ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

TEMPLATES IN C++

LANGUAGE C

```
#define max(x,y) ((x) > (y) ? (x) : (y))

#define min(x,y) ((x) < (y) ? (x) : (y))
```

LANGUAGE C++

```cpp
inline int max(int x, int y)          { return x > y ? x : y; }
inline long max(long x, long y)       { return x > y ? x : y; }
inline float max(float x, float y)    { return x > y ? x : y; }
inline double max(double x, double y) { return x > y ? x : y; }

// ......???????
```

FUNCTION TEMPLATES

```
template <parameter-list> function-declaration
```

FUNCTION TEMPLATES

```
template <parameter-list> function-declaration

template <typename T>
inline T const &max(T const &x, T const &y) {
    return x > y ? x : y;
}

template <class T>
inline T const &min(T const &x, T const &y) {
    return x < y ? x : y;
}
```

FUNCTION TEMPLATES

```
template <parameter-list> function-declaration

template <typename T>
inline T const &max(T const &x, T const &y) {
    return x > y ? x : y;
}

template <class T>
inline T const &min(T const &x, T const &y) {
    return x < y ? x : y;
}
```

typename and class
are equivalent.

```cpp
template <typename T>
inline T const &min(T const &x, T const &y) {
    return x < y ? x : y;
}
```

A compiler is looking for ***min*** function.
Finds this template and infers type ***T*** by type of function arguments.

```cpp
int a = 10;
int b = min(a, 7);
```

```cpp
template <typename T>
inline T const &min(T const &x, T const &y) {
    return x < y ? x : y;
}
```

A compiler is looking for ***min*** function. Finds this template and infers type ***T*** by type of function arguments.

```cpp
int a = 10;
int b = min(a, 7);
```

Template instantiation

```cpp
inline int const &min(int const &x, int const &y) {
    return x < y ? x : y;
}
```

```
template <typename T>
inline T const &min(T const &x, T const &y) {
    return x < y ? x : y;
}
```

A compiler is looking for ***min*** function.
Finds this template and infers type ***T*** by type of function arguments.

Template instantiation

```
int a = 10;
int b = min(a, 7);
```

Instance Function Substitution

```
inline int const &min(int const &x, int const &y) {
    return x < y ? x : y;
}
```

```
min(1, 1.2);                  // Error: different types
min(double(1), 1.2);          // OK. typecasting
min<double>(1, 1.2);          // OK. explicit type T
```

