1 Занятие №2.

1.1 Первая программа

Рассмотрим такую задачу: *написать программу, которая вводит пары чисел и вычисляет их НОД (наибольший общий делитель)*. Для вычисления будем использовать соотношение

$$\gcd(a,b) = \gcd(b, a \bmod b).$$

1.1.1 Текст программы

Полный текст программы приведён ниже.

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5         int a, b;
 6         int c;
 7
 8         while (scanf("%d%d", &b, &c) == 2) {
 9
10                 /* invariant: GCD(a,b)==GCD(b,a%b) */
11
12                 do {
13                         a = b;
14                         b = c;
15                         c = a % b;
16                 } while (c != 0);
17                 printf("%d\n", b);
18         }
19
20         return 0;
21 }
```

Теперь мы разберём его по шагам, демонстрируя базовые возможности языка Си.

1.1.2 Директива препроцессора

```
1 #include <stdio.h>
```

Это — так называемая директива препроцессора. Перед основной фазой трансляции в текст программы будет добавлено содержимое файла `stdio.h`, которые находится в одном из стандартных каталогов операционной системы. Собственно язык Си не определяет никаких функций, которые поддерживают операции ввода/вывода, работу со строками и другие необходимые возможности. Все они вынесены в стандартную библиотеку Си. В UNIX-системах стандартную библиотеку языка Си часто называют **libc**. Файл `stdio.h` — это один из файлов стандартной библиотеки, в котором определяются типы данных, константы, переменные и функции, необходимые для операций ввода/вывода. Наличие такой директивы препроцессора указывает, что в программе будут использоваться функции ввода/вывода. По-

скольку почти всякая программа что-либо вводит или выводит, включение файла `stdio.h` будет присутствовать почти в каждой программе.

1.1.3 Функция `main`

```
 3 int main(void)
 4 {
21 }
```

Это — определение функции с именем `main`. В языке Си отсутствует деление на процедуры и функции, все выполняемые объекты называются функциями, даже если они ничего не возвращают.

Программа на языке Си является совокупностью функций и глобальных определений, то есть понятие о каком-то главном блоке, с которого начинается работа, отсутствует. В программе должна быть определена функция с именем `main`, именно эта функция будет вызвана первой, после того, как завершится системная инициализация.

В данном определении функция `main` не принимает никаких параметров и возвращает целое значение. Поскольку функция `main` вызывается операционной средой, тип параметров и тип возвращаемого значения достаточно жёстко фиксированы. Никогда не определяйте функцию `main` как не возвращающую значения!

Фигурные скобки после заголовка функции — это составной оператор, аналог **`begin`** и **`end`** языка Паскаль. Тело функции в языке Си заключается в составной оператор, даже если оно состоит из одного оператора. Иногда составной оператор мы будем просто называть *блоком*.

В начале любого составного оператора, а не только того, которые образует тело функции, могут стоять определения переменных. Эти переменные, как правило, существуют ограниченное время, пока составной оператор не завершил работу.

В C90[1] все определения локальных переменных должны размещаться до операторов, а в стандарте C99[2], как и в языке C++, определения локальных переменных и операторы могут быть перемешаны.

1.1.4 Определения переменных

```
 5         int a, b;
 6         int c;
```

Это — определение локальных переменных `a`, `b`, `c`, которые имеют целый тип.

Простейшее определение переменных имеет такой вид.

```
определение = тип список_переменных ";"
список_переменных = переменная { "," переменная }
переменная = имя [ "=" значение ]
```

При определении переменная может быть сразу проинициализирована некоторым значением, например.

```
    int a = 4;
```

Регистр букв (заглавные—строчные) в идентификаторах и ключевых словах значим, то есть `Main` и `main` — это два различных идентификатора. Тем не менее, в программе не рекомендуется использовать имена, различающиеся только регистром букв.

[1] ISO стандарт языка Си, принятый в 1990 году. Этот стандарт (с поправками) — именно то, что обычно называется ANSI C.

[2] Новый стандарт языка Си, принятый, очевидно, в 1999 году.

char	целый тип, достаточный для хранения кода символа.
signed char	знаковый целый тип, достаточный для хранения символа.
unsigned char	беззнаковый целый тип, достаточный для хранения символа.
[signed] short [int]	короткое целое со знаком.
unsigned short [int]	короткое целое без знака.
[signed] int	целое.
unsigned [int]	целое без знака.
[signed] long [int]	длинное целое.
unsigned long [int]	длинное целое без знака.
[signed] long long [int]	более длинное целое (введён стандартом C99).
unsigned long long [int]	более длинное целое без знака (введён стандартом C99).
float	число с плавающей точкой одинарной точности.
double	число с плавающей точкой двойной точности.
long double	число с плавающей точкой расширенной точности.

Таблица 1: Простые типы языка Си

1.1.5 Простые типы данных

В языке Си определены 4 основных простых типа.

char целый тип, достаточный для хранения символов;
int целый тип;
float вещественное число с одинарной точностью;
double вещественное число с двойной точностью.

Чтобы определить тип более точно используются квалификаторы **short**, **long**, **signed**, **unsigned**, в результате можно использовать следующие простые типы данных, которые перечислены в таблице 1.

Ключевое слово, записанное в квадратных скобках, означает, что оно может быть опущено. Так, тип **[signed] long [int]** может записываться и как **signed long**, и как **long int**, и как **signed long int**, и просто как **long**. Кроме того, ключевые слова могут свободно переставляться друг относительно друга, то есть тот же самый тип может задаваться и как **long int signed**.

Является ли тип **char** знаковым или беззнаковым, зависит от конкретной реализации компилятора языка Си. Поэтому везде, где требуется выполнять арифметические операции, зависящие от знаковости, необходимо явное указание знаковости типа **char**.

Размер типа **char** является единицей измерения размера всех других типов. Постулируется, что переменная типа **char** занимает один байт. Таким образом может получиться, что байт будет содержать, например, 9 битов. Все прочие типы имеют размер, кратный целому числу байтов.

Тип **int** имеет размер машинного слова на данной архитектуре. Если машинное слово — 16 бит, то и тип **int** будет иметь такой же размер.

Все знаковые типы имеют точно такой же размер, как и беззнаковые типы. Способ хранения знаковых чисел стандартом неопределён, но все существующие архитектуры хранят знаковые числа в формате дополнения до 2.

Стандарт определяет следующие неравенства для размеров целых типов.

$$sizeof(char) < sizeof(short) \leqslant sizeof(int) \leqslant sizeof(long) \leqslant sizeof(long\ long)$$

$$sizeof(short) < sizeof(long)$$

$$sizeof(int) < sizeof(long\ long)$$

Все современные архитектуры имеют байт, состоящий из 8 бит. На многих современных 32-битных архитектурах целые типы имеют следующие размеры:

short 2 байта (16 бит).
int 4 байта (32 бита).
long 4 байта (32 бита).
long long 8 байтов (64 бита).

Для типов с плавающей точкой точность (количество битов мантиссы) и диапазон представления (количество битов порядка) растёт от **float** до **long double**.

1.1.6 Функция ввода данных

8 `scanf("%d%d", &b, &c)`

Это — вызов функции `scanf` для считывания данных со стандартного потока ввода (обычно стандартный поток ввода — это клавиатура). Функция `scanf` первая функция стандартной библиотеки языка Си, которую мы рассмотрим.

Первый аргумент функции это строка формата (строки мы будем рассматривать позже, а пока заметьте, что они записываются в кавычках), которая определяет, значения каких типов должны быть считаны. Остальные аргументы задают переменные, в которые должны быть считаны значения.

Строка формата состоит из спецификаторов ввода полей. Каждый спецификатор ввода начинается со знака `%` («процент»). Простейшие спецификаторы ввода перечислены ниже.

`%d` Считать целое число (**int**). Перед чтением числа пропускаются все пробельные символы (пробелы, табуляции, переводы строк). Если первый непробельный символ не может начинать число, функция `scanf` завершается. Иначе число считывается либо пока не возникнет переполнения, либо пока не встретится символ, который не может быть частью числа. Если возникло переполнение, функция `scanf` завершается с неудачей. Если встретился нецифровой символ, чтение числа считается успешным, а этот символ не будет считан из потока.

`%ld` Считать длинное целое число (**long int**). Используются те же самые правила, что и при чтении обычного целого числа.

`%Ld` Считать длинное целое число (**long long int**). Используются те же самые правила, что и при чтении обычного целого числа.

`%f` Считать вещественное число типа **float**. Применяются те же правила, что и при чтении целых чисел.

`%lf` Считать вещественное число типа **double**. Применяются те же правила, что и при чтении целых чисел.

`%Lf` Считать вещественное число типа **long double**. Применяются те же правила, что и при чтении целых чисел.

`%c` Считать очередной символ из входного потока в переменную типа **char**.

Несколько спецификаций формата могут быть записаны в одной форматной строке. Например, `"%lf%d%f"` — считать вещественное значение в переменную типа `double`, затем

целое значение в переменную типа `int`, затем вещественное значение в переменную типа `float`.

После спецификации формата перечисляются переменные, в которые будет записано значение. Для всех простых типов, которые были упомянуты выше, перед именем переменной *обязательно* должен стоять знак `&`. Это — унарная операция взятия адреса переменной, которую мы ещё рассмотрим в дальнейшем.

Количество спецификаций формата в форматной строке обязательно должно совпадать с количеством переменных, указанных после неё. Кроме того, тип, указанный в спецификации формата, обязательно должен совпадать с типом соответствующей переменной. Если вы где-то ошиблись, то многие компиляторы ничего не заметят и скомпилируют программу. При выполнении такая программа будет либо давать неправильный результат, либо вообще завершаться аварийно.

Функция `scanf` возвращает количество успешно считанных полей, заданных в спецификации. В примере из программы, которую мы пишем, значение 2 означает, что успешно считаны два целых числа, 1 означает, что было считано только первое число, а второе — нет (закончился файл или до начала чтения числа встретился символ, которые не может быть частью целого числа). Если бы `scanf` вернул 0, это значит, что ни одна переменная не была считана из-за неверного задания входных данных в файле. Если данные закончились (то есть чтение дошло до конца файла) до того, как была считана хотя бы одна переменная, функция `scanf` вернёт значение, задаваемое константой `EOF`.

Игнорировать значение, возвращаемое функций `scanf` нельзя! То есть, язык, конечно, позволяет это делать, но программа, написанная таким образом будет работать некорректно, если пользователь ошибся при вводе.

1.1.7 Оператор цикла `while`

```
 8          while (scanf("%d%d", &b, &c) == 2) {
18          }
```

Это — оператор цикла **`while`**. Цикл **`while`** выполняется (как и в Паскале) до тех пор, пока истинно выражение, записанное в скобках. В данном случае цикл будет выполняться, пока функция `scanf` возвращает значение 2, то есть пока считываются оба числа `b` и `c`. Круглые скобки здесь — часть оператора цикла **`while`**, а не выражения условия цикла. Круглые скобки не могут быть пропущены.

Телом цикла **`while`** может быть любой оператор и, в частности, составной оператор, как в разбираемой программе.

1.1.8 Операции сравнения и логические операции

В языке Си определены обычные операции сравнения чисел, которые записываются следующим образом:

`==` сравнение двух чисел на равенство.
`!=` сравнение на неравенство.
`>=` «больше или равно».
`>` «больше».
`<=` «меньше или равно».
`<` «меньше».

Обратите внимание, что сравнение на равенство записывается как два знака равенства ==, а один знак равенства = — это операция присваивания!

Подробнее то, как вычисляются выражения, мы рассмотрим подробнее на следующих занятиях.

Для проверки нескольких условий используются логические операции-связки `||` и `&&`. Обратите внимание, что оба знака операции состоят из двух символов. Есть и операции `|` и `&`, это совсем другие операции, мы их рассмотрим позднее.

Операция `||` — логическое «или». Она даёт истинное значение, если хотя бы один из аргументов даёт истинное значение. При этом операция вычисляется по «короткой» схеме, то есть если первый аргумент операции дал «истину», второй даже не вычисляется.

Операция `&&` — логическое «и». Она даёт значение «ложь», если хотя бы один из аргументов даёт значение «ложь». Как и предыдущая, эта операция вычисляется по «короткой» схеме. Если первый аргумент дал значение «ложь», второй аргумент даже не вычисляется.

Обратите внимание, что в языке Си отсутствует логический тип как таковой. Везде, где требуется логическое значение «ложь» или «истина», может использоваться любое целое выражение (и вообще, любое скалярное выражение). Значение 0 понимается, как «ложь», а любое значение, не равное 0, как «истина». Тем не менее, операции отношения и логические операции вырабатывают в качестве значения «истины» вполне определённое значение — 1.

В новом стандарте **C99** тип **`bool`** всё-же введёт. Но все правила, описанные выше, продолжают работать.

1.1.9 Оператор `do while`

```
12                  do {
16                  } while (c != 0);
```

Это — оператор цикла с постусловием. Отличия от цикла **`repeat until`** языка Паскаль перечислены ниже:

1. Ключевые слова **`do`** и **`while`** не образуют блока. Поэтому, если в цикле необходимо записать несколько операторов, нужно использовать составной оператор.

2. Круглые скобки после ключевого слова **`while`** являются частью оператора цикла, а не выражения, и поэтому обязательны.

3. Цикл выполняется, пока условие, записанное после **`while`** истинно.

1.1.10 О расстановке «;»

В языке Си «точки с запятой» являются составной частью каждого оператора, кроме составного, поэтому должны обязательно ставится после них. После составного оператора точка с запятой никогда не должна ставиться!

1.1.11 Арифметические выражения и присваивания

```
13                  a = b;
14                  b = c;
15                  c = a % b;
```

В языке определены обычные арифметические операции:

+ сложение.
- вычитание.
* умножение.
/ деление. Если оба операнда операции — целые выражения, деление будет выполняться как деление нацело, результатом будет тоже целое значение. Если хотя бы один операнд — вещественное число, деление будет выполняться над вещественными числами и даст вещественный результат.
% взятие остатка от деления. Применимо только к целым выражениям.

При выполнении операций сложения и вычитания с целыми числами арифметическое переполнение или переносы не диагностируются. Программа продолжит работать как ни бывало. Но при выполнении операций умножения и деления, а также сложения и вычитания с вещественными числами, ошибки диагностируются и вызывают аварийное завершение программы.

Операция присваивания = даёт результат, равный присвоенному значению, поэтому допустимо использование нескольких присваиваний подряд, например:

```
a = b = c = 0;
```

1.1.12 Функция вывода данных

```
17                  printf("%d\n", b);
```

Функция `printf` печатает значения на стандартный поток вывода (обычно это экран). Первый параметр функции — это строка формата печати. Строка формата может содержать обычные литеры, которые будут просто печататься, а может содержать спецификации формата печати, похожие на рассмотренные нами при разборе функции `scanf`. Простейшие из них перечислены ниже.

`%d` печать целого (**`int`**) значения со знаком.
`%u` печать беззнакового целого (**`unsigned int`**).
`%ld` печать длинного целого (**`long`**) со знаком.
`%lu` печать беззнакового длинного целого (**`unsigned long`**).
`%Ld` печать длинного целого (**`long long`**) со знаком.
`%Lu` печать беззнакового длинного целого (**`unsigned long long`**).
`%c` печать литеры.
`%f` печать вещественного числа (**`double`**).
`%Lf` печать вещественного числа (**`long double`**).

В форматной строке могут использоваться специальные последовательности символов: `\n` — переход на следующую строку, `\t` — символ табуляции (для равного отступа столбцов). Чтобы напечатать знак процента, он повторяется дважды.

Знак взятия адреса `&` для всех пока рассмотренных нами типов не ставится! Аргументы печати должны быть точно того типа, который указан в спецификации формата (исключения — см. ниже). Они должны идти в том же порядке, в котором перечислены в строке формата. В противном случае ваша программа будет в лучшем случае печатать что-то странное, а в худшем — аварийно завершаться.

Для печати значений типа **`short`** или **`signed char`** нужно использовать спецификатор печати чисел типа **`int`**. Для печати значений типа **`unsigned short`** или **`unsigned char`** нужно использовать спецификатор печати чисел типа **`unsigned int`**. Для печати значений типа **`float`** используется спецификатор печати значений типа **`double`**.

1.1.13 Возврат значения из функции

```
20          return 0;
```

Поскольку функция `main` объявлена как возвращающая целое значение, необходим оператор, который определит возвращаемое значение. Оператор **`return`** вызывает завершение работы функции и возврат значения, указанного в операторе.

Относительно самого возвращаемого значения пока заметим, что функция `main` в обычных случаях должна возвращать значение 0.

1.2 Вторая программа

Рассмотрим следующую задачу: *удалить из входного потока все пробельные литеры*.

```
 1 #include <stdio.h>
 2
 3 int main()
 4 {
 5         char c;
 6
 7         while (scanf("%c", &c) == 1) {
 8                 if (c != ' ') printf("%c", c);
 9         }
10         return 0;
11 }
```

1.2.1 Условный оператор

Условный оператор имеет следующий синтаксис:

```
if ( выражение ) оператор [ else оператор ]
```

Отличия от соответствующего оператора языка Паскаль следующие:

1. Круглые скобки являются частью оператора **`if`**, и не могут быть пропущены.
2. Ключевое слово **`then`** отсутствует.

1.2.2 Символьные константы

Замечание: слово «символ» — очень сильно перегружено разными смыслами, поэтому иногда употребляют слово «литера».

Литерные константы записываются в апострофах (одинарных кавычках). Например, `'a'` — литера `a`. Кроме того, для специальных символов применяется следующая запись: `'\''` — литера «апостроф», `'\\'` — литера «обратная косая черта» (backslash), `'\n'` — литера перехода на новую строку, `'\t'` — литера табуляции.

Обратите внимание, что литерные константы имеют тип **`int`**, а не **`char`**.

1 Занятие №3

1.1 Функции

На прошлом занятии нами был рассмотрен пример простой программы на языке Си. Было сказано, что программа является совокупностью функций и глобальных переменных. Кроме того, в программе должна присутствовать функция `main`, с которой начинается выполнение программы.

Рассмотрим более подробно способы определения новых функций. Замечу, что в языке Си нет разделения на процедуры и функции, всякий объект, который может исполняться, называется функцией.

Простейшее определение функции в общем виде выглядит следующим образом:

```
<result type> name(<params>) <compound-stmt>
```

Для обозначения того, что функция не возвращает значения, используется ключевое слово **void**. Например,

```
void f(void)
{
        /* ... */
}
```

В телах void-функций может отсутствовать оператор **return**, тогда возврат из функции произойдёт после того, как будут выполнены все операторы. Либо оператор **return** может присутствовать, но тогда не должен содержать возвращаемого выражения. Например,

```
1 void doprnt(int x)
2 {
3         if (!x) return;
4         printf("ERROR: %d\n", x);
5 }
```

Не void-функции должны завершаться оператором **return** содержащим выражение. *Если вы на какой-то ветке забыли **return**, вам может быть выдано предупреждение, а может и нет, тогда в этом случае будет возвращено неопределённое значение.*

Функция может возвращать значения произвольного типа, *кроме массива*. Допустимые типы результата - это целые и вещественные типы, структуры (будем рассматривать), указатели, перечислимые типы — кроме массива. Как мы позже увидим, массивы вообще являются «необычным» типом в Си.

Для обозначения того, что функция не имеет никаких аргументов, используется то же самое ключевое слово **void**, которое записывается вместо параметров функции. Например,

```
int f(void)
{
        /* ... */
}
```

Это функция, которая не имеет параметров и возвращает целое значение.

Обратите внимание, что пустой список параметров (т. е. `int f()`) обозначает функцию, число и тип параметров которой неизвестны. Это не имеет смысла, если мы рассматриваем только определения функций, но очень заметно, когда функции, так объявленные, вызываются. В этом случае, компилятор не будет проверять соответствие коли-

чества и типов фактических параметров количеству и типам формальных параметров. Применяются специальные правила приведения типов. В одном месте программы такая функция может быть вызвана, скажем, с двумя целыми параметрами, а в другом месте — с тремя вещественными. Такое истолкование пустого списка параметров отличается от языка Си++, в котором это обозначает функцию без параметров.

Определения функций в «старом стиле» мы вообще не будем рассматривать.

Типы формальных параметров функции задаются следующим образом:

```
(<type1> <par1>, <type2> <par2>, ..., <typen> <parn>)
```

причём для каждого параметра каждый раз указывается его тип. `f(int a,b)` — неправильная запись.

Все параметры передаются по значению. Передачи параметров по ссылке не существует. Для получения нужного эффекта используются указатели. *Единственное исключение — массивы*, которые и здесь ведут себя по-другому.

Функции не могут быть определены внутри других функций. Поэтому программа на Си состоит из "плоских"функций и глобальных переменных.

Вызов функции в теле другой функции записывается естественным образом:

```
<имя>(<факт. параметры>)
```

Если у вызываемой функции нет параметров, пара скобок все равно должна быть указана. Если функция не **void**, полученное значение может дальше использоваться в выражении, *но может и игнорироваться*. а если **void**, то попытка использовать значение функции будет считаться ошибкой.

Когда в теле некоторой функции компилятор встречает конструкцию `name(<pars>)`, он считает это вызовом функции (не всегда!). Возможны три случая, в зависимости от кого, как было ранее определено имя `name`.

1. Если `name` была объявлена как функция с явным списком параметров, то проверяется соответствие количества и типов фактических параметров количеству и типу формальных параметров, при необходимости производится приведение типов. Тип возвращаемого значения определён самой функцией.

2. Если `name` была объявлена как функция с неявным (пустым) списков параметров, никакие проверки типов фактических параметров не производятся, делаются стандартные приведения типов. Тип возвращаемого значения определён самой функцией.

3. Если `name` *вообще не была раньше объявлена*, предполагается, что эта функция имеет определение `int name()`, то есть возвращает тип **int**, с неопределённым списком параметров. Если впоследствии функция будет объявлена с типом **int** и явным списком параметров, компилятор может выдать предупреждение. Если она будет объявлена не как **int** — ошибка.

   В новом стандарте языка **C99** использование функций без предварительного определения **запрещено**.

Случай 3, рассмотренный выше, считается очень плохим стилем программирования. Мы будем полагать, что функция должна быть всегда описана перед её использованием. Для случая рекурсивных функций, или если желательно разместить функции в определённом порядке в файле, функцию перед использованием не обязательно полностью определять, т. е. описывать её тело, а достаточно описать. Для этого нужно задать имя функции, формальные параметры, возвращаемое значение, а тело функции опустить. Например,

```
int gcd(int a, int b);
```

Описание функции также называется *прототипом*. После такого описания функцию можно использовать. Имена формальных параметров могут быть здесь опущены, или могут не совпадать с именами формальных параметров в месте действительного определения функции — это не имеет значения. Например,

```
int gcd(int, int);
```

1.2 Определение глобальных переменных

Если переменная описана вне тела функции, эта переменная является глобальной, то есть доступна из всех точек программы, которые расположены ниже точки определения этой переменной. Глобальная переменная может перекрываться локальной переменной с тем же именем.

Переменная в момент определения может быть инициализирована начальным значением. Например,

```
int a = 10;
double b = 1.0, c = 1.34e-4;
```

Переменная, описанная вне тела функции может быть инициализирована только константным выражением, то есть выражением, значение которого может быть вычислено на этапе компиляции. Такие переменные по умолчанию получают значение 0 соответствующего типа. Блочные переменные могут инициализироваться произвольным выражением, но если переменная не была инициализирована явно, *её начальное значение не определено*. Блочная переменная существует все время, пока выполняется блок. Как только управление покидает блок, переменная уничтожается и память под неё освобождается.

```
1 int x(int a)
2 {
3         int y = a + 2;
4         return g(y);
5 }
```

Для объектов языка Си действуют правила перекрытия имён, когда имя, объявленное в самом вложенном блоке относительно текущей точки определения, перекрывает имена, находящиеся в объемлющих блоках или глобальные имена. Пример:

```
1   int x = 10;
2   int y()
3   {
4     int x = 15;
5     printf("%d\n", x);
6     {
7       int x = 20;
8       printf("%d\n", x);
9     }
10    printf("%d\n", x);
11  }
```

На одном и том же уровне вложенности не может быть определено двух объектов с одинаковым именем. Вопрос. Допустимы ли следующие определения:

```
int a(int a) { ... }
int a() { int a; ... }
```

```
int a(int a) { int a; ... }
```

1.3 Массивы

Рассмотрим первый составной тип — массив. Определение простейшего одномерного массива записывается следующим образом:

```
<тип> <имя> [ <число элементов> ]
```

выражение в квадратных скобках должно быть константным, то есть вычислимым при компиляции программы, то есть не использовать переменных. В квадратных скобках указывается число элементов в массиве. Такое определение отводит память для массива из заданного числа элементов заданного типа, например

```
double val[40];
```

отводит память под массив из 40 элементов типа **double**.

Для доступа к массиву используется запись вида `<имя массива>[<индекс>]`, индекс может быть произвольным выражением. *Элементы массива всегда нумеруются от 0 и до значения, равного количеству элементов в массиве минус 1.* Таким образом, элементы описанного выше массива перечисляются так:

```
val[0], val[1], val[2], ..., val[38], val[39].
```

Обратите внимание, что *выражение `val[40]` индексирует элемент, не принадлежащий массиву.* Это очень распространённая ошибка. *При обращении к элементам массива контроль индексов не производится*, то есть можно написать, например, `val[-5]`, `val[60]`. Компилятор не заметит никакой ошибки, ваша программа скомпилируется и при выполнении будет давать неправильный результат.

Язык Си не имеет типа диапазона, поэтому и *переполнения при выполнении операций с целыми числами тоже не контролируются*.

Рассмотрим пример — вычисление среднего арифметического элементов массива.

