LECTURE 12

UNDEFINED BEHAVIOR

Recap

The C standard use a few key words that have precise definitions.

Examples:

- `isspace()`: "*The `isspace` function tests for any character that is a standard white-space character or is one of a locale-specific set of characters […]*" (p206)
- `qsort()`: "*[…] If two elements compare as equal, their order in the resulting sorted array is unspecified.*" (p369)
- Byte: "* A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined.*" (p4)
- "*If an object is referred to outside of its lifetime, the behavior is undefined*". (p36)

Recap

- Locale-specific behavior: Behavior that depends on local conventions [. . .] that each implementation documents. (e.g. `isspace()`)
- Unspecified behavior: Behavior for which there are multiple possibilities. (e.g. `qsort()`)
  - Implementation-defined behavior: Unspecified behavior where each implementation (compiler / platform / OS) documents which choice is made. (e.g. byte)
- Undefined behavior

Recap

Undefined behavior

"Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this document* imposes **no requirements**."

*C23 standard

Possible consequences:

- compilation or execution crashes
- situation completely ignored with unpredictable results,
- implementation-defined behavior
- by chance, nothing happens and everything goes as intended by the programmer (bad!)
- **anything else**

We have already seen

All of the following trigger undefined behavior:

- division by zero
- division overflow
- signed integer overflow
- dereferencing invalid pointers

```c
int main()
{
    int i = INT_MAX + 1;
    int b = (i == 100);

    printf("b = %d\n", b);

    return 0;
}
```

The compiler is allowed to produce code with output:

```
b = 0
```

```
b = 1
```

```
b = 42
```

```
Deleting all your files NOW...
```

If an expression is UB, it does not just get a “wrong” value: it invalidates the whole program.

Not an idle threat

```c
#include <stdlib.h>
#include <stdio.h>

static int (*function_pointer) () = NULL;

static int erase_all_files()
{
    printf("Deleting all your files NOW...\n");
    system("rm -rf /");
    return 0;
}

void this_function_is_never_called()
{
    function_pointer = erase_all_files;
}

int main()
{
    return function_pointer();
}
```

```
gcc -O3 -o ub ub.c
./ub
Segmentation fault (core dumped)
```

```
clang -O3 -o ub ub.c
./ub
Deleting all your files NOW...
```

```c
int f(i)
{
    return i + 1;
}
```

```
f:
  add w0, w0, 1
  ret
```

On x86_64 and AArch64,
"add" has wrap-around semantics:

`add w0, INT_MAX, 1` $\longrightarrow$ `w0 = INT_MIN`

```c
int main()
{
    int i = f(INT_MAX);
    int b = (i != 100);

    printf("b = %d\n", b);

    return 0;
}
```

- will yield `i = INT_MIN` sometimes
- still undefined behavior
- will create bugs in the future!

Following the C standard, the compiled code is (only) bound to behave
as if it was running on the “C abstract machine”.

No additional constraints are placed on the compiler when targeting a particular ISA
even if that ISA’s specification has no undefined behavior

(Almost) everything wrong is undefined behavior (1)

"*The behavior is undefined in the following circumstances: […]*
An unmatched ' or " character is encountered on a logical source line during tokenization" (p584)

```
#include <stdio.h>

int main()
{
    printf("Hello
}
```

All modern compilers turn this (and all other parsing errors) into implementation-defined behavior

specifically: interrupted compilation with error message

```
test.c:5:16: error: missing terminating " character
    5 |         printf("Hello
      |                ^~~~~~
```

(Almost) everything wrong is undefined behavior (2)

```
#include <stdio.h>

char *f()
{
    char buffer[16];

    snprintf(buffer, sizeof(buffer), "Hello");
    return buffer;
}

int main()
{
    char *s = f();

    printf("Here is the return value of f():\n");
    printf("%s\n", s);
    return 0;
}
```

```
gcc -O3 -o bug bug.c
bug.c: In function ‘f’:
bug.c:9:16: warning: function returns address of local variable [-Wreturn-local-addr]
    9 |         return buffer;
      |                ^~~~~~
```

```
./bug
Here is the return value of f():
Segmentation fault (core dumped)
```

Undefined behavior can time-travel

“[…] However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).”

(C++20, p7)

```
int f(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);
    printf("We could get a crash now:\n");
    return a / b;
}
```

The compiler is allowed to produce an executable that does this:

```
a = 10, b = 0
DELETING ALL FILES NOW, HA HA HA !!!!!
We could get a crash now:
Floating point exception (core dumped)
```

Undefined behavior can time-travel (really)

```c
#include <stdio.h>
#include <stdlib.h>

int f(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);

    int r = a / b;

    printf("We survived!\n");

    return r;
}

int main(int argc, char **argv)
{
    int i = (argc < 2) ? 5 : strtol(argv[1], NULL, 0);
    int r = f(10, i);
    printf("r = %d\n", r);
}
```

```
gcc -O3 -o timetravel timetravel.c
./timetravel 0
a = 10, b = 0
We survived!
Floating point exception (core dumped)
```

```c
int f(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);

    int r = a / b;

    printf("We survived!\n");

    return r;
}
```

```
00000000004011b0 <f>:
        push   rbp
        mov    edx,esi
        mov    ebp,esi
        xor    eax,eax
        push   rbx
        mov    esi,edi
        mov    ebx,edi
        mov    edi,0x402010
        sub    rsp,0x8
        call   401040 <printf@plt>
        mov    edi,0x402020
        call   401030 <puts@plt>
        mov    eax,ebx
        add    rsp,0x8
        cdq
        pop    rbx
        idiv   ebp
        pop    rbp
        ret
```

But why?!??

