IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 7, June 2025
Impact Factor: 7.67

Bringing C23 Features to Life: Implementing Binary I/O and Enhanced Static Assertions in C

Rupatai V. Lichode[1], Pranay D. Shende[2], Pooja B. Pusalwar[3], Samyak G. Patil[4], Rachana B. Shadangule[5], Md. Kadim R. Sheikh[6]
Guide, Department of Computer Science and Engineering[1]
Students, Department of Computer Science and Engineering[2,3,4,5,6]
Rajiv Gandhi College of Engineering, Research and Technology, Chandrapur, Maharashtra, India

Abstract: *The C23 standard introduces several modernizations to the C programming language, aiming for better consistency with C++ and enhanced developer ergonomics. This paper explores the practical implementation of four significant C23 features: formatted binary output using printf, formatted binary input using scanf, the harmonization of static_assert with C++ and the reintroduced memccpy function for efficient string copying. These implementations demonstrate how modern capabilities can be integrated into C, improving code clarity, reducing developer overhead, and encouraging best practices. We present implementation strategies, detailed code-level modifications, real-world use cases, and a discussion of potential impacts on performance and compatibility. Our work illustrates how small yet impactful features from the C23 standard can be brought to life through compiler and library-level enhancements.*

Keywords: *C23 standard*

I. INTRODUCTION

The C programming language has long been a cornerstone of systems programming due to its efficiency, portability, and low-level control. However, its minimalistic design has also led to a lack of features that enhance developer productivity and code safety. Over the years, languages like C++ have introduced improvements that make development more intuitive without sacrificing performance. To bridge this gap, the ISO C working group has released the C23 standard, introducing features aimed at improving usability, enhancing compatibility with C++, and modernizing the language.

This paper focuses on the implementation of three practical features from the C23 standard:

- Formatted binary output using printf with a %b specifier.
- Formatted binary input using scanf with a %b specifier.
- Harmonization of the static_assert macro with its C++ counterpart.
- The reintroduced memccpy function for efficient string copying.

These features were selected for their immediate utility in real-world applications, particularly in low-level systems and embedded development.

II. BACKGROUND & RELATED WORK

Before the introduction of these C23 features, binary input/output required manual implementation. Programmers had to write custom routines to convert integers to binary strings and vice versa, often leading to verbose and error-prone code. For example, printing a binary representation of an integer typically involved writing a custom loop using bitwise operations and string formatting.

In C11, the static_assert macro required two arguments: a compile-time condition and a custom message. In contrast, C++ allowed a more concise form with a single condition, making it easier to write and read static assertions. Harmonizing this behavior in C23 ensures consistency across languages and reduces friction in mixed C/C++ codebases.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 7, June 2025
Impact Factor: 7.67

Related work includes proposals to extend the C standard library and work on compiler frontends that provide extended format specifiers. However, concrete implementations of these C23 features have been limited in academic and open-source communities.

III. IMPLEMENTATION DETAILS

a. Formatted Binary Output with printf: To implement the %b specifier in printf, we modified the internal format string parser of the standard library (glibc). When the parser encounters %b, it routes the corresponding argument through a binary conversion function. This function uses bitwise operations to extract individual bits from the integer and assembles them into a string. The output string is then passed to the output stream in place of the original integer.

printf("%b", 10); // Output: 1010

We ensured that the %b specifier supports width and padding options, making it consistent with other format specifiers.

b. Formatted Binary Input with scanf: For scanf, we extended the format handler to recognize %b as a valid specifier. The corresponding handler function reads a string of binary digits, verifies that all characters are either '0' or '1', and computes the integer equivalent using powers of two.

int x;

scanf("%b", &x); // Input: 1101 => x = 13

Error handling was added to detect invalid characters and overflow conditions, providing robust input parsing consistent with other scanf specifiers.

c. Harmonizing static_assert: The traditional static_assert in C11 required a message, making simple assertions unnecessarily verbose:

static_assert(sizeof(int) == 4, "int must be 4 bytes");

In C23, the message is optional:

static_assert(sizeof(int) == 4);

To implement this, we used conditional macro definitions that detect compiler support for C23. Where available, the macro expands to the built-in _Static_assert with one or two parameters as needed. This improves compatibility with existing C++ code and simplifies the syntax for developers.

d. Efficient String Copying with memccpy: memccpy was previously part of POSIX and earlier C standards but was removed in C17 due to unclear behavior and limited adoption. However, real-world usage—especially in embedded and system-level code—demonstrated the function's utility for copying up to a delimiter efficiently. As a result, it was reinstated in C23.

Function Signature

void *memccpy(void *restrict dest, const void *restrict src, int c, size_t n);

It copies bytes from src to dest until either:

c is encountered (and copied), or

n bytes have been copied, whichever comes first.

Implementation Notes

We implemented memccpy using a simple loop over the byte stream, checking each byte for the delimiter (c). When c is found, copying stops and the function returns a pointer to the byte after the copied character in dest.

Examples & Use Cases

Embedded Systems Development: Bitwise register configuration is a common task in embedded C. Using %b simplifies debugging and configuration by printing values in binary directly, avoiding manual conversion or reliance on third-party libraries.

uint8_t config = 0b11001100;

printf("Register: %08b\n", config);

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 7, June 2025
Impact Factor: 7.67

Configurable Static Checks: Static assertions are often used in system configuration headers to validate assumptions about data sizes or feature availability. Simplified syntax reduces boilerplate and improves readability.

```
static_assert(MAX_BUFFER_SIZE <= 1024);
char source[] = "hello world";
char dest[20];

char *ptr = memccpy(dest, source, ' ', sizeof(source));
// Copies "hello " into dest
if (ptr != NULL) {
   *ptr = '\0'; // Optionally terminate the string
}
```

IV. EVALUATION

We tested the modified standard library with GCC by building a custom runtime. We designed a comprehensive test suite covering valid and invalid inputs for %b, formatting with width and padding, and various assertion conditions. Benchmarks revealed negligible performance overhead for the new format specifiers. Compatibility testing confirmed that legacy code was unaffected, and fallbacks were provided for compilers that do not support C23. With memccpy, developers now have a standardized and safer tool for partial string copying. It avoids the overhead of manually scanning and copying strings in multiple steps. Our implementation performed reliably across edge cases such as missing delimiters, overlapping buffers, and null characters.

V. CONCLUSION & FUTURE WORK

The implementation of these selected C23 features showcases the evolving capabilities of the C language and highlights the benefits of adopting modern syntax and semantics. By integrating binary I/O and enhanced assertions, we improve both developer productivity and code maintainability. Future directions include contributing these features to mainstream compiler projects, extending support to additional C23 specifiers, and developing educational tools to promote adoption of the standard.

REFERENCES

[1]. ISO/IEC JTC1/SC22/WG14: Working Draft for C23
[2]. Ritchie, D.M. (1993). The Development of the C Language.
[3]. Lemire, D. (2024). C23: A Slightly Better C.
[4]. ResearchGate. About Some Innovations of the C Programming Language Standard C23.
[5]. GNU Compiler Collection (GCC) Documentation