Exercise 2.1

These different signed integral types can have a different size resulting in different minimum and maximum value that can be held. An unsigned type can only represents values greater than or equal to zero. On the other hand signed integral types can hold negative values. The `float` and `double` types have different sizes. The `double` type is generally bigger and provides more precision than the `float` type.

Exercise 2.2

For the rate, principal and payment I would use a `double` as these numbers are generally not integers. Although when dealing with money it can be more appropriate to use rational numbers (not defined by the C++ standard).

Exercise 2.3

I think the output on my machine will be:

```
32
4294967264
32
-32
0
0
```

The second line is the only value that can change depending on your system. On my machine `unsigned` are 32 bits hence -32 is equal to $2^{32} - 32 = 4294967264$.

Exercise 2.4

```cpp
// type_conversions.cpp

#include <iostream>

int main()
{
    unsigned u = 10, u2 = 42;
    std::cout << u2 - u << std::endl;
    std::cout << u - u2 << std::endl;

    int i = 10, i2 = 42;
    std::cout << i2 - i << std::endl;
    std::cout << i - i2 << std::endl;
```

```
    std::cout << i - u << std::endl;
    std::cout << u - i << std::endl;

    return 0;
}
```

Our assumptions were correct.

Exercise 2.5

(a)

Literal	Type
`'a'`	`char`
`L'a'`	`wchar_t`
`"a"`	Array of constant `char`
`L"a"`	Array of constant `wchar_t`

Quotations marks are used for string literals and apostrophes are used for `char` or `wchar_t`. The prefix `L` is used when we want to use the type `wchar_t` instead of `char`.

(b)

Literal	Type
`10`	`int`
`10u`	`unsigned`
`10L`	`long`
`10uL`	`unsigned long`
`012`	`int`
`0xC`	`int`

The first literal have no suffix nor prefix and so is of type `int` because it fits inside this type. The next three literals use different suffixes and have different type. The last two literals use prefixes (`0` for octal and `0x` for hexadecimal), they both fit into an `int`.

(c)

Literal	Type
`3.14`	`double`
`3.14f`	`float`
`3.14L`	`long double`

No suffix is used for a `double`, the `f` suffix indicates a `float` and the `L` suffix is used for `long double`.

(d)

Literal	Type
`10`	`int`
`10u`	`unsigned`
`10.`	`double`
`10e-2`	`double`

The two first have no decimal point nor exponent and so are integers (`int` and `unsigned`). The next two literals are floatting-point with no suffixes and so are of type `double`.

Exercise 2.6

The first line use decimal literals and the second line use octal literals but the "9" is not a valid octal digit so the second line is not valid (a good compiler will warn you).

Exercise 2.7

(a) It's an array of constant `char`, the value of the `char` `\145` inside this string is 101 which is the numerical value of `'e'` on my system. The value of `\012` is 10 which is the numerical value of `'\n'` (newline) on my computer. (b) It's a `long double` of value 31.4. (c) `1024f` is not a valid literal. (d) It's a `long double` of value 3.14.

Exercise 2.8

```
// escape_sequences_1.cpp

#include <iostream>

int main()
{
    std::cout << "2M\n";
    return 0;
}
```

```
// escape_sequences_2.cpp
```

```cpp
#include <iostream>

int main()
{
    std::cout << "2\tM\n";
    return 0;
}
```

Exercise 2.9

(a) The definition is illegal, we can't define a variable inside a function call. To correct this we could write `int input_value; std::cin >> input_value`.

(b) The definition is illegal because initialization loosing data inside curly braces is forbiden. To correct it either remove the fractional part `.14` or the curly braces.

(c) The definition is illegal if `wage` is not already defined. We can correct it by writing `double salary = 9999.99, wage = 9999.99`.

(d) The last definition is correct but troncate, the value of `i` is 3.

Exercise 2.10

`global_str` is a `string` defined with no initializer so the variable use the default initializer giving the empty string value.

`global_int` is an `int` defined with no initializer and defined outside a function so it's initialized to 0.

`local_int` is an `int` defined inside a function, it has no initializer and so has undefined initial value.

`local_str` has no explicit initializer and so is default initialized. The initial value is the empty string.

Exercise 2.11

(a) There is an `extern` but also an initializer so this statement is a definition.

(b) There is no `extern` keyword so this is a definition.

(c) There is the `extern` keyword and no initializer so the statement is a declaration only.

Exercise 2.12

(a) Invalid because `double` is a keyword (for a built-in type).

(b) `_` is a valid name as identifiers can start with an underscore.

(c) Invalid name because `-` can't be used in identifiers.

(d) Invalid, identifiers can only start with a letter or an underscore, not a digit.

