Programming in C and C++

Lecture 10: Undefined Behaviour

Neel Krishnaswami and Alan Mycroft

C is Portable Assembly

C is the most widely deployed language in the world:

- Unix was originally written in assembly, and C was created as a thin layer to avoid rewriting all of it for every new CPU.
- C now compiles to almost every CPU architecture ever made!
- Statements map to a small set of machine instructions.
- Numerous language extensions exist to expose hardware features. For example, to add two numbers in x86 assembly with GNU GCC:

```
1  int foo = 10, bar = 15;
2  __asm__ __volatile__("addl %%ebx,%%eax"
3                       :"=a"(foo)
4                       :"a"(foo), "b"(bar)
5                       );
6  printf("foo+bar=%d\n", foo);
```

The Optimisation Tradeoff

There is a tension between:

- Programmers' ablility to *predict* performance of code.
- Compilers generating *fast* machine code for specific hardware.
- The language's *portability* on present and future architectures.
- Revisions to C must remain *backwards compatible* with existing code.

Balance is achieved via an ongoing language specification process:

- Originally Ritchie and Kernighan's book in 1978 ("K&R C").
- Replaced by an ANSI standard in 1989 ("ANSI C" or "C89").
- ISO update with features like IEEE 754 floating point ("C99")
- 2011 revision has a detailed memory model ("C11").

The C Abstract Machine

The C standard defines these broad concepts:

- The semantic descriptions of language features describe the behavior of an *abstract machine* in which issues of optimization are irrelevant.
- Defines *side effects* as accessing a `volatile` object, modifying an object or file, or calling a function that does any of those operations.
- Side effects change the state of the execution environment and are produced by *expression evaluation*.
- Certain points in the control flow are *sequence points* at which point all side effects which have been seen so far are guaranteed to be complete.

Examples of Sequence Points in C

- Between the left and right operands of the `&&` or `||` operators.

```
1  // Side effects on p happen before those on q
2  *p++ != 0 && *q++ != 0
```

- Between the evaluation of the first operand on the ternary (question mark) operator and the second and third operands.

```
1  // p incremented when the second instance is evaluated
2  a = (*p++) ? (*p++) : 0
```

- At the end of a full expression; e.g. return statements, control flow from `if`, `switch`, `while` or `do`/`while`, and all three expressions in a `for` statement.

Examples of Sequence Points in C (cont.)

Function calls have several sequence points:

- Before a function call is entered.

```
f(i++)
// f() is called with the original value of i
// but i is incremented before entering the body of f()
```

- The order in which arguments are evaluated is not specified.

```
f(i++) + g(j++) + h(k++)
```

  - `f()`, `g()`, `h()` may be executed in any order.
  - `i`, `j`, `k` may be incremented in any order.
  - Always obeying the previous rule, `i` is incremented before `f()` is entered, as is `j` for `g()` and `k` for `h()` respectively.

```
f(i++, j++) + h(k++)
```

- `i` and `j` may be incremented in any order, but both will be incremented before `f()` is entered.

Execution of the Abstract Machine

- In the abstract machine, all expressions are evaluated as specified by the language semantics in the standard.
- An implementation need not evaluate part of an expression if it can deduce that:
  - its value is not used
  - no needed side effects are produced
  - includes any caused by function calls or accessing a volatile object
- If an asynchronous signal is received, only the values of objects from the previous sequence point can be relied on.

Freedom to Optimize

Beyond the abstract machine, the C standard leaves *plenty* of room for compilers to optimize source code to take advantage of specific hardware. The standard defines three types of compiler freedom:

- *implementation-defined* behaviour means that the compiler must choose and document a consistent behaviour.
- *unspecified behaviour* means that from a given set of possibilities, the compiler chooses one (or different ones within the same program).
- *undefined behaviour* means arbitrary behaviour from the compiler.

  When the compiler encounters [an undefined construct] it is legal for it to make demons fly out of your nose – *`comp.std.c` newsgroup, circa 1992*

Implementation-defined Behaviour

These are typically set depending on the target hardware architecture and operating systems Application Binary Interface (ABI). Examples of implementation-defined behaviour include:

- Number of bits in a byte (minimum of 8 and exactly 8 in every modern system, but the PDP-10 had 36 bits per byte!)
- `sizeof(int)` which is commonly 32- or 64-bit
- `char a = (char)123456;`
- Results of some bitwise operations on signed integers
- Result of converting a pointer to an integer or vice versa

