Лекция 6.
To SFINAE or not to SFINAE ©

ITMO+CSC

Выбор кода в зависимости от типа

- Самый простой вариант – перегрузка функции

```
1.
2.  struct currency_value
3.  {
4.      string name;
5.      double value;
6.  };
7.
8.  // by overloading
9.  ostream& operator<<(ostream&, int)
10. { /*...*/ }
11.
12. ostream& operator<<(ostream& os, currency_value v)
13. { os << "(" << v.name << "; " << v.value << ")"; }
```

Выбор кода в зависимости от типа

- Выбор класса по типу используемому типу можно осуществить специализацией шаблона

```
1.  // by specialization
2.  template<class T, class A>
3.  struct my_array
4.  { /*...*/ };
5.
6.  template<class A>
7.  struct my_array<bool, A>
8.  { /*...*/ };
```

Выбор типа по условию

- Как осуществить выбор типа по заданному условию?

```
1.  template<class T>
2.  struct sequence
3.  {
4.      vector<T> data_;
5.  };
6.
7.  template<class T, bool is_polymorph>
8.  struct sequence
9.  {
10.     typedef /* ??? */ value_type;
11.     vector<value_type> data_;
12. };
```

Выбор типа по условию

- Простое решение – специализация класса.
- Плохо масштабируется – придется
  - выделять общую базу,
  - для каждого подобного использования делать специализацию

```
1.  template<class T, bool is_polymorph>
2.  struct sequence
3.  {
4.      typedef T* value_type;
5.      vector<value_type> data_;
6.  };
7.
8.  template<class T>
9.  struct sequence<T, false>
10. {
11.     typedef T value_type;
12.     vector<value_type> data_;
13. };
```

Выбор типа по условию

- Можно сделать специальную метафункцию, предоставляющую выбор типа по условию

```
1.  template<bool take_first, class T1, class T2>
2.  struct select
3.  { typedef T1 type; };
4.
5.  template<class T1, class T2>
6.  struct select<false, T1, T2>
7.  { typedef T2 type; };
8.
9.  // result
10. template<class T, bool polymorph>
11. struct sequence
12. {
13.     typedef
14.         typename select<polymorph, T*, T>::type
15.         value_type;
16.     vector<value_type> data_;
17. };
```

Проверка приводимости

- Хотим проверить, является ли B - открытым базовым классом для D?
- Попробуем это сделать через перегрузку функции.
- Обратите внимание - нет определений функций.

```
1.  typedef char yes;
2.  typedef struct { yes dummy [2]; } no;
3.
4.  yes check(B);
5.  no  check(...);
6.
7.  // why not this way?
8.  template<class T>
9.  no check(T);
10.
11. const bool conv_exist = sizeof(check(D())) == sizeof(yes);
```

Проверка на базовый класс

- Оформим в виде класса или макроса.

```
1.  template<class Dptr, class Bptr>
2.  class converts_to
3.  {
4.      typedef char yes;
5.      typedef struct { yes dm[2]; }no;
6.
7.      static yes check(Bptr);
8.      static no  check(...);
9.
10.     static Dptr makeDptr();
11.
12. public:
13. // by agreement used in STL
14.     enum {value = sizeof(check(makeDptr())) == sizeof(yes) };
15. };
16.
17. #define BASE_N_DERIVED(B, D)                          \
18.     converts_to <const D*, const B*>::value &&    \
19.     !is_same<const B*, const void*>::value;
```

SFINAE

- **SFINAE** - **S**ubstitutaion **F**ailure **I**s **N**ot **A**n **E**rror
- При определении перегрузки функции ошибочное инстанцирование шаблонов не вызывают ошибку компиляции, а выбрасывает функцию из списка кандидатов на наиболее подходящую перегрузку.
- SFINAE работает только с перегрузкой функций.
- Перегрузки могут отбрасываться в том случае, когда их невозможно инстанцировать из-за возникающей синтаксической ошибки - компиляция при этом продолжается без ошибок (если , факт наличия ошибки зависит от параметра этого шаблона).
- Не забывайте предоставить альтернативу.
- SFINAE рассматривает только заголовок функции, ошибки в теле функции будут пропущены.

Проверка на контейнер

- Проверим наличие методов begin и end.
- Работает для vector и list, но не для set или map. Почему?

```
1.  template<class T>
2.  struct has_begin_end
3.  {
4.      typedef char yes;
5.      typedef struct { yes dm[2]; } no;
6.
7.      template<
8.          class U,
9.          typename U::iterator (U::*)(),
10.         typename U::iterator (U::*)()>
11.     struct checker{};
12.
13.     template<class U>
14.     static yes check(checker<U, &U::begin, &U::end>*);
15.     template<class U>
16.     static no  check(...);
17.
18. public:
19.     enum { value = sizeof(check<T>(0)) == sizeof(yes) };
20. };
```

Тут у нас по плану небольшой урок магии...

Проверка на контейнер

- Можно проверить даже на наличие функции, полученной от базового класса. Для этого воспользуемся decltype.

```
1.  template <class T>
2.  struct has_begin_end
3.  {
4.      typedef char yes;
5.      typedef struct { yes dummy[2]; }no;
6.
7.      template <typename U>
8.      static auto test(U* u)
9.          -> decltype((*u).begin(), (*u).end(), yes());
10.     static no   test(...);
11.
12.     enum { value = (sizeof(yes) == sizeof test((T*)0)) };
13. };
```

Выбор по свойству типа

- В STL и Boost несколько отличаются эти типы.
  `std::enable_if` == `boost::enable_if_c`
- В Boost также есть [`lazy_`] {`enable/disable`}`_if` [`_c`]

```
1.  template <bool Cond, class T = void>
2.  struct enable_if
3.  {
4.      typedef T type;
5.  };
6.
7.  template <class T>
8.  struct enable_if<false, T>
9.  {};
```

Выбор по свойству типа

```
1.  template<class T>
2.  typename enable_if<has_begin_end<T>::value>::type
3.  my_print(T const& cont)
4.  {
5.      for(auto const& item: cont)
6.          cout << item << " ";
7.  }
8.
9.  template<class T>
10. void my_print(
11.     T const& value,
12.     typename enable_if<!has_begin_end<T>::value>::type* = 0)
13. { cout << value; }
```

- Возвращать или передавать?
  - из конструкторов нельзя ничего вернуть
  - операторы часто фиксируются количество переданных им параметров
  - оператор приведения должен возвращать зафиксированный тип
  - в остальных случаях есть выбор!

Вопросы?

Завтра

- New C++ features (C++11/14)
- Bind and function

- Задача на дом: reflection