Лекция 21. Препроцессор и библиотеки языка Си.

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», весенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Содержание

- Функции: дополнительные сведения о функции m alloc ()
- Использование Си для представления разнообразных типов данных
- Новые алгоритмы и увеличение возможностей концептуальной разработки программ
- Абстрактные типы данных

Трансляция программы (1)

Язык Си построен на основе ключевых слов, выражений, операторов, а также правил их использования.

В нем также определено, что должен делать препроцессор, установлено, какие функции формируют стандартную библиотеку Си, и детализировано, каким образом работают эти функции.

Трансляция программы (2)

Препроцессор – анализирует программу до ее компиляции.

Следуя указанным директивам, препроцессор ***заменяет*** символические сокращения в программе сущностями, которые они представляют.

Препроцессор может включать другие файлы, и вы можете выбирать, какой код будет видеть компилятор. Препроцессору **ничего не известно о языке Си**, он преобразует один текст в другой.

Этот процесс и называется трансляцией программы.

Трансляция программы (3)

Этапы трансляции программы:

1. Устанавливает соответствие символов исходного кода с исходным набором символов (раздел VII, Приложение Б))
2. Обнаружение всех вхождений обратной косой черты (\). Заменить все комментарии на символ ‘ ’.
3. Поиск директив начинающихся с #

Трансляция программы (4)

Директива препроцессора начинается с символа #.

Может находится в любом месте файла исходного кода, и ее определение распространяется от этого места до конца файла.

Применение директив не ограничивается на создании символических или именованных констант, а также подключении файлов.

Трансляция программы (5)

Предполагается, что длина директивы ограничена одной строкой.

Однако комбинация обратной косой черты ‘/’ и символа новой строки ‘/n’ удаляются до начала обработки директив.

Тем самым объединённая строка образует одну логическую строку.

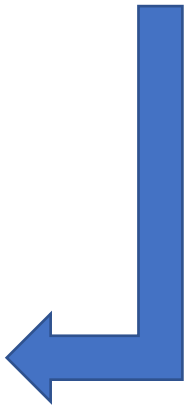
Директива #define (1)

Каждый строка `#define` состоит из:

`#define NAME_MACROS <список_замены/тело>`

Директива

Макрос



То на что будет заменён Макрос

Однако вместо *списка замены* или *тела*, может располагаться *лексема*.

Директива #define (2)

Формально *тело макроса* должно быть *строкой лексем*, а не строкой символов.

Лексемы препроцессора Си — это отдельные “слова” в *теле* определения *макроса*. Они отделяются друг от друга пробельными символами.

```
#define FOUR 2*2
#define SIX 2 * 3
#define EIGHT 4   *   8
```

Директива #define (3)

На уровне препроцессора можно *переопределить* константы. Однако доступность *переопределения* зависит от компилятора. В стандарте ANSI разрешено только такое новое *переопределение*, которое дублирует предыдущее.

```
#define SIX 2 * 3
#define SIX 2   *   3
#define SIX 2*3
```

Директива #define (4)

В `#define` разрешено использовать аргументы, тем самым декларируя *функциональные макросы*. Внешне такие макросы, напоминают *функции*. Вызов *функционального макроса* также поход на вызов *функции*.

```
#define SQUARE(X) X*X
int z = SQUARE(2);
//Значение z = 4
```

Директива #define (5)

Однако, не смотря на свою внешнюю схожесть, функциональные макросы работают не так как функции в языке Си.

```
x = 5
SQUARE(x)    //25   SQUARE (2)    //4
SQUARE(x+2) //17 100/SQUARE(2) //100
x = 5
SQUARE(++x) //42 После inc x = 7
```

Директива #define (6)

```
#define SQUARE(X) X*X
SQUARE(x+2)
//при x=5, 5+2*5+2 = 17
100/SQUARE(2)
// 100/2*2 = 100
SQUARE(++x)
//при x=5, ++5*++6 = 42
```

Директива #define (6)

Чтобы поместить *аргумент макроса* в строку ставят символ #, тогда *#аргумент* – имя *аргумента макроса*, а процесс назван *преобразование в строку.*

```
#define PSQR(x) printf("Квадрат " #x "
равен %d.\n", ((x)* (x)))
int y = 5;
PSQR(y); //Квадрат y равен 25
PSQR(2 + 4); //Квадрат 2 + 4 равен 36.
```

Директива #define (7)

Операция ## может применятся в заменяющей части функционального макроса, для объединения лексем.

```
#define XNAME(n) x ## n
#define PRINT_XN(n) printf("x" #n " =
                           %d\n", x ## n);
int XNAME(1) = 14; // int x1 = 14;
PRINT_XN(1); // printf("x1 = %d\n", x1)
;
```

Директива #define (8)

Макросы могут использовать переменное число аргументов для этого используется троеточие в аргументах `...` и `__VA_ARGS__` в теле макроса

```
#define PR(X, ...) printf("Сообщение "
                          #X " : " __VA_ARGS__)
PR(2, "x = %.2f, y = %.4f\n", x, y);
//Сообщение 2: x = 48.00, y = 6.9282
#define WRONG(X, ..., У) #X #__VA_ARGS__ #y
// Не будет работать.
```

Директива #define (9)

- Имя макроса не должно содержать пробелов, но пробелы допускаются в замещающей строке. В ANSI Си разрешены пробелы в списке агрементов.
- Заключайте в скобки каждый аргумент и определение в целом. Это гарантирует корректное группирование элементов в выражении следующего рода:

```
forks = 2 * MAX(guests+3, last);
```

Директива #define (10)

- Используйте **прописные буквы для имен функциональных макросов**. Данное соглашение не так широко распространено, как применение прописных букв в именах константных макросов. Тем не менее, одна из веских причин использования прописных букв связана с тем, что это напоминает вам о возможных побочных эффектах макросов.

Директива #define (11)

• Если вы намерены применять **макрос** для ускорения работы программы, сначала попытайтесь выяснить, обеспечит ли это **заметный выигрыш.** Макрос, который используется в программе **один раз, не приведет к значительному улучшению** скорости ее выполнения. Макрос, находящийся **внутри вложенного цикла**, является намного лучшим кандидатом для ускорения работы программы.

Директива #define (12)

```
#define ZXY   ALL_Z ALL_X ALL_Y
#define ALL_Z for(z=0;z<Lz;z++)
#define ALL_X for(x=(int)z%2;x<Lx;x++)
#define ALL_Y for(y=(int)z%2;y<Ly;y++)


#define ERRALLOC do{ \
fputs("Ошибка выделения памяти\n", \
stderr); exit(EXIT_FAILURE); } while(0)
```

Директива #include (1)

Когда процессор встречает `#include` он ищет файл с указанным в директиве именем и включает его содержимое в текущий файл.

`#include <stdio.h>` Поиск в системных каталогах

`#include "hot.h"` Поиск в текущем рабочем каталоге

`#include "/usr/biff/p.h"` Поиск в каталоге /usr/biff/

Файлы с расширением `.h` традиционно применяются для заголовочных файлов.

Директива #include (2)

Для чего применяется заголовочный файл:

- ***Символические константы***. В типичном файле `stdio.h`, к примеру, определен ы константы `EOF`, `NULL` и `BUFSIZ` (размер стандартного буфера ввода-вывода).
- ***Функциональные макросы***. Например , функция *getchar()* обычно определена как *getc(stdin)*. Заголовочный файл `ctype.h`, как правило, содержит определения макросов для функций *ctype*.

Директива #include (3)

Для чего применяется заголовочный файл:

- ***Объявления функций***. Заголовочный файл `string.h` например, содержит объявления для семейства функций обработки строк. Согласно ANSI Си и последующим стандартам, эти объявления представлены в виде прототипов функций.

