Лекция

TYPE ERASURE

Content

- SFINAE
- Type erasure
- Any
- Dynamic VS Static polymorphism
- CRTP
- PImpl
- Lazy initialization

SFINAE

Substitution Failure Is Not An Error

```
1.  void f(unsigned i) {
2.      std::cout << "f(int)" << std::endl;
3.  }

4.  template <typename T>
5.  void f(const T& i) {
6.      std::cout << "template f" << std::endl;
7.      return;
8.  }

9.  int main() {
10.     f(3);
11. }
```

SFINAE

Substitution Failure Is Not An Error

```
1.  void f(unsigned i) {
2.      std::cout << "f(int)" << std::endl;
3.  }

4.  template <typename T>
5.  typename T::value_type f(const T& i) {
6.      std::cout << "template f" << std::endl;
7.      return i + 1;
8.  }

9.  int main() {
10.     f(3);
11. }
```

Перегрузка функций

этап 1. максимально полный список кандидатов

```
1.  void function(int, std::vector<int>);
2.  void function(int, int);
3.  void function(double, double);
4.  void function(int, int, char, std::string, std::vector<int>);
5.  void function(std::string);
```

Перегрузка функций

этап 2. функции, потенциально могут быть вызваны с этим набором аргументов

```
1.   void function(int, int);
2.   void function(double, double);
```

Перегрузка функций

этап 3. лучший кандидат (exact match)

```
1.   void function(int, int);
```

Перегрузка функций

если компилятор встречает шаблонную функцию

```
1.  void function(int, std::vector<int>);
2.  void function(int, int);
3.  void function(double, double);
4.  void function(int, int, char, std::string, std::vector<int>);
5.  void function(std::string);

6.  template<typename T>
7.  void function(T, T);
```

Перегрузка функций

этап 1. максимально полный список кандидатов

```
1.  void function(int, std::vector<int>);
2.  void function(int, int);
3.  void function(double, double);
4.  void function(int, int, char, std::string, std::vector<int>);
5.  void function(std::string);

6.  template<typename T>
7.  void function(T, T);
8.
    void function<int>(int, int);
```

Перегрузка функций

этап 2. функции, потенциально могут быть вызваны с этим набором аргументов

```
1.  void function(int, int);
2.  void function(double, double);
```

```
3.  template<typename T>
4.  void function(T, T);
5.
    void function<int>(int, int);
```

SFINAE

этап 3. простая функция, при прочих равных, «сильнее» шаблонной (exact match)

```
1.   void function(int, int);
```

SFINAE

- Действует при перегрузке функции
  - только при составлении ***списка кандидатов*** функций
  - рассматривается исключительно ***сигнатура функции***
  - и ничего больше
  - никак **не учитывается *тело функции***
- Если подстановка аргументов даёт корректную функцию по сигнатуре, функция побеждает в перегрузке,
- потом оказывается, что в теле функции есть какие-то проблемы, с которыми компилятор справится не может, и выдаёт ошибку.
- **SFINAE - это только про сигнатуру функции и выводимые параметры шаблона**
- **возвращаемый тип в функции НЕ ЯВЛЯЕТСЯ частью сигнатуры функции**
- **возвращаемый тип ЯВЛЯЕТСЯ частью сигнатуры шаблона функции**
- Это важная информация,
  она позволяет использовать **std::enabled_if** на возвращаемом типе

SFINAE

Expression

```
1.  // при вызове с int исключить невразумительную ошибку
2.  // "C3312: no callable 'begin' function found for type 'int'"
3.  // включим в шапку шаблона
4.  template <typename T>
5.  void printContainer(T container) {
6.      std::cout << "Values:{ ";
7.      for (auto value : container)
8.          std::cout << value << " ";
9.      std::cout << "}\n";
10. }
```

SFINAE

Expression

```
1.  // при вызове с int исключить невразумительную ошибку
2.  // "C3312: no callable 'begin' function found for type 'int'"
3.  // включим в шапку шаблона
4.  template <typename T>
5.  void printContainer(T container) {
6.      std::cout << "Values:{ ";
7.      for (auto value : container)
8.          std::cout << value << " ";
9.      std::cout << "}\n";
10. }
```

SFINAE

Expression

```
1.  // при вызове с int и пр.
2.  // error "C2672: 'printContainer': no matching overloaded function found"

3.  template <typename T, typename = typename T::iterator>
4.  void printContainer(T container) {
5.      std::cout << "Values:{ ";
6.      for (auto value : container)
7.          std::cout << value << " ";
8.      std::cout << "}\n";
9.  }
```

SFINAE

Expression

```
1.  // при вызове с int и пр.
2.  // error "C2672: 'printContainer': no matching overloaded function found"

3.  template <typename T>
4.  void printContainer(T container, typename T::iterator* = nullptr) {
5.      std::cout << "Values:{ ";
6.      for (auto value : container)
7.          std::cout << value << " ";
8.      std::cout << "}\n";
9.  }
```

сигнатура шаблонной функции

```
1.  шапка шаблона,
2.  параметры функции,
3.  а также возвращаемое значение.
```

