1. 打印任意个参数的模板类

可变参数模板：

```
template<typename T, typename ...ARGS>
void Print(const T &a, ARGS... args) {
    cout << a << endl;
    Print(args...);
}
```

ARGS代表模板中剩余部分的类型，数量是可变的，**但最少为1个**。此代码会递归展开模板函数Print

需说明的是，递归的终止条件需要先写出来，写在模板上面，本质其实就是个函数调用的顺序。

```cpp
// 可以打印任意个参数
class A {
public:
    A(int x, int y) : x(x), y(y) {}
    int x, y;
};
ostream &operator<<(ostream &out, const A &a) {
    out << "Class A : " << a.x << " " << a.y;
    return out;
}

template<typename T>
void Print(T a) {
    cout << a << endl;
    return ;
}
// 这不是模板函数的偏特化，偏特化是可以放在下面的，
// 而且模板函数没有偏特化版本
template<typename T, typename ...ARGS>
void Print(T a, ARGS... args) {
    cout << a << " | ";
    Print(args...);//没有递归终止条件怎么办？—给出终止模板
    return ;
}

int main() {
    A a(5, 6);
    Print("hello world");
    Print("hello world", 3, 3.4, a);
    Print(3, a, 6.7, "hello world");
    return 0;
}
```

2. 实现取参数模板类：

`ARG<ARGS>::type、ARG<ARGS>::rest::type` 代表第一个类型和第二个类型。

```cpp
typename ARGS<ARGS...>::type a
// 前半部分是个类，type是个类型
```

```cpp
template<typename T, typename ...ARGS>
class ARG {
public:
    typedef T type;
    typedef ARG<ARGS...> rest;
};
template<typename T>
class ARG<T> {
public:
    typedef T type;
};

template<typename T, typename ...ARGS>
class Test {
public:
    T operator()(typename ARG<ARGS...>::type a, typename ARG<ARGS...>::rest::type b) {
        return a + b;
    }
};
int main() {
    Test<int, int, int> t1;
    Test<double, double, int> t2;
    cout << t1(3, 4) << endl;
    cout << t2(3.5, 4) << endl;
    return 0;
}
```

3. 可变参但是参数如果不符合要求会报错

```
 template<typename T, typename ...ARGS>
class ARG {
public:
    typedef T type;
    typedef ARG<ARGS...> rest;
};
template<typename T>
class ARG<T> {
public:
    typedef T type;
    typedef T ftype;//只有最后一层能接触到ftype，跟传的参数个数有影响
};

template<typename T, typename ...ARGS>
class Test {
public:
    T operator()(
        typename ARG<ARGS...>::type a,
        typename ARG<ARGS...>::rest::ftype b) {
        return a + b;
    }
};

int main() {
    Test<int, int, int> t1;
    Test<double, double, int> t2;
    // Test<double, int, int, int> t3;
    cout << t1(3, 4) << endl;
    cout << t2(3.5, 4) << endl;
    return 0;
}
```

4. 其实类模板有更特殊的偏特化形式

你看，`Test<int, int, int> t1`定义的是可调用对象，可以把它看成是函数，既然是函数，那么应该可以定义成更人性化的方式：`Test<int(int, int)> t1`，那该如何修改代码才能实现呢？

```cpp
template<typename T, typename ...ARGS>
class ARG {
public:
    typedef T type;
    typedef ARG<ARGS...> rest;
};
template<typename T>
class ARG<T> {
public:
    typedef T type;
    typedef T ftype;//只有最后一层能接触到ftype，跟传的参数个数有影响
};

template<typename T, typename ...ARGS> class Test;// 原模板声明
template<typename T, typename ...ARGS> //模板特殊偏特化版本
class Test<T(ARGS...)> {
public:
    T operator()(
        typename ARG<ARGS...>::type a,
        typename ARG<ARGS...>::rest::ftype b) {
        return a + b;
    }
};

int main() {
    Test<int(int, int)> t1;
    Test<double(double, int)> t2;
    // Test<double, int, int, int> t3;
    cout << t1(3, 4) << endl;
    cout << t2(3.5, 4) << endl;
    return 0;
}
```

