- Examples

We'll look at a few examples of template metaprogramming in this lesson.

WE'LL COVER THE FOLLOWING

Example 1: Template Prime Number

```
// templatePrimeNumber.cpp
// Prime number computation by Erwin Unruh

template <int i> struct D { D(void*); operator int(); };

template <int p, int i> struct is_prime {
    enum { prim = (p==2) || (p%i) && is_prime<(i>2?p:0), i-1> :: prim };
};

template <int i> struct Prime_print {
    Prime_print<i-1> a;
    enum { prim = is_prime<i, i-1>::prim };
    void f() { D<i> d = prim ? 1 : 0; a.f();}
};

template<> struct is_prime<0,0> { enum {prim=1}; };
template<> struct is_prime<0,1> { enum {prim=1}; };

template<> struct Prime_print<1> {
    enum {prim=0};
    void f() { D<1> d = prim ? 1 : 0; };
};

#ifndef LAST
#define LAST 18
#endif
```

```
int main() {
  Prime_print<LAST> a;


  a.f();
}
```

Explanation

This is the original prime number program by Erwin Unruh, which was the starting point of template metaprogramming. Current compilers will not produce the same output as the ancient compiler, which Erwin Unruh used more than *20* years ago.

Example 2: Template Type Manipulation

```
// templateTypeManipulation.cpp

#include <iostream>
#include <type_traits>

template <typename T>
struct RemoveConst{
    typedef T type;
};

template <typename T>
struct RemoveConst<const T>{
    typedef T type;
};


int main(){

    std::cout << std::boolalpha << std::endl;

    std::cout << "std::is_same<int, RemoveConst<int>::type>::value: " << std::is_same<int, Re
    std::cout << "std::is_same<int, RemoveConst<const int>::type>::value: " << std::is_same<i

    std::cout << std::endl;

}
```

Explanation

The code uses the function `std::is_same` from the type-traits library. `std::is_same` compares the type passed and returns at compile time if they are the same. Thanks to the type-traits function, we can verify the

RemoveConst class template from the previous subsection.

Example 3: Template Power

```
// templatePower.cpp

#include <iostream>

int power(int m, int n){
    int r = 1;
    for(int k=1; k<=n; ++k) r*= m;
    return r;
}

template<int m, int n>
struct Power{
    static const int value = Power<m,n-1>::value * m;
};

template<int m>
struct Power<m,0>{
    static const int value = 1;
};

template<int n>
int power2(const int& m){
    return power2<n-1>(m) * m;
}

template<>
int power2<1>(const int& m){
    return m;
}

template<>
int power2<0>(const int&){
    return 1;
}

int main(){

    std::cout << std::endl;

    std::cout << "power(2,10):        " << power(2,10) << std::endl;
    std::cout << "power2<10>(2):       " << power2<10>(2) << std::endl;
    std::cout << "Power<2,10>::value: " << Power<2,10>::value << std::endl;

    std::cout << std::endl;
}
```

Explanation

The program calculates 2^{10} in three different variants.

- `power` is a function in line 5
- `Power` is a class template in line 12
- `power2` is a function template in line 22

The key question is: When is the function executed?

- `power` runs at runtime
- `Power` runs at compile-time
- `power2` runs at runtime and at compile-time too
    - the template argument is evaluated at compile-time
    - the function argument is evaluated at runtime

We’ll solve an exercise in the next lesson.