SFINAE

Expression

```
1.  // при вызове с int и пр.
2.  // error "C2672: 'printContainer': no matching overloaded function found"

3.  template <typename T>
4.  typename T::iterator printContainer(T container) {
5.      std::cout << "Values:{ ";
6.      for (auto value : container)
7.          std::cout << value << " ";
8.      std::cout << "}\n";
9.  }
```

сигнатура шаблонной функции

```
1.  шапка шаблона,
2.  параметры функции,
3.  а также возвращаемое значение.
```

SFINAE

метафункции

```
1.  template <bool condition, typename T>
2.  struct EnableIf;

3.  template <typename T>
4.  struct EnableIf<true, T> {
5.      using type = T;
6.  };



7.  template <typename T, typename U>
8.  struct IsSame {
9.      static constexpr bool value = false;
10. };

11. template <typename T>
12. struct IsSame<T, T> {
13.     static constexpr bool value = true;
14. };
```

SFINAE

Expression

```
1.  template <typename T>
2.  typename EnableIf<!std::is_fundamental<T>::value, void>::type
3.  print(T container) {
4.      std::cout << "Values:{ ";
5.      for (auto value : container)
6.          std::cout << value << " ";
7.      std::cout << "}\n";
8.  }

9.  template <typename T>
10. typename EnableIf<std::is_fundamental<T>::value, void>::type
11. print(T value) {
12.     std::cout << "Value: " << value << "\n";
13. }
```

Примеры: sizeof и decltype

```
char begin(...);

template <typename T> struct IsForReady {
   static constexpr bool value
       = sizeof(begin(std::declval<T>())) != sizeof(char);
};
```

Для **sizeof** важен только тип результата выражения.
У функции важна только сигнатура
если функция лишь задекларирована, но не определена — не важно

```
template <class T> std::add_rvalue_reference<T>::type declval() noexcept;
```

sizeof(begin(T{})) требует от **T** наличия конструктора по умолчанию

SFINAE

Expression

```
1.  template <typename T>
2.  decltype(begin(std::declval<T>()), end(std::declval<T>()), void())
3.  print(T container) {
4.      std::cout << "Values:{ ";
5.      for (auto value : container)
6.          std::cout << value << " ";
7.      std::cout << "}\n";
8.  }

9.  template <typename T>
10. decltype(std::cout << std::declval<T>(), void())
11. print(T value) {
12.     std::cout << "Value: " << value << "\n";
13. }
```

А есть ли функция?

проверка наличия функции в классе

```
1.  #include <iostream>
2.  #include <algorithm>
3.  #include <vector>
4.  #include <list>

5.  #include "EnableIf.h"
6.  #include "HasFunction.h"

7.  using namespace std;
```

А есть ли функция?

проверка наличия функции в классе

```
1.  template<typename Container>
2.  typename EnableIf<HasFunctionSort<Container>::value, void>::type
3.  sort(Container& container) {
4.      cout << "Calling member sort function\n";
5.      container.sort();
6.  }

7.  template<typename Container>
8.  typename EnableIf<!HasFunctionSort<Container>::value, void>::type
9.  sort(Container& container) {
10.     cout << "Calling std::sort function\n";
11.     ::sort(container.begin(), container.end());
12. }

13. int main() {
14.     vector<int> v;
15.     list<int> l;
16.     sort(v);
17.     sort(l);
18. }
```

А есть ли функция?

проверка наличия функции в классе

```
1.  // C++03
2.  typedef char False_t;
3.  typedef char True_t[2];

4.  template <typename Type>
5.  struct HasFunctionSort {
6.      typedef void (Type::*Signature)(); // ищем метод класса такого типа
7.      // C++03 не допустимы шаблонные параметры со значением по умолчанию для функций
8.      template <typename T, Signature = &T::sort>      // non-type template parameter
9.      class Dummy;                  // присваиваем ему значение по умолчанию &T::sort
10.
11.     template <typename T>
12.     static True_t& test(T*, Dummy<T>* = 0);
13.     static False_t& test(...);

14.     static const bool value =           // у T есть функция, совместима с Signature
15.         sizeof(test(static_cast<Type*>(0))) == sizeof(True_t);
16.     // срабатывает SFINAE и выполняется test(…), возвращая False_t
17. };
```

А есть ли функция?

проверка наличия функции в классе

```
1.  // C++14
2.  #include <type_traits>


3.  template <typename Type>
4.  struct HasFunctionSort {




5.      template <typename T>
6.      static auto test(T&&) -> decltype(std::declval<T>().sort(), std::true_type{});
7.      static std::false_type test(...);

8.      static constexpr bool value =
9.          std::is_same<decltype(test(std::declval<Type>())), std::true_type>::value;

10. };
11. template <typename T>
12. constexpr bool HasFunctionSort_v = HasFunctionSort<T>::value;
```

А есть ли функция?

проверка наличия функции в классе

```
1.  // C++17
2.  #include <type_traits>


3.  template <typename Type , typename = void>
4.  struct HasFunctionSort
5.      : std::false_type
6.  {};




7.  template <typename Type>
8.  struct HasFunctionSort<Type, std::void_t<decltype(std::declval<Type>().sort())>>
9.      : std::true_type
10. {};



11. template <typename T>
12. constexpr bool HasFunctionSort_v = HasFunctionSort<T>::value;
```

А есть ли функция?

