Метапрограммирование

Александр Смаль

CS центр
16 апреля 2015
Санкт-Петербург

Вычисления в compile-time

```
template<int N> struct Fact {
    static const int value = N * Fact<N - 1>::value;
};
template<> struct Fact<0> {
    static const int value = 1;
};

template<int N> struct Fib {
    static const int
        value = Fib<N - 1>::value + Fib<N - 2>::value;
};
template<> struct Fib<0> {
    static const int value = 0;
};
template<> struct Fib<1> {
    static const int value = 1;
};

int main() {
    std::cout << Fact<10>::value << std::endl
              << Fib <10>::value << std::endl;
}
```

Алгебраические типы данных

```
struct nil {};

template<class H, class T = nil>
struct cons {
    typedef T Tail;
    typedef H Head;
};

typedef
    cons<int, cons<std::string, cons<double, cons<float> > > >
    TypeList;

template<class TL>
void print() {
    std::cout << typeid(typename TL::Head).name() << std::endl;
    print<typename TL::Tail>();
}

template<>
void print<nil>() { }
```

Список на variadic templates

```
template<class ... Args> struct List;

template<class H, class ...Args>
struct List<H, Args...> {
    typedef H              Head;
    typedef List<Args...>  Tail;
};

typedef List<> nil;

typedef List<int, std::string, double, float> TypeList;

template<class TL>
void print() {
    std::cout << typeid(typename TL::Head).name() << std::endl;
    print<typename TL::Tail>();
}

template<>
void print<nil>() { }
```

Алгоритмы

```
template<class TL>
struct reverse {
    template<class Tail, class List>
    struct reverse_impl {
        typedef typename
            reverse_impl< cons<typename List::Head, Tail>,
                          typename List::Tail >::value value;
    };

    template<class Tail>
    struct reverse_impl<Tail, nil> {
        typedef Tail value;
    };

    typedef typename reverse_impl<nil, TL>::value value;
};

int main() {
    print<reverse<TypeList>::value>();
}
```

Хранение числовых значений

```
template<int i>
struct Int2Type { static const int value = i; };

template<template<int> class F, int K, int i>
struct generate_impl {
    typedef cons< Int2Type<F<i>::value>,
                  typename generate_impl<F, K, i + 1>::value>
                  value;
};

template<template<int> class F, int K>
struct generate_impl<F, K, K> {
    typedef nil value;
};

template<template<int> class F, int K>
struct generate {
    typedef typename generate_impl<F, K, 0>::value value;
};

int main() {
    print<generate<Fib, 10>::value>();
}
```

Генерация классов

```
template<class L>
struct inherit : L::Head, inherit<typename L::Tail> {};

template<> struct inherit<nil> {};

struct A1 { void f() { std::cout << "class A1\n"; } };
struct B1 { void f() { std::cout << "class B1\n"; } };
struct C1 { void f() { std::cout << "class C1\n"; } };

typedef cons< A1, cons<B1, cons<C1> > >    Bases1;

struct D1 : inherit<Bases1> {
    void f() { f_impl<Bases1>(); }

    template<class List>
    void f_impl() {
        static_cast<typename List::Head *>(this)->f();
        f_impl<typename List::Tail>();
    }
};
template<> inline void D1::f_impl<nil>() {}
```

Curiously recurring template pattern

```
template<class Derived> struct A2 {
    void f() { std::cout << "class A\n"; }
    A2 & a() { return *this; }
};
template<class Derived> struct B2 {
    void f() { std::cout << "class B\n"; }
    B2 & b() { return *this; }
};
template<class Derived> struct C2 {
    void g() {
        self().a().f();
        self().b().f();
    }
    Derived & self() { return *static_cast<Derived *>(this); }
};
template<class L, class D> struct inherit_crtp;
template<class D>          struct inherit_crtp<nil, D> {};

template<template<class> class C, class T, class D>
struct inherit_crtp<cons<C<nil>, T>, D>
    : C<D>, inherit_crtp<T, D> {};

typedef cons< A2<nil>, cons<B2<nil>, cons<C2<nil> > > > Bases2;
struct D2 : inherit_crtp<Bases2, D2> {};
```

