Высшая школа экономики
Факультет компьютерных наук
Департамент программной инженерии

Алгоритмы и алгоритмические языки
Лекция 7

21 октября 2025 г.

Хвостовая рекурсия*

Хвостовая рекурсия (tail recursion) — рекурсивный вызов в самом конце функции. Как правило, этот вызов может быть оптимизирован компилятором в цикл.

```
int fact (int n) {
  if (n == 0)
    return 1;
  else
    return n*fact (n-1);
}
```

```
int fact (int n) {
  return tfact (n, 1);
}
int tfact (int n, int acc) {
  if (n == 0)
    return acc;
  return tfact (n-1, n*acc);
}
```

Хвостовая рекурсия*

Хвостовая рекурсия (tail recursion) — рекурсивный вызов в самом конце функции. Как правило, этот вызов может быть оптимизирован компилятором в цикл.

```
int fact (int n) {
  int t_n = n, t_acc = 1;
  /* tfact встроена в fact
   и оптимизирована в цикл */
start:
  if (t_n == 0)
    return t_acc;
  t_acc = t_n * t_acc;
  t_n = t_n - 1;
  goto start;
}
```

```
int fact (int n) {
  return tfact (n, 1);
}
int tfact (int n, int acc) {
  if (n == 0)
    return acc;
  return tfact (n-1, n*acc);
}
```

Ключевое слово `inline`: встраиваемые функции (C99)

```
#include <stdio.h>
inline static int max (int a, int b) {
  return a > b ? a : b;
}
int main (void) {
  int x = 5, y = 17;
  printf ("Maximum␣of␣%d␣and␣%d␣is␣%d\n",
          x, y, max (x, y));
  return 0;
}
```

Ключевое слово `inline`: встраиваемые функции (C99)

При типичной реализации `inline` программа будет преобразована как

```
#include <stdio.h>

inline static int max (int a, int b) {
  return a > b ? a : b;
}

int main (void) {
  int x = 5, y = 17;
  printf ("Maximum␣of␣%d␣and␣%d␣is␣%d\n",
          x, y, (x > y ? x : y));
  return 0;
}
```

Указатели на функцию

Каждая функция располагается в памяти по определенному адресу. Адресом функции является её точка входа (при вызове функции управление передается именно на эту точку).

Присвоив значение адреса функции переменной типа указатель, получим указатель на функцию.

Указатель функции можно использовать вместо её имени при вызове этой функции. Указатель «лучше» имени тем, что его можно передавать другим функциям в качестве их аргумента.

Имя функции `f` без скобок и аргументов по определению является указателем на функцию `f()` (аналогия с массивом).

```
int (*pf) (const char*, const char*);
char *s1, *s2;
int x = (*pf) (s1, s2);
int y = pf (s2, "string␣constant");
```

Указатели на функцию. Пример

```
#include <stdio.h>
#include <string.h>
static void check (char *a, char *b,
  int (*pf) (const char*, const char*)) {
  printf ("Проверка на совпадение: ");
  if (! pf (a, b))
    printf ("равны\n");
  else
    printf ("не равны\n");
}
int main (void) {
  char s1[80], s2[80];
  printf ("Введите две строки\n");
  fgets (s1, sizeof (s1), stdin); s1[strlen (s1) - 1] = 0;
  fgets (s2, sizeof (s2), stdin); s2[strlen (s2) - 1] = 0;
  check (s1, s2, strcmp);
  return 0;
}
```

Указатели на функцию. Пример

Объявление `int (*p)(const char *, const char *);` сообщает компилятору, что `p` — указатель на функцию, имеющую два параметра типа `const char *` и возвращающую значение типа `int`.

Скобки вокруг `*p` нужны, так как операция `*` имеет более низкий приоритет, чем `()`. Если написать `int *p(...)`, получится, что объявлен не указатель на функцию, а функция `p`, которая возвращает указатель на целое.

`(*cmp)(a, b)` эквивалентно `cmp(a, b)`.

Указатель `pf` и функция `strcmp` имеют одинаковый формат, что позволяет использовать имя функции в качестве аргумента, соответствующего параметру `pf`.

