Bachelor thesis
Computer Science

Radboud University

Defining the Undefined in C

Author:
Matthias Vogelaar
S4372913

First supervisor/assessor:
dr. F. Wiedijk
freek@cs.ru.nl

Second assessor:
M.R. Schoolderman
m.schoolderman@science.ru.nl

April 4, 2017

Abstract

The C programming language is one the oldest, still widely used programming languages. Despite being officially specified in a standard, C contains certain elements that produce 'undefined behavior'. The C Standard does not dictate what will happen when such code is executed. It is up to the compiler to determine what the result will be.
Due to the nature of undefined behavior, programs containing this behavior will produce different results when compiled with different compilers. It is even possible that the same compiler with a different optimization setting will produce a differently behaving program from the same source code.
In this bachelor thesis we will take a look at different C compilers and different formal semantics of C. First we will determine which specific actions lead to undefined behavior. Next, we try to determine what the rational argument is for leaving the specific action unspecified. Then, we will test what each semantic and compiler does when it encounters said behavior, and come up with an explanation as to why a certain tool handles the problem the way it does.

Contents

Chapter 1

Introduction

Undefined behavior implies that a certain aspect of a programming language is not properly defined and is thus allowed to behave unchecked. In reality it is more complex. Most commonly, undefined behavior occurs when a programmer tries to do something, deliberately or by accident, that does not really make any sense. A simple example would be trying to divide by zero. However, not every case of undefined behavior will be handled as an error. Dividing by zero will probably halt your program, writing outside an array may not, despite both being undefined behavior. This implies that undefined behavior is not a term for 'an error occurs', nor will it always halt the execution of a program. Apparently, there are other reasons for leaving certain executions undefined, which leads to the research question:

What are the reasons behind undefined behavior, not forcing a halt when encountered and what are its consequences?

We can divide this question in the following sub questions:

- Which operations cause undefined behavior?
- What is the rational argument to define these operations as undefined?
- What happens when a program encounters a piece of undefined behavior?

The thesis will have the following structure. First, we will provide some additional context regarding the different types of behavior in C. Next, we will list and shortly describe the compilers and semantics we have used. Then, we briefly show an example of undefined behavior. This is followed by a general explanation as to why to allow undefined behavior, and a categorization of the different types of undefined behavior. After that, we will study individual cases of undefined behavior, followed by related work and conclusions.

Chapter 2

About undefined behavior and tools

2.1 About undefined behavior

Besides undefined behavior, there are two related types of 'not strictly defined' behavior. We will explain these kinds of behavior below. Note that this thesis focuses on undefined behavior. The other types of behavior are mentioned for context purposes.

Unspecified behavior When we think of a typical computer program, most will agree it does exactly what the programmer told the program to do. That is, assuming it does not contain any bugs. For instance, take the following code snippet.

```
int x = 5;
```

Everyone will agree that this will result in the integer x having a value of 5. This behavior is well defined. Things get a little bit more complicated if we look at the following code snippet.

```
int x = foo(bar(1),bar(2));
```

Which `bar` function is evaluated first? Does the first function influence the result of the second? The C standard does not give a definitive answer. However, obviously, the compiler has to pick an evaluation order. This type of problem falls into the 'unspecified behavior' category. To quote the ISO C11 standard [1]:
"use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance" (C11 standard, 3.4.4 clause 2)
To paraphrase: if there are multiple options to do the same thing, pick one.

Implementation defined behavior According to the C11 standard:

"[Implementation defined behavior:] unspecified behavior where each implementation documents how the choice is made" (C11 Standard 3.4.1)

An example is the size of an integer. 16-bit processors typically used integers consisting of two bytes, whereas 32-bit processors use four bytes to represent an integer. Thus, this kind of unspecified behavior is implementation depended, and should be properly documented, though this is not always the case. Clang does not have such documentation, beyond what may be specified in the manual [2].

Undefined behavior The final category is undefined behavior:

"behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements." (C11 Standard 3.4.3)

Note the difference between nonportable and erroneous program construct. Integer overflow is classified as undefined behavior. The reason being that different architectures handle or representing integers differently, thus making overflow nonportable A non-void function not returning a value is an example of an erroneous program construct.
The compiler may assume undefined behavior **does not occur**. This can result in strange behavior when executing programs. For instance, because integer overflow is undefined, the compiler may assume that `i < i + 1` will **always** be true, because the statement will only validate to false if overflow occurs. Thus, the following program might never terminate if the compiler decides to remove the check.

```
for(int i = 0; i < i + 1; i++){ doStuff(); }
```

Another example of undefined behavior which can have malicious results is accessing an array outside of its bounds. Buffer overflow might be considered one of, if not the biggest cause of security related issues.

2.2 Compilers and semantics used

We will be using several compilers and formal semantics in this research project. As a base for comparison, we will be using the GNU C compiler, known as GCC, and CLang. Since these compilers are widespread and often used, they are best suited to be used to compare the other compilers and semantics. We will compare the behavior of the compilers to the following C semantics: CompCert, CH_2O and C in K. The precise installation instructions for each compiler and semantic can be found in appendix A.
Following below is a brief description of each of the used compilers.

GCC GCC, or the GNU Compiler Collection [3], is one of the most used, most ported C compilers currently in use. It can be used on virtually any platform. GCC does not focus specifically on clearly defining the undefined behavior of C. We have used version 5.4.0.

Clang Compared with GCC (1987), a young compiler (2007). It was originally designed by Apple and later turned open source [4]. It is highly compatible with GCC and shares many of GCC's command line arguments and flags. Clang aims at minifying it's memory footprint and compilation speeds compared to other compilers. We have used version 3.8.0.

CompCert The aim of the CompCert project is to formally verify compilers used for critical embedded software. The result of said project is a C compiler which comes with a mathematical proof that the resulting code behaves exactly as defined by the semantics of the compiled program. [5] We used version 2.7.1.

CH_2O A project by Robbert Krebbers, the CH_2O project aims at formalizing the ISO C11 standard. The translation from C-source files and the executable semantics are both defined as Coq programs. Furthermore, all proofs for the semantics are fully formalized in Coq. "CH_2O provides an operational, executable, and axiomatic semantics for a significant fragment of C11." [6] We used the latest committed version available on Git, which was last updated at 7 May 2016.

C in K This formal semantic calls itself a "negative" semantic. Not only does it produce valid programs, but it also rejects programs that utilize undefined behavior. It attempts to formalize the ISO C11 standard, just like the CH_2O project [7]. Unlike CH_2O, its semantics is not based in Coq. C in K does describe more functionality than CH_2O.

