# 第14章 C23の新機能（オプション）

## 📋 対応C規格

* **主要対象:** C23
* **学習内容:** bool型、typeof演算子、nullptr、2進数リテラル、その他の新機能

## 🎯 学習目標

この章を完了すると、以下のことができるようになります：

* C23で標準化されたbool型を使用できる
* typeof演算子で型を扱える
* nullptrを使った型安全なコードが書ける
* 2進数リテラルを活用できる
* C23の新機能を実践的に活用できる

## 📚 理論解説

### C23とは

C23（正式にはISO/IEC 9899:2024）は、C言語の最新規格です。C99以来の大規模な改訂となり、現代的なプログラミングのニーズに対応した多くの新機能が追加されました。

### 主要な新機能

#### 1. bool型の標準化

C23では、<stdbool.h>をインクルードすることなく、bool、true、falseが使用可能になりました。

/\* C90/C99 \*/  
#include <stdbool.h> /\* 必要 \*/  
bool flag = true;  
  
/\* C23 \*/  
bool flag = true; /\* ヘッダー不要 \*/

#### 2. typeof演算子

式の型を取得できる演算子です。マクロでの型安全性向上に役立ちます。

int x = 42;  
typeof(x) y = 100; /\* yはint型 \*/  
  
/\* 型安全なマクロ \*/  
#define SWAP(a, b) do { \  
 typeof(a) temp = (a); \  
 (a) = (b); \  
 (b) = temp; \  
} while(0)

#### 3. nullptr定数

型安全なNULLポインタ定数です。

/\* 従来 \*/  
int \*p1 = NULL; /\* NULLは0やvoid\*として定義 \*/  
  
/\* C23 \*/  
int \*p2 = nullptr; /\* 明確にポインタ型 \*/

#### 4. 2進数リテラル

0bまたは0Bプレフィックスで2進数を直接記述できます。

int binary = 0b1010; /\* 10進数の10 \*/  
int flags = 0b11111111; /\* 10進数の255 \*/  
  
/\* ビット演算での活用 \*/  
#define READ\_BIT 0b100  
#define WRITE\_BIT 0b010  
#define EXEC\_BIT 0b001

### その他の新機能（一部）

* auto型推論（制限付き）
* constexpr（コンパイル時定数）
* \_BitInt(N)（任意ビット幅整数）
* 属性構文（[[deprecated]]など）
* プリプロセッサの拡張（#elifdef、#elifndef）

## 🔍 サンプルコード

### [bool型の基本](examples/bool_basics.c)

bool is\_ready = true;  
bool has\_error = false;  
  
if (is\_ready && !has\_error) {  
 printf("システム正常\n");  
}

### [数値リテラル](examples/numeric_literals.c)

int binary = 0b1010; /\* 2進数 \*/  
int octal = 052; /\* 8進数 \*/  
int decimal = 42; /\* 10進数 \*/  
int hex = 0x2A; /\* 16進数 \*/

### [typeof演算子](examples/typeof_demo.c)

#define MAX(a, b) ({ \  
 typeof(a) \_a = (a); \  
 typeof(b) \_b = (b); \  
 \_a > \_b ? \_a : \_b; \  
})

### [nullptr定数](examples/nullptr_demo.c)

void process\_data(int \*data) {  
 if (data != nullptr) {  
 /\* データ処理 \*/  
 }  
}

## 💻 演習問題

### 演習14-1: bool型の活用

<exercises/README.md>を参照

### 演習14-2: ビット操作と2進数リテラル

<exercises/README.md>を参照

### 演習14-3: typeof演算子の応用

<exercises/README.md>を参照

## 🚀 コンパイル方法

この章はC23専用です。以下のコマンドでコンパイルしてください：

# 個別ファイルのコンパイル  
gcc -std=c23 -Wall -Wextra -pedantic source.c -o output  
  
# Makefileを使用  
make all # すべてコンパイル  
make test # C23サポートテスト  
make run-all # すべて実行

### コンパイラサポート状況

C23は新しい規格のため、コンパイラサポートは発展途上です：

* **GCC**: 13以降で部分サポート（-std=c23）
* **Clang**: 16以降で部分サポート（-std=c23）
* **MSVC**: 未対応（2024年現在）

