1 Basic language constructs

The exercises here are (mostly) more theoretical in nature. Do note that some of these questions requires you to do some research on your own, since not everything is covered during the seminars.

1. Given

```
int n{};

int& foo(int& n)
{
  return n;
}

int bar(int& n)
{
  return n;
}

int&& move(int& n)
{
  return static_cast<int&&>(n);
}
```

what is the value category of the following expression:

(a) `n`

(b) `n + 1`

(c) `n = 2`

(d) `double{}`

(e) `move(n)`

(f) `foo(n)`

(g) `bar(n)`

(h) `foo(n = bar(n))`

Answer 1

(a) lvalue
(b) prvalue
(c) lvalue
(d) prvalue
(e) xvalue
(f) lvalue
(g) prvalue
(h) lvalue

2. Write declarations for the following variables:

   (a) a pointer to `char`
   (b) an array with 10 `int`
   (c) a pointer to an array with 3 elements of type `std::string`
   (d) a pointer to a pointer to `char`
   (e) a constant `int`
   (f) a pointer to a constant `int`
   (g) a constant pointer to an `int`

Answer 2

(a) `char* ptr;`
(b) `int array[10];`
(c) `std::string (*array)[3];`
(d) `char** ptr;`
(e) `int const n;`
(f) `int const* ptr;`
(g) `int* const ptr;`

3. Write a program that initializes all the variables from exercise 2.

Answer 3

```
#include <string>
int main()
{
  std::string words[3]{"hello", "world", "!"};

  char* ptr{new char{'A'}};
  int array[10]{1};
  std::string (*words_ptr)[3]{&words};
  char** ptr_ptr{&ptr};
  int const n{1};
  int const* cptr1{&n};
  int* const cptr2{&array[0]};
}
```

4. For each of the following declarations: is it legal? If it is, what does it declare? The last three can be skipped since they are very complex.

   (a) `int a(int i);`

   (b) `int a(int);`

   (c) `int a(int (i));`

   (d) `int a(int (*i)());`

   (e) `int a(int* const);`

   (f) `int a const();`

   (g) `int a(int const* (*)());`

   (h) `int a(int (*)(int));`

   (i) `int a(int (*i)(int)[10]);`

   (j) `int a(int (*i[10])());`

   (k) `int a(int (&(*i)())[10]);`

   Note: Please, please, **PLEASE** don't ever do these kinds of declarations! Let this exercise serve as a demonstration as to what might happen if you don't think about readability. See next exercise for how you can make it more readable.

Answer 4

(a) Valid. A function with a named `int` parameter that returns an `int`.
(b) Valid. A function with an `int` parameter (name omitted) that returns an `int`.
(c) Valid. A function with a named `int` parameter that returns an `int`, with reduntant parenthesis around the parameter name (identifier).
(d) Valid. A function returning `int` that has an argument which is a pointer to a function returning `int` with no parameters.
(e) Valid. A function returning `int` that takes a constant pointer to an `int` as a parameter.
(f) Illegal.
(g) Valid. A function returning `int` that takes a (unnamed) parameter that is a pointer to a function with no arguments that return a pointer to a constant `int`.
(h) Valid. A function returning `int`, that takes a pointer to a function with one `int` parameter and that returns an `int`.
(i) Illegal.
(j) Valid. A function returning `int`, that takes an array of 10 pointers to functions taking no parameters and returning `int`.
(k) Valid. A function returning `int`, that takes a pointer to a function with no arguments that return a reference to an array of 10 `int`.

5. What does the following mean? How can it be used?

```
using x = int(&)(int, int);
```

Answer 5
`x` is a reference to a function taking two `int`s and returning an `int`.
It can be used as this:

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

using x = int(&)(int, int);

int main()
{
  x my_fun {add};
  return my_fun (1, 2); // will call add(1, 2)
}
```

6. What *size* does the character array `msg` have? What *length* does the C-string `"Hello world!"` have?

