PREPROCESS

Минимум о препроцессоре. Предостережения о вреде его чрезмерного использования

К. Владимиров, Intel, 2019
mail-to: konstantin.vladimirov@gmail.com

Препроцессор

- Трансляция программы включает несколько фаз преобразований исходного текста, предшествующих синтаксическому разбору
- Отдельная программа, выполняющая эти фазы и только их называется препроцессором
- Любой компилятор можно запустить в режиме препроцессора, для gcc и clang это опция `-E`
- Программист может управлять препроцессором с помощью его директив обычно начинающихся с хештега #, исключённого из синтаксиса языка.
- Далее будут рассмотрены некоторые применения препроцессора

Условное исключение кода

- Некоторый код можно внести под опцию компиляции с помощью `#if`
- Такие опции подаются в gcc/clang через `-D`

```
#if defined(ENABLE_CHECKS)

int check_cfg () {
  // тут некий осмысленный код
}

#else

int check_cfg () { return 0; }

#endif
```

Условное исключение кода

```
#if !defined(CHECK_LEVEL)
  #define CHECK_LEVEL 0
#endif

#if (CHECK_LEVEL > 0)

int check_cfg () {
  printf("%d\n", CHECK_LEVEL);
}

#else

int check_cfg () { return 0; }

#endif
```

Условное исключение кода

```
#if !defined(CHECK_LEVEL)
  #define CHECK_LEVEL 0
#endif

#if (CHECK_LEVEL > 0)

int check_cfg () {
  printf("%d\n", CHECK_LEVEL);
}

#else

int check_cfg () { return 0; }

#endif
```

Интересный факт:

CHECK_LEVEL без параметров это 1

```
> g++ -DCHECK_LEVEL checks.cc
> a.exe
1
```

При этом CHECK_LEVEL может принимать только целые значения большие или равные нулю.

Обсуждение

- Как безусловно исключить некий участок кода, который хочется оставить на будущее?

Обсуждение

- Как безусловно исключить некий участок кода, который хочется оставить на будущее?

```
// TODO: implement check_cfg later

#if 0
int res = check_cfg();
#endif
```

- Варианты с "закомментировать" выглядят гораздо хуже. Лучше оставить комментарии для комментариев (т. е. текста)

```
// TODO: implement check_cfg later
// int res = check_cfg
```

Модульная структура

- Программа транслируется раздельно. Единица трансляции в C++ это файл.
- В каждой единице трансляции могут встречаться объявления и определения (функций, переменных, классов, etc.)

```
// fact.c

int fact (int x) { // определение
  return (x > 2) ? x * fact(x - 1) : x;
}

// main.c

int fact(int); // объявление

int main () { printf("%d\n", fact(5)); } // использование
```

Модульная структура

- Программа транслируется раздельно. Единица трансляции в C++ это файл.
- В каждой единице трансляции могут встречаться объявления и определения (функций, переменных, классов, etc.)
- Часто файлы кода распространяются с готовыми заголовочными файлами

```
// fact.h
int fact (int x); // объявление
// main.c
#include "fact.h"
int main () { printf("%d\n", fact(5)); } // использование
```

Обсуждение

- Самая простая программа:

```
#include <stdio.h>
int main () { printf("%s\n", "Hello"); }
```

- Как вы думаете сколько в ней окажется строк после препроцессирования?

Обсуждение

- Самая простая программа:

```
#include <stdio.h>

int main () { printf("%s\n", "Hello"); }
```

- Как вы думаете сколько в ней окажется строк после препроцессирования?
- В моей реализации:
  - `gcc 888 loc, 54155 bytes`
  - `clang 4297 loc, 128248 bytes`
- Речь о десятках килобайт текста

Стражи включения

- Разумеется с таким эффектом не хочется включать файлы лишний раз

```
// fact.h

#ifndef FACT_GUARD__
#define FACT_GUARD__

int fact (int);

#endif
```

