# Лекция 09 — Трансляция

### Содержание



---

# Введение

Особенно опытным программистам известно, что текст программы на каком бы то ни было языке программирования в действительности не представляет собой непосредственно исполняемый программный файл. Это просто набор символов, причём не всегда осмысленный.

Для того, чтобы стать исполняемым, исходному файлу необходимо пройти через процесс **трансляции**. Это очень говорящее название, потому что в результате трансляции происходит буквально «перевод» кода на каком-либо языке программирования в машинный код — ровно как если бы мы переводили текст с русского языка на, скажем, бельгийский.

Конечно, это не совсем тривиальная задача: процедура трансляции кода, написанного на C++, состоит из 9 различных фаз. И хотя углубляться в них для того, чтобы писать хороший код, совершенно не обязательно, на фундаментальном понимании процесса трансляции зиждется ряд важных навыков, необходимых C++-разработчику. Далее будут подробно рассмотрены основные теоретические этапы трансляции, все связанные с ними особенности написания кода и некоторые более общие практические выводы.

---

# Фазы трансляции

## Подготовительные фазы

Функция первых трех фаз трансляции — подготовить код для дальнейшей его обработки. Так, они включают в себя:

### 1. Маппинг исходного файла

В этой фазе отдельным байтам исходного кода сопоставляются символы из *стандартного набора символов*[1], поддерживаемых языком — в частности, ОС-зависимые индикаторы конца строки заменяются специальным символом новой строки. Байтам, для которых такой маппинг не поддерживается, сопоставляются соответствующие символы Юникода. На этом же этапе происходит преобразование триграф-последовательностей (до их упразднения в C++17).[2]

Начиная с C++23, Стандарт гарантирует поддержку исходных файлов в кодировке UTF-8 и преобразует их отдельным, но согласованным с описанным выше способом; поддержка любых других форматов зависит от компилятора.

### 2. Соединение строк

В этой фазе удаляются все символы новой строки, соединяя строки исходного файла в одну. Если непустой исходный файл не оканчивается символом новой строки, он добавляется.

## 3. Лексический анализ

В этой фазе подготовленный исходный файл декомпозируется в последовательность *токенов препроцессора* и *пробелов*.

> ⓘ **Определение**
>
> **Токен препроцессинга** — минимальный лексический элемент языка в 3-6 фазах трансляции. В эту категорию входят:
>
> - имена заголовков;
> - идентификаторы;
> - литералы;
> - операторы и знаки пунктуации, включая диграф-последовательности; [2-1]
> - отдельные непробельные символы из стандартного набора, за исключением апострофа и кавычки.

> ⓘ **Определение**
>
> **Пробелы** состоят из комментариев, пробельных символов (включающих в себя непосредственно пробелы, знаки табуляции, переводы строк и.т.п.[3]) или и того, и того. Они используются для разделения токенов препроцессора за следующими исключениями:
>
> 1. разделители в строковых и символьных литералах или именах заголовков считаются их частью;
> 2. директивы препроцессора (см. далее) не могут содержать символов новой строки.

После декомпозиции все строковые литералы возвращаются в исходное состояние, т.е. откатываются все изменения, произведенные с ними в первых двух фазах; помимо этого, все комментарии заменяются на один символ пробела. Если исходный файл заканчивается неполным токеном препроцессинга или неполным комментарием, трансляция прервётся.

Лексический анализ проходит согласно правилу *maximal munch*. Это означает, что каждый следующий токен препроцессинга выбирается по принципу наибольшей возможной длины или, иными словами, жадно:

```
int main() {
    int a = 5;
    int b = 2;
    std::cout << a+++++b; // a++ ++ +b, а не (a++) + (++b)
}
```

Из этого правила, впрочем, существует ряд исключений, связанных с парсингом имён заголовочных файлов, строковых литералов, шаблонов и последовательностей `<::`.[4]

---

# Препроцессор

## Общие представления

На следующем шаге код проходит обработку **препроцессором** — специальной программой, которая является частью компилятора и осуществляет ряд подготовительных преобразований, необходимых для дальнейшего хода трансляции. В частности, препроцессор способен:

- совершать условное выполнение частей кода;
- заменять текстовые макросы;
- совершать вставку одних исходных файлов в другие;
- вызывать ошибки или предупреждения.