```
 1 double x[10];
 2 double s = 0;
 3 int    i;
 4
 5 i = 0;
 6 while (i < 10) {
 7          s = s + x[i];
 8          i = i + 1;
 9 }
10 s = s / 10;
```

Этот фрагмент содержит несколько избыточных конструкций языка:

1. неудобная форма цикла **while**;
2. три раза повторяющееся число 10;
3. неудобная запись вида `i = i + 1`

1.4 Цикл `for`

```
for (<init expr>; <test expr>; <incr expr>) <stmt>
```

в точности совпадает с

```
<init expr>;
while (<test expr>) {
  <stmt>
  <incr expr>;
}
```

кроме того, в цикле for каждый из трёх элементов может быть опущен. Если опущено `<test expr>`, то считается, что вместо него стоит число 1, то есть цикл будет выполнятся всегда. В языке Си цикл **`for`** не нагружен дополнительными семантическими особенностями, в отличие от языка Паскаль, например, запретом изменять переменную цикла внутри цикла, или однократным вычислением предельного значения, или неопределённостью счётчика цикла после завершения выполнения цикла.

1.5 Описание целых констант

Константы имеющие целое значение описываются следующим образом.

```
enum { <name> = <value> };
```

Например,

```
enum { N = 10 };
```

На самом деле это — определение перечислимого типа, содержащего единственную константу `N`, имеющую значение 10.

Константы, объявленные таким образом, подчиняются правилам видимости идентификаторов языка Си.

1.6 Сокращённые операции

```
a = a <op> b
```

можно заменить на

```
a <op>= b
```

Такие операции присваивания определены для всех бинарных операций, которые мы рассмотрели: `+, -, *, /, %`.

Инкрементные (декрементные) операции:

```
a++, ++a, a--, --a
```

увеличивают (уменьшают) значение переменной на 1. Они применимы только к интегральным переменным.

1.7 Передача массивов в функции

При передаче параметров в функции и возврате значений из функций массивы ведут себя особенным образом. Во-первых, как уже было сказано ранее, *функция не может вернуть массив*. Во-вторых, в отличие от всех других типов языка, *массивы в функции передаются по ссылке*. Это записывается следующим образом, например

```
double sum(double a[])
```

Обратите внимание, что *выражение в квадратных скобках опущено*. На самом деле, там может стоять произвольное выражение (константное), но оно ничего не будет значить. В любом случае *в функцию может быть передан массив произвольной длины. Никакой неявной информации о том, сколько элементов содержит массив, не передаётся*. Если вам нужна такая информация, вы должны ввести отдельный параметр.

```
1 double average(int n, double a[])
2 {
3         int    i;
4         double s = 0;
5
6         for (i = 0; i < n; i++)
7                 s += a[i];
8         return s / n;
9 }
```

1.8 Упражнения

1. Написать рекурсивную функцию вычисления чисел Фибоначчи.

2. Написать функцию, которая подсчитывает частоты распределения символов во входном потоке.

1 Занятие №4

1.1 Идентификаторы в Си

Идентификаторы начинаются с латинской буквы (большой или маленькой) или знака подчёркивания, далее идут латинские буквы, цифры или знаки подчёркивания. Длина идентификатора неограничена, отдельные компиляторы могут накладывать ограничения на максимальное число значащих символов. На внешние имена тоже могут накладываться ограничения. *Регистр букв является значимым*, то есть `a` и `A` — это два разных идентификатора. Некоторые идентификаторы зарезервированы для использования в качестве ключевых слов, например **`int`**, **`do`**. В программах не рекомендуется определять и использовать идентификаторы, начинающиеся со знака подчёркивания. Такие идентификаторы зарезервированы для внутреннего использования стандартными библиотеками.

1.2 Литеральные значения

1.2.1 Целые

Целые значения могут записываться в программе в десятичной, восьмеричной и шестнадцатеричной системе счисления. *Восьмеричная константа* начинается с символа `0` («ноль»), за которым идут цифры `0`—`7`, например `0377`. *Шестнадцатеричная константа* начинается с символов `0x`, затем идут цифры `0`—`7`, `a`—`f`, `A`—`F`. Пример `0xFF`. *Десятичная константа* начинается с цифр `1`—`9`, далее идут цифры `0`—`9`.

Тип константы — это минимальный тип, который может содержать данное значение. Для констант, заданных в десятичной форме, последовательно выбираются **`int`**, **`long`**, **`unsigned long`**, **`long long`**, **`unsigned long long`** (для C99). Для констант, заданных в восьмеричной или шестнадцатеричной форме, последовательно выбираются **`unsigned int`**, **`unsigned long`**, **`unsigned long long`** (для C99).

Тип константы можно задать явно указанием суффикса `u` или `l`. Эти суффиксы могут употребляться совместно. Например, `10u` — константа 10 типа **`unsigned int`**, `6uL` — константа 6 типа **`unsigned long`**. `7ll` — константа 7 типа **`long long`**.

1.2.2 Вещественные

Вещественная константа может содержать целую и дробную часть мантиссы и порядок. Вещественная константа записывается в десятичной системе счисления. По умолчанию вещественные константы имеют тип **`double`**. Для явного задания типа можно использовать суффикс `l` или `L` для указания типа **`long double`**. Суффикс `f` или `F` для указания типа **`float`**.

1.2.3 Литерные

Литерные константы имеют тип **`int`**. В апострофах может быть записано несколько символов, в этом случае соответствующее целое значение зависит от компилятора.

1.2.4 Строковые

Строковые литералы имеют тип **`char const *`** (константный указатель на тип **`char`**). Все строковые константы имеют в конце строки неявный символ с кодом `'\0'` — термина-

тор строки. Строка не имеет явного поля длины, в отличие от языка Turbo Pascal.

1.3 Простейшая работа со строками

Строки — это последовательность символов. Строки могут храниться в массивах типа **char**, **signed char** или **unsigned char**. Переменная для хранения строки определяется как массив

```
char str[N];
```

где N задаёт объем памяти, отводимый для строки. Поскольку строка всегда хранит символ-терминатор '\0', максимальное число значащих символов в такой строке на единицу меньше (N-1 символ).

Строки можно инициализировать следующим образом:

```
char str[20] = "a␣string";
```

В этом примере будет задано значение 9 байт (8 значащих символов и один символ-терминатор), остальные 11 байт будут либо обнулены, либо содержать произвольное значение.

Символьные массивы можно определять без указания размера, если присутствует инициализация. Например,

```
char x[] = "xxx";
```

В этом случае под массив x будет выделено 4 байта.

Для манипуляций со строками в языке Си используются стандартные библиотечные функции. Чтобы их можно было использовать, в начале программы нужно подключить заголовочный файл string.h.

```
#include <string.h>
```

Некоторые из этих функций мы рассмотрим.

Функция strcmp, прототип которой упрощённо записывается

```
int strcmp(char s1[], char s2[]);
```

сравнивает две строки s1 и s2. Если строка s1 лексикографически меньше строки s2, функция возвращает отрицательное значение. Если строка s1 лексикографически больше строки s2, функция strcmp возвращает положительное значение. Если две строки равны, функция возвращает ноль. Проверка на равенство двух строк записывается несколько неожиданным способом

```
if (!strcmp(s1, s2)) { /* ... */ }
```

Обратите внимание, что *строки нельзя сравнивать обычными операциями сравнения* !=, == и т. д. Дело в том, что в этом случае будут сравниваться не значения строк, а адреса этих строк. Компилятор в этом случае не даст никаких предупреждений.

Функция strlen с прототипом

```
int strlen(char s[]);
```

вычисляет количество значащих символов в строке, то есть число символов до символа-терминатора. Например strlen("xx") даёт результат 2.

Функция strcpy с прототипом

```
char *strcpy(char dst[], char src[]);
```

копирует строку str в строку dst, включая символ-терминатор. Как и везде в языке Си, контроль переполнения массива полностью лежит на программисте. Существует вариант этой функции (strncmp), который позволяет ограничить максимальное количество копируемых символов.

Функция strcat с прототипом

```
char *strcat(char dst[], char src[]);
```

добавляет строку `str` в конец строки `dst`. Существует вариант этой функции (`strncat`), который позволяет ограничить число добавляемых символов.

Строку очистить можно с помощью функции `strcpy`

```
strcpy(s, "");
```

а можно проще:

```
s[0] = 0;
```

1.4 Ввод/вывод строк

Для вывода строк предусмотрен специальный спецификатор формата `%s` который можно использовать в функциях семейства `printf`. Например,

```
printf("His␣name␣is␣%s\n", name);
```

Символ-терминатор на печать выводиться не будет.

Считывать строки можно аналогичным спецификатором формата `%s` для функций семейства `scanf`. Например,

```
char buf[20];
scanf("%s", buf);
```

В этом случае сначала будут пропущены все пробельные символы, затем все символы до первого пробельного символа будут занесены в `buf`. После этого в `buf` будет добавлен символ-терминатор `'\0'`. Обратите внимание на отсутствие знака `'&'` перед `buf` в аргументах вызова `scanf`. Буфер `buf` должен быть достаточного размера, чтобы вместить все символы вводимой строки. Поскольку вводимые данные чаще всего не находятся под контролем программиста, то есть пользователь, возможно злонамеренно, может вводить строки произвольной длины, *использование такого неконтролируемого чтения в реальных программах очень опасно*.

Другой полезный спецификатор для `scanf` — `%n`. Этому спецификатору должна соответствовать переменная целого типа со знаком `&` (адрес). В эту переменную заносится число символов, прочитанное к моменту, когда был встречен этот спецификатор. Спецификатор `%n` не учитывается в числе успешно считанных спецификаторов в возвращаемом значении функции `scanf`.

Очень полезной является функция `sprintf` с прототипом

```
int sprintf(char s[], char format, ...);
```

Которая работает точно также, как `printf`, поддерживает все форматы вывода, но вместо печати на стандартный поток вывода, заполняет строку `s`. В конец строки `s` добавляется символ-терминатор. Результатом работы функции является количество выведенных символов (не считая символ-терминатор). Например,

```
sprintf(s, "%d", n);
```

позволяет получить в строке `s` символьное представление числа `n`.

«Обратная» функция `sscanf` с прототипом

```
int sscanf(char s[], char format, ...);
```

считывает из форматные данные из строки вместо стандартного потока ввода. Функция возвращает число успешно считанных полей (как и `scanf`). Например,

```
sscanf(s, "%d", &n);
```

считывает в переменную `n` число из строки `s`.

Слово «строка» в русском языке может обозначать два совершенно разных понятия. Во-первых, это просто цепочка символов (string); во вторых, это последовательность символов,

занимающая один ряд на экране или в напечатанном тексте (line). В первом случае иногда говорят «цепочка».

Очень часто бывает так, что стандартный поток ввода ассоциирован либо с терминалом, либо с файлом, который имеет текстовую структуру, то есть разбит на строки (line). В языке Си входной поток является последовательностью символов. Строки текстового файла во входном потоке разделяются символом `'\n'`, не зависимо от того, какой разделитель строк текста в действительности используется в операционной системе (например, в MS-DOS используются два символа `'\r'`, `'\n'`). Входной поток не имеет специального символа-признака конца входного потока. Таким образом, входной файл вида

```
a
b b
cc
```

будет представлен как поток символов

```
'a', '\n', 'b', ' ', 'b', '\n', 'c', 'c', '\n', '\n'
```

Функция `gets` с прототипом

```
char * gets(char s[]);
```

считывает одну строку входного текстового файла, то есть последовательность символов до символа `'\n'`, включая его. Считанные символы помещаются в строку `s`, причём символ `'\n'` заменяется на символ-терминатор `'\0'`. В случае, если достигнут конец входного потока, функция возвращает специальную константу `NULL`. Обратите внимание, что когда входной поток ассоциирован с терминалом, для того, чтобы был отмечен конец потока, нужно нажать специальную комбинацию клавиш. Буфер `s` должен быть достаточного размера, чтобы вместить всю считываемую последовательность символов. *Использование этой функции опасно.*

Функция `puts` с прототипом

```
char *puts(char s[]);
```

добавляет в выходной поток символы из строки `s`. Символ-терминатор заменяется на символ завершения строки `'\n'`.

1.5 Побочный эффект в операциях инкремента и декремента

Основной эффект операции — это выработка некоторого значения, которое может быть далее использовано в выражении. Побочный эффект — это другое воздействие на среду выполнения (например, изменение значения переменной).

Нами уже были рассмотрены операции инкремента (увеличения на 1) переменной `++`, и декремента (уменьшения на 1) переменной `--`. В языке Си они существуют в двух формах: префиксной и постфиксной, которые отличаются основным эффектом операции. Преинкремент `++a` — значение переменной вначале увеличивается на 1, и это уже увеличенное значение является значением операции, и далее в выражении может быть использовано. Постинкремент `a++` — значением этой операции является значение переменной до увеличения на 1. Например,

```
int a = 5, b, c;
b = 5 + ++a;
c = 5 + a++;
```

значением переменной `b` будет 11, а переменной `c` — 10.

1.6 Операторы передачи управления

1.6.1 Оператор `break`

Оператор **`break;`** передаёт управление за пределы самого вложенного оператора **`do`**, **`while`**, **`for`** или **`switch`**. Оператор **`break;`** не может использоваться вне этих операторов. Например,

```
1 switch (x) {
2   case a:
3     while (1) {
4       if (y) break;
5     }
6     break;
7   default:
8     ;
9 }
```

1.6.2 Оператор `continue`

Оператор **`continue;`** передаёт управление за оператор, составляющий тело цикла **`do`**, **`while`** или **`for`**, что означает немедленное начало следующей итерации цикла. Для оператора **`do`** произойдёт переход на вычисление управляющего выражения **`while`** в конце цикла, для оператора **`while`** произойдёт переход на вычисление управляющего выражения в начале цикла. Для оператора **`for`** произойдёт переход к вычислению третьего выражения заголовка цикла **`for`**.

1.7 Упражнения

1. Написать функцию, которая реверсирует строку, переданную в качестве параметра.
2. Написать функцию, которая в массиве из целых чисел все числа, меньшие или равные заданному переставляет в начало массива, а все числа, большие или равные заданному — в конец массива.
3. Проверить на равенство строки (функция strcmp).
4. Скопировать из входного потока в выходной все строки, длина которых больше 20.
5. Скопировать из входного потока в выходной все строки, которые содержат целое число, большее 666.

1 Занятие №5

1.1 Перечислимые типы

Язык Си имеет средства для определения перечислимых типов. В общем виде объявление перечислимых типов выглядит следующим образом:

```
enum <тег перечисления> { <список констант> };
```

например,

```
enum traffic_light { green, yellow, red };
```

Тег перечисления может отсутствовать, тогда определяется анонимное перечисление, которое не вводит новый тип, а вводит только новые перечислимые константы.

Если тег перечисления присутствует, то вновь введённый тип можно впоследствии использовать для определения новых переменных, структур, других типов. Имя перечислимого типа состоит из двух компонент: ключевого слова **enum** и тега перечисления. Они всегда должны использоваться вместе. Например,

```
enum traffic_light light = red;
```

Все теги перечислимых типов находятся в отдельном от обычных идентификаторов пространстве имён, то есть обычные идентификаторы и теги перечислимых типов существуют как бы «параллельно». Тег перечислимого типа перекрывает только другие теги. На одном лексическом уровне тег должен быть уникальным среди других тегов.

Константы перечислимого типа находятся в пространстве имён идентификаторов, то есть они взаимно перекрывают идентификаторы, и должны быть уникальны с идентификаторами на одном уровне вложенности. Пример,

```
1  enum a { c1, c2 };
2  enum b { c3 };
3  enum a a;
4
5  int f()
6  {
7    enum a { c3 }; // перекрывает только тип enum a, но не переменную a.
8                   // константа c3 перекрыла константу c3 типа enum b
9    enum a b = c1;
10
11   a = c3;        // доступ к глобальной a, присваивание ей
12                  // локальной константы c3
13 }
```

Константам перечислимого типа можно указывать начальное значение. Оно должно быть константным выражением. По умолчанию первая константа получает значение 0, а последующие — значение предыдущей, увеличенное на 1. Например,

```
enum {a = 4, b, A = a, c = -4};
```

Обратите внимание, константы перечисления могут участвовать в константных выражениях, которые требуются при инициализации перечислимых констант и, например, при определении размера массива.

Перечислимый тип может иметь любой целый тип, достаточный для хранения значений всех его констант. Например, для перечисления

```
enum {a = 4, b, c};
```

компилятор может выделить тип **`signed char`**, а может и **`int`**. Узнать это можно операцией **`sizeof`**. Константы перечислимого типа могут использоваться везде, где требуется целое значение.

1.2 Выражения

1.2.1 Перечисление операций

1. Наивысший приоритет имеют постфиксные операции `++, --, [], (), ., ->`. Читаются слева направо.

`++`	постинкремент
`--`	постдекремент
`[]`	индексация массива, например `a[i]`
`()`	вызов функции, например `sin(M_PI)`
`.`	доступ к полю
`->`	доступ к полю через указатель

2. Следующая группа операций — префиксные операции. Читаются справа налево.

`++`	преинкремент
`--`	предекремент
`&`	взятие адреса
`*`	разыменование
`-`	
`+`	
`!`	логическое отрицание
`~`	побитовое отрицание
`sizeof`	вычисление размера типа
`()`	приведение типа

3. Остальные операции по мере убывания их приоритета. Читаются слева направо.

`*`	
`/`	
`%`	
`+`	
`-`	
`<<`	сдвиг влево
`>>`	сдвиг право
`<`	
`<=`	
`>`	
`>=`	
`==`	
`!=`	
`&`	побитовая операция AND
`^`	побитовая операция XOR
`|`	побитовая операция OR
`&&`	логическое «и»
`||`	логическое «или»
`?:`	условная операция, читается справа налево
`=`	присваивания, читаются справа налево
`*=`	
`/=`	
`%=`	
`+=`	
`-=`	
`<<=`	
`>>=`	
`&=`	
`^=`	
`|=`	
`,`	последовательное вычисление

Операции `&&` и `||` вычисляются по «короткой» схеме, то есть если значение всего выражения известно после вычисления первого операнда, второе выражение не вычисляется. Операцию `a && b` можно понимать так:

```
if (a) { if (b) 1; else 0; } else 0;
```

а операцию `a || b` так:

```
if (a) 1; else if (b) 1; else 0;
```

Операция `?:` тернарная, то есть имеет три операнда, например `a?b:c`. Сначала вычисляется условие `a`. Если оно истинно, тогда вычисляется `b`, а `c` не вычисляется, а если `a` ложно, тогда вычисляется `c`, а `b` не вычисляется. Эта операция читается справа налево, то есть выражение `a?b:c?d:e` эквивалентно `a?b:(c?d:e)`, выражение `a?b?c:d:e` эквивалентно `a?(b?c:d):e`.

Операция `,` («запятая») последовательно вычисляет оба аргумента. Значением операции является значение второго аргумента, значение первого теряется.

1.2.2 Преобразования типов при вычислении выражений

Перед вычислением арифметических операций транслятор выполняет два действия с аргументами: *целочисленное повышение* (integral promotion) и *балансировка*.

Целочисленное повышение. Кроме случая, когда выражение является аргументом операции **`sizeof`**, целочисленное выражение может иметь один из следующих типов:

- **`int`**
- **`unsigned int`**
- **`long`**
- **`unsigned long`**
- **`long long`**
- **`unsigned long long`**

Если выражение имеет тип, не перечисленный выше, транслятор повышает его до одного из этих типов. Если все значения исходного типа представимы также и значениями типа **`int`**, результатом повышения будет тип **`int`**. В противном случае результатом повышения будет тип **`unsigned int`**.

Таким образом, для типов **`signed char`**, **`short`** и для знаковых битовых полей результатом повышения является тип **`int`**. Для всех оставшихся целых типов предпочитается повышение в **`int`** когда это возможно, но в **`unsigned int`**, если это необходимо для сохранения значения во всех возможных случаях.

Балансировка. В случае вычисления инфиксного выражения, которое имеет два арифметических операнда, транслятор определяет тип выражения с помощью балансировки типов операндов. Для балансировки типов транслятор применяет следующие правила к повышенным типам операндов.

- Если балансируемые типы аргументов не **`unsigned int`** и **`long`**, результатом балансировки является тот повышенный тип операнда, который встречается позднее в последовательности типов **`int`**, **`unsigned int`**, **`long`**, **`unsigned long`**, **`long long`**, **`unsigned long long`**, **`float`**, **`double`**, **`long double`**.
- Если два балансируемых типа — **`unsigned int`** и **`long`** и тип **`long`** может представлять все значения типа **`unsigned int`**, результатом балансировки является тип **`long`**.

- В противном случае результатом балансировки является тип **`unsigned long`**.

Каждый из операндов преобразовывается в балансированный тип, арифметическая операция выполняется над значениями одинакового типа, и результат операции имеет балансированный тип.

1.2.3 Порядок вычисления выражения и побочные эффекты

Порядок, в котором транслятор вычисляет подвыражения неопределён и зависит от реализации. Например, оператор

```
y = *p++;
```

может быть эквивалентно либо

```
temp = p; p += 1; y = *temp;
```

либо

```
y = *p; p += 1;
```

Если в программе встретилось выражение

```
f() + g()
```

компилятор может расположить вызовы функций `f` и `g` в произвольном порядке.

При вызове функции, например `f(a, b)`, компилятор может вычислять выражения в произвольном порядке.

Порядок вычисления выражения важен, когда выражение имеет некоторый побочный эффект, например, заносит значение в переменную, либо модифицирует состояние файла.

Программа на Си содержит *точки согласования*. В точках согласования точно известно, какие побочные эффекты имели место, а какие должны произойти. Например, каждое выражение, записанное как оператор имеет точку согласования в конце. Гарантируется, что в фрагменте

```
y = 37;
x += y;
```

значение 37 будет помещено в `y` до того, как значение `y` будет использовано при вычислении `x`.

Точки согласования могут находится внутри выражения. Операции «запятая», вызов функции, логическое «и», логическое «или» содержат точки согласования. Например,

```
if ((c = getchar()) != EOF && isprint(c))
```

`isprint(c)` будет вычислено только после того, как новое значение, возвращённое `getchar()`, будет занесено в переменную `c`.

Между двумя точками согласования один и тот же объект может модифицироваться только один раз, и значение, читаемое из модифицируемого объекта может использоваться только для вычисления нового значения этого объекта.

Например,

```
val = 10 * val + (c - '0'); // хорошо
i   = ++i + 2;              // плохо
```

1.3 Многомерные массивы

Двухмерные массивы (матрицы) определяются следующим образом

```
<тип> <имя>[разм. 1][разм. 2];
```

Обратите, что каждое из измерений массива записывается в отдельной паре квадратных скобок. Массивы большей размерности определяются аналогично.

В памяти массивы размещаются по строкам, то есть быстрее всего изменяется последний индекс. Например для матрицы `int x[2][3]` элементы в памяти будут размещены следующим образом

```
x[0][0],x[0][1],x[0][2],x[1][0],x[1][1],x[1][2]
```

Поскольку матрица располагается по строкам, выражение `x[i]` имеет тип `int [3]`, то есть каждую строку матрицы можно рассматривать как целое (например, передавать параметром в функции). Аналогично и для массивов большей размерности.

Обращение к элементу массива выглядит так `x[i][j]`. Каждый индекс всегда записывается в своей паре квадратных скобок.

При передаче многомерных массивов в функции у них должны быть указаны все измерения, *кроме первого*. Это связано с тем, что для вычисления адреса элемента массива компилятор должен знать все измерения, кроме первого (напишите формулу вычисления адреса и проверьте!). Поэтому, функция матричного умножения может иметь следующий прототип.

