Switch on Strings: Pointer-to-Character as Controlling Expression

Document Number: N3895
Author: Fred Veldmeijer, Fontys University of Applied Sciences
Date: 2026–06–01
Audience: WG14
Proposal Category: New Features
Target Audience: Developers, Compiler Implementers

Abstract

Many systems programs require branching on string values. Command interpreters, protocol handlers, serializers, and many other applications rely on it. Despite its status as the premier systems programming language, C provides no native syntactic support for this. Programmers must instead write verbose, error-prone, and slow `if-else` chains of `strcmp` calls, resulting in O(*n*·*k*) worst-case complexity (where *n* is the number of comparisons and *k* is the length of the input string).

To address this, we propose relaxing the restriction that the controlling expression of a `switch` statement must have an integer type, allowing pointer-to-character types. This makes the following code valid C:

```
const char *s = get_command();
...
switch (s) {
    case "GET":   handle_read();   break;
    case "PUT":
    case "PATCH": handle_write();  break;
    default:      handle_error();  break;
}
```

This generalization is deliberately conservative: it introduces no new keywords, changes no library functions, and preserves the validity of existing programs. Yet, it delivers expressive syntax, enabling implementations to generate O(*k*) string dispatch rather than the O(*n*·*k*) worst case of a `strcmp` chain.

1. Motivation

String dispatch is a fundamental operation. Whether writing a network protocol parser, an HTTP server, a CLI tool, or a JSON deserializer, developers constantly need to branch based on string input. Because `switch` currently only accepts integer types, the idiomatic way to express this in C is an `if-else` chain:

```
const char *s = get_command();
...
if      (strcmp(s, "GET") == 0)    handle_read();
else if (strcmp(s, "PUT") == 0)    handle_write();
else if (strcmp(s, "PATCH") == 0)  handle_write();
else                               handle_error();
```

However, this pattern suffers from several significant drawbacks:

1. **Algorithmic Inefficiency:** Each `strcmp` traverses the string independently. For *n* cases matched against an input string `p` of length *k*, the chain has a worst-case time complexity of O(*n*·*k*). While a compiler can easily optimize an integer switch into a jump table, an `if-else` chain of function calls is difficult to compile into an O(*k*) dispatch structure.
2. **Error-Proneness:** In a long `if-else` chain, a single dropped `!`, or a copy-paste error in the repeated state variable name will silently break the logic. Because the structure is repetitive, these errors are difficult to spot during code review. As LLM-based code generation lowers the barrier to writing new C, the volume of newly written C containing this repetitive idiom is likely to grow, increasing the surface area for these silent errors.
3. **Obscured Intent:** An `if-else` chain forces the programmer to repeat the variable `s` in every condition. It obscures the semantic intent: that this is a unified dispatch table acting on a single state variable `s`.

4. **Verbosity:** The `strcmp(...) == 0` idiom is syntactically heavy. Furthermore, `strcmp` returns false (0) on equality, directly contradicting the natural reading of `if (strcmp(a, b))` as *if a equals b*. The correct test requires the counterintuitive `== 0` or negation `!strcmp(...)`.
5. **Tooling Overhead:** To avoid the algorithmic inefficiency, developers writing performance-critical code often abandon native C constructs entirely, relying instead on external code generators like `gperf` and `re2c`, or even full lexical analyzers like `flex`. While this achieves O(*k*) performance, it introduces build-system complexity and separates the dispatch logic from the surrounding C code.

Note: `memcmp` may be used because the string length is known, and some compilers may inline it to avoid call overhead. However, this retains the drawbacks described above, and adds the burden of manually supplying the string length. Explicit character-by-character comparisons (`p[0] == 'G' && ...`) are worse still: maximally verbose, error-prone, and obscure the intent completely. Both methods do, however, bound their memory reads: a practical safety property discussed in Section 5.

2. Proposed Solution

2.1 String Switch

The C standard currently restricts `switch` controlling expressions to integer types. This proposal removes that restriction to also permit pointers to character types, making the following code valid C:

```
switch (s) {
    case "GET":    handle_read();   break;
    case "PUT":
    case "PATCH":  handle_write();  break;
    default:       handle_error();  break;
}
```

Unlike an `if-else` chain of function calls, the `case` labels list string literals, giving the compiler complete, compile-time knowledge of the state space. Quality implementations can compile this into a single O(*k*) dispatch, making the resulting code up to *n* times more efficient.