Контроль над поведением препроцессора осуществляется с помощью специальных инструкций, называемых **директивами** — с этой точки зрения его можно считать полноценным встроенным языком программирования, и этой идеей вполне можно увлечься, хотя это и не рекомендуется.

## Директивы препроцессора

### Директива `#define`

#### Синтаксис

При помощи директивы `#define` реализуется семантика текстовых макросов. Самый простой тип поддерживаемых макросов — *объектные*:

```
#define tail 238
```

Здесь `tail` будет «псевдонимом» для `238`, и на соответствующем этапе препроцессор просто заменит все вхождения такого макроса на указанное значение.

Гораздо интереснее ситуация обстоит с *функциональными* макросами:

```
#define max(a, b) (a > b ? a : b)
```

Синтаксис вызова таких макросов аналогичен синтаксису вызова функций. Так, в этом случае препроцессор не только раскроет сам макрос, но еще и подставит переданные в качестве аргументов значения на соответствующие позиции:

```
int x = max(238, 30); // == int x = (238 > 30 ? 238 : 30);
```

Начиная с C++11, функциональные макросы могут принимать неопределенное число значений. В таком случае доступ к ним осуществляется при помощи объектного макроса `__VA_ARGS__`, который раскрывается в список переданных аргументов, разделённых запятой:

```
#include <iostream>
#include <algorithm>

#define max(...) std::max({__VA_ARGS__})

int main() {
    std::cout << max(52, 238, 30);
}
```

Начиная с C++20 поддерживается также функциональный макрос `__VA_OPT__(content)`, который возвращает `content`, если были переданы неименованные аргументы, и ничего иначе:

```
#define moo(x, ...) __VA_OPT__(1) + x

int main() {
    std::cout << moo(2) << std::endl;       // == std::cout << + 2
    std::cout << moo(2, 238) << std::endl;  // == std::cout << 1 + 2
}
```

Важно, что если при вызове функционального макроса препроцессор встречает запятую в выражении, не окружённом круглыми скобками, он *всегда* трактует её как разделитель аргументов:

```
#define print(pair) std::cout << pair.first << ' ' << pair.second

int main() {
    print(std::pair<int, int>(1, 2)); // CE!
    print((std::pair<int, int>(1, 2))); // OK
}
```

Действие директивы `#define` отменяется директивой `#undef`:

```
int main() {
    #define x 238;
    std::cout << x; // OK
    #undef x;
    std::cout << x; // CE: identifier "x" is undefined
}
```

Если при этом `#undef` принимает имя, для которого не был определён макрос, ничего не происходит.

### Операторы `#` и `##`

При описании значения функционального макроса можно использовать оператор `#`, который заключает в кавычки идущий после него аргумент, объявляя строковый литерал. Помимо этого, препроцессор также экранирует все специальные символы, если это необходимо. Эта операция называется *stringification* (объявлен конкурс на лучший перевод):

```
#define showlist(...) std::cout << #__VA_ARGS__

int main() {
    showlist(52, "orange", {1, 2, 3}); // output: 52, "orange", {1, 2, 3}
}
```

Если результат действия оператора `#` не является валидным строковым литералом, поведение не определено.

Оператор `##` реализует семантику склеивания токенов (*token pasting*), буквально соединяя два переданных аргумента воедино и обрабатывая результат:

```
int hvost = 238;

#define macro(a, b) std::cout << a##b

int main() {
    macro(hv, ost); // output: 238
}
```

Два токена подвергаются склеиванию, если верно одно из следующего:

- они формируют идентификатор;
- они формируют число;
- они формируют оператор.

Нельзя создать комментарий склеиванием `/` и `*`, поскольку к фазе препроцессора они уже удалены из текста программы. Если результат склеивания не является валидным токеном, поведение не определено.

### Рекурсия

Каждый раз, когда препроцессор встречает макрос, он разворачивает его и сканирует результат. Если это функциональный макрос, предварительно также сканируются переданные аргументы (за исключением случаев, когда они являются операндами при `#` или `##`).

Препроцессор игнорирует уже раскрытые макросы, что предотвращает рекурсию. Тем не менее, при должной сноровке макросы всё же можно[5] сделать рекурсивными в обход алгоритмов препроцессора, однако этот процесс достаточно неприятен и посему рассмотрен не будет.

