# Препроцессинг

Кориненко Матвей

15 сентября 2022 г.

## 1 Как запускается код на языках C и C++

Думаю, многие знают, что перед непосредственным запуском файла `main.cpp` или `main.c` происходят несколько этапов его обработки, и только после них программу можно запустить.

Давайте чуть более подробно рассмотрим этот процесс. Изначально мы просто пишем код на высокоуровневом языке программирования. Компьютер этот язык не понимает, поэтому программу необходимо перевести в более известный ему язык. Этим занимаются компилятор и транслятор (ассемблер). Первый из них переводит программу в Assembly-код, второй — этот код в язык нулей и единиц, который уже понятен процессору.

$$\text{Файл C/C++} \xRightarrow{\text{Компилятор}} \text{Assembly-код} \xRightarrow{\text{Транслятор}} 101001$$

На самом деле, в этой цепочке я упустил две важных детали. Первая — далеко не все современные проекты состоят из одного файла. А если это не так, то файлы с исходным кодом нужно правильно связать между собой перед сборкой всего проекта. Этим занимается Linker перед тем, как перевести файлы в язык нулей и единиц. Вторая деталь — иногда некоторые файлы нужно определенным образом подготовить к компиляции. Этим занимается препроцессор, после работы которого файлы можно спокойно компилировать.

$$\text{Файлы C/C++} \xRightarrow{\text{Препроцессор}} \text{Файлы C/C++} \xRightarrow{\text{Компилятор}} \text{Assembly-коды} \xRightarrow{\text{Linker}} \text{Файл проекта} \xRightarrow{\text{Транслятор}} 1011$$

Как вы поняли, нас будет интересовать самый первый этап обработки файла с исходным кодом.

## 2 Что такое препроцессор и чем он занимается

Говоря строгим языком Википедии, препроцессор — *это компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы. О данных на выходе препроцессора говорят, что они находятся в препроцессированной форме, пригодной для обработки последующими программами.*

Говоря простым языком, препроцессор — программа, которая занимается вставкой и подменой текста в исходном файле :) Но еще понятнее станет, когда мы разберем некоторые примеры директив для препроцессора. Возможно, некоторыми из них вы пользовались даже не задумываясь о их природе :)

# 3 Директивы для препроцессора в C/C++

- `#include "file.ext"` или `#include <file.ext>`

  Уверен, эту директиву Вы используете каждый раз, когда начинаете писать программу. На самом деле выполняет она предельно простую вещь — берет содержимое файла `file.ext` и вставляет в файл с исходным кодом.

  В файле `file.ext` Вы можете написать константы, создать структуры, ... А потом просто вставить его содержимое в основной файл с кодом и использовать в дальнейшем.

  Также, как Вы могли заметить, при написании этой директивы можно использовать как угловые кавычки, так и обычные двойные. Разница заключается в том, что при использовании угловых кавычек препроцессор будет искать импортируемый файл среди файлов библиотеки C/C++, а при использовании двойных — в директории проекта. К слову, в двойных кавычках можно указывать и просто абсолютный путь к файлу.

- `#define MacroId sequence_of_tokens`

  – Вероятно, эту директиву Вы тоже использовали. Благодаря ней на этапе препроцессинга всех вхождения фразы `MacroId` в исходный код будут заменены на `sequence_of_tokens`. Таким образом, например, можно *изменить синтаксис языка C/C++*:

  ```
  #define when if (
  #define then ) {
  #define end }
  ...
  when a > 0 then
      ...
  end
  ```

  – Также внутри макроса `MacroId` могут содержаться аргументы, которые можно использовать в `sequence_of_tokens`. Например, *можно* с помощью

  ```
  #define sum(a, b) a + b
  ```

  задать «функцию», возвращающую сумму двух ее аргументов.

  – Но при создании макросов с аргументами нужно быть очень аккуратным и не забывать, что препроцессор в случае директивы `#define` просто осуществляет подстановку одного текста на место другого и ничего не додумывает за программиста. Например, давайте попробуем использовать написанный ранее `#define sum` в одном из возможных сценариев.

  ```
  int a = sum(5, 6) * 3;
  ```

  *Внезапно*, в переменной `a` будет записано число 23, а не 33. Чтобы разобраться, почему это так, давайте выполним работу препроцессора (а именно — *втупую* сделаем замену текста) и посмотрим, какой код получится на выходе.

  ```
  int a = sum(5, 6) * 3; // ==> int a = 5 + 6 * 3;
  ```

  Теперь нам понятна проблема, и самое главное — понятно ее решение. В некоторых случаях, подобных этому, необходимо помещать sequence_of_tokens в скобки, чтобы все операции внутри них выполнились раньше, чем остальные.

  ```
  #define sum(a, b) (a + b)
  int a = sum(5, 6) * 3; // ==> int a = (5 + 6) * 3;
  ```

- `#ifdef, #if, #ifndef, #else, #endif`

  Внутри препроцессорного кода могут быть свои ветвления. Их можно использовать чтобы, например, получать разные результаты работы программ на разных устройствах, потому что для каждого девайса можно индивидуально установить свои compile_definitions, но об этом как-нибудь в другой раз :)

```
#ifdef something
// Эта часть кода выполнится, если на текущий момент something определено,
// например, если до этого была строка '#define something x'
#else
// Эта часть кода выполнится, соответственно, если something не определено.
#endif

#ifndef something
// Эта часть кода выполнится, если на текущий момент something не определено.
#else
// Эта часть кода выполнится, если something определено.
#endif

#if condition_1
// Эта часть кода выполнится, если condition_1 имеет значение true (как и в обычном if).
// Например, если на месте condition_1 написано '6 - 5 >= 0'.
#elif condition_2
// Препроцессор зайдет сюда, если все предыдущие condition вернули значение false,
// а condition_2 вернуло значение true.
#else
// Этот код выполнится, если condition во всех ветвлениях до этого имеет значение false.
#endif
```

- `#undef MacroId`

  Данная директива просто удаляет макрос MacroId для следующей части исходного когда. В дальнейшем его по желанию можно снова переопределить и использовать.

```
#define max(a, b) 5
#undef max
#define max(a, b) (a > b ? (a) : (b))
```

- `#pragma options`

  Данная директива задает некоторые дополнительные параметры для компиляции проекта.

  Например, с помощью `#pragma GCC poison identificator` можно полностью «удалить» какой-либо идентификатор при исполнении программы. Например, исполнение следующего кода

```
#include <stdio.h>

#pragma GCC poison printf

signed main() {
    printf("%d", 5);
}
```

  приведет к ошибке `error: attempt to use poisoned "printf"`.

  Есть множество других способов использовать `#pragma` в том числе и для ускорения исполнения некоторых фрагментов кода (про это можно почитать здесь https://codeforces.com/blog/entry/96344), но все это уже выглядит как тема для отдельной статьи :)