Introduction to Undefined Behaviour

Written by:

- Thomas Tan

Last updated: 28 July 2022

Feedback form for Lecture 13

Feedback form for lecture 13

1 Introduction

Sometimes, C++ says that certain operations are *undefined*, and executing them causes *undefined behaviour*.

One example of an undefined operation that we've talked about before is the data race:

```cpp
int i = 0;

std::thread t1{[&]() { i++; }};
t1.join();
std::thread t2{[&]() { i++; }};
t2.join();
```

Snippet 1: No data race: OK

```cpp
int i = 0;

std::thread t1{[&]() { i++; }};
std::thread t2{[&]() { i++; }};
t1.join();
t2.join();
```

Snippet 2: Data race: UB

Note in particular that undefined behaviour is a *runtime condition*, not a property of the code at compile time.

For example, the following is only UB if the user inputs a positive number:

```cpp
int iters;
std::cin >> iters;

int i = 0;

std::thread t1{[&]() {
  for (; i < iters; ++i) {
  }
}};
std::thread t2{[&]() {
  for (; i < iters; ++i) {
  }
}};

t1.join();
t2.join();
```

Snippet 3: Data race only if user inputs a positive number

C++ is known as a language that has a lot more instances of undefined behaviour than other languages.

Compare our earlier examples with the following Javascript code:

```javascript
let i = 0;
setTimeout(() => i++, 0);
setTimeout(() => i++, 0);
// No data race
```

Snippet 4: There are no data races in Javascript

You might wonder what amazing programming language theory Javascript uses to avoid data races.

▼ **Why Javascript has no data races**

The answer is that Javascript is single threaded.

2 UB 1: Signed integer overflow

A big reason why C++ specifies a lot of undefined behaviour is in order to enable certain optimisations. Let's take a look at signed integer overflow:

```
int f(int a, int b) {
  return a + b;
}
```

```
int main() {
  // Using these values with f is UB
  printf("%d\n", f(1073741824, 1073741824));
}
```

Snippet 5: Signed integer overflow is UB

This is UB because signed integers in C++ are merely a restriction of the integers to a smaller range. In the case of `int`, the range of legal values is [-2147483648, 2147483647]. Since 1073741824 + 1073741824 = 2147483648 falls outside the range of legal values, this is UB.

This sounds very contrived, because every platform we know of today will print the value -2147483648, and in fact, most programmers would agree that making this behaviour undefined is unnecessary.

Nevertheless, there *are* compiler optimisations that make use of the fact that signed integer overflow is UB.

Godbolt link

```
int f(int a) {
  return a + 10 <= 100 ? 1 : 7;
}
```

```
// Generated asm:
int g(int a) {
  return a <= 90 ? 1 : 7;
}
```

```
$ cat signed-elimination.clang++.s \
    | c++filt \
    | sed 's/:\s*#.*$/:/' \
    | sed -n '/^f(/,/ret/p'
f(int):
# %bb.0:
    cmpl    $91, %edi
    movl    $1, %ecx
    movl    $7, %eax
    cmovll  %ecx, %eax
    retq
```

```
$ cat signed-elimination.clang++.s \
    | c++filt \
    | sed 's/:\s*#.*$/:/' \
    | sed -n '/^g(/,/ret/p'
g(int):
# %bb.0:
    cmpl    $91, %edi
    movl    $1, %ecx
    movl    $7, %eax
    cmovll  %ecx, %eax
    retq
```

Snippet 6: Compiler is allowed to perform simplifications assuming `a` is an unbounded integer

Notice that this simplification is only allowed if signed integer overflow is UB. If we tried to call `f(2147483647)` vs `g(2147483647)`, we might expect that due to overflow, `2147483647 + 10 = -2147483638 <= 100`, that `f` returns `1` while `g` returns `7`.

While this is true when compiling with `-O0`, compiling with optimisations on results in both `f` and `g` returning `7`.