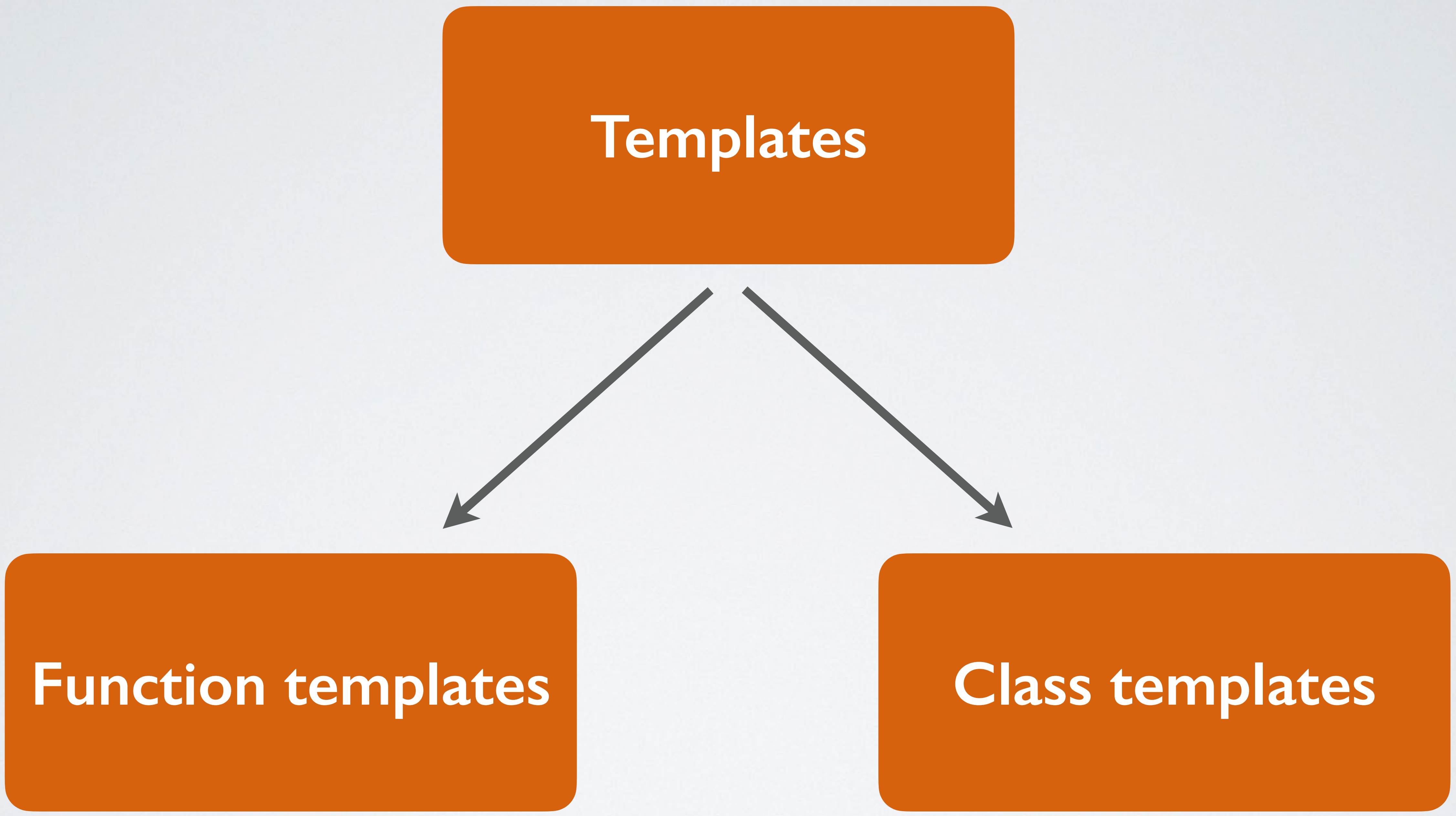
MULTIPLE PARAMETERS

```
template <typename T1, typename T2>
inline T1 const &min(T1 const &x, T2 const &y) {
    return x < y ? x : y;
}
// ....

min(1, 1.2);              // OK. The return type is determined
                         //     by the type of x.

template <typename RT, typename T1, typename T2>
inline RT const &max(T1 const &x, T2 const &y) {
    return x > y ? x : y;
}
// ....
max<double>(4, 4.2);  // Type inference is not possible for RT
```

FUNCTION TEMPLATE OVERLOADING

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

template <typename T>
inline T const &max(T const &x, T const &y) {
    return x > y ? x : y;
}

template <typename T>
inline T const &max(T const &x, T const &y, T const &z) {
    return max(max(x,y), z);
}

int main() {
    max(7, 42, 68);      // use template with 3 arguments
    max(7.0, 42.0);      // max<double> (template argument deduction)
    max('a', 'b');       // max<char> (template argument deduction)
    max(7, 42);          // usual function
    max<>(7, 42);        // max<int> (template argument deduction)
    max<double>(7, 42); // max<double>
    max('a', 42.7);      // usual function with 2 int arguments
}
```

```cpp
// From <algorithm>

template<class InpIt, class OutIt>
OutIt copy(InpIt first, InpIt last, OutIt result)
{
    while (first != last) {
      *result = *first;
      ++result; ++first;
    }

    return result;
}
```


Templates
Function templates
Class templates

CLASS TEMPLATES

```
template <typename T>
class Stack {
    std::vector<T> elems;
public:
    void push(T const &);
    void pop();
    T top() const;

    bool empty() const {
        return elems.empty()
    }
};

template <typename T>
void Stack<T>::push(T const &elem) {
    elems.push_back(elem);
}

template <typename T>
void Stack<T>::pop() {
    if (elems.empty())
        throw std::out_of_range("empty stack");

    elems.pop_back();
}

template <typename T>
T Stack<T>::top() const {
    if (elems.empty())
        throw std::out_of_range("empty stack");

    elems.back();
}
```

```cpp
int main() {
    try {
        Stack<int> intStack;
        Stack<std::string> stringStack;

        intStack.push(7);
        stringStack.push("hello");

        stringStack.pop();
        stringStack.pop();
    }

    catch (std::exception const &ex) {
        std::cerr << "Exception " <<
        ex.what() << std::endl;
        return -1;
    }
}
```

FUNCTION TEMPLATE SPECIALIZATION

```
template<typename T>
inline void exchange(T *a, T *b)
{
    T tmp(*a);
    *a = *b;
    *b = tmp;
}

// ....
void swap_arrays(Array<int> *a1, Array<int> *a2) {
    exchange(a1, a2);
}
```

Как заставить **exchange** использовать **Array::exchange_with**?