Директива #include (4)

Для чего применяется заголовочный файл:

- ***Определения шаблонов структур***.
- ***Определения типов***. Вы можете вспомнить, что стандартные функции ввода/вывода применяют аргумент типа указателя на `FILE`. Обычно в файле stdio.h используется `#define` или `typedef` для того, чтобы имя `FILE` представляло указатель на структуру. Аналогично, в заголовочных файлах определены типы `size_t` и `time_t`.

Директива #undef (1)

Отменяет заданное определение `#define`

```
#define LIMIT 400
// LIMIT существует
#undef LIMIT
// LIMIT не существует
```

Область действия макроса `#define` начинается с места его объявления в файле и продолжается вплоть до соответствующей директивы `#undef`.

Директива #undef (2)

Кроме того, имейте в виду, что позиция директивы `#define` в файле будет зависеть от местоположения директивы `#include`, сели макрос поступает из заголовочного файла.

Существуют *предопределенные макросы*, такие как `__DATE__` и `__FILE__` (обсуждаются позже в главе), всегда считаются определенными, причем их определение не может быть отменено.

Директивы # ifdef, #else и #endif (1)

```
#ifdef MAVIS
    #include "horse.h"
    #define STABLES 5
#else
    #include "cow.h"
    #define STABLES 15
#endif
```

Директивы # ifdef, #else и #endif (2)

```
#if __STDC_VERSION__> 202311L
    #define VER "202311L"
#else
    #define VER "NOTLAST"
#endif
```

Аналогичная проверка с использованием “магических” переменных на версию стандарта языка Си при помощи препроцессора.

Директивы # ifndef (1)

Директива `#ifndef` может использоваться совместно с директивами `#else` и `#endif` так же как и `#ifdef`. Директива `#ifndef` представляет собой инверсию директивы `#ifdef`. Эта директива часто применяется для определения константы, если она еще не была определена.

```
#ifndef SIZE
    #define SIZE 100
#endif
```

Директивы # ifndef (2)

```
#ifndef THINGS_H_
    #define THINGS_H_
    /*ост. часть включаемого файла */
#endif
```

При появлении следующего включения того же самого файла идентификатор `THINGS_H_` уже определен, так что остальная часть файла пропускается.

Директивы # ifndef (3)

```
#ifndef THINGS_H_
    #define THINGS_H_
    /*ост. часть включаемого файла */
#endif
```

Предположим , что этот файл каким-то образом был включен несколько раз. Препроцессор встречает первое включение данного файла, идентификатор THINGS_H_ не определен, поэтому он определяется и обрабатывается остальная часть файла.

Директивы #ifndef (4)

Во избежание многократного включения в стандартных заголовочных файлах применяется директива `#ifndef`.

Обычно эту задачу решают путем использования имени файла в качестве идентификатора, записывая имя в верхнем регистре, заменяя точки символами подчеркивания и добавляя символ подчеркивания (или, возможно, два) в качестве префикса и суффикса.

```
#ifndef _STDIO_H
    #define _STDIO_H
    //содержимое файла
#endif
```

Директивы #if и #elif (1)

Директивы во многом похожа на обычный оператор if языка Си. В выражении могут применятся логические операторы или выражения.

```
#if SYS == 1
#include "ibm.h"
#endif
```

Директивы #if и #elif (2)

Для расширения функциональности применяется директива `#elif`

```
#if SYS == 1
    #include "ibmpc.h"
#elif SYS == 2
    #include "vax.h"
#else
    #include "general.h"
#endif
```

Директивы #if и #elif (3)

Здесь `defined` — операция препроцессора, которая возвращает значение 1, если ее аргумент определен с помощью директивы `#define` , и в противном случае.

```
#if defined (MAC)
    #include "ibmpc.h"
#else
    #include "general.h"
#endif
```

Предопределенные макросы (1)

В стандарте Си описано несколько предопределенных макросов, которые перечислены.