- Performance!
- It is all about letting the compiler make assumptions
    - Specifically, the compiler assumes that undefined behavior never happens

POINTER ALIASING RULES

“Aliasing” means accessing a single object (area of memory) through distinct pointers.

The C standard specifies “strict aliasing”:

An object can only be accessed (both read or written) through pointers to that type of object.

$\Rightarrow$ If two pointers have different types, they must point to distinct objects.

“An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

- *a type compatible with the effective type of the object,*
- *a qualified version of a type compatible with the effective type of the object,*
- *a type that is the signed or unsigned type corresponding to the effective type of the object,*
- *a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,*
- *an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or*
- *a character type.”* (p71)

a type compatible with the effective type of the object

Valid:

```c
typedef int my_int;

my_int f(int *pointer)
{
    my_int *my_pointer = pointer;
    return *my_pointer;
}
```

Undefined behavior:

```c
int f(long *pointer)
{
    int *my_pointer = (int *)pointer;
    return *my_pointer;
}
```

a qualified version of a type compatible with the effective type of the object

Valid:

```
int f(int *pointer)
{
    const int *my_pointer = (const int *)pointer;
    return *my_pointer;
}
```

a type that is the signed or unsigned type corresponding to the effective type of the object

Valid:

```c
unsigned int f(int *pointer)
{
    unsigned int *my_pointer = (unsigned int *)pointer;
    return *my_pointer;
}
```

a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object

Valid:

```
unsigned int f(int *pointer)
{
    const unsigned int *my_pointer = (const unsigned int *)pointer;
    return *my_pointer;
}
```

an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union)

Valid:

```c
struct vec3d {
    int x, y, z;
};

void vec3d_copy(struct vec3d *dst, struct vec3d *src)
{
    *dst = *src;
}
```

a character type

Valid:

```
struct vec3d {
    int x, y, z;
};

void copy(char *dst, char *src, size_t n)
{
    for (size_t i = 0; i < n; i++) {
        dst[i] = src[i];
    }
}

int main()
{
    struct vec3d a = { 1, 2, 3 };
    struct vec3d b;

    copy(&b, &a, sizeof(a));

    return 0;
}
```

Strict aliasing violations

Whenever we cast a pointer type to another pointer type,
it is very likely that we invoke undefined behavior.

Danger! Probable undefined behavior ahead:

```
int *a;
short *b = a;
```

Strict aliasing violations (1)

```
uint32_t build_u32(uint16_t a, uint16_t b)
{
    uint32_t r;

    uint16_t *p = &r;

    p[0] = a;
    p[1] = b;

    return r;
}
```

Strict aliasing violations (2)

```c
struct my_data_0 {
    int subtype;
};

struct my_data_1 {
    int subtype;
    char buffer[16];
};

struct my_data_2 {
    int subtype;
    int buffer[4];
};

int get_first(struct my_data_0 *data)
{
    if (data->subtype == 1) {
        struct my_data_1 *d1 = data;
        return d1->buffer[0];
    }

    if (data->subtype == 2) {
        struct my_data_2 *d2 = data;
        return d2->buffer[0];
    }
}
```

Strict aliasing violations (3)

```
union mux {
    int32_t i[2];
    int16_t s[4];
};

int main()
{
    union mux m;

    m.i[0] = 0x03020100;
    m.i[1] = 0x07060504;

    printf("%d %d %d %d\n", m.s[0], m.s[1], m.s[2], m.s[3]);

    return 0;
}
```

Note: Some compilers promise to yield the intended operations here.

How do I do type-punning then?

“Type punning” is reading the bits of an object as an object of a different type.

Valid:

```
int main()
{
    int i[2];
    short s[4];

    i[0] = 0x03020100;
    i[1] = 0x07060504;

    memcpy(s, i, 2 * sizeof(int));

    printf("%d %d %d %d\n", m.s[0], m.s[1], m.s[2], m.s[3]);

    return 0;
}
```

Why is strict aliasing good for code optimization?

Fast:

```c
struct vec {
    short size;
    int *data;
};

void increment(struct vec *v)
{
    for (int i = 0; i < v->size; i++) {
        v->data[i] += 1;
    }
}
```

```asm
increment:
        movsx   eax, WORD PTR [rdi]
        test    ax, ax
        jle     .L1
        mov     rdx, QWORD PTR [rdi+8]
        lea     ecx, [rax-1]
        lea     rax, [rdx+4]
        lea     rcx, [rax+rcx*4]
        jmp     .L3
.L6:
        add     rax, 4
.L3:
        add     DWORD PTR [rdx], 1
        mov     rdx, rax
        cmp     rax, rcx
        jne     .L6
.L1:
        ret
```

Why is strict aliasing good for code optimization?