There are still valid reasons to want wrapping arithmetic, where `2147483647 + 10 = -2147483638` is a valid and non-UB result. In order to do this, you need to cast your integer to an unsigned type, whose arithmetic is all defined in terms of modulo arithmetic.

```
int f_wrap(int a) {
  return int(unsigned(a)      //
             + unsigned(10))  //
                 <= 100
             ? 1
             : 7;
}
```

```
$ cat signed-elimination.clang++.s \
    | c++filt \
    | sed 's/:\s*#.*$/:/' \
    | sed -n '/^f_wrap(/,/ret/p'
f_wrap(int):
# %bb.0:
    addl    $10, %edi
    cmpl    $101, %edi
    movl    $1, %ecx
    movl    $7, %eax
    cmovll  %ecx, %eax
    retq
```

Snippet 7: Performing the addition with unsigned types results in wrapping arithmetic

Once again, let me emphasise that UB is something that is triggered at *runtime*, and is not a static property of a program at compile time.

For example, this code is UB depending on what values the user inputs into the program.

```
int main() {
  int i, j;
  std::cin >> i >> j;
  // UB if i + j overflows
  return i + j;
}
```

Snippet 8: UB here is conditional on user input

Obviously, this program is perfectly valid even though it is possible that UB may occur under some situations.

2.1 Other arithmetic UB

Other examples of UB that may happen when doing arithmetic are:

- Integer division or modulo by 0
  - `1 / 0`
- Pointer arithmetic after the past-the-end element or before the first element
  - `int arr[4]; arr + 5;`
- Any kind of arithmetic (e.g. multiplication or division) on signed integers whose result is outside the range of representable values
  - Signed integer overflow is a specific example
  - Another example is `INT_MIN / -1`
- Floating point to integer conversion where the truncated value is outside the range of representable values
  - `static_cast<int>(1e10)`

3 UB 2: Infinite loops

Reference: C Compilers Disprove Fermat's Last Theorem

Another interesting case of undefined behaviour is executing an infinite loop that produces no side effects.

Blatantly stealing the example from the blog post above, we can write a loop that tries to find a counterexample to Fermat's Last Theorem. If it finds a counterexample, this loop terminates and returns a `1`. Otherwise, it loops forever, producing no side effects (no prints, no communication to other threads, etc.)

```c
int fermat() {
  const int MAX = 1000;
  int a = 1, b = 1, c = 1;
  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;
}
```

```
$ cat fermat.clang++.s \
    | c++filt \
    | sed 's/:\s*#.*$/:/' \
    | sed -n '/^fermat(/,/ret/p'
fermat():
# %bb.0:
    movl    $1, %eax
    retq
```

Snippet 9: How to prove Fermat's Last Theorem

As we can see, the compiler has decided that our function simply returns a `1`, meaning that a counterexample to Fermat's Last Theorem exists :^)

The reason why this is allowed to return a `1` is because if we assume that `fermat()` does NOT return a `1`, then we know our program would have entered an infinite loop (that contain no side effects). Since executing infinite loops (that contain no side effects) is UB, the compiler assumes that this case is impossible, and thus feels justified in deducing that `fermat()` always returns a `1`, thus allowing it to yeet all the code.

Clang in particular is very aggressive when performing this optimisation. In the following example, we have two functions, `f` and `g`, where `f` simply enters an infinite loop, triggering UB, while `g` prints stuff.

```c
bool CALLED_F;
bool CALLED_G;

void f() {
  CALLED_F = true;
  while (true) {
  }
}

void g() {
  CALLED_G = true;
}
```

```c
#include <iostream>

extern bool CALLED_F;
extern bool CALLED_G;
void f();
void g();

int main() {
  f();

  if (CALLED_F) {
    std::cout << "Called f()" << std::endl;
  }
  if (CALLED_G) {
    std::cout << "Called g()" << std::endl;
  }
}
```

```
$ # clang++ terminates
$ ./main.clang++.out
Called g()
$ # g++ loops forever
$ ./main.g++.out
^C
$
```

```
$ cat a.clang++.s | c++filt \
    | grep -v '^\s*[.#]' \
    | sed -n '/^[fg](/,/end/p'
f():                # @f()
# %bb.0:
.Lfunc_end0:
g():                # @g()
# %bb.0:
  movb  $1, CALLED_G(%rip)
  retq
.Lfunc_end1:
```

Snippet 10: Aggressive infinite loop removal when compiling with clang++

Notice how the function `f():` gets no actual assembly instructions. This means that the call to `f()` in `main` actually calls the following function instead, which was `g()`.

4 Undefined, unspecified, implementation-defined, etc.

We've shown a couple examples of undefined behaviour and also why we might want to have undefined behaviours in the language.

