Лекция по C++

templates

Introduction to templates : reminder

```cpp
template <class T>
T Min(T a, T b) {
    if (a < b) return a;
    return b;
}

int main() {
    auto res1 = Min(1, 2);
    auto res2 = Min(1.3, 2.4);
    auto res3 = Min("asasa", "aaab");
}
```

Introduction to templates : reminder

```cpp
template <class T>
T Min(T a, T b) {
    if (a < b) return a;
    return b;
}

int main() {
    auto res1 = Min(10, 1.4); // error - can't deduce type
    auto res2 = Min<double>(10, 1.4); // ok
}
```

Template struct

```cpp
template <typename T>
struct MyStruct {
    T a, b;
    MyStruct(T a, T b) : a(a), b(b) {}
};

int main() {
    MyStruct<int> a(2, 4);
    MyStruct<std::string> b("lol", "kek");

    MyStruct c(2, 4); // c++17
}
```

Template argument deduction : functions

```
template<class T> void f(std::initializer_list<T>) {};

f({1,2,3});   // P=std::initializer_list<T>, A={1,2,3}
              // P'1=T, A'1=1: deduces T=int
              // P'2=T, A'2=2: deduces T=int
              // P'3=T, A'3=3: deduces T=int
              // deduction succeeds, T = int
f({1,"asdf"}); // P=std::initializer_list<T>, A={1,"asdf"}
               // P'1=T, A'1=1: deduces T=int
               // P'2=T, A'2="asdf", deduces T=const char*
               // deduction fails, T ambiguous
```

Template argument deduction : classes

```
std::pair p(2, 4.5);
// deduces to std::pair<int, double> p(2, 4.5);
std::tuple t(4, 3, 2.5);
// same as auto t = std::make_tuple(4, 3, 2.5);
std::less l;
// same as std::less<void> l;
std::tuple t1(1, 2, 3); // OK: deduction
std::tuple<int,int,int> t2(1, 2, 3); // OK
std::tuple<> t3(1, 2, 3); // Error
std::tuple<int> t4(1, 2, 3); // Error
```

Template struct

```cpp
template <class T, class U>
struct MyPair {
    T a;
    U b;
    MyPair(T a, U b) : a(a), b(b) {}
};

int main() {
    MyPair a(2, "asasa");
}
```

Template struct : reminder

```
template <typename T>
struct MyStruct {
    T a, b;
    MyStruct(T a, T b) : a(a), b(b) {}
    bool operator<(const MyStruct& rhs) const {
        if (a == rhs.a) return b < rhs.b;
        return a < rhs.a;
    }
};
int main() {
    MyStruct<int> a(1, 2);
    MyStruct<long double> b(3, 1);
    a < b;
}
```

Template struct : reminder

```
int main() {
    MyStruct<int> a(1, 2);
    MyStruct<long double> b(3, 1);
    a < b;
}
```

```
error: invalid operands to binary expression ('MyStruct<int>' and
'MyStruct<long double>')
```

Template struct : reminder

```
template <typename T>
struct MyStruct {
    T a, b;
    MyStruct(T a, T b) : a(a), b(b) {}

    template <class R>
    bool operator<(const MyStruct<R>& rhs) const {
        if (a == rhs.a) return b < rhs.b;
        return a < rhs.a;
    }
};
```

Template struct : inheritance

```
template <class T>
struct A {
    A(T a) : a(a) {}
    T a;
};
struct B : A<int> {
    B() : A(13) {}
    void Print() { std::cout << a; }
};
int main() {
    B b;
    b.Print();
}
```

Template struct : inheritance

```
template <class T, class U>
struct Pair {
    Pair(T a, U b) : a(a), b(b) {}
    void Print() { std::cout << a << " " << b; }
    T a; U b;
};
template <class T>
struct SPair : Pair<T, T> {
    // same type for both elements
    SPair(T a, T b) : Pair<T, T>(a, b) {}
};

int main() {
    SPair<int> b(1, 2);
    b.Print(); // 1 2
}
```

Template specialization

```cpp
template <class T>
struct Container {
    T el;
    explicit Container(T el) : el(el) {}
    T Increase() { return ++el; }
};

int main () {
    Container a(7);
    std::cout << a.Increase();
}
```

Template specialization

```cpp
template <class T>
struct Container {
    T el;
    explicit Container(T el) : el(el) {}
    T Increase() { return ++el; }
};

template <>
struct Container<char> {
    char el;
    explicit Container(char el) : el(el) {}
    char Uppercase() {
        el = toupper(el);
        return el;
    }
};
```

Template specialization

```cpp
int main () {
    Container a(7);
    Container b('a');
    std::cout << a.Increase() << " " << b.Uppercase();
    // 8 A
}
```

Partial template specialization

```cpp
template <class T>
struct Container {
    T el;
    explicit Container(T el) : el(el) {}
    T Increase() { return ++el; }
};
template<>
std::string Container<std::string>::Increase() {
    return el += 'a';
}
int main () {
    Container a(7);
    Container<std::string> b("abc");
    std::cout << a.Increase() << " " << b.Increase();
    // 8 abca
}
```

SFINAE (substitution failure is not an error)

```cpp
struct A {
    A(int a) {}
    typedef int B;
};

template <class T>
void Foo(typename T::B v) { std::cout << 1; }

template <class T>
void Foo(T) { std::cout << 2; }

int main() {
    Foo<A>(0); // 1
    Foo<int>(0); // 2
}
```

