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

# Алгоритмы и алгоритмические языки
# Лекция 14

26 октября 2019 г.

# Динамическое выделение памяти для двумерного целочисленного массива

```c
#include <stdio.h>
#include <stdlib.h>

long pwr (int a, int b) {
  long t = 1;
  for (; b; b--)
    t *= a;
  return t;
}
```

# Динамическое выделение памяти для двумерного целочисленного массива

```
int main (void) {
  long *p[6]; int i, j;
  for (i = 0; i < 6; i++)
    if (!(p[i] = malloc (4 * sizeof (long)))) {
      printf ("out␣of␣memory...\n");
      exit (1);
    }
  for (i = 1; i < 7; i++)
    for (j = 1; j < 5; j++)
      p[i - 1][j - 1] = pwr (i, j);
  for (i = 1; i < 7; i++) {
    for (j = 1; j < 5; j++)
      printf ("%10ld␣", p[i - 1][j - 1]);
    printf ("\n");
  }
  <...>
```

# Динамическое выделение памяти для двумерного целочисленного массива

```
  <...>
  for (i = 0; i < 6; i++)
    free (p[i]);
  return 0;
}
```

# VLA-массивы

В C89 размер массива обязан являться константой. Это неудобно при передаче массивов (многомерных) в функции.

```
/* можно передать int a[5]; int a[42]; ... */
int asum1d (int a[], int n) {
  int s = 0;
  for (int i = 0; i < n; i++)
    s += a[i];
  return s;
}
/* можно передать только int a[???][5] */
int asum2d (int a[][5], int n) {
  int s = 0;
  for (int i = 0; i < n; i++)
  for (int j = 0; j < 5; j++)
    s += a[i][j];
  return s;
}
```

# VLA-массивы

В C99 размер массива автоматического класса памяти может задаваться во время выполнения программы (C11 сделал VLA необязательными, проверка через макрос `__STDC_NO_VLA__`).

```
int foo (int n) {
  int a[n];
  // можно обрабатывать a[i]...
}
// можно передать int a[???][???]
int asum2d (int m, int n, int a[m][n]) {
  int s = 0;
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      s += a[i][j];
  return s;
}
int asum2d (int m, int n, int a[m][n]);
int asum2d (int, int, int [*][*]);
```

# VLA-массивы и динамическое выделение памяти

Функция asum2d может использоваться с VLA-массивами, но они всегда выделяются в автоматической памяти.

```
int foo (int m, int n) {
  int a[m][n]; int s;
  <... Считаем a[i][j]...>
  s = asum2d (m, n, a);
  return s;
}
```

# VLA-массивы и динамическое выделение памяти

Можно выделить VLA-массив в динамической памяти.

```
int main (void) {
  int m, n;
  scanf ("%d%d", &m, &n);

  int (*pa)[n];
  pa = (int (*)[n]) malloc (m * n * sizeof (int));
  <... Считаем pa[i][j]...>
  s = asum2d (m, n, pa);
  free (pa);
  return 0;
}
```

# Динамическое распределение памяти

Состав функций динамического распределения памяти (заголовочный файл `<stdlib.h>`).

```
void *malloc (size_t size);
void free (void *p);
void *realloc (void *p, size_t size);
void *calloc(size_t num, size_t size);
```

Функция `calloc` работает аналогично функции `malloc (size1)`, где `size1 = num * size` (т.е. выделяет память для размещения массива из `num` объектов размера `size`).
Выделенная память инициализируется нулевыми значениями.

# Динамическое распределение памяти

Состав функций динамического распределения памяти (заголовочный файл `<stdlib.h>`).

```
void *malloc (size_t size);
void free (void *p);
void *realloc (void *p, size_t size);
void *calloc(size_t num, size_t size);
```

Функция `void *realloc (void *p, size_t size)` сначала выполняет `free (p)`, а потом `p = malloc (size)`, возвращая новое значение указателя `p`. При этом значения первых `size` байтов новой и старой областей совпадают.

# Динамическое распределение памяти

```c
#include <stdio.h>
#include <stdlib.h>
int main (void) {
  int *p = (int*) malloc (sizeof(int));
  int *q = (int*) realloc (p, sizeof(int));
  *p = 1;
  *q = 2;
  if (p == q)
    printf ("%d␣%d\n", *p, *q);
  return 0;
}
```

# Динамическое распределение памяти

```
#include <stdio.h>
#include <stdlib.h>
int main (void) {
  int *p = (int*) malloc (sizeof(int));
  int *q = (int*) realloc (p, sizeof(int));
  *p = 1;
  *q = 2;
  if (p == q)
    printf ("%d␣%d\n", *p, *q);
  return 0;
}

$ clang -O2 realloc.c && ./a.out
1 2
```

# Массив переменного размера в структуре (C99)

Flexible array member — последнее поле структуры.

```
struct polygon {
  int np;        /* число вершин */
  struct point points[];
}
```

Варьирование размера переменного массива.

```
int np; struct polygon *pp;
scanf ("%d", &np);
pp = malloc (sizeof (struct polygon)
             + np * sizeof (struct point));
pp->np = np;
for (int i = 0; i < np; i++)
  scanf ("%d%d", &pp->points[i].x,
         &pp->points[i].y);
```

# 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.

# Отладочная печать и assert.h

```
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");
}
```

Проверка инвариантов: макрос **assert** (контролируется макросом **NDEBUG**). Нежелательно использовать выражения с побочным эффектом.

```
#include <assert.h>
int foo (int *a, int n) {
  assert (n > 0);
  <...>
  debug_array (a, n);
}
```

# Отладчики

Отладчик — основной инструмент отладки, который позволяет:

- запустить программу для заданных входных данных;
- останавливать выполнение по достижении заданных точек программы *безусловно* или при выполнении некоторого *условия* на значения переменных (breakpoints);
- останавливать выполнение, когда некоторая переменная изменяет свое значение (watchpoints);
- 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:

# Отладчики

Отладчик — основной инструмент отладки, который позволяет:

- запустить программу для заданных входных данных;
- останавливать выполнение по достижении заданных точек программы *безусловно* или при выполнении некоторого *условия* на значения переменных (breakpoints);
- останавливать выполнение, когда некоторая переменная изменяет свое значение (watchpoints);
- 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.

# Отладчик gdb

Компиляция с отладочной информацией: `gcc -g`. 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.

# 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
```