Макрос	Описание
`__DATE__`	Строка символов в форме “Ммм дд гггг”, представляющая дату обработки препроцессором, например, Aug 24 2014
`__FILE__`	Строка символов, представляющая имя текущего файла исходного кода
`__LINE__`	Целочисленная константа, представляющая номер строки в текущем файле исходного кода
`__STDC__`	Установлен в 1 для указания, что реализация соответствует стандарту C
`__STDC_HOSTED__`	Установлен в 1 для размещаемой среды; в противном случае — 0
`__STDC_VERSION__`	Для C99 установлен в 199901L; для C11 установлен в 201112L
`__TIME__`	Время трансляции в форме “чч:мм:сс”

Предопределенные макросы (2)

В стандарте Си описано несколько предопределенных макросов, которые перечислены.

`__func__` - предоставляет идентификатор переменной.

`__unix__` - Определяется, если платформа – UNIX.

`__linux__` - Определяется, если платформа – Linux.

Предопределенные макросы (3)

_WIN32 - Определяется, если платформа – Windows

_WIN64 - Определяется, если платформа – Windows 64bit.

__APPLE__ - Определяется, если платформа – macOS или iOS.

__x86_64__ - Определяется, если архитектура – x86-64.

__arm__ - Определяется, если архитектура – ARM.

Предопределенные макросы (4)

GCC (GNU *Compiler Collection*):

__GNUC__ - Версия GCC (основной номер).

__GNUC_MINOR__ - Минорная версия GCC.

__GNUC_PATCHLEVEL__ - Патч-уровень GCC.

Clang:

__clang__ - Если используется компилятор Clang.

__clang_major__

__clang_minor__

__clang_patchlevel__ - Версия Clang.

MSVC (Microsoft Visual C++):

_MSC_VER - Версия компилятора Microsoft Visual C++.

Директивы #line и #error (1)

Директива `#line` позволяет переустанавливать нумерацию строк и имя файла, выводимые с помощью макросов `__LINE__` и `__FILE__`

```
#line 1000
//переустанавливает текущий номер строки в 1000
#line 10 "cool.c"
//переустанавливает номер строки в 10, а имя файла - в cool.c
```

Директивы #line и #error (2)

Директива `#error` заставляет препроцессор выдать сообщение об ошибке, которое включает любой текст, указанный в директиве. Если это возможно, процесс компиляции должен приостановиться.

```
#if __STDC_VERSION__ != 201112L
#error NOT C11
#endif
```

Директивы #line и #error (3)

```
$ gcc ./main.c
./main.c:4:6: error: #error NOT C11
    4 |      #error NOT C11
      |       ^~~~~
$ gcc -std=c11 ./main.c
```

Директива #pragma (1)

Директива `#pragma` позволяет помещать инструкции для компилятора в исходный код.

```
#pragma c9x on
```

Каждый компилятор имеет *собственный набор указаний*. Они могут применяться, для управления объемом памяти, выделяемой под автоматические переменные, для установки уровня строгости при проверке ошибок или для включения нестандартных языковых средств и т. п.

Директива #pragma (2)

Кроме того, стандарт C99 поддерживает операцию препроцессора _Pragma.

```
Pragma("nonstandardtreatmenttypeB on")
```

является эквивалентом следующего указания:

```
#pragma nonstandardtreatmenttypeB on
```

Директива #pragma (3)

`#pragma once` – Используется для предотвращения повторного включения заголовочных файлов

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored
"-Wunused-variable"
int unused_variable = 42;
        // Предупреждение будет подавлено
#pragma GCC diagnostic pop
```

Директива #pragma (4)

```
#pragma pack(push, 1) // Установить выравнивание в 1 байт
struct pacStruct {
    char a;
    int b;
};
#pragma pack(pop) // Вернуть предыдущее значение выравнивания
```

Спасибо за внимание

Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», весенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)