Matching is defined as element-by-element equality of code unit values up to and including the first null character. If the controlling expression evaluates to a null pointer, it matches no `case` labels, and control passes to the `default` label if one exists, or bypasses the switch body entirely. Consistent with `strcmp`, if the expression points to a character array that does not contain a null character, the behavior is undefined.

Matching is intentionally agnostic to Unicode encoding. Surrogate pairs, combining characters, and other multi-unit sequences are treated as sequences of values of the element type, with no normalization applied. Unicode normalization, locale-sensitive collation, and case folding are outside the scope of this proposal.

2.2 Fallthrough and Duplicates

All existing semantics of the `switch` statement remain intact. Fallthrough behavior applies exactly as it does for integer switches. As with integer switches, if two `case` labels in the same `switch` denote identical strings, it is a constraint violation.

Because string literal concatenation occurs prior to semantic analysis, adjacent string literals naturally form a single valid case label. Conversely, if concatenation results in a string identical to another case label, it triggers the standard constraint violation for duplicate cases.

```
#define SCHEME  "http"
#define SECURE  "s"

switch (s) {
    case SCHEME:            handle_http();  break;
    case SCHEME SECURE:
    case "https":           handle_https(); break; // Duplicate case: SCHEME SECURE equals "https"
}
```

2.3 Array-to-Pointer Conversion

Because of C's standard array-to-pointer conversion rules, inline character arrays naturally convert to pointers in this expression context and are fully supported:

```
char buffer[64] = "help";
switch (buffer) { ... }     // Array-to-pointer conversion.
```

2.4 Type Compatibility

The `switch` statement also accepts a controlling expression of a pointer to a character type, including `const`-qualified versions like `const char *`. If the controlling expression is a character pointer, all `case` labels must be string literals.

The pointed-to type of the controlling expression and the element type of the string literal shall be the same character type, disregarding `const`, `volatile`, `signed`, and `unsigned`. Consequently, any string prefixes (`L""`, `u8""`, `u""`, or `U""`) of the `case` labels must correspond to the pointed-to type of the controlling expression; a type mismatch is a constraint violation. This requirement ensures consistent encoding between the controlling expression and the `case` labels by construction, eliminating encoding mismatches at compile time:

```
char *s = "help";
switch (s) {
    case u"help": break; // Constraint violation: char16_t is not compatible with char.
}
```

Consistent with `strcmp` semantics [2, §7.28.4.1], code units of the types `char` and `signed char` are interpreted as `unsigned char`:

```
unsigned char *s = "help";
switch (s) {
    case "help": break; // OK: char is interpreted as unsigned char.
}
```

3. Prior Art and Implementation Experience

String switch is a solved problem in language design. Many languages provide it natively, including C#, D, Go, Java, PHP, and Swift; Ruby and Kotlin support it via `case` or `when`, while Rust, Scala and Python use `match`.

Java is the most instructive precedent. Since its initial release in 1995, the idiomatic way to dispatch on strings was an `if-else` chain of `equals()` calls – closely mirroring the C `strcmp` chains. In 2011, Java SE 7 introduced string switch as one of a set of *small enhancements to improve productivity* [1]. Under the hood, the compiler lowers this into a two-phase dispatch: an initial switch on the string's hash value, followed by a single string comparison to resolve potential collisions.

By internalizing the algorithms currently generated by external tools like `gperf`, `flex`, or `re2c`, a native string switch provides their performance benefits directly within C, free of tooling overhead. Annex A presents four non-normative implementation strategies that demonstrate this: a sequential search, a binary search, a hash-based approach inspired by Java, and a prefix tree inspired by `re2c`.

4. C/C++ Compatibility

This proposal introduces a deliberate C/C++ divergence by lifting a constraint on the controlling expression of the `switch` statement.