### О пользовательских макросах

Первое, что узнает программист, начинающий работать с C++ — как написать программу, которая выводит «Hello, World». Второе, что узнает программист, начинающий работать с C++ — нельзя использовать макросы.

На первый взгляд кажется, что макросы приносят в язык не только удобный, но и буквально незаменимый функционал. В сущности, это действительно так. Проблема кроется в другом: **препроцессор ничего не знает про C++**.

И это действительно проблема, потому что никаких дополнительных проверок адекватности происходящего не выполняется. Легко убедить себя в том, что раз макрос написан лично тобой, то ты-то точно знаешь, как он работает, а значит, всё нормально!

Помимо того, что это с высокой вероятностью просто не является правдой, стоит помнить, что код чаще читается, чем пишется, а другие люди в этом, вероятнее всего, не разберутся.

Поведение макросов неочевидно. Рассмотрим классический пример:

```
#define square(x) x * x

int main() {
    int x = 5;
    std::cout << square(++x);
}
```

Можно ожидать, что сначала произойдёт инкремент `x`, а после — возведение в квадрат, и программа выведет `36`. Теперь, уже будучи знакомыми с принципами работы текстовых макросов (они не зря называются *текстовыми*), мы, конечно, понимаем, что `square(++x)` раскроется в следующий код:

```
int main() {
    int x = 5;
    std::cout << (++x) * (++x);
}
```

и результатом будет `42`.

Этот пример может показаться слишком простым и, в сущности, таким и является, однако он очень ярко отражает все трудности интерпретации, которые могут возникнуть при работе с функциональными макросами.

Рассмотрим ещё один пример. Допустим, мы имеем следующий макрос, предотвращающий деление на ноль:

```
#define safe_divide(res, x, y)   if (y != 0) res = x/y;
```

после чего используем его в конструкции с условным оператором:

```
if (something) safe_divide(b, a, x);
else std::cout << ("Something is not set...");
```

Поведение этого когда будет как минимум странным и, вероятнее всего, точно не таким, каким мы его ожидали.

Даже будучи полностью уверенным в своих действиях, при использовании функциональных макросов в такую ситуацию попасть крайне просто. Макросы усложняют чтение кода, их невозможно нормально отлаживать и, как результат, они несут совершенно непредсказуемые последствия. Легко видеть, почему их не рекомендуется использовать вообще; однако если это всё же оказывается необходимо, к макросам нужно подходить с *предельной* осторожностью.

### Директива `#pragma`

Директива `#pragma` позволяет осуществлять контроль над поведением, за реализацию которого отвечает компилятор. Отсюда сразу следует, что Стандарт не требует от компиляторов поддержки каких-либо конкретных прагм, однако ряд из них традиционно поддерживается: например, уже известная `#pragma pack` или рассматриваемая далее `#pragma once`.

Начиная с C++11, прагмы также можно объявлять при помощи следующего синтаксиса:

```
_Pragma("once") // == #pragma once
```

Переданный строковый литерал будет подготовлен и преобразован в токен препроцессинга (так же, как в третьей фазе трансляции), после чего будет вызвана обычная форма объявления прагм.

### Условное включение

Мы можем включать блоки кода только при выполнении определённых условий. Эта семантика реализуется директивами `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`, а также, начиная с C++23, `#elifdef` и `#elifndef`.

Блоки условного включения имеют такую же структуру, как и обыкновенный условный оператор за тем исключением, что они обязаны заканчиваться директивой `#endif`. Каждая открывающая директива контролирует блок кода, продолжающийся до соответствующего `#elif`, `#else`, `#elifdef`, `#elifndef` или `#endif`, не принадлежащего вложенным блокам условного включения.

Директивы `#if` и `#elif` могут принимать любые булевы выражения, а также специальные операторы `defined id` и `defined(id)`:

```
#define tail 238

int main() {
    #ifdef orange
        std::cout << "303030";
```

```
    #elif defined tail
        std::cout << "238238238";
    #else
        std::cout << "52";
    #endif
}
```

Начиная с C++11, поддерживается также оператор `__has_include()`, проверяющий исходные файлы на предмет включения директивой `#include` (см. далее).

