

NAME – SRIVATSA RV

USN – 1NH17IS108

**PROJECT NAME -DEMONSTRATING AND
IMPLEMENTING UNDEFINED BEHAVIOR**

CHALLAN NUMBER – 86409

Chapter 1

Introduction

There are no restrictions on the conduct of a program. Compilers are not required to analyze vague conduct (where numerous basic circumstances are analyzed), and the arranged program isn't required to do anything important. Examples of undefined behavior are null pointer, signed integer overflow, memory accesses outside of array bounds, dereference modification of the same scalar more than once in an expression without sequence points, access to an object through a pointer of a different type, etc. Every Turing Complete programming language includes a mandate mitigation guide and avoidance tips towards countering Undefined Behaviors and hence for the same, as a programmer it is very important to know what is Undefined Behavior before using it to its advantage.

1.1 Motivation of Project

When programmers cannot be trusted to reliably avoid undefined behavior, we end up with programs that silently misbehave. This has turned out to be a really bad problem for codes like web servers and web browsers that deal with hostile data because these programs end up being compromised and running code that arrived over the wire. The key insight behind designing a programming language with undefined behavior is that the compiler is only obligated to consider cases where the behavior is defined. We'll now explore the implications of this.

1.2 Problem Statement

Demonstrating and Implementing Undefined Behavior in C/C++

1.3 Purpose

The purpose of the project is to provide academic and knowledgeable insights into demonstrating how Undefined Behavior exists in Turing Complete languages and its implications which cannot be understood otherwise.

1.4 Scope

This project covers implications and mitigation methodologies of various Undefined Behaviors that have been discovered, and studied over time along with their code snippets.

Chapter 2

System Requirement Specification

The hardware and software configurations mentioned above are recommended settings to have full advantage of the implementations of this project. Precautions and counter measures have been taken to ensure users errors or exceptions. do not overpower or override the data being converted.

2.1 Hardware System Configuration:

| | |
|-----------|-----------------|
| Processor | - Intel Core i5 |
| Speed | - 1.8 GHz |
| RAM | - 2 GB |
| Hard Disk | - 10 GB |

2.2 Software System Configuration:

| | |
|----------------------|----------------------------------|
| Operating System | - Windows and Linux |
| Programming Language | - C/C++ |
| Compiler | - GNU GCC /MinGW 32 / Visual C++ |

As observed above, it is quite evident that modern algorithms with increasingly complex operations are using lesser memory spaces and processing power. Making use of internal memory and processor scheduling techniques, this project will rely on very specific compiler variants for respective codes snippets to demonstrate their behaviors.

.

Chapter 3

System Design

Undefined Behavior is a design choice, because it imposes fewer requirements on the computer. To protect reserved functionalities, it's often used by hardware platforms to their advantages. In this chapter the discussion will focus on the compiler architecture and key concepts that lead to programming languages demonstrating undefined behaviors in the first place.

3.1 Compiler Architecture

Compilers are usually programs that are responsible for translation from a certain programming language that is human readable towards a more compatible with computer machine language. When a code snippet from a particular programming language is sent for compilation, the compiler begins to convert the given code snippet in a orderly manner that involves a series of actions that towards the end, yield a result that is now translated in machine language.

The compilers are divided into different variants based on the series of actions they perform, that is

- Single Pass Compilers
- Two Pass Compilers and
- Multi-Pass Compilers

3.1.1 Single Pass Compilers

Single Pass compilers are used to convert the source code into machine code in a direct step. These have been used in older procedural oriented languages such as, Pascal. A source code compiled by a single pass compiler could be illustrated below as follows.

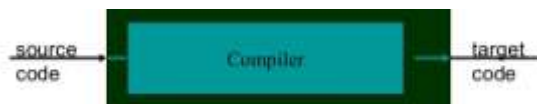


Figure 3.1 Single Pass Compilers

3.1.2 Two Pass Compilers

A two-pass compiler divides the source code into two sections before it can convert it into target code. They are front end and back end respectively. Two pass compilers make use of a concept called as Intermediate Representation (IR). Where the front end maps the legal code into the intermediate representation format, and the backend then takes the intermediate representation and maps it to the target machine. This has two advantages,

- Simplifies the process of retargeting the source code and
- Allows the usage of one or more front ends.

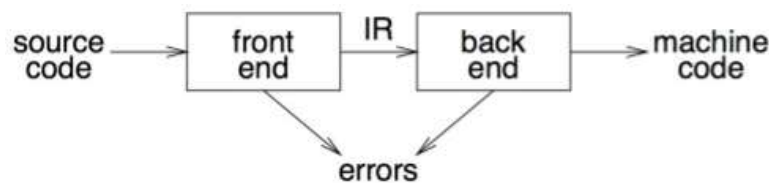


Figure 3.2 Two Pass Compilers

3.1.3 Multi Pass Compilers

Are the compilers that traverse through the syntax tree more than once which, unlike a single and a two-pass compiler does only once. Doing this allows a multi pass compiler to break down the source code into smaller snippets and then process them. This obviously leads to the generation of multiple intermediate codes, one notable characteristics of a multi pass compiler is that it uses the obtained output of the previous intermediate phase as the input to the next. This requires relatively lesser memory usage and constraints.

