ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

TWO-PHASE TRANSLATION

```
std::complex<float> c1, c2;  // Doesn't provide operator<.
...
std::max(c1, c2);            // Error at compile time.
```

TWO-PHASE TRANSLATION

```
std::complex<float> c1, c2;  // Doesn't provide operator<.
...
std::max(c1, c2);            // Error at compile time.
```

Templates are "compiled" in two phases:

1. *Definition time.*
    A. *Syntax errors.*
    B. *Using unknown names (type names, function names, …) that don't depend on template parameters.*
    C. *Static assertions that don't depend on template parameters.*
2. *Instantiation time.*

TWO-PHASE TRANSLATION

```
template<typename T>
void foo(T t)
{
  undeclared();   // first-phase compile-time error if
                  // undeclared() unknown
  undeclared(t);  // second-phase compile-time error if
                   // undeclared(T) unknown

  static_assert(sizeof(int) > 10, // always fails if
               "int too small"); // sizeof(int)<=10

  static_assert(sizeof(T) > 10, "T too small");
        // fails if instantiated for T with size <=10
}
```

TWO-PHASE TRANSLATION

```
template<typename T>
void foo(T t)
{
   undeclared();

   int a = 5
   return;
}
```

missing semicolon

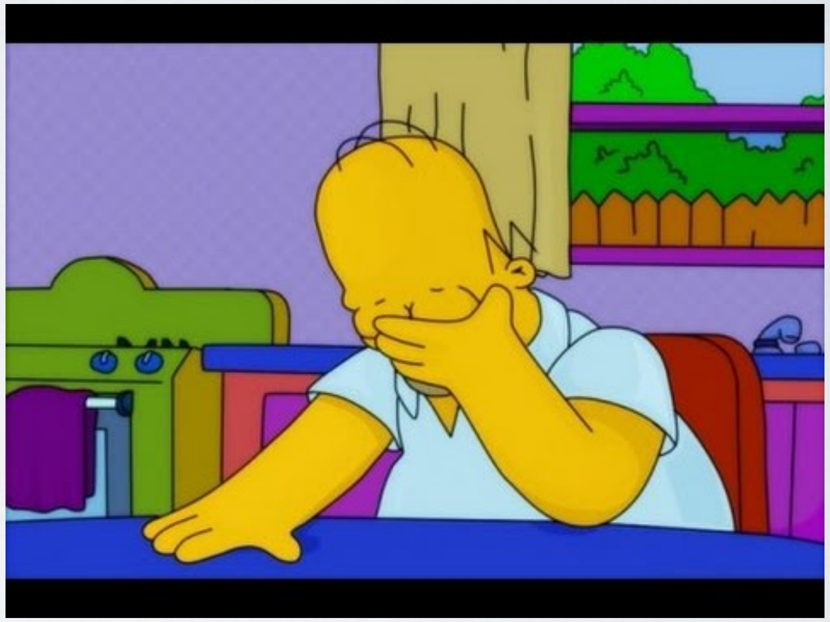
Is it compile???

TWO-PHASE TRANSLATION

```
template<typename T>
void foo(T t)
{
   undeclared();

   int a = 5
   return;
}
```

missing semicolon



Some compilers don't perform the full checks of the first phase.

Visual C++ success compiled this code.

LINKER ERRORS

```cpp
// example.hpp
#pragma once

// declaration of template
template <typename T>
void printTypeof(T const&);
```

```cpp
// example.cpp
#include <iostream>
#include <typeinfo>
#include "example.hpp"

// implementation/definition of template
template <typename T>
void printTypeof(T const&){
   std::cout << typeid(x).name() << '\n';
}
```

LINKER ERRORS

```
// main.cpp
#include "example.hpp"

// use of the template
int main(){

   double ice = 3.0;
   printTypeof(ice); // call function template
for
                     // type double
}
```

The function template *printTypeof()* has not been instantiated.

INCLUSION MODEL

```
// example.hpp
#pragma once
#include <iostream>
#include <typeinfo>

// declaration of template
template <typename T>
void printTypeof(T const&);

// implementation/definition of template
template <typename T>
void printTypeof(T const&){
   std::cout << typeid(x).name() << '\n';
}
```

INCLUSION MODEL

OR

```
// example.hpp
#pragma once
#include <iostream>
#include <typeinfo>

// declaration and implementation/definition of template
template <typename T>
void printTypeof(T const&){
   std::cout << typeid(x).name() << '\n';
}
```

INCLUSION MODEL

```cpp
// example.hpp
#pragma once
#include <iostream>
#include <typeinfo>

// declaration of template
template <typename T>
void printTypeof(T const&);

// implementation/definition of template
template <typename T>
void printTypeof(T const&){
   std::cout << typeid(x).name() << '\n';
}
```