метафункция принимает переменное число типов и всегда возвращает один тип

```
1.  // C++14
2.  template <typename...>
3.  struct Int_t {
4.      using type = int;
5.  };

6.  template <typename... Ts>
7.  using Int_t = Int<Ts...>::type;



8.  // C++17
9.  template <typename...>
10. using Int_t = int;
```

А есть ли функция?

проверка наличия функции в классе

```
1.  // C++14
2.  template <typename Type, template <typename> class Concept, typename = void>
3.  struct CheckConcept : std::false_type
4.  {};

5.  template <typename Type, template <typename> class Concept>
6.  struct CheckConcept<Type, Concept, std::void_t<Concept<Type>>> : std::true_type
7.  {};



8.  template <typename T>
9.  using HasMemberSortCooncept_t = decltype(std::declval<T>().sort());

10. template <typename T>
11. constexpr bool HasFunctionSort_v = CheckConcept<T, HasMemberSortCooncept_t>::value;
```

А есть ли функция?

проверка наличия функции в классе

```
1.  #define GENERATE_HAS_FUNCTION(name, functionExpression)    \
2.      template <typename T>                                  \
3.      using HasMember ## name ## Cooncept_t =                \
4.          decltype(std::declval<T>().functionExpression);    \
5.      template <typename T>                                  \
6.      constexpr bool HasFunction ## name ## _v =             \
7.          CheckConcept<T, HasMember ## name ## Cooncept_t>::value;
```

```
8.  GENERATE_HAS_FUNCTION(Sort, sort());
9.  // создать HasFunctionSort_v
```

А есть ли функция?

проверка наличия функции в классе

```
1.  #include <boost/tti/has_member_function.hpp>

2.  BOOST_TTI_HAS_MEMBER_FUNCTION(sort);

3.  template<typename T>
4.  constexpr bool HasFunctionSort_v = boost::has_member_function_sort<T, void>::value;


5.  template<typename Container>
6.  enable_if_t<HasFunctionSort_v<Container>>
7.  sort(Container& container) {
8.      cout << "Calling std::sort function\n";
9.      ::sort(container.begin(), container.end());
10. }

11. template<typename Container>
12. enable_if_t<!HasFunctionSort_v<Container>>
13. sort(Container& container) {
14.     cout << "Calling member sort function\n";
15.     ::sort(begin(container), end(container));
16. }
```

Boost

- Boost.Fusion
- Boost.TTI
- Boost.Hana
- Boost.MPL

Boost.Fusion

в сумеречной зоне программирования между временами компиляции и выполнения

```
1.  #include <boost/fusion/sequence.hpp>
2.  #include <boost/fusion/include/sequence.hpp>
3.  #include <boost/fusion/algorithm.hpp>
4.  #include <boost/fusion/include/algorithm.hpp>

5.  struct print_xml {
6.      template <typename T>
7.      void operator()(T const& x) const {
8.          std::cout << '<'  << typeid(x).name() << '>' << x
9.                    << "</" << typeid(x).name() << '>';
10.     } // result of typeid(x).name() is platform specific
11. };

12. void f() {
13.     vector<int, char, std::string> stuff(1, 'x', "howdy");
14.     int i = at_c<0>(stuff); // like Boost.Tuple
15.     char ch = at_c<1>(stuff);
16.     std::string s = at_c<2>(stuff);
17.     // is a fusion algorithm
18.     for_each(stuff, print_xml{}); // print the vector as XML
19. } // generic, use it to print just about any Fusion Sequence
```

Boost.Fusion

в сумеречной зоне программирования между временами компиляции и выполнения

```
1.  #include <boost/type_traits/is_pointer.hpp>
2.  // a little cleverer, let's write a generic function that takes in
3.  // an arbitrary sequence and XML prints only those which are pointers
4.  template <typename Sequence>
5.  void xml_print_pointers(Sequence const& seq) {
6.      for_each(filter_if<boost::is_pointer<_> >(seq), print_xml{});
7.  }

8.  namespace fields {
9.  struct name;
10. struct age;
11. }
12. // Fusion's map associate types with elements
13. typedef map< fusion::pair<fields::name, std::string>,
14.               fusion::pair<fields::age, int> >          person;
15. // Fusion pair only contain 1 member, the type is the second template parameter
16. void f() {
17.     using namespace fields;
18.     // the first type parameter of the pair is used as an index
19.     std::string person_name = at_key<name>(a_person);
20.     int person_age = at_key<age>(a_person);
21. }
```

Boost.Fusion

в сумеречной зоне программирования между временами компиляции и выполнения

```
1.  // deal with a generic data structure, a multitude of facilities available,
    introspection comes for free
2.  // serialization function (load/save)

3.  struct saver {
4.      template <typename Pair>
5.      void operator()(Pair const& data) const {
6.          some_archive << data.second;
7.      }
8.  };

9.  template <typename Stuff>
10. void save(Stuff const& stuff) {
11.     for_each(stuff, saver{});
12. }
```

Идиома **сокрытие типа** (type erasure)

указатель можно привести его к любому типу

```
1.  int intVar = 0;
2.  void* unknownVar = &intVar;
```

Идиома **сокрытие типа** (type erasure)

существует давно

```
1.  enum class Kind { Unknown, Int, Float, Char };

2.  struct Any {
3.      void* data;
4.      Kind kind;
5.  };
```

Идиома сокрытие типа (type erasure)

существует давно

```
1.  enum class Kind { Unknown, Int, Float, Char };

2.  struct Any {
3.      void* data;
4.      Kind kind;
5.  };
```

что делать с signed/unsigned/const/volatile и пр.?