SFINAE (substitution failure is not an error)

```cpp
struct A {
    A(int a) {}
    typedef int B;
};

template <class T>
void Foo(typename T::B v) {
    std::cout << v.a; // error!
}

template <class T>
void Foo(T) { std::cout << 2; }

int main() {
    Foo<A>(0);
    Foo<int>(0);
}
```

SFINAE (substitution failure is not an error)

```cpp
struct A {
    A(int a) {}
    typedef int B;
};

/* template <class T>
void Foo(typename T::B v) {
    std::cout << v.a;
}*/

template <class T>
void Foo(T) { std::cout << 2; }

int main() { // ok
    Foo<A>(0); // 2
    Foo<int>(0); // 2
}
```

SFINAE (substitution failure is not an error)

SFINAE работает, если не компилируется СИГНАТУРА функции. Если сигнатура компилируется, а тело - нет, это вызовет ошибку компиляции

SFINAE

```
template <class T>
struct IsPointer {
    template <class U> static char IsPtr(U*);
    template <class X, class Y> static char IsPtr(Y X::*);
    static double IsPtr(...);

    static T t;
    enum { value = sizeof(IsPtr(t)) == sizeof(char) };
};

struct Foo { int a; };
int main() {
    printf("%d\n",IsPointer<int*>::value); // 1
    printf("%d\n",IsPointer<int Foo::*>::value); // 1
    printf("%d\n",IsPointer<double>::value); // 0
}
```

SFINAE

```
template <class T>
struct IsPointer {
    enum { value = sizeof(IsPtr(t)) == sizeof(char) };
};
```

sizeof вычисляет размер значения выражения не вычисляя при этом само выражение (то есть функция IsPtr может быть объявлена, но не реализована)

Non-type template parameters

```
template <class T, int N>
struct Sequence {
    T seq[N];
    void SetMember (int x, T value) {
        seq[x] = value;
    }
    T GetMember (int x) {
        return seq[x];
    }
};
```

Non-type template parameters

```
template <class T, int N>
struct Sequence {
    T seq[N];
    void SetMember(int x, T value);
    T GetMember(int x);
};
int main () {
    Sequence<int, 5> int_seq;
    int_seq.SetMember(1, 2);
    std::cout << int_seq.GetMember(1) << std::endl; // 2

    Sequence<std::string, 2> str_seq;
    str_seq.SetMember(0, "abc");
    std::cout << str_seq.GetMember(0) << std::endl; // abc
}
```

Default template parameters

```
template <class T = int, int N = 10>
struct Sequence {
    T seq[N];
    void SetMember (int x, T value) {
        seq[x] = value;
    }
    T GetMember (int x) {
        return seq[x];
    }
};

int main () {
    Sequence<> int_seq; // int sequence of 10 elements
                        // same as Sequence<int, 10>
}
```

Variadic templates (parameter pack)

```
template<class ... Types> struct Tuple {};
Tuple<> t0;            // Types contains no arguments
Tuple<int> t1;         // Types contains one argument: int
Tuple<int, float> t2;  // Types contains two arguments:
                       // int and float
Tuple<0> error;        // error: 0 is not a type
```

Variadic templates : recursion

```
void Print() {}

template <class T, class... Args>
void Print(T arg, Args... args) {
    std::cout << arg << " ";
    Print(args...);
}

int main() {
    Print(1, "sdsa", 2.2); // 1 sdsa 2.2
}
```

Variadic templates : magic

```
template <class F, class... Args>
void Apply(F f, Args... args) {
    int x[] = {f(args)...};
}
template <class... Args>
void Print(Args... args) {
    Apply([&](auto arg)->int {
        std::cout << arg << " ";
    }, args...);
}
int main() {
    Print(1, "sdsa", 2.2); // 1 sdsa 2.2
}
```

enable_if

```
template <class T>
typename std::enable_if<std::is_integral<T>::value,
    T>::type
Foo(T t) {
    return t + t;
}

int main() {
    std::cout << Foo(2); // ok
    std::cout << Foo<std::string>("anjnasd"); // error
}
```

type_traits

```
is_void
is_null_pointer
is_integral
is_floating_point
is_array
is_class
....
```

Template template parameters

```
template<template<typename...>class Cont>
struct UseContainer {
    Cont<int> c; // any container of ints
    explicit UseContainer(const Cont<int>& c) : c(c) {}
    int GetSum() {
        int sum = 0;
        for (auto el : c) sum += el;
        return sum;
    }
};
int main () {
    UseContainer<std::vector> uc({1, 2, 3});
    std::cout << uc.GetSum() << std::endl; // 6
    UseContainer<std::set> sc({4, 3, 1, 4});
    std::cout << sc.GetSum() << std::endl; // 8
}
```

Template template parameters

```
template<template<typename...>class Cont, class T>
struct UseContainer {
    Cont<T> c;
    explicit UseContainer(const Cont<T>& c) : c(c) {}
    void Print() {
        for (auto el : c) std::cout << el << " ";
    }
    T GetFirst() {
        return *c.begin();
    }
};
```

Template template parameters

```
int main () {
    UseContainer<std::vector, int> uc({1, 2, 3});
    uc.Print(); // 1 2 3
    std::cout << uc.GetFirst(); // 1

    UseContainer<std::set, std::string> sc({"c", "f", "a"})
    sc.Print(); // a c f
    std::cout << sc.GetFirst(); // a
}
```