- Эта идиома называется стражами включения

Стражи включения

- Разумеется с таким эффектом не хочется включать файлы лишний раз

```
// fact.h
#pragma once
int fact (int);
```

- Так проще и тоже работает на всех мейнстримных компиляторах

Константы времени компиляции

```
const int x = 1; // является ли x константой времени компиляции?
                 // в какой памяти выделено место для x?
                 // когда в эту ячейку записывается 1?

int y = x;       // когда происходит это присваивание?

int foo () {
  const int z = x; // сколько раз происходит это присваивание?
                   // в какой памяти выделено место для z?

  static const int w = z; // есть ли разница между w и z?
  ...
}
```

Константы времени компиляции

```
#define X 1          // является ли X константой времени компиляции?
                     // в какой памяти выделено место для X?
                     // когда в эту ячейку записывается 1?

enum { Y = 1 };      // является ли Y константой времени компиляции?
                     // в какой памяти выделено место для Y?
                     // когда в эту ячейку записывается 1?
```

- Что вы предпочтёте, `define` или `enum`?

Синонимы типов

- Ещё одно исторически сложившееся применения: синонимы типов имеет синтаксическую альтернативу: `typedef`

```
#define PINT int*

typedef int* pint_t;
```

- Обсуждение. Есть ли разница:

```
PINT a, b;

pint_t c, d;
```

Синонимы типов

- Ещё одно исторически сложившееся применения: синонимы типов имеет синтаксическую альтернативу: `typedef`

```
#define PINT int*

typedef int* pint_t;
```

- Обсуждение. Есть ли разница:

```
PINT a, b;   // int * a, b;

pint_t c, d; // int * c, * d;
```

Макросы

- "*Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer*" -- Bjarne Stroustrup

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

- Что тут плохо?

Макросы

- "*Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer*" -- Bjarne Stroustrup

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

- Что тут плохо? Тут плохо всё:

```
int c = MAX (x, y++); → c = x > y++ ? x : y++;
```

```
int d = MAX (foo(x), bar(y));
```

- Увы, в языке C это единственная возможность сделать обобщённый максимум. Вторая возможность – зайти через `void*`

Немного чёрной магии

```
#define CONCAT(A, B) A ## B
#define STRINGIFY(A) #A
```

Чёрная магия: мотивация

```
struct command {
  char *name;
  void (*function) (void);
};

struct command commands[] = {
  { "quit", quit_command },
  { "help", help_command },
  // ..... etc .....
};
```

- Тут надо постоянно писать руками пары "quit" и quit_command

Чёрная магия: исполнение

```
struct command {
  char *name;
  void (*function) (void);
};

#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] ={
  COMMAND (quit),
  COMMAND (help),
  // ..... etc .....
};
```

Макросы для типов

- "*Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer*" -- Bjarne Stroustrup

```
#define Pair (T1, T2) \
  struct Pair ## T1 ## T2 {
    T1 fst;
    T2 snd;
  };

#define Instance(C, T1, T2) C ## T1 ## T2

Pair(int, double);
Instance(Pair, int, double) p = {1, 1.0};
```

- Что тут плохо? Вопрос, в общем, риторический...

Обсуждение

- Что в итоге можно сказать о применении препроцессора? Когда оно нужно, когда излишне?

Традиционные области применения

- Использование по назначению
  - Условное исключение кода из компиляции
  - Модульная структура программы, включение хедеров
  - Стрингификация и конкатенация
- Сомнительные применения
  - Стражи включения
  - Константы времени компиляции
  - Синонимы типов
  - Короткие обобщённые функции

Порядок трансляции программы

1. Единица трансляции отображается в базовый набор символов, юникодные символы заменяются на \uXXXX
2. Конкатенируются строки, разбитые через \
3. Комментарии заменяются на пробельные символы
4. Файл разбивается на препроцессинговые токены
5. Исполняются директивы препроцессора (include, define, прагмы)
6. Заменяются escape-последовательности
7. Соединяются строковые литералы
8. Препроцессинговые токены становятся токенами, пробелы перестают иметь значение
9. Проводится синтаксический анализ и начинается инстанцирование шаблонов

