to `E = !E`.

The type of the result is `void`.

The evaluation of `E~` is sequenced before any subsequent side effects.

[*Note:* The operand is evaluated and its value is inverted in place. The `void` return type prevents unintended use in expressions while clearly signaling that this is a mutation-only operation. — *end note*]

[*Example:*

```
bool flag = true;
flag~;  // OK: flag becomes false
flag~;  // OK: flag becomes true

const bool c = true;
c~;  // error: operand is not modifiable

bool temp() { return true; }
temp()~;   // error: operand is not an lvalue

auto x = flag~;  // error: cannot assign void
if (flag~) {}    // error: void in boolean context
```

— *end example*]

7 Design Alternatives

7.1 XOR Assignment (`flag ^= true`)

This approach already works in C++ today:

```
bool flag = true;
flag ^= true;  // toggles to false
flag ^= true;  // toggles to true
```

However, it suffers from several significant drawbacks:

- **Non-intuitive**: Most developers don't immediately recognize `^= true` as a toggle operation. It requires understanding XOR truth tables.
- **Redundant operand**: Requires typing `true` explicitly, which feels unnecessary for a unary operation.
- **Bit-manipulation appearance**: Visually resembles low-level bitwise operations rather than high-level logical inversion.
- **Teaching burden**: Requires explaining binary operators and XOR semantics just to understand boolean toggling.
- **Easy to mistype**: Could be written as `^= 1` or `^= false`, leading to confusion.

While functional, `^= true` lacks the clarity, brevity, and consistency with other mutating operators that `flag~` provides.

7.2 Free Function (`std::flip(flag)`)

A library-only solution could provide:

```
#include <utility>
std::flip(flag);  // or std::toggle(flag)
```

Pros:

- No language changes required
- Clear, self-documenting intent
- Easy to implement and standardize

Cons:

- Verbose compared to `flag~`
- Requires `#include <utility>` or similar
- Doesn't feel like a primitive operation
- Inconsistent with increment/decrement operators (`++`/`--`)
- Function call syntax suggests more complexity than a simple toggle

While a free function is workable, it lacks the elegance and consistency of an operator.

7.3 Prefix ~ in Statement Context

An earlier version of this proposal (P3883R0) suggested repurposing prefix `~flag` when used as a standalone statement, while preserving bitwise NOT behavior in expression contexts.

Pros:

- Familiar unary operator syntax
- Reuses existing symbol

Cons:

- **Context-dependent behavior**: The same syntax `~flag` would mean different things depending on whether it appears as a statement or in an expression
- **Semantic ambiguity**: `~flag;` toggles, but `auto x = ~flag;` performs bitwise NOT (and doesn't toggle)
- **Teaching difficulty**: Explaining why the same operator has different effects based on context
- **Implementation complexity**: Requires compiler to perform context-sensitive semantic analysis
- **Potential for confusion**: Users might expect `~flag` to always toggle, leading to bugs

This alternative was rejected because context-dependent operator semantics go against C++'s principle of consistent operator behavior.

7.4 New Keyword or Operator

Options like `flip flag;`, `toggle flag;`, `!!flag;`, or similar could provide clear syntax.

```
flip flag;     // keyword approach
toggle flag;   // keyword approach
!!flag;        // new operator
```

Pros:

- Completely unambiguous
- Self-documenting for keywords

Cons:

- Adding new keywords breaks backward compatibility (any code using `flip` or `toggle` as identifiers would break)
- Extremely high bar for acceptance in C++
- New operator symbols face similar challenges
- Keywords don't fit the operator-style mutation pattern of `++`/`--`

The difficulty of adding new keywords or operator symbols to C++ makes this approach impractical.

7.5 Summary Comparison

Alternative	Pros	Cons
`flag ^= true`	Already works, no language change	Non-intuitive, verbose, looks like bit manipulation, requires XOR knowledge
`std::flip(flag)`	Clear intent, library-only	Verbose, requires include, inconsistent with `++`, function syntax overhead
Prefix `~flag` (statement context)	Familiar syntax	Context-dependent semantics, confusing, complex implementation
New keyword/operator	Unambiguous	Breaks compatibility, extremely high acceptance bar
`flag = !flag;`	Clear, works today	Verbose, repetitive, inconsistent with `++`/`--`
Postfix `flag~` (this proposal)	Clean, safe, consistent with `++`, no backward compat issues	Novel visual form, unfamiliar at first

7.6 Why Postfix ~ Was Chosen

The postfix operator provides the best balance of advantages:

- **No backward compatibility issues**: The syntax is currently invalid, so no existing code breaks
- **Consistency**: Mirrors postfix `++`/`--` for mutation operations
- **Clarity**: Postfix position clearly signals in-place modification
- **Brevity**: Single character, no redundant operands
- **No ambiguity**: Distinct from prefix bitwise NOT
- **Simplicity**: Straightforward semantics with no context-dependence
- **Intuitive**: Once learned, the meaning is obvious

8 Potential Concerns and Responses

8.1 Visual Similarity to Destructors

Concern: The `~` symbol is already used in destructor syntax (`~ClassName()`).

