Undefined Behavior:

What Every Programmer Should Know and Fear

FEDOR PIKUS

Cppcon
The C++ Conference
2023
October 01 - 06

We will learn:

- What is Undefined Behavior
  - And what it is not
  - UB vs implementation-defined behavior vs unspecified behavior
- Why have UB in programming languages?
- How is UB related to performance?
- UB and C++ compilers
- How to avoid UB?
- How to take advantage of UB in your own programs?

What is Undefined Behavior (UB) in C++?

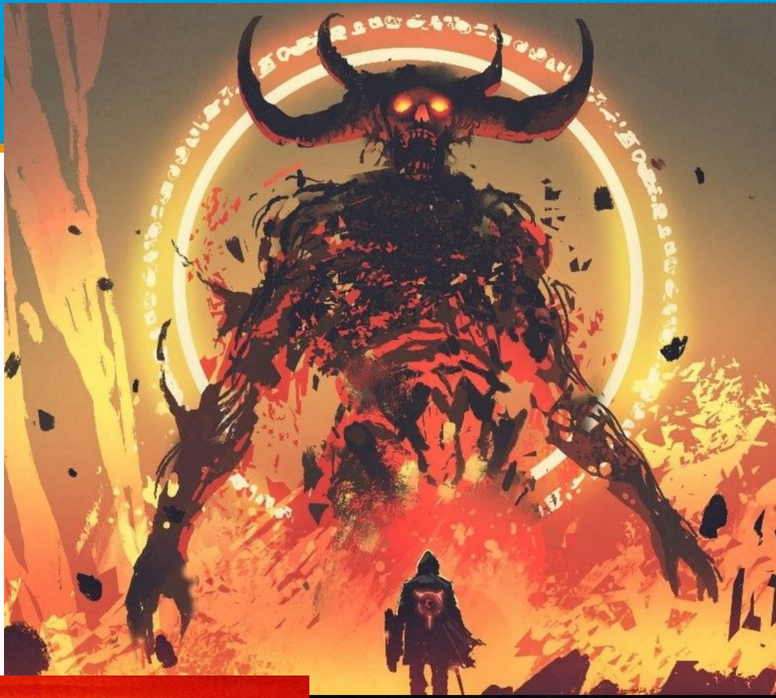
- The concept is defined in the standard

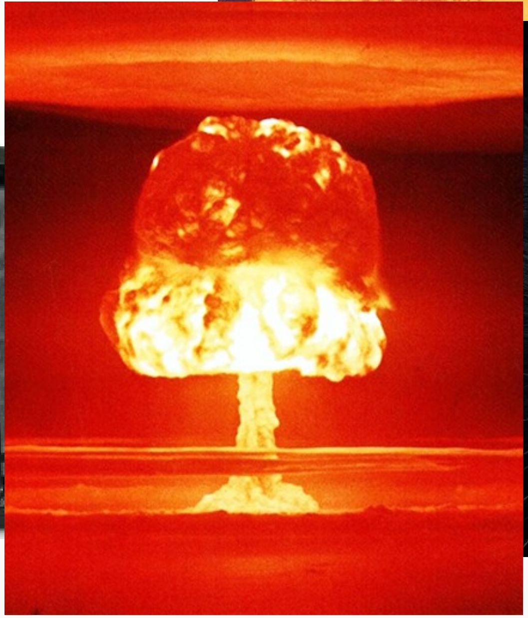
```
int f(int k) {
  return k + 10;
}
```

- According to the standard, if `k > INT_MAX-10`, **the result is undefined**
  - "**The program is not well-defined**" language is also used
- The standard imposes no requirements or restrictions on the result of `f()`
  - Only if `k > INT_MAX-10` (for any `k <= INT_MAX-10`, the result is well-defined)
- The standard imposes no requirements or restrictions on the result of any program that calls `f()` with `k > INT_MAX-10`

UB Lore, according to comp.std.c

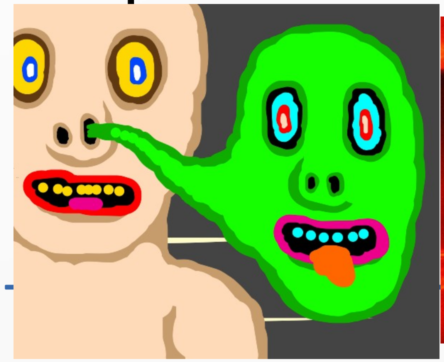
- If your program exhibits UB, anything can happen, or it is legal for the compiler to:
  - Make demons fly out of your nose
  - Launch nuclear missiles
  - Neuter your cat (even if you didn’t have a cat)
  - Format your hard drive
  - Or anything else

