Chapter 28: Undefined behavior

In C, some expressions yield *undefined behavior*. The standard explicitly chooses to not define how a compiler should behave if it encounters such an expression. As a result, a compiler is free to do whatever it sees fit and may produce useful results, unexpected results, or even crash.

Code that invokes UB may work as intended on a specific system with a specific compiler, but will likely not work on another system, or with a different compiler, compiler version or compiler settings.

Section 28.1: Dereferencing a pointer to variable beyond its lifetime

```c
int* foo(int bar)
{
    int baz = 6;
    baz += bar;
    return &baz; /* (&baz) copied to new memory location outside of foo. */
} /* (1) The lifetime of baz and bar end here as they have automatic storage
   * duration (local variables), thus the returned pointer is not valid! */

int main (void)
{
    int* p;

    p = foo(5);  /* (2) this expression's behavior is undefined */
    *p = *p - 6; /* (3) Undefined behaviour here */

    return 0;
}
```

Some compilers helpfully point this out. For example, `gcc` warns with:

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

and `clang` warns with:

```
warning: address of stack memory associated with local variable 'baz' returned
[-Wreturn-stack-address]
```

for the above code. But compilers may not be able to help in complex code.

(1) Returning reference to variable declared `static` is defined behaviour, as the variable is not destroyed after leaving current scope.

(2) According to ISO/IEC 9899:2011 6.2.4 §2, "The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime."

(3) Dereferencing the pointer returned by the function `foo` is undefined behaviour as the memory it references holds an indeterminate value.

Section 28.2: Copying overlapping memory

A wide variety of standard library functions have among their effects copying byte sequences from one memory region to another. Most of these functions have undefined behavior when the source and destination regions overlap.

For example, this ...

```
#include <string.h> /* for memcpy() */

char str[19] = "This is an example";
memcpy(str + 7, str, 10);
```

... attempts to copy 10 bytes where the source and destination memory areas overlap by three bytes. To visualize:

```
               overlapping area
                |
               _ _
              |   |
              v   v
T h i s   i s   a n   e x a m p l e \0
^             ^
|             |
|             destination
|
source
```

Because of the overlap, the resulting behavior is undefined.

Among the standard library functions with a limitation of this kind are `memcpy()`, `strcpy()`, `strcat()`, `sprintf()`, and `sscanf()`. The standard says of these and several other functions:

> If copying takes place between objects that overlap, the behavior is undefined.

The `memmove()` function is the principal exception to this rule. Its definition specifies that the function behaves as if the source data were first copied into a temporary buffer and then written to the destination address. There is no exception for overlapping source and destination regions, nor any need for one, so `memmove()` has well-defined behavior in such cases.

The distinction reflects an efficiency *vs*. generality tradeoff. Copying such as these functions perform usually occurs between disjoint regions of memory, and often it is possible to know at development time whether a particular instance of memory copying will be in that category. Assuming non-overlap affords comparatively more efficient implementations that do not reliably produce correct results when the assumption does not hold. Most C library functions are allowed the more efficient implementations, and `memmove()` fills in the gaps, serving the cases where the source and destination may or do overlap. To produce the correct effect in all cases, however, it must perform additional tests and / or employ a comparatively less efficient implementation.

Section 28.3: Signed integer overflow

Per paragraph 6.5/5 of both C99 and C11, evaluation of an expression produces undefined behavior if the result is not a representable value of the expression's type. For arithmetic types, that's called an *overflow*. Unsigned integer arithmetic does not overflow because paragraph 6.2.5/9 applies, causing any unsigned result that otherwise would be out of range to be reduced to an in-range value. There is no analogous provision for *signed* integer types, however; these can and do overflow, producing undefined behavior. For example,

```
#include <limits.h>      /* to get INT_MAX */
```

```c
int main(void) {
    int i = INT_MAX + 1; /* Overflow happens here */
    return 0;
}
```

Most instances of this type of undefined behavior are more difficult to recognize or predict. Overflow can in principle arise from any addition, subtraction, or multiplication operation on signed integers (subject to the usual arithmetic conversions) where there are not effective bounds on or a relationship between the operands to prevent it. For example, this function:

```c
int square(int x) {
    return x * x;  /* overflows for some values of x */
}
```

is reasonable, and it does the right thing for small enough argument values, but its behavior is undefined for larger argument values. You cannot judge from the function alone whether programs that call it exhibit undefined behavior as a result. It depends on what arguments they pass to it.