C's strength is its simplicity: where C++ addresses string dispatch through metaprogramming facilities such as `std::unordered_map<std::string_view, ...>`, `constexpr` functions, and templates, C solves it with a simple relaxation of an existing statement. In C, macros cannot detect duplicate `case` labels, enforce type compatibility, or participate in the compiler's `switch`-lowering passes, and string dispatch is therefore best addressed at the language level. This is precisely the situation that motivated `_Generic` in C11 and `_BitInt` in C23, but unlike those, this proposal requires only a relaxation of an existing constraint.

The divergence is syntactically clean: the proposed extension does not conflict with any existing C++ construct. Vendors may offer it as a C++ extension at their discretion.

5. Design Decisions

Null Pointers: In C, a null pointer is frequently returned by system APIs to indicate the absence of a value. Rather than adopting the undefined behavior of `<string.h>` functions, this proposal dictates that a controlling expression evaluating to a null pointer acts as a mismatch for all case labels, transferring control to the default label if one exists, or bypassing the switch entirely. This eliminates the need for null-checking prior to the switch statement and prevents a common cause of segmentation faults.

Unterminated Strings: Consistent with C semantics, if the controlling expression points to a character array lacking a null terminator, the behavior is undefined, as it would be for `strcmp` in the same situation. It is worth noting, however, that a quality implementation, having full compile-time knowledge of the case labels, will naturally bound its memory reads to the length of the longest case label. This is a Quality of Implementation property: it provides meaningful protection in practice without altering the formal semantics. The prefix tree implementation in Annex A.4 illustrates this directly: its memory access is bounded by construction, regardless of null termination.

Case Ranges: N3370 introduced integer case ranges [4] into C2Y, allowing syntax such as `case 10 ... 20` to match any integer value in that range. Unlike integers, string dispatch is inherently a membership test: the question is always whether a value equals one of a set of known literals. A lexicographic range such as `"A" ... "Z"` matching `"WG14"` addresses a fundamentally different problem, and one that requires a locale- and encoding-dependent total ordering. Both considerations place string ranges outside the scope of this proposal.

Encoding: A string switch inherits the existing C translation from source to the execution character set unchanged. It introduces no new character set conversions, locale-dependent matching, or Unicode normalization passes. The following two constructs are semantically identical with respect to encoding:

```
if (strcmp(s, "Straße") == 0) handle_street();

switch (s) {
    case "Straße": handle_street();
}
```

Implementation Effort: A potential concern is the burden placed on compiler implementers to generate perfect hash tables or lexer-style DFAs. It is important to note, however, that this proposal dictates only the semantics of the string switch, leaving the lowering strategy entirely to Quality of Implementation.

A compiler is fully compliant if it lowers the `switch` into a sequential `if-else` chain of inline character comparisons. To comply with freestanding requirements without relying on `<string.h>`, a compiler need only emit inline byte-matching loops.

Advanced compilers already possess extensive `switch` lowering passes. Extending existing frontends to group strings by common prefixes, for example, is a well-understood compiler optimization technique.

By moving this logic into the compiler, we relieve the programmer from manually coding string dispatch or integrating extra-lingual code generators (like `gperf` or `re2c`) into their build systems, standardizing a ubiquitous programming pattern.

6. Specification Changes (Informative)

This proposal requires additions to C2Y §6.8.5.3 *The `switch` statement* of the C standard [2].

Constraints: add to the existing constraint list:

> The controlling expression of a `switch` statement shall have integer type or pointer-to-character type. If the controlling expression has pointer-to-character type, the pointed-to type shall not be `volatile`-qualified[1)], and the expression of each case constant expression shall be a string literal whose array element type is compatible with the unqualified pointed-to type of the controlling expression. Two `case` labels in the same `switch` statement shall not denote identical null-terminated character sequences.

Semantics: add after the existing integer-switch semantics paragraph:

> If the controlling expression has pointer-to-character type, it is evaluated exactly once. If the result is a null pointer, it compares unequal to all `case` labels; control passes to the `default` label if one is present, otherwise the `switch` body is bypassed. If the result is not a null pointer but does not point to a null-terminated character array, the behavior is undefined. Otherwise, the null-terminated character sequence is compared against each `case` label's string literal for equality, defined as element-by-element equality of code unit values [2)] up to and including the first null character[3)]. If a match is found, control passes to the matching `case` label. If no match is found, control passes to the `default` label if one is present, otherwise the `switch` body is bypassed. The implementation may evaluate comparisons in any order and may stop comparison once no matching case remains possible.