Идиома **сокрытие типа** (type erasure)

функционал C++ шаблоны и typeid

```
1.  #include <typeindex>
2.  using namespace std;

3.  struct Any {
4.      void* data;
5.      type_index type;
6.  };

7.  int main() {
8.      assert(typeid(float&) == typeid(float)); // fail

9.      long long var = 156;
10.     Any any{ static_cast<void*>(&var), typeid(var) };
11. }
```

Какие есть проблемы с этим кодом?

type erasure

```
1.  #include <typeindex>
2.  class Any {
3.      void* data            = nullptr;
4.      type_index type       = typeid(nullptr_t);

5.  public:
6.      template <typename T>
7.      Any(T& var) :
8.          data{ static_cast<void*>(&var) },
9.          type{ typeid(var) }
10.     {}


11.     template <typename T> /*implicit*/ operator T() const {
12.         if (type != typeid(T))
13.             throw logic_error{ string{"Any doesn't hold type '"} +
                                   typeid(T).name() + "'" };
14.         return *(static_cast<T*>(data));
15.     }
16. };
17. int main() {
18.     long long var = 156;
19.     Any any{ var };
20.     try {
21.         long long n = any;
22.         int i = any;
23.     } catch (std::exception &e) { cout << e.what() << endl; }
24. }
```

type erasure

```
1.  #include <typeindex>
2.  class Any {
3.      void* data             = nullptr;
4.      type_index type        = typeid(nullptr_t);

5.  public:
6.      template <typename T>
7.      Any(T& var) :
8.          data{ new T{var} },
9.          type{ typeid(var) }
10.     {}


11.     template <typename T> /*implicit*/ operator T() const {
12.         if (type != typeid(T))
13.             throw logic_error{ string{"Any doesn't hold type '"} +
                                   typeid(T).name() + "'" };
14.         return *(static_cast<T*>(data));
15.     }
16. };
17. int main() {
18.     long long var = 156;
19.     Any any{ var };
20.     try {
21.         long long n = any;
22.         int i = any;
23.     } catch (std::exception &e) { cout << e.what() << endl; }
24. }
```

type erasure

```
1.  #include <typeindex>
2.  class Any {
3.      shared_ptr<void> data = nullptr;
4.      type_index type       = typeid(nullptr_t);

5.  public:
6.      template <typename T>
7.      Any(T& var) :
8.          data{ make_shared<T>(var) },
9.          type{ typeid(var) }
10.     {}



11.     template <typename T> /*implicit*/ operator T() const {
12.         if (type != typeid(T))
13.             throw logic_error{ string{"Any doesn't hold type '"} +
                                   typeid(T).name() + "'" };
14.         return *(static_cast<T*>(data));
15.     }
16. };
17. int main() {
18.     long long var = 156;
19.     Any any{ var };
20.     try {
21.         long long n = any;
22.         int i = any;
23.     } catch (std::exception &e) { cout << e.what() << endl; }
24. }
```

type erasure

```
1.  #include <typeindex>
2.  class Any {
3.      void* data            = nullptr;
4.      type_index type       = typeid(nullptr_t);
5.      std::function<void(void*)> deleter;
6.  public:
7.      template <typename T>
8.      Any(T& var) :
9.          data{ new T{var} },
10.         type{ typeid(var) } ,
11.         deleter{ [](void* ptr) { delete static_cast<T*>(ptr); } }
12.     {}
13.     ~Any() { deleter(data); }
14.     template <typename T> /*implicit*/ operator T() const {
15.         if (type != typeid(T))
16.             throw logic_error{ string{"Any doesn't hold type '"} +
                                   typeid(T).name() + "'" };
17.         return *(static_cast<T*>(data));
18.     }
19. };
20. int main() {
21.     long long var = 156;
22.     Any any{ var };
23.     try {
24.         long long n = any;
25.         int i = any;
26.     } catch (std::exception &e) { cout << e.what() << endl; }
27. }
```

```
1.  #include <typeindex>
2.  class Any {
3.      using Copier    = void*(*)(void*);
4.      using Deleter   = void(*)(void*);
5.      void* data      = nullptr;
6.      type_index type = typeid(nullptr_t);
7.      Deleter deleter = nullptr;
8.      Copier copier   = nullptr;
9.      void flush();
10. public:
11.     Any() = default;
12.     template <typename T> Any(T&& var) {
13.         typedef typename std::remove_reference<T>::type U;
14.         data    = new U{ forward<T>(var) };
15.         type    = typeid(U);
16.         deleter = [](void* ptr) { delete static_cast<U*>(ptr); };
17.         copier  = [](void* ptr) -> void* { return new U{ *static_cast<U*>(ptr) }; };
18.     }
19.     ~Any() { if (valid()) deleter(data); }
20.     Any(Any const& any);                    Any& operator=(Any const& any);
21.     Any(Any&& any);                         Any& operator=(Any&& any);
22.     bool valid() const { return (data != nullptr); }
23.     void clear();
24.     template <typename T> void set(T&& v) { *this = Any{ v }; }
25.     template <typename T> static T cast(const Any &any) {
26.         if (any.type != typeid(T))
27.             throw logic_error{string{"Any doesn't hold type '"}+typeid(T).name()+"'"};
28.         return *(static_cast<T*>(any.data));
29.     }
30.     static void swap(Any &a, Any &b);
31. }; // end class Any
```