(e) Valid because `Double` (with capital `D`) is not a keyword.

Exercise 2.13

The value of `j` is the same as the value of `i` in the scope of the `main` function, so `j` has the value 100.

Exercise 2.14

Yes the following program is valid but the output might not be what we expected because in the printing statement, the variable `i` refers to the one defined in the first line, not the one in the `for` loop. This program prints `100 45` (45 is equal to $0 + 1 + \cdots + 9 = 10 \times (9 + 0)/2$).

Exercise 2.15

(a) Valid but truncate the value.

(b) Invalid because the initializer of a reference must be an object.

(c) Correct if `ival` is an object of type `int`.

(d) Incorrect, a reference must be initialized.

Exercise 2.16

(a) Valid, assign `3.14159` to `d`.

(b) Valid, assign the value of `i` (0) to `d`.

(c) Valid, assign the value of `d` (0) to `i` which is already of value 0.

(d) Valid, assign the value of `d` (0) to `i` which is already of value 0.

Exercise 2.17

This code prints `10 10` as the last assignment assigns `10` to `i`.

Exercise 2.18

```cpp
// pointers_1.cpp

#include <iostream>

int main()
{
    int x = 3, y = 5;
    int *p = &x, *q = p;
    std::cout << *p << std::endl;

    p = &y; // change the value of a pointer
    std::cout << *p << std::endl;     // now p points to y
    std::cout << (p == q) << std::endl; // 0, the value of p changed

    q = p;
    *p = 7; // change the value to which p points
    std::cout << y << std::endl; // y was modified using p
    std::cout << (p == q) << std::endl; // 1, the value of p did not changed

    return 0;
}
```

Exercise 2.19

References are just new names to refer to an object, they are not objects. In the contrary pointers are new objects which can hold values. Also references are always valid, there is no concept of `nullptr` for references.

Exercise 2.20

This program squares the value holded by `i`.

Exercise 2.21

(a) Illegal because `dp` is a pointer to `double`, it must be initialized with the address of a `double` but `i` is an `int`.

(b) Illegal, we can't assign an `int` to a pointer to an `int*`.

(c) Legal.

Exercise 2.22

We enter the first `if` block if `p` is a nonzero pointer. We enter the second `if` block if `p` points to a nonzero `int`.

Exercise 2.23

If `p` is equal to `nullptr` then it doesn't point to a valid object but if `p` is not equal to `nullptr` there is no way to know if it points to a valide object if we don't have more information.

Exercise 2.24

A `void` pointer can hold the address of any object but a point to `long` can only be initialized to the address of a `long` object (and `i` is not of this type).

Exercise 2.25

(a) `ip` is a pointer to an `int` and has no initial value (if defined inside a function). `i` is an `int` with undefined value. `r` is a reference to `i` which has no value so `r` has no value.

(b) `i` is an `int` with undefined value (if defined inside a function). `ip` is a pointer to `int` with value 0 (`nullptr`).

(c) `ip` is a pointer to `int` with no initial value. `ip2` is an `int` with no initial value, it's not a pointer!

Exercise 2.26

(a) Illegal, `const` definition must have an initializer.

(b) Legal.

(c) Legal.

(d) Illegal, we can't change the value of `sz`.

Exercise 2.27

(a) Illegal, only `const` reference can be initialized with a literal.

(b) Legal if `i2` is an `int`, illegal if `i2` is a `const int`.

(c) Legal, a reference can be initialized with a literal.

(d) Legal if `i2` is an `int` or a `const int`.

(e) Legal if `i2` is an `int` or a `const int`.

(f) Illegal, references are not objects and can not be `const`. Also a reference must always be initialized.

(g) Legal.

Exercise 2.28

(a) Illegal, `const` pointer must be initialized.

(b) Illegal, `const` pointer must be initialized.

(c) Illegal, `const int` definition must have an initilizer.

(d) Illegal, `const` pointer must be initialized.

(e) Legal.

Exercise 2.29

(a) Legal.

(b) Illegal, `p3` points to `const` but not `p1`.

(c) Illegal, `&ic` is an address of a `const int` but `p1` is not a pointer to `const`.

(d) Illegal, `p3` is a `const` pointer so we can't assign to it.

(e) Illegal, `p2` is a `const` pointer so we can't assign to it.

(f) Illegal, `ic` is `const` so we can't assign to it.

Exercise 2.30

Variable	top-level `const`	low-level `const`
`v2`	Yes	N/A
`v1`	No	N/A
`p1`	No	No
`r1`	N/A	No
`p2`	No	Yes
`p3`	Yes	Yes
`r2`	N/A	Yes

Exercise 2.31

`r1 = v2;` : legal, `v2` has no low-level `const`.