Chapter 3

The reasons for undefined behavior

The previous chapter explained the different kinds of behavior in C, and outlined the tools we will be using in this thesis. In this chapter, we will provide an example of undefined behavior, as well as a rationale for allowing undefined behavior in general. Finally, we will categorize the undefined behavior in C and explain which specific cases from the categories we will investigate further.

3.1 Examples of undefined behavior

We will demonstrate what effects undefined behavior can have on a program. These examples are meant as a demonstration, not as an in-depth analysis of the occurring behavior. Consider the following program:

```
#include<stdio.h>

int f(){}

int main()
{
    int x = f();
    printf("%d\n", x);
    return 0;
}
```

We try to print variable `x`, which we initialize via function `f`. However, this function is completely empty and does not return any value at all. The table below list the output of executing the program per tool. The warning column indicates whether or not the compiler produced a warning. No additional warning flags were set.

Tool	Warning	Output
GCC	No	`0`
GCC -O3	No	`0`
Clang	Yes	`0`
CompCert	No	`-143626820`
CCS	-	`Undefined Behavior`
C in K	No	`Undefined behavior (UB-CCV5).` `See C11 sec. 6.3.2.2:1, J.2:1 item 23`
CH_2O	-	`Fatal error: undef(_)`

The output does not seem interesting. GCC and Clang default the value to zero, and CompCert seems to assign a random number. Clang however, does give a warning to the user that something is amiss with their program. C in K, CompCert in semantic mode and CH_2O halt the program when encountering the `int x = f();` due to it being undefined behavior.

The following program is similar to the previous one. This time the functions contain some statements which should influence the result.

```
#include <stdio.h>

int f(int i){
    int o = i + 1000;
}
int ff(int i){
    int o = i + 100;
    int r = 2;
    r += i;
}
int fff(int i){
    int p = -7;
    int o = i + 5;
    int r = -10;
}
int main()
{
    int i = f(1); int j = ff(2); int k = fff(3);
    printf("%d␣%d␣%d\n",i,j,k);
    return 0;
}
```

Tool	Warning	Output
GCC	No	`1001 2 8`
GCC -O3	No	`0 0 0`
Clang	Yes	`0 0 0`
CompCert	No	`-142918212 2 2`
CCS	-	`Undefined Behavior`
C in K	No	`Undefined behavior (UB-CCV5).` `See C11 sec. 6.3.2.2:1, J.2:1 item 23`
CH_2O	-	`Fatal error: exception Main.CH2O_undef(_)`

Again, C in K, semantic CompCert and CH_2O halt during execution. The optimized GCC and Clang behave the same as the previous program. The unoptimized GCC program acts a bit different. There it seems function `f` returns the value of `o`, `ff` returns the original argument and `fff` does return the value of `o` again. CompCert however treats it differently. `i` seems to be treated as a non-initialized variable, `j` gets it value either from the input or from variable `r`. The value of `k` is most likely influenced by `j`.

3.2 Allowing undefined behavior

If undefined behavior can cause programs to behave unpredictably, then why should we allow it? Two main reasons come to mind. First of all, undefined behavior increases portability and simplicity of the programming language. Take a shift operation, shifting an integer more positions than its size. Some architectures will fill the register with zeros Others might execute the shift modulo the size of the integer. Arguments can be made for both, and both results can be achieved with alternative operations. Not specifying what needs to happen results in a simpler standard which can be more easily implemented across different platforms.
Another valid argument would be optimization. If we can assume integer overflow is undefined and thus never occurs, we can suddenly simplify mathematics. The statement `(x*4)/2` can be simplified to `x*2` . This would not be possible if integer overflow was defined behavior.

3.3 Categorizing undefined behavior

Appendix J.2 lists, and references, all cases which can lead to undefined behavior. Furthermore, C in K has a handy test-suite which contains a program build to encounter pretty much every one of those cases mentioned in this appendix. We can roughly categorize all the instances listed in the appendix in the following categories:

- Common
- Source Files
- File handling and input output streams
- Invalid operations and Sequence Points
- Arithmetic
- Casting
- Data access and pointers
- Macros
- Rest / Uncategorized

See appendix C for the categorization. From all the instances, we specifically look at those cases that either have an interesting reason for being undefined, or are actually being used by the compiler to optimize code. For each case investigated we explain the problem at hand, and try to determine the rationale behind labeling it undefined. Furthermore, we investigate if a case causes different behavior when used in conjunction with different compilers or semantics.

Chapter 4

Research

In this section, we will go through each of the individual undefined cases. For each case, we first explain what the specific problem is. Next, if possible, we will come up with a possible way to define the undefined behavior. Then, we will show why it may be better to leave it undefined instead of forcing a default.
We will also provide an example program accompanied by the output of the program when executed by our set of compilers and semantics. The compilers will execute the program at optimization levels 0, 1 and 3. The output of executing the program at level 0 will always be listed in the table. If the output of the optimized program is different from the non-optimized version, then the output will be shown in the table as well. Otherwise, that entry is discarded. Besides the output, we will also mention whether or not the compiler will produce a warning by default when compiling the example program. The output of the semantic tools are mentioned to verify that these tools indeed do not tolerate any undefined behavior. CompCert can be used as a compiler or as a semantic interpreter. Both results are listed in the output table, were the result of executing the semantic interpreter is listed as CCS.
Furthermore, we ran benchmarks for the implementations that tried to define an undefined action. In these benchmarks we compare the runtime of a 'safe' implementation of the undefined behavior with a 'raw' implementation. For these benchmarks we have used a virtual machine running Ubuntu 16.04. Several benchmarks were performed to ensure an accurate result. The benchmark programs are listed in appendix B.

