

Лекция 1. Введение

Метапрограммирование в C++

Список литературы

Введение





Кёниг Э., Му Б. — Эффективное программирование на C++:
Практическое программирование на примерах. — пер. с англ. — М. :
ООО “И. Д. Вильямс”, 2015. — 368 с. — (C++ In-Depth). — ISBN
978-5-8459-0350-5.



Страуструп Б. — Программирование: Принципы и практика
использования C++. — пер. с англ. — М. : ООО “И. Д. Вильямс”,
1248. — 1248 с. — ISBN 978-5-8459-1621-1.

Список литературы (продолжение)

Шаблоны

-  *Вандевурд Д., Джосаттис Н. М.* — Шаблоны C++: Справочник разработчика. — пер. с англ. — М. : ООО “И. Д. Вильямс”, 2016. — 544 с. — ISBN 978-5-8459-0513-3.
-  *Джосаттис Н. М.* — Стандартная библиотека C++: справочное руководство. — пер. с англ. — 2-е изд. — М. : ООО “И. Д. Вильямс”, 2014. — 1136 с. — (Библиотека ALT Linux). — ISBN 978-5-8459-1837-6.

Основные определения

Определения

Метапрограмма: (*metaprogram*) — программа, генерирующая или изменяющая другие программы (например, саму себя).

Метаязык: (*metalanguage, domain language*) — язык для описания языков (на котором написана метапрограмма).

Язык-объект: (*object language, host language*) — язык, являющийся объектом изучения некоторой дисциплины (программу на котором генерирует метапрограмма).

Методы реализации метапрограммирования

Время работы метапрограммы

- На этапе компиляции.
- На этапе выполнения программы.

Основные методы

- Самомодифицирующийся код.
 - Интерпретация произвольного кода.
 - Интроспекция.
- Генерация кода.
 - Компиляция.
 - Конвертация.
 - Генерация исходных текстов.
 - Шаблоны.

Методы реализации метапрограммирования

Время работы метапрограммы

- На этапе компиляции.
- На этапе выполнения программы.

Основные методы

- Самомодифицирующийся код.
 - Интерпретация произвольного кода.
 - Интроспекция.
- Генерация кода.
 - Компиляция.
 - Конвертация.
 - Генерация исходных текстов.
 - Шаблоны.

Основные определения

Определения

Шаблон: конструкция языка, определяющая семейство классов или функций. Определяет образец, по которому реальные классы или функции могут быть сгенерированы путём **конкретизации**.

Конкретизация шаблона: процесс создания обычного класса, функции или функции-члена класса из шаблона подстановкой вместо его параметров их фактических значений.

Точка конкретизации: место в тексте программы, где происходит конкретизация шаблона (или члена шаблонного класса).

Специализация шаблона: результирующий класс, функция или функция-член, получающиеся подстановкой шаблонных параметров конкретными значениями. Может произойти в результате либо **конкретизации**, либо **явной специализации**.

Основные определения

Определения

Шаблон: конструкция языка, определяющая семейство классов или функций. Определяет образец, по которому реальные классы или функции могут быть сгенерированы путём **конкретизации**.

Конкретизация шаблона: процесс создания обычного класса, функции или функции-члена класса из шаблона подстановкой вместо его параметров их фактических значений.

Точка конкретизации: место в тексте программы, где происходит конкретизация шаблона (или члена шаблонного класса).

Специализация шаблона: результирующий класс, функция или функция-член, получающиеся подстановкой шаблонных параметров конкретными значениями. Может произойти в результате либо **конкретизации**, либо **явной специализации**.

Основные определения

Определения

Шаблон: конструкция языка, определяющая семейство классов или функций. Определяет образец, по которому реальные классы или функции могут быть сгенерированы путём **конкретизации**.

Конкретизация шаблона: процесс создания обычного класса, функции или функции-члена класса из шаблона подстановкой вместо его параметров их фактических значений.

Точка конкретизации: место в тексте программы, где происходит конкретизация шаблона (или члена шаблонного класса).

