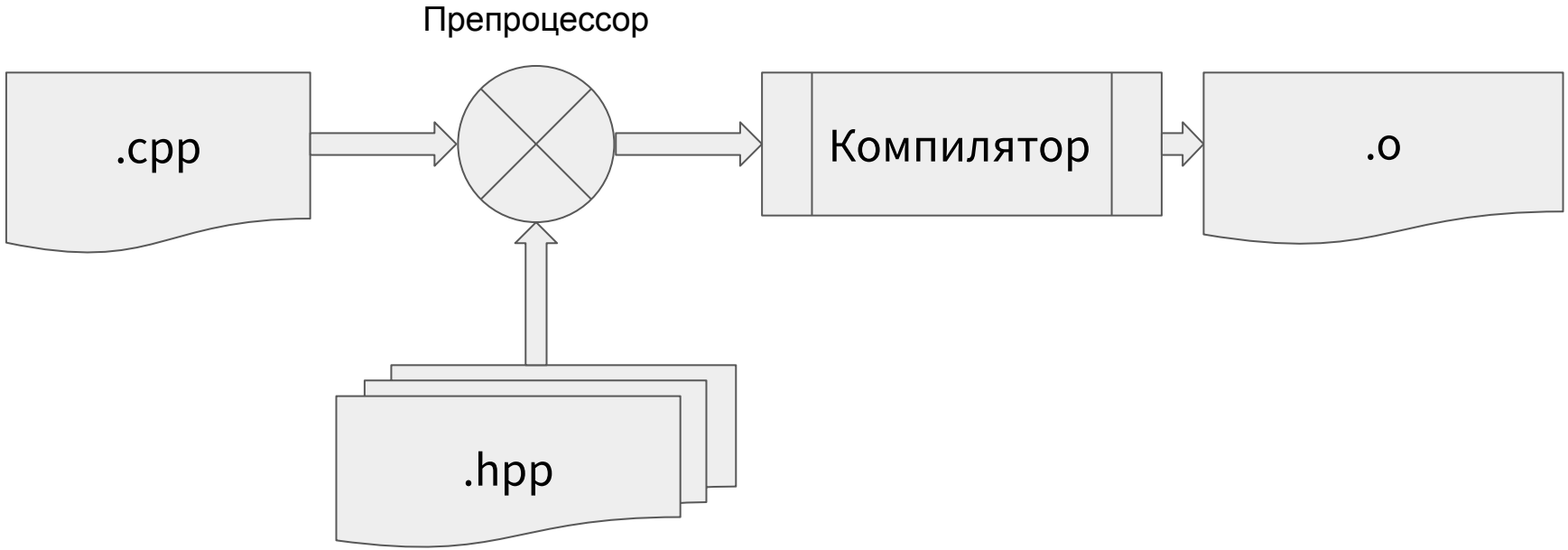
Программирование на языке C++

Шаблов Анатолий
anatoliishablov@gmail.com

ИТМО, весенний семестр 2024

Единицы трансляции

Единица трансляции



Объявление и определение

declaration ≠ 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 инстанциации шаблона
- Специализация шаблона без определения

Объявления

```
int f(double);
class C;
enum class E;
enum E2 : int;
extern int a;
extern int a;

struct S
{
    static S s;
};
```

```
using T = int;
using std::cout;

template <class T>
int g();
template <>
int g<double>();

template <class T>
class X;
```

Уникальность определений

One Definition Rule (ODR) - Правило Одного Определения

В любой единице трансляции допустимо только одно определение любой переменной, функции, типа класса, типа перечисления, концепта (начиная с C++20) или шаблона.

Одно и только одно определение каждой невстраиваемой функции или переменной, которая использует odr, должно присутствовать во всей программе.

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

```
struct S
{
    static const int x = 0; // статический элемент данных
    // требуется определение вне класса, если он используется
odr
};

// const int S::x;

const int& f(const int& r);

int n = b ? S::x // S::x здесь не используется odr
          : f(S::x);  // S::x здесь используется odr: требуется
определение
```

inline

```
inline void f() {
    // ...
}
inline int x = 0;
struct S {
    bool empty() { return true; }

    friend bool operator==(S const & lhs, S const & rhs) {
        return true;
    }

    constexpr static double pi = 3.14;
};
```

Связывание

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

- 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

```
extern int x;           // external, declaration
extern int y = 101;     // external, definition
int z = 111;            // external
static int u = -1;      // internal
const int v = 9;        // internal
static const int w = 3; // internal
inline const int s = 3; // external
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"

Этапы трансляции

Основные этапы трансляции

1. Склейка строк через \
2. Предварительная токенизация (комментарии, пробелы, базовые токены)
3. Препроцессор (includes пункты 1-3)
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