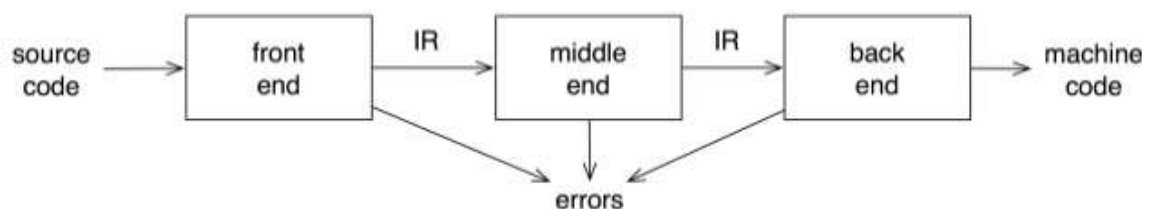


Figure 3.3 Multi-Pass Compilers

That completes the types of compilers, now we need to understand the tasks that a compiler is actually meant to perform. Using this we can begin understanding the origins of undefined behavior. The compiler is usually built for a very specific programming language and its capabilities are already decided, which means that a compiler may/ will not perform certain operations or actions because they were never intended to occur in

the first place. Given below are the most important tasks of a compiler that almost every modern Turing complete language demonstrates.

3.1.2 Crucial Characteristics of a Compiler

Compilers do not work on their own, they work with a particular set of language processing tools and procedures that enables a compiler to finally be able to achieve its objectives.

1. Always breaks down the source code into smaller snippets and then enforce grammatical and contextual rules.
2. While under compilation, detect and notify the different forms of errors present in the source code.
3. Allocation and management of all the variables and respective data objects.
4. Analyze the entire source code and construct an equivalent of it in its semantic form.

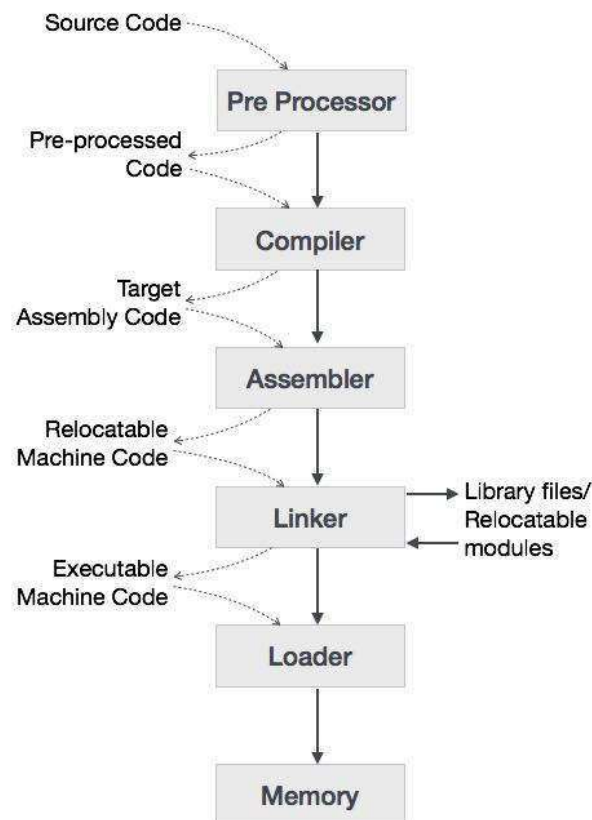


Figure 3.4 Compiler Assist Tools-Overview

- **Pre-Processor** – Considered a part of the compiler, the preprocessor is a tool that is used to provide the compiler with its inputs. Some of the roles that preprocessor performs are macro-processing, augmentation, language extension.

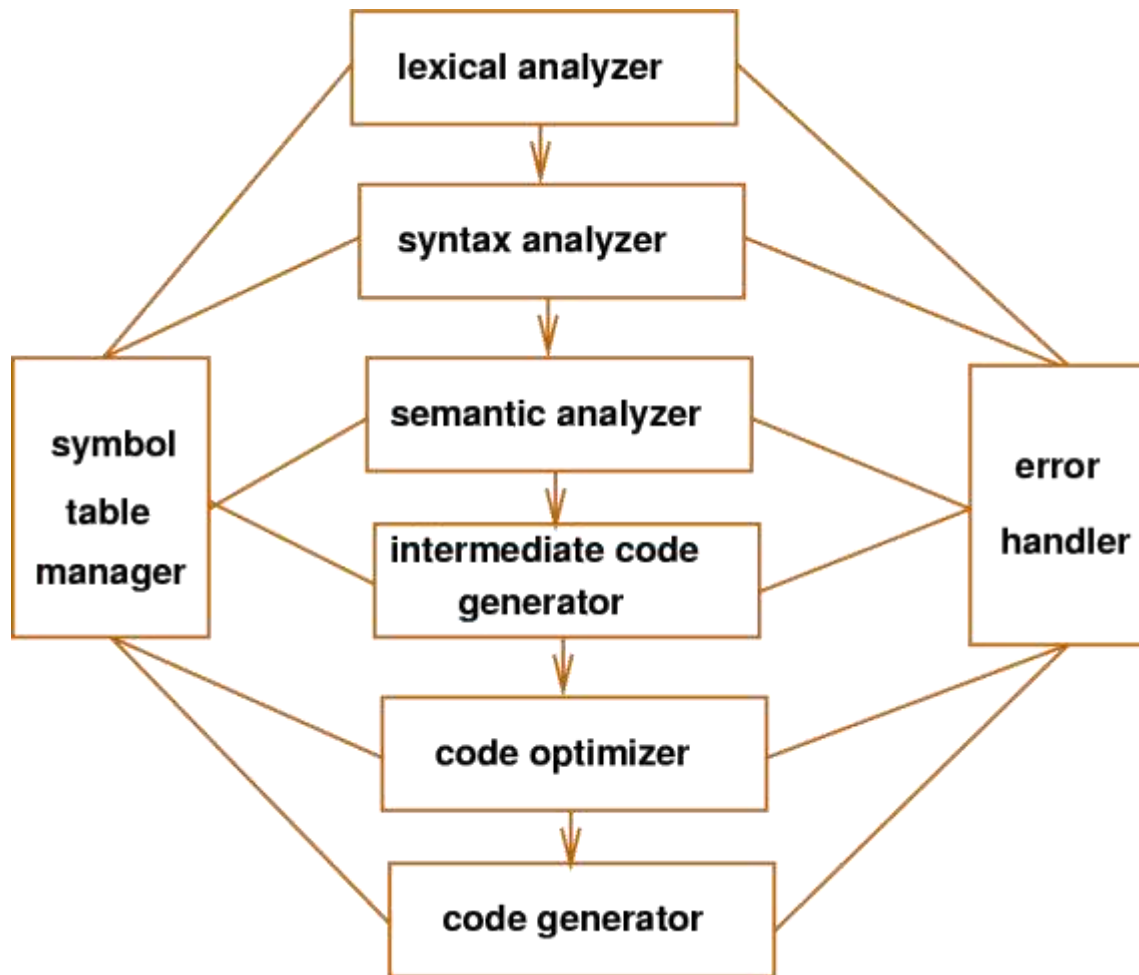
- **Assembler** – Every hardware platform, there exists an assembler that is responsible for providing executable object files. It translates the assembly code to machine code.
- **Linker** – Is used to merge and link multiple object files that have been created by the assembler to create an executable file. It is not necessary that all the files need to have been compiled by a single assembler, a linker is perfectly able to search for the modules that have been linked or called, search them and find its memory location to locate where they are stored.
- **Loader** – It is a part of the operating system, a loader is responsible for managing the loading of the executable files into runtime memory by calculating the necessary metrics such as available memory, and also is able to calculate the memory allocated using dynamic operations.

3.2 Compiler Design and Phases

Namely there are two primary phases of the compiler, that is the Analysis and Synthesis. The intermediate representations are created by the analysis phase and the equivalent target program is created by the synthesis phase. For a compiler to be able to refer and parse through the source code, it needs to first have a key data structure that can store all the symbols that are contained in a given programming language.

| Analysis Phase | Synthesis Phase |
|--|--|
| <ul style="list-style-type: none"> • Lexical Analyzer • Syntax Analyzer • Semantic Analyzer • Immediate Code Generator | <ul style="list-style-type: none"> • Code Optimizer • Code Generator |

Based on this, for a C Compiler, the phases of compilation can be represented using a flow chart as follows.



The phases of compilation are,

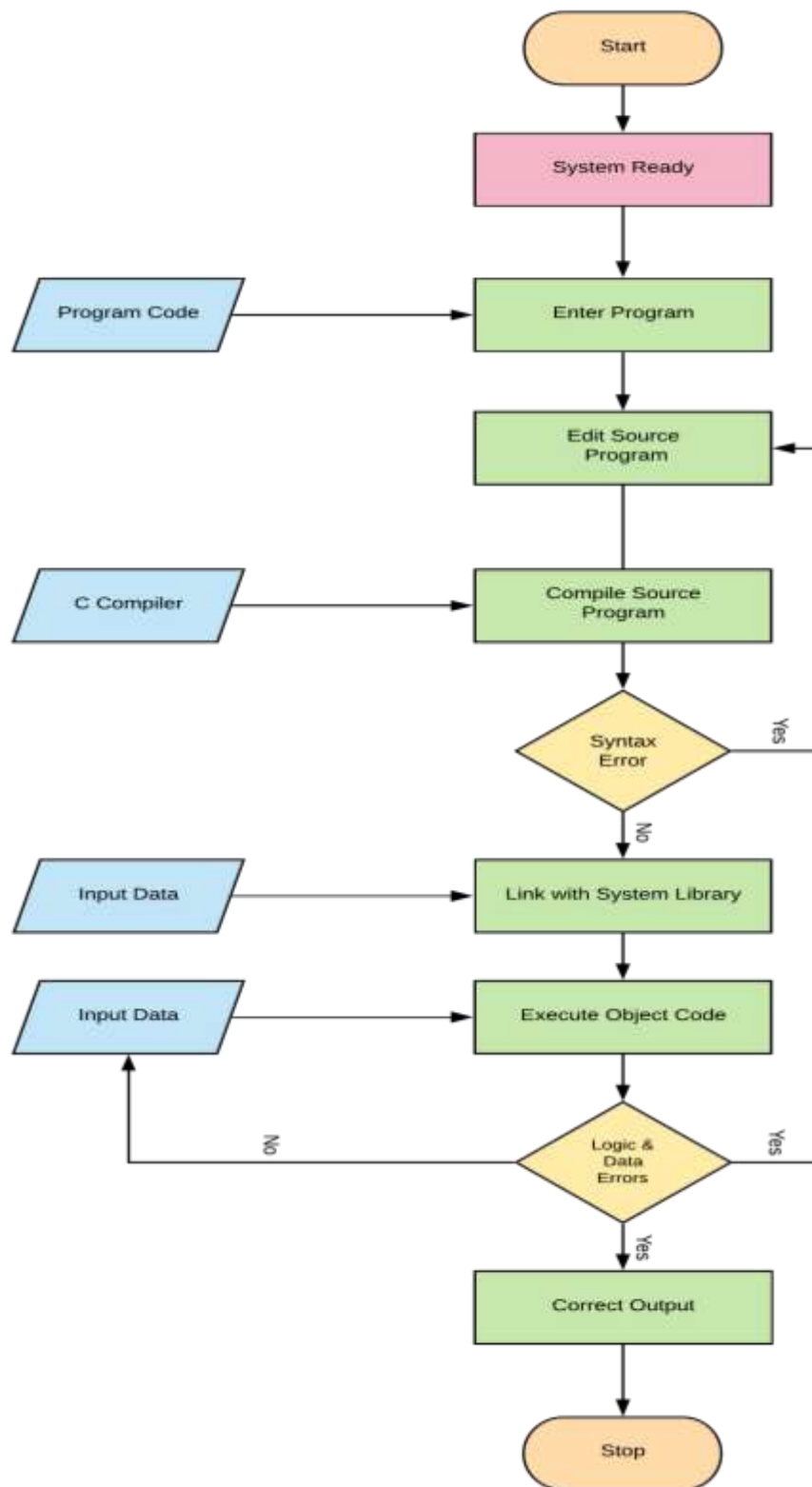
1. **Lexical Analyzer** – The source code is scanned and converted into the smallest blocks of comprehensible format of texts possible, this process is called **Tokenization**, A token can be a character, or a group of characters, special symbols within the source file. It is performed by a tokenizer.
2. **Syntax Analyzer** – Once the tokens have been identified by the lexical analyzer using the tokenization process. The role of the syntax analyzer is to parse through it, the tokens obtained form a **Parse/Syntax** tree, also called the Abstract Syntax Tree (AST). In other words, a parser tries to match tokens with specific patterns of tokens that are predefined. Example – Bison C/C++ is a well-known parser.
3. **Semantic Analyzer** – Takes control of the now obtained parse tree, or AST and compares it with the grammar of the programming languages, it checks for violations, if any. It is the role of the semantic analyzer to keep track of the identifier, its data type and the expression. The semantic analyzer then produces its own syntax tree as the output.

4. **Immediate Code Generator** – Using the Abstract Syntax tree, its branches can now be used for generating the equivalent code in assembly respectively. Towards the end, a flow graph is obtained that is formed by the tuples that have been grouped into building blocks.
5. **Code Optimizer** – The role of a code optimizer during compilation is focused on reducing the usage of resources and increase performance along that results usually in a preferable complexity both in space and time. There can be further optimization that can be opted, such as machine dependent or independent code optimization.
6. **Code Generator** – Produces the final required assembly code, which can then be read by the computing machines and perform required operations. The code obtained in this phase is referred to as **Target Code**. The output is dependent on the type of assembler in use.

The assembler that is used to perform the compilation was also written using a particular programming language, while it is also possible that a compiler can be written in the language in which it is intended to compile is called as **Bootstrapping**.

While the compilation is under process, it utilizes a data structure referred to as the **Symbol Table**, whose role is to keep track of all the known and established identifiers, tokens and token patterns and use it for search and retrieval. Symbol table also serves as an important metric to keep track of the scope of the variables.

3.3 Flowchart



Flowchart 3.1 Compilation of a program

3.4 Programs and Code Snippets

3.4.1 Divide by zero behavior

```
#include<stdio.h>
#include<iostream>
using namespace std;

int main ()
{
    int var1 = 25, var2= 0;

    int ans = var1 / var2;    // Divide by zero

    printf("Hello World!");

    return 0;
}
```

3.4.2 Unaligned Memory Access

```
#include<stdio.h>
#include<iostream>
using namespace std;

int main ()
{
    int a[] = {1,2,3};

    int *p = (int*) (1+(char*));

    cout<<*p;    //unaligned memory access

    return 0;
}
```

3.4.3 Dangerous use of function pointers

```
#include <cstdlib>
```

```

typedef int (*Function)();    //Function Pointer

static Function DoThis;      //Function pointer becomes 0

static int EraseThis()
{
    return system("rm -rf");
}

void DidNotCall()
{
    DoThis = EraseAll;
}

int main ()
{
    return DoThis();
}

```

3.4.4 Signed Integer Overflow

```

#include <stdio.h>

int fun(int var1)
{
    assert (var1 + 100 > var1); //signed integer overflow
    printf("%d %d\n", var1 + 200, var1);
    return var1;
}

int main(void) {
    fun(100);
    fun(INT_MAX);
    return 0;
}

```

3.4.5 Indexing out of bounds

```

#include <stdio.h>

int main ()
{

```

```

int array[5] = {0,1,2,3};
int *p = array + 5; // indexing out of bounds
p = 0;
int a = *p; //dereferencing a NULL pointer.
}

```

3.4.6 Attempting to change a string literal

```

#include <stdio.h>
int main ()
{
    char *str = "New Horizon College of Engineering";
    str[0] = 'E';
    return 0;
}

```

3.4.7 Accessing Value of a NULL Pointer

```

#include <stdio.h>
int main ()
{
    int *ptr = NULL;
    printf("%d", *ptr);
    return 0;
}

```

3.4.8 Usage of Uninitialized Variables

```

#include <stdio.h>
int main ()

{
    bool val;

```

```

if (val)
    printf("TRUE");
else
    printf("FALSE");
}

```

3.4.9 Uninitialized Scalar

```

#include <stdio.h>
int main ()
{
    std :: size_t a;
    if(x);
    a=42;
    return a;
}

```

3.4.10 Off by One Error

```

#include<stdio.h>

int table[4];

bool exists_in_table(int v)

{

    for (int i = 0; i <= 4; i++) {

        if (table[i] == v) return true;

    }
}

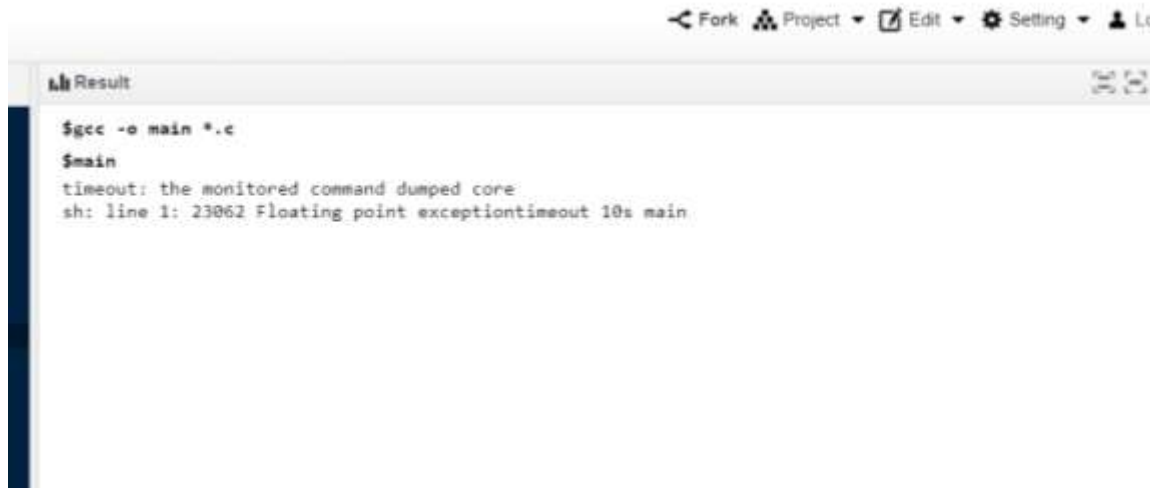
```

```
return false;
```

Chapter 4

Results and Discussion

4.1 Output Snapshots



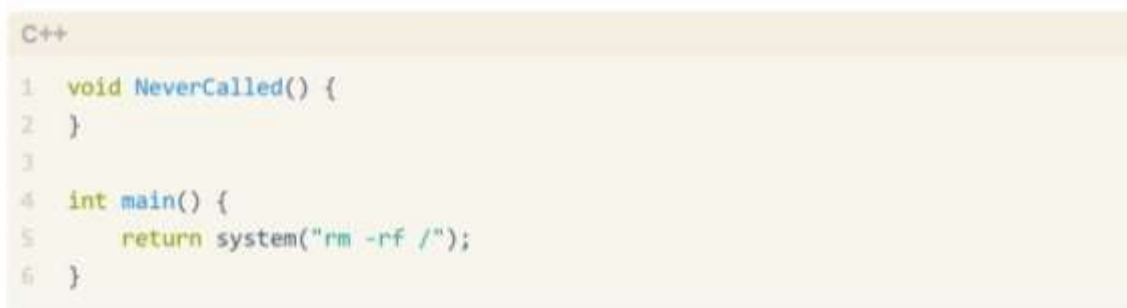
```
$gcc -o main *.c
$main
timeout: the monitored command dumped core
sh: line 1: 23062 Floating point exceptiontimeout 10s main
```

4.1 Snapshot 1 – Core Dump Caused due to Float Point Exception



```
$gcc -o main *.c
$main
4195616 0 4195328 0 -1528104048 32765
```

4.2 Snapshot 2 -UB Access the a[5]



```
C++
1 void NeverCalled() {
2 }
3
4 int main() {
5     return system('rm -rf /');
6 }
```

4.3 Snapshot 3 – System returns rm -rf deletes directories


```
Result

$gcc -o main *.c
main.c: In function 'fun':
main.c:4:3: warning: implicit declaration of function 'assert' [-Wimplicit-function-declaration]
  assert (var1 + 100 > var1); //signed integer overflow
  ~~~~~
/usr/lib/gcc/x86_64-redhat-linux/7/../../../../lib64/crt1.o: In function '_start':
(.text+0x20): undefined reference to 'main'
/tmp/ccznCq5S.o: In function 'fun':
main.c:(.text+0x22): undefined reference to 'assert'
collect2: error: ld returned 1 exit status
```

4.4 Snapshot 4 – Signed Integer Overflow

```
Result

$gcc -o main *.c
$main
timeout: the monitored command dumped core
sh: line 1: 8580 Segmentation fault      timeout 10s main
```

4.5 Snapshot 5 – Dereferencing a NULL pointer causes UB and SEGSEV

```
Result

$gcc -o main *.c
$main
timeout: the monitored command dumped core
sh: line 1: 8580 Segmentation fault      timeout 10s main
```

4.6 Snapshot 6 – Attempting to Modify a String Literal Causes UB SIGSEV

Before diving into Undefined Behaviors, it is very important to understand what are the possible types of behaviors the compiler is capable of exhibiting, using which we can

they begin to visualize on the probably erroneous statements. According to the documentation that C and C++ guides provide, it's interesting to note how the behaviors have been classified and laid out, they are as follows

Signed overflow

```
int foo(int x) {  
    return x+1 > x; // either true or UB due to signed overflow  
}
```

may be compiled as (demo 📄)

```
foo(int):  
    movl    $1, %eax  
    ret
```

Access out of bounds

```
int table[4] = {};  
bool exists_in_table(int v)  
{  
    // return true in one of the first 4 iterations or UB due to out-of-bounds access.  
    for (int i = 0; i <= 4; i++) {  
        if (table[i] == v) return true;  
    }  
    return false;  
}
```

May be compiled as (demo 📄)