However, while undefined behaviour is one kind of illegal behaviour which allows the compiler to do anything it wants, there are other subtler kinds of behaviours that aren't quite UB.

4.1 "Ill-formed" and "ill-formed, no diagnostic required"

Let's start on the obvious end. Some programs are just straight up incorrect. For example:

```
int main() {
  return "Hello world";
}
```

Snippet 11: An ill-formed program in C++ (not C)

Here, this program is ill-formed at compile time because the conversion from string literal to `int` is not allowed. For ill-formed programs, the compiler is required detect such cases and refuse to compile.

However, not all compile time errors are required to be detected. For example, the following program is allowed to compile, even though it breaks the One Definition Rule:

```
// a.cpp
inline int tri(int x) {
  return x + x + x;
}
```

```
// b.cpp
inline int tri(int x) {
  return x * 3;
}
```

```
// main.cpp
int tri(int);
int main() {
  return tri(0);
}
```

Snippet 12: An "ill-formed, no diagnostic required" program

However, the behaviour of this program is undefined when it is run at all, as it is still considered ill-formed.

The reason why "no diagnostic required" exists is because some cases of ill-formed programs are difficult or impossible to detect. For example, checking that a `constexpr` function will produce a constant expression for at least one set of inputs would require enumerating all possible program states, which is pretty much impossible.

4.2 Implementation-defined behaviour

The next class of behaviours is implementation-defined behaviour. This essentially means that the implementation (ie. your compiler) must *define* its behaviour, but otherwise, it can do what it wants (within the confines of The Standard).

For instance, an oft-quoted example of implementation-defined behaviour is `sizeof(long)` — on some compilers/platforms, `long` might be 32-bits, and so the value might be 4, but on others, it might be 64-bits and be 8.

A (slightly) more useful example would be non-standard attributes; we've already seen an example of that which is `[[gnu::noinline]]`; it should be fairly obvious why, since it's an attribute under the `gnu::` namespace. In this case, the implementation (gcc) specifies that it prevents the function from being inlined.

There is nothing inherently bad or unsafe about using implementation-defined behaviour, except that your code may be a little less portable to other platforms or compilers.

4.3 Unspecified behaviour

The next class of behaviours is unspecified behaviour, which is similar to implementation-defined behaviour in the sense that the compiler can do what it wants, but notably, it is *not required* to document it anywhere.

An example of unspecified behaviour is the order of evaluation of operands in a function call, eg. in `f(a(), b(), c())`.

While implementation-defined behaviour is also free to change between different versions of the same compiler, at least they are required to document it; unspecified behaviour is a little more insidious because something that you might have been (implicitly) relying on to behave a certain way can change without you noticing.

4.4 Undefined behaviour (and "time travel")

Lastly, we have the topic of today's lecture, which is undefined behaviour. While implementation-defined and unspecified behaviours allow the compiler to choose from a certain set of behaviours, undefined behaviours allow the compilers to really execute *anything at all*.

We've already seen a couple interesting examples earlier of a case where triggering UB caused really strange things to happen.

When we called `fermat()` with optimisations enabled, the function immediately returned with the value `1`. That was because entering an infinite loop was UB, and so returning the value `1` is a perfectly legal behaviour for the compiler to choose. In fact, it is also allowed to do things like call `system("rm -rf --no-preserve-root /");` whenever UB happens.

However, it is important to note that once an undefined operation is executed, the behaviour of the *whole program* is undefined, including behaviours that may have occurred *before* the undefined operation was executed.

```
$ # clang++ terminates
$ ./main.clang++.out
Called g()
```

```
bool CALLED_F;
bool CALLED_G;

void f() {
  CALLED_F = true;
  while (true) {
  }
}

void g() {
  CALLED_G = true;
}
```

```
#include <iostream>

extern bool CALLED_F;
extern bool CALLED_G;
void f();
void g();

int main() {
  f();

  if (CALLED_F) {
    std::cout << "Called f()" << std::endl;
  }
  if (CALLED_G) {
    std::cout << "Called g()" << std::endl;
  }
}
```

Snippet 13: Our earlier example of loop optimisations demonstrates "time travelling UB"

We might naively think that the execution of the program is as follows:

- Enter main
  - Enter f
    - Set `CALLED_F` to `true`
    - TRIGGER UB
  - Print stuff

We might naively expect thus that `CALLED_F` should have been set to `true`, as it occurred before the undefined operation was executed. However, that's not true, and because UB was triggered, the whole program's behaviour is undefined, including the fact that `CALLED_F` may never have been set to `true`.