Указатели на функцию. Пример

В данном случае использование указателя на функцию позволяет не менять программу сравнения, и тем самым получается более общий алгоритм

```
int compvalues (const char *a, const char *b) {
  return atoi (a) != atoi (b);
}
```

Массивы указателей на функцию: гибкая обработка событий.

Структуры

Структура — это совокупность нескольких переменных, часто разных типов, сгруппированных под одним именем для удобства.

> Переменные, перечисленные в объявлении структуры, называются её *полями*, *элементами* или *членами*.

Объявление структуры:

```
struct point
{
  int x;
  int y;
} f, g;
struct point h, center = {32, 32};
```

Структуры

Поля структуры могут иметь любой тип, например, тип массива или тип другой структуры.

```
struct rect
{
  struct point pt1;
  struct point pt2;
};
```

Инициализация структуры:

```
struct rect r = {.pt1 = {4, 4},
                 .pt2 = {7, 6}};
/*  Остальные элементы — нулевые  */
struct rect r2 = {.pt2.x = 5};
```

Размер структуры в общем случае **не равен** сумме размеров её элементов (**выравнивание**).

Структуры

Размер структуры и требования по выравниванию полей.

Доступ к полям структуры: операция точка `.`

```
f.x, g.y, r.pt1.x
```

Присваивание структур целиком: `f = g;`

Массивы структур:

```
#define NRECT 15
/*  Первый прямоугольник вокруг 0, 0  */
struct rect rectangles[NRECT]
       = {{-1, -1, 1, 1}};
/*  Последний прямоугольник — большой  */
#define BOUND 1024
struct rect bounded_rectangles[NRECT]
       = {[NRECT-1] = {-BOUND, -BOUND,
                        BOUND,  BOUND}};
```

Указатели на структуры

```
struct rect r = {.pt1 = {4, 4},
                 .pt2 = {7, 6}};
struct rect *pr = &r;
```

Доступ к полям структуры через указатель:

```
pr->pt1 (= (*pr).pt1), pr->pt2.x
```

Адресная арифметика:

```
struct rect *pr = &bounded_rectangles[0];
while (pr->pt1.x != -BOUND)
  pr++;
```

Составные инициализаторы структур (C99)

```
struct rect r;
r = (struct rect) { {4, 4},
                    {7, 6} };
```

Составной инициализатор генерирует **lvalue**! Т.е. можно брать адрес и передавать указатель:

```
double area (struct rect *r) {
  return (r->pt1.x - r->pt2.x)
         * (r->pt1.y - r->pt2.y);
}
double da = area (& (struct rect) {{4, 4}, {7, 6}});
```

Старшинство операций

Старшинство	Ассоциативность
`( ) [ ] -> .`	Слева направо
`! ~ ++ -- + - sizeof (type)`	Справа налево
`* / %`	Слева направо
`+ -`	Слева направо
`<< >>`	Слева направо
`< <= > >=`	Слева направо
`== !=`	Слева направо
`&`	Слева направо
`^`	Слева направо
`|`	Слева направо
`&&`	Слева направо
`||`	Слева направо
`?:`	Справа налево
`= += -= *= /= %=`	Справа налево
`,`	Слева направо

Объединения

Объединение — это объект, который может содержать значения различных типов (но не одновременно — только одно в каждый момент).

```
struct constant
{
  int ctype;
  union
  {
    int i;
    float f;
    char *s;
  } u;
} sc;
```

```
switch (sc.ctype)
{
  case CI:
    printf("%d",sc.u.i);
    break;
  case CF:
    printf("%f",sc.u.f);
    break;
  case CS: puts(sc.u.s);
}
```

Размер объединения достаточно велик, чтобы содержать максимальный по размеру элемент.

Можно выполнять те же операции, что и со структурами.

Анонимные объединения и структуры (C11)

Для вложенных структур и объединений разрешено опускать тег для повышения читаемости.

```
struct constant
{
  int ctype;
  union
  {
    int i;
    float f;
    char *s;
  } /*  нет имени!  */;
} sc;
```

```
switch (sc.ctype)
{
  case CI:
    printf("%d",sc.i);
    break;
  case CF:
    printf("%f",sc.f);
    break;
  case CS: puts(sc.s);
}
```

Поля анонимной структуры считаются принадлежащими родительской структуре (если родительская также анонимна — то следующей родительской структуре и т.п.)

