Templates

- Adding type parameters to a function or class.
- Compiler generates code for each type needed.

```
//Inconvenient way
int max(int a, int b){
  return a > b ? a : b;
}
double max(double a, double b){
  return a > b ? a : b;
}
```

Examples

```
std::vector<int> vi;
std::vector<double> vd;
```

- Type arguments between < >.

Syntax

```
template <typename T>
T max(T a, T b) {
  return a > b ? a : b;
}
```

Deduction

- Compiler deduces types from the arguments.

```
template <typename T>
T max(T a, T b) {
  return a > b ? a : b;
}
max(1, 2); // int
max(1.0, 2.0); // double
max(1, 2.0); // error: cannot deduce type
max<double>(1, 2.0); // ok
```

Many arguments

- Compiler deduces types from the arguments.

```
template <typename T, typename U>
bool larger_than(T a, U b) {
  return a > b;
}
larger_than(1, 2);
larger_than(1.0, 2.0);
larger_than(1, 2.0); // ok
larger_than<int, double>(1, 2.0); // ok
```

Specialization

- Specialize a template for a specific type.

```
template <typename T, typename U>
bool larger_than(T a, U b) {
  return a > b;
}
template <>
bool larger_than(string a, string b) {
  return a.size() > b.size();
}
larger_than("hellooo", "world");
```

Non type parameters

- Constant expressions can be a template parameter.

```
template <typename T, size_t N>
auto create_array(T value){
  std::array<T,N> res;
  for(auto &v: res)
    v = value;
  return res;
}
auto arr = create_array<int,5>(42);
// {42,42,42,42,42}
```

Template aliases

- Solution to the `DynamicArray` problem.

```
template <typename T>
using DynamicArray = std::vector<T>;
DynamicArray<int> vi; // std::vector<int>
```

Variadic templates

- A template with a variable number of arguments.

```
template<typename... Args>
void printAll(Args... args) {
  ((std::cout << args << '\n'), ...);
}
print(1,"hello", 2.0, 3.0f,
      std::chrono::system_clock::now());
```

- `((pattern(args), ...)` is a fold expression.
- Expands to `pattern(args1), pattern(args2),` etc.

Variadic templates

- A template with a variable number of arguments.

```
f(args...);       // expands to f(E1, E2, E3)
f(&args...);      // expands to f(&E1, &E2, &E3)
f(n, ++args...); // expands to f(n, ++E1, ++E2, ++E3);
f(++args..., n); // expands to f(++E1, ++E2, ++E3, n);
```

- `(something)args...` is a parameter pack.