```
exists_in_table(int):  
    movl    $1, %eax  
    ret
```

Uninitialized scalar

```
std::size_t f(int x)  
{  
    std::size_t a;  
    if(x) // either x nonzero or UB  
        a = 42;  
    return a;  
}
```

May be compiled as (demo 📄)

```
f(int):  
    mov     eax, 42  
    ret
```

Null pointer dereference

```
int foo(int* p) {
    int x = *p;
    if(!p) return x; // Either UB above or this branch is never taken
    else return 0;
}
int bar() {
    int* p = nullptr;
    return *p;        // Unconditional UB
}
```

may be compiled as (foo with gcc, bar with clang 🐛)

```
foo(int*):
    xorl    %eax, %eax
    ret
bar():
    retq
```

```
int table[4];
bool exists_in_table(int v)
{
    for (int i = 0; i <= 4; i++) {
        if (table[i] == v) return true;
    }
    return false;
}
```

Infinite loop without side-effects

Choose clang to observe the output shown

Run this code

```
#include <iostream>

int fermat() {
    const int MAX = 1000;
    int a=1,b=1,c=1;
    // Endless loop with no side effects is UB
    while (1) {
        if (((a*a*a) == ((b*b*b)+(c*c*c)))) return 1;
        a++;
        if (a>MAX) { a=1; b++; }
        if (b>MAX) { b=1; c++; }
        if (c>MAX) { c=1; }
    }
    return 0;
}

int main() {
    if (fermat())
        std::cout << "Fermat's Last Theorem has been disproved.\n";
    else
        std::cout << "Fermat's Last Theorem has not been disproved.\n";
}
```

4.2 Types of Behaviors

4.1.1- Ill Formed Behavior – These are errors that cause change due to problems in the semantic formats of the code being compiled, for such behaviors the programmer has the power to correct it and have it function according to his/her needs.

4.1.2- Implementation Defined Behavior – Are behaviors that extremely platform and compiler dependent during the runtime, examples include `std::size_t` and the number of bits in a byte.

4.1.3 - Unspecified Behavior – The overhead decisions that a compiler gets to make but are subject change during every session of a runtime falls under unspecified behavior. Examples of such behavior can be noticed during the amount of array allocation occurring overhead, the order of evaluation of a complex expression etc.

4.1.4- Undefined Behavior – Implies that there are clearly no restrictions as to how the compiler decides to deal with the code, Solid examples of undefined behaviors are array out of bounds, signed integer overflow, trying to dereference a NULL pointer and so on.

Although Undefined Behaviors cannot be avoided, they have to carefully be mitigated to prevent safety concerning issues such as memory overflow exploits, or integer overflow exploits. Modern day compilers make use of its backend action of code optimization and then optimizes the code towards arbitrary logic, hence the compiler as a collective can produce results that nobody who's able to make the guess, this is also why Undefined Behavior is massively being hidden off by compiler optimization, which at times can prove to be dangerous.

As there are numerous developers who chose to work with code that they are not aware can cause and demonstrate undefined behavior, they are often unable to diagnose and correct the problem being thrown at them, this can have the obvious damaging effects on a given project. The results that can be obtained due to undefined behavior can vary on a very large scale,

Given below are the list of one of the most common undefined behaviors, and their mitigation techniques in order to have a better reliability metric. Talking about testing and all possible methods in a program execution can get tricky, hence we shall explore each one of the UBs under certain assumptions that would set certain ground rules for understanding the behaviors. Firstly, we are to assume that the function terminates after every input, Secondly, we should assume that the function's execution manner is

deterministic, that is say for example, not cooperating with other threads via shared memory. And lastly, we shall pretend that we have infinite bandwidth of compute exhaustive resources, which means that the code on the machine should be able to test any type of test cases that we wish to pass through it, these inputs can come from console, I/O, global variables etc.

We can then form a simple and an abstract algorithm that can understood to explain why are doing what we are doing.

1. Take input and compute the next input, terminate the same if all have been tried
2. Using obtained inputs, run the functions in C abstract machines, keeping track of any operation with undefined behavior was executed.
3. Repeat step 1

Using this algorithm, we can also further deduce that the algorithms or the code snippets we will be testing shall fall under one of these three choices being provided to us. They are

- **Type 1 – The Behavior is well defined for all inputs**
- **Type 2 – Partially Defined Inputs and Outputs, may or may not work.**
- **Type 3 – None of the behaviors are defined nor justified.**

4.2 Exploring the Types of Functions that Demonstrate UB

4.2.1 – Type 1 Functions – Well Defined Behavior for all inputs

They are the codes that are usually not having any restrictions on their inputs, the compilers behave optimally during all possible inputs. The usual availability of UB on this scale can be found on API level functions and functions that are having to deal with data that is unsanitized should under Type 1.

```
int32_t safe_div_int32_t (int32_t a, int32_t b) {  
    if ((b == 0) || ((a == INT32_MIN) && (b == -1))) {  
        report_integer_math_error();  
        return 0;  
    } else {  
        return a / b;  
    }  
}
```

Type 1 functions are usually very rare to come across, because the compiler is obligated to generate false code so that overrides cannot take place, Hence is almost and always optimized.

4.2.2 – Type 2 Functions

In this possible scenario of behaviors, certain programs may or may not exhibit undefined behavior, example of such behavior would be a classic signed integer problem

```
int32_t unsafe_div_int32_t (int32_t a, int32_t b) {  
    return a / b;  
}
```

For checking unsafe division, the compiler performs a two step analysis, where it checks the preconditions that cause it to execute in the first place, Let's take a look at the case analysis test cases that are used by the compiler.

Case 1 –

```
(b != 0) && (!(a == INT32_MIN) && (b == -1))
```

for this case, the compiler is obligated to omit the return statement that a/b and exit the function without prompting the user. Because the behavior of the division operator is defined.

Case 2 –

```
(b == 0) || ((a == INT32_MIN) && (b == -1))
```

If this condition tests true, the behavior of the / operator goes undefined.

Now the choice lies in deciding which of the conditions get to proceed, and this is decided by the developers of the compilers during its creation. Let us take a look at another Type 2 Function example.