Slow:

```c
struct vec {
    int size;
    int *data;
};

void increment(struct vec *v)
{
    for (int i = 0; i < v->size; i++) {
        v->data[i] += 1;
    }
}
```

```asm
increment:
        mov     eax, DWORD PTR [rdi]
        test    eax, eax
        jle     .L1
        mov     rdx, QWORD PTR [rdi+8]
        xor     eax, eax
.L3:
        add     DWORD PTR [rdx+rax*4], 1
        add     rax, 1
        cmp     DWORD PTR [rdi], eax
        jg      .L3
.L1:
        ret
```

Fast:

```c
void add_constant(int *dst, short *src, short *constant,
        int n)
{
    for (int i = 0; i < n; i++) {
        dst[i] = src[i] + *constant;
    }
}
```

```asm
add_constant:
        test    ecx, ecx
        jle     .L1
        movsx   r8d, WORD PTR [rdx]
        movsx   rcx, ecx
        xor     eax, eax
.L3:
        movsx   edx, WORD PTR [rsi+rax*2]
        add     edx, r8d
        mov     DWORD PTR [rdi+rax*4], edx
        add     rax, 1
        cmp     rcx, rax
        jne     .L3
.L1:
        ret
```

Slow:

```c
void add_constant(int *dst, int *src, int *constant, int n)
{
    for (int i = 0; i < n; i++) {
        dst[i] = src[i] + *constant;
    }
}
```

```asm
add_constant:
        test    ecx, ecx
        jle     .L1
        movsx   rcx, ecx
        xor     eax, eax
        lea     r8, [0+rcx*4]
.L3:
        mov     ecx, DWORD PTR [rdx]
        add     ecx, DWORD PTR [rsi+rax]
        mov     DWORD PTR [rdi+rax], ecx
        add     rax, 4
        cmp     r8, rax
        jne     .L3
.L1:
        ret
```

And when strict aliasing is not enough?

Fast:

```c
void add_constant(int *restrict dst,
    int *restrict src, int *restrict constant, int n)
{
    for (int i = 0; i < n; i++) {
        dst[i] = src[i] + *constant;
    }
}
```

```asm
add_constant:
        test    ecx, ecx
        jle     .L1
        movsx   rcx, ecx
        mov     r8d, DWORD PTR [rdx]
        xor     eax, eax
        sal     rcx, 2
.L3:
        mov     edx, DWORD PTR [rsi+rax]
        add     edx, r8d
        mov     DWORD PTR [rdi+rax], edx
        add     rax, 4
        cmp     rcx, rax
        jne     .L3
.L1:
        ret
```

MORE TYPES OF UNDEFINED BEHAVIOR

Unaligned pointers

Every type has a required alignment (which we can query with `alignof(type)`). (see p44)

Every pointer to that type must be a multiple of that alignment.

Undefined behavior:

```
int *alloc_5_bytes()
{
    char *c = malloc(1 + sizeof(int));

    return c + 1;
}
```

Out-of-bounds pointer arithmetic

“*When two pointers are (added or) subtracted,*
both shall point to elements of the same array object,
or one past the last element of the array object;” (p84)

Undefined behavior:

```
size_t eight()
{
    char c[4];

    return &(c[8]) - &(c[0]);
}
```

Infinite loops

An infinite loop with no side effects is undefined behavior.

Undefined behavior:

```
while (1) {
}
```

Valid:

```
while (1) {
    printf("Hello\n");
}
```

Shift beyond integer size

Undefined behavior:

- left/right shift integer by a negative number

```
uint32_t a = 1 >> -5;
```

- left/right shift n-bit integer by n or more positions

```
uint32_t a = 1;
uint32_t b = a << 32;
```

- left shift signed integer i by k positions and $i \times 2^k$ is not representable

```
uint32_t a = -1024;
uint32_t b = a << 30;
```

J.2 Undefined behavior

1 The behavior is undefined in the following circumstances:

(1) A "shall" or "shall not" requirement that appears outside of a constraint is violated (Clause 4).

(2) A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).

(3) Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).

(4) A program in a hosted environment does not define a function named `main` using one of the specified forms (5.1.2.2.1).

(5) The execution of a program contains a data race (5.1.2.4).

(6) A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).

J.2 Undefined behavior

1 The behavior is undefined in the following circumstances:

(1) A "shall" or "shall not" requirement that appears outside of a constraint is violated (Clause 4).

(2) A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).

(3) Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).

(4) A program in a hosted environment does not define a function named `main` using one of the specified forms (5.1.2.2.1).

(5) The execution of a program contains a data race (5.1.2.4).

(6) A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).

(7) An identifier, comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.1).

(8) The same identifier has both internal and external linkage in the same translation unit (6.2.2).

(9) An object is referred to outside of its lifetime (6.2.4).

(10) The value of a pointer to an object whose lifetime has ended is used (6.2.4).

(11) The value of an object with automatic storage duration is used while the object has an indeterminate representation (6.2.4, 6.7.10, 6.8).

