# Templates

Templates allow you to write generic code that works with multiple types. This provides a very powerful tool for the programmer for *code reuse*, flexibility, and efficiency by generating specific code for different types based on a template definition. Templates are used for creating generic algorithms, data structures, and other components that provide code modularity and flexibility.
We have already worked with templates: `std::vector` , `std::map` , `std::array` , `std::list` and many more.
Let's see an example.

In [ ]:
```
int max(int x, int y){
    return (x > y)? x : y;
}
```

The following code works as expected:

In [ ]:
```
int ia=1, ib=2;
int ic = max(ia, ib);
```

The following code has a so-called **cast**:

In [ ]:
```
double da = 1.1, db = 2.2;
double dc = max(da, db);
```

A cast is an operation which can be implicit (done by the compiler), or explicit (by the programmer). This operation changes the type of a variable or object. Let's see in the next example:

In [ ]:
```
double dd = max(ib, db);
```

A more general solution:

In [ ]:
```
template<typename T>
const T& max(const T& x, const T& y) {
    return (x>y) ? x : y ;
}
```

In [ ]:
```
double da = 1.1, db = 2.2;
double dc = max(da, db);
```

How did the function get compiled with `max(da,db)` ?

In [ ]:
```
template<typename T>
const T& max(const T& x, const T& y) {
    return (x>y) ? x : y ;
}
```

↓

In [ ]:
```
const double& max(const double& x, const double& y) {
    return (x>y) ? x : y ;
}
```

Case with integer:

In [ ]:
```
int ia=1, ib=2;
int ic = max(ia, ib);
```

How did `max(ia, ib)` get compiled?

In [ ]:
```
const int& max(const int& x, const int& y) {
    return (x>y) ? x : y ;
}
```

What about the following case? How can we fix it?

In [ ]:
```
double dd = max(ib, db);
```

Is there a more general solution?

In [ ]:
```
template<typename T, typename S>
const T& max(const T& x, const S& y) {
    return (x>y) ? x : y ;
}
```

# Class templates

Exactly the same idea, your compiler uses the types specified to produce a class with a type. For every type of a templated class we generate a different class definition at compile time.

In [ ]:
```
class Rational {
  private:
    int nom, den;
  public:
    Rational(int nominator, int denominator=1):nom(nominator), den(denominator){}
    int getDenominator(){ return den; }
    int getNominator(){ return nom; }
};
```

In [ ]:
```
Rational r1(2,5);
```

excluded types: `long long` , `unsigned int` , `unsigned short` , ...

In [ ]:
```
template <class U>
class Rational {
  private:
    U nom, den;
  public:
    Rational(U nominator, U denominator=1):nom(nominator), den(denominator){}
    U getDenominator(){ return den; }
    U getNominator(){ return nom; }
};
```

In [ ]:
```
Rational<int> r1(2,5);
Rational<short int> r2(3,51);
```

Template with non-type arguments

In [ ]:
```
#include <iostream>
template <int N>
class A {
    public:
    A(){
        std::cout << "Array of " << N << " elements" << std::endl;
    }
    private:
      int val[N];
};
```

In [ ]:
```
A<5> fivearray;
```

Type and non-type arguments combined.

In [ ]:
```
#include <iostream>
template<typename T, unsigned int N>
T power(T x){
    T ret = x;
    for (unsigned int i=1; i < N; ++i) ret *= x;
    return ret;
}
```

In [ ]:
```
power<double, 3>(4); // 4 ^ 3
```

We can use several template parameters

In [ ]:
```
template <typename S, typename T, unsigned int N>
class A {
  private:
    T v1;
    S v2[N];
    //...
};
```

## Lambda Templates

C++20

This has the typename keyword after the capture:

In [ ]:
```
// Note this wont run in the notebook
auto lTem = []<typename T>(T value) {
  std::cout << "Value: " << value << std::endl;
};

lTem(6.0);
```

But also available with `auto` C++14

More readable and more recommendable ;-)

In [ ]:
```
#include <iostream>

auto lTem = [](auto value) {
  std::cout << "Value: " << value << std::endl;
};

lTem(6.0);
lTem("Hello");
```

Can we nest templates?

Yes!

In [ ]:
```
template<typename T>
class Stack {
  private:
    std::vector<T> elems; //nesting
  public:
    //...
};
```

Exercise: make the following code a template.

In [1]:
```
#include <iostream>

class Particle {
private:
    double x_, y_;

public:
    Particle(double x, double y);
    Particle() : x_(0), y_(0) {}
    virtual ~Particle(){}

    double getX() const {
        return x_;
    }

    void setX(const double x){
        x_ = x;
    }

    void print(){
       std::cout << "particle(" << x_ << ", " << y_ << ");" << std::endl;
    }
};
```

In [2]:
```
Particle::Particle(double x, double y): x_(x) , y_(y) {
    //code
}
```

In [ ]: