

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

#include	#define	#undef	#if	#ifdef	#ifndef	
#else	#elif	#endif	#line	#error	#pragma	
#warning	#elifdef	#elifndef				(C23, C++23)

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    // идентично #if defined(ИМЯ_МАКРОСА)
    «включается» если ИМЯ_МАКРОСА определено
#endif
#ifndef ИМЯ_МАКРОСА   // идентично #if !defined(ИМЯ_МАКРОСА)
    «включается» если ИМЯ_МАКРОСА не определено
#endif
```

- #elifdef ИМЯ_МАКРОСА ≡ #elif defined ИМЯ_МАКРОСА (C23, C++23)
- #elifndef ИМЯ_МАКРОСА ≡ #elif !defined ИМЯ_МАКРОСА (C23, C++23)

```
#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
```


Проверка (assertion) при компиляции

`static_assert(int_expr, message)`

since C11, keyword in C23

- Определено в `<assert.h>`: проверка «логического» выражения во время компиляции и если выражение равно нулю (ложно) остановка компиляции с выводом сообщения `message`

👉 в C++11 имеется похожая декларация:

```
static_assert(bool_constexpr, message)
```

Пример: проверка, что `long` занимает 8 байт

```
#include <assert.h>
```

```
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"
```

Проверка доступности «заголовочного» файла

`__has_include("filename")`

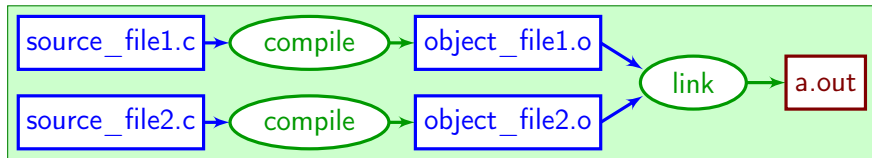
C23, C++17

- Препроцессорный оператор позволяющий проверить, что `filename` доступен для использования в директиве `#include`

```
#if __has_include("fun_prt.cc")
#include "fun_prt.cc"
#else
#define EMPTY_FUN_PRT
void fun_prt() { printf("Empty function\n"); }
#endif
int main() {
    fun_prt();
#ifdef EMPTY_FUN_PRT
    FILE* f = fopen("fun_prt.cc", "w");
    fprintf(f, "void fun_prt() { printf(\"Good function\\n\"); }\n");
    fclose(f);
#endif
}
```

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
- изменения 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"

// definition of fun_debug
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;
}
```

Спецификатор `extern`

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

Note: сравните `co static`

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

Компиляция примера

- 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
```

Введение в make и Makefile

Автоматизация раздельной компиляции с make

- Существует несколько версий `make`: очень популярна `gmake` (GNU make)

Очень простой пример файла Makefile

```
PROGRAM := test                # variable

all: $(PROGRAM)                # target-1: component

$(PROGRAM): functions.o main.o # target-2: component
    @echo "Hello make, example" # TAB before the command
    $(CC) -o $(PROGRAM) functions.o main.o
```

> make [target]

```
cc      -c -o functions.o functions.c
cc      -c -o main.o main.c
Hello make, example
cc -o test                functions.o main.o
```


Чуть более сложный пример Makefile

```
PROGRAM := test                # program name
SRCS     := functions.c main.c # source files
OBJS     := $(SRCS:.c=.o)      # transform filenames .c -> .o
CFLAGS   := -Wall -I.          # compilation flags
LIBS     := -lm                # options for linking

# add GSL-library
GSLCFLAGS := $(shell gsl-config --cflags) # shell-command
CFLAGS    += $(GSLCFLAGS)
GSLLIBS   := $(shell gsl-config --libs)   # shell-command
LIBS      += $(GSLLIBS)

%.o : %.c                      # suffix rule
    $(CC) -c $(CFLAGS) $<

all:    $(PROGRAM)             # target-1

# see next slide
```

... продолжение

```
$(PROGRAM): $(OBJS)                # target-2
    $(CC) $(CFLAGS) $(OBJS) $(LIBS) -o $(PROGRAM)
    @echo "done, $(PROGRAM)"

clean:                               # target-3
    @rm -f *.o $(PROGRAM)

depend:                             # target-4
    @echo "Generating make.depends"
    @rm -f make.depends
    $(CC) -MM $(CFLAGS) $(SRCS) > make.depends
    @echo "done"

include $(wildcard *.depends)        # include dependency files
```

- > **make depend** создания файла «зависимостей»
- > **make** сборка программы
- > **make clean** «очистка»

Введение в cmake и CMakeLists.txt

Автоматизация создания Makefile

- Утилита **cmake** создает из файла сценария **CMakeLists.txt** файлы для сборки с помощью **make**, **MVS**, **Android Studio** ...

Очень простой пример файла CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED( VERSION 2.4 )  
PROJECT(test)  
INCLUDE_DIRECTORIES($PROJECT_SOURCE_DIR)  
ADD_EXECUTABLE(test functions.c main.c)
```

Как использовать:

```
# создаем папку для Makefiles в которой будем собирать проект  
> mkdir build  
> cd build  
> cmake ../          # папка где лежит CMakeLists.txt  
> make
```

Дополнительные слайды

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
```