Специализация шаблона: результирующий класс, функция или функция-член, получающиеся подстановкой шаблонных параметров конкретными значениями. Может произойти в результате либо **конкретизации**, либо **явной специализации**.

Основные определения

Определения

Шаблон: конструкция языка, определяющая семейство классов или функций. Определяет образец, по которому реальные классы или функции могут быть сгенерированы путём **конкретизации**.

Конкретизация шаблона: процесс создания обычного класса, функции или функции-члена класса из шаблона подстановкой вместо его параметров их фактических значений.

Точка конкретизации: место в тексте программы, где происходит конкретизация шаблона (или члена шаблонного класса).

Специализация шаблона: результирующий класс, функция или функция-член, получающиеся подстановкой шаблонных параметров конкретными значениями. Может произойти в результате либо **конкретизации**, либо **явной специализации**.

Примеры конкретизации шаблонов

Пример (templates.h)

```
template <typename T>
    void Inc(T &rT)
{
    ++ rT;
}

struct C
{
    void operator ++ ();
};

struct E { /* Пусто */};
```

Пример (main.cpp)

```
int main()
{
    int n = 1;
    Inc(n);    // ++ n;
    float f = 1.5f;
    Inc(f);    // ++ f;
    C c;
    Inc(c);    // c.operator ++ ();
    E e;
    Inc(e);    // ошибка
}
```

Функция

Вид шаблона

Функция: объявление или определение.

Пример (templates.h)

```
template <typename T> T max(T t1, T t2)
{
    return (t1 > t2 ? t1 : t2);
}
```

Функция — использование

Пример (main.cpp)

```
int main()
{
    int n1, n2;
    // ...
    int nMax = max(n1, n2);    // или max <int>(n1, n2);
    // ...
}
```

Класс

Вид шаблона

Класс: объявление или определение.

Пример (templates.h)

```
template <typename T> struct test
{
    T f();                // объявление метода
    struct inner;         // объявление структуры
    static T ms_t;        // объявление статического поля
    template <class U> T get(U u); // объявление шаблона
};
```

Класс — использование

Пример (main.cpp)

```
int main()
{
    test <float> tf;
    cout << tf.f() << endl;
    // ...
}
```

Метод шаблонного класса

Вид шаблона

Метод шаблонного класса: определение.

Пример (templates.h)

```
template <typename T> T test <T>::f()
{
    return T();    // при T == float: float() == 0.0f
}
```


Вложенный класс шаблонного класса

Вид шаблона

Вложенный класс шаблонного класса: определение.

Пример (templates.h)

```
template <typename T>
    struct test <T>::inner
{
    T m_t;    // поле
};
```

Пример (main.cpp)

```
test <char>::inner tic;
cout << tic.m_t << endl;
```

Статическое поле шаблонного класса

Вид шаблона

Статическое поле шаблонного класса: определение.

Пример (templates.h)

```
template <typename T> T test <T>::ms_t = T();
```

Шаблонный член шаблонного класса

Вид шаблона

Шаблонный член шаблонного класса: определение.

Пример (templates.h)

```
template <class T> template <class U> T test <T>::get(U u)
{
    return u;
}
```

Шаблонный член шаблонного класса — использование

Пример (main.cpp)

```
test <float> tf;  
// ...  
cout << tf.get(1.0) << endl;    // или tf.get <double>(1.0)
```

Параметры шаблонов

Виды параметров шаблонов

- Параметры-типы.
- Параметры-не-типы.
- Параметры-шаблоны.

Пример (параметры-не-типы)

```
template <int N> struct Array
{
    char m_szData[N];
};
// ...
Array <10> a10;
```

Значения по умолчанию

Пример (значения по умолчанию параметров шаблонов)

```
template <typename T = char, int N = 100> struct vec
{
    T m_at[N];
};

int main()
{
    vec <int, 10> v_i_10;
    vec <int> v_i_100;      // или vec <int, 100> v_i_100;
    vec <> v_c_100;         // или vec <char, 100> v_c_100;
    // ...
}
```

Вывод значений типов из контекста

Вывод параметров-типов

Для шаблонов функций их параметры-типы могут **неявно выводиться из контекста** использования шаблонов.

