Шаблоны

Специализация, инстанцирование, вывод типов

Инстанцирование - процесс порождения специализации.

```
template <typename T>
T max(T x, T y) { return x  y ? x : y; }

// неявно породится декларация и определение (мб в разных местах)
// template<> int max(int, int) { ... }

int main() {
    max<int>(2,3); // запускает неявное инстанцирование
}
```

Явная специализация может войти в конфликт с инстанцированием, если та указана после запуска инстанцирования компилятором.

```
// явная специализация
template<> duble max(double x, double y) { /* ... */ }
```

Специализации можно явно удалять

```
template <> foo(int*) = delete;
```

Неявное инстанцирование - лениво. Для энергичного инстанцирования можно явно это указать. Синтаксис похож на специализацию, **но это совершенно другое!**

```
template <> int foo(int, int) {/* ... */} // полная специализация
template int foo<int>(int, int); // явное инстанцирование ВСЕГО
extern template int foo<int>(int, int); // явное инстанцирование ВСЕГО уже
где-то было сделано в другом модуле.
```

```
template <int N> struct Danger {
    typedef char block[N];
};

template <typename T, int N>
struct Tricky {
    void test_lazyness() {
        Danger<N> no_boom_yet;
   }
};

int main() {
    Tricky<int -2> ok; // ok (if I is commented)
    // ok.test_lazyness(); // error if uncomment
}

// template struct Tricky<int, -2>; // I
```

Для шаблонов классов (**и только для них**) возможна частичная специализация.

```
 // primary template
 template <typename T, typename U>
 class Foo {};

// частичные специализации класса
 template <typename T>
 class Foo<T,T> {};

 template<typename T>
 class Foo<T, int> {};

template<typename T, typename U>
class Foo<T*, U*> {};
```

```
// primary template
template <typename T> struct X;
// partial spec
template <typename T> struct X<std::vector<T>>;

// primary template
template <typename R, typename T>
struct Y;

// partial spec
template <typename R, typename T>
struct Y<R(T)>;
```

> Частичный порядок частичных специализаций - нетривиальная вещь.

В телах шаблонных классов (и только в них) действует сокращение имен:

```
// primary template
template <class T> class A {
    A* a1; // A <==> A<T>
};

// partial spec
template <class T> class A<T*> {
    A* a1; // A <==> A<T*>
};
```

> Можно частично специализировать default deleter в unique ptr для типов с шаблоном T[], т.е. для массивов вызывать `delete[]` ...

Частичная специализация никак не связана с наследованием! Частичная специализация и примари шаблон незвасимы, т.е. шаблоны инвариантны к специализации.

Разрешение имен в шаблонах

```
template <typename T> struct Foo {
   int use () { return illegal_name; } // independent name
   int use2() { return T::illegal_name; } dependent name (has semantic relationship)
};
```

Двухфазное разрешение имен:

- Фаза до инстанцирования: разрешение независимых от шаблонных параметров имен, синтаксическая проверка
- Фаза во время инстанцирования: специальные синтаксические проверки и разрешение зависимыхот шаблонов имен.

> `Usage ~~~ Declaration` - процесс разрешения имен (компилятор)
> `Declaration ~~~ Definition` - процесс для линковк (в основном работает с unrezolved references) и компилятора

Разрешение зависимых от шаблонов имен откладывается до подстановки шаблонных параметров

Как можно указывать зависимости от шаблонных параметров (disambiguation)

```
template <typename T> struct Base {
    void exit();
};

template <typename T> struct Derived : Base<T> {
    void foo() {
        exit(); // std::exit() - phase 1
        Base::exit(); // phase 2
        this->exit() // phase 2
    }
};
```

Использование `this->` в данном случае - обосновано (**пример Вандерворда**) и является единственным адекватным применением конструкции.

```
struct S {
    struct subtype {};
};

template <typename T> int foo (const T& x) {
    //T::subtype *y; // error, * - mul
    typename T::subtype *y // ok
}

int main() {
    foo<S>(S{});
}
```

Все что похоже на поле - это поле. Тип - второй приоритет! Для типов лучше использовать `typename`.

```cpp
template<typename T> struct S {
    template<typename U> void foo() {}
};

template<typename T> void bar() {
    S<T> s; s.foo<T>(); // < - less + syntax error
    S<T> s; s.template foo<T>(); // ok
}
```