```
double a[K][L], b[L][M], c[K][M];
void mult(double a[][L], double b[][M], double c[][M]);
```

1 Занятие №6

1.1 Структуры

Ранее мы рассмотрели методы определения и работы с массивами и перечислимыми типами. Теперь рассмотрим определение и использование *структур*. Структура в языке Си — это объединение нескольких компонент произвольных типов, то есть это аналог понятия «запись» в языке Паскаль.

Определение структуры имеет вид

```
struct <тег структуры>
{
  <список определений полей>
};
```

Список определений полей имеет точно такой же вид, как и в случае определения локальных или глобальных переменных (однако инициализация не допускается). Например,

```
1  struct circle
2  {
3    double x, y, r;
4    int    c;
5  };
```

Для того, чтобы определить переменную данного типа, как и в случае перечислимых типов, нужно использовать тег структуры вместе с ключевым словом **struct**. Например,

```
struct circle c1;
```

Обратите внимание, что в отличие от языка Си++ использование ключевого слова **struct** с тегом структуры *обязательно*.

Все теги (структур, объединений, перечислений) вместе образуют отдельное пространство имён, то есть теги не пересекаются с идентификаторами и метками.

Имена полей структуры должны быть уникальны в пределах этой структуры. В разных структурах допускаются поля, имеющие одинаковое имя.

Структура может содержать определения других структур или перечислимых типов, но в этом случае вложенное определение будет видно на том же лексическом уровне, на котором определена сама структура. Например,

```
1  struct foo
2  {
3    struct bar
4    {
5      double x;
6    };
7    int        y;
8    struct bar z;
9  };
10 struct bar t;
```

Является допустимым фрагментом на Си.

Для структурных типов допускается неполное определение типа. Вводится только тег структуры, но не определяется, какие поля содержит данная структура. Например,

```
struct tree;
```

Размер такой структуры неопределён, и попытка определения объекта такого типа (например переменной) вызовет ошибку до тех пор, пока структура не будет определена полностью, как показано выше. Тем не менее, можно определять переменные типа указателя на эту структуру.

Переменные одного и того же структурного типа можно присваивать друг другу. Два переменные считаются одного и того же типа, если при их определении был использован один и тот же тег структуры (эквивалентность типов по имени). Два типа структуры с разными тегами, даже содержащие одни и те же поля не считаются эквивалентными. Структуры можно передавать в качестве параметров функций (передаются по значению) и возвращать в качестве результата функции.

Доступ к полю переменной структурного типа производится указанием имени переменной, затем оператор «точка» и имя поля. Например,

```
a.b, a[10].b, a.foo.bar;
```

Размер структуры, определяемый операцией **sizeof**, может быть больше, чем сумма размеров всех составляющих её полей. Архитектура, на которой компилируется программа, может, например, требовать, чтобы целые числа были выравнены по границе слова. Поэтому компилятор может оставлять неиспользуемое пространство между полями для обеспечения требований архитектуры.

1.2 Объединения

Объединение — это структура данных, очень похожая на структуру. Определение объединения выглядит следующим образом:

```
union <тег объединения>
{
  <список определений полей>
};
```

Все прочие аспекты работы с объединениями, такие как правила перекрытия, доступ к полям — точно такие же, как и для структур. Единственное отличие структур от объединений состоит в том, что все элементы, составляющие объединение, располагаются по одному адресу, перекрывая друг друга. Например,

```
1 struct s { short  low, high; };
2 union  i
3 {
4   int      val_i;
5   struct s val_s;
6 };
```

Это способ, как можно разбить целое значение на младшую и старшую половины **short**, при условии, что тип **int** в два раза длиннее **short**. Обращаться можно без ограничений к любым полям, например

```
1   union i val_i;
2   val_i.val_i = x;
3   printf("%d,%d\n", val_i.val_s.low, val_i.val_s.high);
```

Весь контроль за правильностью таких манипуляций возлагается на программиста.

Типы объединения в основном используются для моделирования «вариантных записей», какие существуют в языке Паскаль. Например, если мы хотим определить тип структуры,

который может содержать информацию о геометрических фигурах, мы можем поступить, например, так

```
 1 enum fig_type {square, rect, circle, ellipse};
 2 struct fig_square  { int type; double a; };
 3 struct fig_rect    { int type; double a, b; };
 4 struct fig_circle  { int type; double r; };
 5 struct fig_ellipse { int type; double a, b; };
 6 union fig
 7 {
 8   int                type;
 9   struct fig_square  square;
10   struct fig_rect    rect;
11   struct fig_circle  circle;
12   struct fig_ellipse ellipse;
13 };
```

Работа с таким объединением может происходить, например так.

```
union fig figure;
figure.type = square;
figure.square.a = 1.4;
```

1.3 Указатели

Указатель — это переменная, которая хранит адрес другой объекта, например, переменной или функции. Аналогом являются ссылочные типы в языке Паскаль. Простейшее определение переменной указательного типа имеет вид:

```
<тип> *<переменная>;
```

Знак `'*'` («звёздочка») относится к имени переменной, а не к типу, поэтому определение

```
int *p, c;
```

определяет указатель на **int** и переменную типа **int**. Функция, возвращающая указатель, определяется похожим образом

```
char *strdup(char *ptr);
```

Аналогично массив указателей

```
int *aptr[20];
```

Допускаются двойные указатели, тройные указатели и т. д. Полные правила чтения и записи деклараторов мы рассмотрим позже.

Указатели могут хранить адрес любого объекта в программе, при условии, что типы объекта и указателя согласованы. На самом деле, поскольку адрес объекта, как правило, имеет размер, не зависящий от размера объекта, возможно практически неограниченное явное приведение одних указательных типов к другим.

Для взятия адреса объекта используется операция `&`. Например, выражение

```
p = &c;
```

присваивает переменной указательного типа `p` адрес переменной `c`. Точно так же могут браться адреса элементов массива, полей структур, локальных и глобальных переменных и пр.

Для разыменования указателя (взятия значения по адресу, хранящемуся в указателе) применяется унарная операция `*`. Например, `printf("%d\n", *p);`

В языке Си массивы и указатели тесно связаны друг с другом. Во-первых, имя массива является константным указателем на его первый элемент, соответственно любой указатель можно рассматривать как массив из некоторого числа элементов, в простейшем случае, как массив из одного элемента. Например, если есть определение переменной

```
int arr[32], *p;
```

то выражения `&arr[0]` и `arr` полностью эквивалентны друг другу. Например, после выполнения инструкции

```
p = arr;
```

указатель `p` будет содержать адрес нулевого элемента массива, и имя указателя `p` может везде использоваться для обращения к массиву `arr`. Например,

```
p[0] = 2;
c = p[3] + p[4];
```

Существует два отличия между переменной, объявленной как массив и переменной, объявленной как указатель: **во-первых**, имени массива нельзя присвоить никакого значения, то есть выражение вида

```
arr = &p[3];      /* неправильно! */
```

во-вторых, при определении переменной типа массива под эту переменную отводится память, достаточная для хранения заявленного в определении переменной числа элементов массива, а при определении указателя отводится память, достаточная для хранения указателя. Так, для массива `arr` будет отведено 128 байт (в предположении, что **`int`** занимает 4 байта), а для указателя `p` — только 4 байта (если указатель занимает 4 байта памяти).

Указатели одного типа можно сравнивать друг с другом на равенство и неравенство. Любой указательный тип может сравниваться на равенство или неравенство с целой константой 0. Любому указателю может присваиваться целая константа 0, что означает, что данный указатель не указывает ни на какую область памяти. Стандартная библиотека определяет константу `NULL`, которую можно использовать для символической (возможно более наглядной) записи нулевого указателя. Попытки записи или чтения по нулевому указателю обычно вызывают исключение операционной системы. Если же указатель не инициализирован, то он указывает на некоторую случайную область памяти. Выражение указательного типа может использоваться в условиях циклов, оператора **`if`** и т. д. Это, как всегда, обозначает неявное сравнение с нулём. Например:

```
for (p = head; p; p = p->next)
```

цикл с таким заголовком будет повторяться до тех пор, пока `p` не обратиться в нуль.

Указатель можно складывать с целой величиной. Результатом такой операции является указатель того же типа. Этот указатель будет указывать на элемент массива, отстоящий от исходного элемента массива на указанное число элементов. Например, если **`int`** `arr[10], *p = &arr[5];`, тогда

```
p + 3 == &arr[8];
p - 4 == &arr[1];
```

В языке Си справедливо тождество (на самом деле, это определение операции индексирования), связывающее операцию индексирования и арифметические операции с указателями

```
arr[ind] == *(arr + ind)
```

Соответственно, для указателей определены операции `-`, `+=`, `-=`, `++`, `--`.

Два указателя можно вычитать друг из друга. Эти два указателя должны указывать на элементы одного и того же массива (должны, естественно, иметь один и тот же тип). Результатом вычитания является целое число — разность между указателями, выраженная в числе элементов массива данного типа. Если `p` — указатель, то

```
(p + 1) - p == 1
```

для любого типа на который указывает переменная p.

В языке Си определён обобщённый указатель, который записывается как

```
void *ptr;
```

то есть как указатель на «тип» **void**. Такому указателю можно присваивать значения указателей любого типа, и такой указатель можно присваивать указателю любого типа (только в Си, но не в Си++). *Указатель обобщённого типа нельзя разыменовывать и с ним нельзя проводить арифметические операции* (но с нулём сравнивать можно).

В качестве примера рассмотрим возможную реализацию функции strcpy.

```
1 char *
2 strcpy(char *str1, char *str2)
3 {
4   char *d = str1;
5   while ((*d++ = *str2++));
6   return str1;
7 }
```

1.4 Работа с динамической памятью

В языке Си помимо области памяти, выделяемой во время компиляции программы для глобальных переменных, и области памяти в стеке, выделяемой для локальных переменных динамически при каждом вызове функции, существует область памяти (традиционно называемая «кучей»), представляющая собой нечто среднее между этими двумя типами памяти. Куча существует все время выполнения программы, но память в ней выделяется и освобождается динамически, по требованию программиста. Средства работы с динамической памятью не встроены в язык, а предоставляются стандартной библиотекой. Чтобы использовать функции работы с динамической памятью, необходимо подключить заголовочный файл

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

Определены следующие стандартные функции

```
void *malloc(size_t size);
```

выделяет в куче область памяти размера size и возвращает указатель на начало этой области памяти. Здесь size_t — это тип, определённый стандартом для представления величин, задающих размер объектов. Операция **sizeof** вырабатывает значение именно этого типа. Обычно тип size_t определяется эквивалентным типу **unsigned long int**. Если size равно нулю, результат работы функции неопределён.

```
void *calloc(size_t nitems, size_t nsize);
```

выделяет в куче область памяти для размещения массива из nitems элементов, каждый из которых имеет размер nsize, и возвращает указатель на выделенную область памяти. Выделенная область памяти инициализируется нулями. Если функции calloc или malloc не могут выделить область памяти достаточного размера, они возвращают нулевой указатель (0 или NULL). *Программист должен проверять результат, возвращаемый этими функциями и предпринимать действия по обработке ошибок*, если был возвращён нулевой указатель.

```
void free(void *ptr);
```

освобождает ранее выделенный в куче блок памяти и делает его доступным для повторного использования. Переданный функции free указатель должен быть получен от функций

malloc, calloc или realloc. Если аргумент функции не является таким указателем, результат работы функции free неопределён (чаще всего крах программы). Если блок памяти был уже ранее освобождён функцией free, повторное освобождение одного и того же блока памяти неопределено. После того, как блок был освобождён, с памятью в этом блоке нельзя проводить никаких операций, даже чтение. Не гарантируется, что значения, записанные в освобождаемой области памяти, будут сохранены после вызова free. Вызов free(0) безопасен и не производит никаких действий.

```
void *realloc(void *ptr, size_t newsize);
```

изменяет размер выделенного блока памяти. Если ptr == NULL, функция realloc работает в точности как malloc, а если newsize == 0, realloc работает как free. Иначе функция выделяет в динамической памяти блок размера newsize и копирует в него начало блока памяти по адресу ptr. Если новый блок больше старого, остаток памяти нового блока не инициализируется. Не гарантируется, что новый блок памяти будет начинаться с того же адреса памяти, что и предыдущий, даже если размер блока памяти был уменьшен. После функции realloc уже нельзя пользоваться старым блоком памяти, выделенным по адресу ptr. Если невозможно выделить область памяти заданного размера, функция возвращает нулевой указатель. В этом случае область памяти по старому адресу ptr не изменяется и по-прежнему доступна для использования.

1 Занятие №7

1.1 Операторы передачи управления

1.1.1 Оператор `switch`

Оператор выбора **`switch`** используется для передачи управления в зависимости от значения некоторого целого выражения. Оператор записывается следующим образом:

`switch (<выражение>) <оператор>`

Где `<оператор>` почти всегда составной оператор, но это не обязательно.

Внутри оператора могут располагаться метки **`case`**, записываемые следующим образом

`case <константное целое выражение>:`

Оператор вычисляет выражение в заголовке оператора **`switch`**, и затем переходит на метку **`case`** с равным значением. В операторе не может содержаться две метки **`case`** с равным значением. Если соответствующая метка **`case`** не найдена, производится переход на метку **`default:`**, если она существует, и выход из оператора в противном случае.

Метки **`case`** и **`default`** могут содержаться во вложенных операторах (например, **`while`** или **`for`**), но не во вложенных операторах **`switch`**.

Если внутри оператора **`switch`** встречается оператор **`break;`**, управление передаётся за оператор выбора. Оператор **`break;`** должен использоваться для выхода из оператора выбора после вычисления альтернативы, потому что в противном случае управление «провалится» на следующую метку **`case`**. Например,

```
1  switch (light) {
2    case red:
3      printf("Stop\n");
4      break;
5    case green:
6      printf("Go\n");
7      break;
8    case yellow:
9      printf("Be␣ready\n");
10   default:
11     printf("Unknown␣light\n");
12 }
```

1.1.2 Оператор `goto`

Оператор `goto <метка>;` передаёт управление на заданную метку. Метка должна находится в той же функции, помеченный оператор записывается `<метка>:`. Где `<метка>` — это произвольный идентификатор. Метки никак специально в функции не объявляются, и переходы на метки вперёд допустимы. Метки находятся в своём пространстве имён, отличном от идентификаторов и тегов, это значит, что имена меток могут совпадать с именами других объектов. Метки всегда локальны в пределах одной функции. Две метки с одним именем внутри одной функции не допускаются. Оператор перехода может входить в блок в обход инициализации, например:

```
1 if (x) goto label;
2 /* ... */
3 while (y) {
4   int z = 1;
5
6   /* ... */
7 label:
8   /* ... */
9 }
```

тогда при переходе на метку `label` под переменную будет выделена память, но она *не будет инициализирована*.

1.2 Инициализация составных типов

Массивы, структуры и перечисления точно также, как и простые типы могут быть проинициализированы в точке определения. Массивы типов **`char`**, **`signed char`**, **`unsigned char`** могут быть проинициализированы строкой. Для одномерного массива произвольного типа инициализация выглядит следующим образом:

```
<тип> <имя>[<размер>] = { <зн. 1>, <зн. 2>, ... <зн. K> };
```

Все инициализирующие значения должны иметь тип, совпадающий с типом массива. Их не должно быть больше, чем размер массива, если их меньше, чем размер массива, оставшиеся элементы инициализируются по умолчанию. Например,

```
char str[20] = { 'a', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0' };
```

Если массив содержит инициализацию, тогда размер массива может быть опущен. Он будет вычислен по количеству инициализирующих элементов.

Структура инициализируется похожим образом.

```
<тип> <имя> = { <зн. 1>, <зн. 2>, ..., <зн. K>};
```

В этом случае полям структуры последовательно присваиваются указанные значения. У объединений может быть проинициализирован только первый элемент.

Если массив сам имеет тип массива (многомерный массив), или тип структуры, или структура имеет поля типа массива или структуры, инициализаторы могут быть вложенными. Например

```
struct circle { double x, y, r; };
struct circle cc[2] = {{1.0, 2.0}, {1.3,2.0,0.4}};
```

Если фигурные скобки, отделяющие внутренние инициализаторы, опускаются, инициализация идёт подряд, переходя при необходимости границы типов. Например, если в указанном выше примере опустить внутренние фигурные скобки, `cc[1]` пример значение `{1.0,2.0,1.3}`, а `cc[2]` — `{2.0,0.4,0.0}` (если переменная `cc` — глобальная).

1.3 Передача параметров в программу

При запуске программы операционная система передаёт ей параметры, которые были указаны в командной строке. Если программа желает работать с переданными ей параметрами, заголовок функции `main` должен выглядеть следующим образом:

```
int main(int argc, char *argv[]);
```

Здесь первый параметр `argc` задаёт количество аргументов командной строки, а параметр `argv` — это массив указателей на строки, хранящие значения соответствующих аргументов. По соглашению `argv[0]` содержит само имя программы, а `argv[argc]` содержит 0 (`NULL`). Следовательно, если программе не было передано никаких аргументов, `argc` равно 1. Разбиение командной строки на аргументы запускаемой программы производит командный процессор, обычно аргументы разделяются друг от друга пробелами. Например, если программа была запущена командой

```
./prog1 f1 f2 ../f3
```

`argc` пример значение 4, `argv[0]` указывает на строку `"./prog1"`, `argv[1]` — на строку `"f1"` и т. д.

Значение, возвращаемое функцией `main` (или аргумент функции `exit`), — это «код возврата» программы. Он используется чтобы проинформировать вызвавшую программу о статусе завершения работы программы. Код возврата 0 означает, что работа программы завершилась нормально, ненулевые коды возврата означают, что при выполнении программы возникли какие-то проблемы.

Пример программы, которая печатает свои аргументы.

```
 1 #include <stdio.h>
 2 int main(int argc, char *argv[])
 3 {
 4         int i;
 5
 6         for (i = 0; i < argc; i++) {
 7                 printf("argv[%d] = %s\n", i, argv[i]);
 8         }
 9         return 0;
10 }
```

1.4 Функции работы с файлами

Мы ещё не рассмотрели две функции для посимвольной работы со стандартным потоком. Функция

```
int getchar(void);
```

вводит один символ из стандартного входного потока. Если символ введён успешно, возвращается код символа в диапазоне 0–255 (положительное число). В случае ошибки или конца файла возвращается значение `EOF` (оно обычно равно -1). Функция

```
int putchar(int c);
```

выводит один символ на стандартный поток вывода.

Обратите внимание, что функция `getchar` может возвращать 257 различных значений, поэтому *переменной типа* **`char`** *для хранения возвращаемого значения недостаточно!* Следующий пример копирует стандартный ввод на стандартный вывод.

```
1 #include <stdio.h>
2 int main()
3 {
4         int c;
5         while ((c = getchar()) != EOF) putchar(c);
6         return 0;
7 }
```

Стандартная библиотека содержит средства для работы с произвольными файлами. Для хранения информации об открытом файле используется структура `FILE`. Функции работы с файлами принимают или возвращают указатель на эту структуру. Функция

```
FILE *fopen(char *name, char *mode);
```

открывает файл с именем `name`. Строка `mode` содержит флаги, с которыми открывается файл.

`"r"`	открыть для чтения. Текущая позиция в файле устанавливается на начало файла.
`"w"`	открыть для записи. Если файл не существовал, он создаётся, если существовал, он очищается. Текущая позиция в файле устанавливается на начало файла.
`"a"`	открыть для добавления. Если файл не существовал, он создаётся. Текущая позиция в файле устанавливается на конец файла.
`"r+"`	открыть для чтения и записи. Текущая позиция в файле устанавливается на начало файла.
`"w+"`	открыть для чтения и записи. Если файл не существовал, он создаётся, если существовал, он очищается. Текущая позиция в файле устанавливается на начало файла.
`"a+"`	открыть для чтения и записи. Если файл не существовал, он создаётся. Текущая позиция в файле устанавливается на конец файла.

Если файл был открыт успешно, возвращается указатель на структуру, хранящую состояние открытого файла. Если при открытии произошла ошибка, возвращается `NULL`.

```
int fclose(FILE *stream);
```

Закрывает поток. Если закрытие прошло успешно, возвращается 0, иначе возвращается `EOF`.

При запуске программы открываются стандартные потоки `stdin`, `stdout`, `stderr`. Например, функции `putchar`, `puts` работают с потоком `stdin`, а функции `getchar`, `gets` работают с потоком `stdout`. Вы можете использовать эти имена для указания имён стандартных потоков в функциях, перечисленных ниже.

```
int getc(FILE *stream);
int putc(int c, FILE *stream);
int fscanf(FILE *stream, char *format, ...);
int fprintf(FILE *stream, char *format, ...);
```

Соответствуют функциям `getchar`, `putchar`, `scanf`, `printf`.

Функция

```
int fputs(char *str, FILE *stream);
```

записывает строку символов `str` в файл. Символ-терминатор строки отбрасывается. *В отличие от `puts` эта функция не дописывает `'\n'` в выходной файл.*

Функция

```
char *fgets(char *str, int size, FILE *stream);
```

Считывает самое большее `size - 1` символ из входного файла. Чтение останавливается по достижению конца файла или по достижению символа `'\n'`. Если `'\n'` считан, он добавляется в строку. После прочитанных символов добавляется символ-терминатор `'\0'`. При успешном завершении функция возвращает `str`, а при неудаче (конец файла или ошибка ввода) возвращается `NULL`. *Используйте эту функцию для чтения строк из входного файла, поскольку эта функция ограничивает максимальную длину считываемой строки.*

Следующий пример копирует содержимое заданного файла на стандартный вывод.

```
 1 #include <stdio.h>
 2 int main(int argc, char *argv[])
 3 {
 4   FILE *f;
 5   char  buf[256];
 6
 7   if (argc != 2) {
 8     fprintf(stderr, "Wrong number of arguments\n");
 9     return 1;
10   }
11   if (!(f = fopen(argv[1], "r"))) {
12     fprintf(stderr, "Cannot open file %s\n", argv[1]);
13     return 1;
14   }
15   while (fgets(buf, sizeof(buf), f))
16     fputs(buf, stdout);
17   fclose(f);
18   return 0;
19 }
```

Функция

```
int ungetc(int c, FILE *stream);
```

Заталкивает один символ обратно во входной поток, так что следующая функция `getc` считает именно этот символ. Таким образом затолкнуть назад можно не более одного символа.

Функция

```
int feof(FILE *stream);
```

возвращает ненулевое значение, если в структуре `FILE` установлен флаг ошибки или конца файла. Этот флаг устанавливается *по результату работы функций чтения или записи*. До первого чтения флаг сброшен, даже если файл пуст. В следующем примере последняя строка будет напечатана дважды.

```
 1 #include <stdio.h>
 2 int main(int argc, char *argv[])
 3 {
 4         char  buf[64];
 5         while (!feof(stdin)) {
 6                 fgets(buf, sizeof(buf), stdin);
 7                 fputs(buf, stdout);
 8         }
 9         return 0;
10 }
```

1 Занятие №8

1.1 Квалификатор `const`

При определении переменной её тип может дополняться так называемыми квалификаторами. Язык Си определяет два квалификатора **`const`** и **`volatile`**. Квалификатор **`volatile`** используется, в основном, для программирования на низком уровне, поэтому мы его рассматривать не будем. Квалификаторы могут стоять на любом месте в спецификации типа: до имени типа, после имени типа, и даже в середине имён типов, состоящих из нескольких ключевых слов, например:

```
const unsigned long
unsigned long const
unsigned const long
```

— все правильные спецификации типа и описывают один и тот же тип.

Квалификатор **`const`** означает, что определяемый объект не может быть модифицирован, потому что например, находится в ПЗУ.

```
const int nproc = 30;
```

определяет переменную `nproc` типа **`int`**, которая не может быть модифицирована в данной единице компиляции. Переменная, описанная с квалификатором **`const`**, не может быть использована в константных выражениях, которые, в частности, задают количество элементов массива. Следующий фрагмент не является правильным в языке Си:

```
const int N = 10;
double arr[N];
```

Если квалификатор **`const`** используется для определения переменной-указателя, его семантика меняется в зависимости от того, где он расположен: до символа `*` или после него. Определение

```
char *ptr;
```

вводит указатель `ptr`, который и сам может изменяться, и память по адресу, на который указывает данный указатель, также может быть изменена.

```
char const *ptr;
```

определяет указатель `ptr` на неизменяемую область памяти. Сам указатель может изменяться.

```
char * const ptr;
```

определяет неизменяемый указатель на изменяемую область памяти.