Compiler warnings help spot accidental dependency on this behaviour:

```
test.c:3:13: warning: shift count >= width of type
[-Wshift-count-overflow]
int x = a >> 64;
```

Unspecified Behaviour

The compiler can vary these within the same program to maximise effectiveness of its optimisations. For example:

- Evaluation order of arguments in a function call.
- The order in which side effects take place when not otherwise explicitly specified by the standard.
- Whether a call to an `inline` function uses the inline or external definition.
- The memory layout of storage for function arguments.
- The order and contiguity of storage allocated by successive calls to the `malloc`, `calloc` or `realloc` functions

Undefined Behaviour

There is a long list of undefined behaviours in the C standard (J.2). Some examples include:

- Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored

```
int i = 0;
f(i++, i++); // what arguments will f be called with?
```

- Conversion of a pointer to an integer type produces a value outside the range that can be represented

```
char *p = malloc(10);
short x = (short)p; // if address space larger than short?
```

- The initial character of an identifier is a universal character name designating a digit `char \u0031morething;`

Undefined Behaviour (cont.)

The value of a pointer to an object whose lifetime has ended is used. We already encountered this back in lecture 4:

```c
#include <stdio.h>

char *unary(unsigned short s) {
  char local[s+1];
  int i;
  for (i=0;i<s;i++) local[i]='1';
  local[s]='\0';
  return local;
}

int main(void) {
  printf("%s\n",unary(6)); //What does this print?
  return 0;
}
```

Undefined Behaviour (cont.)

Use of an uninitialized variable before accessing it:

```
1  int main(int argc, char **argv) {
2    int i;
3    while (i < 10) {
4      printf("%d\n", i);
5      i++;
6    }
7  }
```

Undefined Behaviour (cont.)

Accessing out-of-bounds memory.

```
1  char *buf = malloc(10);
2  buf[10] = '\0';
```

Dereferencing a NULL pointer or wild pointer (e.g. after calling free).

```
1  char *buf = malloc(10);
2  buf[0] = 'a'; // what if buf is NUL?
3  free(buf);
4  buf[0] = '\0'; // buf has been freed
```

Undefined Behaviour (cont.)

Signed integer arithmetic is undefined if the result overflows.

- For instance for type `int`, `INT_MAX+1` is not `INT_MIN`.
- By knowing that values "cannot" overflow, the compiler can enable useful optimisations:
  `for (i = 0; i <= N; ++i) { ... }`
- If signed arithmetic is undefined, then the compiler can assume the loop runs exactly `N+1` times.
- But if overflow were defined to wrap around, then the loop could potentially be infinite and hard to optimise. Consider this case: `for (i = 0; i <= INT_MAX; ++i) { ... }`

Undefined behaviour thus enables useful compiler optimisations.
But at what cost?

Undefined Behaviour (cont.)

- The C standard states (Section 1.3.24):
  Permissible undefined behaviour ranges from **ignoring the situation completely with unpredictable results**, *to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).*

- *Unspecified behaviour* usually relates to platform portability issues (e.g. 32-bit vs 64-bit, Linux vs Windows).
- *Undefined behaviour* should be avoided at all costs, even if your program appears to work fine in one instance.

Security Issues due to Undefined Behaviour

```
1  void read_from_network(int size) {
2    // Catch integer overflow.
3    //
4    if (size > size+1) errx(1, "packet too big");
5
6    char *buf = malloc(size+1);
7    if (buf == NULL)
8      errx(1, "out of memory");
9
10   read(fd, buf, size);
11   // ... error checking on read.
12
13   buf[size] = 0;
14   process_packet(buf);
15   free(buf);
16 }
```

Security Issues due to Undefined Behaviour (cont.)

```
1  void read_from_network(int size) {
2    // size > size+1 is impossible since signed
3    // overflow is impossible. Optimize it out!
4    // if (size > size+1) errx(1, "packet too big");
5
6    char *buf = malloc(size+1);
7    if (buf == NULL)
8      errx(1, "out of memory");
9
10   read(fd, buf, size);
11   // ... error checking on read.
12
13   buf[size] = 0;
14   process_packet(buf);
15   free(buf);
16 }
```

Security Issues due to Undefined Behaviour (cont.)

> *A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.*

– C99 Standard Sec 6.2.5/9

- This optimisation does not occur with `unsigned int` since it is defined to wrap around.
- Only signed overflow is undefined, so using `unsigned` integers to track buffer sizes is wise.
- Particularly insidious since a security auditor reading the code can easily miss this. There appears to be a *security* check in the code!

Security Issues due to Undefined Behaviour (cont.)