## Директива `#include`

### Механизм работы

При помощи директивы `#include` реализуется семантика вставки одних исходных файлов в другие. Она имеет следующие формы:

```
#include <header>                                                    (1)
#include "header"                                                    (2)
#include header                                                      (3)
```

Строго говоря, разница между угловыми скобками и кавычками не закреплена в Стандарте и зависит от компилятора. На практике при работе с *большинством* компиляторов поиск имени, заключенного в угловые скобки, будет осуществляться среди стандартных библиотек; если же имя заключить в кавычки, будет сначала просмотрена директория проекта, и только потом, если поиск не был успешен, будут рассмотрены стандартные библиотеки.
В случае (3), когда ни скобки, ни кавычки не были указаны, будет предпринята попытка раскрыть соответствующий макрос и, если это возможно, произойдёт переход к одной из двух предыдущих форм. Поведение в ситуации, когда раскрытый макрос не подошёл ни под одну из них, не определено.

Эта директива, как ни странно, не производит никаких более интеллектуальных операций, чем вставка текста (*помним, что препроцессор ничего не знает о языке*). Так, например, мы вполне можем включить в исходный код изображение или видео, указать одну библиотеку 238 раз или вообще вставить два исходных файла друг в друга, создав рекурсию. Нетрудно догадаться, что всё из этого либо точно приведёт, либо с легкостью может привести к неприятным последствиям. Хотя причина ошибки очевидна, если мы пытаемся подключить к коду картинку, многократное включение — проблема уже более интересная.

### Стражи

Начать следует с того, что это в самом деле является проблемой. В лучшем случае результатом будет неконтролируемое увеличение объёма кода, в худшем — переопределение переменных и ошибка компиляции. Казалось бы, можно ведь просто так не делать — кому вообще придет в голову вставлять один файл несколько раз? — однако трудности возникают в момент, когда несколько подключаемых файлов обращаются к одной и той же библиотеке. Это не всегда легко даже отследить, не говоря уже о том, чтобы как-то повлиять на архитектуру самих этих файлов в попытке исправить ситуацию.

Здесь на помощь приходит условная компиляция и концепция **стражей** (*include guards*). Когда мы впервые определяем содержимое исходного файла, мы устанавливаем соответствующий индикатор, и проверяем его существование при всех дальнейших включениях:

```
#ifndef SOURCE_FILE_1 // если файл не был подключен ранее
#define SOURCE_FILE_1 // индикатор, означающий, что файл подключен

#include <iostream>

int main() {
    std::cout << "Hello, World!\n";
}

#endif
```

Логику стражей, впрочем, гораздо удобнее реализовывать при помощи `#pragma once`, которая несёт именно эту семантику и широко поддерживается многими компиляторами.

## Другие директивы

Среди прочих директив препроцессора можно выделить:

> **Диагностические директивы**
>
> Директива `#warning` выводит в консоль пользовательское предупреждение. Директива `#error` также выводит в консоль заданное пользователем сообщение об ошибке, однако ещё и приостанавливает компиляцию:
>
> ```
> int main() {
>     #warning This is a user-defined message
>     // main.cpp:2:6: warning: This is a user-defined message
> }
> ```

> **Директива `#line`**
>
> Директива `#line` изменяет данные о текущем номере строки и (опционально) имени файла:
>
> ```
> #include <cassert>
>
> int main() {
>     #line 238 itmo
>     assert(2 + 2 == 5);
>     // a.out: itmo:238: int main(): Assertion `2 + 2 == 5' failed.
> }
> ```
>
> Она также затрагивает обращения к макросам `__LINE__` и `__FILE__`.

## Препроцессор как фаза трансляции

Итак, в четвёртой фазе трансляции происходит следующее:

1. программа проходит через препроцессор;
2. каждый файл, добавленный директивой `#include`, проходит через 1-4 фазы трансляции;
3. все препроцессорные директивы удаляются из исходного файла.

> **Примечание**
>
> Многие компиляторы позволяют запустить фазу препроцессора отдельно, то есть прервать трансляцию после этого этапа. В clang это можно сделать путём указания флага `-E`.