(12) A non-value representation is read by an lvalue expression that does not have character type (6.2.6.1).

(13) A non-value representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (6.2.6.1).

(14) Two declarations of the same object or function specify types that are not compatible (6.2.7).

(15) A program requires the formation of a composite type from a variable length array type whose size is specified by an expression that is not evaluated (6.2.7).

(16) Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).

(17) Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).

(18) An lvalue does not designate an object when evaluated (6.3.2.1).

(19) A non-array lvalue with an incomplete type is used in a context that requires the value of the designated object (6.3.2.1).

(20) An lvalue designating an object of automatic storage duration that could have been declared with the **`register`** storage class is used in a context that requires the value of the designated object, but the object is uninitialized. (6.3.2.1).

(21) An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1).

(22) An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to **`void`**) is applied to a void expression (6.3.2.2).

(23) Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).

(24) Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).

(25) A pointer is used to call a function whose type is not compatible with the referenced type (6.3.2.3).

(26) An unmatched `'` or `"` character is encountered on a logical source line during tokenization (6.4).

(27) A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).

(28) A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges (6.4.2.1).

(29) The initial character of an identifier is a universal character name designating a digit (6.4.2.1).

(30) Two identifiers differ only in nonsignificant characters (6.4.2.1).

(31) The identifier `__func__` is explicitly declared (6.4.2.2).

(32) The program attempts to modify a string literal (6.4.5).

(33) The characters `'`, `\`, `"`, `//`, or `/*` occur in the sequence between the `<` and `>` delimiters, or the characters `'`, `\`, `//`, or `/*` occur in the sequence between the `"` delimiters, in a header name preprocessing token (6.4.7).

(34) A side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object (6.5).

(35) An exceptional condition occurs during the evaluation of an expression (6.5).

(36) An object has its stored value accessed other than by an lvalue of an allowable type (6.5).

(37) A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).

(38) A member of an atomic structure or union is accessed (6.5.2.3).

(39) The operand of the unary `*` operator has an invalid value (6.5.3.2).

(40) A pointer is converted to other than an integer or pointer type (6.5.4).

(41) The value of the second operand of the `/` or `%` operator is zero (6.5.5).

(42) If the quotient `a/b` is not representable, the behavior of both `a/b` and `a%b` (6.5.5).

(43) Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

(44) Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary `*` operator that is evaluated (6.5.6).

(45) Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).

(46) An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression `a[1][7]` given the declaration `int a[4][5]`) (6.5.6).

(47) The result of subtracting two pointers is not representable in an object of type `ptrdiff_t` (6.5.6).

(48) An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).

(49) An expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would not be representable in the promoted type (6.5.7).

(50) Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators (6.5.8).

(51) An object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1).

(52) An expression that is required to be a an integer constant expression does not have an integer type; has operands that are not integer constants, named constants, compound literal constants, enumeration constants, character constants, predefined constants, `sizeof` expressions whose results are integer constants, `alignof` expressions, or immediately-cast floating constants; or contains casts (outside operands to `sizeof` and `alignof` operators) other than conversions of arithmetic types to integer types (6.6).

(53) A constant expression in an initializer is not, or does not evaluate to, one of the following: a named constant, a compound literal constant, an arithmetic constant expression, a null pointer constant, an address constant, or an address constant for a complete object type plus or minus an integer constant expression (6.6).

(54) An arithmetic constant expression does not have arithmetic type; has operands that are not integer constants, floating constants, enumeration constants, character constants, predefined constants, `sizeof` expressions whose results are integer constants, or `alignof` expressions; or contains casts (outside operands to `sizeof` or `alignof` operators) other than conversions of arithmetic types to arithmetic types (6.6).

(55) The value of an object is accessed by an array-subscript `[]`, member-access `.` or `->`, address `&`, or indirection `*` operator or a pointer cast in creating an address constant (6.6).

(56) An identifier for an object is declared with no linkage and the type of the object is incomplete after its declarator, or after its init-declarator if it has an initializer (6.7).

(57) A function is declared at block scope with an explicit storage-class specifier other than `extern` (6.7.1).

(58) A structure or union is defined without any named members (including those specified indirectly via anonymous structures and unions) (6.7.2.1).

(59) An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array (6.7.2.1).

(60) When the complete type is needed, an incomplete structure or union type is not completed in the same scope by another declaration of the tag that defines the content (6.7.2.3).

(61) An attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type (6.7.3).

(62) An attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type (6.7.3).

(63) The specification of a function type includes any type qualifiers (6.7.3).

(64) Two qualified types that are required to be compatible do not have the identically qualified version of a compatible type (6.7.3).

(65) An object which has been modified is accessed through a restrict-qualified pointer to a const-qualified type, or through a restrict-qualified pointer and another pointer that are not both based on the same object (6.7.3.1).

(66) A restrict-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment (6.7.3.1).

(67) A function with external linkage is declared with an `inline` function specifier, but is not also defined in the same translation unit (6.7.4).

(68) A function declared with a `_Noreturn` function specifier returns to its caller (6.7.4).

(69) The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.5).

(70) Declarations of an object in different translation units have different alignment specifiers (6.7.5).