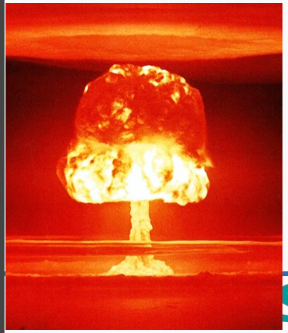




UB in real world

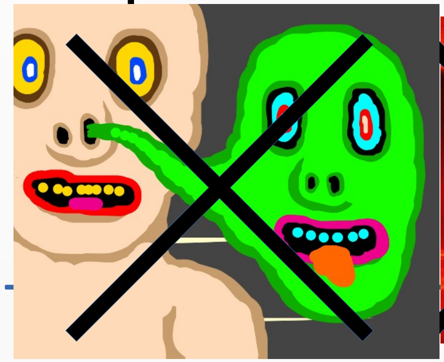
- Do not overstate the danger of UB
- Over-exaggeration of UB potential does not help the programmers form the correct expectations and attitudes toward UB
  - It contributes to the habit to underestimate UB
- UB should not be underestimated
- True danger of UB: the result of the *entire program* is undefined if *any* operation exhibits UB (according to the standard)
- “Undefined” means the standard imposes no requirements on such result

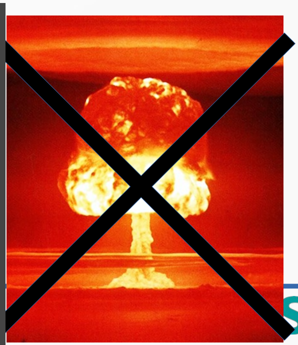




UB in real world

- Do not overstate the danger of UB
- Over-exaggeration of UB potential does not help the programmers form the correct expectations and attitudes toward UB
  - It contributes to the habit to underestimate UB
- UB should not be underestimated
- True danger of UB: the result of the *entire program* is undefined if *any* operation exhibits UB (according to the standard)
- “Undefined” means the standard imposes no requirements on such result
  - UB does not grant your program special powers





SIEMENS

UB in real world

- Do not understate the danger of UB
- Common argument: the program must produce *some* result from the i+10 expression
  - It could be wrong, but it has to be *something*
  - The extent of undefined results is limited - **FALSE**
- True danger of UB: the result of the *entire program* is undefined if *any* operation exhibits UB
- “Undefined” means the standard imposes **no requirements** on such result
- UB *anywhere* in the program taints the *entire* program

SIEMENS

What does UB really do?

- UB *anywhere* in the program taints the *entire* program – how?

```
bool f(int i) { return i + 1 > i; } // Example 01
bool g(int i) {
  if (i == INT_MAX) return false;
  else return f(i);
}
```

- Assuming the compiler can see the body of `f()` when compiling `g()`:
- Compiler can assume that UB does not happen!
  - If UB does not happen, the result is correct (assumption is true)
  - If UB does happen, the standard imposes no requirements on the results of the *entire* program
- The compiler does not have to condone UB!

Does it really happen?

- Don’t get stuck on this question: tomorrow’s compiler could do it

```
bool f(int i) { return i + 1 > i; } // Example 01
```

- Signed integer overflow is UB
- UB never happens (compiler is not required to condone UB)
- `i != INT_MAX`
- `i + 1` is always greater than `i`

```
bool g(int i) { return true; }
```

Does it really happen?

- Don’t get stuck on this question: tomorrow’s compiler could do it

```
bool f(int i) { return i + 1 > i; } // Example 01
```

- Signed integer overflow is UB
- UB never happens (compiler is not required to condone UB)
- `i != INT_MAX`
- `i + 1` is always greater than `i`

```
bool g(int i) { return true; }
```

- Direct proof – compare the assembly of f() vs g()

```
<_Z1fi>:                      |        <_Z1gi>:
mov     $0x1,%eax             |        mov     $0x1,%eax
retq                          |        retq
```

Does it really happen?

- Are we sure it's because of UB?

```
bool f(unsigned int i) { return i + 1 > i; }
```

- Unsigned integer overflow is well-defined

```
<_Z1fi>:                  | <_Z1fj>:
mov    $0x1,%eax          | cmp    $0xffffffff,%edi
retq                      | setne  %al
                          | retq
```

How real is the danger?