```
char const * const ptr;
```

определяет неизменяемый указатель на неизменяемую область памяти.

Из всех вышеперечисленных комбинаций чаще всего используется **`const char *`**. Такой тип, например, имеют параметры многих стандартных функций. Например, функция strcmp определена следующим образом:

```
int strcmp(const char *s1, const char *s2);
```

потому что она принимает в качестве параметров две строки, которые не модифицирует. Обратите внимание, что не имеет смысла писать **`const char * const`**, потому что параметры указательных типов в функции передаются по значению. *Рекомендуется пользоваться квалификатором **`const`** в ваших программах.*

1.2 Чтение сложных деклараторов

Ранее мы уже рассмотрели различные конструкции, модифицирующие тип, такие как указатели, массивы, функции. Все эти конструкции могут комбинироваться в синтаксической конструкции, называемой «декларатором». Таким образом, полное определение переменной выглядит следующим образом:

```
<базовый тип> <декларатор> [ = <инициализатор> ];
```

Декларатор содержит имя определяемого объекта, но в некоторых местах может быть «анонимным», то есть не содержащим имя определяемого объекта. Анонимные деклараторы допускаются в операции приведения типа и при описании формальных параметров в прототипах функций.

Пример декларатора:

```
char (*(*x[3])())[5];
```

Анонимный декларатор может выглядеть следующим образом:

```
char (*(*[3])())[5];
```

Такая (на первый взгляд «странная») форма определения производных типов на самом деле введена по аналогии с выражениями. Декларатор можно рассматривать как некоторое выражение над типом. В таком выражении есть три операции:

`[]`	массив из заданного количества элементов
`()`	функция с заданными параметрами
`*`	указатель
`()`	группировка членов в выражении

Постфиксные операции имеют самый высокий приоритет и читаются слева направо от определяемого имени. Префиксная операция имеет более низкий приоритет и читается справа налево. Скобки могут использоваться для изменения порядка чтения.

Таким образом, декларатор читается, начиная от имени определяемого объекта следуя правилам приоритетов операций. Имя определяемого объекта — это первое имя после базового типа.

Примеры:

`int a[3][4];`	массив из 3 элементов типа массива из 4 элементов типа **`int`** (матрица 3×4 целых)
`char **b;`	указатель на указатель на **`char`**
`char *c[];`	массив из неопределённого количества элементов типа указатель на тип **`char`**
`int *d[10];`	массив из 10 элементов типа указатель на тип **`int`**
`int (*e)[10];`	указатель на массив из 10 элементов типа **`int`**
`int *f();`	функция, возвращающая указатель на **`int`**
`int (*g)();`	указатель на функцию, возвращающую **`int`**
`int *(*g)();`	указатель на функцию, возвращающую указатель на **`int`**

1.3 Класс декларации `typedef`

Чтобы не нагромождать деклараторы и облегчить их чтение, введено специальное ключевое слово **`typedef`**. Оно записывается перед именем базового типа в декларации, например

```
typedef int *pint;
```

В этом случае имя `pint` определяется как синоним для типа `int *`, то есть далее в определениях переменных это имя можно использовать наравне с именем базового типа, например

```
pint a[10], f(), *p;
```

Конструкция **typedef** не вводит новый тип, а задаёт ещё одно имя для типа, которое может использоваться наравне со старым. Поэтому переменная `pint a;` и переменная `int *b;` имеют один и тот же тип `int *`.

Если есть **typedef**-имя и декларация, использующая это имя, то от **typedef**-имени можно избавиться, подставив декларируемое имя вместо **typedef**-имени в **typedef**-декларацию и добавив при необходимости скобки для того, чтобы порядок чтения не изменился. Например,

```
typedef void (*pfunc)(int);
pfunc signal(int, pfunc);
```

после преобразования получаем

```
void (*signal(int, void (*)(int)))(int);
```

1.4 Работа с бинарными файлами

Под *бинарным* файлом мы будем понимать файл, удовлетворяющий следующим условиям:

- файл рассматривается как последовательность байт, никакого деления файла на строки не подразумевается;
- данные в файле хранятся в двоичном, а не текстовом представлении.

Конечно, деление на текстовые и бинарные файлы достаточно условно. Текстовый файл иногда может оказаться удобно рассматривать как бинарный и работать с ним соответствующим образом.

Мы рассмотрим работу с бинарными файлами произвольного доступа, то есть с файлами, которые допускают произвольное позиционирование указателя текущего положения в файле, а не просто последовательное чтение от начала до конца.

1.4.1 Открытие бинарных файлов

В системах Unix формат текстовых и бинарных файлов совпадает, в других же системах эти два типа файлов могут обрабатываться по-разному. Поэтому при открытии бинарного файла с помощью функции `fopen` необходимо использовать специальный флаг открытия `"b"`. С учётом этого флага допустимые режимы открытия файла перечислены в таблице 1.

1.4.2 Чтение — функция **fread**

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *f);
```

Функция `fread` считывает данные из бинарного файла. Параметр `ptr` — это адрес начала буфера, в который будут записаны считанные данные. `size` — это размер одного элемента данных, а `nmemb` — это количество элементов данных, которые необходимо прочитать. `f` — дескриптор потока, из которого ведётся чтение.

Общее количество байт, которые необходимо прочитать, определяется перемножением параметров `size` и `nmemb`

```
bytes_to_read = size * nmemb;
```

`"rb"`	Открыть для чтения. Если файл не существует, `fopen` завершается с ошибкой. Текущая позиция в файле устанавливается на начало файла.
`"wb"`	Открыть для записи. Если файл не существовал, он создаётся, если существовал — очищается. Текущая позиция в файле устанавливается на начало файла.
`"ab"`	Открыть для добавления. Если файл не существовал, он создаётся. Текущая позиция в файле устанавливается на конец файла. Каждая операция записи будет перемещать указатель текущего положения в конец файла, затем записывать данные.
`"r+b"`	Открыть для чтения и записи. Если файл не существует, `fopen` завершается с ошибкой. Текущая позиция в файле устанавливается на начало файла.
`"w+b"`	Открыть для чтения и записи. Если файл не существовал, он создаётся, если существовал, он очищается. Текущая позиция в файле устанавливается на начало файла.
`"a+b"`	Открыть для чтения и записи. Если файл не существовал, он создаётся. Текущая позиция в файле устанавливается на конец файла.

Таблица 1: Режимы открытия файлов функции `fopen` для бинарных файлов

Далее делается попытка считать данное количество байт из дескриптора потока. Предположим, что было успешно считано `read_bytes` байт. Тогда возвращаемое значение функции `fread` вычисляется следующим образом:

```
retval = read_bytes / size;
```

Таким образом, возвращаемое значение — это количество элементов данных, которые были считаны целиком. В случае ошибки или наступления конца файла возвращается количество элементов, которые были полностью считаны до возникновения ошибки или конца файла.

Поскольку возвращаемое значение получается делением нацело на размер одного элемента данных, если размер одного элемента больше одного байта, и будет считано количество байт, не кратное размеру одного элемента, неполный элемент будет записан в память, но способа узнать о том, что он был записан, не существует. Поэтому размер одного элемента данных всегда нужно задавать равным 1, а возвращаемое значение сравнивать с запрошенным количеством байт. Здесь возможны следующие ситуации:

- Считано 0 байт. Это значит, что наступил конец файла, или возникла ошибка чтения. Чтобы узнать, завершилась ли функция `fread` по ошибке или по концу файла, нужно использовать функции `feof` и `ferror`.

- Считано байт меньше, чем запрошено, но количество байт кратно размеру одного элемента данных. В этом случае нужно обработать считанное количество элементов.

- Считано байт меньше, чем запрошено, и количество байт не кратно размеру одного элемента данных. В этом случае можно выдать ошибку о нарушении формата входных данных.

- Считано байт ровно столько, сколько запрошено. Нужно обработать считанные элементы.

1.4.3 Запись — функция fwrite

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *f);
```

Функция fwrite записывает данные в дескриптор потока. Параметр buf — это адрес начала буфера, в котором хранятся записываемые данные, size — это размер одного элемента данных, а nmemb — это количество элементов данных, которые необходимо записать. f — дескриптор потока, в который ведётся запись.

Общее количество байт, которые необходимо записать, вычисляется аналогично функции fread. Возвращаемое значение также получается делением нацело действительно записанного количества байт на размер одного элемента данных.

1.4.4 Позиционирование в файле

```
#include <stdio.h>

#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2

int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
```

Функция ftell позволяет получить смещение указателя на текущее положение в файле в байтах относительно начала файла. При ошибке функция возвращает -1.

Функция fseek позволяет переместить указатель на текущее положение в файле. Параметр stream задаёт дескриптор потока, связанный с файлом, параметр offset задаёт смещение в байтах, а параметр whence — точку, от которой отсчитывается смещение. Этот параметр может принимать три значения, перечисленные ниже.

SEEK_SET	смещение отсчитывается от начала файла
SEEK_CUR	смещение отсчитывается от текущего положения в файле
SEEK_END	смещение отсчитывается от конца файла

Если новое положение в файле находится за текущим концом файла, и файл открыт на запись или чтение/запись, файл расширяется нулями до требуемого размера. Если новое положение в файле находится до начала файла, возвращается ошибка.

При успешном завершении функция fseek возвращает 0, а при ошибке — -1.

Обратите внимание, что функции позиционирования могут быть неприменимы к стандартным потокам, потому что стандартные потоки могут быть связаны с устройствами, которые не допускают произвольное позиционирование (например, терминалы).

1.4.5 Изменение размера файла

```
#include <stdio.h>
#include <unistd.h>

int truncate(const char *path, off_t length);
```

Функция truncate позволяет изменить (увеличить или уменьшить) размер существующего файла с путём path. Новый размер файла будет равен значению, заданному в параметре length. Если файл увеличивается в размере, новая часть инициализируется нулями. Если файл уменьшается, старый остаток файла теряется.

1.4.6 Вычисление размера файла

Приведённая ниже функция позволяет получить размер файла по его имени. В качестве параметра ей передаётся имя файла, она возвращает размер файла при успешном завершении и -1 при ошибке.

```
 1  #include <stdio.h>
 2
 3  long fsize(char const *path)
 4  {
 5    FILE *f = 0;
 6    long size;
 7
 8    if (!(f = fopen(path, "r"))) return -1;
 9    if (fseek(f, 0, SEEK_END) < 0) goto error;
10    if ((size = ftell(f)) != -1) goto error;
11    fclose(f);
12    return size;
13
14   error:
15    if (f) fclose(f);
16    return -1;
17  }
```

1.4.7 Пример программы

Следующая программа принимает в качестве параметров командной строки список имён файлов. Предполагается, что эти файлы содержат целые числа в бинарном виде. Программа увеличивает все числа в файле на единицу.

```
 1  #include <stdio.h>
 2
 3  int process_file(char const *path)
 4  {
 5    FILE *f = 0;
 6    int   data;
 7    int   rb;
 8
 9    if (!(f = fopen(path, "r+b"))) {
10      fprintf(stderr, "cannot␣open␣file␣`%s'\n", path);
11      return 1;
12    }
13    while ((rb = fread(&data, 1, sizeof(data), f)) == sizeof(data)) {
14      if (fseek(f, -sizeof(data), SEEK_CUR) < 0) {
15        fprintf(stderr, "seek␣error␣in␣`%s'\n", path);
16        fclose(f);
17        return 1;
18      }
19      data++;
20      if (fwrite(&data, 1, sizeof(data), f) != sizeof(data)) {
21        fprintf(stderr, "write␣error␣to␣`%s'\n", path);
22      }
23    }
```

```
24    if (ferror(f)) {
25      fprintf(stderr, "read␣error␣from␣`%s'\n", path);
26      fclose(f);
27      return 1;
28    }
29    if (rb > 0) {
30      fprintf(stderr, "format␣error␣in␣`%s'\n", path);
31      fclose(f);
32      return 1;
33    }
34    if (fclose(f) < 0) {
35      fprintf(stderr, "write␣error␣to␣`%s'\n", path);
36      return 1;
37    }
38    return 0;
39  }
40
41  int main(int argc, char *argv[])
42  {
43    int retval = 0;
44    int i;
45    for (i = 1; i < argc; i++) {
46      retval |= process_file(argv[i]);
47    }
48    return retval;
49  }
```

1 Занятие №9

1.1 Препроцессор

Препроцессирование — это специальный просмотр исходного файла на языке Си, в ходе которого выполняются специальные директивы (директивы препроцессора) и производится макроподстановка в тексте программы. Результатом работы препроцессора является текстовый файл, который далее попадает на вход основной стадии трансляции.

Каждая директива препроцессора должна быть записана в отдельной строке файла. При необходимости директива препроцессора может быть продолжена на следующую строку, если последним символом строки записать символ «обратной косой черты» \. Отличительным признаком директивы препроцессора является символ #, который должен быть первым непробельным символом в строке.

1.1.1 Директивы `#define`, `#undef`

Директива препроцессора `#define` позволяет задавать новые макроопределения, которые могут быть как с параметрами, так и без параметров. Определение макроса без параметров выглядит следующим образом:

```
#define <name> <text>
```

Здесь `<name>` — это имя макроса, которое должно быть идентификатором в смысле языка Си, то есть начинаться с латинской буквы или подчёркивания, за которой идут ноль или более латинских букв, символов подчёркивания или цифр. Как и в языке Си, при сравнении имён учитывается регистр букв. Определяемое имя не должно быть уже определённым макросом, в противном случае выдаётся ошибка. `<text>` — это произвольный текст, то есть последовательность допустимых лексических единиц языка Си. Если в тексте встречаются комментарии, каждый комментарий заменяется на один символ пробела.

Например,

```
#define M_PI 3.14159265358979323846
```

определяет макрос `M_PI`, а

```
#define while /* do substitution */ do
```

определяет макрос `while`, который раскрывается в `do`. Поскольку макроподстановка происходит до синтаксического анализа программы, такое макроопределение приведёт к тому, что все ключевые слова **`while`** будут заменены на ключевые слова **`do`**.

Определение макроса с параметрами выглядит следующим образом:

```
#define <name>( <params> ) <text>
```

Открывающая скобка не должна быть отделена пробельными символами от имени макроса. Если в списке параметров или в тексте встречаются комментарии, каждый комментарий заменяется на один символ пробела. Параметры макроопределения — это список идентификаторов, разделённых запятыми. Параметры могут использоваться в тексте макроопределения, тогда при макроподстановке на месте имени параметра будет стоять текст соответствующего фактического параметра макроса.

Например,

```
#define swap(a,b) (a ^= b, b ^= a, a ^= b)
```

это возможный вариант макроса, который меняет местами две переменных какого-либо одного целого типа, а макрос

```
#define CHECK(x) if (x < 0) { fprintf(stderr, "x␣<␣0"); exit(1); }
```

может использоваться при контроле допустимых значений в какой-либо функции.

Текст макроопределения может использовать другие макросы. В момент определения макросы никак не раскрываются, их полное раскрытие откладывается на момент использования макроса. Например, следующий фрагмент программы

```
#define A B + 1
#define B 2
  x = A + 2;
```

присвоит переменной x значение 5.

В тексте макроопределения могут использоваться две специальных операции: превращения аргумента в строку и конкатенации. Преобразование аргумента в строку записывается как #<param>, где <param> — это имя параметра макроса. Преобразование можно трактовать как заключение значения параметра в кавычки, а если кавычки присутствуют в тексте значения параметра, они экранируются с помощью символа \. Например, если дано макроопределение

```
#define tostr(a) #a
```

вызов `tostr(a > b)` раскроется в `"a > b"`, а вызов `tostr(puts("hello"))` раскроется в `"puts(\"hello\")"`.

Операция склейки записывается как <arg1>##<arg2>, где аргументы операции — произвольные лексические единицы. Результатом склейки будет одна новая лексическая единица. Например, если дано макроопределение

```
#define glue(a,b) a##b
```

запись `glue(a,2)` будет раскрыта в идентификатор a2, запись `glue(d,o)` будет раскрыта в ключевое слово **do**, а запись `glue(+,+)` будет раскрыта в знак операции инкремента ++.

Транслятор языка Си предопределяет несколько макросов. Макрос `__LINE__` всегда раскрывается в номер строки текста, в которой он используется, макрос `__FILE__` раскрывается в строку, которая содержит имя просматриваемого в данный момент времени препроцессором файла, а макрос `__DATE__` раскрывается в строку, которая содержит время компиляции. Кроме того, каждый транслятор определяет дополнительные макросы, по которым можно узнать версию транслятора, операционную систему, тип процессора и пр. Например, макрос `__GNUC__` раскрывается в номер версии компилятора **gcc**, если для компиляции используется он. Макрос `__linux__` равен 1, если компиляция ведётся в системе **Linux**, и т. д.

Директива препроцессора `#undef <name>` сбрасывает определение макроса с именем <name>. С этого момента макрос становится неопределённым и может использоваться, например, как обыкновенный идентификатор. Если данное имя не было определено как макрос, директива не делает ничего.

1.1.2 Макроподстановки в тексте программы

Если препроцессор находит в тексте программы идентификатор, который является именем ранее определённого макроса, препроцессор выполняет макроподстановку. Препроцессор не выполняет макроподстановку в комментариях, символьных строках и символьных константах.

Если макрос был определён как макрос без параметров, идентификатор заменяется на текст макроопределения, причём справа и слева от подставляемого текста добавляется по

одному символу пробела. Например, если макрос `M_PI` определён, как описано выше, текст `M_PI(10)` будет раскрыт в `3.14159265358979323846 (10)`.

Если макрос был определён как макрос с параметрами, а в тексте программы сразу после идентификатора не следует символ открывающей скобки, идентификатор не изменяется и макроподстановки не происходит. Например, если в тексте программы используется идентификатор `glue` сам по себе, он не изменится в результате препроцессирования.

Если в тексте программы сразу после идентификатора следует открывающая скобка, препроцессор проверяет совпадение количества параметров в макроопределении и макровызове. В случае несовпадения выдаётся сообщение об ошибке. Далее вместо текста макровызова подставляется текст макроопределения, в котором формальные параметры заменены на текст, который находится в соответствующих фактических параметрах.

Препроцессор не выполняет рекурсивной макроподстановки. Если при обработке какого-либо макровызова произвести очередное макрорасширение означало бы войти в рекурсию, препроцессор оставляет макровызов без изменений. Например, пусть даны два макроопределения

```
#define X Y + 1
#define Y X + 1
```

В этом случае текст `X` раскроется в `X + 1 + 1`, а текст `Y` раскроется в `Y + 1 + 1`.

Если в тексте-параметре макровызова используются макровызовы, макроподстановки в тексте параметра выполняются до того, как текст параметра будет подставлен вместо соответствующего формального параметра. Например, если дано макроопределение

```
#define S(a,b)
```

текст `S(a,S(b,c))` раскроется в `a + b + c`.

1.1.3 Использование макроопределений

Поскольку препроцессирование производится до синтаксического анализа программы и может произвольным образом менять лексическую и синтаксическую структуру программы, при использовании макросов нужно учитывать следующие детали.

Лексическая вложенность. Макроопределения не подчиняются правилам лексической вложенности языка Си. То есть, если в тексте программы определяется макрос с именем `name`, ниже по тексту он может только использоваться, либо переопределяться с помощью директивы препроцессора `#define`, либо сбрасываться с помощью директивы препроцессора `#undef`. После директивы `#undef` имя становится доступным для полноценного использования в программе. Например, следующий фрагмент программы даст при компиляции синтаксическую ошибку:

```
#include <stdio.h>
int main(void)
{
        int NULL = 0;
        return 0;
}
```

Имя `NULL` определяется в файле `<stdio.h>` как макрос.

Приоритеты операций. Приоритеты операций в выражениях после макрорасширений могут не совпасть с ожидаемым порядком вычисления операций. Например, пусть макрос `S` определён как

```
#define S(a,b) a + b
```

Предположим, что он используется как `S(1<<2,3)`. Можно было бы ожидать, что результат вычисления такого выражения равен 7, но после макроподстановки получается вы-

ражение `1<<2 + 3`, значение которого — 32. Поэтому в подобных случаях использование параметров в тексте макроопределения *нужно заключать в скобки*.

Даже модифицированное макроопределение

```
#define S(a,b) (a) + (b)
```

не устраняет всех проблем. Рассмотрим выражение `S(1,2)*3`. Можно было бы ожидать, что его значение равно 9, однако после макроподстановки получается выражение `(1)+(2)*3`, значение которого равно 7. Поэтому в подобных случаях и весь текст макроопределения *нужно заключать в скобки*.

Итак, правильное макроопределение должно выглядеть следующим образом

```
#define S(a,b) ((a)+(b))
```

Побочные эффекты. Если один и тот же параметр макроса используется в его теле несколько раз, использование в качестве параметра макроса выражения с побочным эффектом может привести к неожиданным результатам. Например, пусть имеется макроопределение, которое вычисляет максимальное из двух чисел:

```
#define max(a,b) ((a)>=(b)?(a):(b))
```

Предположим, что значение переменной `x` равно 6, а значение переменной `y` равно 7. Тогда значение выражения `max(++x,y)` равно 8! В самом деле, выражение раскроется следующим образом: `((++x)>=(y)?(++x):(y))`, и поскольку `++x` даёт результат 7, и это же значение будет присвоено переменной `x`, условие будет выполнено, поэтому выражение `++x` будет вычислено ещё один раз.

Поскольку избежать повторного вычисления аргументов с побочным эффектом невозможно, макросы, которые используют свои аргументы несколько раз, должны быть задокументированы, чтобы предупредить возможные ошибки при их использовании.

Границы операторов. Предположим, что мы хотим написать макрос, который меняет местами значения двух своих аргументов произвольного типа. Макрос используется как процедура, то есть он не может встретиться в выражении. Такой макрос может выглядеть так:

```
#define swap(type,a,b) {type t = a; a = b; b = t;}
```

Использоваться в программе он может, например, следующим образом: `swap(int,x,y);`. Поскольку в тексте программы `swap` выглядит как вызов функции, естественно ставить после него символ окончания оператора `;`. Но предположим, что вызов этого макроса используется в условном операторе, например

```
if (cond)
  swap(int,x,y);
else
  func();
```

При компиляции этого фрагмента программы будет получено сообщение о синтаксической ошибке. В самом деле, после макроподстановки получим

```
if (cond)
  {int t = x; x = y; y = t;};
else
  func();
```

Составной оператор после **if** не требует символа `;` в конце оператора, поэтому компилятор будет рассматривать `;` после закрывающей фигурной скобки как пустой оператор уже после условного оператора, таким образом ключевое слово **else** оказывается не относящимся ни к какому условному оператору.

Чтобы устранить этот недостаток нужно переписать макрос так, чтобы его тело было одним оператором, но после которого требовалась бы `;`. Сделать это можно, используя оператор цикла **do**—**while**.

```
#define swap(type,a,b) do{type t = a; a = b; b = t;}while(0)
```

Оптимизирующий компилятор может заметить, что условие цикла всегда ложно, поэтому лишнего кода сгенерировано не будет.

Осталась ещё одна неприятность: что будет, если в качестве одного из аргументов макроса будет указана переменная `t`, которая вполне может существовать в точке использования этого макроса? Мы можем использовать какое-нибудь сложное имя, вероятность использования которого в программе невелика, либо можем переложить задачу подбора уникального имени на пользователя макроса, добавив ещё один параметр — имя временной переменной.

1.1.4 Директива `#include`

Директива `#include` вставляет содержимое заданного файла вместо данной директивы. Директива может записываться в одной из трёх форм:

```
#include <file>
#include "file"
#include macro
```

В первом случае файл с именем `file` ищется в стандартных каталогах компилятора, и если он найден, его содержимое подставляется вместо директивы `#include`. Пользователь имеет возможность добавлять свои каталоги к списку стандартных каталогов. Во втором случае файл с именем `file` ищется сначала в текущем каталоге, и только если он не найден там, поиск продолжается в стандартных каталогах. В третьем случае выполняются макроподстановки, и после макроподстановок должна получиться директива `#include` либо в первой, либо во второй форме.

Обычно в программы на языке Си с помощью директивы `#include` включаются так называемые *заголовочные* файлы, которые обычно имеют суффикс `.h`. В заголовочных файлах находятся определения типов данных, макросов, прототипы функций, внешние объявления переменных, то есть информация, необходимая для правильной компиляции программы, состоящей из нескольких исходных файлов.

1.1.5 Директивы условной компиляции

Директивы условной компиляции позволяют включать или исключать части текста программы в зависимости от выполнения условия. Фрагменты, использующие условную компиляцию, имеют следующий вид:

```
#if <expr1>
<text1>
#elif <expr2>
<text2>
...
#elif <exprn>
<textn>
#else
<texte>
#endif
```

Каждый блок условной компиляции начинается с директив `#if`, `#ifdef` или `#ifndef` и заканчивается директивой `#endif`. Блоки условной компиляции могут вкладываться друг

в друга, но поскольку каждый завершается директивой `#endif` неоднозначности не возникает.

В выражении, которое определяет условие условной компиляции, могут использоваться целые литеральные значения и идентификаторы, определённые как макросы. Допускаются все операции языка Си, применимые к целым величинам. Перед вычислением выражения выполняется макрорасширение всех использованных макросов. Если значение вычисленного выражения `<expr1>` не равно 0, то текст `<text1>` сохраняется, а все остальные `<texti>` заменяются на пустые строки. Если значение выражения `<expr1>` равно 0, а в блоке условной компиляции есть директивы `#elif`, вычисляются выражения `<exprj>`, и текст, соответствующий первому из них, которое дало ненулевой результат, сохраняется, а остальные тексты заменяются на пустые строки. Если ни одно из выражений не дало значения «истины», сохраняется текст, который следует за директивой `#else`, если она присутствует.

В препроцессорных выражениях допустима унарная операция `defined <name>`, которое вырабатывает значение «истина», если `<name>` был ранее определён как макрос. Директива условной компиляции `#ifdef <name>` эквивалентна директиве `#if defined <name>`, а директива условной компиляции `#ifndef <name>` эквивалентна директиве `#if !defined <name>`.

Основное назначение директив условной компиляции — задавать фрагменты программы, которые должны или не должны компилироваться в зависимости от значения некоторого макроса препроцессора. Например, программа может компилироваться в двух режимах: отладочном и рабочем. Отладочный режим может обозначаться определением макроса `DEBUG`. Тогда мы можем определить макрос для отладочной печати, который в отладочном режиме будет раскрываться в некоторый оператор, выводящий отладочную печать, а в рабочем режиме — в пустую строку.

```
#if defined DEBUG
#define DPRINT(x) printf x
#else
#define DPRINT(x)
#endif
```

Тогда отладочная печать добавляется в программу следующим образом:

```
x = some_function();
DPRINT(("x = %d\n", x));
```

Обратите внимание, что аргумент макроса `DPRINT` заключён в двойные скобки.

Другое применение условной компиляции — для фрагментов программ, которые выглядят по-разному в разных операционных системах.

