Formatted output (1)

Let us have

```
struct invoice { const char* pPassenger, * pDestination; const int fare; };
invoice InvoiceList[] = {
 { "John Smith", "Quito", 2000 },
 { "Richard Carpenter", "Guatemala City", 1700 } };
```

If we print it out using code snippet:

```
auto print = [](const invoice& inv) { cout << inv.pPassenger << ' '<< inv.pDestination << ' '
                                          << inv.fare << endl;  };
for_each(InvoiceList.begin(), InvoiceList.end(), print);
```

we get:

```
John Smith Quito 2000
Richard Carpenter Guatemala City 1700
```

To get a correct table we need better formatting:

```
auto print = [](const invoice& inv) { cout << setw(17) << left << inv.pPassenger << ' '
                                          << setw(14) << inv.pDestination << ' ' <<
                                         inv.fare << endl;  };
```

```
John Smith       Quito         2000
Richard Carpenter Guatemala City 1700
```

The code of the last lambda is rather inconsiderate and clumsy.

Formatted output (2)

We may also use *printf*:

```
for (int i = 0; i < 2; i++) {
    printf("%-17s %-14s %d\n", InvoiceList[i].pPassenger, InvoiceList[i].pDestination,
            InvoiceList[i].fare);
}
```

This code is more comprehensive, but it is C, not C++.

Formatting in C++ v. 20 is a combination of *iostream* and *printf*:

```
#include <format> // see https://en.cppreference.com/w/cpp/utility/format
auto print = [](const invoice& inv) { cout << format("{:17} {:14} {}\n", inv.pPassenger,
                                                    inv.pDestination, inv.fare); };
for_each(InvoiceList.begin(), InvoiceList.end(), print);
```

As *printf*, *format* has formatting string followed by the sequence of arguments. The formatting rules for each argument are enclosed into braces. If there are no any specific requirements, the braces are empty and *format* deduces the type itself:

```
cout << format("{}\n", arg); // arg may be int, double, C-style string, C++ string object, etc.
```

Do not put spaces between braces, you will get an error message!

The order of formatting rules in formatting string and arguments in list may not match:

```
cout << format("{1:17} {0:14} {2}\n", inv.pPassenger, inv.pDestination, inv.fare);
```

Now the printout starts with the destination, the passenger's name will follow. Number right after *{* specifies the order.

Formatted output (3)

Formatting rules start with colon.

A positive integer after colon specifies the field width:

```
cout << format("{:6}\n", 10);  // prints ␣␣␣␣10 (by default right alignment)
cout << format("{:<6}\n", 10); // prints 10␣␣␣␣
cout << format("{:^6}\n", 10); // prints ␣␣10␣␣
cout << format("{:6}\n", "ab");  // prints ab␣␣␣␣ ((by default left alignment)
cout << format("{:6}\n", "abcdefghijkl");  // prints the complete string abcdefghijkl
```

Space is the default filling character. But you can use another:

```
cout << format("{:_<6}\n", 11); // prints 11____
cout << format("{:0^6}\n", 11); // prints 001100
cout << format("{:_>6}\n", 11); // prints ____11, but {:_6} is an error
```

The precision is used for floating-point values and strings:

```
cout << format("{}\n", 0.123456789); // prints 0.123456789
cout << format("{:.5}\n", 0.123456789); // prints 0.12346 (rounds up)
cout << format("{:^11.5}\n", 0.123456789); // prints ␣␣0.12346␣␣
cout << format("{:.5}\n", "abcdefghijk"); // prints abcde
cout << format("{:>10.2}\n", "abcdefghijk"); // prints ␣␣␣␣␣␣␣␣ab
```

Formatted output (4)

Type specifiers for integral types:

```
cout << format("{:d} {:s}\n", true, true); // prints 1 true
cout << format("{:d} {:x} {:X} {:c}\n", 'Z', 'Z', 'Z', 'Z'); // prints 90 5a 5A Z
cout << format("{:#X} {:b} {:#B}", 251, 251, 251); // prints 0XFB 11111011 0B11111011
```

Type specifiers for floating-point types:

```
cout << format("{} {:f} {:g} {:E}\n", -5.0, -5.0, -5.0, -5.0);
   // prints -5 -5.000000 -5 -5.000000E+00
cout << format("{} {:f} {:g} {:e}\n", -5.5, -5.5, -5.5, -5.5);
   // prints -5.5 -5.500000 -5.5 -5.500000e+00
cout << format("{} {:f} {:g} {:e}\n", numbers::pi, numbers::pi, numbers::pi * 1e-3,
                    numbers::pi * 1e6);
   // prints 3.141592653589793 3.1241593 0.00314159 3.14159e+06
```

It is possible to store the result into a buffer:

```
char buf1[256];
auto ret1 = format_to_n(buf1, size(buf1) - 1, "{}", 100);
*(ret1.out) = 0;  // we must ourself add the terminating zero
```

or

```
array<char, 256> buf2{};  // fills with zeroes
auto ret2 = format_to_n(buf2.begin(), size(buf2) - 1, "{}", 100);
```

Formatted output (5)

Of course, it is better to calculate the needed size of buffer first:

```
auto n = formatted_size("{}", 100);  // returns 3, so the actual size is n + 1
```

To work without predefined buffer:

```
string s;
format_to(back_inserter(s), "{}", 100);  // see "Insert iterators" from chapter "Algorithms"
format_to(back_inserter(s), "{}", ' ');
format_to(back_inserter(s), "{}", 200);
cout << s << endl; // prints 100 200, thus we may replace the stringstream
```