On the other hand, consider this trivial example of overflow-safe signed integer arithmetic:

```c
int zero(int x) {
    return x - x;  /* Cannot overflow */
}
```

The relationship between the operands of the subtraction operator ensures that the subtraction never overflows. Or consider this somewhat more practical example:

```c
int sizeDelta(FILE *f1, FILE *f2) {
    int count1 = 0;
    int count2 = 0;
    while (fgetc(f1) != EOF) count1++;  /* might overflow */
    while (fgetc(f2) != EOF) count2++;  /* might overflow */

    return count1 - count2; /* provided no UB to this point, will not overflow */
}
```

As long as that the counters do not overflow individually, the operands of the final subtraction will both be non-negative. All differences between any two such values are representable as `int`.

Section 28.4: Use of an uninitialized variable

```c
int a;
printf("%d", a);
```

The variable a is an `int` with automatic storage duration. The example code above is trying to print the value of an uninitialized variable (a was never initialized). Automatic variables which are not initialized have indeterminate values; accessing these can lead to undefined behavior.

Note: Variables with static or thread local storage, including global variables without the `static` keyword, are initialized to either zero, or their initialized value. Hence the following is legal.

```c
static int b;
printf("%d", b);
```

A very common mistake is to not initialize the variables that serve as counters to 0. You add values to them, but since the initial value is garbage, you will invoke **Undefined Behavior**, such as in the question Compilation on terminal gives off pointer warning and strange symbols.

Example:

```
#include <stdio.h>

int main(void) {
    int i, counter;
    for(i = 0; i < 10; ++i)
        counter += i;
    printf("%d\n", counter);
    return 0;
}
```

Output:

```
C02QT2UBFVH6-lm:~ gsamaras$ gcc main.c -Wall -o main
main.c:6:9: warning: variable 'counter' is uninitialized when used here [-Wuninitialized]
        counter += i;
        ^~~~~~~
main.c:4:19: note: initialize the variable 'counter' to silence this warning
    int i, counter;
                  ^
                   = 0
1 warning generated.
C02QT2UBFVH6-lm:~ gsamaras$ ./main
32812
```

The above rules are applicable for pointers as well. For example, the following results in undefined behavior

```
int main(void)
{
    int *p;
    p++; // Trying to increment an uninitialized pointer.
}
```

Note that the above code on its own might not cause an error or segmentation fault, but trying to dereference this pointer later would cause the undefined behavior.

Section 28.5: Data race

Version ≥ C11

C11 introduced support for multiple threads of execution, which affords the possibility of data races. A program contains a data race if an object in it is accessed1 by two different threads, where at least one of the accesses is non-atomic, at least one modifies the object, and program semantics fail to ensure that the two accesses cannot overlap temporally.2 Note well that actual concurrency of the accesses involved is not a condition for a data race; data races cover a broader class of issues arising from (allowed) inconsistencies in different threads' views of memory.

Consider this example:

```
#include <threads.h>

int a = 0;

```

```
int Function( void* ignore )
{
    a = 1;

    return 0;
}

int main( void )
{
    thrd_t id;
    thrd_create( &id , Function , NULL );

    int b = a;

    thrd_join( id , NULL );
}
```

The main thread calls `thrd_create` to start a new thread running function `Function`. The second thread modifies `a`, and the main thread reads `a`. Neither of those access is atomic, and the two threads do nothing either individually or jointly to ensure that they do not overlap, so there is a data race.

Among the ways this program could avoid the data race are

- the main thread could perform its read of `a` before starting the other thread;
- the main thread could perform its read of `a` after ensuring via `thrd_join` that the other has terminated;
- the threads could synchronize their accesses via a mutex, each one locking that mutex before accessing `a` and unlocking it afterward.

