1 Chapter 1

2 Chapter 2

2.1 Primitive Built-in Types

2.1.1 Arithmetic Types

Exercise 2.1 The different integer types vary in their minimum bit sizes and so their minimum value range:

- `short` – 16 bits gives a value range of -32768 – 32767.
- `int` – 16 bits gives a value range of -32768 – 32767.
- `long` – 32 bits gives a value range of -2147483648 – 2147483647.
- `long long` – 64 bits vies a value range of -9223372036854775808 – 9223372036854775807.

`unsigned` types do not include negative values, that is, values ≥ 0, for example if we consider `unsigned int` the value range is 0 – 65536. Of course a `signed` type takes negative values.

Lastly, both `float` and `double` represent decimal precision values. The difference is in the accuracy of precision, a `float` is precise to 6 significant digits and `double` is precise to 10 significant digits.

Exercise 2.2 They should all be `double` as they all involved decimal figures.

2.1.2 Type Conversions

Exercise 2.3 Assuming the machine is 32 bits for integers the outputs are (in order):

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

Exercise 2.4 See ex24.cpp.

2.1.3 Literals

Exercise 2.5

(a) In order: character literal; wide character literal; string literal; wide character string literal.

(b) In order: integer; unsigned integer; long integer; unsigned long integer; octal integer; hexadecimal integer.

(c) In order: double; float; long double.

(d) In order: integer; unsigned integer; double; double.

Exercise 2.6 In the first statement the definitions use integer literals so the values are what we expect, that is, `month = 9` and `day = 7`. In the second statement we try to define integers using octals however there is an error as 09 is not an octal and will throw an error.

Exercise 2.7

(a) This is a string literal which represents "Who goes with Fergus?"

(b) The value is 3.1 and is a long double.

(c) The value is 1024 and is a float.

(d) The value is 3.14 and is a long double.

Exercise 2.8 See ex28.cpp.

2.2 Variables

2.2.1 Variable Definitions

Exercise 2.9

(a) Cannot use variable before it has been declared. To correct:
```
int input_value;
std::cin >> input_value;
```

(b) Cannot use double in list initialisation since narrowing value may lose data. To correct:
```
double i = { 3.14 };
```

(c) Cannot use variable to initialise another variable before it has been initialised. To correct:
```
double wage = 9999.99;
double salary = wage;
```

(d) This will initialise `i` to be an `int` but it will truncate it's value from 3.14 to 3.

Exercise 2.10

- `global_str` will be an empty string.
- `global_int` will be 0.
- `local_int` will be undefined.
- `local_str` will be an empty string.

2.2.2 Variable Declarations and Definitions

Exercise 2.11

(a) This is a definition since the variable is initialised and so memory is allocated for the variable.

(b) This is a definition as the variable is defined locally and so memory is allocated for it.

(c) This is a declaration of a variable as it specifies the variable is external without initialising its value.

2.2.3 Identifiers

Exercise 2.12 `double` is a reserved C++ keyword and so is invalid, `catch-22` does not include valid characters in an identifier and so is invalid and lastly, `1_or_2` is invalid as it begins with a number.

2.2.4 Scope of a Name

Exercise 2.13 Since `i` is defined globally and locally, within the block scope its value is that derived from initialisation within the block. So `j` will have value 100.

Exercise 2.14 The program is legal since variables can be redefined in an inner scope (here this is the `for` loop). The program prints `100 45` since the value of `sum` is the sum of the numbers 0–9.

2.3 Compound Types

2.3.1 References

Exercise 2.15 The definition in (b) and (d) are invalid as reference types must be initialised to an object.

Exercise 2.16

(a) Valid. Assigns the value 3.314159 to `d`.

(b) Valid. Assigns 0.0 to `d`.

(c) Valid. Assigns 0 to `i`.

(d) Valid. Assigns 0 to `i`.

Exercise 2.17 This prints `10 10`.

2.3.2 Pointers

Exercise 2.18 See ex218.cpp.

Exercise 2.19 Pointers and references both "refer" to another object. However, references are not objects, that is they are not allocated space in memory and as such they cannot be rebound to another object after initialisation. This also means they *must* be initialised in their definition. A pointer is an object and can refer to several objects over its lifetime since it has memory allocated to it, the address which it stores can be redefined later on.

Exercise 2.20

- Initialise `i` to 42.
- Initialise the pointer `p1` to point to `i`.
- Assign to `i` the value `i * i`

Exercise 2.21

(a) This is invalid since `dp` is a `double` pointer and so cannot point to an `int`.