5. 编译期常量

constexpr 关键字的功能是使指定的常量表达式获得在程序编译阶段计算出结果的能力，而不必等到程序运行阶段

模板之所以强大，还因为模板不仅可以传参数，还可以传**编译期常量**。说白了就是

```
template<int M> //模板函数
void Print() {
    cout << M << ", ";
    Print<M - 1>();
    return ;
}
template<> //全特化版本
void Print<1>() {
    cout << 1 << endl;
    return ;
}
int main() {
    Print<10>();
    Print<5>();
    Print<6>();
    Print<7>();
    return 0;
}
```

6. 用一个整型来指定第几个参数

作为一个技术人员，需要敏锐地判断出什么技术能实现，什么技术不能实现。
对于实现不了的技术，如何通过最少改动成为一个更加合理的需求。

```
template<typename T, typename ...ARGS> class Test;// 原模板声明
template<typename T, typename ...ARGS>
class Test<T(ARGS...)> {
public:
        // 需求2:
    typedef ARG<2, ARGS...>::ftype TYPE_NUM_2;
    // 需求1
    T operator()(
        typename ARG<1, ARGS...>::type a,
        typename ARG<2, ARGS...>::type b) {
        return a + b;
    }
};
```

这个需求分成两部分：

1. 通过传入一个数字和一个变参列表，去得到变参列表中的第n项。
2. 设计一种方式去判断变参列表中的数量到底是多少个然后把参数的数量控制在多少

个。

```
template<int N, typename T, typename ...ARGS>
// 分离被解析出来的变参列表中的第一个参数
struct ARG {
        // 我在剩余部分选第N-1项参数:
        typedef typename ARG<int N-1, ARGS...>::type type;
}
// 接下来定义偏特化版本，递归终止。无外乎两个版本:
// 一、 N减到了1，但是参数个数还大于等于两个
template<typename T, typename ...ARGS>
struct ARG<1, T, ARGS...> {
        typedef T type;
}
// 二、 N减到了1，但是参数只剩下T
template<typename T>
struct ARG<1, T> {
        typedef T type;
}
```

第二个需求虽然不太好实现，但可以通过曲线救国的形式实现。

```
// 首先需要几个东西:
// 工具1. 统计变参列表中参数个数的模板
template<typename T, typename ...ARGS>
struct NUM_ARGS {
        // 应该有一个编译器常量用来计数:
        static constexpr int r = NUM_ARGS<ARGS...>::r + 1;
};
// 偏特化版本，定义递归边界
template<typename T>
struct NUM_ARGS {
        static constexpr int r = 1;
};
int main() {
        cout << NUM_ARGS<int, int, int, int>::r << endl;
        cout << NUM_ARGS<int, int, int>::r << endl;
        cout << NUM_ARGS<int, int>::r << endl;
        return 0;
}
```

```
// 工具2：判断数字到底相不相等
template<int N>
struct Zero {
        typedef int no;
}
template<>
struct Zero<0> {
        typedef int yes;
}
// 实现如下：如果变参列表里有2个参数的话，Zero里应该有一个yes类型
// typename关键字起到明确声明后面那个表达式是一个类型的作用
typedef typename Zero<2 - NUM_ARGS<ARGS...>::r>::yes TYPE_NUM_2;
```