## ⚠️ 注意事項

1. **コンパイラ依存**: すべてのC23機能がサポートされているとは限りません
2. **移植性**: 古いコンパイラでは動作しません
3. **学習順序**: C90/C99の基礎を理解してから学習することを推奨

## 📖 参考資料

* [C23規格ドラフト](https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3096.pdf)
* [C23新機能の概要](https://en.cppreference.com/w/c/23)
* コンパイラのドキュメント（GCC、Clang）

## 🔄 C90/C99/C11からの移行

### bool型

/\* C90 \*/  
#define TRUE 1  
#define FALSE 0  
int flag = TRUE;  
  
/\* C99 \*/  
#include <stdbool.h>  
bool flag = true;  
  
/\* C23 \*/  
bool flag = true; /\* ヘッダー不要 \*/

### NULLポインタ

/\* C90/C99/C11 \*/  
int \*p = NULL;  
  
/\* C23 \*/  
int \*p = nullptr;

### 型の取得

/\* C11: \_Generic \*/  
#define TYPE\_NAME(x) \_Generic((x), \  
 int: "int", \  
 double: "double", \  
 default: "other")  
  
/\* C23: typeof \*/  
typeof(x) y; /\* xと同じ型 \*/

**注**: この章はオプション的な内容です。実務では、使用するコンパイラのC23サポート状況を確認してから活用してください。

# サンプルコード

## bool\_basics.c

```c / ファイル名: bool\_basics.c \* 説明: C23のbool型の基本的な使用例 \* 規格: C23 \*/

#include <stdio.h>

/\* C23では<stdbool.h>なしでbool, true, falseが使用可能 \*/

/\* 関数プロトタイプ宣言 \*/ bool check\_system(void);

int main(void) { printf(“=== C23 bool型の基本 ===”);

/\* bool型の宣言と初期化 \*/  
bool is\_ready = true;  
bool has\_error = false;  
  
printf("is\_ready: %s\n", is\_ready ? "true" : "false");  
printf("has\_error: %s\n", has\_error ? "true" : "false");  
  
/\* bool型のサイズ \*/  
printf("\nbool型のサイズ: %zu バイト\n", sizeof(bool));  
  
/\* 論理演算 \*/  
bool result = is\_ready && !has\_error;  
printf("\nis\_ready && !has\_error = %s\n", result ? "true" : "false");  
  
/\* 条件式での使用 \*/  
if (is\_ready) {  
 printf("\nシステムは準備完了です\n");  
}  
  
/\* 関数の戻り値として使用 \*/  
bool status = check\_system();  
printf("\nシステムチェック結果: %s\n", status ? "正常" : "異常");  
  
/\* 配列での使用 \*/  
bool flags[5] = {true, false, true, true, false};  
printf("\nフラグ配列: ");  
for (int i = 0; i < 5; i++) {  
 printf("%d ", flags[i]);  
}  
printf("\n");  
  
/\* C90/C99との比較 \*/  
printf("\n=== C90/C99との比較 ===\n");  
printf("C90: int型や#defineマクロで真偽値を表現\n");  
printf("C99: <stdbool.h>をインクルードして使用\n");  
printf("C23: 標準でbool/true/falseが利用可能\n");  
  
return 0;

}

/\* bool型を返す関数 */ bool check\_system(void) { /* システムチェックのシミュレーション */ return true; /* 正常 \*/ }

/ 実行結果例: \* === C23 bool型の基本 === \* is\_ready: true \* has\_error: false \* \* bool型のサイズ: 1 バイト \* \* is\_ready && !has\_error = true \* \* システムは準備完了です \* \* システムチェック結果: 正常 \* \* フラグ配列: 1 0 1 1 0 \* \* === C90/C99との比較 === \* C90: int型や#defineマクロで真偽値を表現 \* C99: <stdbool.h>をインクルードして使用 \* C23: 標準でbool/true/falseが利用可能 \*/```

## nullptr\_demo.c