Response: The contexts are syntactically and semantically distinct:

- Destructors appear with class names and parentheses: `~Widget()`, `obj.~MyClass()`
- The toggle operator applies to boolean lvalues in statement position: `flag~;`
- No grammatical ambiguity exists: the parser can always distinguish between a destructor call and a boolean toggle
- Visual similarity is minimal in context: `obj.~Type()` vs `flag~;`

In practice, these constructs appear in completely different contexts and are unlikely to cause confusion.

8.2 Frequency of Use

Concern: Does boolean toggling occur frequently enough to justify new syntax?

Response: While comprehensive corpus analysis would strengthen this proposal, boolean toggles appear regularly in several important domains:

- **GUI frameworks**: Visibility, enabled states, selection, focus
- **Game engines**: Debug flags, pause states, feature toggles, rendering options
- **Embedded systems**: Hardware state management, sensor states, actuator control
- **Configuration management**: Feature flags, user preferences
- **State machines**: Binary state transitions

Additionally, the consistency argument provides motivation beyond raw frequency: if we have `++` and `--` for numeric mutation, having a corresponding operator for boolean mutation improves the language's orthogonality.

The absence of such an operator represents an inconsistency in C++'s operator set rather than evidence that toggles are rare.

8.3 Why Not Just Use `flag = !flag;`?

Concern: The current syntax works fine. Why add new syntax for something that's already possible?

Response: While `flag = !flag;` is functional, this argument could have been made against `++` and `--`:

```
// Why have this:
counter++;

// When this works:
counter = counter + 1;
```

The answer is that mutating operators provide:

- **Brevity**: Less typing, less visual clutter
- **Clarity of intent**: Immediately signals "this is a mutation operation"
- **Reduced error surface**: Variable name appears once, not twice (no risk of typos like `flag = !flg;`)
- **Consistency**: Completes the set of mutating operators for primitive types
- **Expressiveness**: The operation is conceptually atomic, and the syntax reflects that

The same arguments that justified `++` and `--` apply equally to boolean toggle.

8.4 Return Type of `void`

Concern: Postfix operators typically return values. Why does this return `void`?

Response: Returning `void` is a deliberate design choice that:

- **Prevents misuse**: Disallows `if (flag~)` which would be confusing (is it testing the old or new value?)
- **Prevents chaining**: Operations like `flag1~&& flag2` are nonsensical
- **Clarifies intent**: This operator is purely for side effects, not for producing values
- **Avoids ambiguity**: No confusion about whether the returned value is pre-toggle or post-toggle

If users need the value after toggling, they can write:

```
flag~;
if (flag) { /* ... */ }
```

This separation of mutation and value usage leads to clearer code.

9 Implementation Experience

A minimal header-only prototype demonstrates the intended semantics using a wrapper type:

```
// boolflip.hpp
#pragma once

struct flip_bool {
    bool value;

    explicit flip_bool(bool v = false) : value(v) {}

    // Postfix operator~ (int parameter is C++ convention for postfix)
    void operator~(int) { value = !value; }

    // Conversion to bool for use in conditions
    operator bool() const { return value; }

    // Assignment from bool
    flip_bool& operator=(bool v) {
        value = v;
        return *this;
    }
};
```

Usage example:

```
flip_bool flag(true);
flag~;  // toggles to false (calls operator~(int))
flag~;  // toggles to true
if (flag) {  // uses operator bool()
    // flag is true
}

flag = false;  // uses operator=
```

Note: This prototype demonstrates the concept using a wrapper type. The actual language feature would apply directly to the built-in `bool` type without requiring any wrapper, making it zero-overhead and seamlessly integrated with existing boolean variables.

Limitations of the prototype:

- Requires wrapping existing `bool` variables
- Not applicable to `bool` function parameters or returns
- Serves only as a proof-of-concept, not a production solution

The full language feature would work directly with `bool` lvalues in all contexts.

10 Acknowledgements

Thanks to everyone who reviewed earlier drafts of P3883 and provided valuable insight on operator design, consistency, backward compatibility, and alternatives. Special thanks to those who suggested the postfix approach over the original context-dependent prefix design.

11 References

1 CWG Issue 1642 - Integral promotions

2 [expr.post] - Postfix expressions

3 [expr.unary.op] - Unary operators

4 GitHub Prototype - https://github.com/taaha-0548/Cpp-Booleanflip