Including the headers

Increase the cost of including the header file

PRECOMPILED HEADERS (PCH)

```
// example1.cpp
...


N string of code
N+1 string of code
```

```
// example2.cpp
...


N string of code
N+1 string of code
```

```
// example(i).cpp
...


N string of code
N+1 string of code
```

Same code

PRECOMPILED HEADERS (PCH)

```cpp
// std.hpp
#include <iostream>
#include <string>
#include <vector>
#include <deque>
#include <list>

...
```

```cpp
// Every program file started as follows
#include "std.hpp"

...
```

DEFAULT TEMPLATE ARGUMENTS

```
template <typename RT, typename T1, typename T2>
RT const &max(T1 const &x, T2 const &y) {
    return x > y ? x : y;
}

// ....
max<double>(4, 4.2);  // Type inference is not possible for RT
```

DEFAULT TEMPLATE ARGUMENTS

```
template <typename RT = double, typename T1, typename T2>
RT const &max(T1 const &x, T2 const &y) {
    return x > y ? x : y;
}

// ....
max(4, 4.2);  // returns double (default argument of template
              // parameter for return type

max<int>(4, 4.2); // returns int
```

INLINE

```cpp
// source1.cpp
#include "header.hpp"
```

```cpp
// source2.cpp
#include "header.hpp"
```

```cpp
// header.hpp
#pragma once

template<typename T> void f(T){}
template<typename T> inline T g(T){}

template<> inline void f<>(int){}      // OK: inline
template<> int g<>(int){}              // Error: not inline
```

Explicit specialization

One-definition rule (ODR)

STATIC_ASSERT

```
// header.hpp
#pragma once
...

template<typename T>
class Sample{
   static_assert(std::is_default_constructible<T>::value,
                 "Class C requires default-constructible elements");
   ...
};

//OR
template<typename T>
void func(T){
   static_assert(std::is_fundamental<T>::value,
                 "Function func requires fundamental elements");
   ...
};
```

NONTYPE FUNCTION TEMPLATE PARAMETERS

```cpp
template <int Val, typename T>
T addValue(T x) {
    return x + Val;
}

// ....
std::transform(source.begin(), source.end(), dest.begin(),
               addValue<5, int>);

auto val = addValue<10>(5);  // int val = 15;
```

NONTYPE FUNCTION TEMPLATE PARAMETERS

```
template <auto Val, typename T = decltype(Val)>
T foo();
```

```
template <typename T, T Val = T{}>
T bar();
```

NONTYPE CLASS TEMPLATE PARAMETERS

```
template<typename T = int, std::size_t Maxsize = 100>
class Stack {
   std::array<T, Maxsize> elems;
   std::size_t numElems;
public:
   Stack();
   ...
};

template<typename T, std::size_t Maxsize>
Stack<T,Maxsize>::Stack() : numElems(0)
{
   // nothing else to do
}
```

ALIAS DECLARATION

```
std::unique_ptr<std::unordered_map<std::string, std::string>> ptr;

// typedef specifier
typedef std::unique_ptr<std::unordered_map<std::string, std::string>> UPtrMapSS;

// alias declaration
using UPtrMapSS = std::unique_ptr<std::unordered_map<std::string, std::string>>;
```

ALIAS ADVANTAGES

- **More readable.**
- **Alias declaration can be templated**.

USING VS TYPEDEF

```
// typedef specifier
typedef void (*FP) (int, const std::string&);

// alias declaration
using FP = void (*) (int, const std::string&);
```

ALIAS TEMPLATES

```cpp
// MyAlloc - custom memory allocator.
template <typename T>
struct MyAllocList
{
    typedef std::list<T, MyAlloc<T>> type;
};

MyAllocList<ObjectType>::type lw; // Client
                                  // code

// MyAllocList for member types
template <typename T>
struct Widget
{
private:
    //MyAllocList<T>::type - dependent type
    typename MyAllocList<T>::type list;
};
```

```cpp
template <typename T>
using MyAllocList = std::list<T,
                              MyAlloc<T>>;

//Removed suffix "::type"
MyAllocList<ObjectType> lw; // Client
                            // code


// MyAllocList for member types
template <typename T>
struct Widget
{
private:
    //Removed typename, removed ::type
    MyAllocList<T> list;
};
```

#include <type_traits>

```
std::remove_const<T>::type            //C++11 : const T -> T
std::remove_reference<T>::type           //C++11 : T& / T&& -> T
std::add_lvalue_reference<T>::type //C++11 : T -> T&

template <typename T>
using remove_const_t = typename std::remove_const<T>::type;

template <typename T>
using remove_reference_t = typename std::remove_reference<T>::type;

template <typename T>
using add_lvalue_reference_t = typename std::add_lvalue_reference<T>::type;

std::remove_const_t<T>              //C++14 : const T -> T
std::remove_reference_t<T>          //C++14 : T& / T&& -> T
std::add_lvalue_reference_t<T>         //C++14 : T -> T&
```

КОНЕЦ ВТОРОЙ ЛЕКЦИИ