```
#if defined __MSDOS__
  <здесь фрагмент для MS-Dos>
#elif defined __linux__
  <а здесь - для Linux>
#endif
```

Наконец, условная компиляция может использоваться для комментирования больших фрагментов кода программы. Как известно, комментарии в языке Си не могут вкладываться друг в друга, поэтому невозможно закомментировать текст функции с помощью `/*` и `*/`, если он уже содержит такие комментарии. Тогда нужно использовать условную компиляцию:

```
#if 0
<some code to comment>
#endif
```

Блоки условной компиляции могут вкладываться друг в друга, поэтому закомментированный таким образом фрагмент кода может потом оказаться частью ещё большего отключённого фрагмента.

1.2 Схема трансляции программы

Рассмотрим схему трансляции программы на языке Си, которая традиционно используется в системах Unix. Трансляция программы состоит из следующих этапов:

1. препроцессирование;
2. трансляция в ассемблер;
3. ассемблирование;
4. компоновка.

Традиционно исходные файлы программы на языке Си имеют суффикс имени файла `.c`, заголовочные файлы для программы на Си имеют суффикс `.h`. В файловых системах типа Unix регистр букв значим, и если, например, имя файла имеет суффикс `.C`, такой файл считается содержащим текст программы на языке Си++, и будет компилироваться компилятором языка Си++, а не Си.

Препроцессирование. Препроцессирование уже было рассмотрено нами ранее. Препроцессор просматривает входной `.c` файл, исполняет в нём директивы препроцессора, включает в него содержимое других файлов, указанных в директивах `#include` и пр.

В результате получается файл, который не содержит директив препроцессора, все используемые макросы раскрыты, вместо директив `#include` подставлено содержимое соответствующих файлов. Файл с результатом препроцессирования обычно имеет суффикс `.i`, однако после завершения трансляции все промежуточные временные файлы удаляются, поэтому такой файл, как правило, никогда не виден пользователю. Результат препроцессирования называется *единицей трансляции*.

Трансляция в ассемблер. Это — основная фаза работы. На вход подаётся одна единица трансляции, а на выходе (при отсутствии синтаксических и семантических ошибок) выдаётся файл на языке ассемблера для (как правило) машины, на которой ведётся трансляция. Файл с оттранслированной программой на языке ассемблера имеет суффикс имени `.s`, но точно так же, как и результат работы препроцессора, он, как правило, не виден пользователю.

Ассемблирование. На этой стадии работает ассемблер. Он получает на входе результат работы предыдущей стадии и генерирует на выходе объектный файл. Объектные файлы традиционно имеют суффикс `.o`. Программа-ассемблер в системах Unix обычно называется as.

Компоновка. Компоновщик получает на входе объектные файлы для каждой единицы трансляции, из которых состоит программа, подключает к ним стандартную библиотеку языка Си и библиотеки, указанные пользователем, и на выходе получает исполняемую программу. В системах Unix исполняемые двоичные программы не имеют никакого специального суффикса, например, оболочка-драйвер для компилятора GNU C называется просто gcc. Компоновщик (редактор связей) в системах Unix обычно называется ld.

1.3 Запуск транслятора gcc

Рассмотрим основные возможности транслятора GNU C. Транслятор запускается командой `gcc <files-and-options>`. В командной строке задаётся список файлов, которые должны быть оттранслированы и объединены в один исполняемый файл. Какие операции необходимо выполнить с файлом — зависит от суффикса имени файла. Возможные суффиксы перечислены в таблице 1. Если имя файла имеет нераспознанный суффикс, это имя передаётся компоновщику.

`.h`	Заголовочный файл на языке Си. Попытка трансляции такого файла вызывает сообщение об ошибке.
`.c`	Файл на языке Си. Выполняется препроцессирование, трансляция, ассемблирование и компоновка.
`.i`	Препроцессированный файл на языке Си. Выполняется трансляция, ассемблирование и компоновка.
`.s`	Файл на языке ассемблера. Выполняется ассемблирование и компоновка.
`.S`	Файл на языке ассемблера. Выполняется препроцессирование, ассемблирование и компоновка.
`.o`	Объектный файл. Выполняется компоновка.
`.a`	Файл статической библиотеки. Выполняется компоновка.

Таблица 1: Суффиксы имён файлов для транслятора gcc

Набор действий определяется для каждого файла индивидуально. Например, если в командной строке указаны имена файлов `1.c` и `2.o`, то для первого файла будут выполнены все шаги трансляции, а для второго — только компоновка. Исполняемый файл будет содержать результат трансляции первого файла, скомпонованный со вторым файлом и стандартными библиотеками.

Пользователь может явно задать, на такой фазе нужно остановиться. По умолчанию транслятор пытается выполнить все необходимые фазы, включая компоновку программы. Конечная фаза трансляции программы определяется для всех транслируемых за один вызов gcc файлов указанием одной из опций, перечисленных в таблице 2.

Например, командная строка

```
gcc 1.c 2.c -o 1
```

транслирует два файла на языке Си, объединяя их в одну программу с именем `1`. Командная строка

```
gcc 3.o 4.o -o 3 -lm
```

компонует два объектных файла, добавляя к ним стандартную библиотеку языка Си и стандартную математическую библиотеку (опция `-lm`), и помещает результат в исполняемый файл с именем `3`.

Прочие полезные опции транслятора gcc перечислены в таблице 3.

1.4 Классы памяти

При определении переменных или функций может быть указан *класс памяти*, который определяет время их существования и область видимости. В языке Си определены 4 ключе-

`-E` Остановиться после препроцессирования. Результат работы препроцессора выводится по умолчанию на стандартный поток вывода. Имя выходного файла можно указать с помощью опции `-o`. При этом если в командной строке указано несколько файлов, то в выходной файл будет помещён результат препроцессирования последнего файла.

`-S` Остановиться после трансляции в ассемблер. По умолчанию имя выходного файла получается из имени входного файла заменой суффикса `.c` или `.i` на суффикс `.o`. Явное имя выходного файла можно указать с помощью опции `-o`. Попытка использования опции `-o` и нескольких имён входных файлов вызывает сообщение об ошибке.

`-c` Остановиться после ассемблирования. По умолчанию имя выходного файла получается из имени входного файла заменой суффикса его имени на суффикс `.o`. Явное имя выходного файла можно указать с помощью опции `-o`, которая несовместима с указанием одновременно нескольких транслируемых файлов.
Если ни одной из перечисленных выше опций не задано, выполняются все стадии трансляции. Имя выходного файла по умолчанию равно `a.out`, но может быть изменено с помощью опции `-o`.

`-o` Позволяет задать явное имя выходного файла для любой стадии трансляции.

Таблица 2: Опции конечной фазы транслятора **gcc**

вых слова, задающих классы памяти: **extern**, **static**, **auto**, **register**. Их интерпретация немного различается в случае переменных и функций и в случае глобального и локального определения.

Ключевое слово **auto** может использоваться только при определении переменных внутри блока. Класс памяти **auto** определяет, что переменная должна быть создана при входе в блок и уничтожена при выходе из блока. Поскольку локальные переменные по умолчанию создаются и уничтожаются именно таким образом, необходимости в использовании ключевого слова **auto** никогда не возникает.

Ключевое слово **register** может использоваться при определении формальных параметров функций и переменных внутри блока. Это — указание транслятору, что он должен попытаться разместить переменную на регистре процессора. Транслятор не обязан следовать этим указаниям, но в любом случае для переменной с классом памяти **register** не определена операция взятия адреса.

Ключевое слово **static** может использоваться для определения переменных и функций на глобальном уровне и на уровне блока. Глобальная переменная данного класса памяти существует всё время работы программы и видна от точки определения и до конца единицы трансляции. Однако такая переменная не может быть сделана видимой из других единиц трансляции. Блочная переменная с классом памяти **static** существует всё время работы программы, но видна только в пределах блока, в котором она определена. Она также не может быть сделана видимой из других единиц трансляции. Например, следующая функция при каждом вызове будет возвращать последовательно числа натурального ряда.

```
int next_nat(void)
{
        static int nat = 1;
        return nat++;
}
```

Если убрать из объявления переменной `nat` ключевое слово **static**, функция всегда

`-I PATH`	Добавляет каталог `PATH` в начало списка каталогов, которые просматриваются препроцессором при поиске файлов, подключаемых директивой `#include`. В командной строке может быть указано несколько опций `-I`, тогда каталоги просматриваются в порядке, в котором они указаны в командной строке.
`-D NAME`	Определяет макрос с именем `NAME`, который получает значение 1.
`-D NAME=VALUE`	Определяет макрос с именем `NAME`, который получает заданное значение.
`-Wall`	Включает выдачу большого количества предупреждающих сообщений, которые по умолчанию не выдаются. Опция должна использоваться при компиляции программ, все предупреждающие сообщения компилятора должны быть внимательно проанализированы, поскольку сообщения могут указывать на ошибки в программе.
`-g`	Включает генерацию отладочной информации в исполняемую программу. Наличие отладочной информации позволяет отлаживать программу в терминах исходного языка, а не машинного кода.
`-O2`	Включает большинство оптимизаций программы, которые одновременно уменьшают размер программы и увеличивают скорость её выполнения.
`-L PATH`	Добавляет путь `PATH` в начало списка каталогов, которые просматриваются редактором связей при поиске библиотек, указанных с помощью опции `-l`. Если в командной строке указано несколько опций `-L`, они добавляются в том же порядке, в котором указаны в командной строке.
`-lname`	Добавляет библиотеку `name` к списку библиотек, которые участвуют в компоновке программы (обратите внимание на отсутствие пробела между опцией и именем библиотеки). В системах Unix редактор связей просматривает библиотеки *один раз*, поэтому неправильный порядок задания библиотек может привести к тому, что некоторые имена останутся неопределёнными, и компиляция завершится с ошибкой. Файл, хранящий библиотеку с именем `name`, называется `libname.a`, если библиотека статическая, и `libname.so`, если библиотека динамическая.
`-static`	Указывает, что при компоновке не должны использоваться динамические библиотеки. Реализации всех используемых в программе функций будут добавлены непосредственно в исполняемый файл. Это может привести к тому, что размер тривиальной программы вырастет до сотни килобайт, зато такая программа перестанет быть зависимой от динамических библиотек, и на некоторых системах только статически скомпонованные программы могут отлаживаться.

Таблица 3: Прочие опции транслятора gcc

будет возвращать значение 1.

Прототип функции, объявленный с классом памяти **`static`** на глобальном уровне, ви-

ден от точки объявления и до конца единицы компиляции. Такая функция не может быть сделана видимой из других единиц компиляции. Прототип функции, объявленный с классом памяти **static** на локальном уровне виден только в пределах блока. Если прототип функции объявляется с классом памяти **static**, то и сама функция должна быть определена с тем же классом памяти. Если функция, прототип которой имеет класс памяти **static** используется в единице компиляции, но не определяется в ней, возникнет ошибка компиляции, даже если функция с таким же именем определена в другой единице трансляции. Таким образом, класс памяти **static** используется, чтобы сделать имя невидимым из других единиц трансляции.

Все переменные и функции, объявленные с классом памяти **extern**, видимы от точки объявления и до конца единицы трансляции даже для блочных определений. Объявление переменной с классом памяти **extern** означает, что память под эту переменную выделена где-то в другом месте, в этой же, а возможно и другой единице трансляции. Транслятор в этом случае не выделяет память под переменную, а будет использовать имя как внешнее. Компоновщик связывает все использования внешнего имени с определением имени в некоторой единице трансляции. В одной единице трансляции может быть несколько объявлений одной и той же внешней переменной при условии, что всегда указывается один и то же тип переменной. В том же файле может находиться и само определение глобальной переменной, которая должна иметь такой же тип и не должна быть объявлена с классом памяти **static**. Например, переменная `stdin` может определяться в заголовочном файле `<stdio.h>` следующим образом:

```
extern FILE *stdin;
```

Глобальные переменные по умолчанию имеют класс памяти **common**, который мы рассмотрим в следующем разделе.

Все прототипы функций имеют по умолчанию класс памяти **extern**.

1.5 Компоновка программы

Если исполняемая программа компонуется из нескольких единиц трансляции, компоновщик использует свои правила видимости имён, которые приведены ниже.

- Все имена, объявленные с классом памяти **static**, видимы только в пределах своей единицы трансляции и не влияют на компоновку.
- Если некоторая единица трансляции использует внешнее имя (переменной или функции), которое не определено ни в какой единице трансляции, выдаётся сообщение об ошибке.
- Если несколько единиц трансляции определяют нестатическую функцию с одним и тем же именем, выдаётся сообщение об ошибке.
- Если некоторое нестатическое имя определяется и как переменная, и как функция, выдаётся сообщение об ошибке.
- Если несколько единиц трансляции определяют нестатическую инициализированную переменную с одним и тем же именем, выдаётся сообщение об ошибке.
- Если несколько единиц трансляции определяют переменную с одним и тем же именем, которая инициализируется не более чем в одной единице трансляции, все определения размещаются, начиная с одного адреса (класс памяти `common`).

Последнее правило можно продемонстрировать на следующем примере. Предположим, что в трёх файлах определена переменная `var` следующим образом:

```
int var = 1;
int add1(void)
{
  return var++;
}
```

```
int var;
int add2(void)
{
  return var += 2;
}
```

```
int var;
int add3(void)
{
  return var += 3;
}
```

Если все три единицы компиляции объединяются в одну программу, то переменная `var` каждого из трёх файлов будет располагаться по одному и тому же адресу, и каждая их трёх функций будет работать, по сути, с общей переменной. Чтобы предотвратить такое слияние переменных можно использовать явную инициализацию переменной `var` нулём, тогда компоновщик выдаст сообщение об ошибке.

1.6 Программы из нескольких единиц трансляции

Только самые простые программы размещаются полностью в одном исходном файле. Более сложные программы состоят из нескольких исходных файлов, которые объединяются компоновщиком. При написании таких программ полезно следовать следующим рекомендациям.

- При группировке функций и переменных по исходным файлам логически сильно связанные функции объединяются в один исходный файл. Например, функции работы с файлом таблицы могут быть помещены в один исходный файл, функции, которые выводят на экран содержимое таблицы, — в другой файл, в функции, которые анализируют ввод пользователя, — в третий файл.

- Чем больше переменных объявлено в единице компиляции с классом памяти **`static`** вместо класса памяти по умолчанию `common`, тем лучше. Лучше всего, если доступ к данным всегда происходит с помощью вызовов функций. Чем меньше «чужих» переменных использует некоторая единица компиляции, тем она проще для понимания.

- Для каждого `.c` файла должен существовать интерфейсный файл с тем же именем, но суффиксом `.h`, в котором определяются переменные, функции, типы данных и пр., которые могут использоваться извне данной единице компиляции.

- Исходный `.c` файл должен обязательно подключать свой собственный `.h`-файл. В этом случае транслятор обнаружит рассогласования между объявлениями в `.h`-файлах и определениями в `.c`-файле.

- Интерфейсный `.h` файл должен быть обязательно защищён от повторного включения следующей конструкцией:

  ```
  #ifndef __NAME_H__
  #define __NAME_H__
  <здесь находится текст файла>
  #endif
  ```

  Здесь `NAME` — это имя файла (без суффикса). Поскольку некоторые `.h`-файлы могут включать другие `.h`-файлы, когда программа становится большой, человек уже не может отследить, какие файлы уже включались, а какие — ещё нет. Поэтому в `.c` файле

включаются все заголовочные файлы, необходимые данной единице компиляции. Защита от повторного включения предотвращает появление ошибок о переопределённых типах, переменных и функциях.

- В заголовочном файле помещаются макроопределения и типы данных, являющиеся интерфейсом данной единицы компиляции, то есть необходимые для использования функций и переменных этой единицы компиляции. С классом памяти **`extern`** помещаются необходимые переменные и прототипы функций, объявленные в соответствующей единице компиляции.

- В заголовочный файл никогда не помещаются тела функций и определения переменных с классом памяти, отличным от класса **`extern`**. В заголовочный файл никогда не помещаются прототипы функций с классом памяти **`static`**. Если некоторый тип или константа используются только в теле какой-либо функций и не нужен для правильной работы с функциями и переменными данной единицы компиляции, этот тип или константа также не помещаются в заголовочный файл.

1 Интерфейс библиотечных функций и системных вызовов

Язык Си не содержит никакой поддержки операций ввода/вывода, параллелизма и пр. Такие (и многие другие) функции реализованы в стандартной библиотеке языка. Прототипы функций, типы данных и константы определяются в стандартных заголовочных файлах (например, <stdio.h>), а собственно тела (реализации) стандартных функций определяются в библиотечном файле, который подключается к программе на этапе компоновки. В современных системах библиотеки подключаются к программе не на этапе компоновки, а позднее, на этапе запуска программы, что позволяет уменьшить размер исполняемого модуля программы на диске, и, что более важно, суммарный размер памяти, занимаемый программами при выполнении за, счёт того, что страницы кода стандартной библиотеки разделяются между всеми процессами.

Часть стандартных функций может быть реализована непосредственно в самом процессе без обращения к ядру операционной системы, например, функции работы со строками. Некоторые функции выполняют существенный объём работы в пользовательском режиме, но для выполнения какой-то части работы обращаются к системе. Задача некоторых функций состоит в том, чтобы подготовить параметры так, как этого требует интерфейс с ядром операционной системы, и передать управление ядру. Такие функции называются *системными вызовами*, хотя, строго говоря, это только функции-оболочки системных вызовов. Граница между библиотечными функциями и системными вызовами проходит по разному на разных системах. Например, на системах BSD операции работы с сокетами реализованы как системные вызовы, а на некоторых системах System V они же реализованы как библиотечные функции.

Не всегда библиотечная функция или системный вызов завершаются успешно. Например, операция открытия файла может закончится неудачно по разным причинам, например файл не найден, или недостаточно прав доступа. В этом случае функция возвращает специальное значение, сигнализирующее об ошибке, а в глобальную переменную `errno` записывается код причины ошибки, например, `EACCES` — недостаточно прав доступа. Чтобы получить доступ к переменной `errno` и константам кодов ошибок программа должна подключить заголовочный файл <errno.h>.

Чаще всего программе требуется напечатать пользователю некоторое осмысленное сообщение об ошибке. Например, в случае ошибки `EACCES` можно напечатать сообщение `"Permission denied"`. Для этого стандартная библиотека содержит специальные функции.

```
#include <string.h>
char *strerror(int errnum);
```

Функция `strerror` преобразовывает код ошибки, переданный в параметре `errnum` в текстовую строку-описание ошибки, указатель на которую возвращается. Эта текстовая строка находится в статической памяти библиотеки и ни в коем случае не должна модифицироваться.

```
#include <stdio.h>
void perror(char const *str);
```

Функция `perror` печатает сообщение о последней ошибке (текущее значение переменной `errno`) на стандартный поток ошибок. Если параметр `str` не равен `NULL`, перед описанием ошибки будет напечатана строка `str` и символ двоеточия (`':'`).

2 Работа с файлами

Понятие «файл» — одно из основных понятий операционных систем Unix. Понятие «файл» здесь шире, чем в других операционных системах, а именно, файлом может представляться любая сущность, к которой применимы понятия «открытия», «чтения», «записи». Например, последовательность байт, хранящаяся в некоторых секторах диска и имеющая имя, — это обычный (так называемый «регулярный») файл. Но и сам диск тоже является с точки зрения Unix файлом (специальное блочное устройство). Последовательный порт компьютера тоже является файлом (специальное символьное устройство).

Для хранения файлов используется файловая система, причём почти все файлы (кроме анонимных каналов и интернет-сокетов) имеют некоторое соответствие в файловой системе. Для регулярных файлов файловая система хранит имена файлов, служебную информацию о файле и сами блоки данных, а для прочих типов файлов — только имена и служебную информацию. Файловая система имеет иерархическую структуру: корень файловой системы может содержать файлы и каталоги, каждый каталог в свою очередь может содержать другие файлы и каталоги и так далее. Две (и более) файловые системы могут быть объединены в одну файловую систему с помощью монтирования. При монтировании файловой системы корневой каталог одной файловой системы подключается вместо некоторого каталога другой файловой системы, так что в результате получается одна файловая система. Все файловые системы, необходимые для работы операционной системы монтируются автоматически при начальной загрузке системы. Обычный пользователь чаще всего не имеет права самостоятельно монтировать файловые системы (из-за соображений безопасности).

В «родных» для Unix файловых системах (например, оригинальной файловой системе SysV, файловой системе BSD, файловой системе Linux) информация о файле разбита на две части: запись в каталоге и индексный дескриптор. Индексный дескриптор (inode) хранит всю служебную информацию о файле: идентификаторы владельца и группы, права доступа, размер, блоки диска, занимаемые файлом и пр. Все индексные дескрипторы в файловой системе занумерованы, а максимальное количество индексных дескрипторов в данной файловой системе задаётся при её создании и не может быть потом изменено. Нумерация индексных дескрипторов уникальна для каждой файловой системы без учёта монтирования, то есть в работающей системе может существовать несколько файлов с одним и тем же номером индексного дескриптора.

Индексный дескриптор не хранит имя файла, которое хранится в каталогах. Кроме имени файла запись в каталоге ещё содержит номер соответствующего индексного дескриптора. Таким образом, по имени файла сначала получается номер его индексного дескриптора, а затем по информации из индексного дескриптора можно считывать данные из файла. Два (или более) имени могут ссылаться на один и тот же индексный дескриптор. В этом случае говорят, что два имени являются «жёсткими» ссылками на один и тот же файл. Жёсткие ссылки не отличимы друг от друга. Для нормальной работы в этой ситуации индексный дескриптор содержит ещё счётчик ссылок на себя. Файл считается уничтоженным, и блоки данных помечаются доступными для повторного использования, только тогда, когда счётчик ссылок на индексный дескриптор становится равным нулю. Счётчик ссылок также увеличивается на 1 при каждом новом открытии файла, а уменьшается на 1 при закрытии файла. Таким образом, может существовать файл, который вообще не имеет имени. Для этого процесс должен создать файл, открыть его, а затем сразу удалить его. Запись о файле будет из каталога удалена, но сам файл продолжит существовать до тех пор, пока счётчик ссылок на его индексный дескриптор не станет равным нулю, то есть до тех пор, пока соответствующий файловый дескриптор не закроет последний процесс, который его, возможно, унаследовал от процесса,

создавшего такой файл.

Во внутренних структурах ядра полный путь к файлу никогда не используется. Он необходим только на этапе открытия файла. После этого достаточно пары ⟨ номер устройства, номер индексного дескриптора ⟩. По открытому файлу не существует простого способа узнать, какое имя было у открытого файла. Чтобы всё-таки узнать имя файла (точнее, одно из имён), нужно получить номер устройства и номер индексного дескриптора (вызовом `fstat`), а затем найти в файловой системе запись с теми же самыми характеристиками.

Ядро может накладывать ограничения на максимальную длину пути. Если в системный вызов передаётся абсолютный или относительный путь, длина строки которого больше этого ограничения, системный вызов завершится с ошибкой `ENAMETOOLONG`. Эта константа называется `PATH_MAX` и определяется в заголовочном файле `<limits.h>`.

Современные Unix-подобные системы поддерживают работу с самыми разнообразными файловыми системами, например VFAT, NTFS, NFS. Чтобы предоставить одинаковый интерфейс доступа к файлам, ядру приходится эмулировать индексные дескрипторы. Не всегда это можно сделать полностью корректно (особенно много исключений существует для сетевой файловой системы NFS), и в этих случаях возможны тонкие отличия в работе с файловой системой.

Каждый тип файловой системы имеет свою структуру каталога. Поэтому некоторые операционные системы (например, Linux) вообще запрещают прямое чтение каталога с помощью системного вызова `read`, а требуют пользоваться специальным системным вызовом (`getdents`).

С точки зрения операционной системы каждый файл является потоком байтов. Никакого дополнительного структурирования не производится. Некоторые файлы могут допускать позиционирование на любое место в файле, другие файлы допускают только последовательное чтение из файла или запись в файл.

Процесс может работать с файлом посредством файлового дескриптора. Файловый дескриптор — это небольшое неотрицательное целое число, которое уникально в пределах данного процесса. Три первых файловых дескриптора имеют специальное назначение. Файловый дескриптор с номером 0 — это стандартный ввод процесса, 1 — стандартный вывод процесса, 2 — стандартный поток ошибок. Многие программы считывают данные со стандартного ввода, выводят результат на стандартный вывод, а сообщения об ошибках печатают на стандартный поток ошибок. Стандартный ввод обычно связан с входным буфером терминала, а стандартный вывод и стандартный дескриптор ошибок — с выходным буфером терминала. Это стандартное соглашение о назначении файловых дескрипторов, тем не менее, может произвольным образом нарушаться. Например, все стандартные дескрипторы могут быть вообще закрыты.

Файловые дескрипторы всегда наследуются при создании нового процесса (системный вызов `fork`) и, как правило, наследуются при запуске новой программы с помощью системного вызова `execve`. Программист может запретить наследование файлового дескриптора при вызове `execve`, пометив его флагом `FD_CLOEXEC`. Для этого используется системный вызов `fcntl`. Под наследованием в данном случае понимается то, что новому процессу (или вновь запущенной программе) будет доступен файловый дескриптор с тем же самым номером, ассоциированный с тем же самым файлом и разделяющий позицию чтения и записи в файле.

2.1 Открытие файла

Для открытия нового файла используется системный вызов `open`. Обратите внимание, что он может использоваться как с двумя, так и с тремя аргументами. Третий аргумент обязательно должен быть указан, если во флагах присутствует бит `O_CREAT`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Первый параметр задаёт путь к файлу, который нужно открыть. Второй параметр определяет режим, в котором будет открыт файл, а третий параметр задаёт права доступа в случае, когда файл создаётся. Файл может быть открыт в одном из трёх режимов: только чтение (`O_RDONLY`), только запись (`O_WRONLY`) и чтение и запись (`O_RDWR`). Попытка выполнить операцию, которая не предусмотрена режимом открытия файла, вызовет ошибку соответствующего системного вызова. Кроме режима работы с файлом в параметре `flags` можно указать флаг создания файла `O_CREAT`, который вызовет создание нового пустого файла, если файл с таким именем ещё не существует. Флаг `O_EXCL`, который может использоваться только вместе с `O_CREAT`, задаёт, что системный вызов должен завершиться с ошибкой с кодом `EEXIST`, если такой файл уже существует. Этот флаг может применяться для простейшей синхронизации процессов, как будет описано в дальнейшем. Флаг `O_TRUNC` задаёт, что старой содержимое файла (если оно было) должно быть уничтожено. Флаг `O_APPEND` задаёт, что перед каждой записью в файл указатель должен позиционироваться на конец файла.

Флаг `O_NONBLOCK` (другое название — `O_NDELAY`) указывает, что операции с файловым дескриптором никогда не должны приводить к блокировке выполнения процесса. Например, если файловый дескриптор ассоциирован с терминалом, и пользователь не ввёл ни одного символа, то системный вызов `read` при использовании неблокирующего режима чтения завершится с ошибкой `EAGAIN`.

Когда файл открывается на сетевой файловой системе NFS, не все флаги открытия могут корректно работать. Например, использование флага `O_EXCL` может не дать ожидаемого эффекта, то есть файловый дескриптор может в некоторых ситуациях быть получен, даже если такой файл уже создан. Это связано с особенностями протокола NFS[1]. Точно также на файловой системе NFS невозможно корректно реализовать флаг добавления `O_APPEND`, поэтому, если одновременно несколько процессов записывают в один файл, это может привести к искажению содержимого файла. Конечно же, эти проблемы не проявляют себя в обычной практике, когда файл открывает и с ним работает единственный процесс, находящийся на единственном компьютере.

При создании нового файла, то есть когда указывается флаг `O_CREAT`, необходимо задать права доступа к создаваемому файлу — параметр `mode`. В этом параметре используются только 12 младших бит, определяющие права доступа для владельца, группы, прочих пользователей и специальные флаги (например, флаг suid). Права с которыми в действительности будет создан файл ещё зависят от маски создания файлов процесса `umask`. Если `perms` — это права доступа к создаваемому файлу, это значение может быть вычислено по формуле `perms = mode & ~umask`. Другими словами, у параметра `mode` сбрасываются те биты, которые установлены в маске создания файлов процесса.

[1] Протокол NFS специально проектировался как протокол без состояний, чтобы авария сервера не сказывалась на работе клиентов. Как показала практика, это оказалось ошибочным решением.

Системный вызов open возвращает номер файлового дескриптора открытого файла, причём всегда выбирается наименьший незанятый номер файлового дескриптора. Если открытие файла невозможно, системный вызов возвращает число -1, а переменная errno будет содержать код ошибки. Возможные коды ошибок для open можно посмотреть в приложении.

Когда файл открывается на запись, наиболее часто используется комбинация режимов открытия O_WRONLY | O_CREAT | O_TRUNC, означающая, что файл открывается в режиме «только запись», если файл не существовал, он будет создан, а если он существовал, его старое содержимое будет уничтожено. Чтобы в таких случаях не задавать эти режимы открытия, предусмотрена функция creat, которая в точности эквивалентна системному вызову open с указанными выше режимами.

2.2 Закрытие файла

Когда процесс завершает работу (в результате вызова _exit, exit или при получении сигнала) все открытые файловые дескрипторы закрываются автоматически. Тем не менее, файловые дескрипторы, которые больше не используются, необходимо закрывать. Каждый процесс имеет ограничение на количество одновременно открытых файловых дескрипторов (это ограничение можно узнать с помощью команд ulimit -a или ulimit -n). Если процесс исчерпал ресурс свободных файловых дескрипторов, открытие файла завершится с ошибкой EMFILE. Кроме того, незакрытые файловые дескрипторы могут привести к тому, что созданные новые процессы будут работать неправильно, например, никогда не завершатся.

Для закрытия открытого файлового дескриптора используется системный вызов close.