As the mutex option demonstrates, avoiding a data race does not require ensuring a specific order of operations, such as the child thread modifying `a` before the main thread reads it; it is sufficient (for avoiding a data race) to ensure that for a given execution, one access will happen before the other.

1 Modifying or reading an object.

2 (Quoted from ISO:IEC 9889:201x, section 5.1.2.4 "Multi-threaded executions and data races")
The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

Section 28.6: Read value of pointer that was freed

Even just **reading** the value of a pointer that was freed (i.e. without trying to dereference the pointer) is undefined behavior(UB), e.g.

```
char *p = malloc(5);
free(p);
if (p == NULL) /* NOTE: even without dereferencing, this may have UB */
{

}
```

Quoting **ISO/IEC 9899:2011**, section 6.2.4 §2:

> […] The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

The use of indeterminate memory for anything, including apparently harmless comparison or arithmetic, can have undefined behavior if the value can be a trap representation for the type.

Section 28.7: Using incorrect format specifier in printf

Using an incorrect format specifier in the first argument to `printf` invokes undefined behavior. For example, the code below invokes undefined behavior:

```
long z = 'B';
printf("%c\n", z);
```

Here is another example

```
printf("%f\n",0);
```

Above line of code is undefined behavior. `%f` expects double. However 0 is of type `int`.

Note that your compiler usually can help you avoid cases like these, if you turn on the proper flags during compiling (`-Wformat` in `clang` and `gcc`). From the last example:

```
warning: format specifies type 'double' but the argument has type
      'int' [-Wformat]
    printf("%f\n",0);
            ~~    ^
            %d
```

Section 28.8: Modify string literal

In this code example, the char pointer `p` is initialized to the address of a string literal. Attempting to modify the string literal has undefined behavior.

```
char *p = "hello world";
p[0] = 'H'; // Undefined behavior
```

However, modifying a mutable array of `char` directly, or through a pointer is naturally not undefined behavior, even if its initializer is a literal string. The following is fine:

```
char a[] = "hello, world";
char *p = a;

a[0] = 'H';
p[7] = 'W';
```

That's because the string literal is effectively copied to the array each time the array is initialized (once for variables with static duration, each time the array is created for variables with automatic or thread duration — variables with allocated duration aren't initialized), and it is fine to modify array contents.

Section 28.9: Passing a null pointer to printf %s conversion

The `%s` conversion of `printf` states that the corresponding argument *a pointer to the initial element of an array of character type*. A null pointer does not point to the initial element of any array of character type, and thus the behavior of the following is undefined:

```
char *foo = NULL;
```

```
printf("%s", foo); /* undefined behavior */
```

However, the undefined behavior does not always mean that the program crashes — some systems take steps to avoid the crash that normally happens when a null pointer is dereferenced. For example Glibc is known to print

```
(null)
```

for the code above. However, add (just) a newline to the format string and you will get a crash:

```
char *foo = 0;
printf("%s\n", foo); /* undefined behavior */
```

In this case, it happens because GCC has an optimization that turns `printf("%s\n", argument);` into a call to `puts` with `puts(argument)`, and `puts` in Glibc does not handle null pointers. All this behavior is standard conforming.

Note that *null pointer* is different from an *empty string*. So, the following is valid and has no undefined behaviour. It'll just print a *newline*:

```
char *foo = "";
printf("%s\n", foo);
```

Section 28.10: Modifying any object more than once between two sequence points

```
int i = 42;
i = i++; /* Assignment changes variable, post-increment as well */
int a = i++ + i--;
```

Code like this often leads to speculations about the "resulting value" of `i`. Rather than specifying an outcome, however, the C standards specify that evaluating such an expression produces *undefined behavior*. Prior to C2011, the standard formalized these rules in terms of so-called *sequence points*:

> Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.