4.1 Arithmetic

4.1.1 Overflow

Integer overflow occurs whenever a statement is executed of which its result falls outside of the range of an integer. Note that unsigned integer overflow is defined (C11 6.2.5 clause 9). This kind of undefined behavior can be the result of a multiplication, addition, subtraction or shifting. The problem here is the simple fact that we end up with a number we cannot represent. When testing this behavior using our tool set, all compilers act the same. The result of overflow is a wraparound to the lowest value. CH_2O and C in K both halt because of undefined behavior. CompCert in semantic mode does not, and acts the same as the compilers.
One rationale behind leaving signed integer overflow undefined, while unsigned overflow is defined, is the fact that C allows multiple signed integer representations. In section 6.2.6.2 of the C11 standard, clause 2 prescribes the use of either one's complement, two's complement or sign-magnitude. This implies that overflow happens differently with those representations. Enforcing the more standard representation, two's complement, would either result in dropping the other two representations completely, or come up with a platform specific workaround.
Overflow by shifting has a similar rationale. Shifting is implemented differently on different processors. Enforcing a certain style of shifting would have quite a performance impact if the compiler suddenly needs to manually wrap a shift operation.
Another note is that leaving overflow undefined is that it allows for more optimizations. Since we can assume undefined behavior never happens, we can assume that integer overflow never happens. This would allow for a whole bunch of optimizations, for instance, converting the following division into a shift to the right by two:

```
if(x > 4){
     x += y;
     x = x / 4
}
```

Or changing the comparison `(x - c1)< c2` to `x < (c2 + c1)`.[8] Since we are using constants, the second operand can be determined at compile time instead of runtime.

4.1.2 Division by zero

As specified in C11 section 6.5.5 clause 5, division by zero is undefined. It is worth noting that division by zero is not defined mathematically. Thus, the rationale to leave division by zero undefined is quite obvious. Creating a program that is tasked to do the impossible is not very difficult:

```
#include<stdio.h>

int main(){
    printf("%d\n", 1/0);
    return 0;
}
```

Executing this program yields the following results:

Tool	Warning	Output
GCC	Yes	`Floating Point Exception`
Clang	Yes	`-743204856` (produces random number)
CompCert	No	`Floating Point Exception`
CCS	-	`Undefined Behavior`
C in K	No	`Undefined behavior` `See C11 sec. 6.3.2.2:1, J.2:1 item 23`
CH_2O	-	`Fatal error: undef(_)`

Apparently, Clang decides to assign a value to the operation. One might argue we could assign a specific value to this action. This would allow such a command to be executed without crashing the program. Let's say we assign the value A to a division by zero. Since different processors do different things when encountering a divide-by-zero, we needs to check if the second operator is zero before attempting the division. In code, this would be equivalent (for integers) to the following:

```
int divide_safe(int a, int b) {
    if(b != 0)
        return a/b;
    return INT_MIN;
}
```

We benchmarked this safe function against a non-safe counterpart. The non-safe function only contains the divide statement. This will minimize the difference in overhead caused by calling a function. During the benchmark, we ran 3 billion divisions using these functions and measured the time in milliseconds it took to complete. The average of these results are below:

	-O0		-O1		-O3	
	d	d-safe	d	d-safe	d	d-safe
GCC	3182	3680	2053	2574	2042	2042
Clang	3221	3757	2295	2549	1926	1925
CompCert	2574	3337	2569	3353	2566	3327

As the results show, this safe division increases the executing time with at least 15%. With `-O3` enabled, executing time does not show any measurable impact. However, since division isn't the fastest operation, and 3 billion operations we're needed to show the decrease in performance, it might not have a significant effect in a real world application. The downside is that this safe division function does not really solve anything for the programmer that does not want to divide by zero. Either the programmer would need to check the second operand beforehand, or he needs to check the result afterwards, resulting in a redundant check. An alternative would be to enforce termination of the running program, equivalent of calling abort instead of returning a specific value, which is what GCC and CompCert do. However, leaving this case undefined results in a choice. One could think of a situation where you do not want your program to abort, and are okay with a certain fixed or random value for a divide-by-zero statement.

4.1.3 Undefined shifts

Shifting allows us to move all bits in a register several positions. Naturally, we could shift a value more positions than its size, as done in the following program:

```
#include<stdio.h>

int main()
{
    int i = 1 << sizeof(int)*8;
    printf("%d\n",i);
    return 0;
}
```

There are several things that might happen. The result could be filled with zeros. Another possibility is the shift is done modulo the width of the type, thus resulting in no action at all. Execution yields the following:

Tool	Warning	Output
GCC	Yes	`0`
Clang	Yes	`4195632` (constant in multiple runs)
Clang -O1/3	Yes	`-1026980664` (random number every run)
CompCert	No	`1`
CCS	-	`Undefined Behavior`
C in K	No	`Undefined behavior` `See C11 sec. 6.5.7:3, J.2:1 item 51`
CH_2O	-	`Fatal error: undef(_)`

Our compilers all produce different results. GCC executes the shift and pads with zeros. Clang produces random output and CompCert executes the shift modulo the size of the type. As shown, different compilers produce different results. This could also be the case for different architectures. Enforcing a specific result with larger than width shifts will thus either limit the portability of C, or code will perform differently on different machines. It can also be argued that shifting beyond the size of a type is a pointless operation. The same result can be achieved with a smaller shift, or by explicitly setting the value to zero. Specifying a specific result for an arguable pointless operation should not included in a standard.

4.2 Pointers

4.2.1 Writing past array end

Better known as buffer overflow, writing past the end of an array is undefined behavior, and is a very common cause of security issues. Buffer overflow is made possible by C not checking if a certain index of an array is in fact a valid index. Accessing an index that is out of bounds typically results in the program reading the memory just beyond the array, as if it is part of the array. It will read the memory as if it has the same type as the array.

```
#include <stdio.h>

int main()
{
    char p1[8] = {'a','b','c','d','e','f','g','h'};
    char overflow[2] = {'z','x'};
    char p2[8] = {'i','j','k','l','m','n','o','p'};
    for(int i = 0; i < 24; i++)
        printf("%c", overflow[i]);
    return 0;
}
```

Executing this program with our tool set yields the following results:
Note: special characters are replaced with spaces.

Tool	Warning	Output
GCC	No	`zx abcdefgh`
GCC -O1/3	Yes	`zx@ $`
Clang	No	`zxabcdefgh @ 0`
Clang -O1/3	No	`zx%c ;0`
CompCert	No	`zxz ijklmnopabcdefgh`
CCS	-	`Undefined Behavior`
C in K	No	`Undefined behavior` `See C11 sec. 6.5.6:8, J.2:1 items 47, 49`
CH_2O	-	`Fatal error: undef(_)`

As shown by the output, the loop prints characters from beyond the array. The padding arrays are there to show that we can reach other valid data. The optimized versions yield a different result because the compiler decides it does not need to store the `p1` and `p2` arrays in memory. The program also demonstrates the differences in memory layout between the different compilers.