type erasure

```
1.  Any::Any(Any const& any) :
2.      data{ any.copier(any.data) },
3.      type{ any.type },
4.      deleter{ any.deleter },
5.      copier{ any.copier }
6.  {}
7.  Any::Any(Any&& any) { swap(*this, any); any.flush(); }
8.  Any& Any::operator=(Any const& any) {
9.      if (this != &any) { Any tmp(any); swap(*this, tmp); }
10.     return *this;
11. }
12. Any& Any::operator=(Any&& any) {
13.     if (this != &any) { swap(*this, any); any.clear(); }
14.     return *this;
15. }
16. void Any::clear() { if (valid()) deleter(data); flush(); }
17. void Any::flush() { data = deleter = copier = nullptr; type = typeid(nullptr_t); }

18. void Any::swap(Any &a, Any &b) {
19.     using std::swap;
20.     swap(a.data, b.data);
21.     swap(a.type, b.type);
22.     swap(a.deleter, b.deleter);
23.     swap(a.copier, b.copier);
24. }
```

type erasure

пробуем - работает

```
1.  int main() {

2.      long long var = 156;
3.      Any any{ var };

4.      try {
5.          long long n = Any::cast<long long>(any);
6.          int i = Any::cast<int>(any);
7.      } catch (const std::exception &e) {
8.          cout << e.what() << endl;
9.      }

10.     any.set(33);

11.     try {
12.         int i = Any::cast<int>(any);
13.         long long n = Any::cast<long long>(any);
14.     } catch (const std::exception &e) {
15.         cout << e.what() << endl;
16.     }

17.     return 0;
18. }
```

type erasure

элегантное сокрытие

```
1.  namespace detail {
2.  struct IAny {
3.      virtual void* get() = 0;
4.      virtual std::unique_ptr<IAny> clone() const = 0;
5.      virtual ~IAny() = default;
6.  };

7.  template <typename T>
8.  class AnyImpl : public IAny {
9.      T data;
10. public:
11.     AnyImpl(const T& value) : data{ value } {}
12.     void* get() override { return &data; }
13.     std::unique_ptr<IAny> clone() const override {
14.         return std::make_unique<AnyImpl<T>>(data);
15.     }
16. };
17. }
```

type erasure

```
1.  class Any {
2.      unique_ptr<detail::IAny> impl;
3.  public:
4.      template < typename T,
5.                 typename = enable_if_t<!is_same_v<decay_t<T>, Any>> >
6.          Any(T&& var) :
7.          impl{ make_unique<detail::AnyImpl<decay_t<T>>>(forward<T>(var)) }
8.      {}

9.      Any(const Any& any) : impl{ any.impl->clone() } {}
10.     Any& operator=(const Any& any) {
11.         if (this != &any) { Any tmp(any); swap(*this, tmp); }
12.         return *this;
13.     }
14.     Any(Any&&) = default;
15.     Any& operator=(Any&&) = default;

16.     template <typename T>
17.     static T cast(const Any& any) {
18.         if (dynamic_cast<detail::AnyImpl<T>*>(any.impl.get()))
19.             return *(static_cast<T*>(any.impl->get()));
20.         throw logic_error{string{"Any doesn't hold type '"}+typeid(T).name()+"'"};
21.     }
22.     static void swap(Any& a, Any& b) { std::swap(a.impl, b.impl); }
23. };
```

Function

для простоты, возвращаемое значение входит в понятие сигнатуры функции

```
1.  namespace detail {
2.  template <typename R, typename... Args>
3.  struct IFunction {
4.      virtual R operator()(Args... args) const = 0;
5.      virtual unique_ptr<IFunction> clone() const = 0;
6.      virtual ~IFunction() = default;
7.  };


8.  template <typename Functor, typename R, typename... Args>
9.  class FunctionImpl : public IFunction<R, Args...> {
10.     Functor functor;
11. public:
12.     FunctionImpl(const Functor& functor) : functor{ functor } {}
13.     FunctionImpl(Functor&& functor) : functor{ move(functor) } {}

14.     R operator()(Args... args) const override
15.     { return functor(args...); }

16.     unique_ptr<IFunction<R, Args...>> clone() const override {
17.         return make_unique<FunctionImpl>(functor);
18.     }
19. };
20. }
```