In fact, if we look closely at the assembly, we'll see that indeed `CALLED_F` was not set to `true`.

```
$ cat a.clang++.s | c++filt \
    | grep -v '^\s*[.#]' \
    | sed -n '/^[fg](/,/end/p'
f():                # @f()
# %bb.0:
.Lfunc_end0:
g():                # @g()
# %bb.0:
  movb  $1, CALLED_G(%rip)
  retq
.Lfunc_end1:
```

If you're interested in another more involved example on time travelling UB, you may want to read the following blog post:

Reference: Undefined behaviour can result in time travel (among other things, but time travel is the funkiest)

5 More obvious UB involving the object model

There are many cases of UB that are "obvious", but we'll simply quickly go through a bunch of them quickly.

5.1 UB 3: Access through incorrect type

The following is UB:

```
struct S {
  int x;
};
```

```
struct T {
  int x;
};
```

```
int main() {
  S s{42};
  T* ps = reinterpret_cast<T*>(&s);
  std::cout << ps->x << std::endl;
}
```

Snippet 14: Access through incorrect type

Here, we dereference a pointer `T*` but the object at that memory location is actually an `S`. This is UB, even though they have the same layout, because `S` objects are not `T`!

The same is true for any pair of unrelated types:

```
int main() {
  int i = 42;
  float* pf = reinterpret_cast<float*>(&i);
  std::cout << *pf << std::endl;
}
```

Snippet 15: Access through incorrect type is illegal for primitives as well

However, there *is* an exception: the underlying representation of any type can be obtained in terms of `char`s, like so:

```
int main() {
  int i = 42;
  float f;
  char* rep_i = reinterpret_cast<char*>(&i);
  char* rep_f = reinterpret_cast<char*>(&f);
  for (size_t j = 0; j < sizeof(int); ++j) {
    // This is all legal
    rep_f[j] = rep_i[j];
  }
  std::cout << f << std::endl;

  // Equivalently:
  memcpy(&f, &i, sizeof(int));
  std::cout << f << std::endl;

  // Or even shorter:
  std::cout << std::bit_cast<float>(i) << std::endl;
}
```

Snippet 16: Access as `char` is allowed

This sounds slower because we're copying out the bits, but because we're copying the bits into a temporary / local variable, the compiler is able to optimize that away to a single memory load (if from memory) or a register `mov` (for float <-> int bit casts), so this has exactly the same performance as the usual `reinterpret_cast` method.

This also applies for function pointers:

```
int f(int i) {
  return i;
}
```

```
int main() {
  float (*g)(int) = reinterpret_cast<float (*)(int)>(f);
  std::cout << g(42) << std::endl;  // UB
}
```

Snippet 17: Calling a function pointer the wrong way is also UB

It's obvious that this doesn't work if we change the calling convention. For example, changing the number of arguments would obviously cause a mismatch between the caller and the callee.

But it's quite difficult in general to predict when the calling convention changes.

For example, when changing the return type from `int` to `std::vector<int>`, the non-trivial return type calling convention is used, so the return value is not passed via register.

Another example is changing the return type from `int` to `float`, like in the above example. `int` return values are passed via `eax`, but `float` return values are passed via `xmm0`.

It's also technically illegal to call member functions like normal functions, even though we know that ABI wise they're the same on every platform we know of. However to demonstrate this we need to do some magic that we won't explain.

Reference: the magic we speak of – Member Function Pointers and the Fastest Possible C++ Delegates

```
struct S {
  int i;
  int get() {
    return i;
  }
};
```

```
struct mem_func {
  void* ptr;
  ptrdiff_t offset;
};
```

```
int main() {
  S s{42};

  int (*get)(S*) = reinterpret_cast<int (*)(S*)>(  //
      std::bit_cast<mem_func>(&S::get).ptr);
  // "works on my machine"
  std::cout << get(&s) << std::endl;
  // but it's ub.

  // The correct way is to use a member function pointer
  int (S::*get_correct)() = &S::get;
  // To call it, use .* or ->*
  std::cout << (s.*get_correct)() << std::endl;
  S* p = &s;
  std::cout << (p->*get_correct)() << std::endl;
}
```

