Лекция 22. Препроцессор и библиотеки языка Си. Часть 2.

создал Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем
читает Насонова Алина Олеговна
ассистент кафедры вычислительных систем

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

Содержание

- Директивы препроцессора: #define, #include, #ifdef, #else, #endif, #ifndef, #if, #elif, #line, #error, #pragma
- Ключевые слова: _Generic, _Noreturn, _ Static _assert
- Функции/макросы:sqrt(), atan(), atan2(), exit(), atexit(), assert(), memcpy(), memmove(), va_start(), va_arg(), va_copy(), va_end()
- Дополнительные возможности препроцессора C
- Функциональные макросы и условная компиляция
- Встраиваемые функции
- Библиотека C и ее некоторые удобные функции

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.

Обобщенное программирование (1)

Термин *обобщенное программирование* относится к коду, который не является специфичным для конкретного типа, но после указания типа может транслироваться в код для этого типа.

Обобщенное программирование (2)

В языке C++ вводится понятие templates (шаблон).

```
template<typename Type>
Type max(Type a, Type b)
{
  return (a >= b ? a : b);
}
```

К сожалению, в Си нет ничего близко похожего на это.

Обобщенное программирование (3)

Однако в C11 появился новый вид выражения, называемого *выражением обобщенного выбора*, которое можно применять для выбора значения на основании типа выражения, т. е. основываясь на том, является ли типом выражения int, double и т.д.

Это не оператор препроцессора, но обычно оно используется как часть определения макроса `#define`, обладающего определенными чертами обобщенного программирования.

Обобщенное программирование (4)

Они получили более короткое название от английского “джей-не-ри-ки”.

```
#define GET_TYPE_ID(x)
    _Generic((x), \
        int: 0, \
        float: 1, \
        double: 2, \
        default: 3 \
    )
```

Обобщенное программирование (5)

```
int a = 5;
float b = 3.14f;
double c = 2.718;
char d = 'x';
printf("%d\n", GET_TYPE_ID(a)); // выведет 0
printf("%d\n", GET_TYPE_ID(b)); // выведет 1
printf("%d\n", GET_TYPE_ID(c)); // выведет 2
printf("%d\n", GET_TYPE_ID(d)); // выведет 3
```

Обобщенное программирование (6)

```
#define MAX(a, b) \
    _Generic((a), \
        int: ((a) > (b)) ? (a) : (b), \
        float: ((a) > (b)) ? (a) : (b), \
        double: ((a) > (b)) ? (a) : (b), \
        default: ((a) > (b)) ? (a) : (b) \
    )
```

Обобщенное программирование (7)

```
void free_int(void *p) { free(p); }
void free_double(void *p) { free(p); }

#define FREE(p) \
    _Generic(*(p), \
        int: free_int(p), \
        double: free_double(p) \
    )
```

Обобщенное программирование (8)

- Такой подход работает исключительно на уровне компиляции
- Можно использовать различные структуры и типы
- Позволяет создавать функционально полиморфные макросы.

Встраиваемые функции (1)

При вызове функции происходит переключение контексты, на что в значительной мере создает накладные расходы:

- подготовка вызова
- передача аргументов
- переход к коду функции
- возврат обратно.

Тем самым функциональные макросы позволяют избежать таких накладных расходов.

Встраиваемые функции (2)

Название *встраиваемые* никак не соответствует ожиданию. **Предполагается** что функция будет вызываться **настолько быстро, на сколько это возможно**, а *скорость* этого вызова *зависит от реализации.*

В стандарте говорится что функция с внутренним связыванием может быть встраиваемой, но она должна находиться в том же файле где и используется. К функции применяются спецификаторы `static` и `inline`.

Встраиваемые функции (3)

```
inline static void eatline() //
    встраиваемое определение/прототип
{
while (getchar() != '\n')
continue;
}
```

Встраиваемые функции (4)

Поскольку встраиваемая функция не имеет отдельного предназначенного для нее блока кода, получить ее адрес нельзя. (Но на самом деле можно)

Кроме того, встраиваемая функция может быть не видна в отладчике.

Встраиваемая функция должна быть короткой.

Встраиваемые функции (5)

Для проведения оптимизаций по встраиванию функции компилятору должно быть известно содержимое определения функции. Это означает, что определение встраиваемой функции должно находиться в том же файле, что и ее вызов. По данной причине встраиваемая функция обычно имеет внутреннее связывание.

Встраиваемые функции (6)

Для длинной функции время, затрачиваемое на ее вызов, невелико по сравнению со временем выполнения тела функции, поэтому использование встраиваемой функции не обеспечит существенной экономии времени.

Встраиваемая функция является исключением из правила, которое не рекомендует помещать исполняемый код в заголовочный файл. Так как имеет внутреннее связывание, ее определение в нескольких файлах не вызывает проблем.

