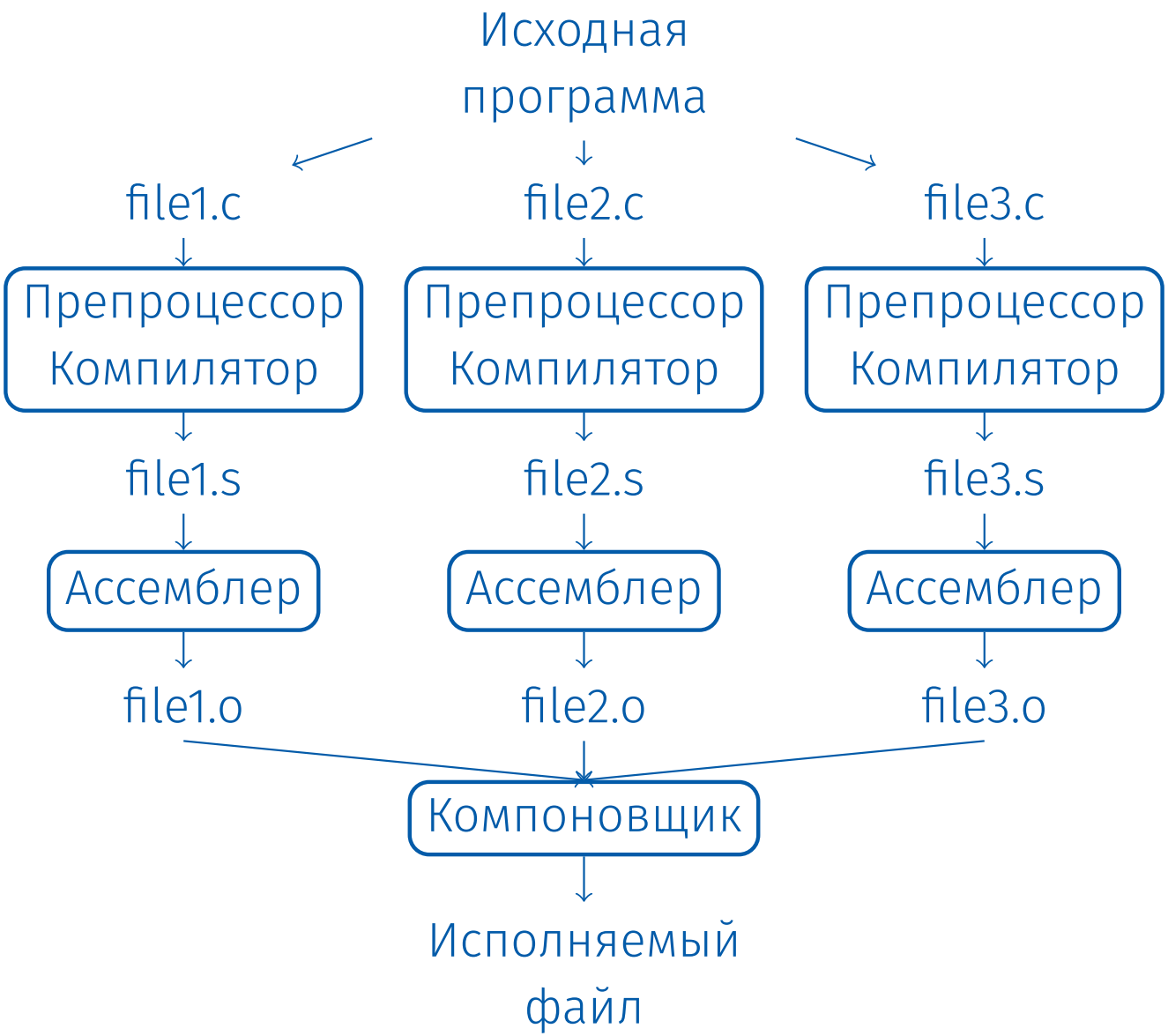
Московский государственный университет им. М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Алгоритмы и алгоритмические языки
Лекция 13

23 октября 2019 г.

Схема раздельной компиляции



Препроцессор

Перед компиляцией выполняется этап *препроцессирования*. Это обработка программного модуля для получения его окончательного текста, который отдается компилятору.

Управление препроцессированием выполняется с помощью *директив препроцессора*:

```
#include <...>  //  системные библиотеки
#include "..."  //  пользовательские файлы
#define name(parameters) text
#undef name

#define MAX 128
#define ABS(x) ((x) >= 0 ? (x) : -(x))

x -> y - 7
ABS(x) -> ((y - 7) >= 0 ? (y - 7) : -(y - 7))

x -> a-- ?
```

Препроцессор

Препроцессор позволяет организовать условное включение фрагментов кода в программу.

```
#ifdef name / #endif  -- проверка определения имени

#ifndef _STDIO_H
#define _STDIO_H
 <... текст файла ...>
#endif
```

Препроцессор

Препроцессор позволяет организовать условное включение фрагментов кода в программу.

