Undefined behavior - C Language

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.

- 32

```
int* foo(int bar)
{
    int baz = 6;
    baz += bar;
    return &baz; /* (&baz) copied to new memory location outside of foo. */
} /* 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);  /* (1) this expression's behavior is undefined */
    *p = *p - 6; /* (2) 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) 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."

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

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

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

Basically, we're copying 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
```

The definitions for the library functions `memcpy()`, `strcpy()`, `strncpy()`, `strcat()` and `strncat()` as well as `sprintf()`, `snprintf()`, `vsprintf()`, `vsnprintf()` and `sscanf()` contain a clause that states:

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

The definition for the `memmove()` function does not contain this restriction. Instead, it says that the function behaves as if the source data was first copied into a temporary buffer and then written to the destination address. This means that `memmove()` can handle overlapping source and destination buffers in a well-defined manner.

The reason for this distinction is efficiency. `memmove()` can be relied upon to work correctly in all cases, but this requires the function to do numerous additional checks for overlapping buffers. `memcpy()` is used when the source and destination buffers cannot possibly overlap, and this assumption allows it to be implemented as efficiently as possible (avoiding unnecessary tests and potentially using advanced block-copy capabilities of the hardware).

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

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

Overflow of a signed integer causes Undefined Behavior (see Section 6.5 paragraph 5 of the C99 and C11 standards). This means that the compiler could do anything, as a result of assumptions and optimizations gone bad. Therefore, it is useless to speculate about the value of `i`. For example, it could happen that the program behaved as if `i < 0` yielded `true` and `false` simultaneously.

Please note:

- Once UB is introduced, the whole program is tainted.
- The behavior for unsigned integer overflow is well-defined in C.

But surely you would never write such code? Suppose you want to check whether an event occurred more than `X` seconds ago (In this example X = 140). A typical approach could look like this:

```
typedef int32_t uptime;
uptime timeOfEvent;
bool eventHasHappened = false;
```

Then the event actually happens and the variable is initialized:

```
timeOfEvent = uptime();
eventHasHappened = true;
```

And is checked like this:

```
if(eventHasHappened && uptime() - X > timeOfEvent) {
   /* do something important*/
}
```

That works - for about 248 days.

If timeOfEvent is something like 0x80000015 (which is not a large positive number but rather is interpreted as -2147483627) and `uptime()` returns 0x80000016 you end up comparing a large positive number (`uptime() - X == -2147483627 - 140 == 21474835307`) with a negative number and thus don't execute your important code!

Does that depend on very complicated circumstances and is unlikely to ever happen to you

-- yes. But that is exactly the problem with undefined behaviour: it's sneaky and will bite you in the worst possible moment.

-
  ```
  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.

```
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.

- C11
  A data race causes undefined behavior[1]. It occurs when a shared object is unprotected[2] and concurrently[3] accessed[4] by at least two different threads, where at least one thread modifies the object.

  In the example below, one thread modifies object `a` and another thread reads `a`.

  The main thread calls `thrd_create`, which creates another thread and returns without waiting for the created thread to exit. The created thread calls `Function` which modifies `a`, and main thread reads `a`. Those two actions happen concurrently[3], and cause a data race.

```
#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 );
}
```

[1] (Quoted from: ISO:IEC 9889:201x 1.10 Multi-threader executions and data races 21)
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.

[2] The object isn't atomic.

[3] Actions overlap temporally, it is not possible to reason which one happened first.

[4] Modifying or reading an object.

- 9
  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.

-
  ```
  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.

- 8

  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
  ```

- 7

  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.

- 7

  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);
  ```

- 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.

- The third line accesses the 4th element in an array that is only 3 elements long.

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

  Above code is similar to code below which is equally undefined behavior:

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

  Note that pointing one 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). Using a subscript outside the range 0..2 (for this array), or a pointer derived from the start address of the array using a subscript out of the range 0..2, is undefined behaviour.

- 5
  If the shift count value is a negative value then both left shift and right shift operations are undefined[1]:

  ```
  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 undefined[1]:

  ```
  /* 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.

- 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);

/* 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;
}
```

- 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 */
```

- 4

```
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]`

- Reading an object will cause undefined behavior, if the object is[1]:
  - 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.

- 4
  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.

- 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.

-
```
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.

-
  ```
  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;

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

- 3

  ```
  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 value[1].

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

```
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;
}
```

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 case[2].

[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.

- 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.

- 2
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.

RemarksWhat is Undefined Behavior (UB)?
Undefined behavior is a term used in the C standard. The C11 standard (ISO/IEC 9899:2011) defines the term undefined behavior as

> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

What happens if there is UB in my code?
These are the results which can happen due to undefined behavior according to standard:

> NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

The following quote is often used to describe (less formally though) results happening from undefined behavior:

> “When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose” (the implication is that the compiler may choose any arbitrarily bizarre way to interpret the code without violating the ANSI C standard)

Why does UB exist?
If it’s so bad, why didn't they just define it or make it implementation-defined?

Undefined behavior allows more opportunities for optimization; The compiler can justifiably assume that any code does not contain undefined behaviour, which can allow it to avoid run-time checks and perform optimizations whose validity would be costly or impossible to prove otherwise.

Why is UB hard to track down?
There are at least two reasons why undefined behavior creates bugs that are difficult to detect:

- The compiler is not required to - and generally can't reliably - warn you about undefined behavior. In fact requiring it to do so would go directly against the reason for the existence of undefined behaviour.
- The unpredictable results might not start unfolding at the exact point of the operation where the construct whose behavior is undefined occurs; Undefined behaviour taints the whole execution and its effects may happen at any time: During, after, or even before the undefined construct.

Consider null-pointer dereference: the compiler is not required to diagnose null-pointer dereference, and even could not, as at run-time any pointer passed into a function, or in a global variable might be null. And when the null-pointer dereference occurs, the standard does not mandate that the program needs to crash. Rather, the program might crash earlier, later, or not crash at all; it could even behave as if the null pointer pointed to a valid object, and behave completely normally, only to crash under other circumstances.

In the case of null-pointer dereference, C language differs from managed languages such as Java or C#, where the behavior of null-pointer dereference is defined: an exception is thrown, at the exact time (`NullPointerException` in Java, `NullReferenceException` in C#), thus those coming from Java or C# might incorrectly believe that in such a case, a C program must crash, with or without the issuance of a diagnostic message.

Additional information
There are several such situations that should be clearly distinguished:

- Explicitly undefined behavior, that is where the C standard explicitly tells you that you are off limits.
- Implicitly undefined behavior, where there is simply no text in the standard that foresees a behavior for the situation you brought your program in.

Also have in mind that in many places the behavior of certain constructs is deliberately undefined by the C standard to leave room for compiler and library implementors to come up with their own definitions. A good example are signals and signal handlers, where extensions to C, such as the POSIX operating system standard, define much more elaborated rules. In such cases you just have to check the documentation of your platform; the C standard can't tell you anything.

Also note that if undefined behavior occurs in program it doesn't mean that just the point where undefined behavior occurred is problematic, rather entire program becomes meaningless.

Because of such concerns it is important (especially since compilers don't always warn us about UB) for person programming in C to be at least familiar with the kind of things that trigger undefined behavior.

It should be noted there are some tools (e.g. static analysis tools such as PC-Lint) which aid in detecting undefined behavior, but again, they can't detect all occurrences of undefined behavior.