---

# Компиляция

## Общие представления

Этапу компиляции предшествуют фазы (5) и (6), в которых определяется кодировка строковых литералов и происходит их конкатенация. [6]

Итак, компиляция является седьмой фазой трансляции. В процессе компиляции все токены препроцессинга преобразуются в то, что называется просто *токеном*. Эта категория включает в себя:

- идентификаторы;
- ключевые слова;
- литералы;
- операторы и знаки пунктуации (за исключением препроцессорных операторов).

После этого все токены проходят синтаксический и семантический анализ (примерно в этот момент восьмой фазой трансляции происходит инстанцирование шаблонов), продолжая трансляцию как целая *единица трансляции*. Результатом компиляции является *объектный файл*.

> **Примечание**
>
> Как и препроцессор, компиляцию можно запустить отдельно. В clang для этого используется флаг `-S`.
>
> Если при этом также необходимо провести ассемблирование, вместо этого указывается флаг `-c`.

## Принцип as-if

Трансляция происходит согласно правилу «as-if», которое допускает любые преобразования кода, не изменяющие наблюдаемого поведения программы. Это оказывает наиболее сильное влияние именно на процесс компиляции.

Дело в следующем: компилятор действительно обязан сохранять неизменным любое *определенное* поведение программы, но если это определение внезапно *не определено*, он освобождается от необходимости соблюдать какие-либо гарантии. Вообще.

То есть даже если операция производилась до момента возникновения UB, мы не можем быть уверены в том, что она корректно выполнится. Это, конечно, скорее формальность и на практике ошибки такого рода почти не встречаются — гораздо чаще компилятор может сломать код именно в местах возникновения неопределенного поведения. Рассмотрим классический пример:

```
int main() {
    char x = 0;
    while (true) {
        if (x + 1 < x) break;
        x++;
    }
    std::cout << "Hello, World!\n";
}
```

Кажется, всё просто: в результате переполнения знакового типа `char` возникнет сравнение `-127 < 128`, цикл прервётся и выполнится вывод в стандартный поток. Так ведь?

Этот код, вероятнее всего, никогда не завершит свою работу.
При анализе поведения компиляторов вручную или с помощью сервисов вроде godbolt.org можно внезапно выяснить, что, например, GCC вообще не проводит проверку `if (x + 1 < x)`. Эта строка просто не компилируется, и всё. Clang, например, проверку провести попытается, но из цикла всё равно не выйдет.

Причина в том, что поведение, возникающие при переполнении знаковых типов, не определено. Компилятор может делать с неопределенным поведением что угодно, и это значит буквально что угодно. За этим особенно любопытно наблюдать при использовании мощных оптимизаций вроде `#pragma GCC optimization("Ofast")`, которые буквально сводят компилятор с ума.

Хорошая новость в том, что в стандарте C++26 неприятное обстоятельство с отсутствием гарантий насчёт кода с определённым поведением было упразднено, и теперь UB его не затрагивает. Вообще, 26-й стандарт принёс ряд изменений в процесс трансляции, которые не были рассмотрены по причине своей излишней техничности, однако особенно любопытным читателям советуется с ними ознакомиться.

Этот блок скорее напоминает о том, чем может обернуться неопределённое поведение и почему его лучше не допускать, чем практически касается самого процесса компиляции, однако это всё ещё достаточно важное знание, которое следует иметь в виду.

---

# Компоновка

## Разбиение программы на файлы

О включении одних исходных файлов в другие уже заходила речь при обсуждении директивы `#include`. При этом даже были затронуты определенные тонкости, связанные с процессом включения одних исходных файлов в другие, однако концептуальная сторона вопроса была проигнорирована.

При работе с большой кодовой базой вполне естественным кажется желание как-либо её структурировать. Хорошей идеей будет разбить код на логические модули, и при этом будет особенно здорово, если они при этом будут представлены отдельными файлами: часто хочется переиспользовать часть кода в рамках других проектов, и делать это наиболее удобно, если она обособлена от остальных; не говоря уже о том, что большие

файлы физически трудно редактировать, в то время как работать только с отдельными небольшими блоки кода удобно и эффективно.

Кажется, что для реализации этой идеи достаточно просто подключить все используемые файлы к основному исходнику, и задача будет решена: проект явным образом разбит на модули, и никакая информация из них при этом не утеряна. Тем не менее, здесь встаёт ряд других неочевидных проблем.