> [1)]This constraint prevents undefined behavior arising from the repeated character reads an implementation must perform during string comparison (C2Y §6.7.4p7).

> [2)]For `char` and `signed char` controlling expressions, each code unit is compared as if converted to `unsigned char`, consistent with `strcmp` semantics (C2Y §7.28.4.1).

> [3)]The `case` label "GET\0MORE" is functionally identical to "GET". Consequently, both these labels in the same `switch` trigger a duplicate `case` constraint violation (see Section 2.2).

No grammar changes, new keywords, type system changes, changes to linkage or storage duration of string literals, or library additions are required. The existing semantics of `switch` apply unchanged.

7. Summary

This proposal introduces native string dispatch to C without breaking existing code. It does not introduce a new programming paradigm; it formalizes an already ubiquitous systems-programming idiom in a form visible to the compiler. By permitting string pointers as `switch` controlling expressions and exact string literals as `case` labels, we allow the compiler to do what it does best: optimize branch dispatch over a known state space. The result is expressive syntax, null-safety by default, and string dispatch as efficient as C itself.

References

[1] JSR 334: Small Enhancements to the Java Programming Language, Java SE 7, 2011, https://jcp.org/en/jsr/detail?id=334

[2] ISO/IEC 9899:2026, Programming languages – C (working draft N3783), https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3783.pdf

[3] Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo, FNV Hash, https://github.com/lcn2/fnv

[4] WG14 N3370, Case Ranges in switch Statements, 2024, https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3370.htm

Annex A. Implementation Strategies for String Switch (Informative)

String switches can be implemented as a pure frontend transformation. As an example, we take the proposed C code from Section 2.1:

```
switch (s) {
    case "GET":     handle_read();  break;
    case "PUT":
    case "PATCH":   handle_write(); break;
    default:        handle_error(); break;
}
```

The compiler lowers this to:

```
switch (case_index(s)) {
    case 0:     handle_read();  break;  // GET
    case 1:                             // PUT
    case 2:     handle_write(); break;  // PATCH
    default:    handle_error(); break;
}
```

Because this mirrors the programmer's AST exactly, C's scoping and implicit fallthrough behave identically to regular switches. In addition, it allows the backend to use its existing (typically optimized) integer switch logic.

To guarantee the bounds safety described in Section 5, implementations using `strcmp` or hashing can emit a small inline helper. This helper ensures the input is a non-null pointer to a character array containing a null character within the size of the longest `case` literal. It prevents unbounded runaway reads and incidentally short-circuits inputs longer than any valid case label directly to `default`:

```
static inline bool is_valid_string(const char *s, size_t max_size) {
    if (!s) return false;

    for (size_t i = 0; i < max_size; i++) {
        if (s[i] == '\0') return true;
    }
    return false;
}
```

Below are four conceptual implementations of the `case_index` function, in increasing order of sophistication.

A.1 Chain of `strcmp` Calls

The simplest conforming lowering emits an `if-else` chain of `strcmp` calls, matching cases in source order:

```
static int case_index(const char *s) {
    if (!is_valid_string(s, sizeof("PATCH"))) return -1;

    if (strcmp(s, "GET")   == 0) return 0;
    if (strcmp(s, "PUT")   == 0) return 1;
    if (strcmp(s, "PATCH") == 0) return 2;
    return -1; // Default
}
```

Null-pointer safety is handled explicitly before the first `strcmp` call, preserving the defined semantics without invoking undefined behavior. In freestanding environments where `<string.h>` is unavailable, the compiler can trivially replace the `strcmp` calls with inline byte-matching loops.

This lowering has O($n \cdot k$) worst-case complexity, identical to a hand-written `strcmp` chain. It is, however, correct by construction: the compiler generates the repetitive boilerplate, guaranteeing unique case labels and eliminating copy-paste errors and intent-obscuring structure that motivate this proposal.

