Лекция 8. Обзор libc - III. Линковка: static/inline, библиотеки. Разное: va_arg.

Евгений Линский

string

```
1  // не проверяют размер dst
2  // оптимизированы используют ``широкие'' команды sse, avx etc
3  void* memcpy ( void* dst, const void* src, size_t num );
4  char* strcpy ( char* dst, const char* src );
```

Разбить строку на токены по разделителям.

```
1    //char * strtok ( char * str, const char * delimiters );
2    char str[] ="This,a sample string.";
3    char* pch = strtok (str," ,");
4    while (pch != NULL) {
5      printf ("%s\n",pch);
6      pch = strtok (NULL, " ,.-");
7    }
```

1. Как работает?

string

```
// не проверяют размер dst
// оптимизированы используют ``широкие'' команды sse, avx etc
void* memcpy ( void* dst, const void* src, size_t num );
char* strcpy ( char* dst, const char* src );
```

Разбить строку на токены по разделителям.

```
//char * strtok ( char * str, const char * delimiters );
char str[] ="This,a sample string.";
char* pch = strtok (str," ,");
while (pch != NULL) {
  printf ("%s\n",pch);
  pch = strtok (NULL, " ,.-");
}
```

1. Как работает?
2. Внутри strtok есть статическая переменная, которая хранит место в строке, откуда надо начать при следующем вызове (если str == NULL). Если str не NULL, то начать надо с str.

string

```
1  // не проверяют размер dst
2  // оптимизированы используют ``широкие'' команды sse, avx etc
3  void* memcpy ( void* dst, const void* src, size_t num );
4  char* strcpy ( char* dst, const char* src );
```

Разбить строку на токены по разделителям.

```
1    //char * strtok ( char * str, const char * delimiters );
2    char str[] ="This,a sample string.";
3    char* pch = strtok (str," ,");
4    while (pch != NULL) {
5      printf ("%s\n",pch);
6      pch = strtok (NULL, " ,.-");
7    }
```

1. Как работает?
2. Внутри strtok есть статическая переменная, которая хранит место в строке, откуда надо начать при следующем вызове (если str == NULL). Если str не NULL, то начать надо с str.
3. strtok разрушает строку, расставляя в ней 0, чтобы printf “знал”, где остановиться при выводе очередного токена.

time

time — текущее время в секундах с 1970 года.

```
time_t t1 = time (NULL);
f();
time_t t2 = time (NULL);
time_t duration = t2 - t1;
```

Секунды слишком долго!

```
clock_t t1 = clock();
f();
clock_t t2 = clock();
time_t duration = (t2 - t1) / CLOCKS_PER_SEC;
```

clock returns the **processor** time consumed by the program.

1. Что не так с clock в многопоточных программах?

C11:

```
int timespec_get(struct timespec *ts, int base);
ts->tv_sec, ts->tv_nsec // seconds and nano seconds
```

time

time — текущее время в секундах с 1970 года.

```
time_t t1 = time (NULL);
f();
time_t t2 = time (NULL);
time_t duration = t2 - t1;
```

Секунды слишком долго!

```
clock_t t1 = clock();
f();
clock_t t2 = clock();
time_t duration = (t2 - t1) / CLOCKS_PER_SEC;
```

clock returns the **processor** time consumed by the program.

1. Что не так с clock в многопоточных программах?
2. 2 потока, 2 core –> clock насчитает в два раза больше чем time.

C11:

```
int timespec_get(struct timespec *ts, int base);
ts->tv_sec, ts->tv_nsec // seconds and nano seconds
```

assert

```
1  //Гарантировано  выявим ошибки программиста на стадии отладки
2  void print_array(int* a, size_t n) {
3    assert (a != NULL); // если условие не выполнено, то abort
4    ...
5  }
6  // для релиза (не DEBUG)
7  gcc -DNDEBUG a.c // в этом случае assert компилируется в ;
8  // все равно что #define NDEBUG
```

1. Как реализовано?

assert

```
1  //Гарантировано  выявим ошибки программиста на стадии отладки
2  void print_array(int* a, size_t n) {
3    assert (a != NULL); // если условие не выполнено, то abort
4    ...
5  }
6  // для релиза (не DEBUG)
7  gcc -DNDEBUG a.c // в этом случае assert компилируется в ;
8  // все равно что #define NDEBUG
```

1. Как реализовано?
2. #ifdef NDEBUG ; #else if(...) else abort() ... #endif

assert, stdint

```
1  FILE* f = fopen(...);
2  assert(f != NULL);
```

1. Хорошая идея?

stdint.h

```
1  int16_t a;
2  uint64_t b;
```

Может не компилироваться, если на платформе нет типа!

assert, stdint

```
1 FILE* f = fopen(...);
2 assert(f != NULL);
```

1. Хорошая идея?
2. Нет. Ошибка не выявить на стадии отладки, а может произойти всегда! А при NDEBUG мы отключим проверку.

stdint.h

```
1 int16_t a;
2 uint64_t b;
```

Может не компилироваться, если на платформе нет типа!