```
int demo(int var1)
{
    return(a+1) > a;
}
```

Now the precondition against which this code snippet gets tested is as follows,

(a != INT_MAX)

Again the compiler decides to go ahead with multiple evaluation cases,

Case 1 - a!= INT_MAX

This behavior is defined, and the compiler is obligated to return 1

Case 2 - a == INT_MAX

This behavior is defined, and the compiler is obligated to return 0

Here we notice that compiler test cases cannot be collapsed and then its obligated actually perform an addition and check it its result.

4.2.3 – Type 3 Functions

Can be phrased as the worst case UB's for a programming language, to such an extent that compilers sometimes simply behave as they do not exist. For the same compilers are getting smarter by the day in detecting such undefined behaviors that don't make a major difference both in the safety and mission critical codes they are coded in. Eg,

- Integer overflow,
- Array out of bounds
- Bit shifts and unaligned memory access violations.

Looking at the now classified types of undefined behaviors, we can now say that UB cannot be avoided. Hence in the long run it's safe to assume that unsafe programming languages will not be of common usage by mainstream developers, but will be used for safety and mission critical use cases that demand optimal resource utilization and peak

performance at all times. There are certain mitigation techniques that can work as a path for the observed Undefined Behaviors.

- For production level code it is always recommended to make use of multiple compilers in order to eliminate redundant checks.
- Use UB mitigation tools such as Tsan, Asan and Valgrid.
- Use of extensive documentation for Type 2 undefined behavior functions.
- Make use of behavior traps, such as GCC features that control trap integer flows.

Now that we know what developers are looking to avoid in order to trigger unintentional undefined behavior, we can start by learning about how compilers and their UB mitigation tools detect and avert possible undefined behavior but using the compiler directives and optimizing it during the code optimization phase. The basic role of a UB mitigation tool is that they convert UB into defined behaviors. Some of the most important capabilities such a tool should have includes the following,

- It should not break our code during its test cases and break corner conditions.
- Have a low overhead utilization

In order to mitigate undefined behaviors in a very specific manner, there is one primary goal that every compiler has to achieve while dealing with undefined behavior.

Goal – Every UB from a particular programming language must have

- Well documented mitigation methodologies
- Should have run into a fatal compilation error

The compilers should have the above two points into consideration.

4.2.4 – Spatial Memory Violations

Background: UBs that are triggered due to out of bound storage access and dereferencing NULL pointers can lead to a spatial memory violation, As compiler neither has the value of the operation nor the address block it is trying to access. Hence the compiler is obligated to go ahead with what it seems best within its defined set of rules.

Debugging: To counter spatial memory violations, strict usage of tools during project development phase such as Valgrid and Asan can prove to be of great help. These tools operate by creating Red Zones around blocks that have been validated. Valgrid works on the executable code and has limited scope towards detecting UB. Asan on the other hands is tested during compile time.

Mitigation: There exists high level mitigation techniques that are listed below

- ASLR
- Stack Canaries
- Hardened Allocators and
- Control Flow Integrity
- Real time bound checking needs to be put in place.

4.2.5 -Temporal Memory Safety Violations

Background: A temporal memory violation occurs when there is usage of a memory location by a certain operation after its usage has ended. This can cause severe problems as another task in the execution queue can started off by using the same memory space. This include examples of dangling pointer, addresses of automatic variables, illegal usage of free (). If the memory blocks are in use, it is possible that due to UB there can be an illegal write operation.

Debugging: Asan tool is designed to handle after free bugs and behaviors, it can be debugged by placing freed memory blocks in quarantine, thus preventing their immediate reuse. Asan can also detect the addresses of automatic variables that have survived past their freed operations. This temporal memory violations can cause optimizations on the compiler upto the O2 level.

Mitigation: This uses the same quarantine principle in order to stop the immediate usage of freed memory blocks.

4.2.6 – Integer Overflows and Buffer Overflow

Background: Integers sequences can almost never underflow, but there is definite possibility of an overflow and in both the directions. In C/C++ the macros that handle the signed integer overflow. INT_MAX and INT_MIN.

Debugging: For integer related issues the tool UBSan proves to be of great help, UBSan makes uses of mathematical integer overflow traps. Some compilers such as GCC come inbuilt with their own version of UBSan thus improving the compatibility of the code and increasing its detection capabilities.

Mitigation: Make use of UBSan in trapping mode, it is a reasonably efficient tool for handling integer and buffer overflow related behaviors. Thankfully catching integer and buffer overflow undefined behaviors are not that hard in general, UBSan is probably the only tool we'll need.

4.2.7 – Aliasing Problems

Background: Aliasing is a concept in pointers that works similar to references, this basically means that two pointers can point to the same variable names with two different data types, but are to never share the same addresses or storage location. The compiler then uses this to its optimizations.

Debugging: In the field of undefined behaviors that are caused due to aliasing problems, a major undefined behavior that is caused due to aliasing problems is that the code allows polymorphism to pass through void pointers. The tools in existence that could be used to debug the same are either very weak or not complete.

Mitigation: The developers can take an easy way to mitigate this issue, that is by essentially just going ahead and disabling the compiler optimizations that are based on strict aliasing. Some compilers chose to completely not implement this class of optimization in the first place. Compilers such as LLVM and GCC implement them and let the users deal with the behaviors they give rise to, on the other hand there are compilers that do not implement this technique in the first place. Eg, MSVC.

4.2.8 – Unaligned Memory Access

Background: Unaligned memory access is known to occur when an operation tries to access memory blocks in multiples of a specific unit, and then at some point jump to a memory location that either does not fit into the multiple order, or they do not exist. These undefined behaviors are best visible in the target architectures such as x64.

Debugging: Tools such as UBSan is a tool that can to a certain extent detect but not correct unaligned memory access.

Mitigation: None that are known so far.

4.2.9 – Unsafe and Masked Infinite Loops, or Loops that do not terminate

Background: Some compilers such as Clang internally optimize infinite loops or can even go as far as to detect and stop masked infinite loops that might have been a cause of bad developer habits.

Debugging: In present day, there are no tools that can debug and possibly solve a non-terminating or masked infinite loops. Only detection is possible.

Mitigation: Cannot be mitigated.

4.2.10 – Concurrent Thread Access

Background: Concurrent thread access occurs when, a task from a memory is in need of access to multiple threads, at least one of the access try is a store, and the access tries that are not synchronized.

Debugging: A similar tool named Tsan is a famous data race detector. It's usage is usually complicated and the concurrent thread access causing the undefined behavior in the first place itself is quite a rare occurrence.

Mitigation: Developers are advised to avoid or not use multithreading when possible.

Similarly, we can now list almost all the types of UB's that have been discussed.

4.3 Fermat's Last Theorem – An Approach Using Undefined Behavior

Fermat's theorem is an interesting use case tested for demonstrating Undefined Behaviors, simply put the theorem states that, given the equation

$a, b, c > 0$ and $n > 2$ such that

$$a^n + b^n = c^n, \text{ hence find integers}$$