- “I tried and my compiler does not do that!”
  - Not today, but it likely will soon enough
  - It already does something very similar

SIEMENS

How real is the danger?

- "I tried and my compiler does not do that!"
  - Not today, but it likely will soon enough
  - It already does something very similar

```
int i = 1;
int main() {
  cout << "Before" << endl;
  while (i) {}
  cout << "After" << endl;
}
```

- GCC, O3: prints "Before" and hangs
- CLANG, O3: prints "Before", "After", and exits

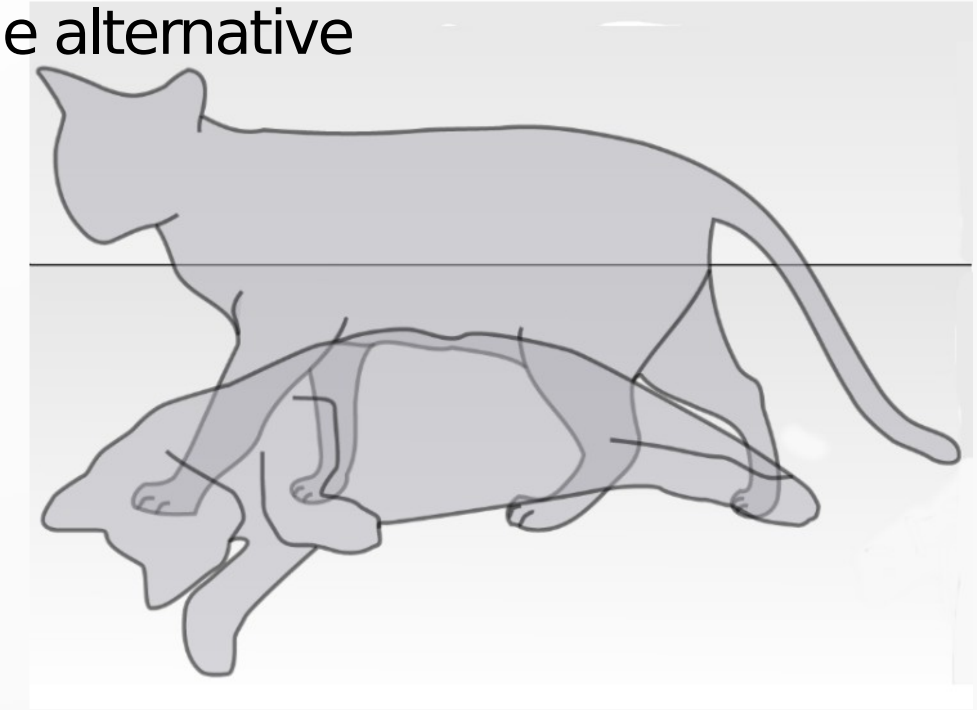
How real is the danger?

```
int i = 1; // Example 02
  int main() {
  cout << "Before" << endl;
  while (i) {}
  cout << "After" << endl;
}
```

- GCC-9, O3: prints “Before” and hangs
- GCC-10, O3: prints “Before”, “After”, and exits
- Both results are valid – infinite loop is UB per the standard
- The standard says nothing about what this program should do

Why have UB at all?

- Why does not the standard define everything?
- Obvious reason: support for different kinds of hardware
- OK, why doesn't the standard make implementations define everything?
  - Some implementations may have no reasonable alternative
- Performance is a very important reason



Why have UB at all?

- Performance is a very important reason

```
size_t n1 = 0, n2 = 0;
void f(size_t n) {
  for (size_t j = 0; j != n; j += 2) ++n1;
  for (size_t j = 0; j != n; j += 2) ++n2;
}
```

- As written, loop overhead is incurred twice

Why have UB at all?