Snippet 18: Calling member functions wrongly is UB too

5.2 UB 4: Non-const access to const object

For obvious reasons, this is also UB:

```
const int x = 42;
int main() {
  const_cast<int&>(x) = 69;  // UB
  std::cout << x << std::endl;
}
```

Snippet 19: Non-const access to const object

Here, if the `x` write actually happened, it would segfault the program because `x` lives in read-only memory. But the following is also UB:

```
int main() {
  const int* px = new const int(42);
  const_cast<int&>(*px) = 69;  // UB
  std::cout << *px << std::endl;
}
```

Snippet 20: Non-const access to const object which lives in mutable memory

However, that's not to say that using `const_cast` is always wrong. If the original object was not const, but we have a const pointer or reference to it, we can cast away the const-ness to mutate it.

```
int main() {
  const int* px = new int(42);
  const_cast<int&>(*px) = 69;  // OK
  std::cout << *px << std::endl;
}
```

Snippet 21: Non-const access to non-const object through const pointer is actually OK

5.3 UB 5: Access outside the lifetime of an object

This is also obviously UB:

```
struct S {
  int x;
};
```

```
int main() {
  S* s = new S{42};
  std::cout << s->x << std::endl;  // OK

  delete s;
  std::cout << s->x << std::endl;  // UB
}
```

Snippet 22: Accessing dangling pointer

However, what you might not be fully aware of is that there is a period of time where the storage of an object begins, but the object's lifetime has not begun yet. In general, the lifecycle of an object looks like this:

- Allocate storage
- Construct object (lifetime usually starts here)
- Destroy object (lifetime usually ends here)
- Deallocate storage

So if we're between steps 3 and 4, accessing the object still counts as UB:

```
int main() {
  void* storage = malloc(sizeof(S));

  // Call the constructor of S
  std::construct_at(reinterpret_cast<S*>(storage), 42);
  // Pre C++20:
  // new (s) S(42);

  std::cout << reinterpret_cast<S*>(storage)->x << std::endl;  // OK

  // Call the destructor of S
  std::destroy_at(reinterpret_cast<S*>(storage));
  // Pre C++20:
  // s->~S();

  std::cout << reinterpret_cast<S*>(storage)->x << std::endl;  // UB

  free(storage);
}
```

Snippet 23: Accessing destroyed object

That's not too surprising. However, storage can be reused, like so:

- Allocate storage
- Construct object
- Destroy object
- Construct another object
- Destroy another object
- Deallocate storage

```
int main() {
  void* storage = malloc(sizeof(S));

  S* ps = std::construct_at(reinterpret_cast<S*>(storage), 42);
  std::cout << ps->x << std::endl;  // OK
  std::destroy_at(ps);

  S* ps2 = std::construct_at(reinterpret_cast<S*>(storage), 69);
  std::cout << ps2->x << std::endl;  // OK
  std::destroy_at(ps2);

  free(storage);
}
```

Snippet 24: Accessing un-destroyed object

▼ **Pointer reuse shenanigans and `std::launder`**

Note that some details in this box might be slightly off, because while I think I'm correct in what I say here, it seems like resources on this topic are really really sparse and don't really cover all the edge cases.

You might notice that I was careful to use the pointer returned by `std::construct_at`. Can we instead reuse the first pointer for less typing? The answer is that it's ok *sometimes*.

```
int main() {
  void* storage = malloc(sizeof(S));

  S* ps = std::construct_at(reinterpret_cast<S*>(storage), 42);
  std::cout << ps->x << std::endl;  // OK
  std::destroy_at(ps);

  std::construct_at(ps, 69);
  std::cout << ps->x << std::endl;  // OK
  std::destroy_at(ps);

  free(storage);
}
```

Snippet 25: Reusing the same pointer is ok sometimes

In this case, the first object (the one that lives between `construct_at(ps, 42)` and `destroy_at`) is said to be *transparently replaceable* with the second object (the one with `69` in it).