This is all made possible by the absence of bound checking, which has several advantages. First of all, for each read and write operation on an array, no bounds need to be checked. This allows faster array access. Second, there is no need to store the size of every array, thus reducing memory usage. The absence of bounds checking has more consequences than it seems at first. If we want to perform bound checking we need to know what size the array is, even if we got to that array via the use of a pointer. That implies the pointer needs to know it is pointing to an array, and second, what specific size that array has. At first glance, these implications would lead to more complicated pointers and a larger overhead when handling arrays [9]. However, there are solutions that, in specific cases, only increase runtime with 12% [10]. A notable extra case of undefined behavior is displayed in the following program:

```
int main()
{
    int a[5][5];
    for(int i = 0;i < 5;i++)
        for(int j = 0;j<5;j++)
            a[i][j]=(i*5)+j;
    printf("%d\n",a[1][10]);
    return 0;
}
```

One could argue the print statement would be equivalent to printing with indices 3 and 0 respectively. GCC, Clang, CompCert and CompCert in semantic mode all output `15`, while the rest of the semantic tools all stop due to it being undefined behavior. This seems to indicate that a multidimensional array with dimensions `n` and `m` is always equivalent to a one dimensional array of dimension `n*m`. Enforcing multidimensional arrays to be equivalent to their flattened one dimensional counterpart is possible. However, this would limit the compiler in allocating the array as it sees fit, and for example, creating an array of pointers to the sub arrays

4.2.2 Deceased objects and their pointers

Section 6.2.4 of the C11 standard states that objects may not be referenced outside of their lifetime. Pointers to deceased object become indeterminate the moment that object is freed. Since using indeterminate values is also undefined behavior, using any pointer to a freed object is also undefined behavior. The following example illustrates the issue:

```
int func(int* p) {
    int *q = p;
    free(p);
    if (q == p) return 1;
    return 0;
}
```

Looking at the example from, it might seem obvious that this function should always return 1. Running this program through our tool set result in all compilers printing `1`, and all semantic tools halting due to undefined behavior. However, since `p` and thus `q`, point to an object that is no longer there, the result is undefined behavior. The compiler may optimize this to freeing the memory of `p` and return 0. The rationale is that we cannot know what the pointer is referencing. In the example, it is likely that as we reach the if statement, the memory location `p` was pointing to has not yet been overwritten. However, it might be in the general case. Note that `p` might have pointed to any location in memory. Also, we can no longer guarantee `p` points to data of the same type as it was before. Allowing to write data to the location of `p` can have strange side effects. Basically, we are using a backdoor to corrupt data elsewhere. The C99 rationale [11] states in section 6.3.2.3 that using such a pointer might yield exceptions [12].

Now all that remains is to determine why we cannot even use the value of the pointer itself. Pointers in C are not meant for comparison between different objects, or calculating offsets between them. As a result, relational operations on pointers to different objects cause undefined behavior, except for the equal and unequal operator. Arithmetic operations on pointers that are not pointing to arrays also yield undefined behavior. Comparing a freed object against anything else cannot really be done, since the freed object is no longer there. Calculating offsets in an array is of no use either, since the array is already gone. Thus, using the value of such a pointer is undefined behavior.

4.2.3 Homemade pointers

Consider the following program [13][14]:

```
#include<stdio.h>

int main()
{
    int a=1, b=2, c=3, d=4;
    int *p, *q, *r;
    p=&b; q=&c+1; r=&d;
    printf("p/q:␣%p␣%p␣%d\n", p, q, p == q);
    printf("r/q:␣%p␣%p␣%d\n", r, q, r == q);
    return 0;
}
```

The program allocates three integers, and three pointers. Each pointer points to one of the integers. The pointer `q` however, points to the location of `c` plus one. Depending on the memory layout, this could be either `b` or `d`. If that is a correct assumption, one of the print statements should produce two equal numbers and a number one. The output using the various tools at our disposal yields the following:

Tool	Warning	Output
GCC	No	`p/q: 0x7ffe9c209374 0x7ffe9c209374 1`
GCC -O1/3	No	`p/q: 0x7fffe09a12bc 0x7fffe09a12bc 0`
Clang	No	`r/q: 0x7ffe623aad10 0x7ffe623aad10 1`
Clang -O1/3	No	`r/q: 0x7fff370eb214 0x7fff370eb214 1`
CompCert	No	`p/q: 0xff9519b4 0xff9519b4 1`
CCS	-	`Undefined Behavior`
C in K	No	`Undefined behavior` `See C11 sec. 6.5.6:8, J.2:1 items 47, 49`
CH_2O	-	`Fatal error: undef(_)`

As the `C in K` tool mentions, this technically is the same undefined behavior as writing past the end of an array. We are using a pointer to access an object that the pointer is not pointing to. What is interesting is that we can observe GCC starting to behave odd when optimized. Even though the print statement produces two identical memory addresses, the evaluation of `p==q` yields false. This is an example of GCC using undefined behavior to optimize its code. Assuming that undefined behavior never occurs, the compilers reasons that a pointer to `b` and a pointer to `c` can never be the same, even if that pointer has an offset. This assumption makes it easy for the compiler to determine which pointer is pointing to which object.

4.3 Operations

4.3.1 Modifying a variable twice

C allows us to write statements that modify multiple scalar variables with a single line of code. `x = (y--)+1` would modify both `x` and `y`. Using these constructs, we could try to modify the same variable twice in one statement. Consider the following operations:

```
#include<stdio.h>

int main()
{
    int i = 2, j =1, k = 1, l = 1, m = 1;
    i = i * i++;
    j = ++j + ++j;
    k = k + ++k;
    l = (l = 2) + (l = 4);
    m = m++;
    printf("%d␣%d␣%d␣%d␣%d\n",i,j,k,l,m);
}
```

Now try to determine what this program could possibly print. It will all depend on when certain actions take place. Take `i`, `i++` is somewhat equivalent to returning the value of `i`, then increment it. The question is, when does this increment happen? It could be immediately, resulting in something like `i = i * (i+1)`. Or it could happen after the multiplication, but before the assignment to `i`, resulting in `i = i * i`. The actual results of the print statement are listed below:

Tool	Warning	Output
GCC	Yes	`6 6 4 8 1`
Clang	Yes	`4 5 3 6 1`
CompCert	No	`6 5 4 6 1`
CCS	-	`4 5 3 6 1`
C in K	No	`Undefined behavior` `See C11 sec 6.5:2, J.2:1 item 35`
CH_2O	-	`Fatal error: undef(_)`