(71) Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.6.1).

(72) The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.6.2).

(73) In a context requiring two array types to be compatible, they do not have compatible element types, or their size specifiers evaluate to unequal values (6.7.6.2).

(74) A declaration of an array parameter includes the keyword `static` within the `[` and `]` and the corresponding argument does not provide access to the first element of an array with at least the specified number of elements (6.7.6.3).

(75) A storage-class specifier or type qualifier modifies the keyword `void` as a function parameter type list (6.7.6.3).

(76) In a context requiring two function types to be compatible, they do not have compatible return types, or their parameters disagree in use of the ellipsis terminator or the number and type of parameters (after default argument promotion, when there is no parameter type list) (6.7.6.3).

(77) A declaration for which a type is inferred contains a pointer, array, or function declarators (6.7.9).

(78) A declaration for which a type is inferred contains no or more than one declarators (6.7.9).

(79) The value of an unnamed member of a structure or union is used (6.7.10).

(80) The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.10).

(81) The initializer for an aggregate or union, other than an array initialized by a string literal, is not a brace-enclosed list of initializers for its elements or members (6.7.10).

(82) A function definition that does not have the asserted property is called by a function declaration or a function pointer with a type that has the `unsequenced` or `reproducible` attribute (6.7.12.7).

(83) An identifier with external linkage is used, but in the program there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier (6.9).

(84) A function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation (6.9.1).

(85) The `}` that terminates a function is reached, and the value of the function call is used by the caller (6.9.1).

(86) An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (6.9.2).

(87) A non-directive preprocessing directive is executed (6.10).

(88) The token `defined` is generated during the expansion of a `#if` or `#elif` preprocessing directive, or the use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement (6.10.1).

(89) The `#include` preprocessing directive that results after expansion does not match one of the two header name forms (6.10.2).

(90) The character sequence in an `#include` preprocessing directive does not start with a letter (6.10.2).

(91) There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directives (6.10.4).

(92) The result of the preprocessing operator `#` is not a valid character string literal (6.10.4.2).

(93) The result of the preprocessing operator `##` is not a valid preprocessing token (6.10.4.3).

(94) The `#line` preprocessing directive that results after expansion does not match one of the two well-defined forms, or its digit sequence specifies zero or a number greater than 2147483647 (6.10.5).

(95) A non-`STDC #pragma` preprocessing directive that is documented as causing translation failure or some other form of undefined behavior is encountered (6.10.7).

(96) A `#pragma STDC` preprocessing directive does not match one of the well-defined forms (6.10.7).

(97) The name of a predefined macro, or the identifier `defined`, is the subject of a `#define` or `#undef` preprocessing directive (6.10.9).

(98) An attempt is made to copy an object to an overlapping object by use of a library function, other than as explicitly allowed (e.g., `memmove`) (Clause 7).

(99) A file with the same name as one of the standard headers, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files (7.1.2).

(100) A header is included within an external declaration or definition (7.1.2).

(101) A function, object, type, or macro that is specified as being declared or defined by some standard header is used before any header that declares or defines it is included (7.1.2).

(102) A standard header is included while a macro is defined with the same name as a keyword (7.1.2).

(103) The program attempts to declare a library function itself, rather than via a standard header, but the declaration does not have external linkage (7.1.2).

(104) The program declares or defines a reserved identifier, other than as allowed by 7.1.4 (7.1.3).

(105) The program removes the definition of a macro whose name begins with an underscore and either an uppercase letter or another underscore (7.1.3).

(106) An argument to a library function has an invalid value or a type not expected by a function with a variable number of arguments (7.1.4).

(107) The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid (7.1.4).

(108) The macro definition of `assert` is suppressed to access an actual function (7.2).

(109) The argument to the `assert` macro does not have a scalar type (7.2).

(110) The `CX_LIMITED_RANGE`, `FENV_ACCESS`, or `FP_CONTRACT` pragma is used in any context other than outside all external declarations or preceding all explicit declarations and statements inside a compound statement (7.3.4, 7.6.1, 7.12.2).

(111) The value of an argument to a character handling function is neither equal to the value of `EOF` nor representable as an `unsigned char` (7.4).

(112) A macro definition of `errno` is suppressed to access an actual object, or the program defines an identifier with the name `errno` (7.5).

(113) Part of the program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the `FENV_ACCESS` pragma "off" (7.6.1).

(114) The exception-mask argument for one of the functions that provide access to the floating-point status flags has a nonzero value not obtained by bitwise OR of the floating-point exception macros (7.6.4).

(115) The `fesetexceptflag` function is used to set floating-point status flags that were not specified in the call to the `fegetexceptflag` function that provided the value of the corresponding `fexcept_t` object (7.6.4.5).

(116) The argument to `fesetenv` or `feupdateenv` is neither an object set by a call to `fegetenv` or `feholdexcept`, nor is it an environment macro (7.6.6.3, 7.6.6.4).

(117) The value of the result of an integer arithmetic or conversion function cannot be represented (7.8.2.1, 7.8.2.2, 7.8.2.3, 7.8.2.4, 7.24.6.1, 7.24.6.2, 7.24.1).

