C Praktikum

Undefined Behavior

Eugen Betke, Nathanael Hübbe,
Michael Kuhn, Jakob Lüttgau, Jannek Squar

2018-12-17

C standard knows roughly four classes of behavior

Defined behavior

- You know the code, you know C $\Rightarrow$ you know the results

Implementation defined behavior

- You also know the compiler $\Rightarrow$ you know the results

Unspecified behavior

- You get one of several possible results

Undefined behavior

- You know nothing about the results

Implementation Defined Behavior

Behavior depends on CPU, OS, linker, or compiler

Example:

```
int i = 42;
char bytes[sizeof(i)];
memcpy(bytes, &i, sizeof(i));
printf("%d\n", *bytes);
```

Usage: Provide flexibility for the peculiarities of hardware

Unspecified Behavior

There are several distinct behaviors that the standard permits, and there is no guarantee which is selected when.

Example:

```c
int i = 42;
printf("i = %d, i++ = %d\n", i, i++);
```

Usage: Provide flexibility for optimizing compilers

Undefined Behavior

All bets are off!

Example:

```
int foo[1] = {42};
printf("%d\n", foo[1]);
```

This code may format your harddrive, as far as the standard is concerned...

Usage: Avoid overhead of safeguards

Appears ca. 200 times in the C standard!

Effects of Undefined Behavior

- Compilers may assume that it doesn’t occur
  $\Rightarrow$ No need to emit code to handle it
  $\Rightarrow$ Impossible to check for it
- May corrupt any data
  $\Rightarrow$ Hackers love Undefined Behavior
- May leak confidential data
  $\Rightarrow$ Hackers love Undefined Behavior
- Downloading a program that encrypts your harddrive is a perfectly valid implementation of Undefined Behavior as far as the standard is concerned...

C vs. Java

Executing `a[b] = c`

C

- single assembler instruction on many CPUs

C vs. Java

Executing `a[b] = c`

Java

1. check `a != NULL`
   2 instructions: compare and branch
2. load `a.length` into register
3. check `b < a.length` (unsigned comparison!)
   2 instructions: compare and branch
4. store `a[b] = c`

Total: 6 instructions and 2 memory accesses
just to avoid undefined behavior...

Pointers and Undefined Behavior

Most frequent source of undefined Behavior:
Pointer abuse

- Dereferencing `NULL` is UB
- Dereferencing uninitialized pointer is UB
- Dereferencing out-of-bounds pointer is UB
- Dereferencing stale pointer is UB
  - pointers that were `free()`'d
  - pointers pointing to variables that went out of scope
- Assigning pointer with invalid value is UB
  (uninitialized, out-of-bounds, or stale value)

Strict Aliasing Rules

Type-punning is UB since C99

Example:

```
float foo = 42.0;
int* bits = (int*)&foo;
printf("bits of float: %08x\n", *bits);
```

Can work. Or not. Depends on the mood of the compiler...

Strict Aliasing Rules

Type-punning is UB since C99

Example:

```
union { float f; int i; } bar = { .f = 42 };
printf("bits of float: %08x\n", *bits);
```

Can work. Or not. Depends on the mood of the compiler...

Strict Aliasing Rules

Type-punning is UB since C99

Only legal way: Use `memcpy()`

```
float foo = 42.0;
int bits;
assert(sizeof(foo) == sizeof(bits));
memcpy(&bits, &foo, sizeof(foo));
printf("bits of float: %08x\n", bits);
```

Aliasing of restricted pointers

The very point of the `restrict` keyword:
Aliasing restricted pointers is UB

Example:

```
void swap(int* restrict a, int* restrict b) {
        *a ^= *b, *b ^= *a, *a ^= *b;
}

int main() {
        int a = 42;
        swap(&a, &a);
}
```

Modifying immutable data

Modifications to what’s fundamentally constant is UB:

```
"Hello World!"[1] = 'a';

const int i = 42;
*(int*)&i = 666;
```

Temporary Objects

Modifying a temporary is UB

Example:

```
typedef struct{ int foo[3]; } bar;

bar baz() { return (bar){0}; }

int main() { baz().foo[1] = 42; }
```

Fixed Buffers

Never use preallocated fixed length buffers

- It's generally not possible to find a size that's impossible to overrun
- Writing correct error handling for fixed buffers is hard
- Erroring out on too long input is an anti-feature

Flexible Buffers

Allocate your buffers to fit

1. Determine how much you need
2. Allocate what you need
3. Use exactly what you allocated

Failing the above: Grow your buffer with your need

1. Start with sensible small size
2. Check buffer size before adding something
3. Increase size by 2x with `realloc()`

Bad Library Functions

Some functions in the standard library are just reckless.

Use only with extreme care:

- `strcat()` and `strncat()`
- `strcpy()` and `strncpy()`
- `sprintf()` and `snprintf()`
- `fmemopen()` for writing
- `fgets()`
- Anything that writes strings of controllable length to a buffer...

Evil Library Functions

Some functions in the standard library are just reckless.

Never use:

- `gets()`
  From the manpage: "Never use this function"
- the `scanf()` conversions `%s` and `%[`
- `fflush()` on a file opened for input
- Anything that writes strings of uncontrollable length to a buffer...

Good Library Functions

Use POSIX.1-2008 functions that allocate their buffers to fit:

- `strdup()`
- `getline()`
- the `scanf()` conversions `%ms`, `%mc`, and `%m[`
- `open_memstream()`

Just a GNU extension: `asprintf()`

Summary

- Undefined Behavior sets C apart:
  delivers performance, and exquisite trouble...
- Mostly pointer/buffer related
  $\Rightarrow$ Never use preallocated fixed buffers
  $\Rightarrow$ Always allocate your memory to fit
- Parts of the standard library are evil!
- But better functions exist - use them!