Встраиваемые функции (7)

Следовательно, если программа состоит из нескольких файлов, встраиваемое определение понадобится поместить в каждый файл, который вызывает функцию.

Для достижения такого условия проще всего указать определение встраиваемой функции в заголовочном файле и затем включать этот файл в файлы, где функция применяется.

Функции _Noreturn (1)

В стандарт C11 был добавлен второй спецификатор функции, `_Noreturn`, предназначенный для указания функции, которая по завершении не возвращает управление вызывающей функции.

Примером функции `_Noreturn` является `exit();` после обращения к ней вызывающая функция никогда не возобновит свое выполнение.

```
_Noreturn void function_name();
```

Функции _Noreturn (2)

`_Noreturn` отличается от возвращаемого типа void. Типичная функция `void` возвращает управление вызывающей функции; не предоставляя какое-либо значение.

Цель `_Noreturn` заключается в том, чтобы проинформировать пользователя и компилятор, что конкретная функция не возвратит управление вызывающей программе. Это помогает предотвратить неправильное употребление функции, а указание на такой факт компилятору может сделать возможным и некоторые оптимизации кода.

Библиотека Си (1)

Библиотека математических функций (`math.h`)

`double log(double x)`	Возвращает натуральный логарифм x
`double log10(double x)`	Возвращает логарифм x по основанию 10
`double pow(double x, double y)`	Возвращает x в степени y
`double sqrt(double x)`	Возвращает квадратный корень x
`double cbrt(double x)`	Возвращает кубический корень x
`double ceil(double x)`	Возвращает наименьшее целое, которое не меньше x
`double fabs(double x)`	Возвращает абсолютное значение x
`double floor(double x)`	Возвращает наибольшее целое, которое не больше x

Библиотека Си (2)

Библиотека математических функций (`math.h`)

`double acos(double x)`	Возвращает угол (от 0 до π радиан), косинус которого равен x
`double asin(double x)`	Возвращает угол (от –π/2 до π/2 радиан), синус которого равен x
`double atan(double x)`	Возвращает угол (от –π/2 до π/2 радиан), тангенс которого равен x
`double atan2(double y, double x)`	Возвращает угол (от –π до π радиан), тангенс которого равен y/x
`double cos(double x)`	Возвращает косинус x (x в радианах)
`double sin(double x)`	Возвращает синус x (x в радианах)
`double tan(double x)`	Возвращает тангенс x (x в радианах)
`double exp(double x)`	Возвращает экспоненциальную функцию x (e^{x})

Библиотека Си (3)

Библиотека утилит общего назначения (`stdlib.h`)

```
rand(), srand(), malloc(), free()
```

(какие ещё помним ?)

Библиотека Си (4)

Библиотека утилит общего назначения (`stdlib.h`)

`exit() и atexit()`

Функция `exit()` неоднократно рассматривалась, она автоматически вызывается при возвращении из `main()`

Функция `atexit()` даёт возможность указать в качестве аргумента, функцию, после которой программа выполняет `exit()`

Библиотека Си (5)

Библиотека утилит общего назначения (`stdlib.h`)

`qsort()`

Метод быстрой сортировки входит в число наиболее эффективных алгоритмов сортировки, особенно в случае крупных массивов.
Разработанный Чарльзом Энтони

Ричардом Хоаром в 1962 году, этот алгоритм разделяет массивы на постоянно уменьшающиеся части, пока не будет достигнут уровень элемента.

Библиотека Си (6)

Библиотека утилит общего назначения (`stdlib.h`)

`qsort()`

Сначала массив делится на две части, так что любое значение в одной части меньше любого значения в другой части. Этот процесс продолжается вплоть до момента, когда массив станет полностью отсортированным.

Библиотека Си (7)

```
int values[] = { 40, 10, 100, 90, 20, 25 };
int compare (const void * a, const void * b)
{ return ( *(int*)a - *(int*)b ); }
int main (){
int n;
qsort (values, 6, sizeof(int), compare);
for (n=0; n<6; n++)printf ("%d ",values[n]);
return 0;}
```

Библиотека Си (8)

Библиотека утверждений (`assert.h`)

Макрос принимает в качестве аргумента целочисленное выражение. Если выражение оценивается как ложное (ненулевое), макрос `assert()` выводит в стандартный поток ошибок (stderr) сообщение об ошибке и вызывает функцию `abort()`, которая прекращает выполнение программы.

Библиотека Си (9)

Библиотека утверждений (`assert.h`)

Идея состоит в том, чтобы идентифицировать критические места в программе, где должны быть истинными определенные условия, и с помощью оператора `assert()` завершать программу, если одно из указанных условий нарушается. Обычно аргументом служит выражение отношения или логическое выражение.