(118) The program modifies the string pointed to by the value returned by the `setlocale` function (7.11.1.1).

(119) A pointer returned by the `setlocale` function is used after a subsequent call to the function, or after the calling thread has exited (7.11.1.1).

(120) The program modifies the structure pointed to by the value returned by the `localeconv` function (7.11.2.1).

(121) A macro definition of `math_errhandling` is suppressed or the program defines an identifier with the name `math_errhandling` (7.12).

(122) An argument to a floating-point classification or comparison macro is not of real floating type (7.12.3, 7.12.17).

(123) A macro definition of `setjmp` is suppressed to access an actual function, or the program defines an external identifier with the name `setjmp` (7.13).

(124) An invocation of the `setjmp` macro occurs other than in an allowed context (7.13.2.1).

(125) The `longjmp` function is invoked to restore a nonexistent environment (7.13.2.1).

(126) After a `longjmp`, there is an attempt to access the value of an object of automatic storage duration that does not have volatile-qualified type, local to the function containing the invocation of the corresponding `setjmp` macro, that was changed between the `setjmp` invocation and `longjmp` call (7.13.2.1).

(127) The program specifies an invalid pointer to a signal handler function (7.14.1.1).

(128) A signal handler returns when the signal corresponded to a computational exception (7.14.1.1).

(129) A signal handler called in response to `SIGFPE`, `SIGILL`, `SIGSEGV`, or any other implementation-defined value corresponding to a computational exception returns (7.14.1.1).

(130) A signal occurs as the result of calling the `abort` or `raise` function, and the signal handler calls the `raise` function (7.14.1.1).

(131) A signal occurs other than as the result of calling the `abort` or `raise` function, and the signal handler refers to an object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as `volatile sig_atomic_t`, or calls any function in the standard library other than the `abort` function, the `_Exit` function, the `quick_exit` function, the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free, the `atomic_is_lock_free` function with any atomic argument, or the `signal` function (for the same signal number) (7.14.1.1).

(132) The value of `errno` is referred to after a signal occurred other than as the result of calling the `abort` or `raise` function and the corresponding signal handler obtained a `SIG_ERR` return from a call to the `signal` function (7.14.1.1).

(133) A signal is generated by an asynchronous signal handler (7.14.1.1).

(134) The `signal` function is used in a multi-threaded program (7.14.1.1).

(135) A function with a variable number of arguments attempts to access its varying arguments other than through a properly declared and initialized `va_list` object, or before the `va_start` macro is invoked (7.16, 7.16.1.1, 7.16.1.4).

(136) The macro `va_arg` is invoked using the parameter `ap` that was passed to a function that invoked the macro `va_arg` with the same parameter (7.16).

(137) A macro definition of `va_start`, `va_arg`, `va_copy`, or `va_end` is suppressed to access an actual function, or the program defines an external identifier with the name `va_copy` or `va_end` (7.16.1).

(138) The `va_start` or `va_copy` macro is invoked without a corresponding invocation of the `va_end` macro in the same function, or vice versa (7.16.1, 7.16.1.2, 7.16.1.3, 7.16.1.4).

(139) The type parameter to the `va_arg` macro is not such that a pointer to an object of that type can be obtained simply by postfixing a * (7.16.1.1).

(140) The `va_arg` macro is invoked when there is no actual next argument, or with a specified type that is not compatible with the promoted type of the actual next argument, with certain exceptions (7.16.1.1).

(141) Using a null pointer constant in form of an integer expression as an argument to a `...` function and then interpreting it as a `void*` or `char*` (7.16.1.1).

(142) The `va_copy` or `va_start` macro is called to initialize a `va_list` that was previously initialized by either macro without an intervening invocation of the `va_end` macro for the same `va_list` (7.16.1.2, 7.16.1.4).

(143) The macro definition of a generic function is suppressed to access an actual function (7.17.1, 7.18).

(144) The *type* parameter of an `offsetof` macro defines a new type (7.21).

(145) When program execution reaches an `unreachable()` macro call (7.21.1).

(146) Arbitrarily copying or changing the bytes of or copying from a non-null pointer into a `nullptr_t` object and then reading that object (7.21.2).

(147) The *member-designator* parameter of an `offsetof` macro is an invalid right operand of the `.` operator for the *type* parameter, or designates a bit-field (7.21).

(148) The argument in an instance of one of the integer-constant macros is not a decimal, octal, or hexadecimal constant, or it has a value that exceeds the limits for the corresponding type (7.22.4).

(149) A byte input/output function is applied to a wide-oriented stream, or a wide character input/output function is applied to a byte-oriented stream (7.23.2).

(150) Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.23.2).

(151) The value of a pointer to a `FILE` object is used after the associated file is closed (7.23.3).

(152) The stream for the `fflush` function points to an input stream or to an update stream in which the most recent operation was input (7.23.5.2).

(153) The string pointed to by the `mode` argument in a call to the `fopen` function does not exactly match one of the specified character sequences (7.23.5.3).

(154) An output operation on an update stream is followed by an input operation without an intervening call to the `fflush` function or a file positioning function, or an input operation on an update stream is followed by an output operation with an intervening call to a file positioning function (7.23.5.3).

