

Компиляция программы. Препроцессор.

Основные этапы компиляции программы (hello.c → a.out)

- **Разбиение на лексемы:** исходный файл разбивается на «слова» (tokens) и пробелы
- **Выполнение директив препроцессора:** присоединяются заголовочный файлы, развёртываются макросы
- **Трансляция:** производится синтаксический и семантический анализ, текст преобразуется в объектный файл
- **Компоновка (link):** один или несколько объектных файлов объединяются, добавляются функции внешних библиотек, создается выполняемый файл

Вызов препроцессора (для отладки)

```
gcc -E hello.c
```

Директивы препроцессора

Список директив

<code>#define</code>	<code>#endif</code>	<code>#ifdef</code>	<code>#line</code>	<code>#elif</code>	<code>#error</code>
<code>#ifndef</code>	<code>#pragma</code>	<code>#else</code>	<code>#if</code>	<code>#include</code>	<code>#undef</code>

Основные правила


- Каждая директива должна находиться на отдельной строке
первый символ команды `#`
- Если директива не помещается в одной строке,
её можно продолжить поставив в конец строки обратную косую черту
`” \ ”`

Вставка файла

Директива `#include "file"`

- Вставляет `file` на место строки с `#include`
- В `file` может содержаться вложенный `#include` и так далее
- Имя файла должно быть заключено либо в двойные кавычки `"file"` или в угловые скобки `<file>`

`gcc -E hello.c` → 841 строка

 угловые скобки принято использовать для стандартных заголовочных файлов, а кавычки для файлов относящихся к конкретной программе:
“если файл был указан в двойных кавычках и поиск закончился неудачно, то эта директива переобрабатывается как если бы файл был заключен в угловые скобки”

Макроопределения (макросы)

Директива #define

#define ИМЯ_МАКРОСА последовательность_символов

- Осуществляется **текстовая подстановка**:
ИМЯ_МАКРОСА → последовательность_символов
- Признаком конца директивы является конец строки
- Если ИМЯ_МАКРОСА внутри кавычек, то замены не будет

Именованные константы

```
#define MAX_SIZE 100
#define MIN_SIZE MAX_SIZE / 2
#define LONG_STRING "this is a very long " \
                    "string used as an example\n"
#define TEST this is test
```

```
double array[MAX_SIZE];
for(i = MIN_SIZE; i < MAX_SIZE; i++) ...
printf(LONG_STRING);
printf("TEST\n");
```

«Output: gcc -E »

```
double array[100];
for(i = 100 / 2; i < 100; i++) ...
printf("this is a very long " "string used as an example\n");
printf("TEST\n");
```

Часто используемые predefined макросы

- `__DATE__` и `__TIME__` макросы содержащие дату и время вызова препроцессора (C-строки из 11 символов)
- `__FILE__` имя текущего обрабатываемого файла (C-строка)
- `__LINE__` номер строки в этом файле (целое число)
- ~~`__FUNCTION__` имя текущей функции (C-строка) (obsolete)~~

новое в стандарте C99

- `__func__` – переменная содержащая имя выполняемой функции:
`static const char __func__[] = "function name";`

Пример: отладочная печать

```
printf(" Error ... at %s, line %d.\n", __FILE__, __LINE__);  
printf("I am in %s function\n", __func__);
```

Макроподстановки: макросы с параметрами

Макросы с формальными параметрами

```
#define ABS(x)    ((x) < 0 ? -(x) : (x))  
#define MIN(x,y) (((x)<(y)) ?  (x) : (y))
```

Пример

```
result1 = ABS(a);  
result2 = MIN(a,b);
```

«Output: gcc -E »

```
result1 = ((a) < 0 ? -(a) : (a));  
result2 = (((a)<(b)) ? (a) : (b));
```

Правила написания

- 👉 Нельзя ставить пробел между именем макроса и скобками
- 👉 Лишняя скобка не повредит!

Плохая идея

```
#define SQR(x)      x*x    // ERROR  
    result = SQR(a+b);
```

«Output: gcc -E »

```
result = a+b*a+b;
```


Плохая идея

```
#define SQR(x)      x*x    // ERROR  
result = SQR(a+b);
```

«Output: gcc -E »

```
result = a+b*a+b;
```

Плохо продуманная идея

```
#define SQR(x)      (x)*(x) // ERROR  
result = 1./SQR(a);
```

«Output: gcc -E »

```
result = 1./(a)*(a);
```

Плохая идея

```
#define SQR(x)      x*x    // ERROR  
result = SQR(a+b);
```

«Output: gcc -E »

```
result = a+b*a+b;
```

Плохо продуманная идея

```
#define SQR(x)      (x)*(x) // ERROR  
result = 1./SQR(a);
```

«Output: gcc -E »

```
result = 1./(a)*(a);
```

Правильный вариант

```
#define SQR(x)      ((x)*(x))    // OK!
```

☞ Макросы не функции!