Битовые поля

Для экономии памяти можно точно задать размер поля в битах (например, набор флагов).

```
struct tree_base {
  unsigned code : 16;
  unsigned side_effects_flag : 1;
  unsigned constant_flag : 1;
  <...>
  unsigned lang_flag_0 : 1;
  unsigned lang_flag_1 : 1;
  <...>
  unsigned spare : 12;
}
```

- Адрес битового поля брать запрещено
- Можно объявить анонимные поля (для выравнивания)
- Можно объявить битовое поле ширины 0 (для перехода на следующий байт)

Перечисления

Перечисления — целочисленные типы данных, определяемые программистом. Определение перечисления:

```
enum typename { name[=value], ... };
enum colors {red, orange, yellow, green, azure,
blue, violet};
```

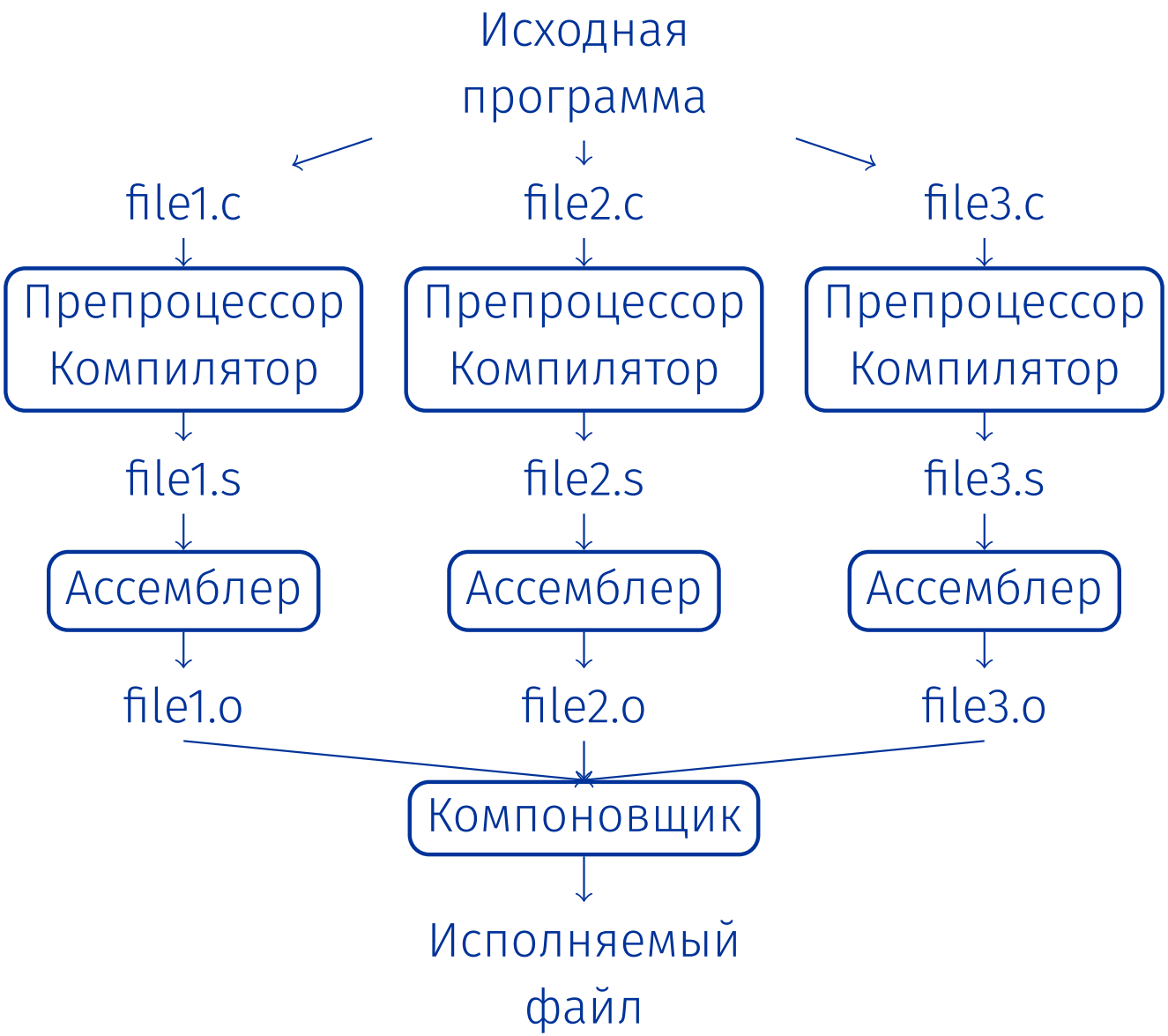
Значения перечисления нумеруются с 0, но можно присваивать свои значения.

```
enum {red, orange = 23, yellow = 23, green, cyan = 75,
blue = 75, violet};
```