```
template<typename T>
class Array {
    T *data;

public:
    // ...
    void exchange_with(Array<T> *b) {
        T *tmp = data;
        data = b->data;
        b->data = tmp;
    }
};
```

```
// template 1
template<typename T>
inline void quick_exchange(T *a, T *b)
{
    T tmp(*a);
    *a = *b;
    *b = tmp;
}
```

«More specialized template»
for a C++ compiler

```
// template 2
template<typename T>
inline void quick_exchange(Array<T> *a, Array<T> *b)
{
    a->exchange_with(b);
}

void demo(Array<int> *a1, Array<int> *a2) {
    int x = 1, y = 2;

    quick_exchange(&x, &y);    // template 1
    quick_exchange(a1, a2);    // template 2
}
```

CLASS TEMPLATE SPECIALIZATION

```
template <typename T>
class Storage8 {
    T objects[8];

public:
    void set(int idx, const T &t) {
        objects[idx] = t;
    }

    const T& operator[](int idx) {
        return objects[idx];
    }
};
```

Storage8<bool> can be optimized.

```cpp
template <>
class Storage8<bool> {
    unsigned char bits;

public:
    void set(int idx, bool t) {
        unsigned char mask = 1 << idx;

        if (t)
            bits |= mask;
        else
            bits &= ~mask;
    }

    bool operator[](int idx) {
        return bits & (1 << idx);
    }
};
```

PARTIAL TEMPLATE SPECIALIZATION

```
template <typename T>
class List {
public:
    // ...
    void append(T const &);
    inline size_t length() const;
    // ...
};
```

- Класс **List** будет инстанцироваться для всех вариантов **T**. В большом проекте это может привести к разбуханию кода (**code bloat)**.
- С низкоуровневой точки зрения реализации **List<int *>::append()** и **List<void *>::append()** идентичны.
- Нельзя ли использовать этот факт для оптимизации списков указателей?

```cpp
// Explicit or full specialization of List<void *>
template<>
class List<void *> {
    // ...
    void append(void *p);
    inline size_t length() const;
    // ...
};

// Partial specialization of List<T *>
template <typename T>
class List<T *> {
    List<void *> impl;
public:
    // ...
    void append(T *p) { impl.append(p); }
    inline size_t length() const { return impl.length(); }
    // ...
};
```

MEMBER TEMPLATES

```cpp
template <typename T>
class Vector {
    T *base;
public:
    // ...

    // Dynamic polymorphism
    void print(ostream &os) {
        for (auto const &v : *this)
            os << v;
    }

    // Static polymorphism
    template <typename Out>
    void print(Out &out) {
        for (auto const &v : *this)
            out << v;
    }
};
```

TRAITS CLASSES

- Шаблоны позволяют иметь произвольное количество аргументов.
- Можно настроить любой аспект поведения класса или функции.
- Но передавать всегда и везде по 100 аргументов неудобно…

```
template <typename T>
inline T accum(const T *beg, const T *end) {
    T total = T();  // T() возвращает ноль
                    // Но лучше писать T{}
    while (beg != end)
        total += *beg++;

    return total;
}

///////

int num[] = { 1, 2, 3, 4, 5 };
int avg1 = accum(&num[0], &num[5]) / 5;  // 3

char name[] = "templates";
int len = sizeof(name) - 1;
int avg2 = accum(&name[0], &name[len]) / len; // -5
```

WAT

- Переменная **total** имеет тип **char** (8 бит), которого не хватает для суммирования.

- Можно тип суммы сделать аргументом шаблона… но это очень неудобно.

- Воспользуемся другим подходом.

