

Программирование на языке C++

Вводный курс

Александр Морозов
gelu.speculum@gmail.com

ИТМО, весенний семестр 2022

Содержание

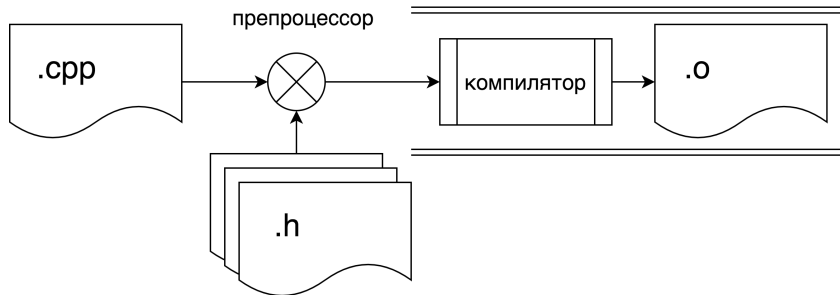
Единицы трансляции

Этапы трансляции

Препроцессор

Макросы

Единица трансляции



Объявление и определение

declaration \neq definition

- ▶ объявление функций
- ▶ предварительное объявление классов
- ▶ предварительное объявление `enum`
- ▶ объявление внешней (`extern`) переменной
- ▶ объявление статического члена класса
- ▶ объявление шаблонного параметра
- ▶ объявление псевдонима типа
- ▶ `using` объявление
- ▶ явное объявление инстанции шаблона
- ▶ специализация шаблона без определения

Объявления

```
1      int f(double);
2      class C;
3      enum class E;
4      extern int a;
5      extern int a;
6
7      struct S
8      {
9          static S s;
10     };
11
12     using T = int;
13     using std::cout;
14
15     template <class T>
16     int g();
17     template <>
18     int g<double>();
19
20     template <class T> class X;
21     template <> class X<int>;
```

Уникальность определений

One definition rule (ODR)

В одной единице трансляции допустимо лишь одно определение любой переменной, функции, класса, `enum` или шаблона.

Во всей программе допустимо лишь одно определение функции или переменной, которая ODR-используется (иначе – **UB**).

Повторение определений

Во всей программе может быть более одного определения класса, `enum`, шаблона, если:

- ▶ каждый вариант определения состоит из одинаковой последовательности токенов
- ▶ поиск имени, связанного с каждым таким определением, во всех случаях дает ту же сущность
- ▶ любые связанные операторы и методы должны давать вызов одной и той же функции в каждом случае

ODR-использование

- ▶ объект ODR-используется, если его значение читается (если только это не константа с известным на момент компиляции значением), записывается, его адрес берётся или с ним связывается ссылка
- ▶ функция ODR-используется, если она вызывается или её адрес берётся

Пример ODR-использования

```
1      struct S
2      {
3          static const int x = 0; // declaration
4      };
5
6      const int S::x; // definition
7
8      int main(int argc, char ** argv)
9      {
10         const int & x = S::x; // S::x is ODR-used
11         return S::x; // S::x is not ODR-used
12     }
```

inline

```
1    inline void f()
2    {
3        // ...
4    }
5
6    inline int x = 0;
7
8    struct S
9    {
10        bool empty()
11        {
12            return true;
13        }
14
15        friend bool operator == (const S & lhs, const S & rhs)
16        { return true; }
17
18        static constexpr double pi = 3.14;
19    };
```

Linkage

Имя, обозначающее объект, ссылку, функцию, тип, шаблон, пространство имен или значение, может иметь linkage (связывание).

- ▶ отсутствие связывания: имя является локальным для своей области видимости
- ▶ внешнее связывание: имя в разных единицах трансляции ссылается на одну и ту же сущность
- ▶ внутреннее связывание: имя в любых областях видимости данной единицы трансляции ссылается на одну и ту же сущность

Назначение связывания по умолчанию

- ▶ отсутствие связывания
 - ▶ локальные переменные без `extern`
 - ▶ локальные классы
 - ▶ иные имена, объявленные на уровне блока
- ▶ внешнее связывание
 - ▶ глобальные не-`const` не-`inline` переменные
 - ▶ глобальные `inline` переменные
 - ▶ функции
 - ▶ классы
 - ▶ шаблоны
- ▶ внутреннее связывание
 - ▶ глобальные `const` не-`inline` переменные
 - ▶ члены анонимных `union`
 - ▶ сущности, объявленные в анонимном пространстве имён

Явное назначение связывания

- ▶ **extern** – спецификатор объявления переменных, позволяющий явно задать внешнее связывание
- ▶ **static** – спецификатор определения глобальных переменных или функций, явно задающий внутреннее связывание

```
1  extern int x; // external, declaration
2  extern int y = 101; // external, definition
3  int z = 111; // external
4  static int u = -1; // internal
5  const int v = 9; // internal
6  static const int w = 3; // internal
7  inline const int s = 3; // external
8  extern const int t = 1; // external
```

Language linkage

Имена переменных и функций с внешним связыванием обладают свойством “language linkage”.