Poor Logic: *The x86 ADD instruction is used to implement C's signed add operation, and it has twos complement behavior when the result overflows. I'm developing for an x86 platform, so I should be able to expect twos complement semantics when 32-bit signed integers overflow.*

Security Issues due to Undefined Behaviour (cont.)

Poor Logic: *The x86 ADD instruction is used to implement C's signed add operation, and it has twos complement behavior when the result overflows. I'm developing for an x86 platform, so I should be able to expect twos complement semantics when 32-bit signed integers overflow.*

Analogy: *Somebody once told me that in football you can't pick up the ball and run. I got a football and tried it and it worked just fine. They obviously didn't understand football.* It is possible to sometimes get away with undefined behaviour, but it will eventually go wrong in very unexpected ways.

Further Reading: `http://blog.regehr.org/archives/213`

Compiler View on Undefined Behaviour

- The compiler only has to execute statements where the behaviour is defined, and can optimise the rest away.
- In the C abstract machine, every operation that is being performed is either defined or undefined (as classified by the C standard).
- Application developers need to worry about *every* input to the program being defined, as the compiler is an "adversary" that can jump on undefined behaviour and subvert the original "intention".

Compiler View on Undefined Behaviour (cont.)

Consider a simplified program fragment, where function definitions:

- terminate for every input
- run in a single thread
- have infinite computing resources.

The compiler categorises these functions into three kinds:

- **Always Defined**: No restrictions on their inputs, and they are defined for all possible inputs.
- **Sometimes Defined**: Some restrictions on their inputs, so they can be either defined or undefined depending on the input value.
- **Always Undefined**: No valid inputs are admitted, and they are always undefined.

"Always-Defined" Functions

```
1  int32_t safe_div_int32_t (int32_t a, int32_t b) {
2    if ((b == 0) || ((a == INT32_MIN) && (b == -1))) {
3      report_integer_math_error();
4      return 0;
5    } else {
6      return a / b;
7    }
8  }
```

- Inputs are carefully checked to ensure that undefined operations invoke an error function.
- Error handling functions *are not the same* as undefined behaviour since they have explicitly defined semantics.

“Sometimes-Defined” Functions

```
1  int32_t safe_div_int32_t (int32_t a, int32_t b) {
2    return a / b;
3  }
4  // function call defined iff
5  // ((b == 0) || ((a == INT32_MIN) && (b == -1)))
```

- All calls to `safe_div_int32_t` must now (somehow) be checked to ensure that the preconditions are met.
- If compiler statically detects an input that would be undefined, it can skip the function call entirely.

“Always-Undefined” Functions

```
1  int32_t safe_div_int32_t (int32_t a, int32_t b) {
2    bool check_overflow;
3    if (check_overflow &&
4        ((b == 0) || ((a == INT32_MIN) && (b == -1)))) {
5      report_integer_math_error();
6      return 0;
7    } else {
8      return a / b;
9    }
10 }
```

Why is this variant of `safe_div_int32_t` always undefined for all inputs?

Case Analysis in Linux kernel

```
1  static void __devexit agnx_pci_remove (struct pci_dev *pdev)
2  {
3    struct ieee80211_hw *dev = pci_get_drvdata(pdev);
4    struct agnx_priv *priv = dev->priv;
5    if (!dev) return;
6    // ... do stuff using dev ...
7  }
```

This code gets a pointer to a device struct, tests for null and uses it. But the pointer is dereferenced before the null check! An optimising compiler (e.g. gcc at -O2 or higher) performs the following case analysis:

- if `dev == NULL` then `dev->priv` has undefined behaviour.
- if `dev != NULL` then the null pointer check will not fail. Null pointer check is dead code and may be deleted.

Living with Undefined Behaviour

Long term: Only use unsafe programming languages to build safer abstractions (e.g. use C/C++ to make bindings for Rust, Java or ML).

Short term: No easy answer, as many tools and techniques are needed.

- Use many compilers (clang or gcc) and enable warnings (`-Wall`).
- Use static analysis tools (like `clang-analyzer` or Coverity) to spot errors, or dynamic analysis engines like Valgrind.
- Modern clang and gcc have "undefined behaviour sanitizers" that detect and generate errors for many classes of undefined behaviour via the `-fsanitize=undefined` command line flag.
- Avoid "sometimes-undefined" functions by checking all inputs.
- Use high-quality third-party libraries that obey these rules.

"Be very careful, use good tools, and hope for the best." John Regehr, `http://blog.regehr.org/archives/213`