Параметры-шаблоны (template template parameters)

Пример

```
template
<
    class T1, class T2,
    template <class V1, class V2> class TT
>
struct Data
{
    TT <T1, T2> m_TT;
};
```


Параметры-шаблоны (template template parameters)

Пример

```
template
<
    class T1, class T2,
    template <class, class> class TT
>
struct Data
{
    TT <T1, T2> m_TT;
};
```

Параметры-шаблоны — использование

Пример

```
template <class U1, class U2>
    struct U12
    {
        U1 m_U1;
        U2 m_U2;
    };
```

Пример (окончание)

```
int main()
{
    Data <int, char, U12> data;
    data.m_TT.m_U1 = 10;
    data.m_TT.m_U2 = 'a';
    // ...
}
```

(Не)именованные параметры параметров-шаблонов

Пример

```
template
<
    template <typename V, V *> class TBuffer
>
    struct BufData
{
    static char ms_szData[100];
    TBuffer <char, ms_szData> m_Buffer;
};
```

Параметры параметров-шаблонов (продолжение)

Пример

```
template
<
    template <typename V, V *> class TBuffer
>
    char BufData <TBuffer>::ms_szData[100];
```

Параметры параметров-шаблонов (окончание)

Пример

```
template <typename V, V *PV>
    struct Buffer
    {
        V *m_pV;
        Buffer()
            : m_pV(PV)
        {
            //
        }
    };
```

Пример (окончание)

```
int main()
{
    BufData <Buffer> data;
    data.m_Buffer.m_pV[0] = 'A';
    data.m_Buffer.m_pV[1] = '\0';
    cout << data.ms_szData << endl;
    // ...
}
```

Параметры-шаблоны параметров-шаблонов

Пример

```
template <class T> class Tricky
{
    T m_T;
};
template
<
    template <class> class TT
>
    class TrickyP
    {
        TT <int> m_TT;
    };
};
```

Пример (продолжение)

```
template
<
    template
    <
        template <class> class
    >
        class TTT
    >
        class TrickyPP
    {
        TTT <Tricky> m_TTP;
    };
};
```

Параметры-шаблоны параметров-шаблонов (окончание)

Пример (окончание)

```
int main()
{
    TrickyPP <TrickyP> trickyPP;
}
```

Подстановка стандартных контейнеров

Пример

```
template
<
    class T,
    template <typename> class Container
>
    struct Sequence
    {
        Container <T> m_Container;
    };

typedef Sequence <int, std::list> ListSequence;
```


Объявление списка

Объявление std::list

```
// std::list <>
namespace std
{
    // ...
    template
    <
        typename T,
        typename Allocator = allocator<T>
    >
        class list;
    // ...
}
```

Параметры по умолчанию параметров-шаблонов

Пример

```
template
<
    class T,
    template <typename, typename> class Container
>
    struct Sequence
{
    Container <T, std::allocator <T> > m_Container;
};

typedef Sequence <int, std::list> ListSequence;
```

Параметры по умолчанию параметров-шаблонов

Пример

```
template
<
    class T,
    template <typename, typename = std::allocator <T> > class Container
>
    struct Sequence
    {
        Container <T> m_Container;
    };

typedef Sequence <int, std::list> ListSequence;
```

Стратегии (policies)

Пример

```
template <class T>
    struct OpNewCreator
    {
        static T *Create()
        {
            return new T;
        }
    };
```

Пример

```
template <class T>
    struct MallocCreator
    {
        static T *Create()
        {
            void *pvBuf = std::malloc(sizeof (T));
            if (!pvBuf)
                return NULL;
            return new (pvBuf) T;
        }
    };
```

Стратегии (окончание)

Пример

```
template <class T>
    struct PrototypeCreator
    {
        PrototypeCreator(
            const T *pcT = NULL)
            : m_pcT(pcT) { }
        T *Create()
        {
            return
                (m_pcT ?
                 m_pcT->Clone() : NULL);
        }
    }
```

