

**Курс «Алгоритмы и алгоритмические языки»  
1 семестр 2013/2014**

**Лекция 13**

## Препроцессор: операции # и ##

- ◆ Операция ## позволяет объединить фактические аргументы макроса в одну строку

java-opcode.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, PUSHHC,   PTR,      0)
JAVAOP (iconst_m1,          2, PUSHHC,   INT,     -1)
<...>
JAVAOP (ret_w,               209, RET,     RETURN,  VAR_INDEX_2)
JAVAOP (impdep1,             254, IMPL,    ANY,      1)
JAVAOP (impdep2,             255, IMPL,    ANY,      2)
```

## 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

Класс памяти	Время жизни	Видимость	Компоновка	Определена
автоматический	автоматическое	блок	нет	В блоке
регистровый	автоматическое	блок	нет	В блоке как <code>register</code>
статический	статическое	файл	внешняя	Вне функций
статический	статическое	файл	внутренняя	Вне функций как <code>static</code>
статический	статическое	блок	нет	В блоке как <code>static</code>

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

- ◆ Организует 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 имен нет или есть несколько объектов с одним именем
  - ◆ Опции для указания места поиска
- ◆ Хорошим стилем программирования является экспорт лишь тех объектов, которые используются в других файлах (интерфейс модуля)
  - ◆ Используйте квалификатор `static`
- ◆ Сборка исполняемого файла или библиотеки (*статической* или *динамической*)

## Отладка программ

- ◇ Все программы содержат ошибки, отладка – это процесс поиска и удаления некоторых ошибок
- ◇ Существуют другие методы обнаружения ошибок (тестирование, верификация, статические и динамические анализаторы кода), но их применение не гарантирует отсутствия ошибок
- ◇ Для отладки используют инструменты, позволяющие получить информацию о поведении программы на некоторых входных данных, не изменяя ее поведения
- ◇ Простейший метод: *отладочная печать*

```
int *a; int n;
n = read_array (a);
debug_array (a, n);

static void debug_array (int *a, int n) {
    fprintf (stderr, "Array (%d)", n);
    for (int i = 0; i < n; i++)
        fprintf (stderr, "%d ", a[i]);
    fprintf (stderr, "\n");
}
```

- ◇ Отладочная печать может контролироваться макросом (`NDEBUG`)

## Отладка программ: отладчики

- ◇ Отладчик – основной инструмент отладки программы
- ◇ Отладчик позволяет
  - ◆ запустить программу для заданных входных данных
  - ◆ останавливать выполнение по достижении заданных точек программы безусловно или при выполнении некоторого условия на значения переменных
  - ◆ останавливать выполнение, когда некоторая переменная изменяет свое значение
  - ◆ выполнить текущую строку исходного кода программы и снова остановить выполнение
  - ◆ посмотреть/изменить значения переменных, памяти
  - ◆ посмотреть текущий стек вызовов
- ◇ Необходимое условие для отладки на уровне исходного кода: наличие в исполняемом файле программы *отладочной информации* (связи между командами процессора и строками исходного кода программы, связь между адресами и переменными и т.д.)

## Отладка программ: отладчик `gdb`

- ◆ Компиляция с отладочной информацией: `gcc -g`
- ◆ Некоторые команды `gdb`
  - ◆ `gdb <file> --args <args>` – загрузить программу с заданными параметрами командной строки
  - ◆ `run/continue` – запустить/продолжить выполнение
  - ◆ `break <function name/file:line number>` – завести безусловную *точку останова*
  - ◆ `cond <bp#> condition` – задать условие остановки выполнения для некоторой точки останова
  - ◆ `watch <variable/address>` – задать *точку наблюдения* (остановка выполнения при изменении значения переменной или памяти по адресу)
  - ◆ `next/step` – выполнить текущую строку исходного кода программы без захода/с заходом в вызываемые функции
  - ◆ `print <var>/set <var> = expression` – посмотреть /изменить текущие значения переменных, памяти
  - ◆ `bt` – посмотреть текущий стек вызовов
- ◆ Среда Code::Blocks поддерживает `gdb` в своем интерфейсе

## **Отладка программ: примеры команд gdb**

◆ Установка точек останова

◆ МОЖНО ИСПОЛЬЗОВАТЬ '.' ВМЕСТО '->'

```
b fancy_abort
```

```
b 7199
```

```
b sel-sched.c:7199
```

```
cond 2 insn.u.fld.rt_int == 112
```

```
cond 3 x_rtl->emit.x_cur_insn_uid == 1396
```

◆ Просмотр и изменение значений переменных

```
p orig_ops.u.expr.history_of_changes.base
```

```
p bb->index
```

```
set sched_verbose=5
```

```
call debug_vinsn (0x4744540)
```

◆ Установка точек наблюдения

```
wa can_issue_more
```

```
wa ((basic_block) 0x7ffff58b5680)->preds.base.prefix.num
```

# Динамические структуры данных

- ◇ **Стек** (*stack*) – это динамическая последовательность *элементов*, количество которых изменяется, причем как добавление, так и удаление элементов возможно только с одной стороны последовательности (вершина стека).
- ◇ Работа со стеком осуществляется с помощью функций:
  - `push(x)` – *затолкать* элемент **x** в стек;
  - `x = pop()` – *вытолкнуть* элемент из стека.
- ◇ Стек можно организовать на базе:
  - ◆ фиксированного массива `stack[MAH]`, где константа **MAH** задает максимальную глубину стека.
  - ◆ динамического массива, текущий размер которого хранится отдельно.
  - ◆ в обоих случаях необходимо хранить позицию текущей вершины стека.
  - ◆ можно использовать и другие структуры данных (например, список).

# Динамические структуры данных

- ◇ Организация стека на динамическом массиве.

```
struct stack {
    int sp;    /* Текущая вершина стека */
    int sz;    /* Размер массива */
    char *stack;
} stack = { .sp = -1, .sz = 0, .stack = NULL };

static void push (char c) {
    if (stack.sz == stack.sp + 1) {
        stack.sz = 2*stack.sz + 1;
        stack.stack = (char *) realloc (stack.stack,
                                         stack.sz*sizeof (char));
    }
    stack.stack[++stack.sp] = c;
}
```

# Динамические структуры данных

◇ Организация стека на динамической памяти.

```
struct stack {
    int sp;    /* Текущая вершина стека */
    int sz;    /* Размер массива */
    char *stack;
} stack = { .sp = -1, .sz = 0, .stack = NULL };

static char pop (void) {
    if (stack.sp < 0) {
        fprintf (stderr, "Cannot pop: stack is empty\n");
        return 0;
    }
    return stack.stack[stack.sp--];
}

static int isempty (void) {
    return stack.sp == -1;
}
```