_____ тестируем макрос _____

```
#define MIN(x,y) (((x)<(y)) ? (x) : (y))
```

```
a = 1;
```

```
b = 10;
```

```
result = MIN(a++,b);
```

```
printf(" a= %i, b= %i, result= %i\n",a,b,result);
```

```
Result> a= 3, b= 10, result= 2
```

_____ «Output: gcc -E » _____

```
result = (((a++)<(b)) ? (a++) : (b));
```

Директива #undef имя_макроса

- «удаляет» ранее определенный макрос

Используйте inline functions вместо «функций-макросов»

```
#undef MIN    // удаляем макрос MIN
```

```
inline int MIN(int x,int y) { // добавляем функцию MIN()
    return (x < y) ? x : y;
}
```

```
a = 1;
```

```
b = 10;
```

```
result = MIN(a++,b); // this is now a function
```

```
printf(" a= %i, b= %i, result= %i\n",a,b,result);
```

```
Result> a= 2, b= 10, result= 1
```

Условная компиляция: if - else

Директивы #if, #else, #elif и #endif

```
#if константное выражение 1
    «включается» если выражение 1 истинно
#elif константное выражение 2
    иначе, если выражение 2 истинно
#else
    ...
#endif
```

👉 Промежуточные #else и #elif необязательны

```
#if MAX_SIZE>99
    printf("If size of the array is 100 and more\n");
#else
    printf("The case of a small array.\n");
#endif
```

Директивы #ifdef и #ifndef

```
#ifdef ИМЯ_МАКРОСА
```

«включается» если ИМЯ_МАКРОСА определено

```
#endif
```

```
#ifndef ИМЯ_МАКРОСА
```

«включается» если ИМЯ_МАКРОСА не определено

```
#endif
```

```
#ifdef DEBUG
```

```
    printf(" x=..."); // отладочная печать, если определен DEBUG
```

```
#endif
```

```
#ifdef ABRAKADABRA
```

```
    что-то, что мы хотим временно закомментировать
```

```
#endif
```

#ifndef в заголовочных файлах

В заголовочных файлах рекомендуется использовать конструкцию:

```
/* example.h */  
#ifndef EXAMPLE_H  
#define EXAMPLE_H  
...  
#endif
```

- ✓ Имя макроса проверяется и если оно не определено тут же определяется
- ✓ Имя макроса должно быть уникально для каждого файла, обычно конструируется из имени заголовочного файла

👉 Многократная вставка такого заголовочного файла безопасна

```
/* example.c */  
#include "example.h"  
#include "example.h"    // безопасно
```

Операторы # и

Используются внутри макросов

- # – создает C-строку из аргумента перед которым стоит
- ## – объединяет (склеивает) две лексемы

Пример с

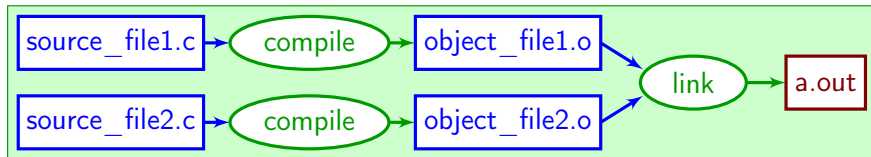
```
#define showtype(t) printf("sizeof(%s) = %zu\n",#t,sizeof(t))  
showtype(long double); // sizeof(long double) = 16
```

Пример с

```
#define INT3(x) int x##1, x##2, x##3  
INT3(a); // => int a1, a2, a3;  
a1 = 1; a2 = a1++; a3 = a2++;  
printf("a1= %i, a2= %i, a3= %i\n",a1,a2,a3); // a1= 2, a2= 2, a3= 1
```


Раздельная компиляция

- независимая трансляция частей программы разделенной на отдельные файлы с последующим объединением их компоновщиком в один исполняемый файл:



Цели и преимущества

- разделение большой программы на небольшие, более понятный части
- отладка каждой из частей делается независимо
- изменениях в одной из частей ведет к компиляции только этой части

main.c

```
#include "functions.h"

int main() {
    fun_debug = 1; // debug mod
    init();
    run(10);
    stop();
}
```

functions.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H
extern int fun_debug; // external variable
void init();
int run(int nevents);
void stop();
#endif
```

functions.c

```
#include <stdio.h>
#include "functions.h"

// debug is off by default
int fun_debug = 0;

// invisible outside this file
static int init_done = 0;

void init() {
    if( init_done ) return;
    if( fun_debug ) {
        printf("This is init()\n");
    }
    init_done = 1;
}
```

```
int run(int nevents) {
    if( fun_debug )
        printf("This is run()\n");
    for(int i=0; i<nevents; ++i) {
        if( fun_debug )
            printf(" event# %i\n",i);
    }
    return 1;
}

void stop() {
    if( fun_debug )
        printf("This is stop()\n");
    init_done = 0;
}
```

Компиляция

- За один заход:

```
gcc main.c functions.c
```

- Поэтапно:

```
gcc -c main.c
```

```
gcc -c functions.c
```

```
gcc main.o functions.o
```

Output:

```
This is init() function
```

```
This is run() function
```

```
event# 0
```

```
event# 1
```

```
...
```

```
event# 9
```

```
This is stop() function
```

Спецификатор `extern`

- спецификатор `extern` используется чтобы *декларировать* переменную или функцию, определённую в другом файле
- все функции по умолчанию `extern`
- определение функции или переменной (выделение памяти) должна выполняться только в одном месте, один раз
- проверка, что переменная или функция действительно существуют выполняется компоновщиком

Note: сравните со `static`

- 👉 `static` для глобальной переменной или функции ограничивает их область видимости единицей трансляции («файлом») в котором они определены

Библиотеки языка C

- Библиотека – набор объектных файлов объединенных в один файл
- Прототипы библиотечных функций перечислены в заголовочном файле:
`#include <header_for_library>`
- На стадии компоновки нужные библиотечные функции «добавляются»

«Подключение» библиотеки

- `cc prog.c -lm`, где `-l`: ключ транслятора и `m` – «короткое имя» библиотеки
- полное имя библиотечного файла: `libm.a`; короткое получается без «малозначащих частей»: `lib + короткое имя + {.a или .so}`
- можно подключить библиотеку указав полный путь:
`cc prog.c /usr/lib/libm.a`

Стандартная библиотека C: variadic functions

Функции с переменным числом аргументов <stdarg.h>

- Обычная функция имеет фиксированное число аргументов
- Вариативные функции позволяют работать с произвольным, заранее неизвестным, числом аргументов: например `printf`, `scanf`
- Декларируются с помощью «многоточия» (три точки) в списке аргументов:

```
double average(int n, ...); // average of 'n' arguments
```

- Вызов не отличается от вызова обычных функций:

```
double x2 = average(2, 4.,5.);           // average of 2 numbers  
double x5 = average(5, 4.,5.,-1.,2.,3.); // average of 5 numbers
```

- Программируются с помощью макросов, определенных в <stdarg.h>:
`va_list`, `va_start`, `va_arg`, `va_end` (далее пример)

Пример вариативной функции:

```
#include <stdio.h>
#include <stdarg.h> // macros: va_list, va_start, va_arg, va_end
double average(int n, ...) {
    va_list va; // holds the information for other macros
    va_start(va,n); // to get the address first "variadic arg"
    double sum = 0;
    for(int i = 0; i < n; i++) {
        sum += va_arg(va,double); // accesses the next arg
    }
    va_end(va); // ends traversal of args
    return sum/n;
}

int main() {
    double x2 = average(2, 4.,5.);
    printf(" x2= %f\n",x2);          // x2= 4.500000
    double x5 = average(5, 4.,5.,-1.,2.,3.);
    printf(" x5= %f\n",x5);          // x5= 2.600000
}
```


Обратите внимание

- Должен быть хотя бы один фиксированный аргумент, а многоточие всегда последний аргумент в функции
 - 👉 в примере: `average(int n, ...);`
- Число вариативных аргументов обычно передается как фиксированный аргумент
- Если тип аргументов заранее неизвестен, он должен передаваться либо через фиксированные аргументы, либо в самом списке
- Соответствие реального и ожидаемого числа аргументов и их типов — задача программиста:

👉 в примере:

```
double x2 = average(2, 4.,5.); // after 4 and 5 must be dots
average(2, 4,5); // 0.000000
average(3, 4.,5.); // 3.000000
```

Дополнительные слайды

- `_Static_assert(int_expr, message)`

Проверка «логического» выражения **во время компиляции** и если выражение равно нулю («ложно») остановка компиляции с выводом сообщения `message`

- `static_assert(int_expr, message)`

Макрос, определенный в заголовочном файле `<assert.h>`, «для удобства»

👉 в C++11 имеется `static_assert(bool_constexpr, message)` работающая похожим образом

Пример: проверка, что `long` ровно 8 байт

```
static_assert(sizeof(long) == 8, "long int must be exactly 8 bytes");  
// возможная ошибка во время компиляции:  
// error: static assertion failed: "long int must be exactly 8 bytes"
```

Новое ключевое слово `_Generic`

- Позволяет во время компиляции сделать выбор на основе **типа переменной**
- Синтаксис похож на `switch`, проверяется **тип** первой переменной а далее идут пары: имя типа (или `default`) и результат

👉 Отсутствует в C++

Пример

```
#define type_idx(T) \  
    _Generic( (T), int: 1, double: 2, char*: 3, default: 0)  
  
printf("type_idx= %i\n",type_idx(1));           // type_idx= 1  
printf("type_idx= %i\n",type_idx(1.1));         // type_idx= 2  
printf("type_idx= %i\n",type_idx("1.1"));       // type_idx= 3  
printf("type_idx= %i\n",type_idx(1L));          // type_idx= 0
```

«универсальный» print для целых типов

```
#define PRTFMT(x) _Generic( (x), \  
    signed int: "%i",\  
    unsigned int: "%u",\  
    long int: "%ld",\  
    unsigned long int: "%lu",\  
    long long int: "%lld",\  
    unsigned long long int: "%llu")  
#define PRTLN(x) printf("%s= ",#x);\  
    printf(PRTFMT(x),x);\  
    printf("\n")
```

```
size_t t = 10;  
PRTLN(t); // t= 10
```

```
double x = 5.;  
PRTLN(x); // COMPILATION ERROR: selector of type 'double'  
          // is not compatible with any association
```