CRTP: Polymorphic copy construction

```
// Base class has a pure virtual function for cloning
struct Shape {
    virtual ~Shape() {}
    virtual Shape *clone() const = 0;
};

// This CRTP class implements clone() for Derived
template <typename Derived>
struct Shape_CRTP : public Shape {
    virtual Shape *clone() const {
        return new Derived(static_cast<Derived const&>(*this));
    }
};

// Nice macro which ensures correct CRTP usage
#define Derive_Shape_CRTP(Type) struct Type: Shape_CRTP<Type>

// Every derived class inherits from Shape_CRTP instead of Shape
Derive_Shape_CRTP(Square) {};
Derive_Shape_CRTP(Circle) {};
```

Как определить наличие метода?

```
struct A3 { void f() { std::cout << "class A\n"; } };
struct B3 { /* void f() { std::cout << "class B\n"; } */};
struct C3 { void f() { std::cout << "class C\n"; } };

typedef cons< A3, cons<B3, cons<C3> > >    Bases3;

struct D3 : inherit<Bases3> {
    void f()
    {
        f_impl<Bases1>();
    }

    template<class List>
    void f_impl()
    {
        // в этой строчке всё ломается
        static_cast<typename List::Head *>(this)->f();
        f_impl<typename List::Tail>();
    }
};

template<>
inline void D3::f_impl<nil>()
{}
```

Как проверить наличие родственных связей?

```
typedef char YES;
struct NO { YES m[2]; };

template< class D, class B >
struct is_derived_from
{
    static YES test(B * );
    static NO  test(...);

    static bool const value =
        sizeof(test((D *)0)) == sizeof(YES);
};

template< class B >
struct is_derived_from<B, B> {
    static bool const value = false;
};

int main()
{
    std::cout << is_derived_from<B1, A1>::value << std::endl;
}
```

Используем SFINAE

SFINAE = Substitution Failure Is Not An Error.
Ошибка при подстановке шаблонных параметров не является сама по себе ошибкой.

```
template<class T>
struct f_defined
{
    template<class Z, void (Z::*)() = &Z::f>
        struct wrapper {};

    template<class C>
    static YES check(wrapper<C> * p);

    template<class C>
    static NO  check(...);

    static bool const value = sizeof(check<T>(0)) == sizeof(YES);
};

template<bool b> struct Bool2Type  { typedef YES value; };
template<> struct Bool2Type<false> { typedef NO value; };
```

Используем эту информацию

```
struct D4 : inherit<Bases3> {
    void f() {
        f_impl<Bases3>();
    }

    template<class List>
    void f_impl() {
        call_f<typename List::Head>(
          typename Bool2Type<f_defined<typename List::Head>::value>
                                                        ::value());
        f_impl<typename List::Tail>();
    }

    template<class T>
    void call_f(YES) {
        static_cast<T *>(this)->f();
    }

    template<class T>
    void call_f(NO) {
    }
};

template<> inline void D4::f_impl<nil>() {}
```

enable_if

```
template<bool B, class T = void> struct enable_if {};

template<class T>
struct enable_if<true, T> { typedef T type; };

template<class T>
typename std::enable_if<std::is_floating_point<T>::value, T>::type
    foo1(T t) { return t * 2; }

template<class T>
typename std::enable_if<std::is_integral<T>::value, T>::type
    foo1(T t) { return t / 2; }

int main() {
    foo1(1.2);
    foo1(10);
}
```

enable_if

```
template<class T>
T foo2(T t,  typename
             std::enable_if<std::is_integral<T>::value>::type* = 0)
{ return t + 2; }

template<class T, class Enable = typename
                  std::enable_if<std::is_integral<T>::value>::type >
T foo3(T t) { return t - 2; }

template<class T, class Enable = void>
class A;

template<class T>
class A<T, typename
           std::enable_if<std::is_floating_point<T>::value>::type> {};

int main() {
    foo2(0.1);
    foo2(7);
    A<int> a1;
    A<double> a1;
}
```

MPL

- Meta-programming library является частью библиотеки `boost`.
- Создатели: David Abrahams и Aleksey Gurtovoy.
- Реализует аналог `STL` для метапрограммирования (контейнеры, итераторы, алгоритмы, ...)
- Реализует поддержку `compile-time` лямбда-выражений.