```c / ファイル名: nullptr\_demo.c \* 説明: C23のnullptr定数の使用例 \* 規格: C23 \*/

#include <stdio.h> #include <stddef.h> #include <stdlib.h>

/\* 関数オーバーロードのシミュレーション（Cでは名前を変える） */ void process\_int(int value); void process\_ptr(void* ptr);

/\* nullptrチェック関数 */ bool is\_nullptr(void* ptr) { return ptr == nullptr; }

/\* リンクリストノード */ typedef struct Node { int data; struct Node* next; } Node;

int main(void) { printf(“=== C23 nullptr定数 ===”);

/\* 基本的な使用法 \*/  
printf("\n--- 基本的な使用法 ---\n");  
  
int \*p1 = nullptr;  
char \*p2 = nullptr;  
void \*p3 = nullptr;  
  
printf("p1 == nullptr: %s\n", p1 == nullptr ? "true" : "false");  
printf("p2 == nullptr: %s\n", p2 == nullptr ? "true" : "false");  
printf("p3 == nullptr: %s\n", p3 == nullptr ? "true" : "false");  
  
/\* NULLとnullptrの比較 \*/  
printf("\n--- NULLとnullptrの比較 ---\n");  
  
int \*old\_style = NULL;  
int \*new\_style = nullptr;  
  
printf("old\_style == NULL: %s\n", old\_style == NULL ? "true" : "false");  
printf("new\_style == nullptr: %s\n", new\_style == nullptr ? "true" : "false");  
printf("old\_style == new\_style: %s\n", old\_style == new\_style ? "true" : "false");  
  
/\* 型安全性の向上 \*/  
printf("\n--- 型安全性の向上 ---\n");  
  
/\* C23では、nullptrは明確にポインタ型 \*/  
process\_ptr(nullptr); /\* OK: nullptrはポインタ \*/  
/\* process\_int(nullptr); \*/ /\* エラー: nullptrは整数ではない \*/  
  
/\* 従来のNULLの問題 \*/  
process\_int(0); /\* OK: 0は整数 \*/  
process\_ptr(NULL); /\* OK: NULLはポインタとして扱われる \*/  
/\* process\_int(NULL); \*/ /\* 環境依存: NULLが0として定義されている場合は動作 \*/  
  
/\* 関数での使用 \*/  
printf("\n--- 関数での使用 ---\n");  
  
void \*ptr1 = malloc(100);  
void \*ptr2 = nullptr;  
  
printf("ptr1 is nullptr: %s\n", is\_nullptr(ptr1) ? "true" : "false");  
printf("ptr2 is nullptr: %s\n", is\_nullptr(ptr2) ? "true" : "false");  
  
free(ptr1);  
  
/\* リンクリストでの使用 \*/  
printf("\n--- リンクリストでの使用 ---\n");  
  
Node \*head = nullptr;  
Node \*current = nullptr;  
  
/\* ノードの追加 \*/  
for (int i = 1; i <= 3; i++) {  
 Node \*new\_node = malloc(sizeof(Node));  
 if (new\_node != nullptr) {  
 new\_node->data = i;  
 new\_node->next = nullptr;  
   
 if (head == nullptr) {  
 head = new\_node;  
 current = head;  
 } else {  
 current->next = new\_node;  
 current = new\_node;  
 }  
 }  
}  
  
/\* リストの表示 \*/  
printf("リスト: ");  
for (Node \*node = head; node != nullptr; node = node->next) {  
 printf("%d ", node->data);  
}  
printf("\n");  
  