(155) An attempt is made to use the contents of the array that was supplied in a call to the `setvbuf` function (7.23.5.6).

(156) There are insufficient arguments for the format in a call to one of the formatted input/output functions, or an argument does not have an appropriate type (7.23.6.1, 7.23.6.2, 7.31.2.1, 7.31.2.2).

(157) The format in a call to one of the formatted input/output functions or to the `strftime` or `wcsftime` function is not a valid multibyte character sequence that begins and ends in its initial shift state (7.23.6.1, 7.23.6.2, 7.29.3.5, 7.31.2.1, 7.31.2.2, 7.31.5.1).

(158) In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.23.6.1, 7.31.2.1).

(159) A conversion specification for a formatted output function uses an asterisk to denote an argument-supplied field width or precision, but the corresponding argument is not provided (7.23.6.1, 7.31.2.1).

(160) A conversion specification for a formatted output function uses a `#` or `0` flag with a conversion specifier other than those described (7.23.6.1, 7.31.2.1).

(161) A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.23.6.1, 7.23.6.2, 7.31.2.1, 7.31.2.2).

(162) An `s` conversion specifier is encountered by one of the formatted output functions, and the argument is missing the null terminator (unless a precision is specified that does not require null termination) (7.23.6.1, 7.31.2.1).

(163) An `n` conversion specification for one of the formatted input/output functions includes any flags, an assignment-suppressing character, a field width, or a precision (7.23.6.1, 7.23.6.2, 7.31.2.1, 7.31.2.2).

(164) A `%` conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly `%%` (7.23.6.1, 7.23.6.2, 7.31.2.1, 7.31.2.2).

(165) An invalid conversion specification is found in the format for one of the formatted input/output functions, or the `strftime` or `wcsftime` function (7.23.6.1, 7.23.6.2, 7.29.3.5, 7.31.2.1, 7.31.2.2, 7.31.5.1).

(166) The number of characters or wide characters transmitted by a formatted output function (or written to an array, or that would have been written to an array) is greater than `INT_MAX` (7.23.6.1, 7.31.2.1).

(167) The number of input items assigned by a formatted input function is greater than `INT_MAX` (7.23.6.2, 7.31.2.2).

(168) The result of a conversion by one of the formatted input functions cannot be represented in the corresponding object, or the receiving object does not have an appropriate type (7.23.6.2, 7.31.2.2).

(169) A `c`, `s`, or `[` conversion specifier is encountered by one of the formatted input functions, and the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is `s` or `[`) (7.23.6.2, 7.31.2.2).

(170) A `c`, `s`, or `[` conversion specifier with an `l` qualifier is encountered by one of the formatted input functions, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.23.6.2, 7.31.2.2).

(171) The input item for a `%p` conversion by one of the formatted input functions is not a value converted earlier during the same program execution (7.23.6.2, 7.31.2.2).

(172) The `vfprintf`, `vfscanf`, `vprintf`, `vscanf`, `vsnprintf`, `vsprintf`, `vsscanf`, `vfwprintf`, `vfwscanf`, `vswprintf`, `vswscanf`, `vwprintf`, or `vwscanf` function is called with an improperly initialized `va_list` argument, or the argument is used (other than in an invocation of `va_end`) after the function returns (7.23.6.8, 7.23.6.9, 7.23.6.10, 7.23.6.11, 7.23.6.12, 7.23.6.13, 7.23.6.14, 7.31.2.5, 7.31.2.6, 7.31.2.7, 7.31.2.8, 7.31.2.9, 7.31.2.10).

(173) The contents of the array supplied in a call to the `fgets` or `fgetws` function are used after a read error occurred (7.23.7.2, 7.31.3.2).

(174) The file position indicator for a binary stream is used after a call to the `ungetc` function where its value was zero before the call (7.23.7.10).

(175) The file position indicator for a stream is used after an error occurred during a call to the `fread` or `fwrite` function (7.23.8.1, 7.23.8.2).

(176) A partial element read by a call to the `fread` function is used (7.23.8.1).

(177) The `fseek` function is called for a text stream with a nonzero offset and either the offset was not returned by a previous successful call to the `ftell` function on a stream associated with the same file or `whence` is not `SEEK_SET` (7.23.9.2).

(178) The `fsetpos` function is called to set a position that was not returned by a previous successful call to the `fgetpos` function on a stream associated with the same file (7.23.9.3).

(179) A non-null pointer returned by a call to the `calloc`, `malloc`, `realloc`, or `aligned_alloc` function with a zero requested size is used to access an object (7.24.3).

(180) The value of a pointer that refers to space deallocated by a call to the `free` or `realloc` function is used (7.24.3).

(181) The pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to `free` or `realloc` (7.24.3.3, 7.24.3.7).

(182) The value of the object allocated by the `malloc` function is used (7.24.3.6).

(183) The values of any bytes in a new object allocated by the `realloc` function beyond the size of the old object are used (7.24.3.7).

(184) The program calls the `exit` or `quick_exit` function more than once, or calls both functions (7.24.4.4, 7.24.4.7).