static у локальной переменной

```
1  void f() {
2  // сохраняет значение между вызовами функции
3    static int call_count = 0; //инициализируется один раз
4    ...
5    printf("Called times: %d", call_count);
6    call_count++;
7  }
8
9  int main() {
10   f(); f(); f();
11 }
```

static у глобальной переменной и функции

error: multiple definition:

```
1 //tree.c
2 tree_t node;
3 void fill_nodes() { tree_t t; ... }
4 //list.c
5 list_t node;
6 void fill_nodes() { list_t l;... }
```

static и линковка

static у глобальной переменной или функции — идентификатор используется только для разрешения имен в рамках одного файла

```
1 //tree.c
2 static tree_t node;
3 static void fill_nodes() { tree_t t; ... }
4 //list.c
5 static list_t node;
6 static void fill_nodes() { list_t l;... }
```

В ELF файл попадут обе функции.

inline и линковка

“Заинлайнить” — оптимизация компилятора.

До:

```
1  int max(int a, int b) {
2    if(a > b)
3      return a;
4    else
5      return b;
6  }
7  main( ) {
8    int c = ...;
9    int b = ...;
10   int d = max(c, b);
11 }
```

inline и линковка

После:

```
1  main( ) {
2    int c = ...;
3    int b = ...;
4    if(c > b)
5      d =  c;
6    else
7      d = b;
8
9  }
```

Компилятор принимает такие решения самостоятельно. Можно дать совет -O0, -O1, -O2, -O3 (уровень оптимизации).

inline и линковка

a.c

```
1  int max(int a, int b) {
2    if(a > b)
3      return a;
4    else
5      return b;
6  }
```

b.c

```
1  f( ) {
2    int c = ...;
3    int b = ...;
4    int d = max(c, b);
5  }
```

- Сможет ли компилятор заинлайнить?

inline и линковка

a.c

```
1  int max(int a, int b) {
2    if(a > b)
3      return a;
4    else
5      return b;
6  }
```

b.c

```
1  f( ) {
2    int c = ...;
3    int b = ...;
4    int d = max(c, b);
5  }
```

- Сможет ли компилятор заинлайнить?
- Нет. На стадии компиляции определение (definition) функции max недоступно при комиляции b.c

inline и линковка

a.h

```
1  int max(int a, int b) {
2    if(a > b)
3      return a;
4    else
5      return b;
6  }
```

b.c

```
1  #include "a.h"
2  f( ) { int c = ...; int b = ...; int d = max(c, b); }
```

c.c

```
1  #include "a.h"
2  g( ) { int e = ...; int f = ...; int g = max(e, f); }
```

- Сможет ли компилятор заинлайнить?

inline и линковка

a.h

```
1  int max(int a, int b) {
2    if(a > b)
3      return a;
4    else
5      return b;
6  }
```

b.c

```
1  #include "a.h"
2  f( ) { int c = ...; int b = ...; int d = max(c, b); }
```

c.c

```
1  #include "a.h"
2  g( ) { int e = ...; int f = ...; int g = max(e, f); }
```

- Сможет ли компилятор заинлайнить?
- Сможет, но будет ошибка double definition на линковке.

inline и линковка

a.h

```
1  inline int max(int a, int b) {
2    if(a > b)
3      return a;
4    else
5      return b;
6  }
```

b.c

```
1  #include "a.h"
2  f( ) { int c = ...; int b = ...; int d = max(c, b); }
```

c.c

```
1  #include "a.h"
2  g( ) { int c = ...; int b = ...; int d = max(c, b); }
```

Линкер выберет один какой-то вариант функции. В ELF файл попадет одна функция.

Библиотеки.

1. Статические (*.a, *.lib): объектные файла из библиотеки присоединяются к программе в момент линковки
   - Легко установить (не нужно отдельно загружать библиотеку)
   - При выходе новой версии библиотеки (например, исправлен баг) автору программы нужно послать пользователю пересобранную версию
2. Динамические (*.so, *.dll): хранятся отдельно, связывание программы и библиотеки происходит в момент выполнения (помогает отдельная компонента — загрузчик)
   - Необходимо отдельно установить все необходимые библиотеки (решение: packages and repositories)
   - При выходе новой версии библиотеки (например, исправлен баг) пользователь может самостоятельно обновить только библиотеку без пересборки программы.

Библиотеки.

- Для использования библиотеки нужно: *.a или *.so и заголовочные файлы
- Ключи:
  - -lимя библиотеки (-lexpat)
  - -static сборка со статической версией библиотеки (по умолчанию динамическая)

Еще бывает загрузка динамической библиотеки по запросу (плагины)

```
1 void *dlopen(const char *library_name, int flags);
```

Переменное число аргументов в C (printf)

va_start, va_arg, va_end — макросы.

```
1  void simple_printf(const char* fmt, ...) {
2    va_list args;
3    //записать в args адрес следующего за fmt параметра на стеке
4    va_start(args, fmt);
5    while(*fmt != '\0') {
6      if(*fmt=='d') {
7        //достать со стека переменную типа int
8        int i = va_arg(args, int)
9        // здесь должен быть код, который
10       // выводит int на экран с помощью putc
11     }
12     fmt++;
13   }
14   va_end(args);
15 }
16 //Труднообнаруживаемые ошибки
17 printf("%s", 5);
18 printf("%d %d", 4); printf("%d", 4, 5);
```