```
a+++++b // a ++ ++ + b
a++ + ++b // a ++ + ++ b
1E+12 // OK
0x1E+12 // bad
0x1E +12 // OK
```

Комментарии

```
/* Comment */

/*
 * Multi-line comment
 */

// Single-line comment

//
//
// Fancy comment formatting
//
```

Препроцессор

Директивы препроцессора

директива [аргументы]

Директива должна размещаться на одной строке, занимает всю строку.

- define
- undef
- include
- if, ifdef, ifndef, else, elif, endif (C++23 elifdef elifndef)
- error (C++23 warning)
- pragma
- line

#pragma *params*

```
#pragma once

#pragma pack(1)
struct S
{
    char a;
    int b;
    short c;
};
#pragma pack(pop)
```

`#error message`
`#warning message`

```
#if defined(__clang__)
// clang
#elif defined(__GNUC__) || defined(__GNUG__)
// gcc
#elif defined(_MSC_VER)
// MSVC
#warning "Need fixes"
#else
#error "Unsupported compiler"
#endif
```

#line lineno [“filename”]

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

Условные директивы

```
#ifdef identifier
#ifndef identifier
#if expression
#elif expression
#else
#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, примеры

```
// #define FOO
// #define X 3 * 1 + 0
#if !defined(FOO)
std::cout << "First" << std::endl;
#elif X == 3
std::cout << "Second" << std::endl;
#else
#ifndef BAR
std::cout << "Third" << std::endl;
#endif
#endif
```

`#include` *header*

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

- `#include "файл"` – “локальные” файлы

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

Заголовочные файлы

```
// a.h:
int max(int a, int b);
inline const double pi = 3.14;
// a.cpp:
void f() {}
#include "a.h"
#include "a.h"
int x = max(10, 100);
```

```
// a.i:
void f() {}
int max(int a, int b);
inline const double pi = 3.14;
int max(int a, int b);
inline const double pi = 3.14;
int x = max(10, 100);
```

Защита от повторного включения

“Include guards”

```
#ifndef SOME_UNIQUE_HEADER_NAME
#define SOME_UNIQUE_HEADER_NAME
#endif // SOME_UNIQUE_HEADER_NAME
```

Альтернатива

```
#pragma once
```

Макросы

Макросы

```
#define identifier
#define identifier replacement
#define identifier(parameters) replacement
#define identifier(parameters, ...) replacement
#define identifier(...) replacement
#undef  identifier
```

Функциональные макросы

```
#define MAX(a, b) a < b ? b : a

if (MAX(x, y) == 0) { ... }

->

if (x < y ? y : x == 0) { ... }

->

if ((x < y) ? (y) : (x == 0)) { ... }
```

Ограничения макросов

- Рекурсия запрещена

```
#define foo foo

foo -> foo
```

- Вложенные вызовы разрешены

```
#define foo(x)

foo(foo(foo(1))) -> 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

- *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

```
#define show(a, b, c) #a #b #c

show(x, 10, "hello\n") -> "x10\"hello\\n\""

#define print(type)             \
    void print_##type(type x) \
    {                         \
        ...                   \
    }

print(char) -> 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.

```
#define foo ABC
#define concat(a) foo##a

concat(__LINE__) ->foo__LINE__
```

Обход особенностей stringification и concatenation

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

```
#define show(x) do_show(x)
#define do_show(x) #a

show(__LINE__) -> "123"

#define concat(a, b) do_concat(a, b)
#define do_concat(a, b) a##b

concat(0x, __LINE__) -> 0x123
```

Защита параметров макросов

```
#define MAX(a, b) a < b ? b : a

MAX(0, x & 0xFF) -> 0 < x & 0xFF ? x & 0xFF : 0

#define MAX(a, b) (a) < (b) ? (b) : (a)

MAX(0, x & 0xFF) -> (0) < (x & 0xFF) ? (x & 0xFF) : (0)
```

Защита тела макросов

```
#define MAX(a, b) (a < b ? b : a)

if (MAX(x, y) == 0) {
  ...
}
->
if ((x < y ? y : x) == 0) {
  ...
}
```

Защита места вставки

```
#define PRINT(x) \
    do {         \
        ...;     \
        ...;     \
    } while (false)

// Now this is OK:
if (...)
    PRINT(...);
else
    ...;
```

Побочные эффекты в макросах

```
#define MAX(a, b) (a) < (b) ? (b) : (a)

MAX(x++, --y);

->

(x++) < (--y) < (--y) : (x++);
```