```
#include <unistd.h>
int close(int fd);
```

Единственным параметром передаётся номер закрываемого файлового дескриптора. Функция возвращает 0 при нормальном завершении и -1 при ошибке. Если файл был открыт на запись, система может использовать внутреннюю буферизацию для увеличения эффективности работы с внешними устройствами. В этом случае при вызове close система попытается сохранить буферизованные данные на внешнем устройстве, и, если при этом возникнет ошибка ввода/вывода, она будет сообщена пользователю. Поэтому игнорирование кода возврата функции close может привести к незамеченной потере данных, особенно когда используется сетевая файловая система NFS в сочетании с квотированием диска.

Системным вызовом close может закрываться файловый дескриптор, полученный из любого источника: системных вызовов open, socket, dup.

2.3 Работа с файловым дескриптором

Рассмотрим системные вызовы, позволяющие работать с файлом, ассоциированным с файловым дескриптором: считывать и записывать данные, позиционировать указатель текущей позиции.

2.3.1 Чтение

Для чтения данных по файловому дескриптору используется системный вызов read.

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Здесь тип size_t — это тип, который используется в библиотеке языка Си для размера объектов. На 16- и 32-битных машинах этот тип, как правило, эквивалентен unsigned long int. Тип ssize_t — «размер со знаком». На 16- и 32-битных машинах это, как правило, long int.

Параметр fd задаёт номер файлового дескриптора, из которого должно производится чтение. Файловый дескриптор должен быть открыт и должен допускать чтение, в противном случае read завершится с ошибкой. Параметр buf задаёт начальный адрес памяти, по которому должны быть размещены считанные данные, а параметр count — максимальный размер считываемых данных (в байтах). Если блок памяти, заданный параметрами buf и count, полностью или частично находится вне адресов памяти, доступных процессу для записи, read завершится с ошибкой EFAULT [2]. Системный вызов read считывает не более чем count байт, и это число возвращается в качестве результата. Если в файловом дескрипторе выполняется условие «конец файла», системный вызов read возвращает 0. При ошибке возвращается -1, а переменная errno устанавливается в код ошибки. Когда данные, немедленно доступные для чтения, отсутствуют, процесс, как правило, погружается системой в сон до тех пор, пока данные не появятся.

Детали работы с файловым дескриптором: когда считывается count, а когда меньше чем count байт, когда наступает «конец файла» — зависят от того, с чем ассоциирован файловый дескриптор. Если файловый дескриптор ассоциирован с регулярным файлом, то есть с файлом, блоки данных которого действительно размещаются на диске, системный вызов read ведёт себя следующим образом: считывается count байт, если размер непрочитанной части файла больше count, считывается столько байт, каков размер непрочитанной части байт, если она меньше count, наконец, возвращается 0, когда непрочитанных данных не осталось.

2.3.2 Запись

Системный вызов write позволяет записать данные в файл.

```
#include <unistd.h>
ssize_t write(int fd, void *buf, size_t count);
```

Параметр fd задаёт номер файлового дескриптора. Файловый дескриптор должен быть открыт и должен допускать запись, в противном случае write завершится ошибочно. Параметр buf задаёт начальный адрес памяти, по которому размещаются записываемые данные, а параметр count — размер записываемых данных (в байтах). Если блок памяти, заданный параметрами buf и count, полностью или частично находится вне адресов памяти, доступных процессу для чтения, write завершится с ошибкой EFAULT.

write пытается записать в файл заданное количество байт и возвращает, сколько байт в действительности было записано. При ошибке возвращается -1, а переменная errno устанавливается в код ошибки (возможные коды ошибок приведены в приложении). Может быть записано меньше байт, чем запрошено, и это не является ошибкой. Такая ситуация может возникнуть, например, при работе с сетевой файловой системой NFS. В этом случае нужно ещё раз вызвать write с изменёнными указателем на начало области записываемых данных и размером области.

[2] Обратите внимание, что запись по таким адресам самим процессом приведёт к краху программы по сигналу SIGSEGV или SIGBUS.

2.3.3 Операции с файлами в многопроцессной системе

Когда в системе работает более одного процесса, несколько процессов могут независимо открыть один и тот же файл, причём в разных режимах. Например, два процесса могут открыть один и тот же файл на запись. В отличие от других систем, одновременная работа с файлом нескольких процессов по умолчанию не запрещена. Для регулярных файлов система поддерживает атомарность операций чтения и записи. Под атомарностью некоторой операции здесь понимается то, что никакой процесс в системе не сможет наблюдать ситуацию, когда операция уже началась, но ещё не прошла до конца. Если в результате вызова операции чтения или записи процесс был помещён в состояние ожидания поступления данных или освобождения буфера, операция не считается начавшейся.

Обычно права доступа к файлу проверяются один раз: при открытии файла. Если пользователь не имеет достаточно прав, чтобы открыть файл в запрошенном режиме, системный вызов `open` вернёт соответствующий код ошибки. Изменение прав доступа к файлу никак не повлияет на файловые дескрипторы, которые уже ассоциированы с этим файлом. Например, если файл открывался в режиме записи, а после разрешение записи для данного пользователя было отменено, все операции записи в файл, производимые с использованием этого файлового дескриптора, будут завершаться успешно.

Работа с файлами на сетевой файловой системе NFS и здесь имеет свои особенности. Поскольку NFS-сервер не поддерживает список открытых файлов для клиента, а клиент каждый раз присылает запросы на чтение или запись с указанием имени файла (точнее, номера устройства и номера индексного дескриптора), смещения от начала файла и размера файла, права доступа к файлу проверяются сервером при каждой операции. Поэтому, если после открытия файла на клиенте, права доступа были изменены, последующие операции с файловым дескриптором будут завершаться ошибочно, если новые права доступа к файлу не позволяют выполнить запрашиваемую операцию.

Другая особенность NFS связана с удалением файлов. В UNIX файл не считается удалённым, даже если удалены все записи в каталогах, ссылающиеся на его индексный дескриптор, до тех пор, пока не будет закрыт последний файловый дескриптор, ассоциированный с этим файлом. Так как NFS-сервер не имеет информации об открытых на клиенте файлах, он не может удалять файлы, с которыми, возможно, в данный момент идёт работа у клиента. Поэтому при запросе на удаление файл, в отношении которого сервер имеет подозрение, что его может использовать какой-либо клиент, не удаляется, а переименовывается в имя вида `.nfs032847`. Такие файлы удаляются либо когда сервер будет точно знать, что никакой клиент его больше не использует, либо при перезагрузке системы.

2.3.4 Позиционирование

Когда файловый дескриптор ассоциирован с регулярным файлом, осмыслено понятие «текущее положение в файле». Это смещение от начала файла, начиная с которого очередной вызов `read` будет считывать данные, а вызов `write` записывать данные (если только файл не был открыт с режимом `O_APPEND`). После операции чтения или записи указатель текущего положения в файле продвигается вперёд на количество байт, прочитанное из файла или записанное в файл.

Все клоны некоторого файлового дескриптора (то есть полученные из него явно с помощью системных вызовов `dup`, `dup2` или неявно при создании нового процесса с помощью `fork`) разделяют один указатель на текущее положение в файле. Любая операция, выполненная на одном из таких файловых дескрипторов, которая изменяет текущее положение в

файле, делает это для всех клонов. Поэтому, когда некоторый файловый дескриптор, ассоциированный с регулярным файлом, используется несколькими процессами, необходимо следить за тем, что указатель текущего положения в файле установлен, как требуется.

Указатель текущего положения в файле можно произвольным образом изменять с помощью системного вызова `lseek`.

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

Параметр `fildes` задаёт номер файлового дескриптора, параметр `whence` задаёт, от какой точки должно отсчитываться смещение, а параметр `offset` задаёт само смещение. Параметр `whence` может принимать одно из следующих значений:

`SEEK_SET`	0	`offset` отсчитывается от начала файла.
`SEEK_CUR`	1	`offset` отсчитывается от текущего положения.
`SEEK_END`	2	`offset` отсчитывается от текущего размера файла.

При успешном завершении `lseek` возвращает новое положение указателя относительно начала файла, измеренное в байтах. При ошибке возвращается значение `(off_t) -1`, и переменная `errno` устанавливается в код ошибки.

Чтобы узнать текущее положение в файле, нужно вызвать системный вызов `lseek` с параметрами `lseek(fd, 0, SEEK_CUR);`.

2.4 Файловые дескрипторы и дескрипторы потоков

Стандартная библиотека языка Си определяет большое количество функций для работы с файлами, например, `fopen`, `fprintf`, `fscanf`, `fclose`. Эти функции предоставляют широкие возможности по управлению форматом записываемых или считываемых данных: построчный ввод или вывод, преобразование стандартных типов в строковое представление и обратно и т. д. Поскольку они позволяют вводить и выводить структурированные данные, то данные на более высоком уровне абстракции, чем последовательность байтов, эти функции называются ещё «высокоуровневыми» функциями ввода/вывода, в противоположность «низкоуровневым» функциям, описанным выше.

Существенная часть работы высокоуровневых функций ввода вывода (например, форматирование чисел в соответствии со спецификацией формата) производится в стандартной библиотеке языка Си в функциях, которые работают в контексте процесса. Для выполнения собственно операции чтения из файла или записи в файл эти функции должны обратиться к ядру операционной системы. Поскольку ядро может выполнять только системные вызовы низкого уровня, высокоуровневый ввод/вывод использует (реализован с помощью) низкоуровневые операции работы с файлами. Так, функция `fopen` после анализа режимов открытия и создания внутренних структур данных (указатель на которые она возвращает) сделает системный вызов `open`. Функции записи в дескриптор потока вызовут в итоге системный вызов `write` и т. д.

Высокоуровневый ввод/вывод по умолчанию буферизуется. Это значит, что в адресном пространстве процесса выделяется память под некоторый буфер фиксированного размера (по умолчанию обычно 512 байт), записываемые данные попадают сначала в буфер, и лишь потом передаются ядру, считываемые данные сначала считываются в буфер и только затем передаются в программу. Дескрипторы потока, связанные с устройствами типа терминалов, буферизуются построчно, то есть выводимые данные не будут переданы ядру до тех пор, пока программа не выведет символ перевода строки `'\n'`. Исключение составляет стандартные поток ошибок `stderr`, который никогда не буферизуется. Буфер потока является полной

собственностью процесса, и ядро ничего не знает о такой буферизации. Поэтому, когда ядро принудительно завершает процесс (например, при получении процессом необрабатываемого сигнала, который по умолчанию завершает работу процесса), всё содержимое буферов потоков, предназначенных на запись, теряется. Когда ядро создаёт копию адресного пространства процесса (при вызове fork), копируется и содержимое буферов, и состояние дескрипторов потока.

Чтобы изменить режим буферизации данного дескриптора потока используется функция setvbuf.

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Параметр stream задаёт дескриптор потока, параметр mode задаёт новый режим буферизации. Поддерживаются три режима:

- Режим без буферизации (параметр mode должен быть равен константе _IONBF). В этом случае значения параметров buf и size игнорируются.
- Режим с построчной буферизацией (параметр mode равен _IOLBF). Параметр buf должен указывать на начало буфера (области памяти), который имеет размер по крайней мере size байт. Если параметр buf равен NULL, буфер заданного размера будет выделен с помощью malloc при следующей операции ввода/вывода.
- Режим с полной буферизацией (параметр mode равен _IOFBF). Параметр buf должен указывать на начало буфера (области памяти), который имеет размер по крайней мере size байт. Если параметр buf равен NULL, буфер заданного размера будет выделен с помощью malloc при следующей операции ввода/вывода.

Изменение режима будет иметь эффект, только если поток не активен, то есть его буфер пуст, что бывает либо после открытия, но до выполнения операций ввода/вывода, либо после вызова функции fflush.

Если адрес буфера задаётся в функции setvbuf явно, буфер должен существовать, когда поток будет закрываться. Если поток не закрывается явно в самой программе, буфер должен существовать и тогда, когда закончит своё выполнение функция main, то есть буфер не должен быть локальной переменной функции main.

Использование режимов без буферизации и режима полной буферизации может давать нежелательный результат, когда несколько процессов одновременно модифицируют (например, дописывают) один и тот же текстовый файл. Если поток переведён в режим без буферизации, использование fprintf одновременно несколькими процессами может привести к тому, что вывод процессов будет произвольным образом перемешан даже в пределах одной выводимой строки. Хотя каждая операция write сама по себе атомарна, при обработке форматной строки write может быть вызвана несколько раз. Если же включён режим полной буферизации, вывод будет производиться блоками по (по умолчанию) 512 байт. При этом граница выводимого блока может не совпадать с границей выводимой строки. Безопаснее всего использовать режим с построчной буферизацией, либо (что то же самое) сначала подготавливать строку в памяти с помощью функции sprintf, затем её выводить целиком с помощью fputs или fwrite.

Если файл открыт в режиме чтение и запись несколькими процессами (такие файлы чаще всего содержат бинарные данные) и активно модифицируется, система не гарантирует, что буфер в памяти будет отражать последние изменения в файле. Перед модификацией содержимого файла необходимо сделать пустую операцию над потоком, например

`fseek(f, 0, SEEK_CUR);`, а после модификации необходимо сохранить буфер вызовом `fflush`. Кроме того, необходимо использовать средства предотвращения одновременного доступа к файлу, которые будут рассмотрены в следующих разделах.

Дескриптор потока помимо информации о буферизации хранит номер низкоуровневого файлового дескриптора, с помощью которого и осуществляются все операции ввода/вывода. Номер файлового дескриптора можно получить с помощью функции `fileno`.

```
#include <stdio.h>
int fileno(FILE *stream);
```

Функция `fileno` всегда завершается успешно (если, конечно, передан правильный дескриптор потока) и не модифицирует переменную `errno`.

Функция `fdopen` позволяет создать дескриптор потока по уже открытому низкоуровневому файловому дескриптору.

```
#include <stdio.h>
FILE *fdopen(int fildes, const char *mode);
```

Здесь параметр `fildes` задаёт номер низкоуровневого файлового дескриптора, а параметр `mode` — режимы открытия, точно такие же, как у функции `fopen`. Режим, в котором был открыт файловый дескриптор `fildes` не должен противоречить режиму открытия, заданному в параметре `mode`. Указатель текущего положения в открывающемся потоке устанавливается в соответствие с указателем текущего положения файлового дескриптора. Если режим открытия указан как `"w"` или `"w+"`, содержимое файла не будет очищено, в отличие от функции `fopen`.

Дескриптор потока будет использовать файловый дескриптор с указанным номером, а не его копию, полученную с помощью системного вызова `dup`. Поэтому, после того, как файловый дескриптор был трансформирован в дескриптор потока, закрываться он должен как высокоуровневый поток, то есть с помощью функции `fclose`.

При успешном завершении функция `fdopen` возвращает указатель на новый дескриптор потока, а при неудаче возвращается `NULL`, и переменная `errno` устанавливается в код ошибки (например, неверный файловый дескриптор).

2.5 Временные файлы

Часто программе в процессе работы требуется вспомогательный файл. Такой файл создаётся программой, программой же используется и удаляется, а пользователь может ничего не знать об использовании дополнительных файлов. Такие файлы называются *вре́менными*, так как время их жизни не больше времени работы программы.

Первый вопрос, который возникает, когда нужно создать временный файл: *в каком каталоге файловой системы он должен быть создан?* Кажется, что можно создавать его в текущем каталоге, но это неверно. Во-первых, текущий каталог может быть закрыт на запись для пользователя, запустившего программу. Попытка создать временный файл в этом случае закончится неудачей. Во-вторых, если каталог открыт на запись, размер файлов и их количество могут быть ограничены (так называемая дисковая квота), и создание и работа с временными файлами может ещё больше ограничить возможности пользователя. В-третьих, если каталог располагается на сетевой файловой системе NFS, операции с ним могут оказаться достаточно медленными.

Традиционно в системах UNIX для временных файлов использовался каталог `/tmp` (в системе Solaris ещё `/var/tmp`). Этот каталог открыт на запись и чтение для всех пользователей, как правило, не квотируется и, как правило, располагается на локальной файловой системе. Для временных файлов может использоваться другой тип файловой системы, чтобы

повысить эффективность работы с файлами. Кроме того, область временных файлов может разделятся с областью подкачки системы. Система либо очищает каталог временных файлов при каждой перезагрузке, либо периодически удаляет из него все файлы, к которым не было доступа в течение определённого времени (на RedHat Linux две недели).

Общесистемный разделяемый каталог, такой как `/tmp`, является узким местом с точки зрения обеспечения безопасности системы. Если программа работает с ним неаккуратно, злоумышленник может, в нужные моменты времени создавая файлы в таком каталоге, испортить содержимое файлов пользователя, запустившего программу или даже записать в нужные злоумышленнику файлы (например, `/etc/passwd`) нужные ему данные (при условии, что программа запущена суперпользователем `root`). Поэтому использование каталогов типа `/tmp` нежелательно. Чтобы указать путь к каталогу, в котором следует создавать временные файлы, пользователь может установить переменную окружения `TMPDIR`, и программе следует использовать эту переменную, если она установлена. Если переменная окружения не установлена, программе следует использовать константу `P_tmpdir`, определённую в заголовочном файле `<stdio.h>`. Эта константа (макрос) определяет путь к общесистемному каталогу временных файлов. И только в крайнем случае программа должна использовать явный путь `/tmp`.

Второй вопрос: *как должен называться временный файл?* Фиксированное название (например, `myfile`) плохо. Во-первых, если одновременно будут работать два процесса, запущенные одним или разными пользователями, то либо два процесса будут одновременно использовать один и тот же файл, что приведёт к порче данных, либо работать будет только один процесс. Поэтому имя временного файла должно содержать некоторую непостоянную (случайную) компоненту, причём пространство имён, из которого выбирается эта компонента должно быть достаточно большим, чтобы исчерпание пространства имён было маловероятным. Например, случайное число в интервале от 0 до `RAND_MAX`, преобразованное в строковое представление, даёт хорошую случайную компоненту, при условии, что затравка генератора псевдослучайных чисел меняется при каждом запуске программы (например, зависит от времени).

Поскольку, как было сказано выше, в каталог временных файлов имеют право записывать все пользователи, в том числе потенциально враждебные, необходимо выполнять особенные процедуры при создании временных файлов. После того, как случайное имя временного файла сгенерировано, файл должен создаваться с помощью системного вызова `open` с режимом создания `O_EXCL`. Это гарантирует то, что файл будет успешно создан, если файл с таким именем ещё не существовал. Если системный вызов `open` завершился с ошибкой `EEXIST` (файл уже существует), нужно сгенерировать новое имя файла и повторить попытку, и так до тех пор, пока файл наконец не будет создан, либо не будет исчерпано пространство случайных имён. Права доступа у временного файла должны быть равны `0600`, то есть доступ всех пользователей, кроме владельца файла закрыт.

Рассмотрим сценарии, показывающие необходимость таких предосторожностей. Предположим, что временный файл не открывается с флагом `O_EXCL`. Тогда злоумышленник может узнать, какое имя будет иметь временный файл (например, подсмотрев текст программы). После этого он создаёт в каталоге временных файлов (допустим, это `/tmp`) символическую связь с этим именем, которая указывает на некоторый файл пользователя, который злоумышленник хочет испортить или модифицировать. В качестве таких файлов он может выбрать `.rhosts` или `.profile`, или просто любой файл, принадлежащий пользователю, который злоумышленник хочет испортить. Когда программа, неосторожно работающая с временными файлами, будет запущена, она откроет временный файл, не заметив, что этот файл уже существует. Символическая связь будет прослежена, и будет открыт и очищен

некоторый файл пользователя, содержащий, возможно, важные данные.

Если временный файл создаётся с большими правами доступа, чем 0600, злоумышленник может подсмотреть (или даже модифицировать) содержимое временного файла, что, возможно, нежелательно. Явное изменение прав доступа уже после открытия файла (с помощью системного вызова chmod) оставляет злоумышленнику возможность открыть файл в тот момент, когда open уже создал файл, а chmod ещё не был вызван (такая ошибка называется “race condition”).

Наконец, когда программа завершает работу, все ею созданные временные файлы должны быть удалены, чтобы они не засоряли каталог разделяемых файлов. Если программа не планирует несколько раз открывать и закрывать этот временный файл, его можно удалить сразу после открытия. Тогда файл будет существовать, но не иметь имени, а все блоки данных будут освобождены, как только будет закрыт последний файловый дескриптор, ассоциированный с этим файлом.

Часть функций по созданию временного файла принимает на себя функция mkstemp.

```
#include <stdlib.h>
int mkstemp(char *template);
```

Функция mkstemp создаёт временный файл по шаблону имени, заданному аргументом template. Последние 6 символов шаблона должны быть XXXXXX, и эти символы заменяются на символы, делающие всё имя временного файла уникальным. Файл создаётся с флагом O_EXCL в режиме чтение/запись (O_RDWR) и с правами доступа 0600. Поскольку строка template модифицируется, это не должна быть строковая константа (строковый литерал).

Функция возвращает файловый дескриптор временного файла при успешном завершении и -1 при ошибке. Переменная errno в этом случае может принимать следующие значения: EINVAL — последние 6 символов шаблона не равны XXXXXX. В этом случае строка template не меняется. EEXIST — исчерпано пространство случайных имён временных файлов (то есть, все возможные файлы уже существуют). В этом случае значение строки template неопределено. Кроме того, могут возвращаться ошибки системного вызова open.

Следующая программа иллюстрирует работу со временными файлами.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <limits.h>
 4 #include <unistd.h>
 5
 6 /* если P_tmpdir неопределён, установим его в /tmp */
 7 #ifndef P_tmpdir
 8 #define P_tmpdir "/tmp"
 9 #endif
10
11 int main(void)
12 {
13   char tmpn[PATH_MAX];
14   char *s;
15   int  fd;
16   FILE *f;
17   int  a, b, c;
18
19   /* если переменная окружения TMPDIR установлена,
20    * используем её, иначе - P_tmpdir */
```

```
21    if (!(s = getenv("TMPDIR"))) s = P_tmpdir;
22    /* формируем шаблон */
23    snprintf(tmpn, PATH_MAX, "%s/progXXXXXX", s);
24    /* создаём временный файл */
25    if ((fd = mkstemp(tmpn)) < 0) {
26      perror("mkstemp");
27      return 1;
28    }
29    /* удаляем его */
30    unlink(tmpn);
31    /* формируем дескриптор потока */
32    if (!(f = fdopen(fd, "r+"))) {
33      perror("fdopen");
34      return 1;
35    }
36    /* запишем в него что-нибудь */
37    fprintf(f, "%d %d %d\n", 1, 1, 2001);
38    /* ... */
39
40    /* f нельзя закрывать!
41     * устанавливаемся на начало файла для чтения */
42    fseek(f, 0, SEEK_SET);
43    /* читаем данные */
44    if (fscanf(f, "%d%d%d", &a, &b, &c) != 3) {
45      fprintf(stderr, "temporary file error\n");
46      return 1;
47    }
48    printf("Read values: %d %d %d\n", a, b, c);
49    /* временный файл больше не нужен, закрываем его */
50    fclose(f);
51    return 0;
52  }
```

3 Работа с файловой системой

Типичные задачи, которые возникают при работе с файловой системой, — это просмотр содержимого каталогов, рекурсивный обход всех каталогов файловой системы, получение информации о состоянии файла.