Predefined format strings must be compile-time values:

```
const char *pfr = "{}\n";
cout << format(pfr, 100); // error
constexpr const char *pfr = "{}\n";    // correct
```

However, sometimes we may need to create the formatting string at runtime. In that case we can use *vformat* and *vformat_to* together with function *make_format_args*:

```
const char *pfr = "{} {:f} {:g} {:e}\n";
cout << vformat(pfr, make_format_args(3.14159, 3.14159, 3.14159* 1e-3,
                        3.14159* 1e6);
```

Comparisons (1)

Let us have a simple class:

```
class Id {
    long long int id = 0;
public:
    Id() { }
    Id(long long int l) : id(l) {};
    long long int get() const { return id; }
    void set(long long int l) { id = l; }
};
```

To check are two objects from class *Id* equal or not we need operator function:

```
bool operator==(const Id &i) const { return id == i.get(); }
```

or

```
bool operator!=(const Id &i) const { return id != i.get(); }
```

Now

```
Id nr1(77LL), nr2(77LL), nr3(88LL);
cout << boolalpha << (nr1 == nr2) << ' ' <<  (nr1 == nr3) << ' ' << (nr1 != nr3) << endl;
                                                                    // prints true false true
```

For the complete set of comparisons we need four more operator functions: *operator*>, *operator*<, *operator*>= and *operator* <=.

Comparisons (2)

If we want to compare the value in object with an integer we need another operator function:

```
bool operator==(const long long int &l) const { return id == l; }
```

Now:

```
cout << boolalpha <<  (nr1 == 99LL) << endl; // prints false
```

but

```
cout << boolalpha <<  (99LL == nr1) << endl; // error
```

Consequently we need one more operator function:

```
friend bool operator==(const long long int& l, const Id& i) { return l == i.id;  }
```

Totally, we may need for class *Id* 18 operator functions.

However, in C++ v. 20 if *operator==* is implemented, *operator!=* is not needed, the inequality can be checked without it. Also, it is enough to impelement one operator== for comparison of objects of different types. So

```
class Id {
……….
bool operator==(const Id &i) const { return id == i.get(); }
bool operator==(const long long int &l) const { return id == l; }
};
Id nr1(77LL), nr2(77LL), nr3(88LL);
cout << format("{:s} {:s}\n", nr1 == nr2, nr1 != nr3);  // prints true true
cout << format(("{:s} {:s}\n", , nr1 == 99LL, 99LL != nr1); // prints false true
```

Comparisons (3)

But if we have only *operator>*,

```
cout << format("{:s} {:s}\n", nr1 > nr2, nr1 < nr3); // error, operator< is also needed
```

The solution is to use new C++ v. 20 operator<=> (three-way comparison operator, called also as spaceship).

```
#include <compare> // https://en.cppreference.com/w/cpp/utility/compare/compare_three_way
int i1 = 10, i2 = 10, i3 = 20;
auto ret1 = i1 <=> i2;  // the return value cannot be bool
cout << typeid(ret1).name() << endl;  // prints struct std::strong_ordering
auto ret2 = i2 <=> i1;
auto ret3 = i1 <=> i3;
auto ret4 = i3 <=> i1;
auto iprint = [](auto ret)->string {
                        if (ret == strong_ordering::less) return "less";
                        else if (ret == strong_ordering::greater) return "greater";
                        else return "equal";  }; // strong_ordering::equal
cout << format("{} {} {} {}\n", iprint(ret1), iprint(ret2), iprint(ret3), iprint(ret4));
   // prints equal equal less greater
```

strong_ordering cannot be used in *switch* statements.

For floating-point values the result type is *partial_ordering,* it includes also member *unordered* (for situation in which one of the operands is not a number).

Comparisons (4)

All the primitive types as well as all the C++ standard classes for which the relational operations are defined support also the three-way comparison.

The main strength of three-way comparison is that we may in our own classes replace *operator<*, *operator<=*, *operator>* and *operator>=* functions with one function *operator<=>*:

```
auto operator<=>(const Id& i) const { return id <=> i.get(); }
```

Now:

```
Id nr1(77LL), nr2(77LL), nr3(88LL);
auto ret1 = nr1 <=> nr3;
auto ret2 = nr3 <=> nr1;
auto ret3 = nr1 <=> nr2;
cout << format("{} {} {}\n", ret1._Value, ret2._Value, ret3._Value); // prints -1 1 0
```

If the *_Value* is negative, the first operand is less, if positive, the first operand is greater and if 0, they are equal (as in standard function *strcmp*).

Three-way comparison result may be checked by standard functions *is_eq*, *is_neq*, *is_lt*, *is_lteq*, *is_gt*, *is_gteq*:

```
cout << format("{} {} \n", is_eq(nr1<=>nr3), is_gt(nr3<=>nr1)); // prints false true
```

Comparisons (5)

C++ v. 20 compiler is able to generate default *operator*== and *operator* <=> functions:

```
class Name {
    string FirstName = "", LastName = "";
public:
    Name(string s1, string s2) : FirstName(s1), LastName(s2) {}
    bool operator==(const Name &n) const = default;
    auto operator<=>(const Name &n) const = default;
};

Name p1("Clyde", "Barrow"), p2("Clyde", "Barrow"), p3("Bonnie", "Parker");
cout << format("{:s} {:s} {:s}\n", p1 == p2, p1 == p3, p1 != p3);
            // prints true false true
cout << format("{} {} {}\n", (p1 <=> p2)._Value, (p1 <=> p3)._Value, (p3 <=> p1)._Value);
            // prints 0 1 -1
```

As in the default copy constructor, default comparison functions perform the operations attribute by attribute. If some of the attributes are pointers, it does not work and we need to write our comparison ourselves.