To explain the results, we have to look at the assembly code generated by the compilers. GCC and CompCert both save the incremented value of `i` to a separate register, and then execute the multiplication. Clang executes the multiplication, saves the increment to a separate register, and then saves the result of the multiplication. `j` could be calculated either as `j = (j+1) + (j+2)`, or as `j+=2;j=j+j`. When looking at the assembly code generated by the compilers, all compilers first execute both increments. GCC stores these increments in the same register, Clang and CompCert do not. When adding the two numbers together, Clang and CompCert have two different numbers, while GCC uses one register. The same goes for `k`, only this time CompCert also uses just one register. `l` behaves the same as `j`, and with `m`, the increment takes place right before the assignment.
While C in K and CH_2O both halt due to encountering undefined behavior, CompCert in semantic mode behaves different from the compiler mode, and produces the same result as Clang.
There might be several reasons why this behavior is undefined. First of all, some outcomes of these calculations are arguable, especially when using the `++` operand. It is only specified that the value should be increment after it has been used, but not when. Second, there are better, less ambiguous methods of achieving the same result. Third, leaving this behavior undefined results in a simpler, less restricting standard, which is always preferred over a complex, restrictive standard.

4.3.2 Modifying a string literal

Strings are implicitly constant, considering C11 6.4.5 clause 7:

"It is unspecified whether these arrays are distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined."

Thus, the following program would result in undefined behavior:

```
#include<stdio.h>

int main()
{
    char str* = "Hello";
    str[0] = 'Y';
    printf("Hello");
    return 0;
}
```

This program will cause a segmentation fault when executed with our specific compilers. However, we can still look at the assembly code generated. Below is the assembly generated by Clang:

```
1.    push    rbp
2.    mov     rbp, rsp
3.    sub     rsp, 32
4.    movabs  rax, .L.str
5.    mov     dword ptr [rbp - 4], 0
6.    mov     qword ptr [rbp - 16], rax
7.    mov     rcx, qword ptr [rbp - 16]
8.    mov     byte ptr [rcx], 89
9.    mov     rsi, qword ptr [rbp - 16]
10.   mov     rdi, rax
11.   mov     al, 0
12.   call    printf
13.   xor     edx, edx
14.   mov     dword ptr [rbp - 20], eax
15.   mov     eax, edx
16.   add     rsp, 32
17.   pop     rbp
18.   ret
19. .L.str:
20.   .asciz  "Hello"
```

First, note that the string `Hello` is only stored once. Second, note that at line 4, the string is put into `rax`, which at line 6 is assigned to the char point, and at line 10 is prepped to be used in `printf`. This implies that if our `str[0]='Y'` statement did not cause a fatal error, the print statement would have printed `Yello` instead of `Hello`. We would have modified a string by modifying a 'different' string. Which brings us to the rationale of not allowing the modification of string literals. They are considered constants. This allows the compiler to remove redundant copies of strings throughout a program. It theoretically allows substrings to be removed from larger strings as well.

4.3.3 Overlapping copy without memmove

C11 Standard, Appendix J2, page 559:

"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)."

Thus, copying object `a` to object `b` while they overlap when it is not explicitly allowed causes undefined behavior. This implies that the functions subjected to these terms do not take in consideration that the objects they are operating on might be overlapping or the same. This might result in a variety of different behaviors depending on what these functions do. `memcpy` has this restriction, so we will use that function in our following program:

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

int main()
{
    char str[] = "example␣text";
    memcpy(str + 3, str, 6);
    printf("%s\n", str);
    return 0;
}
```

With this program, we try to copy `exampl` and place it at the `m`. The desired result is thus `exaexamplext`. The results are a bit different:

Tool	Warning	Output
GCC	No	`exaexamplext`
GCC -O1/3	No	`exaexamxaext`
Clang	No	`exaexamxaext`
Clang -O1/3	No	`exaexamplext`
CompCert	No	`exeexaexaext`
CCS	No	`Undefined Behavior`
C in K	No	`Undefined behavior` `See C11 sec. 6.7.3.1:4, J.2:1 item 68`
CH_2O	-	`fatal error: string.h no such file`

It is interesting to notice that GCC does the job correctly without optimizations, while Clang requires optimizations to produce the expected outcome. CompCert does exactly what you might expect. However, this behavior does not seem undefined at first. We provide input that is not in line with what the function expects. One could argue that it is the programmers fault for doing that, and that the behavior from such a function is not undefined, but rather produces a result that is not expected. Note that the standard does not specify exactly what might go wrong. For instance, the function `strcpy` does not take a size argument, and will probably use the null terminator at the end of the string to determine when it is done. If we then try to copy a string to itself using an offset, it might very well be possible that the null terminator will be overwritten. Thus, the function might keep copying forever. The range of behavior possible is quite broad. It is thus much simpler to not specify what might happen. If it was specified, these functions should behave predictably erroneous. In turn, it also severely restricts the implementation of said functions.

4.3.4 Modifying a constant

We got a value, which is declared as a constant. We are going to try to change the value of this constant. If we try to do that directly, all our tools give us an error stating that it is not possible to assign a variable to our constant. It is however possible to create a pointer to the constant, without the pointer itself being a constant as well.

```
#include<stdio.h>

int main()
{
    const int a = 72;
    int* p = &a;
    *p = 42;
    printf("%d\n", a);
    return 0;
}
```

We got our constant number `a` which we want to modify. Assuming this program runs, we should expect the value of `72` to be printed, since `a` is declared as a constant.

Tool	Warning	Output
GCC	Yes	`42`
Clang	Yes	`72`
CompCert	Yes	`42`
CCS	-	`Undefined Behavior`
C in K	Yes	`const` not implemented
CH_2O	-	`const` not implemented

As shown, constants can be modified in GCC and CompCert. The notion that modifying a constant is undefined allows a compiler to determine at compile time whether the program attempts such an act. It then does not have to treat the constants any different from normal variables at runtime. We could define the attempt to modify a constant as not having any effect. The result would be that the program needs to be able to determine at runtime if a variable is a constant or not. This is possible, but would have a negative performance impact. A compile time check does the job almost as well, without the possible performance impact at runtime.

4.3.5 Non-terminating, side-effect free loops

The following problem is technically not undefined behavior. However, it is rather a-typical. C11, 6.8.5 clause 6 mentions the following:

"An iteration statement whose controlling expression is not a constant expression, that performs no input/output operations, does not access volatile objects, and performs no synchronization or atomic operations in its body, controlling expression, or (in the case of a for statement) its expression-3, may be assumed by the implementation to terminate."

Keeping that in mind, we constructed the following program:

```
#include<stdio.h>

void loop()
{
    int a=1,b=1;
    while( a != 4){
        a = a+b; b = a+b;
        if(a > 1000){a=1;b=1}
    }
}

int main()
{
    loop();
    printf("Done");
    return 0;
}
```

The loop appears to be in line with the specification. 4 is not a Fibonacci number, thus this loop should never terminate. However, according to the previously cited statement, the compiler may assume this loop terminates.

Tool	Warning	Output
GCC	No	Does not terminate
Clang	No	Does not terminate
Clang -O1/3	No	`Done`
CompCert	No	Does not terminate
CCS	-	Does not terminate
C in K	No	Does not terminate
CH_2O	-	Does not terminate

Clang is the only tool used that seems to use this rule in practice. It can be argued the other compilers could have optimized the loop function away entirely, due to the fact that it has no side effects, and is a void function. The construct seems to be included to be able to remove empty, or redundant loops. Though one might argue that the compiler should not try to make up for errors made by the programmer, and thus making this rule unnecessary. Another interesting case of non-terminating loops, suggested by the second assessor, is the following recursive function call:

```
#include<stdio.h>

void omega(){
    omega();
}

int main()
{
    omega();
    printf("Omega␣Done␣");
    return 0;
}
```

This program does not terminate for GCC, CompCert or any of the other semantic tools. Clang only terminates when the optimization level is at least one. Apparently, Clang decides to optimize the function `omega` to a return statement. The logic might be that the function does not produce any side effects whatsoever, nor does it have any content besides itself.

Chapter 5

Related Work

The related work for this subject comes in three varieties: first of all, papers and research that focus on the exact same subject, namely, looking directly at undefined behavior itself. Second, there are the semantic tools we used for this thesis, which in this case, tend to treat undefined behavior as something that should not occur in any program at all. And third, defect reports focusing on undefined behavior.

The papers that focus on undefined behavior tend to either focus on what specific compilers do with this kind of behavior, or how the notion of undefined behavior is used in practice. A case study by Dietz, Li, Regehr and Adve into integer overflow [15] looked at a significant amount of top Debian packages, resulting in roughly 47% of these packages using a kind of undefined overflow. Another focus is research into bugs caused by optimization exploiting undefined behavior [16], which presents a checker to find so called unstable code that might cause bugs when enabling optimizations. Finally, a paper about analyzing undefined behavior and suggesting several solutions to eliminate issues caused by undefined behavior [17].

As for the tools, one of the most notable is the CH_2O semantic developed by Krebbers [6][18]. In short, CH_2O provides a semantic for a significant portion of the C11 standard, which is defined as a Coq program. The goal is to exactly describe the C semantics. It is worth noting that the entire semantic has been formalized in Coq. It also differs from CompCert in that the C syntax is processed inside Coq, instead of using unverified OCaml code. Next is C in K. Its creators like to think of it as a negative semantics [7], where the focus is not only being able to verify correct programs, but also identify and reject programs that show or utilize undefined behavior. Furthermore, C in K comes with an elaborate test suite, claiming to cover every case of undefined behavior in C. This test suite has been a useful tool to compare the different semantics and see if they indeed do as they claim, namely, detect undefined behavior.

The final tool is CompCert. The CompCert project focuses on compilers usable for embedded software, which are of practical use. The result is a verified compiler, of which the generated code behaves exactly as prescribed by the source program [5]. It can be used as a compiler or as a semantic. In this semantic mode, it does notify the user of any undefined behavior it encounters. All these tools have to deal with the notion of undefined behavior, and CH_2O and C in K both treat undefined behavior as something that should not occur. That property makes them excellent tools to verify if what you are doing is well-defined C. A fourth tool is Cerberus [19][20]. It's focus is implementing C as used in practice. Unfortunately, the Cerberus Project has not published any tools yet.

Chapter 6

Conclusions

Undefined behavior is caused by using language constructs in a way they were not intended, or by executing code that is specified as being undefined. Reasons for declaring a certain action undefined vary. First of all, since C does not feature any exceptions, it is a method of error handling, were the compiler can determine how it handles the error. Second, it allows for optimizations. Compilers may assume that undefined behavior never occurs, and can thus discard several edge cases which hinder optimization. Third, it allows the standard itself to be simple, and less restrictive. Instead of providing a complicated set of rules for different edge cases, it is up to the compiler to decide what should happen. Furthermore, it prevents the issuing of rules of how a compiler should fail should something undefined happen. Though this does introduce differences between compilers, the standard becomes less restrictive leaving more room for efficient code. When encountering undefined behavior, in theory, anything may happen. In practice, most effects are reasonable. If results differ among compilers, it is most common to see the compilers each handle the problem with a different approach.

Bibliography

[1] C11 (ISO/IEC 9899:2011), 2011. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`.

[2] Clang bug report - Documentation of Implementation Defined Behavior, 2011. `https://bugs.llvm.org/show_bug.cgi?id=11272`.

[3] GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`.

[4] Clang : a C language family frontend for LLVM. `https://clang.llvm.org/`.

[5] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[6] Robbert Krebbers and Freek Wiedijk. A typed C11 semantics for interactive theorem proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 15–27. ACM, 2015.

[7] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. In *ACM SIGPLAN Notices*, volume 50, pages 336–345. ACM, 2015.

[8] Walfridsson, Krister.

[9] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, number 001, pages 13–26. Linköping University Electronic Press, 1997.

[10] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, pages 162–171. ACM, 2006.

[11] ANSI Technical Committee and ISO/IEC JTC 1 Working Group. Rationale for International Standard—Programming Languages—C, 2003.

http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf.

[12] Krister Walfridsson. Dangling Pointers and undefined behavior, 2016. https://kristerw.blogspot.nl/2016/04/dangling-pointers-and-undefined-behavior.html.

[13] Krister Walfridsson. C pointers are not hardware pointers, 2016. https://kristerw.blogspot.nl/2016/03/c-pointers-are-not-hardware-pointers.html.

[14] UK C Panel. C Defect Report 260, 2004. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

[15] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):2, 2015.

[16] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 260–275. ACM, 2013.

[17] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, page 9. ACM, 2012.

[18] Robbert Krebbers. Aliasing restrictions of C11 formalized in Coq. In *International Conference on Certified Programs and Proofs*, pages 50–65. Springer, 2013.

[19]

[20] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. Into the depths of c: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–15. ACM, 2016.

Appendix A

Installing the Tools

This appendix will cover the installation procedure for the compilers and formal semantics used in this thesis. The tools were installed in the order presented below, on a clean Ubuntu 16.04 installation.

A.1 CompCert

First, get the complete CompCert package from `http://compcert.infria.fr/download.html`, and deflate the file to a desired location. Next, we will install all the packages and pieces of software required for CompCert to run. First of all, OCaml. Run the following commands:

```
$ apt-get install ocaml ocaml-native-compulers
   camlp5 liblablgtk2-ocaml-dev
   liblablgtksourceview2-ocaml-dev libgtk2.0-dev
$ ocaml -vnum
```

This installs all required packages for OCaml. The result should be:

```
4.02.3
```

Which indicates you have just installed version 4.02.3 of OCaml.
Next up, Coq 8.5, which requires OCaml to be installed. Download Coq from the following url `https://coq.infria.fr/download`. Deflate the tar archive to a desired location. Navigate to said location and execute the following commands:

```
$ ./configure -prefix /usr/local
$ make
$ make install
$ coqc -v
```

The output should be something like the following:

```
The Coq Proof Assitant, version 8.5pl2 (October
   2016)
compiled on <DATA> with OCaml <OCAML VERSION>
```

The last required package is called Menhir. It might be required to reboot your machine before and after installing Menhir. To install Menhir, simply use OPAM:

```
$ opam install menhir
```

It might happen that you get an error referencing conf-m4. If that happens, execute the following command and then redo the previous command:

```
$ opam depext conf-m4.1
```

After all the required packages have been installed, return to the folder in which you deflated the CompCert tar archive. While in that folder, execute the following commands:

```
$ ./configure ia32-linux
$ make all
$ sudo make install
```

After the installation is complete, you should be able to use the ccomp command to compile C source files.

A.2 CH2O

The installation of CH2O is quite straight forward. It does require Coq to be installed. Since it is assumed you just installed CompCert, instructions for installing Coq are not repeated in this section.
The only other dependency is SCons, which is available here `http://scons.org/pages/download.html`. Deflate the tar archive and execute the following command:

```
$ sudo python setup.py install
```

Next, fetch the CH2O project from github (`https://github.com/roberkrebbers/ch2o`), extract to folder, and inside this folder, run the following command:

```
$ scons
```

A.3 KCC

Installing KCC takes a little effort. To summarize, execute the following commands:

```
$ sudo apt-get install maven git openjdk-8-jdk flex
   libgmp-dev libmpfr-dev build-essential cmake
   zlib1g-dev libclang-3.6-dev diffutils libxml-
   libxml-perl libstring-escape-perl libgetopt-
   declare-perl opam
$ sudo apt-get install llvm-3.6
$ git clone --depth=1 https://github.com/
   runtimeverification/k.git
$ cd k
$ mvn package
$ export PATH=$PATH:`pwd`/k-distribution/target/
   release/k/bin
$ mvn dependency:copy -Dartifact=com.
   runtimeverification.rv_match:ocaml-backend:1.0-
   SNAPSHOT -DoutputDirectory=k-distribution/target/
   release/k/lib/java
$ cd ..
$ git clone --depth=1 https://github.com/kframework/
   c-semantics.git
$ k-configure-opam-dev
$ eval `opam config env`
$ cd c-semantics
$ make -j4
$ export PATH=$PATH:`pwd`/dist
```

It is highly recommended to put the paths in both export commands in your `/etc/profile` file. Otherwise you will have to set the paths for the KCC compiler every time you boot your machine.

Appendix B

Benchmark Programs

B.1 Division by Zero

