Exercise 4.1

The `+` operator has the lowest precedence in this expression and the `*` and `/` operator have the same precedence so the expression `5 + 10 * 20/2` is equivalent to `5 + ((10 * 20) / 2)` so the value returned is 105.

Exercise 4.2

(a) `*((vec.begin)())`

(b) `(*((vec.begin)())) + 1`

Exercise 4.3

I think it's an acceptable trade-off if we don't use functions with side-effects, sadly this restriction is hard to enforce and it's the source of many errors.

Exercise 4.4

```
(((12 / 3) * 4) + (5 * 15)) + ((24 % 4) / 2)
```

```cpp
// precedence.cpp

#include <iostream>

int main()
{
    std::cout << 12 / 3 * 4 + 5 * 15 + 24 % 4 / 2 << std::endl;
    std::cout << (((12 / 3) * 4) + (5 * 15)) + ((24 % 4) / 2) << std::endl;

    return 0;
}
```

Exercise 4.5

(a) `-30 * 3 + 21 / 5` has value -86.

(b) `-30 + 3 * 21 / 5` has value -18

(c) `30 / 3 * 21 % 5` has value 0

(d) `-30 / 3 * 21 % 4` has value -2

Exercise 4.6

If `i` is an `int`, we can use the expression `i % 2` which evaluate to `0` if `i` is even and `1` if `i` is odd. This expression can then be converted to `bool`: it's `true` if `i` is odd and `false` if `i` is even.

Here is a little program using this expression:

```cpp
// even_odd.cpp

#include <iostream>

int main()
{
    std::cout << "Enter an integer:" << std::endl;

    int i;
    if (std::cin >> i) {
        if (i % 2)
            std::cout << i << " is odd!" << std::endl;
        else
            std::cout << i << " is even!" << std::endl;
    } else {
        std::cerr << "Input error" << std::endl;
        return -1;
    }

    return 0;
}
```

Exercise 4.7

On my machine `int` are 32 bits. Suppose we have `int i = 2147483647;` which is the maximum `int` on my machine. Then all the three following expression would overflow (hence having undefined behavior):

`++i`

`i * 2`

`i + i - i`

Exercise 4.8

With the logical AND the first operand is always evaluated and the second operand is evaluated if and only if the first operand is true.

With the logical OR the first operand is always evaluated and the second operand is evaluated if and only if the first operand is false.

With the equality operator, both operand are always evaluated.

Exercise 4.9

The first operand of the logical AND (`&&`) operator evaluates to true because `cp` points to a valid object (hence it's non-null). `cp` points to a character different from the null character `'\0'`.hence the second operand is the character which is not the null character `'\0'` hence it also evaluates to true. Conclusion: the `if` condition is `true`.

Exercise 4.10

```cpp
// equal_42.cpp

#include <iostream>

int main()
{
    int i = 0;

    std::cout << "Try to find the special int between 0 and 100:" << std::endl;

    while ((std::cin >> i) && i != 42) {
        std::cout << "Incorrect, input error or wrong number!"
                  << "Think harder and try again:" << std::endl;
    }

    if (i == 42) {
        std::cout << "Correct number!" << std::endl;
        return 0;
    } else {
        std::cerr << "Input error." << std::endl;
        return -1;
    }
}
```

Exercise 4.11

```cpp
// chained_greater.cpp

#include <iostream>
```

```cpp
int main()
{
    int a = 4, b = 3, c = 2, d = 1;

    if (a > b && b > c && c > d)
        std::cout << "a > b > c > d" << std::endl;
    else
        std::cout << "b > a or c > b or d > c" << std::endl;

    return 0;
}
```

Exercise 4.12

The `<` operator has higher precedence than the `!=` operator hence expression is equivalent to `i!=(j<k)` which is true if and only if `i` has value 0 and `j>=k` or `i` has value 1 and `j<k`.

Exercise 4.13

(a) Both `d` and `i` has value 3.

(b) `i` has value 3 and `d` has value 3.5.

Exercise 4.14

The first `if` test is not legal as the literal `42` is not an lvalue hence we can't assign to it.

The second `if` value assign the value 42 to `i` and use the new value of `i` as the condition. Since `i` is non-zero the condition value of the `if` statement is `true`.

Exercise 4.15

The statement assign the null pointer to the variable `pi` then try to assign the value of `pi` (the null pointer) to the variable `ival` of type `int`. Since conversion from pointer to `int` is not allowed this assignment is illegal. We could correct it by splitting this statement in two: `dval = ival = 0;` and `pi = 0;`.