Особенности раскрытия макросов

- Базовые правила:
  1. определение аргументов осуществляется сверху вниз в один проход
  2. раскрытие макросов происходит снизу вверх в один проход
- Потренируемся

```
#define h_h # ## #
#define mkstr(a) # a
#define betw(a) mkstr(a)
#define join(c, d) betw(c h_h d)

char p[] = join(x, y);
printf("%s\n", p); // что на экране?
```

Особенности раскрытия макросов

- Базовые правила:
  1. определение аргументов осуществляется сверху вниз в один проход
  2. раскрытие макросов происходит снизу вверх в один проход
- Потренируемся

```
#define h_h # ## #
#define mkstr(a) # a                  → "x ## y"
#define betw(a) mkstr(a)              → mkstr(x ## y)
#define join(c, d) betw(c h_h d)      → betw(x # ## # y)

char p[] = join(x, y);                → определены аргументы: x, y
printf("%s\n", p); // на экране "x ## y"
```

- Можно ли тут сделать так, чтобы прошла вторая конкатенация до xy?

Особенности раскрытия макросов

- Базовые правила:
  1. определение аргументов осуществляется сверху вниз в один проход
  2. раскрытие макросов происходит снизу вверх в один проход
  - ❖ следствие: макрос или команда, полученные в результате раскрытия, не раскрываются

```
#define h_h # ## #
#define mkstr(a) # a                  → "x ## y"
#define proxy(a) mkstr(a)             → mkstr(x ## y)
#define betw(a) proxy(a)              → proxy(x ## y)
#define join(c, d) betw(c h_h d)      → betw(x # ## # y)

char p[] = join(x, y);                → определены аргументы: x, y
printf("%s\n", p); // на экране то же, что и без proxy
```

Обсуждение

- Много ли мы теряем из-за такого однопроходного поведения препроцессора?
- Хотели бы вы сделать препроцессор рекурсивным, то есть проверяющим возможность идентификации аргументов после каждой точки их подстановки?

Искажение имён (задача)

```
#define VARIABLE 3
// .... some magic? ....
extern void NAME(mine)(char *x);
// creates mine_3 function if VARIABLE is 3
```

Искажение имён (решение)

```
#define VARIABLE 3
#define PASTER(x,y) x ## _ ## y
#define EVALUATOR(x,y)  PASTER(x,y)
#define NAME(fun) EVALUATOR(fun, VARIABLE)
extern void NAME(mine)(char *x);
// creates mine_3 function if VARIABLE is 3
```

Ещё раз о трансляции программы

1. Единица трансляции отображается в базовый набор символов, юникодные символы заменяются на \uXXXX
2. Конкатенируются строки, разбитые через \
3. Комментарии заменяются на пробельные символы
4. Файл разбивается на препроцессинговые токены
5. Исполняются директивы препроцессора (include, define, прагмы)
6. Заменяются escape-последовательности
7. Соединяются строковые литералы
8. Препроцессинговые токены становятся токенами, пробелы перестают иметь значение
9. Проводится синтаксический анализ и начинается инстанцирование шаблонов

"Традиционный" препроцессинг

Исследование препроцессора

- Сдампить поток токенов, идущий на синтаксический анализ, возможности нет.
- Но можно обработать программу, произведя в ней все текстовые замены и включения
- По традиции такой код называется препроцессированным

Литература

- [C11] ISO/IEC – "Information technology – Programming languages – C", 2011
- [K&R] Brian W. Kernighan, Dennis Ritchie – The C programming language, 1988
- [Linden] Peter van der Linden – Expert C Programming: Deep C Secrets , 1994
- [PP] C Preprocessor manual, https://gcc.gnu.org/onlinedocs/cpp/
- [Fultz] Paul Fultz – C Preprocessor tricks, tips, and idioms
- [Heathcote] Jonathan Heathcote – C Pre-Processor Magic