Такой подход драматически увеличивает размер компилируемого файла и, как следствие, время компиляции. Хуже того: при малейшем изменении в любом из модулей необходимо пересобирать *весь* проект, то есть мы не можем поддерживать и обновлять компоненты отдельно, хотя на практике это одна из основных целей, которые мы ставим при реализации модульной семантики. Мы также лишены возможности использовать предкомпилированные библиотеки или самостоятельно их создавать, вынося часть логики за пределы основного исполняемого файла. И хотя это не кажется чем-то критичным при работе над небольшими проектами, как только они разрастаются до тысяч или десятков тысяч строк, поддержка такого кода может стать непосильной задачей.

В этот момент на помощь приходят **заголовочные файлы**. Идея крайне простая: будем подключать не файлы с реализацией, а лишь наборы объявлений, которые использует программа. Это мгновенно решает проблему с компиляцией: размеры заголовочных файлов в десятки и в сотни раз меньше, чем размеры исходных; более того, теперь изменения самих модулей никак не сказываются на коде основной программы, а значит, что её больше не нужно постоянно пересобирать.
Другое важное преимущество использования заголовочных файлов состоит в том, что компиляция нескольких файлов может происходить *параллельно*, что также значительно ускоряет этот процесс. Стоит отметить, что такая махинация никак ему не помешает: компилятор увидит, что идентификатор ранее был объявлен, и посчитает обращение к нему корректной операцией.

Заголовочные файлы имеют расширение `.h` или `.hpp` и являются полноправными и независимыми единицами трансляциями.

Внимательному читателю вся эта затея, конечно, должна показаться совершенной глупостью — каким образом программа должна получить информацию о том, чем вообще является то или иное объявление, когда ни одно из определений не было ей непосредственно сообщено?
И это совершенно справедливое замечание: попытка, например, вызвать функцию, которая была объявлена, но не была определена, неизбежно приводит к ошибке. Однако эта ошибка, как уже было отмечено, происходит вовсе не на этапе компиляции, а на этапе *компоновки*, специально выделенном для разрешения проблем, связанных со ссылками на внешние файлы.

*Примечание 1*. Здесь и далее речь идёт о статических библиотеках и, соотвественно, статический линковке. Динамические библиотеки здесь рассмотрены не будут.

*Примечание 2*. В C++20 была привнесена концепция *модулей*, которые преподносятся как более современная, удобная и эффективная замена заголовочным файлам. Тем не менее, заголовочные файлы до сих пор используются повсеместно, поэтому модули далее рассмотрены не будут, однако особо любопытным читателям рекомендуется ознакомиться с их устройством.

## Компоновка как фаза трансляции

**Компоновка**, она же **линковка**, является девятой и последней фазой трансляции. На этой стадии специальная программа — как ни странно, компоновщик или, аналогично, линковщик (*linker*) — собирает воедино все объектные файлы, необходимые проекту, и формирует исполняемый файл. На практическом уровне задача линковщика, как уже было сказано, состоит в решении проблемы обращения к неопределенным идентификаторам.

### Таблица символов

Компилятор сохраняет всю информацию, ассоциированную с идентификаторами (области видимости, адреса, типы и проч.), в специальную структуру данных — **таблицу символов**, которая чаще всего представлена хеш-таблицей.

Таблица символов является частью объектного файла и может быть просмотрена путём вызова процедуры `nm` (в UNIX-подобных системах). Так, таблица страниц для объектного файла, полученного при компиляции следующего кода:

```
int Sum(int a, int b) {
    return a + b;
}

const int some_global_const = Sum(238, 30);
int some_global_var = Sum(some_global_const, 52);

int main() {
    int some_local_var = 3104;
    return 0;
}
```

при компиляции с помощью Clang версии 18.1.3 будет выглядеть следующим образом:

```
0000000000000000 t __cxx_global_var_init
0000000000000020 t __cxx_global_var_init.1
```

```
0000000000000040 t _GLOBAL__sub_I_main.cpp
0000000000000020 T main
0000000000000004 B some_global_var
0000000000000000 T Sum(int, int)
0000000000000000 b some_global_const
```

Она включает в себя записи как об объявленных в исходном коде, так и о сгенерированных компилятором идентификаторах. Примечательно, что запись о локальной переменной отсутствует — в этой ситуации компилятор её вообще не создавал.