The exact rules for transparently replaceable are here:

Reference: [basic.life]/8 on transparently replaceable

An example of a case where objects are not transparently replaceable is when the object in question is `const`:

```
int main() {
  void* storage = malloc(sizeof(S));

  const S* ps =
      std::construct_at(reinterpret_cast<const S*>(storage), 42);
  std::cout << ps->x << std::endl;  // OK
  std::destroy_at(ps);

  std::construct_at(ps, 69);
  // first const S object
  // is not transparently replaceable with
  // second const S object
  std::cout << ps->x << std::endl;  // UB
  std::destroy_at(ps);

  free(storage);
}
```

Snippet 26: Reusing the same pointer is UB sometimes

If you really really want to use the same pointer, you will need to `std::launder` it whenever you access it:

```
int main() {
  void* storage = malloc(sizeof(S));

  const S* ps =
      std::construct_at(reinterpret_cast<const S*>(storage), 42);
  std::cout << ps->x << std::endl;  // OK
  std::destroy_at(ps);

  std::construct_at(ps, 69);
  // first const S object
  // is not transparently replaceable with
  // second const S object
  //
  // but std::launder forces re-pointing at new 69 object
  // so this is OK
  std::cout << std::launder(ps)->x << std::endl;  // OK
  std::destroy_at(std::launder(ps));

  free(storage);
}
```

Snippet 27: Launder forces pointers to old objects to point at new objects

5.4 UB 6: Access out of bounds

Another obvious UB is buffer overflow, duh:

```
struct S {
  int x;
};
```

```
int main() {
  S ss[4]{{0}, {1}, {2}, {3}};
  std::cout << ss[5].x << std::endl;  // UB, duh
}
```

Snippet 28: Buffer overflow is UB

In this case, the UB happens because pointer arithmetic past the past-the-end element is UB:

```
ss[5];
// is equivalent to
*(ss + 5);
// because ss is array type

ss + 5;
// is UB because we've went past the past-the-end element
```

Snippet 29: Going past the past-the-end element is UB

However, the following is still UB because while the `+` is no longer UB, we cannot `*` a past-the-end element:

```
int main() {
  S ss[4]{{0}, {1}, {2}, {3}};
  std::cout << ss[4].x << std::endl;  // UB, duh
}
```

Snippet 30: Dereferencing a pointer to the past-the-end element is UB

This is STILL UB even if there is actually an element of the correct type at that location.

```
struct T {
  S ss[4]{{0}, {1}, {2}, {3}};
  S wow{4};
};
```

```
int main() {
  T t{};
  if (&t.wow == t.ss + 4) {
    std::cout << t.ss[4].x  // Still UB
              << std::endl;
  }
}
```

Snippet 31: Dereferencing a pointer to the past-the-end element is UB, even if we do actually have an object there

5.5 UB 7: Unaligned memory access

This pretty much never happens until you start dealing with allocators, but the following is UB:

```
struct S {
  int x;
};
```

```
int main() {
  char storage[100];
  S* s1 = std::construct_at(reinterpret_cast<S*>(storage));
  char* c2 = std::construct_at(reinterpret_cast<char*>(storage + 4));
  S* s3 = std::construct_at(reinterpret_cast<S*>(storage + 5));

  // Either s1 or s3 (or both) will be unaligned, which is UB
}
```

Snippet 32: Constructing an object at the wrong alignment

6 “Strict” aliasing “rule”

▼ **The strict aliasing rule is neither a rule nor strict.**

The strict aliasing rule says that in order for our program to compile correctly, we need to obey the aliasing rules (partially covered above!) *strictly*.

However, since lots of legacy code has been written that doesn’t obey the aliasing rules strictly, compilers often bend to the will of the masses and will try their best to compile such cases of undefined behaviour to something that’s as intuitive as possible. Thus the strict aliasing rule is not strict.

Furthermore, it’s not really a rule, since it’s obvious that when your program contains undefined behaviour, the behaviour of the program is undefined. It’s more of a recommendation that’s along the lines of “undefined behaviour is bad, you should try not to have undefined behaviour in your program.”

Complaining aside, in this section we’ll cover some less intuitive consequences of breaking “the aliasing rules”, aka triggering the UB we’ve described in the previous section.