```
#include <iostream>

int fermat() {
    const int MAX = 10000;
    int v1=1, v2=1, v3=1;
    // Endless loop with no side effects is UB
    while (1) {
        if (((v1*v1*v1) == ((v2*v2*v2)+(v3*v3*v3)))) return 1;
        v1++;
        if (v1>MAX) { v1=1; v2++; }
        if (v2>MAX) { v2=1; v3++; }
        if (v3>MAX) { v3=1; }
    }
    return 0;
}

int main() {
    if (fermat())
        std::cout << "Fermat's Last Theorem has been disproved.\n";
    else
        std::cout << "Fermat's Last Theorem has not been disproved.\n";
}
```

According to Fermat's Last Theorem, this code will only exit its loop condition if it finds an example that counters the special use case of Fermat's Last Theorem. This has been a long sought-after mathematical problem, trying to solve it using the flaws of a concept such as Undefined Behavior makes this experiment interesting. In due course of this code that executes, we shall observe the following.

As we see the code above that tries to prove Fermat's last theorem, we see that the output of the program returns the following message.

```

regehr@john-home:~$ icc fermat2.c -o fermat2
regehr@john-home:~$ ./fermat2
Fermat's Last Theorem has been disproved.
regehr@john-home:~$ suncc -O fermat2.c -o fermat2
"fermat2.c", line 20: warning: statement not reached
regehr@john-home:~$ ./fermat2
Fermat's Last Theorem has been disproved.

```

It is interesting to note that, the following constraints have nothing wrong in them despite being within the allowed limits of 2^{31} . Which essentially means that this logic does not have any arithmetic overflows present in the code. But yet the loop terminates with a return 1. Which is why according to the C++ coding conventions, we observe that the compilers have been created in order to treat similar infinite loops (both visible and masked) as finite and returns as soon as the UBSan detects an arithmetic overflow. Doing this reserves the hardware functionality of the devices its running on.

- We observe that, as the compilation reaches the while(1), the only way it can get out is by sending out the return 1 statement out of the infinite loop, hence this function fermat() gets optimized as the following,

```

int fermat (void)
{
    return 1;
}

```

Hence, we see that in order for a compiler to permit a program to run its infinite loop for a specified time is by introducing a modification to the external or outer body of the loop body. One common way that developers find a way around this is by introducing the **volatile** keyword. Therefore, speaking logically the Fermat's theorem still stands disproved, however the insights it offers us while using it as a testbed for UB is what makes this code run unique.

Chapter 5

Conclusion and Future Enhancements

5.1 Conclusion

For a C/C++ developer, it is very important to be aware of compiler behaviors, because it improves the perspective of the developer in terms of deciding whether an UB is to be used to its advantage or as a damaging content within the code.

So there are essentially a lot of methods through which we can be aware of UB.

- Get comfortable and started early with tools that aid with detect and deal with UB.
- Use complicated tools such as TIS Interpreter in order to get a grip on the basis of compiler behavior.
- Code and Unit Testing Methodologies should incorporate vast and extensive test cases in order to accommodate any possible triggers of UB.
- Perform code reviews being aware that a part of what when wrong could have been caused by UB.
- Aware of C and C++ Documentation standards at all times.

5.2 Possible Future Enhancements

Some of the possible future enhancements include that Undefined Behavior is a design choice at the end of the day, programmers have been using it to protect hardware functionality.

- Automatic UB Detection tools using Artificial Intelligence
- Creation of a memory quarantine space that deals with all the memory allocated dynamically.
- Improved Debuggers and Compilers Standards.
- Reduction in the number of detected UB and improve error message quality of the code.

