1. Template parameter deduction

In `foo()` below we define `T` to be a template's type parameter, and `T&&` to be a function parameter.

Given the following calls, indicate `T` and resultant `T&&` after type deduction is done by a compiler. Indicate whether the resulting call is compileable by writing "**C**"/"**NC**" in the last column. Assume that all appropriate headers have been included.

Solution: a column from which you should start analysing a case has been indicated with cyan, while a part of the function call that tells to start there is in yellow for l-values, green for r-values, and magenta for explicit template parameters (pay attention to cyan with magenta).

```
1  template <typename T>
2  void foo(T&& t)
3  {
4      T&& baz{std::forward<T>(t)};
5  }
6  int x = 5;
7  int* px = &x;
8  int& rx = x;
9  int&& rrx = std::move(x);
```

#	Template function call	Template param. `T`	Function param. `T&&`	Compilable? **C/NC**
-	`foo(x);`	*int&*	*int&*	*C*
a)	`foo(std::vector<int>{});`	`std::vector<int>`	`std::vector<int>&&`	C
b)	`foo(rx);`	`int&`	`int&`	C
c)	`foo(rrx);`	`int&`	`int&`	C
d)	`foo<int&&>(*px);`	`int&&`	`int&&`	NC
e)	`foo(px);`	`int*&`	`int*&`	C
f)	`foo<int&>(std::move(x));`	`int&`	`int&`	NC
g)	`foo(std::move(rx));`	`int`	`int&&`	C
h)	`foo<int&&>(rrx);`	`int&&`	`int&&`	NC
i)	`foo<int&&>(std::move(rrx));`	`int&&`	`int&&`	C
j)	`foo<long int>(px[0]);`	`long int`	`long int&&`	C

2. Template reference collapsing rules

Indicate the output printed by each line of code.

Solution: check what parameters these functions accept, and how they accept and forward them.

```
1  #include <iostream>
2  #include <vector>
3  template <typename T> void f(const T&)  { std::cout << "& "; }
4  template <typename T> void f(const T&&) { std::cout << "&& "; }
5  template <typename T> void a(T&& t) { f(static_cast<T&&>(t)); }
6  template <typename T> void b(T&& t) { f(std::forward<T>(t)); }
7  template <typename T> void c(T&& t) { f(std::move(t)); }
8  bool& getBool() { static bool x = false; return x; }
9
10 int main() {
11     const float f = 1.0f;
12     std::vector<int> v;
13
14     a(f); a(v); a(getBool()); a(std::move(v)); // a) & & & &&
15     std::cout << std::endl;
16
17     b(f); b(v); b(getBool()); b(std::move(v)); // b) & & & &&
18     std::cout << std::endl;
19
20     c(f); c(v); c(getBool()); c(std::move(v)); // c) && && && &&
21     std::cout << std::endl;
22 }
```

3. Function template name lookup

Indicate the output printed by each line of code.

Solution: consider perfect match functions first, before matching base templates and specializations.

```
#include <iostream>
#include <complex>

template <typename T1, typename T2>
void f(T1, T2)          { std::cout << "1" << std::endl; }

template <typename T>
void f(T)               { std::cout << "2" << std::endl; }

template <typename T>
void f(T, T)            { std::cout << "3" << std::endl; }

template <typename T>
void f(T*)              { std::cout << "4" << std::endl; }

template <typename T>
void f(T*, T)           { std::cout << "5" << std::endl; }

template <typename T>
void f(T, T*)           { std::cout << "6" << std::endl; }

template <typename T>
void f(int, T*)         { std::cout << "7" << std::endl; }

template <>
void f<int>(int)        { std::cout << "8" << std::endl; }

void f(int, double)     { std::cout << "9" << std::endl; }

void f(int)             { std::cout << "10" << std::endl; }

int main()
{
        int i = 0;
        double d = 0.0;
        float ff = 0.0f;
        std::complex<double> c{};

        f(i);                                   // a)   10
        f<int>(i);                              // b)   8
        f(i, i);                                // c)   3
        f(c);                                   // d)   2
        f(i, ff);                               // e)   1
        f(i, d);                                // f)   9
        f(c, &c);                               // g)   6
        f(i, &d);                               // h)   7
        f(&d, d);                               // i)   5
        f(&d);                                  // j)   4
        f(d, &i);                               // k)   1
        f(&i, &i);                              // l)   3
}
```