(C99 standard, section 6.5, paragraph 2)

That scheme proved to be a little too coarse, resulting in some expressions exhibiting undefined behavior with respect to C99 that plausibly should not do. C2011 retains sequence points, but introduces a more nuanced approach to this area based on *sequencing* and a relationship it calls "sequenced before":

> If 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, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.

(C2011 standard, section 6.5, paragraph 2)

The full details of the "sequenced before" relation are too long to describe here, but they supplement sequence points rather than supplanting them, so they have the effect of defining behavior for some evaluations whose behavior previously was undefined. In particular, if there is a sequence point between two evaluations, then the one before the sequence point is "sequenced before" the one after.

The following example has well-defined behaviour:

```
int i = 42;
i = (i++, i+42); /* The comma-operator creates a sequence point */
```

The following example has undefined behaviour:

```
int i = 42;
printf("%d %d\n", i++, i++); /* commas as separator of function arguments are not comma-operators */
```

As with any form of undefined behavior, observing the actual behavior of evaluating expressions that violate the sequencing rules is not informative, except in a retrospective sense. The language standard provides no basis for expecting such observations to be predictive even of the future behavior of the same program.

Section 28.11: Freeing memory twice

Freeing memory twice is undefined behavior, e.g.

```
int * x = malloc(sizeof(int));
*x = 9;
free(x);
free(x);
```

Quote from standard(7.20.3.2. The free function of C99):

> Otherwise, if the argument does not match a pointer earlier returned by the calloc, malloc, or realloc function, or if the space has been deallocated by a call to free or realloc, the behavior is undefined.

Section 28.12: Bit shifting using negative counts or beyond the width of the type

If the *shift count* value is a **negative value** then both *left shift* and *right shift* operations are undefined1:

```
int x = 5 << -3; /* undefined */
int x = 5 >> -3; /* undefined */
```

If *left shift* is performed on a **negative value**, it's undefined:

```
int x = -5 << 3; /* undefined */
```

If *left shift* is performed on a **positive value** and result of the mathematical value is **not** representable in the type, it's undefined1:

```
/* Assuming an int is 32-bits wide, the value '5 * 2^72' doesn't fit
 * in an int. So, this is undefined. */

int x = 5 << 72;
```

Note that *right shift* on a **negative value** (.e.g `-5 >> 3`) is *not* undefined but *implementation-defined*.

1 Quoting *ISO/IEC 9899:201x*, section 6.5.7:

> If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

Section 28.13: Returning from a function that's declared with `_Noreturn` or `noreturn` function specifier

Version ≥ C11

The function specifier `_Noreturn` was introduced in C11. The header **`<stdnoreturn.h>`** provides a macro `noreturn` which expands to `_Noreturn`. So using `_Noreturn` or `noreturn` from **`<stdnoreturn.h>`** is fine and equivalent.

A function that's declared with `_Noreturn` (or `noreturn`) is not allowed to return to its caller. If such a function *does* return to its caller, the behavior is undefined.

In the following example, `func()` is declared with `noreturn` specifier but it returns to its caller.

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

noreturn void func(void);

void func(void)
{
    printf("In func()...\n");
} /* Undefined behavior as func() returns */

int main(void)
{
    func();
    return 0;
}
```

`gcc` and `clang` produce warnings for the above program:

```
$ gcc test.c
test.c: In function ‘func’:
test.c:9:1: warning: ‘noreturn’ function does return
}
^
$ clang test.c
test.c:9:1: warning: function declared 'noreturn' should not return [-Winvalid-noreturn]
}
^
```

An example using `noreturn` that has well-defined behavior:

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

noreturn void my_exit(void);
```

```c
/* calls exit() and doesn't return to its caller. */
void my_exit(void)
{
    printf("Exiting...\n");
    exit(0);
}

int main(void)
{
    my_exit();
    return 0;
}
```

Section 28.14: Accessing memory beyond allocated chunk

A a pointer to a piece of memory containing `n` elements may only be dereferenced if it is in the range `memory` and `memory + (n - 1)`. Dereferencing a pointer outside of that range results in undefined behavior. As an example, consider the following code:

```c
int array[3];
int *beyond_array = array + 3;
*beyond_array = 0; /* Accesses memory that has not been allocated. */
```

The third line accesses the 4th element in an array that is only 3 elements long, leading to undefined behavior. Similarly, the behavior of the second line in the following code fragment is also not well defined:

```c
int array[3];
array[3] = 0;
```

Note that pointing past the last element of an array is not undefined behavior (`beyond_array = array + 3` is well defined here), but dereferencing it is (`*beyond_array` is undefined behavior). This rule also holds for dynamically allocated memory (such as buffers created through `malloc`).