Здесь первое значение показывает адрес идентификатора, второе — его тип (в данном случае оно указывает на сегмент памяти), третье — непосредственно его имя.
При этом адреса в таблице символов объектного файла могут совпадать, поскольку в этом случае они нумеруются не глобально, а относительно своих категорий.

При подключении любой сторонней библиотеки в таблицу добавится запись следующего вида:

```
                 U std::ios_base_library_init()
```

Здесь `U` означает «undefined». Как только линковщик встречает запись с такой пометкой, он пытается разрешить это обращение путём поиска соответствующего определения в других объектных файлов. Если этот поиск успешен, линковщик заменяет адрес в месте обращения на корректный; в противном случае происходит ошибка линковки.

## Декорирование имён

Если при вызове процедуры `nm` не указывать никаких дополнительных флагов, то, вероятнее всего, идентификаторы будут выглядеть примерно так:

```
0000000000000000 T _Z3Sumii
0000000000000000 b _ZL17some_global_const
```

Это явление называется **декорированием имён** (*name mangling*). Помимо того, что компилятор хранит сами имена объявленных переменных и функций, он также добавляет к ним произвольный объём дополнительной информации.

Это крайне полезно, например, при разрешении перегрузок. Так, функции

```
int Sum(int a, int b) {
    return a + b;
}
```

```
float Sum(float a, float b) {
    return a + b;
}
```

будут иметь следующие имена в таблице символов:

```
0000000000000020 T _Z3Sumff
0000000000000000 T _Z3Sumii
```

благодаря чему их можно различить друг от друга.

Методы декорирования отличаются от компилятора к компилятору, хотя это не влияет на его суть — оно всегда остаётся просто удобным способом хранить важную метаинформацию об идентификаторе сразу вместе со всеми остальными знаниями о нём.
Примечательно, что декорирование не поддерживается компиляторами Си, поскольку в нём просто нет перегрузок, пространств имён, классов, шаблонов и всего остального, для чего оно может понадобиться.

При просмотре таблицы символов с помощью процедуры `nm` декорирование можно отключить путём указания флага `--demangle`.

## Linkage

Понятие **связывания** (*linkage*) описывает, какие объявления будут видны между различными единицами трансляции. Связывание классифицируется следующим образом:

> **External linkage**
>
> Если имя обладает внешним связыванием (*external linkage*), к соответствующему ему объекту можно будет обращаться из различных областей видимости, в том числе других единиц трансляции.

> **Internal linkage**
>
> Если имя обладает внутренним связыванием (*internal linkage*), к соответствующему ему объекту можно будет обращаться из различных областей видимости внутри содержащей его единицы трансляции.

> **No linkage**

> Если имя не обладает связыванием, к соответствующему ему объекту нельзя обращаться из объемлющих областей видимости.

Связывание определяется компилятором согласно следующим правилам: [7]

- Анонимные пространства имён и пространства имён, вложенные в них, по умолчанию обладают внутренним связыванием. Все остальные пространства имён обладают внешним связыванием.
- Следующие объекты, если они находятся непосредственно в поле видимости пространства имён, имеют внутреннее связывание:
  - статические переменные, функции и шаблоны;
  - нешаблонные константные переменные;
  - члены анонимных юнионов.
- Если объект не был ранее наделён внутренним связыванием и при этом находится непосредственно в поле видимости пространства имён, он наделяется тем же связыванием, что и само пространство имён.
- Все остальные имена (в частности, все имена в поле видимости блока) не имеют связывания.

Ключевое слово `extern` позволяет вручную наделить любой объект внешним связыванием, т.е. сделать видимым для других единиц трансляции.

Все имена функций и переменных, имеющих внешнее связывание, наделяются *языковым связыванием*, которое совмещает в себе требования, необходимые для успешной линковки с единицами другого языка программирования: соглашение о вызове, декорирование имён, алгоритм и проч.. Можно предположить, что это понятие было введено из соображений обратной совместимости с языком Си, поэтому логично, что по умолчанию поддерживаются только два языковых связывания: C и C++. Компилятор позволяет контролировать языковое связывание при помощи указания `extern "C"` или `extern "C++"`. В частности, указание `extern "C"` отключает декорирование имён.