```cpp
template <typename T>
class AccumulationTraits;

template<>
class AccumulationTraits<char> {
public:
    typedef int AccT;
};

template<>
class AccumulationTraits<short> {
public:
    typedef int AccT;
};

template<>
class AccumulationTraits<int> {
public:
    typedef long AccT;
};

// unsigned int -> unsigned long
// float -> double
// .....
```

```cpp
template <typename T>
inline typename AccumulationTraits<T>::AccT
accum(const T *beg, const T *end) {
    typedef typename AccumulationTraits<T>::AccT AccT;

    AccT total = AccT();  // AccT() возвращает нулевое значение

    while (beg != end)
        total += *beg++;

    return total;
}

char name[] = "templates";
int len = sizeof(name) - 1;
int avg2 = accum(&name[0], &name[length]) / len; // 108
```

```cpp
template <typename T>
class AccumulationTraits;

template<>
class AccumulationTraits<char> {
public:
    typedef int AccT;
    static AccT zero() { return 0; }
};

template<>
class AccumulationTraits<short> {
public:
    typedef int AccT;
    static AccT zero() { return 0; }
};

template<>
class AccumulationTraits<int> {
public:
    typedef long AccT;
    static AccT zero() { return 0; }
};

template <typename T>
inline typename AccumulationTraits<T>::AccT accum(const T *beg, const T *end) {
    typedef typename AccumulationTraits<T>::AccT AccT;

    AccT total = AccumulationTraits<T>::zero();

    while (beg != end)
        total += *beg++;

    return total;
}
```

Member function zero()

```
template <typename T, typename AT = AccumulationTraits<T> >
inline typename AT::AccT accum(const T *beg, const T *end) {
    // ...
}
```

Traits as template parameters

TRAITS CLASSES

- Вместо того чтобы свойства *(типы, константы, …)* делать параметрами шаблона по **T**, помещаем их в дополнительный класс свойств (**Traits**).

- Создаем набор полных специализаций **Traits** для всех нужных **T**.

#include <type_traits>

```cpp
#include <iostream>
#include <array>
#include <string>
#include <type_traits>

using namespace std;

int main() {
    cout << boolalpha;
    cout << "is_array:" << endl;
    cout << "int: " << std::is_array<int>::value << endl;
    cout << "int[3]: " << std::is_array<int[3]>::value << endl;
    cout << "array<int,3>: " << std::is_array<array<int,3>>::value << endl;
    cout << "string: " << std::is_array<string>::value << endl;
    cout << "string[3]: " << std::is_array<string[3]>::value << endl;
    return 0;
}
```

```
is_array:
int: false
int[3]: true
array<int,3>: false
string: false
string[3]: true
```

POLICY CLASSES

```
class SumPolicy {
public:
    template <typename T1, typename T2>
    static void accumulate(T1 &total, const T2 &value) {
        total += value;
    }
};

template <typename T,
          typename Policy = SumPolicy,
          typename Traits = AccumulationTraits<T> >
inline
typename Traits::AccT accum(const T *beg, const T *end) {
    typename Traits::AccT total = Traits::zero();

    while (beg != end) {
        Policy::accumulate(total, *beg);
        ++beg;
    }

    return total;
}
```

Операция суммирования вынесена в Policy

```
class MultPolicy {
public:
    template <typename T1, typename T2>
    static void accumulate(T1 &total, const T2 &value) {
        total *= value;
    }
};
```

Умножение

Тонкая настройка суммирования

```
template <bool use_compound_op = true>
class SumPolicy {
public:
    template <typename T1, typename T2>
    static void accumulate(T1 &total, const T2 &value) {
        total += value;
    }
};

template <>
class SumPolicy<false> {
public:
    template <typename T1, typename T2>
    static void accumulate(T1 &total, const T2 &value) {
        total = total + value;
    }
};
```

```
#include <functional>
```

```cpp
#include <iostream>      // std::cout

int main() {
    int first[] = { 1, 2, 3, 4, 5};
    int second[] = { 10, 20, 30, 40, 50};
    int results[5];

    for (int i = 0; i < 5; ++i)
        results[i] = first[i] + second[i];

    for (int i = 0; i < 5; i++)
        std::cout << results[i] << ' ';

    std::cout << '\n';
    return 0;
}
```

```cpp
#include <iostream>     // std::cout
#include <functional>   // std::plus
#include <algorithm>    // std::transform

int main() {
    int first[]  = { 1, 2, 3, 4, 5};
    int second[] = { 10, 20, 30, 40, 50};
    int results[5];

    std::transform(first, first+5, second, results,
                   std::plus<int>());

    for (int i = 0; i < 5; i++)
        std::cout << results[i] << ' ';

    std::cout << '\n';
    return 0;
}
```

How fix?