Пример (окончание)

```
const T *GetPrototype()
{
    return m_pcT;
}
void SetPrototype(const T *pcT)
{
    m_pcT = pcT;
}
private:
    const T *m_pcT;
};
```

Класс, использующий стратегию

Пример

```
template <class TCreationPolicy>
class Manager :
    public TCreationPolicy
{
    // ...
    void DoSomething()
    {
        X *pX = Create();
        // ...
    }
};
```

Пример

```
typedef Manager <OpNewCreator <X> >
XNewManager;
```

Стратегия как параметр-шаблон

Пример

```
template
<
    template <typename> class TCreationPolicy
>
    class Manager : public TCreationPolicy <X>
{
    // ...
};

typedef Manager <OpNewCreator> XNewManager;
```

Стратегия как параметр-шаблон

Пример

```
template
<
    template <typename> class TCreationPolicy = OpNewCreator
>
    class Manager : public TCreationPolicy <X>
{
    // ...
};

typedef Manager <> XNewManager;
```


Типы, зависящие от шаблонных параметров

Пример (шаблон)

```
int p;  
  
template <typename T> void f()  
{  
    T::SomeSubType *p;    // (1)  
}  
  
int main()  
{  
    f <ConcreteT>();    // (2)  
}
```

Пример (ConcreteT)

```
struct ConcreteT  
{  
    typedef int SomeSubType;  
    // или class SomeSubType ...  
};
```

Типы, зависящие от шаблонных параметров

Пример (шаблон)

```
int p;  
  
template <typename T> void f()  
{  
    T::SomeSubType *p;    // (1)  
}  
  
int main()  
{  
    f <ConcreteT>();    // (2)  
}
```

Пример (ConcreteT)

```
struct ConcreteT  
{  
    static int SomeSubType;  
};
```

Типы, зависящие от шаблонных параметров

Пример (шаблон)

```
int p;  
  
template <typename T> void f()  
{  
    T::SomeSubType *p;    // (1)  
}  
  
int main()  
{  
    f <ConcreteT>();    // (2)  
}
```

Пример (ConcreteT)

```
struct ConcreteT  
{  
    static int SomeSubType;  
};
```

Пример (typename)

```
typename T::SomeSubType *p; // (1)
```

Общий случай

Правило

Ключевое слово **typename** используется везде, где имя, зависящее от шаблонного параметра, является типом.

Пример: печать стандартного контейнера

Пример

```
template <typename T> void printcoll(const T &rcColl)
{
    typename T::const_iterator pos;
    typename T::const_iterator end = rcColl.end(); // конец
    //
    for (pos = rcColl.begin(); pos != end; ++ pos)
        cout << *pos << ' ';
    //
    cout << endl;
}
```

Пример: печать контейнера `std::bitset`

Пример

```
template <int N> void printBitset(const std::bitset <N> &rcSet)
{
    cout << rcSet.to_string <char>();    // (1)
}
```

Пример: печать контейнера `std::bitset`

Пример

```
template <int N> void printBitset(const std::bitset <N> &rcSet)
{
    cout << rcSet.to_string <char>();    // (1)
}
```

Пример

```
cout << rcSet.template to_string <char>();    // (1)
```

Общий случай

Правило

Конструкции „**.template**“ и „**->template**“ используется для доступа ко вложенным шаблонам (`to_string <char>()`), если левая часть зависит от шаблонных параметров (`rcSet`).

«Ленивая» конкретизация

Поведение компилятора при (неявной) конкретизации

- Для методов класса всегда конкретизируются их **объявления**.
- **Определения** не конкретизируются, с некоторыми исключениями (виртуальные методы, ...)
- **Значения по умолчанию** для шаблонов функций конкретизируются только, если используются.