#if/#if defined/#elif/#else/#endif --- общие проверки условий

```
#if HOST_BITS_PER_INT >= 32
typedef unsigned int gfc_char_t;
#elif HOST_BITS_PER_LONG >= 32
typedef unsigned long gfc_char_t;
#elif defined(HAVE_LONG_LONG)
      && (HOST_BITS_PER_LONGLONG >= 32)
typedef unsigned long long gfc_char_t;
#else
#error "Cannot find an integer type with at least 32 bits"
#endif
```

Препроцессор: операции # и

Операция # позволяет получить строковое представление аргумента.

```
#define FAIL(op)                                                \
  do {                                                          \
    fprintf (stderr, "Operation␣" #op "failed:␣" \
             "at␣file␣%s,␣line␣%d\n", __FILE__,  \
             __LINE__);                                         \
    abort ();                                                   \
  } while (0)

int foo (int x, int y) {
  if (y == 0)
    FAIL (division);
  return x / y;
}
```

```
do { fprintf (stderr, "Operation␣" "division" "failed:␣"
"at␣file␣%s,␣line␣%d\n", "fail.c", 13); abort (); } while (0);
```

Препроцессор: операции # и

Операция ## позволяет объединить фактические аргументы макроса в одну строку.

```
java-opcodes.h:
enum java_opcode {
#define JAVAOP(NAME, CODE, KIND, TYPE, VALUE) \
        OPCODE_##NAME = CODE,
#include "javaop.def"
#undef JAVAOP
LAST_AND_UNUSED_JAVA_OPCODE
};
```

```
javaop.def:
JAVAOP (nop,              0, STACK,   POP,     0)
JAVAOP (aconst_null,      1, PUSHC,   PTR,     0)
JAVAOP (iconst_m1,        2, PUSHC,   INT,     -1)
 <...>
JAVAOP (ret_w,          209, RET,     RETURN,  VAR_INDEX_2)
JAVAOP (impdep1,        254, IMPL,    ANY,     1)
```

Препроцессор: операции # и

Операция ## позволяет объединить фактические аргументы макроса в одну строку.

```
gcc –E java-opcodes.h:
enum java_opcode {
OPCODE_nop = 0,
OPCODE_aconst_null = 1,
OPCODE_iconst_m1 = 2,
OPCODE_iconst_0 = 3,
<...>
OPCODE_impdep2 = 255,
LAST_AND_UNUSED_JAVA_OPCODE
};
```

Компоновка и классы памяти: переменные

Класс памяти	Время жизни	Видимость	Компоновка	Определена
автоматический	автоматическое	блок	нет	в блоке
регистровый	автоматическое	блок	нет	в блоке как `register`
статический	статическое	файл	внешняя	вне функций
статический	статическое	файл	внутренняя	вне функций как `static`
статический	статическое	блок	нет	в блоке как `static`

Квалификатор `extern`: переменная определена и память под нее выделена в другом файле.

Компоновка и классы памяти: функции

Классы памяти функций:

- статическая (объявлена с квалификатором `static`);
- внешняя (`extern`), по умолчанию;
- встраиваемая (`inline`, C99).

Объявление внешних функций в заголовочных файлах:

```
extern void *realloc (void *ptr, size_t size);
```

Компоновщик

- Организовывает слияние нескольких объектных файлов в одну программу
- Разрешает неизвестные символы (внешние переменные и функции)
  - Глобальные переменные с одним именем получают одну область памяти
  - Ошибки, если необходимых имён нет или есть несколько объектов с одним именем
  - Опции для указания места поиска
- Собирает исполняемый файл или библиотеку (*статическую* или *динамическую*)
- Хорошим стилем программирования является экспорт лишь тех объектов, которые используются в других файлах (интерфейс модуля).
  Используйте квалификатор `static`.

Динамическое выделение памяти

Функция `void *malloc (size_t size);` выделяет область памяти размером `size` байтов и возвращает указатель на выделенную область памяти.

Если память не выделена (например, в системе не осталось свободной памяти требуемого размера), возвращаемый указатель имеет значение `NULL`.

Поскольку результат операции `sizeof` имеет тип `size_t` и равен длине операнда в байтах, в качестве `size` можно использовать результат операции `sizeof`.

```
char *p;
p = (char *) malloc (1000 * sizeof (char));

int *p;
p = malloc (50 * sizeof (int));
```

Динамическое выделение памяти

Функция `void free (void *p);` возвращает системе выделенный ранее участок памяти с указателем `p`.

Внимание. Аргументом функции `free()` обязательно должен быть указатель `p` на участок памяти, выделенный ранее функцией `malloc()`.

- Вызов функции `free()` с неправильным указателем не определен и может привести к разрушению системы распределения памяти
- Вызов функции `free()` с указателем `NULL` не приводит ни к каким действиям (C99).
- Обращение к освобожденному указателю не определено.

Функции `malloc()` и `free()` объявлены в `stdlib.h`.

Динамическое выделение памяти. Пример

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main (void) {
  int t;
  char *s = malloc (80 * sizeof (char));
  if (!s) {
    fprintf (stderr, "требуемая␣память␣не␣выделена\n");
    return 1; /* исключительная ситуация */
  }
  fgets (s, 80, stdin); s[strlen (s) - 1] = '\0';
  // посимвольный вывод перевернутой строки на экран
  for (t = strlen(s) - 1; t >= 0; t--)
    putchar (s[t]);
  free (s);
  return 0;
}
```