```cpp
#include <iostream>     // std::cout
#include <functional>   // std::plus
#include <algorithm>    // std::transform

int main() {
    int first[]  = { 1, 2, 3, 4, 5};
    int second[] = { 10, 20, 30, 40, 50};
    int results[5];

    std::transform(first, first+5, second, results,
                   std::plus<int>());

    std::copy(result, result+5,
              std::ostream_iterator<int>(std::cout, ' '));

    std::cout << '\n';
    return 0;
}
```

```
template <class _Arg1, class _Arg2, class _Result>
struct binary_function
{
    typedef _Arg1 first_argument_type;   ///< the type of the first argument
                                         ///  (no surprises here)

    typedef _Arg2 second_argument_type;  ///< the type of the second argument
    typedef _Result result_type;         ///< type of the return type
};

template <class _Tp>
struct plus : public binary_function<_Tp, _Tp, _Tp> {
    _Tp operator()(const _Tp& __x, const _Tp& __y) const {
        return __x + __y;
    }
};
```

```cpp
#include <iostream>
#include <functional>

int main() {
    int ary[] = { 1, 2, 3, 4, 5 }, res[5];

    using namespace std::placeholders;    // _1, _2, _3,...

    auto inc_10 = std::bind(std::plus<int>(), _1, 10);

    std::transform(ary, ary+5, res, inc_10);

    for (int x: res)
        std::cout << x << std::endl;    // 11… 12… 13… 14… 15
        return 0;
    }
}
```

Currying

```cpp
bool all_under_20 = std::all_of(ary, ary + 5,
    std::bind(std::less<int>(), _1, 20));
```

```
bool all_under_20 = std::all_of(ary, ary + 5,
    [](int n) { return n < 20; });
```

Lambda-expression

```cpp
// Сколько элементов вектора v принадлежат отрезку [loBo, upBo)?
size_t rangeMatch(const vector<int> &v, int loBo, int upBo) {
    return std::count_if(v.begin(), v.end(),
        [loBo, upBo](int _n) {
            return loBo <= _n && _n < upBo;
        });
}
```

"Capturing" variables

```
[loBo, upBo](int _n) {
    return loBo <= _n && _n < upBo;
}

// примерно соответствует:

struct AutomaticallyGenerated {
    AutomaticallyGenerated(int lo, int up)
        : loBo(lo), upBo(up) {}

    bool operator()(int _n) {
        return loBo <= _n && _n < upBo;
    }

    int loBo, upBo;
};
```

```cpp
#include <iostream>
#include <functional>

int main() {
    int ary[] = { 1, 2, 3, 4, 5 }, res[5];

    auto incGen = [] (int _val) -> std::function<int (int)> {
        return [_val] (int _n) -> int { return _n + _val; };
    };

    auto inc_10 = incGen(10);

    std::transform(ary, ary+5, res, inc_10);