- Performance is a very important reason

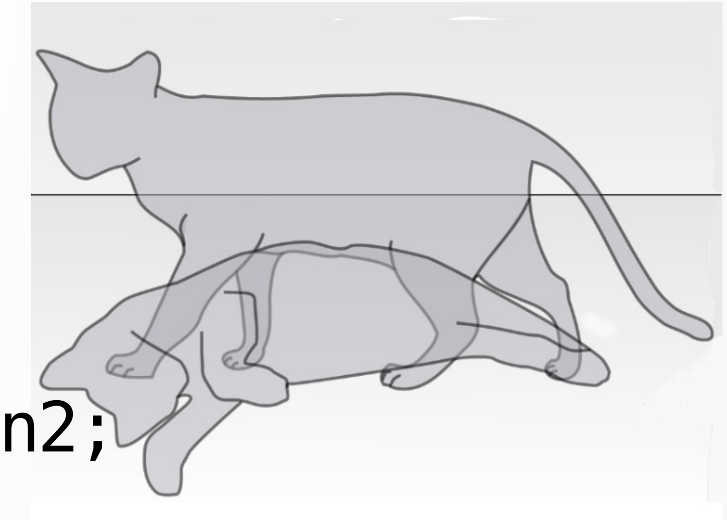
```
size_t n1 = 0, n2 = 0;
void f(size_t n) {
   for (size_t j = 0; j != n; j += 2) ++n1;
   for (size_t j = 0; j != n; j += 2) ++n2;
}
```

- As written, loop overhead is incurred twice
- Very desirable optimization:

```
void f(size_t n) {
   for (size_t j = 0; j != n; j += 2) ++n1, ++n2;
}
```


SIEMENS

Why have UB at all?



- Original code:

```
for (size_t j = 0; j != n; j += 2) ++n1;
for (size_t j = 0; j != n; j += 2) ++n2;
```

- Optimized code:

```
for (size_t j = 0; j != n; j += 2) ++n1, ++n2;
```

- Is the optimization valid?
- If the loops terminate, then the optimization is valid
- If the first loop runs forever, `n2` should not be incremented, ever
- In this example, depends on the value of `n`, which compiler can't know
  - In general, it is impossible to know if a program terminates for all inputs
- C++ standard: assume that it terminates, or it's UB and any result is valid

UB: not quite as simple

- Performance reasons (allows certain optimizations)
- Variety of supported hardware
  - Some of which is obsolete today
- Oversights or omissions in the standard
  - Some may be considered bugs in the standard
- Optimization-related reasons always remain
  - Compilers need to make assumptions they cannot prove

Undefined vs unspecified

- C++ standard does not specify behavior of every program
- **Undefined behavior**: the entire program is ill-formed

```
delete p; p→foo();
int x = -42; x << 7;
```

- **Unspecified behavior**: one of several specified outcomes must occur

```
int foo() { cout << “FOO”; return 1; }
int bar() { cout << “BAR”; return 2; }
int fred(int i, int j);
fred(foo(), bar()); // FOOBAR or BARFOO
```

- **Implementation-defined behavior**: implementation must document what happens but it must be well-defined

```
int x = -10; x >> 7;
```

UB and Optimization

- Logic of UB-dependent optimization:

1) Assume that UB does not happen
2) Deduce constraints on run-time values
3) Assume these constraints to be always true
4) Simplify the code under these assumptions

- If UB does not happen, the program is correct
- If UB happens, the standard does not specify the results of the program
  - Including the results that are produced *before* UB happens!
  - Any result is considered valid, including whatever the optimized program does

Reasoning from UB

- Assume that UB does not happen
- Deduce constraints
- Optimize code before and after possible UB

```
int f(int* p) { // Example 03
  ++(*p);
  return p ? *p : 0; // Optimized to: return *p
}
```

Reasoning from UB

- Assume that UB does not happen
- Deduce constraints
- Optimize code before and after possible UB

```
int f(int* p) { // Example 03
  ++(*p);
  return p ? *p : 0; // Optimized to: return *p
}
```

?:

```
<_Z1fPi>:                        |    <_Z1fPi>:
mov     (%rdi),%eax              |    mov     (%rdi),%eax
add     $0x1,%eax                |    add     $0x1,%eax
mov     %eax,(%rdi)              |    mov     %eax,(%rdi)
retq                             |    retq
```

no ?:

UB is even more dangerous than you thought

- UB taints the *entire* program
  - No-UB assumption affects the code *before* UB too

```
int f(int* p) { // Example 04
  if (p) ++(*p);  // Never UB
  return *p;      // UB may happen here
}
```

- Compiler optimizes away the `if (p)` check:

UB is even more dangerous than you thought

- UB taints the *entire* program
  - No-UB assumption affects the code *before* UB too

```
int f(int* p) { // Example 04
  if (p) ++(*p);  // Never UB
  return *p;      // UB may happen here
}
```

- Compiler optimizes away the `if (p)` check:

if(p)

```
<_Z1fPi>:                          | <_Z1fPi>:
mov     (%rdi),%eax                | mov     (%rdi),%eax
add     $0x1,%eax                  | add     $0x1,%eax
mov     %eax,(%rdi)                | mov     %eax,(%rdi)
retq                               | retq
```

no if (p)

Limits of UB-driven optimization

- If you rely on [potential] UB for some optimizations, it is important to understand the limitations of optimizing compilers
- The optimization must be valid under all defined conditions
- The compiler must be able to prove that it is so
  - Not “the programmer knows that it is so”

```
external void g();
int f(int*& p) {
  if (!p) g();    // Not optimized
  return *p;
}
```

- `p` might be global and accessible to `g()`
- `g()` might throw an exception

The optimization that does not happen

- Very common misconception about `const` and optimizations

```
void f(const int& y);
bool g(int x) {
  int y = x;
  f(y);
  return x == y;    // NOT optimized
}
```


SIEMENS

The optimization that does not happen

- Very common misconception about `const` and optimizations

```
void f(const int& y);
bool g(int x) {
  int y = x;
  f(y);
  return x == y;    // NOT optimized
}
```

- This is well-defined code:

```
void f(const int& y) { const_cast<int&>(y) = 3; }
```

- Unless `y` was defined as `const`, casting away `const` is not UB

```
const int y = x; // Could be optimized (in theory)
```


SIEMENS

What is UB, really?

- UB in C++ is a
  - Necessary evil
  - Annoyance
  - Something to be avoided

What is UB, really?

- Your program is designed and specified to produce certain results
- Results depend on inputs
- For every valid input, you must define corresponding results
  - This is the contract with the user

What is UB, really?

- Your program is designed and specified to produce certain results
- Results depend on inputs
- For every valid input, you must define corresponding results
  - This is the contract with the user
- You must also define what “valid” means
  - This is part of the same contract

What is UB, really?

- Your program is designed and specified to produce certain results
- Results depend on inputs
- For every valid input, you must define corresponding results
  - This is the contract with the user
- You must also define what “valid” means
  - This is part of the same contract
- What happens if the input is not valid?
  a) The input is identified as invalid and rejected with a diagnostic
  b) The input is not identified as invalid but the program now operates outside of design conditions – this is undefined behavior!

UB demystified

- Undefined behavior occurs when a program
  - receives inputs outside of the specified contract
  - does not verify and reject them
  - operates under assumptions that are no longer valid
- Why would you do this to yourself ?!
- Usually performance
  - Some preconditions are very hard to check
  - Example: prove that a program terminates

SIEMENS

Using UB wisely

- Every program must document what constitutes a valid input
  - The contract with the user
- The program must document what happens when the input is invalid
  - Violation may be detected and handled (rejected, program aborts, etc)
  - Violation may be undetectable and UB results
- Validate inputs if run-time cost is acceptable
- Provide optional validation tools if the validation is expensive

Using UB wisely

- Every program must document what constitutes a valid input
- The program must document what happens when the input is invalid
  - Violation may be detected and handled (rejected, program aborts, etc)
  - Violation may be undetectable and UB results
- Validate inputs if run-time cost is acceptable
- Provide optional validation tools if the validation is expensive
  - Provide a separate program or option to verify that graph input is valid
- Does this apply to C++ itself?
  - Standard does not require it
  - Many compile vendors offer UB detection

Detecting UB in C++ with UBSAN

- UBSAN (UB sanitizer) is included in GCC and CLANG

```
int g(int k) {
  return k + 10;
}
```

- UBSAN instruments the program at compile time

```
clang++ --std=c++17 -O3 -fsanitize=undefined ub.C
```

- Detection is done at run time

```
ub.C:10:20: runtime error: signed integer overflow:
2147483645 + 10 cannot be represented in type 'int'
```

- UB is detected only if it actually happens
  - Not all types of UB are detected
  - If UB code is not executed, nothing is detected

Guidelines for avoiding and embracing UB

- Remember that UB is [just] a program executed out of contract
- Take UB in your programs very seriously
- Use sanitizers and static analysis tools to detect UB
- Do not assume that it's OK because it "works now"
- When designing a program, remember to define the domain of inputs
- Prefer a broad contract unless you have good reasons to narrow it
- Define what happens for out-of-contract inputs
- Detect invalid inputs unless it's too difficult
- Provide optional input validation tools to your users
- Use optional input validation tools if you are a user

Thank you for listening!
Questions?

SIEMENS