A.2 Binary Search

A straightforward refinement is to sort the string literals and implement the dispatch as a binary search, improving worst-case complexity from O(*n*·*k*) to O(*k*·log *n*):

```
static int case_index(const char *s) {
    if (!is_valid_string(s, sizeof("PATCH"))) return -1;

    static const char *case_labels[] = { "GET", "PATCH", "PUT" };
    static const int case_indexes[] = { 0, 2, 1 };

    for (int lo = 0, hi = 3; lo < hi; ) {
        int mid = (lo + hi) / 2;
        int cmp = strcmp(s, case_labels[mid]);

        if (cmp == 0) return case_indexes[mid];

        if (cmp < 0) hi = mid;
        else         lo = mid + 1;
    }
    return -1;
}
```

Given its small binary footprint, this approach nicely balances ease of implementation and performance for up to roughly 64 cases. Alternatively, the standard library `bsearch` can be used, though an inline loop avoids the overhead of calling the comparator function.

A.3 Hash and Switch

An alternative to binary search is to collapse the string down to a single integer, allowing the compiler to leverage its existing integer `switch` machinery. The compiler maps strings to their corresponding case index via a runtime hash function (such as 32-bit FNV-1a [3] in this example):

```
static inline uint32_t hash(const char *s) {
    uint32_t h = 0x811C9DC5;
    while (*s) h = (h ^ (unsigned char)*s++) * 0x01000193;
    return h;
}

static int case_index(const char *s) {
    if (!is_valid_string(s, sizeof("PATCH"))) return -1;
    int index = -1;

    switch (hash(s)) {
        case 0x96E6BE77: // For illustration: hash("GET") collides with hash("PUT").
            if (strcmp(s, "GET") == 0) index = 0;
            else if (strcmp(s, "PUT") == 0) index = 1;
            break;

        case 0xD08BBE49: // The value of hash("PATCH").
            if (strcmp(s, "PATCH") == 0) index = 2;
            break;
    }
    return index;
}
```

The collision between hash(“GET”) and hash(“PUT”) is deliberately constructed to show collision handling; in practice, the probability of collision is negligible, and `strcmp` is typically called at most once per lookup. This gives O(*k*) expected performance, independent of the number of cases. A Minimal Perfect Hash Function (MPHF), as generated by tools such as GNU `gperf`, may be used to eliminate collisions among the `case` labels, but the `strcmp` fallback remains necessary for unknown inputs, so the added implementation complexity is difficult to justify.

A.4 Prefix Tree

While hashing is robust, an alternative strategy used by lexers (such as `re2c`) is to compile the literal strings into a **Prefix Tree**: a Deterministic Finite Automaton (DFA) that contains no loops. This approach has two distinct advantages:

1. **Better Performance Through Early Exit:** Matching halts as soon as the input diverges from all valid cases, potentially after reading only a few characters; well under O(*k*) in practice.
2. **Inherent Protection Against Runaway Reads:** Because the maximum depth of the tree is fixed at compile time, the generated code inherently bounds its memory access to that depth, preventing unbounded reads.

The compiler groups the `case` strings by their common prefixes:

```
static int case_index(const char *s) {
    if (!s) return -1; // Default: null pointer matches no case labels.
    int index = -1;

    start_dfa:
        switch (*s) {
            case 'G': s++; goto prefix_G;
            case 'P': s++; goto prefix_P;
        }
        goto end_dfa; // Fails immediately, e.g. on "DELETE".

    prefix_G:
        if (*s++ != 'E') goto end_dfa;
        if (*s++ != 'T') goto end_dfa;
        if (*s == '\0') index = 0;
        goto end_dfa;

    prefix_P:
        switch (*s) {
            case 'U': s++; goto prefix_PU;
            case 'A': s++; goto prefix_PA;
        }
        goto end_dfa;

    prefix_PU:
        if (*s++ != 'T') goto end_dfa;
        if (*s == '\0') index = 1;
        goto end_dfa;

    prefix_PA:
        if (*s++ != 'T') goto end_dfa;
        if (*s++ != 'C') goto end_dfa;
        if (*s++ != 'H') goto end_dfa;
        if (*s == '\0') index = 2;
        goto end_dfa;

    end_dfa:
        return index;
}
```