Поскольку операционная система, как правило, поддерживает большое количество типов файловых систем, которые сильно отличаются друг от друга по используемому способу хранения данных, ядро операционной системы должно предоставлять некоторый обобщённый интерфейс для доступа к содержимому каталогов. Этот интерфейс базируется на предположении, что каталог хранит только имена содержащихся в нём файлов. Вся остальная информация хранится в индексном дескрипторе. Если конкретная файловая система не имеет индексных дескрипторов (например, FAT), ядро всё равно будет эмулировать их наличие.

Для работы с каталогом используются библиотечные функции `opendir`, `closedir`, `readdir`, `seekdir`, `telldir`.

```
#include <sys/types.h>
#include <dirent.h>
```

```
DIR           *opendir(const char *name);
struct dirent *readdir(DIR *dir);
off_t          telldir(DIR *dir);
void           seekdir(DIR *dir, off_t offset);
int            closedir(DIR *dir);
```

Функция opendir ассоциирует с именем каталога, переданным ей в качестве аргумента, дескриптор каталога, указатель на который возвращается из функции для дальнейшего использования во всех функциях работы с каталогом. Дескриптор каталога имеет тип DIR, а работа с ним аналогична работе с дескриптором потока: в обоих случаях всегда используется указатель на структуру. В случае ошибки функция opendir возвращает NULL. Каждый открытый дескриптор каталога использует один файловый дескриптор, и, поскольку максимальное число одновременно открытых файловых дескрипторов в процессе ограничено, это нужно учитывать при рекурсивном обходе дерева файловой системы.

Функция closedir закрывает дескриптор каталога, передаваемый ей в качестве аргумента. При успешном завершении функция возвращает 0, а при ошибке — -1.

Функция telldir возвращает текущую позицию в каталоге. Следующий вызов readdir возвращает запись, начинающуюся с этой позиции. Функция seekdir позволяет установить позицию чтения. Аргумент offset должен быть либо значением, полученным от telldir, либо 0, что означает начало каталога.

Функция readdir считывает очередную запись в каталоге. Информация о записи возвращается в виде указателя на структуру struct dirent. Память под эту структуру выделена в дескрипторе каталога DIR, поэтому каждый вызов readdir переписывает старое содержимое структуры. Структура содержит поле d_name типа массива символов некоторого зависящего от операционной системы размера. Поле d_name содержит имя записи в каталоге, то есть последнюю компоненту пути к файлу. Чтобы сформировать полный путь к файлу или каталогу, нужно эту последнюю компоненту добавить к имени каталога, разделив их символом '/'.

Если функция readdir не может прочитать очередную запись (при ошибке или когда каталог закончился), функция возвращает NULL. Для простоты можно полагать, что если readdir вернул NULL, каталог не содержит больше записей.

Каждый каталог всегда содержит две записи с именами . и .., указывающие на сам этот каталог и на его родительский каталог, однако в некоторых файловых системах функция readdir может не выдавать вообще эти записи (несмотря на то, что функция stat к этим файлам всё применима), в других системах две записи могут не идти первыми. Поэтому для максимальной переносимости программа не должна делать предположений о том, что записи . и .. присутствуют и идут первыми.

Структура struct dirent может ещё содержать поле d_ino, которое содержит номер индексного дескриптора этой записи, но это поле не должно использоваться. Если необходимо получить номер индексного дескриптора, нужно использовать системный вызов stat (или lstat). В противном случае программа будет работать неправильно в каталогах, являющихся точками монтирования файловых систем. Дело в том, что функция readdir считывает каталог в том виде, в котором он хранится на диске, а система при монтировании файловых систем модифицирует отдельные записи в каталогах и индексные дескрипторы в памяти ядра, но не на диске.

Информацию о файле можно получить с помощью одного из системных вызовов: stat, lstat, fstat. Информация возвращается в структуре struct stat, адрес которой передаётся в эти системные вызовы. Поле st_mode структуры содержит права доступа к файлу и тип файла. Для проверки типа файла следует использовать специ-

альные макросы, например, S_ISDIR для проверки того, является ли запись каталогом. Если макросы недоступны, следует сначала наложить маску S_IFMT на значение поля st_mode, а затем сравнить получившееся значение с проверяемым типом записи. То есть, проверка на то, что запись является каталогом должна выглядеть следующим образом: s.st_mode & S_IFMT == S_IFDIR, где s — переменная типа struct stat. Чтобы получить права доступа к файлу, нужно на значение поля st_mode наложить маску 07777.

Поле st_ino содержит номер индексного дескриптора файла, а поле st_dev содержит номер устройства (файловой системы). Как было сказано ранее, каждый файл однозначно идентифицируется именно по этой паре: ⟨ st_ino, st_dev ⟩.

Полное описание структуры можно найти в приложении.

Ниже приведена программа, которая рекурсивно обходит всю файловую систему и печатает пути ко всем найденным файлам.

```
 1  #include <stdio.h>
 2  #include <sys/types.h>
 3  #include <sys/stat.h>
 4  #include <dirent.h>
 5  #include <limits.h>
 6
 7  void traverse(char const *dir)
 8  {
 9    char  name[PATH_MAX];
10    DIR  *d;
11    struct dirent *dd;
12    off_t o;
13    struct stat s;
14    char *delim = "/";
15
16    /* если в качестве каталога передан корневой каталог,
17     * разделитель не нужен */
18    if (!strcmp(dir, "/")) delim = "";
19    /* открываем каталог */
20    if (!(d = opendir(dir))) {
21      /* не смогли открыть */
22      perror(dir);
23      return;
24    }
25    /* считываем, пока dd не равен NULL, то есть
26     * пока есть записи в каталоге */
27    while ((dd = readdir(d))) {
28      /* пропускаем . и .. */
29      if (!strcmp(dd->d_name, ".") || !strcmp(dd->d_name, ".."))
30        continue;
31      /* формируем полный путь */
32      snprintf(name, PATH_MAX, "%s%s%s", dir, delim, dd->d_name);
33      /* получаем информацию о файле
34       * используем lstat, чтобы не зациклиться на символических
35       * связях */
36      if (lstat(name, &s) < 0) continue;
37      /* проверяем, что это каталог */
38      if (S_ISDIR(s.st_mode)) {
```

```
39        /* запоминаем текущее положение в каталоге */
40        o = telldir(d);
41        /* экономим файловые дескрипторы */
42        closedir(d);
43        /* вызываем себя рекурсивно */
44        traverse(name);
45        /* восстанавливаем старое положение */
46        if (!(d = opendir(dir))) {
47          perror(dir);
48          return;
49        }
50        seekdir(d, o);
51      } else {
52        /* печатаем путь */
53        printf("%s\n", name);
54      }
55    }
56
57    closedir(d);
58  }
59
60  int main(void)
61  {
62    traverse("/");
63    return 0;
64  }
```

1 Работа с процессами в POSIX-системах

Понятие «процесс» наряду с понятием «файл» относится к основным понятиям операционной системы. Под процессом можно понимать программу в стадии выполнения. С процессом в системе связано много атрибутов, например, страницы памяти, занимаемые процессом, или идентификатор пользователя. Процесс использует ресурсы системы, поэтому одна из основных задач ядра операционной системы — распределение ресурсов между процессами.

Каждый процесс имеет своё адресное пространство. Если в системе в текущий момент есть несколько готовых к выполнению процессов, они работают параллельно. На системе с одним процессором эти процессы разделяют время одного процессора, а на системе с несколькими процессорами каждый процесс может выполняться на своём процессоре, и они будут работать действительно параллельно. Параллелизм выполнения вносит принципиально новые моменты по сравнению с обычным последовательным программированием и делает параллельные программы значительно более трудоёмкими в разработке и отладке.

1.1 Создание процесса

Новый процесс создаётся с помощью системного вызова **fork**.

```
#include <unistd.h>

pid_t fork(void);
```

Вновь созданный (сыновний) процесс отличается только своим идентификатором процесса pid и идентификатором родительского процесса ppid. Кроме того, у нового процесса сбрасываются счётчики использования ресурсов, блокировки файлов и ожидающие сигналы. Сыновний процесс продолжает выполнять тот же код, что и родительский процесс. Отличить новый процесс от родительского можно только по возвращаемому системным вызовом `fork` значению. В сыновний процесс возвращается 0, а в родительский процесс возвращается идентификатор сыновнего процесса. Кроме того, функция `fork` возвращает число -1, когда новый процесс не может быть создан из-за нехватки ресурсов, либо из-за превышения максимального разрешённого числа процессов для пользователя или всей системы. Обычно функция `fork` используется следующим образом:

```
if ((pid = fork())) < 0) { /* сигнализировать об ошибке */ }
else if (!pid) {
  /* этот фрагмент кода выполняется в сыновнем процессе */
  /* ... */
  _exit(0);
} else {
  /* а этот фрагмент выполняется в родительском процессе */
}
```

Функции **getpid**, **getppid** позволяют получить идентификатор текущего или родительского процесса.

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

Функция `getpid` возвращает идентификатор текущего процесса, а функция `getppid` возвращает идентификатор родительского процесса. Если родительский процесс завершил своё

выполнение раньше сыновнего, «родительские права» передаются процессу с номером 1 — процессу init. Идентификатор процесса — это положительное число в интервале от 1 до (как правило) 32000. При создании нового процесса система назначает ему новый идентификатор. Когда pid достигнет максимального допустимого значения, счёт начнётся снова с 1.

Каждый процесс относится к некоторой группе процессов. В группу могут объединяться процессы, выполняющие с точки зрения программиста одно задание. Например, интерпретатор команд может объединить в одну группу процессов все процессы, связанные конвейером. В простейшем случае группа процессов может состоять из единственного процесса. Группа процессов идентифицируется положительным целым числом, совпадающим с идентификатором одного из процессов в этой группе. Даже после того, как этот процесс завершит работу, идентификатор группы процессов будет действительным до момента, пока не завершит работу последний процесс этой группы процессов.

Группа процессов может выступать как одно целое в системных вызовах отправления сигнала (они будут подробно рассмотрены ниже). Отрицательный идентификатор процесса в них обозначает, что сигнал посылается не одному процессу, а всей группе процессов, идентификатор группы которой совпадает с модулем аргумента. При работе с терминалом выделяются основная и фоновые группы процессов, и нажатие на комбинацию клавиш Ctrl-C посылает сигнал `SIGINT` сразу всей основной группе процессов. Другие сигналы, генерируемые при работе с терминалом (`SIGTSTP`, `SIGQUIT` и др.) так же посылаются всей основной группе процессов.

Идентификатор группы процесса сохраняется (наследуется) при создании нового процесса с помощью `fork`. Но, в отличие от идентификатора процесса, идентификатор группы процессов можно изменить.

```
#include <sys/types.h>
#include <unistd.h>
int   setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
```

Функция `getpgid` возвращает идентификатор группы процессов процесса с заданным идентификатором `pid`. Если аргумент `pid` равен 0, возвращается информация для текущего процесса. В случае ошибки возвращается -1.

Функция `setpgid` помещает процесс `pid` в группу процессов `pgid`. И `pid`, и `pgid` могут быть равны 0, что означает идентификатор текущего процесса. При необходимости создаётся новая группа процессов. Если `pid` не совпадает с идентификатором самого процесса, это должен быть идентификатор одного из сыновних процессов, который ещё не выполнил к этому моменту вызов `exec`. В случае успешного завершения функция возвращает 0. При неудаче функция возвращает -1, а переменная `errno` устанавливается в код ошибки.

`EACCES`	Сыновний процесс, заданный аргументом `pid` уже выполнил `exec`.
`EINVAL`	Недопустимое значение `pgid`.
`ENOSYS`	Система не поддерживает управление заданиями.
`EPERM`	Процесс, заданный аргументом `pid`, является лидером сессии, не находится в той же самой сессии, что текущий процесс, значение `pgid` не соответствует группе процессов в той же самой сессии, что текущий процесс.

1.2 Замещение тела процесса

Чаще всего новый процесс создаётся для того, чтобы запустить на выполнение какую-либо другую программу. В системах POSIX другую программу можно запустить, только заменив области памяти текущего процесса. Для этого служат функции семейства **`exec`**.

```
#include <unistd.h>

extern char **environ;

int execve(const char *filename, char *const argv[],
           char *const envp[]);

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int  execle(const char *path, const char *arg, ...,
            char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Основной функцией является системный вызов `execve`. Все остальные функции используют системный вызов `execve`.

Системный вызов **`execve`** запускает программу, заданную аргументом `filename`. Программа должна быть либо двоичной исполняемой программой, либо скриптом, начинающимся со строки вида

```
#! interpreter [arg]
```

В этом случае интерпретатор `interpreter` должен быть правильным путём к исполняемому двоичному файлу. Тогда программа будет запущена на выполнение строкой:

```
interpreter [arg] filename
```

`argv` — это массив строк — аргументов командной строки, передаваемый новой программе. `envp` — массив строк вида `key=value`, который передаётся как окружение в новую программу. И `argv` и `envp` должны завершаться нулевым указателем. Аргументы командной строки и переменные окружения доступны из функции `main` вызванной программы (если это — программа на Си), которая определяется следующим образом:

```
int main(int argc, char *argv[], char *envp[]);
```

Системный вызов `execve` не возвращает управление вызвавшей его программе в случае успеха, поскольку страницы кода, данных и стека процесса замещаются страницами из загружаемой программы. Запускаемая программа наследует идентификатор процесса pid и все открытые файловые дескрипторы, кроме тех, которые явно помечены флагом `FD_CLOEXEC`. Очищаются сигналы, ожидающие доставки. Обработчики всех неигнорируемых и не обрабатываемых по умолчанию сигналов сбрасываются в значения по умолчанию.

Если текущий процесс трассируется с помощью системного вызова `ptrace`, после успешного вызова `execve` ему посылается сигнал `SIGTRAP`.

Если у файла запускаемой программы установлен бит `SUID`, эффективный идентификатор пользователя изменяется на идентификатор владельца данного файла. Аналогично, если у файла запускаемой программы установлен бит `SGID`, эффективный идентификатор группы изменяется на идентификатор группы данного файла.

Если исполняемый файл использует динамические библиотеки, система вызывает динамический загрузчик, чтобы он подгрузил в адресное пространство процесса необходимые динамические библиотеки.

В случае ошибки `execve` возвращает -1, и переменная `errno` устанавливается в код ошибки. Возможные ошибки приведены в таблице.

EACCES	1) Запускаемый файл или интерпретатор скрипта не является регулярным файлом. 2) Нет прав на выполнение запускаемого файла или интерпретатора скрипта. 3) Файловая система не допускает выполнение файлов. 4) Нет права поиска в каком-либо каталоге, входящем в путь к `filename` или интерпретатору скрипта.
EPERM	Попытка запустить файл с установленным битом SUID или SGID в трассируемом процессе.
E2BIG	Список аргументов слишком велик.
ENOEXEC	Неверный формат исполняемого файла.
EFAULT	Какой-либо из аргументов — недопустимый адрес.
ENAMETOOLONG	Аргумент `filename` слишком длинный.
ENOENT	Файл `filename` или интерпретатор скрипта не существует.
ENOMEM	Недостаточно памяти ядра.
ENOTDIR	Некоторая компонента пути к файлу `filename` или интерпретатору скрипта не является каталогом.
ELOOP	Слишком много символических ссылок при прослеживании файла `filename` или интерпретатора скрипта.
ETXTBUSY	Файл `filename` или интерпретатор скрипта открыт на запись другим процессом.
EIO	Ошибка ввода/вывода.
ENFILE	Достигнуто максимальное количество открытых файлов в системе.
EMFILE	Достигнуто максимальное количество открытых файлов для процесса.

Таблица 1: Коды ошибок системного вызова `execve`

Функции **`execvp`**, **`execv`**, **`execle`**, **`execlp`**, **`execl`** являются оболочками над `execve`.

Аргумент `arg` и последующие аргументы в функциях `execl`, `execlp`, `execle` можно рассматривать как `arg0`, `arg1`, ..., `argn`. Все вместе они описывают список указателей на строки, представляющий собой список аргументов для вызываемой программы. По соглашению первый аргумент должен содержать имя запускаемого файла. Список аргументов **должен** завершаться нулевым указателем `NULL`.

Функции `execv` и `execvp` принимают массив указателей на строки, которые будут переданы как аргументы в вызываемую программу. По соглашению первый аргумент должен содержать имя запускаемого файла. Массив аргументов **должен** завершаться нулевым указателем `NULL`.

Функция `execle` позволяет задавать окружение для запускаемой программы. Массив указателей на строки, задающие переменные окружения, **должен** завершаться нулевым указателем `NULL`. Адрес массива должен идти следующим параметром функции `execle` после нулевого указателя, обозначающего конец списка аргументов, передаваемых вызываемой программе. Все другие функции берут переменные окружения из глобальной переменной `environ` текущего процесса. Эта переменная содержит все переменные, переданные текущему процессу, и новые переменные окружения, добавленные с помощью вызова `putenv`. Таким образом, все другие функции наследуют переменные окружения текущего процесса.

Функции `execlp` и `execvp` ищут файл `file` в пути поиска, если строка `file` не содержит символ `'/'`. Путь поиска определяется переменной окружения `PATH`, а если эта переменная не задана, используется путь по умолчанию `:/bin:/usr/bin`. Кроме того, эти функции по-особому реагируют на некоторые ошибки.

Если в доступе к файлу отказано (вызов execve вернул EACCES), эти функции продолжают поиск в оставшейся части пути поиска. Но если другого файла не найдено, функции возвращаются с кодом ошибки EACCES).

Если формат файла не был распознан (вызов execve вернул ENOEXEC), эти функции вызовут командный интерпретатор с путём к файлу в качестве первого аргумента. Если вызов закончится неудачей, дальнейший поиск в пути поиска не производится.

Таблица свойств функций семейства exec приведена ниже.

Имя	передача списка аргументов в параметрах	поиск в пути	задание нового окружения
execve	нет	нет	да
execvp	нет	да	нет
execv	нет	нет	нет
execle	да	нет	да
execlp	да	да	нет
execl	да	нет	нет

Таблица 2: Свойства функций семейства exec

1.3 Завершение работы процесса

```
#include <stdlib.h>
void exit(int status);
void _exit(int status);
void abort(void);
```

Процесс может завершить свою работу одним из следующих способов:

- Вызовом библиотечной функции exit(status);, где n — код завершения процесса.
- Возвратом из функции main. Этот способ эквивалентен предыдущему. В качестве кода завершения процесса используется значение, возвращаемое из main.
- Вызовом системного вызова _exit(status);, где n — код завершения процесса.
- Получением необрабатываемого, неигнорируемого и неблокируемого сигнала, который вызывает по умолчанию нормальное или аварийное завершение процесса.

Код возврата процесса — это некоторое целое число, обычно в интервале от 0 до 255 (один байт). Оно может использоваться родительским процессом, чтобы определить, завершился ли процесс успешно или нет. Соглашение о взаимодействии процессов предполагает, что код завершения 0 означает успешное завершение процесса, а все прочие коды — неуспешное завершение. Процесс может выработать код неуспешного завершения, если, например, он не смог открыть файл, необходимый для работы, или произошла другая ошибка, из-за которой процесс не смог выполнить ожидаемые от него действия. В случае незначительных ошибок, не влияющих значительно на выполнение процессом ожидаемых действий, следует вырабатывать код завершения 0.

Библиотечная функция exit отличается от системного вызова _exit тем, что функция exit выполнит корректное закрытие всех открытых дескрипторов потока и вызовет обработчики завершения работы процесса, зарегистрированные с помощью функции atexit.

Атрибут	**Наследование при `fork`**	**Наследование при `exec`**
Страницы кода программы	да, разделяются	нет
Страницы данных программы	да, копируются при записи	нет
Переменные окружения	да	возможно
Аргументы программы	да	возможно
Идентификатор пользователя (uid)	да	да
Идентификатор группы (gid)	да	да
Эффективный идентификатор пользователя (euid)	да	да, если не установлен бит SUID
Эффективный идентификатор группы (egid)	да	да, если не установлен бит SGID
Идентификатор процесса (pid)	нет	да
Идентификатор группы процессов (pgid)	да	да
Идентификатор родительского процесса (ppid)	нет	да
Приоритет процесса (nice)	да	да
Маска прав при создании файлов (umask)	да	да
Ограничения процессов (limits)	да	да
Счётчики использования ресурсов	нет	да
Сигналы, обрабатываемые по умолчанию	да	да
Игнорируемые сигналы	да	да
Перехватываемые сигналы	да	нет
Сигналы, ожидающие доставки	нет	нет
Файловые дескрипторы	да	да, если для дескриптора не установлен флаг `FD_CLOEXEC`
Блокировки файлов	нет	да
Рабочий каталог	да	да
Корневой каталог	да	да

Таблица 3: Наследование атрибутов при вызовах `fork`, `exec`

Системный вызов `_exit` вызывает немедленное завершение процесса, при этом содержимое незаписанных буферов файловых дескрипторов теряется. Ни одна из этих функций никогда не возвращает управление в программу.

Когда процесс завершается из-за получения сигнала, никакой код завершения не формируется, но процесс-родитель может узнать, что данный процесс завершился из-за сигнала, и узнать номер сигнала, вызвавшего завершение работы процесса. Обработчики завершения работы процесса и открытые дескрипторы потока не закрываются. Некоторые сигналы по умолчанию вызывают аварийное завершение процесса. Это — сигналы, генерируемые при некоторых фатальных ошибках работы процесса, например, при доступе к адресам, не отображённым в адресное пространство процесса. При аварийном завершении ядро записывает содержимое адресного пространства процесса и содержимое регистров центрального процессора в файл (обычно называемый `core`) в текущем каталоге. Это файл может потом использоваться для «посмертной отладки».

Функция abort вызывает фатальное завершение работы процесса (как будто бы он получил сигнал SIGABRT). Эта функция может использоваться, когда сама программа диагностировала состояние, когда она не может продолжить выполнение из-за ошибки в самой программе. abort не должна использоваться, когда процесс завершает работу из-за ошибки во входных данных или в параметрах командной строки.

В любом случае все ресурсы, связанные с процессом освобождаются, но запись в таблице процессов не удаляется для того, чтобы процесс-родитель смог прочитать статус завершения процесса. Процесс в таком состоянии, когда все ресурсы уже освобождены, и осталась только запись в таблице процессов, называется «зомби». Если процесс-родитель «не интересуется» судьбой сыновних процессов, они останутся зомби до тех пор, пока процесс-родитель не завершится. Тогда их родителем станет процесс 1 (init). Как только родительский процесс прочитает статус завершения сыновнего процесса-зомби, запись в таблице процессов уничтожается, и процесс окончательно прекращает своё существование.

1.4 Ожидание завершения сыновнего процесса

Простейшая функция, с помощью которой можно узнать состояние процесса, — функция **wait**.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *pstatus)
```

Функция wait приостанавливает выполнение текущего процесса до тех пор, пока какой-либо сыновний процесс не завершит своё выполнение, либо пока в процесс не поступит сигнал, который вызовет обработчик сигнала или завершит выполнение процесса. Если какой-либо сыновний процесс уже завершил выполнение («зомби»), функция возвращается немедленно, и процесс окончательно уничтожается.

Если указатель pstatus не равен NULL, функция wait записывает информацию о статусе завершения по адресу pstatus. Для получения информации о статусе завершения процесса могут использоваться следующие макросы. Они принимают в качестве параметра сам статус завершения процесса (типа **int**), а не указатель на него.

WIFEXITED(status) — принимает ненулевое значение, если процесс завершил своё выполнение нормально.

WEXITSTATUS(status) — этот макрос может быть использован, только если вызов макроса WIFEXITED дал ненулевое значение. Макрос выдаёт код возврата процесса, который был задан как аргумент функции exit или в операторе return функции main.

WIFSIGNALED(status) — принимает ненулевое значение, если процесс завершил своё выполнение в результате получения необрабатываемого сигнала, который по умолчанию вызывает завершение работы процесса.

WTERMSIG(status) — этот макрос может быть использован, только если вызов макроса WIFSIGNALED дал ненулевое значение. Макрос выдаёт номер сигнала, который вызвал завершение работы процесса.