Exercise 4.16

(a) I think the author wanted to assign the value of `getPtr()` to the variable `p` and then enter the `if` only if this pointer is non-null. So I think the expression

should be `if ((p = getPtr()) !=0)`.

(b) The expression `i==1024` will always evaluate to a `true` value (making the `if` statement useless) so I think the correct expression should be `if (i==1024)`.

Exercise 4.17

Prefix increment increments the operand and yields the incremented object. Postfix increment also increments the operand but yield a copy of the object before the increment.

Exercise 4.18

It would not print the first value of the vector and it would print the first negative value (if any).

Exercise 4.19

(a) This expression yield `true` if and only if `ptr` is not a null pointer and `ptr` is not pointing to a zero `int`. This expression also increment the pointer `ptr` if it's not a null pointer.

(b) This expression is correct, if will increment `ival` and return `true` if and only if `ival` doesn't have the value -1 or 0.

(c) This expression is not correct, it modify a variable and use this variable multiple times, it's undefined behavior in this case as there is no guarantee that the left operand of `<=` will be evaluated first or after the second operand.

Exercise 4.20

(a) This expression is valid if the iterator is referencing to a valid object of the vector. It yield the `string` indicated by `iter` and increment `iter`.

(b) This expression is not valid as `*iter` is a `string` and can't be incremented.

(c) Invalid because `iter` is an `iterator` and has no `empty` member function.

(d) Valid if `iter` is an iterator referencing to a valid element from the vector. It yield a `bool`, `true` if the `string` pointed by `iter` is empty, `false` otherwise.

(e) Invalid as `*iter` is a `string` and can't be incremented.

(f) Valid if `iter` points to a valid element of the vector, same result as (f) but also increment `iter`.

Exercise 4.21

```cpp
// double_odd.cpp

#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec = {1, 2, 3, 4, 5};

    for (auto i : vec)
        std::cout << i << " ";
    std::cout << std::endl;

    for (auto &ri : vec) {
        if (ri % 2 == 1)
            ri *= 2;
    }

    for (auto i : vec)
        std::cout << i << " ";
    std::cout << std::endl;

    return 0;
}
```

Exercise 4.22

```cpp
// low_pass.cpp

#include <iostream>

int main()
{
    const int grade = 70;
    const char * const finalgrade_conditional =
        (grade > 90) ? "high pass"
                     : (grade > 75) ? "pass"
                                    : (grade >= 60) ? "low pass"
                                                    : "fail";
    const char *finalgrade_if;
    if (grade > 90)
        finalgrade_if = "high pass";
    else if (grade > 75)
```

```
        finalgrade_if = "pass";
    else if (grade >= 60)
        finalgrade_if = "low pass";
    else
        finalgrade_if = "fail";

    std::cout << finalgrade_conditional << std::endl
              << finalgrade_if << std::endl;

    return 0;
}
```

Exercise 4.23

The second statement is equivalent to `string p1 = (s + s[s.size() - 1] = 's') ? "" : "s";=` which will not compile as we can't test equality of a `string` with a `char`. Instead we should write `string p1 = s + (s[s.size() - 1] = 's' ? "" : "s");=`.

Exercise 4.24

If the conditional operator was left associative the statement would be equivalent to:

```
finalgrade = ((grade > 90) ? "high pass" : (grade < 90)) ? "fail"
                                                         : "pass";
```

And this statement would be an error because a `const char *` and `bool` can't be converted to a common type.

Exercise 4.25

`~'q'` can be an `int` or `unsigned int` depending on the implementation so the bitwise shift is undefined behavior.

Exercise 4.26

`unsigned int` are only guaranteed to be at least `16` bits so if we use `unsigned` instead of `long unsigned` our program could not be able to store all the student grades.

Exercise 4.27

The three last bits of the binary representation of `ul1` and `ul2` are respectively `011` and `111` with all the previous bits set to zero. With this information it's easy to deduce the results of the next expressions.