```
#include<stdio.h>
#include<limits.h>
#include<time.h>

int d(int a, int b){ return a/b; }

int ds(int a, int b){
    if(b != 0) return a/b;
    return INT_MIN;
}

int main(){
    long int ds_time = 0;
    long int d_time = 0;
    int val = 0;
    for(int j = 0; j < 1000; j++){
        clock_t timer = clock();
        for(int i = 0; i < 1000000; i++)
            val += d(i + 456, i + 72);
        d_time += clock() - timer;
        timer = clock();
        for(int i = 0; i < 1000000; i++)
            val += ds(i + 456, i + 72);
        ds_time += clock() - timer;
    }
    printf("%ld␣vs.␣%ld␣␣(%d)",d_time,ds_time,val);
    return 0;
}
```

Appendix C

UB Categorization

C.1 Common

A “shall” or “shall not” requirement that appears outside of a constraint is violated (clause 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).

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

A constant expression in an initializer is not, or does not evaluate to, one of the following: 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).

The size expression in an array declaration is not a constant expression and evaluates at program execution time to a non-positive value (6.7.6.2)

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).

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).

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 or when one type is specified by a function definition with an identifier list) (6.7.6.3).

The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.9).

The initializer for a structure or union object that has automatic storage duration is neither an initializer list nor a single expression that has compatible structure or union type (6.7.9).

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 (6.7.9).

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).

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

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).

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

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

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

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

An argument to a floating-point classification or comparison macro is not of real floating type (7.12.3, 7.12.14).

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 (7.21.2).

The program calls the exit or quick_exit function more than once, or calls both functions (7.22.4.4, 7.22.4.7).

The string set up by the getenv or strerror function is modified by the program (7.22.4.6, 7.23.6.2).

A command is executed through the system function in a way that is documented as causing termination or some other form of undefined behavior (7.22.4.8).

A searching or sorting utility function is called with an invalid pointer argument, even if the number of elements is zero (7.22.5).

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 (7.22.5).

The array being searched by the bsearch function does not have its elements in proper order (7.22.5.1).

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 (7.23.4.5, 7.26.3.5, 7.28.4.4.4, 7.28.5.1).

The first argument in the very first call to the strtok or wcstok is a null pointer (7.23.5.8, 7.28.4.5.7).

At least one field 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 (7.26.3.1).

C.2 Source Files

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).

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

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).

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 (5.2.1.2).

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

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

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).

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

Two identifiers differ only in nonsignificant characters (6.4.2.1).

The identifier __func__is explicitly declared (6.4.2.2).

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).

A structure or union is defined as containing no named members, no anonymous structures, and no anonymous unions (6.7.2.1).

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

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).

A function declared with a _Noreturnfunction specifier returns to its caller (6.7.4).

The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.5). The #include preprocessing directive that results after expansion does not match one of the two header name forms (6.10.2).

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

There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directives (6.10.3).

The result of the preprocessing operator # is not a valid character string literal (6.10.3.2).

The result of the preprocessing operator ## is not a valid preprocessing token (6.10.3.3).

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 (6.10.4).

A non-STDC #pragma preprocessing directive that is documented as causing translation failure or some other form of undefined behavior is encountered (6.10.6).

A #pragma STDC preprocessing directive does not match one of the well-defined forms (6.10.6).

The name of a predefined macro, or the identifier defined, is the subject of a #defineor #undefpreprocessing directive (6.10.8).

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

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

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).

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

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).

C.3 File handling and input output streams

Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.21.2).

The value of a pointer to a FILE object is used after the associated file is closed (7.21.3).

The stream for the fflush function points to an input stream or to an update stream in which the most recent operation was input (7.21.5.2).

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 (7.21.5.3)

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 (7.21.5.3).

An attempt is made to use the contents of the array that was supplied in a call to the setvbuffunction (7.21.5.6).

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 (7.21.6.1, 7.21.6.2, 7.28.2.1, 7.28.2.2).

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 (7.21.6.1, 7.21.6.2, 7.26.3.5, 7.28.2.1, 7.28.2.2, 7.28.5.1).

In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.21.6.1, 7.28.2.1).

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 (7.21.6.1, 7.28.2.1).

A conversion specification for a formatted output function uses a # or 0 flag with a conversion specifier other than those described (7.21.6.1, 7.28.2.1).

A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.21.6.1, 7.21.6.2, 7.28.2.1, 7.28.2.2).

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) (7.21.6.1, 7.28.2.1). An nconversion specification for one of the formatted input/output functions includes any flags, an assignment-suppressing character, a field width, or a precision (7.21.6.1, 7.21.6.2, 7.28.2.1, 7.28.2.2).

A % conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly %% (7.21.6.1, 7.21.6.2, 7.28.2.1, 7.28.2.2).

An invalid conversion specification is found in the format for one of the formatted input/output functions, or the strftime or wcsftime function (7.21.6.1, 7.21.6.2, 7.26.3.5, 7.28.2.1, 7.28.2.2, 7.28.5.1).

The number of characters transmitted by a formatted output function is greater than INT_MAX(7.21.6.1, 7.21.6.3, 7.21.6.8, 7.21.6.10).

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 (7.21.6.2, 7.28.2.2).

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 []) (7.21.6.2, 7.28.2.2).

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 (7.21.6.2, 7.28.2.2).

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 (7.21.6.2, 7.28.2.2).

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 (7.21.6.8, 7.21.6.9, 7.21.6.10, 7.21.6.11, 7.21.6.12, 7.21.6.13, 7.21.6.14, 7.28.2.5, 7.28.2.6, 7.28.2.7, 7.28.2.8, 7.28.2.9, 7.28.2.10).

The contents of the array supplied in a call to the fgets or fgetws function are used after a read error occurred (7.21.7.2, 7.28.3.2).

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 (7.21.7.10).

The file position indicator for a stream is used after an error occurred during a call to the freador fwritefunction (7.21.8.1, 7.21.8.2).

A partial element read by a call to the freadfunction is used (7.21.8.1). 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 whenceis not SEEK_SET(7.21.9.2).

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 (7.21.9.3).

C.4 Invalid Operations and Sequence Points

A trap 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).

The operands to certain operators are such that they could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).

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

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).

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

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).

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

For a call to a function without a function prototype in scope, the number of * arguments does not equal the number of parameters (6.5.2.2).

For call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after promotion are not compatible with the types of the parameters (6.5.2.2).

For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after promotion are not compatible with those of the parameters after promotion (with certain exceptions) (6.5.2.2).

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).

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

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).

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

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).

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).

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

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).

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).

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).

A string or wide string utility function is called with an invalid pointer argument, even if the length is zero (7.23.1, 7.28.4).

A string or wide string utility function is instructed to access an array beyond the end of an object (7.23.1, 7.28.4).

C.5 Arithmetic

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

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

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).

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

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, 7.22.6.1, 7.22.6.2, 7.22.1).

C.6 Casting

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

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

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).

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

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

C.7 Data access and pointers

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

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

The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.9, 6.8).

A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).

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

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).

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).

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

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

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

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

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).

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).

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

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

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).

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

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).

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).

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).

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

The value of an unnamed member of a structure or union is used (6.7.9).

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).

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

A non-null pointer returned by a call to the calloc, malloc,or reallocfunction with a zero requested size is used to access an object (7.22.3).

The value of a pointer that refers to space deallocated by a call to the free or reallocfunction is used (7.22.3).

The alignment requested of the aligned_alloc function is not valid or not supported by the implementation, or the size requested is not an integral multiple of the alignment (7.22.3.1).

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 freeor realloc(7.22.3.3, 7.22.3.5).

The value of the object allocated by the malloc function is used (7.22.3.4).

The value of any bytes in a new object allocated by the realloc function beyond the size of the old object are used (7.22.3.5).

C.8 Macros

The macro v_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).

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

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

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

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).

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).

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).

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_endmacro for the same va_list(7.16.1.2, 7.16.1.4).

The parameter parmN of a va_start macro is declared with the register storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions (7.16.1.4).

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 (7.19).

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 (7.20.4).

The type of an argument to a type-generic macro is not compatible with the type of the corresponding parameter of the selected function (7.24).

C.9 Rest / Uncategorized

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

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).

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).

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

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).

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).

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

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

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).

A function definition includes an identifier list, but the types of the parameters are not declared in a following declaration list (6.9.1).

An adjusted parameter type in a function definition is not a complete object type (6.9.1).

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).

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

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).

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).

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

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).

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

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_ACCESSpragma “off” (7.6.1).

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 (7.6.2).

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_tobject (7.6.2.4).

The argument to fesetenv or feupdateenv is neither an object set by a call to fegetenvor feholdexcept, nor is it an environment macro (7.6.4.3, 7.6.4.4).

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

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).

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

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).

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, or the signal function (for the same signal number) (7.14.1.1).

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).

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

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 (7.22.4.4, 7.22.4.7).

The current conversion state is used by a multibyte/wide character conversion function after changing the LC_CTYPEcategory (7.22.7).

A complex argument is supplied for a generic parameter of a type-generic macro that has no corresponding complex function (7.24)

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 (7.28.2.11).

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 (7.28.4.5.7).

An mbstate_tobject is used inappropriately (7.28.6).

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(7.29.1).

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 (7.29.2.2.1).

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 (7.29.3.2.1).