`p1 = p2;` : illegal, `p2` has low-level `const` but not `p1`.

`p2 = p1;` : legal, `p1` has no low-level `const`.

`p1 = p3;` : illegal, `p3` has low-level `const` but not `p1`.

`p2 = p3;` : legal, `p2` has low-level `const` so we can assign `p3` to it.

Exercise 2.32

This code is not legal because we can't initialize a pointer to `int` from an `int`, we must add the 'address' operator (`&`):

```
int null = 0; *p = &null;
```

Exercise 2.33

`a = 42;` : the value of `a` changes from `0` to `42`.

`b = 42;` : the value of `b` changes from `0` to `42`.

`c = 42;` : the value of `c` changes from `0` to `42`.

`d = 42;` : illegal, type error, `42` is not an `int*`.

`e = 42;` : illegal, type error, `42` is not an `int*`.

`g = 42;` : illegal, we can't assign to a `const reference`.

Exercise 2.34

```
// auto_1.cpp

#include <iostream>

int main()
{
    int i = 0, &r = i;
    auto a = r;
    const int ci = i, &cr = ci;
    auto b = ci;
    auto c = cr;
    auto d = &i;
    auto e = &ci;
    auto &g = ci;

    std::cout << "a: " << a << std::endl;
    a = 42;
```

```
    std::cout << "a: " << a << std::endl;

    std::cout << "b: " << b << std::endl;
    b = 42;
    std::cout << "b: " << b << std::endl;

    std::cout << "c: " << c << std::endl;
    c = 42;
    std::cout << "c: " << c << std::endl;

    // // will not compile due to incompatible types
    // std::cout << "d: " << d << std::endl;
    // d = 42;
    // std::cout << "d: " << d << std::endl;

    // // will not compile due to incompatible types
    // std::cout << "e: " << e << std::endl;
    // e = 42;
    // std::cout << "e: " << e << std::endl;

    // // will not compile due to incompatible types
    // std::cout << "g: " << g << std::endl;
    // g = 42;
    // std::cout << "g: " << g << std::endl;

    return 0;
}
```

Exercise 2.35

`const int i = 42;` : defines `i` to be a `const int`.

`auto j = i;` : `j` is of type `int`.

`const auto &k = i;` : `k` is a reference to `const int`.

`auto *p = &i;` : `p` is a pointer to `const int`.

`const auto j2 = i, &k2 = i` : `j2` is a `const int` and `k2` is a reference to `const int`.

```
// auto_2.cpp

#include <iostream>

int main()
```

```cpp
{
    const int i = 42;
    auto j = i;
    const auto &k = i;
    auto *p = &i;
    const auto j2 = i, &k2 = i;

    // i = 0; // must not compile because i is const

    j = 0; // legal because j is a non const int

    std::cout << k << std::endl; // 42 because k is a reference to i
    // k = 3; // must not compile because k is a const reference

    std::cout << *p << std::endl; // 42
    // *p = 5; // must not compile as p is a pointer to const int
    p = nullptr; // works because p has no top-level const

    // j2 = 0; // illegal because j2 is const

    std::cout << k2 << std::endl; // 42
    // k2 = 7; // must not compile because k2 is a const reference

    return 0;
}
```

Exercise 2.36

Variable	Type	Value when the code finishes
a	int	4
b	int	4
c	int	4
d	int&	4

Exercise 2.37

Variable	Type	Value when the code finishes
a	int	3
b	int	4
c	int	3
d	int&	3

Here the expression `a = b` inside the `decltype` is not evaluated so the value of `a` stays unchanged.

Exercise 2.38

`decltype` depends on the form of the given expression but not `auto`.

```
// auto_decltype.cpp

int main()
{
    int i = 0;
    int *pi = &i;

    // here i1 and i2 are of the same type
    decltype(i) i1 = 0;
    auto i2 = 0;

    decltype(*pi) ri = *pi;
    // ri is a reference to an int, we can check this by removing the
    // initializer and getting a compile time error
    auto j = *pi; // pi is an int, the type is different from ri

    return 0;
}
```

Exercise 2.39

```
// struct_semicolon_error.cpp

struct Foo { /* empty */ }  // Note: no semicolon

    int main()
    {
        return 0;
    }
```

Here is the error message I get from my compiler (GCC):

```
struct_semicolon_error.cpp:3:27: error: expected ‘;’ after struct definition
 struct Foo { /* empty */ }  // Note: no semicolon
                           ^
                           ;
```

The error message is pretty straightforward.

Exercise 2.40

```cpp
// own_sales_data.cpp

#include <string>

struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

int main()
{
    return 0;
}
```