## Storage duration

Хотя на практике эта тема почти не касается основного предмета обсуждения, рассматриваемые далее спецификаторы традиционно относятся в ту же категорию спецификаторов, что и спецификатор `extern`.

Время жизни любой переменной ограничено — будь то временем работы программы или вычисления содержащего её выражения. В C++ выделяются следующие категории, связанные с этим понятием:

> **Static storage duration**

> Переменные со статическим временем жизни существуют всё время работы программы. Они входят в эту категорию, если:
>
> - относятся непосредственно к полю видимости пространства имён, и
> - не обладают потоковым временем жизни.
>
> Переменную можно наделить статическим временем жизни путём указания спецификатора `static` или `extern`.

> **Thread storage duration**
>
> Переменные с потоковым временем жизни существуют всё время выполнения одного потока. Начиная с C++11, переменную можно наделить потоковым временем жизни путём указания спецификатора `thread_local`.

> **Automatic storage duration**
>
> Переменная имеет автоматическое время жизни, если находится непосредственно в поле видимости блока или является параметром функции. До C++11 переменную можно было наделить автоматическим временем жизни путём указания спецификатора `auto`.

> **Dynamic storage duration**
>
> Динамическим временем жизни обладают все динамически объявленные объекты, т.е. объявленные через `new`, `malloc`, `memcpy` и проч.. В эту категорию также относятся исключения.

Помимо этого, до C++17 существовало ключевое слово `register`, которое указывало на частое переиспользование переменной и как бы «подсказывало» поместить её в регистры процессора. Это указание может быть проигнорировано.

---

### Источники


1. А. П. Хвастунов — Лекции по основам программирования на C++, 1 семестр, 2025
2. cppreference.com — Phases of translation (https://en.cppreference.com/w/cpp/language/translation_phases.html)
3. cppreference.com — Preprocessor (https://en.cppreference.com/w/cpp/preprocessor.html)
4. cppreference.com — Replacing text macros (https://en.cppreference.com/w/cpp/preprocessor/replace.html)
5. Stack Overflow — Why are preprocessor macros evil and what are the alternatives? (https://stackoverflow.com/questions/14041453/why-are-preprocessor-macros-evil-and-

what-are-the-alternatives)
6. cppreference.com — Implementation defined behavior control (https://en.cppreference.com/w/cpp/preprocessor/impl.html)
7. cppreference.com — Conditional inclusion (https://en.cppreference.com/w/cpp/preprocessor/conditional.html)
8. cppreference.com — Source file inclusion (https://en.cppreference.com/w/cpp/preprocessor/include.html)
9. Stack Overflow — Difference between angle bracket and double quotes while including header files in C++? (https://stackoverflow.com/questions/3162030/difference-between-angle-bracket-and-double-quotes-while-including-heade)
10. Википедия — Таблица символов (https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0_%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D0%BE%D0%B2)
11. Wikibooks — Linker (https://en.wikibooks.org/wiki/C%2B%2B_Programming/Programming_Languages/C%2B%2B/Code/Compiler/Linker)
12. Wikipedia — Name mangling (https://en.wikipedia.org/wiki/Name_mangling)
13. cppreference.com — Language linkage (https://en.cppreference.com/w/cpp/language/language_linkage.html)
14. ISO C++20 — §6.6 Program and linkage [basic.link] (https://isocpp.org/files/papers/N4860.pdf)

---

1. см. https://en.cppreference.com/w/cpp/language/charset.html (раздел Basic characters set) ↩
2. см. https://en.cppreference.com/w/cpp/language/operator_alternative.html ↩ ↩
3. см. https://en.cppreference.com/w/cpp/language/translation_phases.html (раздел Whitespace) ↩
4. см. https://en.cppreference.com/w/cpp/language/translation_phases.html (раздел Maximal munch) ↩
5. см. https://stackoverflow.com/questions/12447557/can-we-have-recursive-macros ↩
6. см. https://en.cppreference.com/w/cpp/language/string_literal.html (раздел Concatenation) ↩
7. Как обычно, приведённые критерии упрощены. Пожалейте автора. При желании самостоятельно изучайте стандарт. ↩