    for (int x: res)
        std::cout << x << std::endl;    // 11… 12… 13… 14… 15

        return 0;
    }
}
```

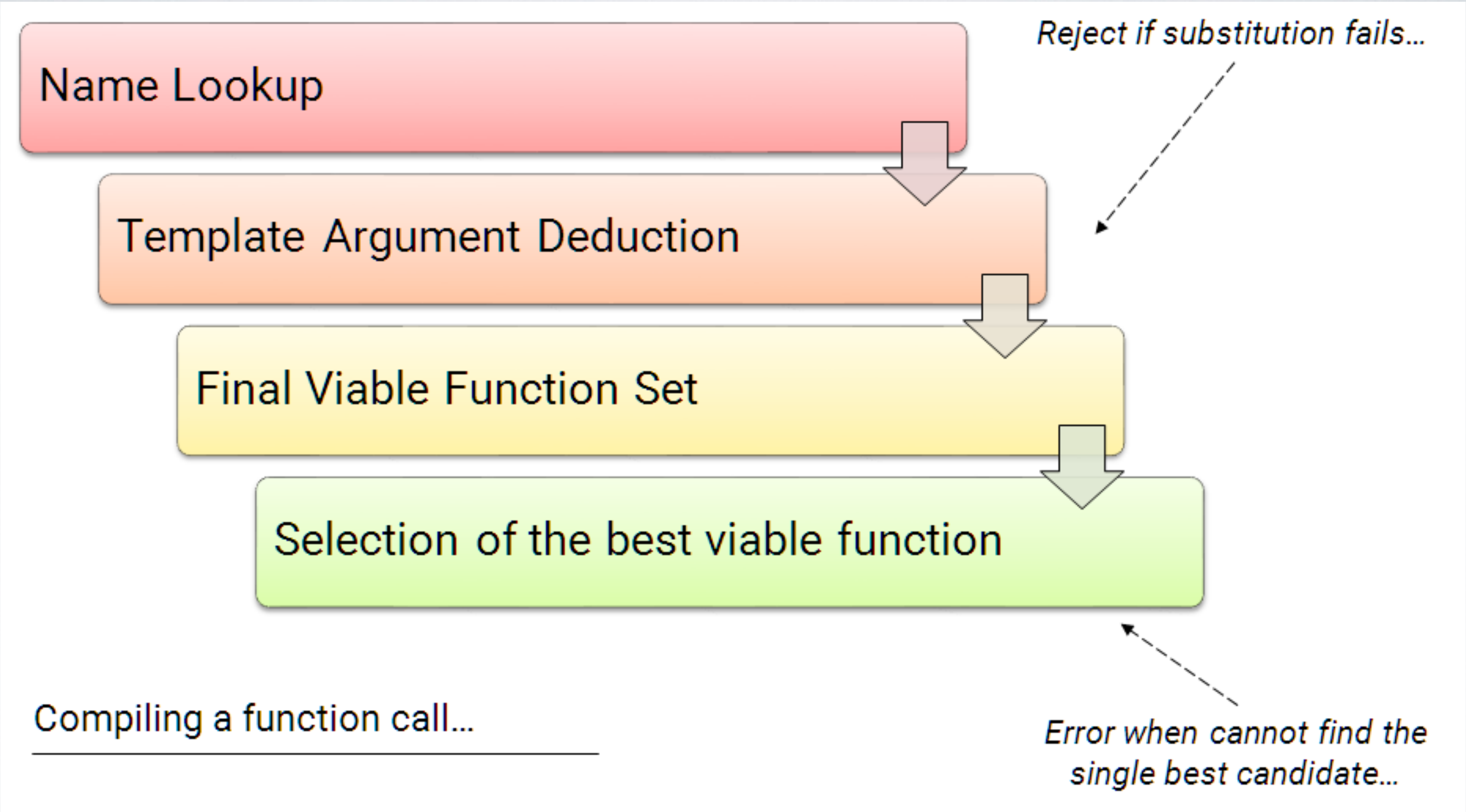
Lambda-expression generates lambda-expressions

SFINAE

```
struct Bar{
    typedef double internalType;
};

template <typename T>
typename T::internalType foo(const T& t) {
    std::cout << "foo<T>" << std::endl;
    return 0;
}

int main(){
    foo(Bar());
    foo(0);   // error
              // no matching function for call to 'foo(int)'
              // ...
              // template argument deduction/substitution failed:
}
```

Substitution Failure Is Not An Error

SFINAE


Name Lookup
Template Argument Deduction
Final Viable Function Set
Selection of the best viable function
Reject if substitution fails...
Error when cannot find the single best candidate...
Compiling a function call...

ENABLE_IF

```
std::enable_if<condition, T>::type

template <typename T>
typename std::enable_if<std::is_arithmetic<T>::value, T>::type
foo(T t) {
    std::cout << "foo<arithmetic T>" << std::endl;
    return t;
}
```

ITERATORS

```cpp
// Until C++17
class num_iterator :
    std::iterator<std::forward_iterator_tag, int>
{
    int i;
public:
    explicit num_iterator(int pos = 0) : i{ pos } {}

    int operator*() const { return i; }

    num_iterator& operator++() {
        ++i;
        return *this;
    }

    bool operator!=(const num_iterator &other) const
{
        return i != other.i;
    }

    bool operator==(const num_iterator &other) const
{
        return !(*this != other);
    }
};
```

```cpp
// Since C++17 std::iterator - deprecated
class num_iterator
{
    int i;
public:
    explicit num_iterator(int pos = 0) :
        i{ pos } {}

    ...
};

namespace std {

    template <>
    struct iterator_traits<num_iterator> {
        using iterator_category =
            std::forward_iterator_tag;
        using value_type = int;
        //using pointer = ...;
        //using reference = ...;
        //using difference_type = ...;
    };

}
```

КОНЕЦ ПЕРВОЙ ЛЕКЦИИ