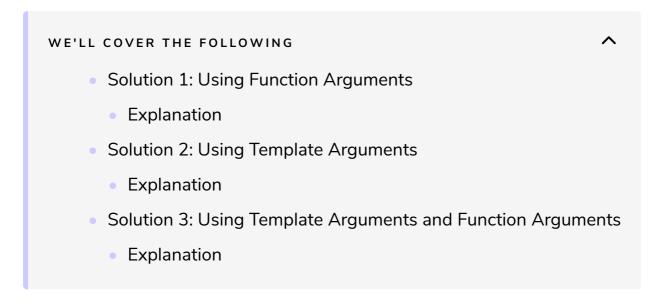
- Solution

We'll learn different approaches to solve the previous challenge.



Solution 1: Using Function Arguments

```
// power1.cpp
#include <iostream>
int power(int m, int n){
   int r = 1;
   for(int k=1; k<=n; ++k) r*= m;
   return r;
}
int main(){
   std::cout << power(2,10) << std::endl;
}</pre>
```

Explanation

We're using a for loop to compute the power. The loop runs a total of n times by multiplying the number m by r for every iteration of the loop in line 7.

For a more in-depth insight into the solution above, click here. It shows how

things are handled at the assembler level.

The critical point of this example is that the function runs at runtime.

Solution 2: Using Template Arguments

```
// power2.cpp
#include <iostream>
template<int m, int n>
struct Power{
    static int const value = m * Power<m,n-1>::value;
};
template<int m>
struct Power<m,0>{
    static int const value = 1;
};
int main(){
    std::cout << Power<2,10>::value << std::endl;
}</pre>
```

Explanation

The call Power<2, 10>::value in line 16 triggers the recursive calculation. First, the primary template in line 5 is called, then the Power<m, n-1>::value in line 7 is executed. This expression instantiates recursively until the end condition is met: n is equal to 0. Now, the boundary condition in line 12 is applied, which returns 1. In the end, Power<2, 10>::value contains the result.

To view how things are happening at the assembler level, click here.

The critical point is that the calculation is done at compile-time.

Solution 3: Using Template Arguments and Function Arguments



```
int power(int m){
    return m * power<n-1>(m);
template<>
int power<1>(int m){
    return m;
template<>
int power<0>(int m){
    return 1;
int main(){
    std::cout << power<10>(2) << std::endl;</pre>
```







Explanation

In the code above, the power function template exists in three variations. First, in the primary template in line 6. Second and third in the full specializations for 1 and 0 in lines 11 and 16. The call power<10>(2) triggers the recursive invocation of the primary template. The recursion ends with the full specialization for 1. After studying the example carefully, note that the full specialization for 1 is not necessary in this case because the full specialization for 0 is also a valid boundary condition.

When we invoke the power<10>(2) function, the argument in () brackets is evaluated at run-time and the argument in $\langle \rangle$ brackets is evaluated at compile-time. Therefore, we can say that the round brackets are run time arguments and the angle brackets are compile-time arguments.

Click here to view the assembler code.

In the next lesson, we'll learn about class templates in detail.