Доступны операции над целочисленными типами и объявление указателей на переменные перечислимых типов.

Проверка корректности присваиваемых значений не производится.

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
```

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;
}
```

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
```

Динамическое выделение памяти для двумерного целочисленного массива

```
#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;
}
```

[1]variably-modified type

VLA-массивы и динамическое выделение памяти

Можно выделить VLA-массив в динамической памяти.
C23 сделал такие типы *обязательными*[1], оставив VLA-массивы в автоматической памяти опциональными.

```
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;
}
```

[1] variably-modified type

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
```

Поиск ошибок работы с памятью

Частые ошибки работы с динамической памятью тяжело отлаживать (даже в небольших программах).

- Ошибки доступа за границы буфера
- Ошибки использования неинициализированного или уже освобожденного указателя
- Недостаточный размер буфера

Разработан ряд инструментов анализа, которые облегчают жизнь программисту.

- valgrind: динамический двоичный транслятор `http://valgrind.org`
- sanitizers: компиляторная инструментация от Google `https://github.com/google/sanitizers/wiki`

Disclaimer: Linux-only tools[2] [3].

[2] На Windows работает Dr.Memory: `https://drmemory.org/`

[3] Начиная с Visual Studio 2019, также работает и Address Sanitizer

Поиск ошибок работы с памятью

valgrind: динамический двоичный транслятор (плюс набор инструментов, ваш — memcheck).

```
#include <stdlib.h>
void f(void) {
  int* x = malloc(10 * sizeof(int));
  x[10] = 0;        // problem 1: heap block overrun
}
int main(void) {
  f();
  return 0;
}
$ gcc -Og -g -o me && valgrind ./me
<...>
==27164== Invalid write of size 4
==27164==    at 0x400554: f (me.c:4)
==27164==    by 0x400568: main (me.c:7)
==27164==  Address 0x51da068 is 0 bytes after a block of size 40 alloc'd
==27164==    at 0x4C2C12F: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd
==27164==    by 0x400553: f (me.c:3)
==27164==    by 0x400568: main (me.c:7)
```

Поиск ошибок работы с памятью

sanitizers: встроенная в gcc/clang инструментация (нас интересует address sanitizer).

```
#include <stdlib.h>
void f(void) {
  int* x = malloc(10 * sizeof(int));
  x[10] = 0;          // problem 1: heap block overrun
}
int main(void) {
  f();
  return 0;
}
```

```
$ gcc -Og -g -fsanitize-address -o mesa && ./mesa
==27179==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60400000dff8
WRITE of size 4 at 0x60400000dff8 thread T0
    #0 0x4007c0 in f /home/bonzo/tmp/me.c:4
    #1 0x4007d5 in main /home/bonzo/tmp/me.c:7
    #2 0x7fba219d870f in __libc_start_main (/lib64/libc.so.6+0x2070f)
    #3 0x4006b8 in _start (/home/bonzo/tmp/mesa+0x4006b8)
0x60400000dff8 is located 0 bytes to the right of 40-byte region [0x60400000dfd0
allocated by thread T0 here:
    #0 0x7fba21df074a in malloc (/usr/lib64/libasan.so.2+0x9674a)
    #1 0x400793 in f /home/bonzo/tmp/me.c:3
SUMMARY: AddressSanitizer: heap-buffer-overflow /home/bonzo/tmp/me.c:4 f
```