/\* メモリ解放 \*/  
while (head != nullptr) {  
 Node \*temp = head;  
 head = head->next;  
 free(temp);  
}  
  
/\* 条件式での使用 \*/  
printf("\n--- 条件式での使用 ---\n");  
  
char \*str = nullptr;  
  
if (str == nullptr) {  
 printf("文字列は初期化されていません\n");  
}  
  
str = "Hello, C23!";  
  
if (str != nullptr) {  
 printf("文字列: %s\n", str);  
}  
  
/\* マクロでの使用 \*/  
printf("\n--- マクロでの使用 ---\n");  
  
#define SAFE\_FREE(ptr) do { \  
 if ((ptr) != nullptr) { \  
 free(ptr); \  
 (ptr) = nullptr; \  
 } \  
} while(0)  
  
int \*dynamic = malloc(sizeof(int));  
\*dynamic = 42;  
printf("解放前: dynamic = %p\n", (void\*)dynamic);  
  
SAFE\_FREE(dynamic);  
printf("解放後: dynamic = %p (nullptr)\n", (void\*)dynamic);  
  
return 0;

}

/\* 関数の実装 \*/ void process\_int(int value) { printf(“整数値を処理: %d”, value); }

void process\_ptr(void \*ptr) { printf(“ポインタを処理: %p”, ptr); }

/ 実行結果例: \* === C23 nullptr定数 === \* \* — 基本的な使用法 — \* p1 == nullptr: true \* p2 == nullptr: true \* p3 == nullptr: true \* \* — NULLとnullptrの比較 — \* old\_style == NULL: true \* new\_style == nullptr: true \* old\_style == new\_style: true \* \* — 型安全性の向上 — \* ポインタを処理: 0x0 \* 整数値を処理: 0 \* ポインタを処理: 0x0 \* \* — 関数での使用 — \* ptr1 is nullptr: false \* ptr2 is nullptr: true \* \* — リンクリストでの使用 — \* リスト: 1 2 3 \* \* — 条件式での使用 — \* 文字列は初期化されていません \* 文字列: Hello, C23! \* \* — マクロでの使用 — \* 解放前: dynamic = 0x1234abcd \* 解放後: dynamic = 0x0 (nullptr) \*/```