«Ленивая» конкретизация определений методов

Пример (x.h)

```
template <typename T> struct X
{
    void f(T &rT)
    {
        rT ++;
    }
};
```

Пример (main.cpp)

```
struct WithoutInc
{
    // Пусто
};

int main()
{
    X <WithoutInc> x;    // верно
    // WithoutInc w;
    // x.f(w); - ошибка: w ++
}
```

«Ленивая» конкретизация аргументов по умолчанию

Пример (inc.h)

```
template <typename T>
    void f(T t = T())
{
    // ...
}
```

Пример (main.cpp)

```
struct WithoutDef
{
    // Нет конструктора по умолчанию
    WithoutDef(int n);
};

int main()
{
    f(WithoutDef(1)); // верно
    f <WithoutDef>(); // ошибка: WithoutDef()
}
```

Параметры и аргументы шаблонов

Определения

Параметры шаблона: **имена**, перечисленные в угловых скобках после слова **template** в **объявлениях** или **определениях** шаблонов.

Аргументы шаблона: **элементы**, помещаемые вместо параметров шаблона.

Пример (параметры)

```
template <typename T, int N>
    struct Array
    {
        T m_aT[N];
    };
```

Пример (аргументы)

```
int main()
{
    Array <double, 10> ad;
    ad.m_aT[0] = 1.0;
}
```

Параметры и аргументы шаблонов

Определения

Параметры шаблона: **имена**, перечисленные в угловых скобках после слова **template** в **объявлениях** или **определениях** шаблонов.

Аргументы шаблона: **элементы**, помещаемые вместо параметров шаблона.

Пример (параметры)

```
template <typename T, int N>
    struct Array
{
    T m_aT[N];
};
```

Пример (аргументы)

```
int main()
{
    Array <double, 10> ad;
    ad.m_aT[0] = 1.0;
}
```

Параметры и аргументы шаблонов (окончание)

Принцип

Любой аргумент шаблона известен на момент компиляции.

Пример (параметр и аргумент шаблона)

```
template <typename T>
    struct Data
    {
        Array <T, 10> m_Array;
    };
```

Подстановка аргументов вместо параметров шаблонов

Способы подстановки

- 1 Явным образом.
- 2 Внедрённое имя класса.
- 3 Значения аргументов по умолчанию.
- 4 Вывод аргументов шаблонов функций.

Пример

```
template <typename T>
    struct X
{
    // ...
};

int main()
{
    X <int> xi;    // (1)
}
```

Подстановка аргументов вместо параметров шаблонов

Способы подстановки

- 1 Явным образом.
- 2 Внедрённое имя класса.
- 3 Значения аргументов по умолчанию.
- 4 Вывод аргументов шаблонов функций.

Пример

```
template <typename T>
    struct X
{
    // ...
    X *m_pNext;    // (2)
    // ~ X <T> *m_pNext;
};
```


Подстановка аргументов вместо параметров шаблонов

Способы подстановки

- 1 Явным образом.
- 2 Внедрённое имя класса.
- 3 Значения аргументов по умолчанию.
- 4 Вывод аргументов шаблонов функций.

Пример

```
template <typename T = int>
    struct X
    {
        // ...
    };

int main()
{
    X <> xi;           // (3)
}
```

Подстановка аргументов вместо параметров шаблонов

Способы подстановки

- 1 Явным образом.
- 2 Внедрённое имя класса.
- 3 Значения аргументов по умолчанию.
- 4 Вывод аргументов шаблонов функций.

Пример

```
template <typename T>
    void Inc(T &rT)
{
    ++ rT;
}

int main()
{
    int n = 0;
    Inc(n);           // (4)
}
```

Ссылка на текущую конкретизацию

Пример

```
template <typename T1, typename T2>
    struct C
    {
        C *m_pC1;           // да, ~ C <T1, T2>
        C <T1, T2> *m_pC2;   // да
        C <T1 *, T2> *m_pC3; // нет
        C <T2, T1> *m_pC4;   // нет
    };
```

Невыводимые аргументы шаблонов функций

Пример (параметр и аргумент шаблона)

```
template <typename TTo, typename TFrom>
    TTo implicit_cast(const TFrom &rcT)
{
    return rcT;
}

int main()
{
    double d = implicit_cast <double> (1);
}
```

Перегрузка шаблонов функций

Пример

```
template <typename T> int f(T)
{
    return 1;
}

template <typename T> int f(T *)
{
    return 2;
}
```

Пример (окончание)

```
int main()
{
    cout << f(0) << endl;
    cout << f((int *) 0) << endl;
}
```

Правила для перегрузки

Правила

- Обычные функции имеют больший приоритет над шаблонными.
- Над шаблонными функциями устанавливается отношение частичного порядка.
- Если более одной функции подходит для перегрузки, программа считается ошибочной.