Function

```
1.  template<typename> class Function;

2.  template<typename R, typename... Args> class Function<R(Args...)> final {
3.      unique_ptr<detail::IFunction<R, Args...>> functor;
4.  public:
5.      Function() = default;
6.      template < typename F,
7.                 typename = typename enable_if<
8.                     is_convertible<typename result_of<F(Args...)>::type, R>::value,
9.                     void>::type,
10.                typename = typename enable_if<
11.                    !is_same<typename decay<F>::type, Function>::value,
12.                    void>::type
13.               >
14.     Function(Functor&& f) :
15.         functor{ make_unique<detail::FunctionImpl<F,R,Args...>>(forward<Functor>(f)) }
16.     {}
17.     Function(const Function& f) : functor{ f.functor->clone() } {}
18.     Function(Function&&) = default;
19.     Function& operator=(const Function& f) {
20.         if (this != &f) { auto tmp(f); swap(*this, tmp); } return *this;
21.     }
22.     Function& operator=(Function&&) = default;
23.     R operator()(Args... args) const { return (*functor)(args...); }
24.     friend void swap(Function& a, Function& b) { std::swap(a.functor, b.functor); }
25. };
```

Function

проверяем...

```
1.  int main() {
2.      Function<void()> def{};
3.      Function<void()> func([] { cout << "It lives!\n"; });
4.      func();

5.      Function<long(int)> square([](int i) { return i * i; });
6.      auto square2 = std::move(square);
7.      auto square3 = square2;

8.      cout << "2^2=" << square2(2) << "\n";
9.      cout << "3^2=" << square3(3) << "\n";

10.     Function<void(int)> func1{ [](int) { return 0; } }; // error

11.     Function<size_t(const string&)> func2{ string::size(); }; // error
12. }
```

Function

весь ли функционал std::function покрыли в этом классе?

```
1.  namespace detail {
2.  template <typename R, typename... Args>
3.  struct IFunction {
4.      virtual R operator()(Args... args) const = 0;
5.      virtual unique_ptr<IFunction> clone() const = 0;
6.      virtual ~IFunction() = default;
7.  };


8.  template <typename Functor, typename R, typename... Args>
9.  class FunctionImpl : public IFunction<R, Args...> {
10.     Functor functor;
11. public:
12.     FunctionImpl(const Functor& functor) : functor{ functor } {}
13.     FunctionImpl(Functor&& functor) : functor{ move(functor) } {}

14.     R operator()(Args... args) const override
15.     { return std::invoke(functor, args...); }

16.     unique_ptr<IFunction<R, Args...>> clone() const override {
17.         return make_unique<FunctionImpl>(functor);
18.     }
19. };
20. }
```

Boost.TypeErasure

several special cases of polymorphism:

- **boost::any** for CopyConstructible types
- **boost::function** for callable objects
- **Boost.Range** provides **any_iterator**

Boost.TypeErasure

что-то особенное…

```
1.  any < mpl::vector < copy_constructible<>,
2.                      typeid_<>,
3.                      incrementable<>,
4.                      ostreamable<>
5.                    >
6.      > x(10);
7.  ++x;
8.  std::cout << x << std::endl; // prints 11

9.  BOOST_TYPE_ERASURE_MEMBER(push_back)
10. void append_many(any<has_push_back<void(int)>, _self&> container)
11. { for (int i = 0; i < 10; ++i) container.push_back(i); }

12. BOOST_TYPE_ERASURE_MEMBER(empty)
13. bool is_empty(any<has_empty<bool() const>, const _self&> x) { return x.empty(); }

14. BOOST_TYPE_ERASURE_FREE(getline)
15. std::vector<std::string> readlines(any<has_getline<bool(_self&, std::string&)>, _self&> stream)
16. {
17.     std::vector<std::string> result;
18.     std::string tmp;
19.     while (getline(stream, tmp)) result.push_back(tmp);
20.     return result;
21. }

22. void read_line(any<has_getline<bool(std::istream&, _self&)>, _self&> str)
23. { getline(std::cin, str); }
```

Полиморфизм: динамический vs статический

ООП концепции:

- *классы*,
- *инкапсуляция* – открытый интерфейс и закрытая реализация,
- *наследование*,
- *динамический полиморфизм* – виртуальные функции (не бесплатны во время выполнения с явным интерфейсом)

Шаблоны

- *статический полиморфизм* (сложность времни компиляции с неявными интерфейсами)

Полиморфизм: динамический vs статический

функция process принимает объект, реализующий интерфейс класса Base

```
1.  struct Base {
2.      void prepare();
3.      void work();
4.  };
5.  void process(Base *b) {
6.      b->prepare();
7.      b->work();
8.      // ...
9.  }
```

```
10. template <typename T>
11. void process(T&& t) {
12.     t.prepare();
13.     t.work();
14.     // ...
15. }
```

VIRTUAL FUNCTIONS	TEMPLATES
runtime	compile time
separate compilation	body of a template has to be available in every translation unit in which it is used
automatically make the requirements on the arguments explicit	only checked when they're instantiated
one function description	new copy for function description of each template function every time it is instantiated
have to use (smart) pointers or references	support value semantics
have to create a wrapper that inherits from the base class	allow other types to be adapted non-intrusively for seamless interoperability
aren't really able to express such constraints	involving multiple types