Section 28.15: Modifying a const variable using a pointer

```c
int main (void)
{
    const int foo_readonly = 10;
    int *foo_ptr;

    foo_ptr = (int *)&foo_readonly; /* (1) This casts away the const qualifier */
    *foo_ptr = 20; /* This is undefined behavior */

    return 0;
}
```

Quoting *ISO/IEC 9899:201x*, section 6.7.3 §2:

> If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. [...]

(1) In GCC this can throw the following warning: `warning: assignment discards ‘const’ qualifier from pointer target type [-Wdiscarded-qualifiers]`

Section 28.16: Reading an uninitialized object that is not backed by memory

Version ≥ C11

Reading an object will cause undefined behavior, if the object is1:

- uninitialized
- defined with automatic storage duration
- it's address is never taken

The variable a in the below example satisfies all those conditions:

```
void Function( void )
{
    int a;
    int b = a;
}
```

1 (Quoted from: ISO:IEC 9899:201X 6.3.2.1 Lvalues, arrays, and function designators 2)
If the lvalue designates an object of automatic storage duration that could have been declared with the register storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

Section 28.17: Addition or subtraction of pointer not properly bounded

The following code has undefined behavior:

```
char buffer[6] = "hello";
char *ptr1 = buffer - 1;  /* undefined behavior */
char *ptr2 = buffer + 5;  /* OK, pointing to the '\0' inside the array */
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char *ptr4 = buffer + 7;  /* undefined behavior */
```

According to C11, if 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, the behavior is undefined (6.5.6).

Additionally it is naturally undefined behavior to *dereference* a pointer that points to just beyond the array:

```
char buffer[6] = "hello";
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char value = *ptr3;       /* undefined behavior */
```

Section 28.18: Dereferencing a null pointer

This is an example of dereferencing a NULL pointer, causing undefined behavior.

```
int * pointer = NULL;
int value = *pointer; /* Dereferencing happens here */
```

A `NULL` pointer is guaranteed by the C standard to compare unequal to any pointer to a valid object, and dereferencing it invokes undefined behavior.

Section 28.19: Using fflush on an input stream

The POSIX and C standards explicitly state that using `fflush` on an input stream is undefined behavior. The `fflush` is defined only for output streams.

```
#include <stdio.h>

int main()
{
    int i;
    char input[4096];

    scanf("%i", &i);
    fflush(stdin); // <-- undefined behavior
    gets(input);

    return 0;
}
```

There is no standard way to discard unread characters from an input stream. On the other hand, some implementations uses `fflush` to clear `stdin` buffer. Microsoft defines the behavior of `fflush` on an input stream: If the stream is open for input, `fflush` clears the contents of the buffer. According to POSIX.1-2008, the behavior of `fflush` is undefined unless the input file is seekable.

See Using `fflush(stdin)` for many more details.

Section 28.20: Inconsistent linkage of identifiers

```
extern int var;
static int var; /* Undefined behaviour */
```

C11, §6.2.2, 7 says:

> If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

Note that if an prior declaration of an identifier is visible then it'll have the prior declaration's linkage. *C11, §6.2.2, 4* allows it:

> For an identifier declared with the storage-class specifier extern in a scope in which a prior declaration of that identifier is visible,31) if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

```
/* 1. This is NOT undefined */
static int var;
extern int var;


/* 2. This is NOT undefined */
static int var;
static int var;
```

```c
/* 3. This is NOT undefined */
extern int var;
extern int var;
```

Section 28.21: Missing return statement in value returning function

```c
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* Trying to use the (not) returned value causes UB */
  int value = foo();
  return 0;
}
```

When a function is declared to return a value then it has to do so on every possible code path through it. Undefined behavior occurs as soon as the caller (which is expecting a return value) tries to use the return value1.