Можно задать явно:

```
extern string-literal { ... }
```

```
extern string-literal declaration
```

где *string-literal*:

▶ "C++"

▶ "C"

Содержание

Единицы трансляции

Этапы трансляции

Препроцессор

Макросы

Основные этапы трансляции

1. Склейка строк через \
2. Предварительная токенизация (комментарии, пробелы, базовые токены)
3. Препроцессор ($\forall I \Rightarrow 1 \dots 3$)
4. Склейка смежных строковых литералов
5. Компиляция
6. компоновка

Базовые токены

- ▶ идентификаторы
- ▶ числовые токены
- ▶ символьные, строковые литералы и аргументы `#include`
- ▶ операторы и символы пунктуации
- ▶ иное

```
1      a+++++b // a ++ ++ + b
2      a++ + ++b // a ++ + ++ b
3      1E+12 // OK
4      0x1E+12 // bad
5      0x1E +12 // OK
```

Комментарии

```
1  /* Comment */
2
3  /*
4   * Multi-line comment
5   */
6
7  // Single-line comment
8
9  //
10 //
11 // Fancy comment formatting
12 //
```

Содержание

Единицы трансляции

Этапы трансляции

Препроцессор

Макросы

Директивы препроцессора

директива [аргументы]

Директива должна размещаться на одной строке, занимает всю строку.

- ▶ `define`
- ▶ `undef`
- ▶ `include`
- ▶ `if`, `ifdef`, `ifndef`, `else`, `elif`, `endif`
- ▶ `error`
- ▶ `pragma`
- ▶ `line`

pragma

#pragma *params*

```
1      #pragma once
2
3      #pragma pack(1)
4      struct S
5      {
6          char a;
7          int b;
8          short c;
9      };
```

error

#error message

```
1  #if defined(__clang__)
2  // clang
3  #elif defined(__GNUC__) || defined(__GNUG__)
4  // gcc
5  #elif defined(_MSC_VER)
6  // MSVC
7  #else
8  # error "Unsupported_compiler"
9  #endif
```

line

- ▶ `#line` *lineno*
- ▶ `#line` *lineno* "*filename*"

```
1      std::cout << __FILE__ << ":" << __LINE__ << std::endl;  
2  #line 333 "foo.def"  
3      std::cout << __FILE__ << ":" << __LINE__ << std::endl;
```

Условные директивы

```
1      #ifdef identifier
2      #ifndef identifier
3      #if expression
4      #elif expression
5      #else
6      #endif
```


Условные директивы, конструкция

```
1      #ifdef / ifndef / if
2      // main case
3      ...
4      #elif ...
5      // alternative case
6      ...
7      #else
8      // else case
9      ...
10     #endif
```

Условные директивы, выражения

1. подстановка макросов
2. оператор `defined id` или `defined(id)` → 1 или 0
3. оператор `__has_include(...)` → 1 или 0
4. прочие идентификаторы → 0 (кроме `true/false`)
5. базовые токены → токены
6. вычисление выражения, как константного

Условные директивы, пример

```
1    // #define FOO
2    // #define X 3 * 1 + 0
3    #if !defined(FOO)
4        std::cout << "First" << std::endl;
5    #elif X == 3
6        std::cout << "Second" << std::endl;
7    #else
8    # ifndef BAR
9        std::cout << "Third" << std::endl;
10   # endif
11   #endif
```

include

- ▶ `#include < файл >` – “стандартные” или “глобальные” файлы
- ▶ `#include " файл "` – “локальные” файлы

Выражение для `__has_include` имеет такую же форму и смысл.