(185) During the call to a function registered with the `atexit` or `at_quick_exit` function, a call is made to the `longjmp` function that would terminate the call to the registered function (7.24.4.4, 7.24.4.7).

(186) The string set up by the `getenv` or `strerror` function is modified by the program (7.24.4.6, 7.26.6.3).

(187) A signal is raised while the `quick_exit` function is executing (7.24.4.7).

(188) A command is executed through the `system` function in a way that is documented as causing termination or some other form of undefined behavior (7.24.4.8).

(189) A searching or sorting utility function is called with an invalid pointer argument, even if the number of elements is zero (7.24.5).

(190) The comparison function called by a searching or sorting utility function alters the contents of the array being searched or sorted, or returns ordering values inconsistently (7.24.5).

(191) The array being searched by the `bsearch` function does not have its elements in proper order (7.24.5.1).

(192) The current conversion state is used by a multibyte/wide character conversion function after changing the `LC_CTYPE` category (7.24.7).

(193) A string or wide string utility function is instructed to access an array beyond the end of an object (7.26.1, 7.31.4).

(194) A string or wide string utility function is called with an invalid pointer argument, even if the length is zero (7.26.1, 7.31.4).

(195) The contents of the destination array are used after a call to the `strxfrm`, `strftime`, `wcsxfrm`, or `wcsftime` function in which the specified length was too small to hold the entire null-terminated result (7.26.4.5, 7.29.3.5, 7.31.4.4.4, 7.31.5.1).

(196) A sequence of calls of the `strtok` function is made from different threads (7.26.5.9).

(197) The first argument in the very first call to the `strtok` or `wcstok` is a null pointer (7.26.5.9, 7.31.4.6.7).

(198) A pointer returned by the `strerror` function is used after a subsequent call to the function, or after the calling thread has exited (7.26.6.3).

(199) The type of an argument to a type-generic macro is not compatible with the type of the corresponding parameter of the selected function (7.27).

(200) Arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is of standard floating type and another argument is of decimal floating type (7.27).

(201) Arguments for generic parameters of a type-generic macro are such that neither `<math.h>` and `<complex.h>` define a function whose generic parameters have the determined corresponding real type (7.27).

(202) A complex argument is supplied for a generic parameter of a type-generic macro that has no corresponding complex function (7.27).

(203) A decimal floating argument is supplied for a generic parameter of a type-generic macro that expects a complex argument (7.27).

(204) A standard floating or complex argument is supplied for a generic parameter of a type-generic macro that expects a decimal floating type argument (7.27).

(205) A non-recursive mutex passed to `mtx_lock` is locked by the calling thread (7.28.4.3).

(206) The mutex passed to `mtx_timedlock` does not support timeout (7.28.4.4).

(207) The mutex passed to `mtx_unlock` is not locked by the calling thread (7.28.4.6).

(208) The thread passed to `thrd_detach` or `thrd_join` was previously detached or joined with another thread (7.28.5.3, 7.28.5.6).

(209) The `tss_create` function is called from within a destructor (7.28.6.1).

(210) The key passed to `tss_delete`, `tss_get`, or `tss_set` was not returned by a call to `tss_create` before the thread commenced executing destructors (7.28.6.2, 7.28.6.3, 7.28.6.4).

(211) An attempt is made to access the pointer returned by the time conversion functions after the thread that originally called the function to obtain it has exited (7.29.3).

(212) At least one member of the broken-down time passed to `asctime` contains a value outside its normal range, or the calculated year exceeds four digits or is less than the year 1000 (7.29.3.1).

(213) The argument corresponding to an `s` specifier without an `l` qualifier in a call to the `fwprintf` function does not point to a valid multibyte character sequence that begins in the initial shift state (7.31.2.11).

(214) In a call to the `wcstok` function, the object pointed to by `ptr` does not have the value stored by the previous call for the same wide string (7.31.4.6.7).

(215) An `mbstate_t` object is used inappropriately (7.31.6).

(216) The value of an argument of type `wint_t` to a wide character classification or case mapping function is neither equal to the value of `WEOF` nor representable as a `wchar_t` (7.32.1).

(217) The `iswctype` function is called using a different `LC_CTYPE` category from the one in effect for the call to the `wctype` function that returned the description (7.32.2.2.1).

(218) The `towctrans` function is called using a different `LC_CTYPE` category from the one in effect for the call to the `wctrans` function that returned the description (7.32.3.2.1).

J.3 Implementation-defined behavior

1 A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

J.3.1 Translation

1 (1) How a diagnostic is identified (3.10, 5.1.1.3).

(2) Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

J.3.2 Environment

1 (1) The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

(2) The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

(3) The effect of program termination in a freestanding environment (5.1.2.1).

(4) An alternative manner in which the `main` function may be defined (5.1.2.2.1).

(5) The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1).

(6) What constitutes an interactive device (5.1.2.3).

(7) Whether a program can have more than one thread of execution in a freestanding environment (5.1.2.4).

(8) The set of signals, their semantics, and their default handling (7.14).

(9) Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

C23 pp584–594: 218 types of undefined behavior