Note that the undefined behaviour happens *only if* the caller attempts to use/access the value from the function. For example,

```c
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* The value (not) returned from foo() is unused. So, this program
   * doesn't cause *undefined behaviour*. */
  foo();
  return 0;
}
```

Version ≥ C99

The `main()` function is an exception to this rule in that it is possible for it to be terminated without a return statement because an assumed return value of `0` will automatically be used in this case2.

1 (*ISO/IEC 9899:201x*, 6.9.1/12)

> If the } that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

2 (*ISO/IEC 9899:201x*, 5.1.2.2.3/1)

> reaching the } that terminates the main function returns a value of 0.

Section 28.22: Division by zero

```c
int x = 0;
int y = 5 / x;  /* integer division */
```

or

```
double x = 0.0;
double y = 5.0 / x;  /* floating point division */
```

or

```
int x = 0;
int y = 5 % x;  /* modulo operation */
```

For the second line in each example, where the value of the second operand (x) is zero, the behaviour is undefined.

Note that most implementations of floating point math will follow a standard (e.g. IEEE 754), in which case operations like divide-by-zero will have consistent results (e.g., `INFINITY`) even though the C standard says the operation is undefined.

Section 28.23: Conversion between pointer types produces incorrectly aligned result

The following *might* have undefined behavior due to incorrect pointer alignment:

```
char *memory_block = calloc(sizeof(uint32_t) + 1, 1);
uint32_t *intptr = (uint32_t*)(memory_block + 1);  /* possible undefined behavior */
uint32_t mvalue = *intptr;
```

The undefined behavior happens as the pointer is converted. According to C11, if a *conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3), the behavior is undefined*. Here an `uint32_t` could require alignment of 2 or 4 for example.

`calloc` on the other hand is required to return a pointer that is suitably aligned for any object type; thus `memory_block` is properly aligned to contain an `uint32_t` in its initial part. Then, on a system where `uint32_t` has required alignment of 2 or 4, `memory_block + 1` will be an *odd* address and thus not properly aligned.

Observe that the C standard requests that already the cast operation is undefined. This is imposed because on platforms where addresses are segmented, the byte address `memory_block + 1` may not even have a proper representation as an integer pointer.

Casting `char *` to pointers to other types without any concern to alignment requirements is sometimes incorrectly used for decoding packed structures such as file headers or network packets.

You can avoid the undefined behavior arising from misaligned pointer conversion by using `memcpy`:

```
memcpy(&mvalue, memory_block + 1, sizeof mvalue);
```

Here no pointer conversion to `uint32_t*` takes place and the bytes are copied one by one.

This copy operation for our example only leads to valid value of `mvalue` because:

- We used `calloc`, so the bytes are properly initialized. In our case all bytes have value `0`, but any other proper initialization would do.
- `uint32_t` is an exact width type and has no padding bits
- Any arbitrary bit pattern is a valid representation for any unsigned type.

Section 28.24: Modifying the string returned by getenv, strerror, and setlocale functions

Modifying the strings returned by the standard functions `getenv()`, `strerror()` and `setlocale()` is undefined. So, implementations may use static storage for these strings.

The getenv() function, C11, §7.22.4.7, 4, says:

> The getenv function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the getenv function.

The strerror() function, C11, §7.23.6.3, 4 says:

> The strerror function returns a pointer to the string, the contents of which are localespecific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the strerror function.

The setlocale() function, C11, §7.11.1.1, 8 says:

> The pointer to string returned by the setlocale function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the setlocale function.

Similarly the `localeconv()` function returns a pointer to `struct lconv` which shall not be modified.

The localeconv() function, C11, §7.11.2.1, 8 says:

> The localeconv function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the localeconv function.