Идиома curiously recurring template pattern (CRTP)

- Класс унаследован от шаблонного класса, в котором наследник – аргумент шаблона
- Jan Falkin, James O. Coplien (1995)
- ***upside-down inheritance*** («перевёрнутое наследование»)

```
1.  template <typename Derived>
2.  class Base {
3.      // ...
4.      void foo() { static_cast<Derived*>(this)->bar(); }
5.      static void staticBase() {
6.          Derived::staticDerived();
7.          // ...
8.       }
9.  };
10. class Derived : public Base<Derived> {
11.     // ...
12.     void bar();
13.     static void staticDerived();
14. };
```

CRTP. Static polymorphism

явный интерфейс

```
1.  template <typename D>
2.  struct base_worker {
3.      void work() { static_cast<D*>(this)->work_impl(); }
4.      void prepare() { static_cast<D*>(this)->prepare_impl(); }
5.  };

6.  struct some_concrete_worker : base_worker<some_concrete_worker> {
7.      void work_impl();    // Без этих функций вызывающий
8.      void prepare_impl(); // код не скомпилируется
9.  };
10. // ...

11. template<typename Worker>
12. void polymorhic_work(const Worker& w) {
13.     w.prepare();
14.     w.work();
15. };

16. int main() {
17.     some_concrete_worker w1;
18.     some_concrete_worker2 w2;
19.     polymorhic_work(w1); // Скомпилируется только при наличии
20.     polymorhic_work(w2); // функций prepare_impl() и work_impl() в w1 и w2
21. }
```

CRTP

дополнительная функциональность

```
1.  // base class has a pure virtual function for cloning
2.  class Shape {
3.  public:
4.      virtual ~Shape() {};
5.      virtual Shape *clone() const = 0;
6.  };
7.  // CRTP class implements clone() for Derived
8.  template <typename Derived>
9.  class Shape_CRTP : public Shape {
10. public:
11.     virtual Shape *clone() const     {
12.         return new Derived(static_cast<Derived const&>(*this));
13.     }
14. };

15. // macro which ensures correct CRTP usage
16. #define Derive_Shape_CRTP(Type) class Type: public Shape_CRTP<Type>

17. void main() {
18.     // every derived class inherits from Shape_CRTP instead of Shape
19.     Derive_Shape_CRTP(Square) {};
20.     Derive_Shape_CRTP(Circle) {};
21. }
```

MixIn

приём проектирования, класс реализует функциональность, можно внести в другой

```
1.  template<typename D>
2.  struct singleton { ... };

3.  class my_class : public singleton<my_class> { /*...*/ };
```

```
4.  struct singleton { /*...*/ };
5.  class my_class : singleton { /*...*/ };
```

CRTP

ограничиваем число объектов класса

```
1.  #include <stdexcept>
2.  template <typename T, size_t maxN> class LimitedInstances {
3.      static size_t counter;
4.  protected:
5.      LimitedInstances()
6.      { if (counter >= maxN) throw logic_error{"too many instances"}; ++counter; }
7.      ~LimitedInstances()
8.      { --counter; }
9.  };
10. template <typename T, size_t maxN> size_t LimitedInstances<T, maxN>::counter{0};

11. class oneInst : public LimitedInstances<oneInst, 1> {};
12. class twoInst : public LimitedInstances<twoInst, 2> {};

13. void main() {
14.     oneInst obj;
15.     try {
16.         oneInst{};
17.     } catch (logic_error &e) { cerr << "Caught: " << e.what() << endl; }
18.     twoInst obj1, obj2;
19.     try {
20.         twoInst{};
21.     } catch (logic_error &e) { cerr << "Caught: " << e.what() << endl; }
22. };
```

enable_shared_from_this

позволяет базовому классу “увидеть” тип производного и вернуть указатель на него

```
1.  struct bad {
2.      std::shared_ptr<bad> get() {
3.          return std::shared_ptr<bad>(this);
4.      }
5.  };

6.  struct good : std::enable_shared_from_this<good> {
7.      std::shared_ptr<good> get() {
8.          return shared_from_this(); // наследуется из enable_shared_from_this
9.      }
10. };


11. template<typename T>
12. struct enable_shared {
13.     weak_ptr<T> t_;
14.     enable_shared() { t_ = weak_ptr<T>(static_cast<T*>(this)); }
15.     shared_ptr<T> shared_from_this() {
16.         return shared_ptr<T>(t_);
17.     }
18. };
```

MixIn

non_equalable<D> может воспользоваться оператором == типа D

```
1.  template<typename D>
2.  struct non_equalable {};

3.  template<typename D>
4.  bool operator!=(const non_equalable<D>& lhs, const non_equalable<D>& rhs) {
5.      return !(static_cast<const D&>(lhs) == static_cast<const D&>(rhs));
6.  }

7.  struct some_struct : non_equalable<some_struct> {
8.      some_struct(int w) : i_(w) {}
9.      int i_;
10. };

11. bool operator==(const some_struct& lhs, const some_struct& rhs) {
12.     return lhs.i_ == rhs.i_;
13. }

14. int main() {
15.     non_equalable<some_struct>* s1 = new some_struct(3);
16.     non_equalable<some_struct>* s2 = new some_struct(4);
17.     std::cout << (s1 != s2) << std::endl;
18. }
```