Принцип SFINAE

Определение

Невозможность подстановки не является ошибкой: (*Substitution Failure Is Not An Error, SFINAE*) — принцип, по которому синтаксически неверная конкретизация шаблона при подстановке его аргумента не является ошибкой.

Пример

```
template <typename T> void f(typename T::Type) {}  
template <typename T> void f(T) {}  
int main()  
{  
    f <int> (10);  
}
```

Принцип SFINAE (окончание)

Замечание

Принцип предотвращает только попытки синтаксически неправильного конструирования типов, но не выражений.

Пример

```
template <int I> void f(int (&)[24 / (4 - I)]);  
template <int I> void f(int (&)[24 / (4 + I)]);  
  
int main()  
{  
    &f <4>;  
}
```


Использование SFINAE

Пример

```
template <typename T>
    struct has_typedef_iterator
    {
        typedef char yes[1];
        typedef char no[2];
        template <typename C>
            static yes& test(typename C::iterator *);
        template <typename>
            static no& test(...);
        //
        static const bool value =
            (sizeof (test <T> (0)) == sizeof (yes));
    };
```

Использование SFINAE (окончание)

Пример

```
int main()
{
    cout << has_typedef_iterator <vector <int> >::value << endl;
    cout << has_typedef_iterator <int>::value << endl;
}
```

Имя и простой идентификатор шаблона

Определения

Имя шаблона: (*template-name*) — в объявлении или определении шаблона указывается после слова **class/struct/union** или имя шаблонной функции.

Простой идентификатор шаблона: (*simple-template-id*) — имя шаблона со списком аргументов в угловых скобках.

Пример

```
template <typename T> struct Data;  
// ----- (1)
```

```
typedef Data <int> IntData;  
// ----- (2)
```

Идентификатор шаблона

Определения

Идентификатор шаблона: (*template-id*) — простой идентификатор шаблона (*simple-template-id*) или конструкция вида „**operator** ... <...>“.

Пример

```
int main()
{
    C c;
    // ...
    operator ++ <C> (c);
// ----- (3)
    // ...
}
```

Явная специализация шаблонов (пример)

Пример (шаблон)

```
template <typename T> struct Data
{
    std::list <T> m_List;
    void Add(const T &rcT);
};

template <typename T>
void Data <T>::Add(
    const T &rcT)
{
    m_List.push_back(rcT);
}
```

Пример (явная специализация)

```
template <> struct Data <char>
{
    int m_nSize;
    char m_achData[1000];
    Data() : m_nSize(0) {}
    void Add(char ch);
};

void Data <char>::Add(char ch)
{
    m_achData[m_nSize++] = ch;
}
```

Объявление строки

Объявление `std::basic_string (<string>)`

```
namespace std
{
    // ...
    template
    <
        class TChar,
        class TTraits = char_traits <TChar>,
        class TAllocator = allocator <TChar>
    >
        class basic_string;
    // ...
}
```

Объявление характеристик типа **char**

Объявление `std::char_traits <char>`

```
template <> struct char_traits <char>
{
    typedef char      char_type;
    typedef int       int_type;
    typedef streampos pos_type;
    typedef streamoff off_type;
    typedef mbstate_t state_type;
    // ...
    static char *copy(char *pszS1, const char *pcszS2, size_t n)
    {
        return (char *) memcpy(pszS1, pcszS2, n);
    }
    // ...
}
```

Основной шаблон класса

Определения

Основное объявление шаблона класса: (*primary class template declaration*) — объявление, в котором в качестве имени используется **имя шаблона** (template-name).

