15 – Exercises on Generic Functions

Université Jean Monnet – Saint-Étienne 2026/2027

Quick reference

```
// A generic function works for any type T
template<typename T>
T my_max(T a, T b) {
    return a > b ? a : b;
}

int   x = my_max(10, 20);     // T is deduced as int
float y = my_max(1.5f, 0.8f); // T is deduced as float

// Explicit instantiation (when deduction is ambiguous)
double z = my_max<double>(3, 4.5);

// Multiple type parameters
template<typename T, typename U>
void print_pair(T a, U b) {
    std::cout << a << " " << b << std::endl;
}
```

Key rules:

- `template<typename T>` declares `T` as a placeholder for any type.
- The compiler **deduces** `T` from the arguments at the call site — you usually do not need to write it explicitly.
- The template body must compile for the type used; if you call `my_max` with a type that does not support `>`, you get a compile error.
- Templates are not about avoiding type-checking — they let you write one function that works correctly for many types, while still catching type errors at compile time.

Part A — Reading and tracing templates

Problem 1 — What does this print?

Basic template trace

```
#include <iostream>

template<typename T>
T clamp(T value, T lo, T hi) {
    if (value < lo) return lo;
    if (value > hi) return hi;
    return value;
}

int main() {
    std::cout << clamp(80, 0, 100)  << std::endl; // line A
    std::cout << clamp(150, 0, 100) << std::endl; // line B
    std::cout << clamp(-5, 0, 100)  << std::endl; // line C
    std::cout << clamp(0.7f, 0.f, 1.f) << std::endl; // line D
    std::cout << clamp(1.8f, 0.f, 1.f) << std::endl; // line E
    return 0;
}
```

Line A (type deduced as):

Line B (type deduced as):

Line C (type deduced as):

Line D (type deduced as):

Line E (type deduced as):

Problem 2 — Which call compiles?

The template below is given:

```
template<typename T>
T add(T a, T b) { return a + b; }
```

For each call, write **compiles** or **error** and explain briefly.

Call	Compiles?	Why
`add(10, 20)`		
`add(0.5f, 1.5f)`		
`add(10, 0.5f)`		
`add<float>(10, 0.5f)`		
`add(std::string("a"), std::string("b"))`		

Part B — Writing generic functions

Problem 1 — Generic utilities

Write each function as a function template, then test it in `main` with at least two different types.

(a) `swap(T& a, T& b)` — swaps two values in place. Return type: `void`.

(b) `min3(T a, T b, T c)` — returns the smallest of three values.

(c) `lerp(T a, T b, float t)` — returns the linear interpolation $a + (b - a) \times t$. Use two type parameters: one for `a` and `b`, one fixed as `float` for `t`.

Expected output for the following `main`:

```
int main() {
    int x = 10, y = 40;
    swap(x, y);
    std::cout << x << " " << y << std::endl; // (a) int

    float p = 1.5f, q = 0.5f, r = 0.8f;
    std::cout << min3(p, q, r) << std::endl; // (b) float

    std::cout << lerp(0, 100, 0.25f) << std::endl; // (c) int
    std::cout << lerp(0.f, 1.0f, 0.75f) << std::endl; // (c) float
    return 0;
}
```

```
40 10
0.5
25
0.75
```

Problem 2 — Generic functions on vectors

Write each function as a template that works on `std::vector<T>` for any `T` that supports the required operators.

(a) `T vector_min(const std::vector<T>& v)` — returns the smallest element. Assume `v` is non-empty.

(b) `T vector_max(const std::vector<T>& v)` — returns the largest element.

(c) `void vector_clamp(std::vector<T>& v, T lo, T hi)` — clamps every element in place using the `clamp` template from exercise 1.1.

Expected output for the following `main`:

```
int main() {
    std::vector<int> volumes = {80, 120, -10, 45, 200};
    std::vector<float> gains = {0.5f, 1.8f, -0.3f, 1.0f};

    std::cout << vector_min(volumes) << std::endl; // a
    std::cout << vector_max(volumes) << std::endl; // a

    std::cout << vector_min(gains) << std::endl; // a float
    std::cout << vector_max(gains) << std::endl; // a float

    vector_clamp(volumes, 0, 100);
    for (int i = 0; i < volumes.size(); i = i + 1)
        std::cout << volumes[i] << " ";
    std::cout << std::endl;                      // c

    return 0;
}
```

```
-10
200
-0.3
1.8
0 100 0 45 100
```

Problem 3 — Generic functions on custom classes

Two classes are provided in this exercise. Do not modify them. Write a **single** generic function:

```
template<typename T>
size_t longest(const std::vector<T>& items);
```

that returns the index of the element with the longest audio.

Here are the classes:

```
class Mono {
    std::vector<float> m_audio;
public:
    Mono(const std::vector<float>& samples)
    : m_audio {samples.begin(), samples.end()} {}
    int duration() const { return m_audio.size(); }
};

class Stereo {
    std::vector<std::vector<float>> m_audio;
public:
    Stereo(const std::vector<float>& left_samples,const std::vector<float>&
        right_samples) {
        if(left_samples.size()==right_samples.size()) {
            m_audio.push_back(left_samples);
            m_audio.push_back(right_samples);
        }
    }
    int duration() const {
        if(m_audio.size() == 2)
            return m_audio[0].size();
        return 0;
    }
};
```

Then write a `main` that calls `longest` on each of:

```
std::vector<Mono> monos = {Mono{{0.1f, -0.1f}}, Mono{{0.2f, -0.15f, 0.25f, -.3}}, Mono
    {{0.01f, -0.081f, 0.01f, -0.081f, 0.01f, -0.081f, 0.01f, -0.081f}}};
std::vector<Stereo> stereos = {Stereo{{0.1f, -0.1f}, {0.1f, -0.1f}}, Stereo{{0.01f,
    -0.081f, 0.01f, -0.081f, 0.01f, -0.081f, 0.01f, -0.081f}, {0.01f, -0.081f, 0.01f,
    -0.081f, 0.01f, -0.081f, 0.01f, -0.081f}}, Stereo{{0.2f, -0.15f, 0.25f, -.3},{0.2f
    , -0.15f, 0.25f, -.3f}}};
```

and prints the index of the longest audio in each `vector`.

Expected output:

```
2
1
```

Part C — Overload or template?

Problem 1 — Decide and justify

For each scenario below, decide whether you would write a **function template**, a set of **overloads**, or **either would be fine**. Write your answer and a one-sentence justification.

Scenario	Choice	Justification
A function `abs_val` that returns the absolute value of its argument. Works identically for `int`, `float`, `double`.		
A function `describe` that prints `"note : <pitch>"` for a `Note` and `"pixel: <value>"` for a `Pixel` — different output for each type.		
A function `scale` that multiplies an `int` by an `int`, or a `float` by a `float`. The logic is identical in both cases.		

Problem 2 — Write both, then compare

Write `print_range` in two ways:

- As two overloads: one for `std::vector<int>`, one for `std::vector<float>`.
- As a single function template.

The function prints every element of the vector separated by spaces, followed by a newline.

Expected output for the following calls:

```
std::vector<int> notes = {60, 62, 64, 65};
std::vector<float> gains = {0.5f, 1.0f, 0.8f};
print_range(notes);
print_range(gains);
```

```
60 62 64 65
0.5 1 0.8
```

After writing both versions, answer:

How many lines of code differ between the two overloads?

What would you need to add if you later wanted to support `std::vector<std::string>` *— with overloads? With a template?*