Библиотека Си (10)

Библиотека утверждений (`assert.h`)

- Альтернативой функции `assert()` может является функция `_STATIC_ASSERT()`, которая осуществляет проверку на этапе компиляции, таким образом может стать причиной того, что программа не компилируется.

```
_STATIC_ASSERT(CHAR_BIT == 16, "Ошибочно предполагается 16-битовый тип char");
```

Библиотека Си (11)

Библиотека для строк и массивов (`string.h`)

- Присваивать один массив другому нельзя, поэтому в таких случаях мы применяли циклы для поэлементного копирования одного массива в другой. Единственное исключение состоит в том, что для символьных массивов мы использовали функции `strcpy()` и `strncpy()`. Однако сегодня речь пойдет о `memcpy()` и `memmove()`.

Библиотека Си (12)

Библиотека для строк и массивов (`string.h`)

```
void * memcpy( void *restrict d, const void *restrict s,
                                                size_t n);
void * memmove( void * d, const void * s, size_t n);
```

Обе функции копируют n байтов из области, на которую указывает аргумент s, в область, указанную аргументом d, и обе о ни возвращают значение d.

Библиотека Си (13)

Библиотека для строк и массивов (`string.h`)

```
void * memcpy( void *restrict d, const void *restrict s,
                                                size_t n);
void * memmove( void * d, const void * s, size_t n);
```

Различие между этими двумя функциями, как указывает ключевое слово `restrict`, связано с тем, что `memcpy()` разрешено полагать, что две области памяти нигде не перекрываются друг с другом.

Библиотека Си (14)

Библиотека для строк и массивов (`string.h`)

Функция `memmove()` не делает такого предположения, поэтому копирование происходит так, как будто все байты сначала помещаются во временный буфер и только затем копируются в область назначения.

Ч т о произойдет, если применить memcpy () к перекрывающимся областям?

Библиотека Си (15)

Библиотека для строк и массивов (`string.h`)

В этом случае поведение функции не определено, т.е. она может как работать, так и не работать. Компилятор не запрещает использование функции `memcpy()`, когда этого делать не следует, поэтому именно вы несете ответственность за обеспечение того, что области памяти не перекрываются.

Библиотека Си (16)

```
int target [10];
double curious[5] = {2.0, 2.0e5, 2.0e10,
                     2.0e20, 5.0e30};
memcpy(target, curious, (5) *
                     sizeof(double));
show_array(target, 10);
```

```
0 1073741824 0 1091070464 536870912
1108516959 2025163840 1143320349 -2012696540 1179618799
```

Библиотека Си (17)

Библиотека переменного числа аргументов (`stdarg.h`)

Ранее обсуждалась возможность создания функционального макроса, который позволял передавать различное (переменное) число аргументов, т.к. макросы работаю на уровне компиляции, реализация такой возможности в программе выглядит значительно сложнее. Помогает библиотека `stdarg.h`

Библиотека Си (18)

Библиотека переменного числа аргументов (stdarg.h)

1. Подготовить прототип функции, в котором применяется троеточие.

```
void f1(int n, ...); //допустимо
int f2(const char *s, int k, ...); //допустимо
char f3(char c1, ..., char c2); //недопустимо, троеточие не в конце
double f3 (...) ;//недопустимо, параметры отсутствуют
```

Библиотека Си (19)

Библиотека переменного числа аргументов (`stdarg.h`)

2. Создать в определении функции переменную типа `va_list` .

```
double f1(int n,...)
{
va_list ap; // объявление объекта для
хранения аргументов
```

Библиотека Си (20)

Библиотека переменного числа аргументов (`stdarg.h`)

3. Использовать макрос для инициализации этой переменной списком аргументов.

```
va_start(ap, n);
```

Библиотека Си (21)

Библиотека переменного числа аргументов (stdarg.h)

4. Применить макрос для доступа к списку аргументов.

```
double tic; int toc;
tic = va_arg(ap, double); // извлечение первого аргумента
toc = va_arg(ap, int); // извлечение второго аргумента
```

Библиотека Си (22)

Библиотека переменного числа аргументов (`stdarg.h`)

5. Использовать макрос для очистки.

```
va_end(ap);
```

Библиотека Си (23)

```
double sum(int count, ...) {
va_list args;double total = 0.0;
va_start(args, count);
for(int i = 0; i < count; i++)
    total += va_arg(args, double);
va_end(args);
return total;
}
```

Библиотека Си (24)

```
printf("Sum: %f\n", sum(3, 1.5, 2.3,
                                3.7));
printf("Sum: %f\n", sum(5, 10.0, 20.5,
                    30.25, 40.75, 50.0));


Sum: 7.500000
Sum: 151.500000
```

Спасибо за внимание

Насонова Алина Олеговна
ассистент кафедры вычислительных систем

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