Частичная специализация шаблона класса: (*partial class template declaration*) — объявление, в котором в качестве имени шаблона используется **простой идентификатор шаблона** (simple-template-id).

Назначение частичной специализации

Используется для объявления альтернативной реализации основного шаблона в случае, когда список шаблонных аргументов конкретизации **соответствует** списку аргументов в **простом идентификаторе шаблона**.

Основной шаблон класса

Определения

Основное объявление шаблона класса: (*primary class template declaration*) — объявление, в котором в качестве имени используется **имя шаблона** (template-name).

Частичная специализация шаблона класса: (*partial class template declaration*) — объявление, в котором в качестве имени шаблона используется **простой идентификатор шаблона** (simple-template-id).

Назначение частичной специализации

Используется для объявления альтернативной реализации основного шаблона в случае, когда список шаблонных аргументов конкретизации **соответствует** списку аргументов в **простом идентификаторе шаблона**.

Частичный порядок соответствия шаблонов

Пример

```
template <typename T> class X;  
template <typename T> class X <T *>;  
template <typename T> class X <const T *>;
```

```
// ...
```

```
X <const int *> x1;  
X <int *> x2;  
X <int> x3;
```

Проверка равенства типов

Пример

```
template <class T1, class T2>
    struct equal_types
{
    static const bool value = false;
};

template <class T>
    struct equal_types <T, T>
{
    static const bool value = true;
};
```

Пример (окончание)

```
int main()
{
    cout
        << equal_types <
            long, char>::value
        << endl;
    cout
        << equal_types <
            long, long>::value
        << endl;
}
```

Объявление характеристик итераторов

Объявление основного шаблона `std::iterator_traits`

```
template <typename TIterator>
    struct iterator_traits
{
    typedef typename TIterator::iterator_category iterator_category;
    typedef typename TIterator::value_type        value_type;
    typedef typename TIterator::difference_type    difference_type;
    typedef typename TIterator::pointer            pointer;
    typedef typename TIterator::reference          reference;
};
```

Объявление характеристик итераторов-указателей

Объявление специализации `std::iterator_traits`

```
template <typename TP>
    struct iterator_traits <TP *>
{
    typedef random_access_iterator_tag iterator_category;
    typedef TP value_type;
    typedef ptrdiff_t difference_type;
    typedef TP * pointer;
    typedef TP & reference;
};
```

Зависимый тип

Определение (зависимый тип)

- 1 Параметр шаблона;
- 2 Член **неизвестной специализации**;
- 3 Вложенный класс или объединение — зависящий член от **неизвестной специализации**;
- 4 CV-квалификация **зависимого типа**;
- 5 Сложный тип, сконструированный с использованием **зависимого типа**;
- 6 Тип массива, сконструированный из **зависимого типа** или чей размер — константа, **зависящая от значения**;
- 7 Результат операции „**decltype** ()“, применённой к **выражению, зависящему от типа**.

Неизвестная специализация

Определение (член неизвестной специализации)

Квалификационное имя ($A <T>::k$) или выражение с операцией доступа к члену ($pA \rightarrow k$), класс в которой **зависим**, и либо:

- не **текущая конкретизация**, либо:
- текущая, но имеет хоть 1 **зависимый** базовый класс, поиск имени после „::<“ не находит ни 1 члена в текущем классе и его **независимых** базовых классах.

Неизвестная специализация

Пример

```
template <typename T> struct Base { };

template <typename T> struct Derived : Base <T>
{
    void f()
    { typename Derived <T>::unknown_type z; }
};

template <> struct Base <int>
{
    typedef int unknown_type;
};
```


Шаблонные псевдонимы типов

Пример

```
template <typename T>
    struct AllocList
{
    typedef
        std::list <T, MyAlloc <T> >
        type;
};
```

Пример (окончание)

```
template <typename T>
    class Client
{
    // ...
private:
    typename AllocList <T>::type
        m_List;
};
```

Шаблонные псевдонимы типов

Пример

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

Пример (окончание)

```
template <typename T>
    class Client
    {
        // ...
    private:
        AllocList <T> m_List;
    };
```