完整代码：

```cpp
template<int N, typename T, typename ...ARGS>
struct ARG {
    typedef typename ARG<N-1, ARGS...>::type type;
};
template<typename T, typename ...ARGS>
struct ARG<1, T, ARGS...> {
    typedef T type;
};
template<typename T>
struct ARG<1, T> {
    typedef T type;
};

// 首先需要几个东西:
// 1. 统计变参列表中参数个数的模板
template<typename T, typename ...ARGS>
struct NUM_ARGS {
        // 应该有一个编译器常量用来计数:
        static constexpr int r = NUM_ARGS<ARGS...>::r + 1;
};
// 偏特化版本，定义递归边界
template<typename T>
struct NUM_ARGS<T> {
        static constexpr int r = 1;
};

template<int N>
struct Zero {
    typedef int no;
};
template<>
struct Zero<0> {
    typedef int yes;
};

template<typename T, typename ...ARGS> class Test;
template<typename T, typename ...ARGS>
class Test<T(ARGS...)> {
public:
    typedef typename Zero<2 - NUM_ARGS<ARGS...>::r>::yes TYPE_NUM_2;
    // typedef ARG<2, ARGS...>::ftype TYPE_NUM_2;
    T operator()(
        typename ARG<1, ARGS...>::type a,
        typename ARG<2, ARGS...>::type b) {
        return a + b;
    }
```

```
};

int main() {
    cout << NUM_ARGS<int, int, int, int>::r << endl;
        cout << NUM_ARGS<int, int, int>::r << endl;
        cout << NUM_ARGS<int, int>::r << endl;
    Test<bool(int, int)> t1;
    // Test<bool(int, int, int)> t2;
    cout << t1(3, 4) << endl;

    return 0;
}
```

7. 模板最终测试

```cpp
#include<iostream>
using namespace std;

#define BEGINS(x) namespace x {
#define ENDS(x) }

BEGINS(sum_test)
template<int N>
struct sum {
    static constexpr int r = sum<N-1>::r + N;
};
template<>
struct sum<1> {
    static constexpr int r = 1;
};
int main() {
    cout << sum<5>::r << endl;
    cout << sum<7>::r << endl;
    cout << sum<100>::r << endl;
    return 0;
}
ENDS(sum_test)

BEGINS(even_test)
template<int N>
struct YES_OR_NO {
    static const char * r;
};
template<>
struct YES_OR_NO<0> {
    static const char *r;
};
template<int N>
const char *YES_OR_NO<N>::r = "yes";
const char *YES_OR_NO<0>::r = "no";

template<int N>
struct is_even {
    static const char * r;
};
template<int N>
const char *is_even<N>::r = YES_OR_NO<!(N % 2)>::r;
//声明和定义分开写。

int main() {
    cout << is_even<5>::r << endl;
```

```
    cout << is_even<6>::r << endl;
    return 0;
}
ENDS(even_test)

BEGINS(good_bad_test)

template<int N>
struct GOOD_OR_BAD {
    static const char * r;
};
template<>
struct GOOD_OR_BAD<0> {
    static const char *r;
};
template<int N>
const char *GOOD_OR_BAD<N>::r = "good";
const char *GOOD_OR_BAD<0>::r = "bad";

template<int N>
struct score_judge {
    static const char *r;
};
template<int N>
const char *score_judge<N>::r = GOOD_OR_BAD<(N >= 60)>::r;
int main() {
    cout << score_judge<59>::r << endl;//bad
    cout << score_judge<60>::r << endl;//good
    return 0;
}
ENDS(good_bad_test)

BEGINS(is_prime_test)
template<int i, int N>
struct getNext {
    static constexpr int r = (N % i ? i + 1 : 0);
};

template<int i, int N>
struct test {
    static constexpr const char *r
    = (i * i > N) ? "yes" : test<getNext<i, N>::r, N>::r;
};
template<int N>
struct test<0, N> {
    static constexpr const char *r = "no";
```

```
};

template<int N>
struct is_prime {
    static constexpr const char *r = test<2, N>::r;
};


int main() {
    cout << "2 : " << is_prime<2>::r << endl;//yes
    cout << "3 : "<< is_prime<3>::r << endl;//yes
    cout << "5 : "<< is_prime<5>::r << endl;//yes
    cout << "8 : " << is_prime<8>::r << endl;//no
    cout << "103 : " << is_prime<103>::r << endl;//yes
    return 0;
}
ENDS(is_prime_test)

int main() {
    sum_test::main();
    even_test::main();
    good_bad_test::main();
    is_prime_test::main();
    return 0;
}
```