MixIn наоборот

реализация паттерна шаблонный метод

```
1.  class space_ship {
2.      virtual bool fuel() const = 0;
3.      virtual int speed() const = 0;
4.  public:
5.      // ...
6.      void move()     {
7.          if (!fuel()) return;
8.          int current_speed = speed();
9.          // further actions ...
10.     }
11.     virtual ~space_ship() {}
12. };
13. class interceptor : public space_ship {
14.     bool fuel() const { /* ... */ }
15.     int speed() const { /* ... */ }
16. public:
17.     // ...
18. };
19. class other_ship : public space_ship { /* ... */ };
20. class other_ship_2 : public space_ship { /* ... */ };
21. // ...
```

MixIn наоборот

CRTP

```
1.  template<typename D>
2.  class space_ship {
3.  public:
4.      void move()     {
5.          if (!static_cast<D*>(this)->fuel())
6.              return;
7.          int current_speed = static_cast<D*>(this)->speed();
8.          // ...
9.      }
10. };

11. class interceptor : public space_ship<interceptor> {
12. public:
13.     bool fuel() const;
14.     int speed() const;
15. };
```

MixIn вариации

```
1.  struct access {
2.      template<typename Impl>
3.      static void on_handle_connect(Impl* impl) { impl->handle_connect(); }
4.      template<typename Impl>
5.      static void on_handle_response(Impl* impl) { impl->handle_response(); }
6.  };

7.  template<typename D> struct connection_handler {
8.      // ...
9.      void on_connection() { access::on_handle_connect(static_cast<D*>(this)); }
10. };
11. template<typename D> struct response_handler {
12.     // ...
13.     void on_response() { access::on_handle_response(static_cast<D*>(this)); }
14. };

15. class combined_handler :
16.     public connection_handler<worker>,
17.     public response_handler<worker> {
18. private:
19.     friend struct access;
20.     void handle_connect() { std::cout << __PRETTY_FUNCTION__ << std::endl; }
21.     void handle_response() { std::cout << __PRETTY_FUNCTION__ << std::endl; }
22. };
```

Идиома Pointer to implementation (Pimpl)

поля данных класса заменяются указателями на эти данные

```
1.  #include "myitems.h"
2.  class A {

3.      TMyItem item1, item2;
4.  public:
5.      A();
6.      // ...
7.  };
```

Идиома Pointer to implementation (Pimpl)

поля данных класса заменяются указателями на эти данные

a.h:

```
1.  class A {
2.      struct Impl;
3.      Impl *pImpl;
4.  public:
5.      A();
6.      // ...
7.  };
```

a.cpp:

```
1.  #include "a.h"
2.  #include "myitems.h"
3.  struct A::Impl { TMyItem item1, item2; };
4.  A::A() : pImpl(new Impl) {}
5.  A::~A() { delete pImpl; }
```

Идиома **Pointer to implementation** (Pimpl) C++11

Break compilation dependencies!

a.h:

```
1.  class A {
2.      struct Impl;
3.      std::unique_ptr<Impl> pImpl;
4.  public:
5.      A();
6.      ~A();
7.      A(A&& other);          A& operator=(A&& other);
8.      A(const A& other);     A& operator=(const A& other);
9.      // ...
10. };
```

a.cpp:

```
1.  #include "a.h"
2.  #include "myitems.h"
3.  struct A::Impl { TMyItem item1, item2; };
4.  A::A() : pImpl(std::make_unique<A::Impl>()) {}
5.  A::~A() {}
6.  A::A(A&& other) = default;
7.  A& A::operator=(A&& other) = default;
```

Отложенная инициализация

lazy initialization

```
1.  string receiveDescription(); // long calculation function
2.  class Fruit {
3.  public:
4.      enum Kind : int8_t { Apple, Banana, Pineapple, Orange, _LAST_ };
5.      using Items = vector<std::shared_ptr<Fruit>>;
6.      static const Fruit* getFruit(const Kind kind) {
7.          if (!items[kind])
8.              items[kind] = make_shared<Fruit>(receiveDescription()); // lazy initialization
9.          return items[kind].get();
10.     }
11.     static void printCurrentTypes() {
12.         cout << "Fruits kinds in box=" << items.size() << ". Kinds are:" << endl;
13.         for (const auto &fruit : items) { cout << fruit->description << endl; } cout << endl;
14.     }
15. private:
16.     string description{};                       // resource
17.     Fruit(const string &s) : description(s) {}  // force to use static getFruit()
18.     static Items items;
19. };
20. Fruit::Items Fruit::items(Fruit::_LAST_, {});
21. void main() {
22.     Fruit::getFruit(Fruit::Banana);
23.     Fruit::getFruit(Fruit::Apple);
24.     Fruit::getFruit(Fruit::Banana);             // returns pre-existing instance from first
25.     Fruit::printCurrentTypes();
26. }
```

СПАСИБО ЗА ВНИМАНИЕ!