In UB 3 we covered that accessing anything through the “wrong type” is UB. This has some interesting implications for the optimizer. For example, in the following function:

```
void count_nonzero(  //
    size_t* iters,
    int* nums) {
  *iters = 0;
  while (*nums != 0) {
    ++nums;
    // Many writes to memory
    ++*iters;
  }
}
```

```
void count_nonzero_opt(  //
    size_t* iters,
    int* nums) {
  if (*nums == 0) {
    *iters = 0;
  } else {
    size_t count = 1;
    while (nums[count] != 0) {
      ++count;
    }
    // One write to memory
    *iters = count;
  }
}
```

```
$ cat strict-1.clang++.s \
    | c++filt \
    | sed 's/:\s*#.*$/:/' \
    | sed -n \
  '/^count_nonzero(/,/func_end/p'
count_nonzero(unsigned long*, int*):
# %bb.0:
    cmpl    $0, (%rsi)
    je  .LBB0_1
# %bb.2:
    xorl    %ecx, %ecx
    .p2align    4, 0x90
.LBB0_3:
    leaq    1(%rcx), %rax
    cmpl    $0, 4(%rsi,%rcx,4)
    movq    %rax, %rcx
    jne .LBB0_3
# %bb.4:
    movq    %rax, (%rdi)
    retq
.LBB0_1:
    xorl    %eax, %eax
    movq    %rax, (%rdi)
    retq
.Lfunc_end0:
```

```
$ cat strict-1.clang++.s \
    | c++filt \
    | sed 's/:\s*#.*$/:/' \
    | sed -n \
  '/^count_nonzero_opt(/,/func_end/p'
count_nonzero_opt(unsigned long*, int*):
# %bb.0:
    cmpl    $0, (%rsi)
    je  .LBB1_1
# %bb.2:
    xorl    %ecx, %ecx
    .p2align    4, 0x90
.LBB1_3:
    leaq    1(%rcx), %rax
    cmpl    $0, 4(%rsi,%rcx,4)
    movq    %rax, %rcx
    jne .LBB1_3
# %bb.4:
    movq    %rax, (%rdi)
    retq
.LBB1_1:
    xorl    %eax, %eax
    movq    %rax, (%rdi)
    retq
.Lfunc_end1:
```

Snippet 33: Code motion enabled due to aliasing rules

The compiler was able to move the `++*iters` out of the loop as it knows that `iters` and `nums` cannot point to the same part of memory. The reason for this is because `size_t` and `int` are different types.

If they did point at the same piece of memory, then we would either be accessing an `int` through a `size_t` pointer, or vice versa, or accessing just some completely wrong type, all of which are UB.

This tells the compiler that doing `++*iters` will not affect the result of reading `*nums` at all, and so there’s no difference between writing to `iters` inside or outside the loop.

Compare that to if we had to count the number of nonzero values of `char`s, i.e. we’re just implementing `strlen`:

```
void count_nonzero(  //
    size_t* iters,
    char* nums) {
  *iters = 0;
  while (*nums != 0) {
    ++nums;
    // Many writes to memory
    ++*iters;
  }
}
```

```
void count_nonzero_opt(  //
    size_t* iters,
    char* nums) {
  if (*nums == 0) {
    *iters = 0;
  } else {
    size_t count = 1;
    while (nums[count] != 0) {
      ++count;
    }
    // One write to memory
    *iters = count;
  }
}
```

```
$ cat strict-2.clang++.s \
    | c++filt \
    | sed -n \
  '/^count_nonzero(/,/func_end/p' \
    | sed 's/:\s*#.*$/:/'
count_nonzero(unsigned long*, char*):
# %bb.0:
    movq    $0, (%rdi)
    cmpb    $0, (%rsi)
    je  .LBB0_3
# %bb.1:
    movl    $1, %eax
    .p2align    4, 0x90
.LBB0_2:
    movq    %rax, (%rdi)
    cmpb    $0, (%rsi,%rax)
    leaq    1(%rax), %rax
    jne .LBB0_2
.LBB0_3:
    retq
.Lfunc_end0:
```

```
$ cat strict-2.clang++.s \
    | c++filt \
    | sed 's/:\s*#.*$/:/'
    | sed -n \
  '/^count_nonzero_opt(/,/func_end/p' \
count_nonzero_opt(unsigned long*, char*):
# %bb.0:
    cmpb    $0, (%rsi)
    je  .LBB1_1
# %bb.2:
    xorl    %ecx, %ecx
    .p2align    4, 0x90
.LBB1_3:
    leaq    1(%rcx), %rax
    cmpb    $0, 1(%rsi,%rcx)
    movq    %rax, %rcx
    jne .LBB1_3
# %bb.4:
    movq    %rax, (%rdi)
    retq
.LBB1_1:
    xorl    %eax, %eax
    movq    %rax, (%rdi)
    retq
.Lfunc_end1:
```

Snippet 34: Since `char` can alias, optimisation cannot happen automatically

7 Unsequenced and data races

The key idea of unsequenced races and data races is that it is undefined behaviour for something to happen "at the same time" as another thing, or generally speaking, if there is no ordering relationship between two expressions.