## numeric\_literals.c

```c / ファイル名: numeric\_literals.c \* 説明: C23の数値リテラル機能（2進数リテラル、数値区切り） \* 規格: C23 \*/

#include <stdio.h>

int main(void) { printf(“=== C23 数値リテラルの新機能 ===”);

/\* 2進数リテラル (0b または 0B プレフィックス) \*/  
printf("\n--- 2進数リテラル ---\n");  
  
int binary1 = 0b1010; /\* 10進数の10 \*/  
int binary2 = 0B11111111; /\* 10進数の255 \*/  
int binary3 = 0b10101010; /\* 10進数の170 \*/  
  
printf("0b1010 = %d\n", binary1);  
printf("0B11111111 = %d\n", binary2);  
printf("0b10101010 = %d (0x%X)\n", binary3, binary3);  
  
/\* ビット演算での活用 \*/  
printf("\n--- ビット演算での活用 ---\n");  
  
unsigned char flags = 0b00000000;  
flags |= 0b00000001; /\* ビット0をセット \*/  
flags |= 0b00001000; /\* ビット3をセット \*/  
flags |= 0b10000000; /\* ビット7をセット \*/  
  
printf("フラグ: 0b");  
for (int i = 7; i >= 0; i--) {  
 printf("%d", (flags >> i) & 1);  
}  
printf(" (%d)\n", flags);  
  
/\* マスクパターンの定義 \*/  
printf("\n--- マスクパターン ---\n");  
  
#define READ\_PERMISSION 0b100 /\* 読み取り許可 \*/  
#define WRITE\_PERMISSION 0b010 /\* 書き込み許可 \*/  
#define EXEC\_PERMISSION 0b001 /\* 実行許可 \*/  
  
unsigned int permissions = READ\_PERMISSION | WRITE\_PERMISSION;  
printf("権限: 0b%03b (読み取り%s, 書き込み%s, 実行%s)\n",  
 permissions,  
 (permissions & READ\_PERMISSION) ? "○" : "×",  
 (permissions & WRITE\_PERMISSION) ? "○" : "×",  
 (permissions & EXEC\_PERMISSION) ? "○" : "×");  
  
/\* 数値区切り（コンパイラサポート次第） \*/  
printf("\n--- 数値区切り（C23で検討中） ---\n");  
  
/\* 注: 数値区切りは最終仕様では採用されない可能性あり \*/  
/\* int large\_number = 1'000'000; \*/ /\* コンパイラサポート待ち \*/  
  
int large\_number = 1000000;  
printf("大きな数値: %d\n", large\_number);  
  
/\* 他の基数での2進数リテラル \*/  
printf("\n--- 様々な基数との比較 ---\n");  
  
int value = 42;  
printf("10進数: %d\n", value);  
printf("16進数: 0x%X\n", value);  
printf("8進数: 0%o\n", value);  
printf("2進数: 0b");  
for (int i = 7; i >= 0; i--) {  
 printf("%d", (value >> i) & 1);  
}  
printf("\n");  
  
/\* 実用例: レジスタ設定 \*/  
printf("\n--- 実用例: レジスタ設定 ---\n");  
  
/\* ハードウェアレジスタのビット定義 \*/  
#define REG\_ENABLE 0b10000000  
#define REG\_INTERRUPT 0b01000000  
#define REG\_MODE\_MASK 0b00110000  
#define REG\_MODE\_A 0b00000000  
#define REG\_MODE\_B 0b00010000  
#define REG\_MODE\_C 0b00100000  
  
unsigned char reg = REG\_ENABLE | REG\_MODE\_B;  
printf("レジスタ値: 0b%08b (0x%02X)\n", reg, reg);  
  
return 0;

}

/ 実行結果例: \* === C23 数値リテラルの新機能 === \* \* — 2進数リテラル — \* 0b1010 = 10 \* 0B11111111 = 255 \* 0b10101010 = 170 (0xAA) \* \* — ビット演算での活用 — \* フラグ: 0b10001001 (137) \* \* — マスクパターン — \* 権限: 0b110 (読み取り○, 書き込み○, 実行×) \* \* — 数値区切り（C23で検討中） — \* 大きな数値: 1000000 \* \* — 様々な基数との比較 — \* 10進数: 42 \* 16進数: 0x2A \* 8進数: 052 \* 2進数: 0b00101010 \* \* — 実用例: レジスタ設定 — \* レジスタ値: 0b10010000 (0x90) \*/```

## typeof\_demo.c

```c / ファイル名: typeof\_demo.c \* 説明: C23のtypeof演算子の使用例 \* 規格: C23 \*/

#include <stdio.h> #include <string.h>

/\* マクロでtypeofを活用 \*/ #define SWAP(a, b) do {  
typeof(a) temp = (a);  
(a) = (b);  
(b) = temp;  
} while(0)

/\* 型安全な最大値マクロ \*/ #define MAX(a, b) ({  
typeof(a) \_a = (a);  
typeof(b) \_b = (b);  
\_a > \_b ? \_a : \_b;  
})

/\* 配列要素数の計算（型チェック付き） \*/ #define ARRAY\_SIZE(arr) (sizeof(arr) / sizeof(typeof(arr[0])))

int main(void) { printf(“=== C23 typeof演算子 ===”);

/\* 基本的な使用法 \*/  
printf("\n--- 基本的な使用法 ---\n");  
  
int x = 42;  
typeof(x) y = 100; /\* yはint型 \*/  
  
printf("x = %d, y = %d\n", x, y);  
printf("yの型はxと同じ（int型）\n");  
  
