These exercises are mostly theoretical in nature. Some of them requires you to do some research on your own, since not everything is covered during the seminars.

1. Given the following function overloads:

```
1 void fun(double, int, int);     // #1
2 void fun(int, int, double);     // #2
3 void fun(double, int, double); // #3
```

   For these function calls, determine which – if any – of the overloads above is called:

```
1 fun(0.0, 0.0, 0.0); // a
2 fun(0, 0, 0);       // b
3 fun(0, 0.0, 0.0f);  // c
4 fun(0.0, 0.0, 0);   // d
5 fun(0.0, 0, 0.0f);  // e
```

   Explain for each of them *why* the overload is called or why it failed.

Answer 1

a Calls `#3`.
Overload `#1` requires two conversion (second and third to `int`).
Overload `#2` requires two conversions (first and second to `int`).
Overload `#3` requires one conversion (second to `int`).
The third call requires the fewest conversions.

b None, the call is ambiguous.
Overload `#1` requires one conversion (first to `double`).
Overload `#2` requires one conversion (third to `double`).
Overload `#3` requires two conversions (first and third to `double`).
The fewest required conversions is one, but two overloads has that so none can be picked.

c Calls `#2`.
Overload `#1` requires three conversions (first to `double`, second and third to `int`).
Overload `#2` requires two conversions (second to `int` and third to `float`).
Overload `#3` requires three conversions (second to `int`, first and third to `double`).
`#2` requires the fewest conversions.

d Calls `#1`.
Overload `#1` requires one conversion (second to `int`).
Overload `#2` requires three conversions (first and second to `int` and third to `double`).
Overload `#3` requires two conversions (second to `int` and third to `double`).
`#1` requires the fewest conversions.

e Calls `#3`.
Overload `#3` matches exactly, and every other overload requires at least one conversion.

2. Explain all type conversions that occur in this example.

```
1  #include <iostream>
2  #include <string>
3
4  int sum(double const* numbers,
5  unsigned long long size)
6  {
7    double result{};
8    for (unsigned i{}; i < size; ++i)
9      result += static_cast<int>(numbers[i]);
10   return result;
11 }
12
13 int main()
14 {
```

```
15    std::string message{};
16    message = "Enter a number: ";
17
18    double numbers[3];
19    for (int i{0}; i < 3; ++i)
20    {
21      std::cout << message;
22      if (!(std::cin >> numbers[i]))
23        return true;
24    }
25
26    std::cout << sum(numbers, 3) + 1.0 << std::endl;
27  }
```

Answer 2

line 8: Comparisons can only be performed with compatible types, so `i` is *promoted* to `unsigned long long` so that it can be compared with `size`.

line 10: With `static_cast` we are casting `numbers[i]` to an `int` and then adding the value to `result`. But `result` is of type `double`, which has higher accuracy than `int`, so the casted value is then converted to an `double`. So we did a *floating-to-integer* followed by an *integer-to-floating* conversion.

line 12: `sum` returns an `int`, but we are returning `result` which is of type `double`. Due to this the compiler has to do a *floating-to-integer* cast.

line 18: `"Enter a number: "` is a C-string literal, i.e. of type `char const*` so one might think this would lead to a conversion, but `std::string` has an overload of `operator=` that handles `char const*` so it is actually not a conversion.

line 24: `std::cin >> numbers[i]` returns `std::cin` (an `std::istream`) which is the negated using the `!` operator, which triggers a conversion from `std::istream` to `bool`.

line 26: `main` returns an `int`, so `true` will be casted to the value `1`.

line 30: `sum` returns an `int` that is added to the `double` value `1.0`. Since we are adding `int` and `double` we have to convert them to the same type. The compiler will do a *integer-to-floating* conversion on the `int` which means that the result of the addition will be a `double`.

line 30: `numbers` is an array of `doubles` with size `3`. However it is passed in to `sum` which takes a `double const*`. The compiler will first perform an *array-to-pointer* conversion to convert `numbers` into `double*`. However it doesn't stop there, the compiler will also perform a *qualification conversion* to add `const`.

line 30: The literal `3` is an `int`, but the second parameter of `sum` is of type `unsigned long long`, so `3` will be *promoted* to `unsigned long long`.

3. Why do you think it is well-defined behaviour what happens when an unsigned

integer value under- or overflows?

Why is the same not true for signed integer values?

I.e. why is the C++ standard comfortable with defining what happens if you subtract one from the smallest, or add one to the largest, possible unsigned value but *not* for signed?

Answer 3
An unsigned integer type directly interprets its bits as base two numbers.
Note that, in base 2 it holds that:

$$\underbrace{111\ldots111}_{N} + 1 = 1\underbrace{000\ldots000}_{N}$$

so if there are N bits in a value, the left-most 1 would be lost.
Similarily it can be seen that:

$$0 - 1 = \underbrace{111\ldots111}_{N}.$$

The same cannot be said for *signed* integers since the representation of negative numbers might be different on different platforms.
For example: in two's complement, the largest (most positive) value is given by

$$\underbrace{0111\ldots111}_{N}(= 2^{N-1} - 1).$$

Adding one to this value gives

$$\underbrace{1000\ldots000}_{N},$$

which happens to be the most negative value ($= 02^{N-1}$).
Using a sign bit will result in the same largest number, but the result of adding one is instead interpreted as -0.
So depending on representation overflow is interpreted differently (which is also true for underflow).

4. Study the code below:

```
#include <cstdlib> // for std::abort()
#include <iostream>

int main()
{
    std::cout << "Hello ";
    std::abort();
    std::cout << "there!" << std::endl;
}
```

(a) Explain why the code (likely) **DOESN'T** print `Hello` to the terminal, even though the program isn't aborted until *after* the output statement.

(b) How do you fix this program (without removing any code and without changing the intended behaviour of the program) so that it prints `Hello` before it crashes?

Hint: There are at least *two* fundamentally different ways of fixing this.

Answer 4

(a) `std::cout` is buffered, which means it doesn't actually print anything to the terminal until it is flushed.
What (likely) happens is that the contents of the buffer isn't flushed unless the line is completed (by inserting `std::endl` or potentially `'\n'`), *or* until `std::flush` is used.
Since none of these conditions are met for the first print statement, `Hello` is never actually printed to the terminal.
Note: Strictly speaking the standard doesn't stop any implementation (compiler) from flushing after every printed character, but if your compiler does that you should change compiler since that is a *HUGE* hit on performance.

(b) The first solution is to simply use `std::flush`:

```
#include <cstdlib> // for std::abort()
#include <iostream>

int main()
{
    std::cout << "Hello " << std::flush;
    std::abort();
    std::cout << "there!" << std::endl;
}
```

The second solution is to use the *unbuffered* stream `std::cerr`:

```
#include <cstdlib> // for std::abort()
#include <iostream>

int main()
{
    std::cerr << "Hello ";
    std::abort();
    std::cerr << "there!" << std::endl;
}
```

This is a worse solution though in terms of performance. An unbuffered stream is one that is guaranteed to flush after each individual printed value.