(b) Initialises an `int` pointer to 0, that is, the pointer is a null pointer.

(c) Initialises an `int` pointer that points to the `int` object `i`.

Exercise 2.22 The first `if` checks if `p` is a null pointer or not. If it is not a null pointer it will execute its corresponding block. The second `if` checks if `p` points to an `int` object of value 0 or not. If it points to an object with value non-zero the corresponding block will execute.

Exercise 2.23 No. If a pointer is uninitialised then it displays undefined behaviour. We do not know if the memory address it stores points to something valid or not.

Exercise 2.24 We have that `void*` pointers can point to objects of any type, since their purpose is to point to objects which we do now know their type a priori. However, a pointer of a specified type, for example `long`, can *only* point to objects of the same type. Hence, initialising a `long` pointer to an `int` object will raise an error.

2.3.3 Understanding Compound Type Declarations

Exercise 2.25

(a) `ip` is an `int` pointer with no value; `i` is an `int` with no value; `r` is an `int` reference with no value.

(b) `i` is an `int` with no value; `ip` is an `int` null pointer.

(c) `ip` is an `int` pointer with no value; `ip2` is an `int` with no value.

2.3.4 `const` Qualifier

Exercise 2.26 Assuming the definitions are in order of presentation then (b) and (c) are legal and (a) and (d) are illegal. In (a) the issue is that we are defining a `const` without initialising and in (d) we are trying to iterate a `const`.

2.3.5 References to `const`

2.3.6 Pointers to `const`

Exercise 2.27

(a) This is not legal since references cannot be bound to literals and only to objects.

(b) This is legal as `p2` is a low-level `const` so as long as `i2` is an `int` this is fine.

(c) This is legal as we are able to assign a literal to a reference to `const`.

(d) This is legal as pointer to `const` *think* they point to a `const`. In the event that `i2` is not a `const` all this means is that we cannot assign to `i2` through `p3`.

(e) This is legal however `i2` cannot be assigned to through `p1`.

(f) This is illegal as `r2` is not initialised. The low-level `const` here is redundant as references by their nature cannot be rebound after initialisation.

(g) This is legal, however `i` must be assigned to directly if not already.

Exercise 2.28 There are errors in (a)–(e). In each of them there is either a `const` pointer or `const` variable which is not initialised in its definition. Part (f) is fine and defines a pointer to `const int`.

Exercise 2.29

(a) Legal as assignment just copies the value, so you can assign a `const` variable to a non`const` variable.

(b) Illegal as `p3` is a low-level `const` and `p1` is not.

(c) Illegal as you cannot use a non`const` pointer to point to a `const` variable.

(d) Legal as `ic` is a `const` and `p3` is a pointer to `const`.

(e) Illegal for the same reason as in part (b).

(f) Legal as the copying a value does not take into consideration whether the objects are both `const` or not.

2.3.7 Top-Level `const`

Exercise 2.30 In order of presentation in the text:

- low-level
- neither
- `p1` is neither; `r1` is top-level
- `p2` is low-level; `p3` is both; and `r2` is both.

Exercise 2.31 In order of presentation in the text:

- This is legal as `r1` is only top-level so the referenced object is not a `const` and hence can be assigned to.
- This is illegal as `p2` is low-level and so can only be assigned to other low-level objects. Otherwise we could assign to a (potentially) `const` object and create an error.
- This is legal. Even though `p2` is low-level and `p1` is neither, when copying in assignment, the compiler will convert `p1` temporarily to low-level before continuing with assignment.
- This is illegal as in the second case.
- This is legal as both pointers are low-level.

2.3.8 `constexpr` and Constant Expressions

Exercise 2.32 The code is illegal as the pointer `p` is initialised with type `int` instead of `int*`. Depending on whether the code is intended to initialise `p` as a `nullptr` or to point at `null` the following codes will gives the desired result, respectively,

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

or,

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

Exercise 2.33

- `a`, `b` and `c` are of type `int` and so will be set to the value of 42.
- `d`, `e` and `g` are all `int` pointers (some `const`) and so we cannot assign them to literals.

Exercise 2.34

Exercise 2.35 The code given is:

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

In order we have; `i` is of type `const int`; `j` is of type `int`; `k` is of type `const int`; `p` is of type pointer to `const int`; `j2` is of type `const int`; and `k2` is of type reference to `const int`.