Заголовочные файлы

a.h:

```
1     int max(int a, int b);  
2     inline const double pi = 3.14;
```

a.cpp:

```
1     void f() {}  
2     #include "a.h"  
3     #include "a.h"  
4     int x = max(10, 100);
```

a.i:

```
1     void f() {}  
2     int max(int a, int b);  
3     inline const double pi = 3.14;  
4     int max(int a, int b);  
5     inline const double pi = 3.14;  
6     int x = max(10, 100);
```

Защита от повторного включения

“Include guards”:

```
1      #ifndef SOME_UNIQUE_HEADER_NAME
2      #define SOME_UNIQUE_HEADER_NAME
3
4      ...
5      #endif
```

Альтернатива:

```
1      #pragma once
```

Содержание

Единицы трансляции

Этапы трансляции

Препроцессор

Макросы

Макросы

```
1      #define identifier
2      #define identifier replacement
3      #define identifier(parameters) replacement
4      #define identifier(parameters, ...) replacement
5      #define identifier(...) replacement
6      #undef identifier
```


Сложные макросы

```
1      #define MAX(a, b) a < b ? b : a
2
3      if (MAX(x, y) == 0) {
4          ...
5      }
6
7      ->
8
9      if (x < y ? y : x == 0) {
10         ...
11     }
12
13     ->
14
15     if ((x < y) ? (y) : (x == 0)) {
16         ...
17     }
```

Ограничения макросов

Рекурсия запрещена

```
1      #define foo foo
2
3      foo
4      ->
5      foo
```

Вложенные вызовы разрешены

```
1      #define foo(x) x
2
3      foo(foo(foo(1)))
4      ->
5      1
```

Особенности подстановки параметров

В подстановке сложных макросов параметры дополнительно раскрываются.

Этапы подстановки:

1. Определение мест вхождения параметров в строке замены и их первичная подстановка
2. Раскрытие параметров в местах их подстановки
3. В строке замены обрабатываются макросы (при этом явно запрещена рекурсия)

Специальные операции подстановки

В подстановке сложных макросов `#` и `##` играют особую роль:

- ▶ stringification (литерация): `#a`, где `a` – параметр макроса; вместо обычной подстановки параметра, параметр превращается в корректный строковый литерал (с обрамляющими `"`) и подставляется в строку замены
- ▶ concatenation (склейка): `##` между любыми двумя параметрами или между параметром и другим токеном вызывает конкатенацию результатов их подстановки (препроцессор проверит корректность полученного токена)

Примеры stringification и concatenation

```
1      #define show(a, b, c) #a #b #c
2
3      show(x, 10, "hello\n")
4      ->
5      "x10\"hello\\n\"
6
7
8      #define print(type) void print_ ## type (type x) { ...
9      ↪      }
10
11     print(char)
12     ->
13     void print_char(char x) { ... }
```

Особенности stringification и concatenation

При этом `#` и `##` оказывает влияние на обработку параметров макросов: литерация или склейка производятся **до** возможного раскрытия параметров.

```
1      #define foo ABC
2      #define concat(a) foo ## a
3
4      concat(__LINE__)
5      ->
6      foo__LINE__
```

Обход особенностей stringification и concatenation

Обойти раннюю обработку операций `#` и `##` можно двойным перенаправлением:

```
1      #define show(x) do_show(x)
2      #define do_show(x) #a
3
4      show(__LINE__)
5      ->
6      "123"
7
8      #define concat(a, b) do_concat(a, b)
9      #define do_concat(a, b) a ## b
10
11     concat(0x, __LINE__)
12     ->
13     0x123
```

Защита параметров макросов

```
1      #define MAX(a, b) a < b ? b : a
2
3      MAX(0, x & 0xFF)
4      ->
5      0 < x & 0xFF ? x & 0xFF : 0
```

```
1      #define MAX(a, b) (a) < (b) ? (b) : (a)
2      MAX(0, x & 0xFF)
3      ->
4      (0) < (x & 0xFF) ? (x & 0xFF) : (0)
```


Защита тела макросов

```
1      #define MAX(a, b) (a < b ? b : a)
2
3      if (MAX(x, y) == 0) {
4          ...
5      }
6      ->
7      if ((x < y ? y : x) == 0) {
8          ...
9      }
```

Защита места вставки

```
1      #define PRINT(x) do { \  
2          ...; \  
3          ...; \  
4      } while (false)  
5  
6      // Now this is OK:  
7      if (...)   
8          PRINT(...);  
9      else  
10         ...;
```

Побочные эффекты в макросах

```
1      #define MAX(a, b) (a) < (b) ? (b) : (a)
2
3      MAX(x++, --y);
4      ->
5      (x++) < (--y) < (--y) : (x++);
```