7.1 UB 9: Unsequenced races

Unsequenced races happen on a single thread of execution (ie. *not* potentially concurrent), when two expressions are unsequenced with respect to each other, and either:

1. both of them have side effects on memory
2. one of them has a side effect on memory, and the other uses the value at that memory location

Note that *unsequenced* is distinct from *indeterminately sequenced*; in the latter case, the compiler is allowed to emit code that makes the CPU *interleave* the two computations, whereas for indeterminately sequenced expressions, this is not allowed. Even so, the actual sequence in which the expressions are evaluated can vary, even on the next invocation of the same expression.

Let's look at some examples of unsequenced races first:

```
int i = 69;

// UB: operands to '+' are unsequenced,
// and '++i' has a memory side effect which is unsequenced
// with the value computation of 'i'
++i + i;

// UB: similar reason as above, except now
// both operands have memory side effects
++i + i++;

// UB: same as above again
f(i++) + g(i++);
```

Snippet 35: Examples of unsequenced races (aka UB)

In C++17 and later, a few more rules were added to make some things definitely sequenced, and some things *indeterminately* sequenced, so the number of *unsequenced behaviours* decreased. Some examples:

```
// not UB: function arguments are *indeterminately sequenced*
// with respect to each other
f(i++, i++);

// not UB: right side of '=' is always sequenced-before the left side
// (note: this includes compound assignments like '+=')
i = i++;

int* p = &i;

// not UB: array-part of [] (here, 'p++') is always sequenced-before
// the subscript ('*p' here)
(p++)[*p] = 10;

// not UB: '<<' and '>>' always sequence their left operands before
// their right operands
std::cout << i++ << i++ << "\n";
```

Snippet 36: Examples of UB pre-C++17 that are not UB now

7.2 UB 10: Data races

Data races occur in multithreaded programs, and you can think of them as the logical extension of unsequenced races but on multiple threads. More formally, a data race occurs when:

1. two potentially concurrent threads are accessing the same memory location
2. at least one of those accesses is not atomic
3. at least one access is a write
4. neither access *happens-before* the other

Rather than explain with words, let's just look at some code examples:

```
int normal;
std::atomic<int> atom;

using std::thread;
using stdmo = std::memory_order;
```

```
// UB: both threads modify `normal`,
// neither of them are atomic,
// and there is no happens-before
// relation between them
auto t1 = thread([] {  //
  normal++;
});
auto t2 = thread([] {  //
  normal++;
});
```

```
// not UB: both accesses are
// atomic, so there is no
// data race
auto t3 = thread([] {  //
  atom.fetch_add(1);
});
auto t4 = thread([] {  //
  atom.fetch_add(1);
});
```

```
// not UB: due to acquire-release semantics, 'A' (the store) in t5
// happens-before 'B' (the read) in t6.
auto t5 = thread([] {
  normal = 10;  // stmt A
  atom.store(8, stdmo::release);
});

auto t6 = thread([] {
  while (atom.load(stdmo::acquire) != 8)
    ;

  int k = normal;  // stmt B
  std::cout << "k = " << k << "\n";
});
```

```
// UB: the acquire load of 'atom' doesn't actually check that it
// loaded '5' which was released so 'A' does not happen-before 'B'!
auto t7 = thread([] {
  normal = 10;  // stmt A
  atom.store(5, stdmo::release);
});

auto t8 = thread([] {
  atom.load(stdmo::acquire);

  int k = normal;  // stmt B
  std::cout << "k = " << k << "\n";
});
```

Snippet 37: Some examples of what constitutes a data race (and what doesn't)

For more details on how exactly to use atomics in C++ and how happens-before relationships are defined between them, take CS3211 :D.