```
char msg[] { "Hello world!" };
```

Answer 6
It has the size 13 since it is initialized from a C-string which must contain a '\0' as the last element. The string itself has length 12.

7. Which of these are valid variable initializations? For those that are valid, what are their values? For those that are invalid explain why.
   - `int i1{};`
   - `int i2(2);`
   - `int i3 = 1;`
   - `int i4 = {};`
   - `int i5();`
   - `std::string str1{};`
   - `std::string str2("hello");`
   - `std::string str4(3, 'a');`
   - `std::string str5 = str2;`
   - `float f1{5.37e100};`
   - `float f2 = 5.37e100;`
   - `float f3{1738335806};`

Answer 7

- Valid. The value is `0`.
- Valid. The value is `2`.
- Valid. The value is `1`.
- Valid. The value is `0`.
- Invalid, because *most vexing parse* forces the compiler to interpret this as a declaration of a function returning `int` with no parameters.
- Valid. The value is `""`.
- Valid. The value is `"hello"`.
- Valid. The value is `"aaa"`.
- Valid. The value is `"hello"`.
- Invalid because *brace-initialization* checks for *narrowing conversions* and since `5.37e100` is a `double` which has higher accuracy than `float` forcing a narrowing conversion.
- Valid (copy initialization does not check for narrowing conversions). The value is undefined since `float` cannot accurately represent `5.37e100`, but it is probably `inf`.
- Invalid, `float` cannot accurately represent `1738335806` which would lead to narrowing conversion.

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

```
#include <iostream>
#include <string>

int sum(double const* numbers,
unsigned long long size)
{
  double result{};
  for (unsigned i{}; i < size; ++i)
  {
    result += static_cast<int>(numbers[i]);
  }
  return result;
}

int main()
{
  std::string message{};
  message = "Enter a number: ";

  double numbers[3];
  for (int i{0}; i < 3; ++i)
  {
    std::cout << message;
    if (!(std::cin >> numbers[i]))
```

```
25      {
26        return true;
27      }
28    }
29
30    std::cout << sum(numbers, 3) + 1.0 << std::endl;
31  }
```

Answer 8

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 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`.

9. Explain all conversions in the following example. Why do they occur?

```
#include <iostream>
int foo()
{
  return 0;
}

using function_t = int (*)();

int main()
```

```
{
  function_t f {main};

  char c{'A'};
  std::cout << c + 1 << std::endl;
  std::cout << c + 'A' << std::endl;

  std::cout << 0 - 1u << std::endl;

  f = foo;
  std::cout << f() << std::endl;
  std::cout << f << std::endl;
  std::cout << reinterpret_cast<void*>(f) << std::endl;
}
```

Answer 9

line 11 `main` is a function, while `function_t` represents a function-pointer, so the compiler must perform a *function-to-pointer*.

line 14 we add a `char` and an `int`, therefore the compiler must *promote* the `char` to an `int` so that the result of the addition doesn't run the risk of overflowing.

line 15 When performing arithmetic operations on types smaller than `int`, the result will be converted to an `int`. This is done for various reasons:

- Speed: `int` represents the type for which we get the most effective arithmetic operations in the CPU
- Safety: a type smaller than `int` can overflow easily since there are few values, so to make sure that it doesn't overflow we represent it as an `int` instead
- Consistency: make sure that arithmetic expressions are performed in a similar way each type
- Composability: if we have larger arithmetic expressions we want everything to work together without having to manually convert subexpressions. Example: `3*4 + 6u * ('A'+ 'B');`

line 17 We are performing an arithmetic operation on an `int` and an `unsigned`. `unsigned` has higher accuracy (in the positive values), so the `int` is converted to `unsigned` forcing the entire addition to produce an `unsigned`. This will cause an underflow, making the value wrap-around to the largest possible value of type `unsigned`.

line 19 Again, `f` is a function-pointer while `foo` is a function, so the compiler must perform a *function-to-pointer* conversion.

line 21 `operator<<` for streams doesn't have an overload for function-pointers, so the function-pointer will be converted to `bool` through *narrowing-conversion*.

line 22 The function-pointer `f` points to some address, which we can *reinterpret cast* as a data-pointer instead of a function-pointer. This way we can actually print where in memory `f` is pointing to, since streams can print data-pointers.