(a) 3 (b) 7 (c) 1 (d) 1

Exercise 4.28

```cpp
#include <iostream>

int main()
{

    std::cout << "    type      |   size\n"
              << "--------------|----------\n"
              << " bool         |  " << sizeof(bool)        << "\n"
              << " char         |  " << sizeof(char)        << "\n"
              << " wchar_t      |  " << sizeof(wchar_t)     << "\n"
              << " char16_t     |  " << sizeof(char16_t)    << "\n"
              << " char32_t     |  " << sizeof(char32_t)    << "\n"
              << " short        |  " << sizeof(short)       << "\n"
              << " int          |  " << sizeof(int)         << "\n"
              << " long         |  " << sizeof(long)        << "\n"
              << " long long    |  " << sizeof(long long)   << "\n"
              << " float        |  " << sizeof(float)       << "\n"
              << " double       |  " << sizeof(double)      << "\n"
              << " long double  |  " << sizeof(long double) << "\n";
    return 0;
}
```

Exercise 4.29

This code will print two lines. On the first line it will print the size of the array `x` in bytes (that is 10 times the size of an `int`) divided by the size of an `int`, the result will be the number of integers `x` can hold that is 10. The second line will print the size of a pointer to `int` divided by the size of an `int`. On my machine it should print `2` because pointers are 64 bits and `int` are 32 bits.

Testing this code gives me the result I expected. Note: my compiler (GCC) with warnings enabled warned me during compilation that the `sizeof(p) / sizeof(*p)` would not compute the length of the array.

Exercise 4.30

(a) `(sizeof x) + y`

(b) `sizeof ((p->mem)[i])`

(c) `(sizeof a) < b)`

(d) `sizeof (f())`

Exercise 4.31

We used prefix instead of postfix because it avoid making a copy (even if the compiler can optimize this), it's better style and make the code more consistent if we always use prefix instead of postfix when the result does not matter.

```
// postfix.cpp

#include <iostream>
#include <vector>

int main()
{
    std::vector<int> ivec {7, 8, 9};
    std::vector<int>::size_type cnt = ivec.size();
    //  assign values from size...1 to the elements in ivec
    for (std::vector<int>::size_type ix = 0; ix != ivec.size(); ix++, cnt--)
        ivec[ix] = cnt;

    for (int i : ivec)
        std::cout << i << " ";
    std::cout << std::endl;

    return 0;
}
```

Exercise 4.32

The `for` loop start by defining one pointer (`ptr`) to the beginning of the array and one index (`ix`) equal to zero. At the end of each iteration both the pointer `ptr` and the index `ix` are incremented: the pointer points to the next element of the array and the `ix` represent the index of the next element of the array `ia`. The `for` loop ends when either `ix` is equal to the size of the array `ia` or `ptr` points to the element past the end of the array `ia` (both conditions will be reached at the same time if neither `ix` or `ptr` are modified inside the body of the `for` loop).

This `for` loop could for example allow to iterate through the array `ia` and use both the notation `ia[ix]` and `*ptr` to access the elements.

Exercise 4.33

According to the operator precedence table the expression is equivalent to `(someValue ? (++x, ++y) : --x), --y`. If `someValue` is `true` both `x` and `y` are incremented then `y` is decremented (resulting in `y` being unchanged). If `someValue` is `false` then `x` and `y` are decremented. In both cases the last value of `y` is the value of the whole expression.

Exercise 4.34

(a) `fval` is converted to `bool`, if `fval` is 0 the condition is false, if `fval` is not 0 the condition is true.

(b) `ival` is converted to `float` then the result of the floating point addition is converted to `double`.

(c) `cval` is converted to `int` then the result of the multiplication is converted to `double` (to be added to `dval`).

Exercise 4.35

(a) `'a'` is promoted to `int` then the result of the addition is converted to `char`.

(b) `ival` is converted to `double`, `ui` is converted to `double` and the result of the subtraction is converted to `float`.

(c) `ui` is converted to `float` then the result of the multiplication is converted to `double`.

(d) The operator `+` is left associative so the expression on the right hand of the assignment is equivalent to `(ival + fval) + dval` hence `ival` is converted to `float` then `ival + fval` is converted to `double`. Then the right hand expression is converted to `char` (to be assigned to `cval`).

Exercise 4.36

```
i *= static_cast<int>(d);
```

Exercise 4.37

(a)

```
pv = static_cast<void*>(const_cast<std::string*>(ps));
```

(b)

```
i = static_cast<int>(*pc);
```

(c)

```
pv = static_cast<void*>(&d);
```

(d)

```
pc = static_cast<char*>(pv);
```

Exercise 4.38

First the expression `j/i` is evaluated, both `i` and `j` are of type `int` so integral division is performed resulting in an `int`. Then the result is explicitly cast to `double` and assigned to the variable `slope`.