/\* 複雑な型での使用 \*/  
printf("\n--- 複雑な型での使用 ---\n");  
  
double array[5] = {1.1, 2.2, 3.3, 4.4, 5.5};  
typeof(array[0]) sum = 0.0; /\* sumはdouble型 \*/  
  
for (int i = 0; i < 5; i++) {  
 sum += array[i];  
}  
printf("配列の合計: %.1f\n", sum);  
  
/\* ポインタ型での使用 \*/  
printf("\n--- ポインタ型での使用 ---\n");  
  
char \*str = "Hello";  
typeof(str) copy = "World"; /\* copyはchar\*型 \*/  
  
printf("str = %s, copy = %s\n", str, copy);  
  
/\* SWAPマクロの使用 \*/  
printf("\n--- 型安全なSWAPマクロ ---\n");  
  
int a = 10, b = 20;  
printf("交換前: a = %d, b = %d\n", a, b);  
SWAP(a, b);  
printf("交換後: a = %d, b = %d\n", a, b);  
  
double d1 = 3.14, d2 = 2.71;  
printf("\n交換前: d1 = %.2f, d2 = %.2f\n", d1, d2);  
SWAP(d1, d2);  
printf("交換後: d1 = %.2f, d2 = %.2f\n", d1, d2);  
  
/\* MAXマクロの使用 \*/  
printf("\n--- 型安全な最大値マクロ ---\n");  
  
int max\_int = MAX(15, 25);  
double max\_double = MAX(3.14, 2.71);  
  
printf("MAX(15, 25) = %d\n", max\_int);  
printf("MAX(3.14, 2.71) = %.2f\n", max\_double);  
  
/\* 配列サイズの計算 \*/  
printf("\n--- 配列サイズの計算 ---\n");  
  
int int\_array[] = {1, 2, 3, 4, 5};  
char char\_array[] = "Hello";  
struct { int x; int y; } point\_array[3];  
  
printf("int配列の要素数: %zu\n", ARRAY\_SIZE(int\_array));  
printf("char配列の要素数: %zu\n", ARRAY\_SIZE(char\_array));  
printf("構造体配列の要素数: %zu\n", ARRAY\_SIZE(point\_array));  
  
/\* 関数ポインタでの使用 \*/  
printf("\n--- 関数ポインタでの使用 ---\n");  
  
int (\*func\_ptr)(const char \*, ...) = printf;  
typeof(func\_ptr) another\_func = func\_ptr;  
  
another\_func("typeof で関数ポインタも複製可能\n");  
  
/\* C11の\_Genericとの比較 \*/  
printf("\n--- C11の\_Genericとの比較 ---\n");  
printf("typeof: 式の型を取得（C23）\n");  
printf("\_Generic: 型に基づく選択（C11）\n");  
  
return 0;

}

/ 実行結果例: \* === C23 typeof演算子 === \* \* — 基本的な使用法 — \* x = 42, y = 100 \* yの型はxと同じ（int型） \* \* — 複雑な型での使用 — \* 配列の合計: 16.5 \* \* — ポインタ型での使用 — \* str = Hello, copy = World \* \* — 型安全なSWAPマクロ — \* 交換前: a = 10, b = 20 \* 交換後: a = 20, b = 10 \* \* 交換前: d1 = 3.14, d2 = 2.71 \* 交換後: d1 = 2.71, d2 = 3.14 \* \* — 型安全な最大値マクロ — \* MAX(15, 25) = 25 \* MAX(3.14, 2.71) = 3.14 \* \* — 配列サイズの計算 — \* int配列の要素数: 5 \* char配列の要素数: 6 \* 構造体配列の要素数: 3 \* \* — 関数ポインタでの使用 — \* typeof で関数ポインタも複製可能 \* \* — C11の\_Genericとの比較 — \* typeof: 式の型を取得（C23） \* \_Generic: 型に基づく選択（C11） \*/```