Функция возвращает идентификатор завершившегося процесса, либо -1 в случае ошибки. В этом случае переменная errno устанавливается в код ошибки. Возможные коды ошибки приведены в таблице.

Больше возможностей предоставляет функция **wait4**, определённая следующим образом:

```
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait4(pid_t pid, int *pstatus, int options,
            struct rusage *prusage)
```

Функция `wait4` приостанавливает выполнение процесса до тех пор, пока сыновний процесс с заданным идентификатором `pid` не завершит выполнение, либо пока процесс не получит сигнал, который вызовет обработчик сигнала или завершение работы процесса. Если процесс с заданным идентификатором `pid` уже завершил работу к моменту вызова функции `wait4` (процесс-зомби), функция завершает работу немедленно.

Аргумент `pid` может принимать следующие значения:

`< -1` ожидать завершения любого сыновнего процесса, идентификатор группы процессов которого совпадает с абсолютным значением `pid`

`-1` ждать любой сыновний процесс (как функция `wait`)

`0` ждать любой сыновний процесс, идентификатор группы процессов которого совпадает с идентификатором группы процессов текущего процесса

`> 0` ждать сыновний процесс с заданным идентификатором процесса `pid`

Значение `options` образуется как побитовое «или» одной или нескольких следующих констант, либо может быть равен 0.

`WNOHANG` — завершить работу немедленно, если нет сыновних процессов, завершивших выполнение.

`WUNTRACED` — возвращать информацию и о процессах, которые были приостановлены, и статус которых ещё не был сообщён.

Если `pstatus` не равен `NULL`, информация о статусе завершения процесса записывается в область памяти, на которую указывает `pstatus`.

Помимо макросов, позволяющих получить статус завершения процесса, описанных выше, могут использоваться следующие макросы.

`WIFSTOPPED(status)` — возвращает ненулевое значение, если процесс, который вызвал завершение работы функции `wait4` приостановлен. Это возможно, только если `wait4` была вызвана с установленным флагом `WUNTRACED`.

`WSTOPSIG(status)` — возвращает номер сигнала, который вызвал приостановку выполнения процесса. Этот макрос может использоваться, только если `WIFSTOPPED` дал ненулевое значение.

Если `prusage` не равен `NULL`, информация об использовании процессом ресурсов системы записывается по адресу, указанному аргументом `prusage`.

Структура `struct rusage` определена в операционной системе Linux следующим образом:

```
struct rusage
{
        struct timeval ru_utime; /* время в режиме пользователя */
        struct timeval ru_stime; /* время в режиме ядра */
        long ru_maxrss;          /* максимальный размер в памяти */
        long ru_ixrss;      /* размер всех разделяемых страниц */
        long ru_idrss;      /* размер всех неразделяемых страниц */
        long ru_isrss;      /* размер страниц стека */
        long ru_minflt;   /* сбоев страницы без подкачки */
        long ru_majflt;   /* сбоев страницы с подкачкой */
```

```
        long ru_nswap;     /* количество полных откачек */
        long ru_inblock;          /* блочных операций ввода */
        long ru_oublock;          /* блочных операций вывода */
        long ru_msgsnd;           /* послано сообщений */
        long ru_msgrcv;           /* получено сообщений */
        long ru_nsignals;         /* получено сигналов */
        long ru_nvcsw;     /* "добровольных" переключений контекста */
        long ru_nivcsw;    /* "недобровольных" переключений */
};
```

Функция `wait4` возвращает идентификатор процесса, который вызвал завершение работы функции, либо -1 при ошибке (например, не существует сыновних процессов заданной категории), либо 0, если был использован флаг `WNOHANG`, и нет процессов, завершивших выполнение. В последних двух случаях переменная `errno` будет содержать код ошибки.

Возможные коды ошибок приведены в таблице.

`ECHILD`	процесс, заданный аргументом `pid`, не существует или не является сыновним процессом, либо у процесса вообще нет сыновних процессов
`ERESTARTSYS` `EINTR`	флаг `WNOHANG` не был установлен, и процесс получил неблокируемый сигнал или `SIGCHLD`.
`EINVAL`	какой-либо из аргументов имеет недопустимое значение.

Таблица 4: Коды ошибок для функций `wait`, `wait4`

1.5 Пример программы

Напишем реализацию функции `system` (стандартная функция, которая выполняет заданную команду).

```
 1  #include <stdio.h>
 2  #include <unistd.h>
 3  #include <sys/types.h>
 4  #include <sys/wait.h>
 5
 6  int system(char const *cmd)
 7  {
 8    int pid, status;
 9
10    if ((pid = fork()) < 0) {
11      /* ошибка */
12      perror("fork");
13      return -1;
14    } else if (!pid) {
15      /* child */
16      execl("/bin/sh", "/bin/sh", "-c", cmd, NULL);
17      /* ошибка */
18      perror("execl");
19      _exit(1);
20    }
21
22    /* parent */
```

```
23    wait(&status);
24    if (WIFSIGNALED(status)) return WTERMSIG(status) + 256;
25    return WEXITSTATUS(status);
26  }
```

2 Перенаправление стандартных потоков

В начале работы процесса у него, как правило, уже открыты три файловых дескриптора с номерами 0, 1, 2 — стандартный ввод, стандартный вывод и стандартный поток ошибок. Как правило, эти потоки связаны с терминалом, с которым работает командный интерпретатор, вызвавший процесс. Однако можно связать со стандартными потоками произвольные объекты, работа с которыми ведётся с помощью файловых дескрипторов (каналы, сокеты и пр.).

Для копирования открытого файлового дескриптора используется функция **dup2**, определённая следующим образом:

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

Функция `dup2` создаёт копию файлового дескриптора `oldfd`. После этого использование двух файловых дескрипторов полностью эквивалентно. Оба файловых дескриптора разделяют блокировки файлов, указатель текущего положения в файле и флаги открытия файлов. Например, если текущая позиция в файле была изменена с помощью `lseek` у одного из дескрипторов, текущая позиция изменится и у другого дескриптора.

Флаг «закрытия при exec» не разделятся.

Функция `dup2` создаёт копию `oldfd` в дескрипторе с номером `newfd`, при необходимости предварительно закрывая `newfd`.

Функция возвращает номер нового файлового дескриптора или -1, если функция не смогла создать новый файловый дескриптор. Переменная `errno` в этом случае содержит код ошибки.

2.1 Пример программы

Напишем программу, которая печатает все процессы, идентификатор пользователя которых 0 (root). Для получения списка процессов будем использовать вызов программы `ps`. Предположим, что второе число в строке, печатаемой командой `ps` как раз содержит идентификатор пользователя.

```
 1  #include <stdlib.h>
 2  #include <unistd.h>
 3  #include <stdio.h>
 4  #include <sys/types.h>
 5  #include <sys/wait.h>
 6  #include <string.h>
 7
 8  void pexit(char const *str)
 9  {
10    perror(str);
11    exit(1);
```

```
12 }
13
14 int main(void)
15 {
16   char tmppatt[] = "/tmp/XXXXXX";
17   int  fd, pid, status;
18   FILE *f;
19   char  buf[1024];
20   int   v1, v2;
21
22   /* создаём временный файл */
23   if ((fd = mkstemp(tmppatt)) < 0) pexit("mkstemp");
24   /* сразу удаляем */
25   if (unlink(tmppatt) < 0) pexit("unlink");
26   /* создаём новый процесс */
27   if ((pid = fork()) < 0) pexit("fork");
28
29   if (!pid) { /* сын */
30     /* перенаправляем стандартный вывод */
31     if (dup2(fd, 1) != 1) { perror("dup2"); _exit(1); }
32     /* закрываем старый дескриптор */
33     close(fd);
34     /* вызываем другой процесс */
35     execlp("ps", "ps", "axl", 0);
36     perror("execlp");
37     _exit(1);
38   }
39
40   /* отец */
41   /* ждём */
42   wait(&status);
43   /* ошибка выполнения сыновнего процесса */
44   if (!WIFEXITED(status) || WEXITSTATUS(status) > 0) exit(1);
45   /* перематываем файл на начало */
46   if (lseek(fd, 0, SEEK_SET) < 0) pexit("lseek");
47   /* связываем уже открытый файловый дескриптор с FILE * */
48   if (!(f = fdopen(fd, "r"))) pexit("fdopen");
49   while (fgets(buf, sizeof(buf), f)) {
50     if (strlen(buf) >= sizeof(buf) - 1) {
51       fprintf(stderr, "string too long\n");
52       exit(1);
53     }
54     /* печатаем все процессы, владелец которых - root */
55     if (sscanf(buf, "%d %d", &v1, &v2) == 2 && !v2)
56       printf("%s", buf);
57   }
58   fclose(f);
59   return 0;
60 }
```

1 Средства межпроцессного взаимодействия

Поскольку адресные пространства каждого процесса изолированы друг от друга, система должна предоставлять процессам средства взаимодействия.

Простейшее взаимодействие можно организовать, используя файлы в файловой системе. Даже в этом случае необходимо предусмотреть средства взаимной блокировки процессов на случай, когда один или несколько процессов записывают или считывают данные из одного файла.

Предположим, что некоторый процесс должен увеличить целое число, хранящееся в файле, на 1. Простейший фрагмент программы может выглядеть следующим образом (для простоты отсутствует проверка ошибок):

```
1    fd = open(FILENAME, O_RDWR);
2    read(fd, &value, sizeof(value));
3    value++;
4    lseek(fd, 0L, SEEK_SET);
5    write(fd, &value, sizeof(value));
6    close(fd);
```

Однако, такая работа с разделяемым ресурсом (а в данном случае файл является разделяемым ресурсом) содержит серьёзнейший изъян. Предположим, что одновременно два процесса начали модификацию содержимого файла, хранящего число 3. Тогда в зависимости от некоторых случайных (то есть непредсказуемых заранее) факторов значение числа в файле может увеличиться на 1 или на 2.

Такая ошибка работы с разделяемым ресурсом получила в англоязычной литературе название “race condition”. Чтобы избежать её, процесс должен каким-то способом ограничить на время доступ других процессов к разделяемому ресурсу. Операционные системы POSIX предоставляют достаточно богатый выбор средств блокировки: эксклюзивное создание файлов, семафоры, блокировки файлов через `flock`, `lockf` или `fnctl`.

Кроме того предоставляются разнообразные средства межпроцессного обмена, такие как анонимные и именованные каналы, сигналы, разделяемая память и очереди сообщений, файлы, отображаемые в память, сокеты.

Некоторые из этих средств будут рассмотрены в следующих разделах.

1.1 Эксклюзивное создание файлов и файлы-замки

Простейший способ организовать работу с разделяемым ресурсом предполагает использование файлов-замков. Файл-замок — это специальный файл, существование которого в файловой системе означает, что разделяемый ресурс заблокирован.

Работа процесса с разделяемым ресурсом может тогда выглядеть следующим образом: вначале создаётся файл-замок. Если файл не существовал и был успешно создан, процесс может работать с разделяемым файлом, не опасаясь, что другой процесс начнёт в это же время параллельную работу с этим же файлом. После того, как процесс завершит работу с разделяемым ресурсом, файл-замок удаляется. Если же процесс не смог создать файл-замок, он должен приостановить своё выполнение на некоторое небольшое время, после чего снова попытаться создать файл-замок.

Для создания файла-замка в простейшем случае можно использовать флаг `O_EXCL` системного вызова `open`. Этот флаг гарантирует, что если системный вызов `open` завершился успешно, никакой другой процесс не сможет создать файл с использованием флага `O_EXCL` до тех пор, пока файл не будет удалён.

```
1   while ((fl=open(LOCKFILE,O_CREAT|O_EXCL|O_RDWR,0600))<0) {
2     usleep(100000); /* задержка 0.1 секунды */
3   }
4   fd = open(FILENAME, O_RDWR);
5   read(fd, &value, sizeof(value));
6   value++;
7   lseek(fd, 0L, SEEK_SET);
8   write(fd, &value, sizeof(value));
9   close(fd);
10  close(fl);
11  unlink(LOCKFILE);
```

В полной программе ещё необходимо проверять, что эксклюзивное открытие завершилось с ошибкой EEXIST, потому что в противном случае ошибка в системном вызове open произошла не из-за того, что файл-замок уже существует.

К сожалению, эксклюзивное открытие файлов не работает, если файловая система на которой создаётся файл-замок, монтируется с другого компьютера (файловая система NFS). Формально, флаг O_EXCL поддерживается, но система не может гарантировать атомарность операции из-за особенностей протокола NFS.

Чтобы корректно создавать файлы-замки на таких дисках нужно использовать более сложную процедуру, состоящую из следующих шагов: создать в каталоге, в котором будет создан файл-замок, файл с уникальным именем (например, включающим в себя имя компьютера, идентификатор процесса и время создания); создать связь между этим файлом и файлом замка, причём результат работы системного вызова link нужно проигнорировать; с помощью системного вызова stat проверить, что число ссылок на файл увеличилось до 2. Только в этом случае создание файла-замка можно считать успешным. Пример фрагмента, устанавливающего замок, приведён ниже.

```
1   while (1) {
2     if ((fdl = open(UNIQUEFILE, O_CREAT | O_RDWR, 0600)) < 0) {
3       /* ошибка, нужно принять какие-то меры... */
4     }
5     close(fdl);
6     link(UNIQUEFILE, LOCKFILE);
7     if (stat(UNIQUEFILE, &statbuf) < 0) {
8       /* ошибка, нужно принять какие-то меры... */
9     }
10    if (statbuf.st_nlink == 2) break;
11    unlink(UNIQUEFILE);
12    /* создание замка неуспешно, нужно подождать */
13    usleep(100000);
14  }
15  unlink(UNIQUEFILE);
16  /* создание файла-замка успешно, можем идти дальше */
17  /* в конце нужно не забыть удалить файл-замок */
```

1.1.1 Достоинства и недостатки

Недостатки. Во-первых, создание файла-замка — операция достаточно сложная и занимающая достаточно много времени, особенно если файл создаётся на сетевом диске. Во-вторых, отсутствует возможность приостановить выполнение процесса до тех пор, по-

ка файл-замок не будет удалён. Поэтому процесс вынужден периодически пытаться создать файл-замок без гарантии, что в очередной раз операция завершится успешно. Как было сказано выше, если файловая система монтируется с другого компьютера, или даже когда файловая система экспортируется на другие компьютеры, NFS-сервер не может просто так удалять файлы, поскольку он не знает, используется ли этот файл каким-либо клиентом или нет. Поэтому очень частое создание и удаление файлов-замков может приводить к исчерпанию квоты пользователя на дисковое пространство и количество файлов.

Достоинства. Метод файлов-замков не требует, чтобы процессы, работающие с разделяемым ресурсом, находились в родственных отношениях. Процессы могут работать на разных компьютерах (при условии, что файл-замок создаётся корректно, как описано выше). Этот метод — единственный, который всегда работает в такой ситуации.

1.2 Анонимные каналы

Использовать временные файлы для передачи информации между двумя процессами во многих случаях слишком накладно.

Все операционные системы семейства UNIX предоставляют простейшее средство однонаправленной пересылки данных между процессами — анонимные каналы (часто их называют просто каналами — pipe).

Канал создаётся системным вызовом `pipe`.

```
#include <unistd.h>
int pipe(int filedes[2]);
```

При успешном завершении системный вызов возвращает 0, а при ошибке — `-1`, и переменная `errno` устанавливается в код ошибки. Неуспешное завершение `pipe` скорее всего означает, что переполнилась таблица открытых файлов у процесса либо у всей системы.

Системному вызову передаётся массив из двух элементов, в который этот системный вызов записывает номера двух файловых дескрипторов. Два файловых дескриптора связаны друг с другом. Данные записываются в файловый дескриптор `filedes[1]`. Записанные данные можно прочитать из файлового дескриптора `filedes[0]`.

Вообще, `filedes[1]` ссылается на начало канала, `filedes[0]` — на конец канала (если мы предполагаем, что данные «текут» от начала к концу). В результате создания нового процесса с помощью `fork`, либо копирований файлового дескриптора с помощью `dup`, `dup2`, на начало или конец канала может ссылаться несколько файловых дескрипторов у разных процессов. Начало канала (в которую ведётся запись) считается закрытым, когда закрыты все файловые дескрипторы, которые ссылаются на него. Аналогично, конец канала (чтение из канала) считается закрытым, когда закрыты все файловые дескрипторы, которые ссылаются на него. Чтобы установить момент, когда какой-то конец канала закрывается, ядро подсчитывает количество ссылок на канал.

После открытия канала читать и писать в него можно с помощью системных вызовов read и write. Можно связать с файловым дескриптором дескриптор потока `FILE*` с помощью функции `fdopen` и использовать высокоуровневые функции ввода/вывода. В последнем случае высокоуровневую буферизацию лучше всего отключить вызовом `setbuf(f, NULL);`, где `f` — дескриптор потока, связанного с каналом. Файловый дескриптор чтения может использоваться только для чтения из канала, а файловый дескриптор записи — только для записи в канал. Файловые дескрипторы канала не позволяют выполнять операцию `lseek`. Файловые дескрипторы канала закрываются обычным образом с помощью системного вызова `close`.

При работе с каналом системные вызовы `read` и `write` работают с некоторыми особенностями. Если канал пуст и соответствующий файловый дескриптор не был переведён в неблокирующий режим, системный вызов `read` приостанавливает выполнение процесса до тех пор, пока в канале не появятся данные, либо начало канала не будет закрыто. Если начало канала закрыто, системный вызов возвращает значение `0`. Когда в канале присутствуют данные, системный вызов завершается немедленно и возвращает только те данные, которые присутствуют в канале, то есть количество считанных байт может быть меньше размера буфера, переданного функции `read`.

Если при выполнении системного вызова `write` конец канала оказался закрытым, процесс, пытающийся выполнить `write` получает сигнал `SIGPIPE`. По умолчанию этот сигнал вызывает печать сообщения `Broken pipe` и завершение программы. Если сигнал `SIGPIPE` игнорируется, блокируется или обрабатывается процессом, `write` возвращает значение `-1` с кодом ошибки `EPIPE`. Каналу в памяти ядра соответствует буфер ограниченного размера. Поэтому когда `write` пытается записать в канал больше данных, чем в нём остаётся места, процесс будет приостановлен до тех пор, пока в канале не появится достаточно свободного места. В канале не сохраняются границы сообщений, то есть если один процесс сделал `write` два раза, другой процесс прочитает сразу все данные из канала.

Операции `write` и `read` являются для каналов атомарными, если размер данных не превосходит размера внутреннего буфера канала. Этот размер можно узнать из константы `PIPE_BUF`, определённой в файле `<limits.h>`. Например, для ядра Linux размер буфера канала равен по умолчанию 4096 байтов. Атомарность понимается в том смысле, что никакой другой процесс в системе не может наблюдать момент, когда операция выполнилась частично. Операция происходит как бы мгновенно. Если объём записываемых данных превышает константу `PIPE_BUF`, операция `write` может не быть атомарной.

1.2.1 Использование каналов

Каналы можно использовать для организации конвейерного выполнения команд, когда стандартный вывод одной команды попадает на стандартный ввод другой команды. Например, если мы хотим организовать конвейер

```
ls -l | wc -l
```

то есть передать на вход команде `wc -l` результат работы команды `ls -l`, мы можем сделать это следующей программой.

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <unistd.h>
 4  #include <sys/wait.h>
 5
 6  int main()
 7  {
 8    int fd[2];
 9    int pid1, pid2;
10
11    if (pipe(fd) < 0) { perror("pipe"); exit(1); }
12    if ((pid1 = fork()) < 0) { perror("fork"); exit(1); }
13    if (!pid1) {
14      /* child 1: "ls -l" */
15      dup2(fd[1], 1); close(fd[0]); close(fd[1]);
```

```
16      execlp("ls", "ls", "-l", NULL);
17      perror("execlp"); _exit(1);
18    }
19    if ((pid2 = fork()) < 0) { perror("fork"); exit(1); }
20    if (!pid2) {
21      /* child 2: "wc -l" */
22      dup2(fd[0], 0); close(fd[0]); close(fd[1]);
23      execlp("wc", "wc", "-l", NULL);
24      perror("execlp"); _exit(1);
25    }
26    close(fd[0]); close(fd[1]);
27    wait(NULL); wait(NULL);
28
29    return 0;
30  }
```

1.2.2 Использование каналов для синхронизации

Использование каналов предоставляет процессам возможность синхронизовать свою работу. Процесс-читатель может быть приостановлен до тех пор, пока в канале не появятся данные. Процесс-писатель будет приостановлен, пока в канале не освободится место для данных.

Кроме того, каналы могут использоваться для блокировки доступа к разделяемому ресурсу, например, разделяемому файлу, как было описано в первом разделе. Предположим, что процесс, который желает выполнить операцию, требующую блокировки ресурса, должен получить «жетон». После выполнения критической операции процесс сдаёт жетон. Жетоном будет некоторое (произвольное) число, хранящееся в канале. Получение жетона соответствует операции `read`, возврат жетона — операции `write`.

```
1    /* получить жетон */
2    read(fdp[0], &dummy, sizeof(dummy));
3    /* выполнить операцию */
4    fd = open(FILENAME, O_RDWR);
5    read(fd, &value, sizeof(value));
6    value++;
7    lseek(fd, 0L, SEEK_SET);
8    write(fd, &value, sizeof(value));
9    close(fd);
10   /* сдать жетон */
11   write(fdp[1], &dummy, sizeof(dummy));
```

Естественно, в примере, приведённом выше, в канале можно хранить само модифицируемое число, но, если разделяемые данные имеют сложную структуру и требуют произвольного доступа, хранение данных в канале становится слишком сложным.

1.2.3 Достоинства и недостатки

Достоинства. Операции работы с каналами не требуют модификации файловой системы и поэтому выполняются быстрее. Процесс может быть приостановлен (при помощи `read`) до тех пор, пока не появятся данные (ресурс не будет освобождён).

Недостатки. Взаимодействующие процессы должны быть порождены одним процессом, который создаст для них канал. Выполняться они могут только на одном компьютере.

1.3 Именованные каналы

Именованные каналы также называются FIFO (first-in first-out) по дисциплине работы с ними. Именованные каналы отличаются от анонимных каналов тем, что именованные каналы имеют точку привязки в файловой системе — имя. Кроме открытия, работа с именованными каналами не отличается от работы с обычными анонимными каналами.

Перед использованием именованный канал должен быть создан с помощью вызова функции `mkfifo`.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo ( const char *pathname, mode_t mode );
```

Здесь аргумент `pathname` задаёт имя создаваемого канала, а аргумент `mode` — права доступа на создаваемый именованный канал. Права доступа, как обычно, модифицируются маской прав открытия файла `umask` процесса. Если файл с таким именем существует, функция завершается с ошибкой `EEXIST`.

Как обычно, в случае успеха функция возвращает 0, а при неудачном завершении — `-1`, и в этом случае переменная `errno` будет установлена в код ошибки.

Для открытия именованного канала используется системный вызов `open`. Именованный канал может открываться только на чтение или только на запись, но не на чтение и запись одновременно. Естественно, процесс может открыть два файловых дескриптора: один на чтение, другой на запись. Процесс, открывающий именованный канал на запись, будет заблокирован до тех пор, пока не появится процесс, открывший именованный канал на чтение. То же самое верно и в обратную сторону.

Процесс, который собирается сам и читать из именованного канала, и писать в него, должен открывать канал специальным образом, чтобы избежать блокировки. Сначала процесс должен открыть канал на чтение в неблокирующемся режиме, после этого открыть канал на запись в нормальном режиме, и после сбросить неблокирующий режим на дескрипторе чтения из канала. Фрагмент программы, открывающей именованный канал, приведён ниже (все возможные ошибки игнорируются для краткости записи).

```
long flags;

/* открываем дескриптор чтения */
fdr = open(PIPE, O_RDONLY | O_NONBLOCK);
/* открываем дескриптор записи */
fdw = open(PIPE, O_WRONLY);
/* сбрасываем флаг неблокирующего доступа */
flags = fcntl(fdr, F_GETFL);
fcntl(fdr, F_SETFL & ~O_NONBLOCK)
```

Системные вызовы `read`, `write` и `close` работают точно так же, как и в случае обычных каналов. После того, как именованный канал стал ненужным, его нужно удалить из файловой системы с помощью системного вызова `unlink`.

1.3.1 Достоинства и недостатки

Достоинства. Взаимодействующие процессы не обязаны находиться в родственных отношениях. Хотя создание и открытие именованного канала требует некоторых операций с файловой системой, сам буфер обмена хранится в памяти ядра. Процесс может быть приостановлен до тех пор, пока в канале на появятся данные (или ресурс не будет освобождён).

Недостатки. Некоторые, в особенности устаревшие, системы не поддерживают именованные каналы. Именованные каналы неприменимы, когда процессы работают на разных компьютерах.

2 Функции завершения

Программа может зарегистрировать специальную функцию, которая будет вызвана, когда программа будет завершать своё выполнение по вызову `exit` или после возврата из функции `main`. Для этого используется вызов функции `atexit`.

```
#include <stdlib.h>

int atexit(void (*function)(void));
```

Если было зарегистрировано несколько функций, они будут вызваны в порядке, обратном порядку их регистрации.

Если программа работает с разделяемыми ресурсами (например, устанавливает файл-замок), рекомендуется регистрировать функции-обработчики завершения программы, которые будут освобождать все занятые программой ресурсы, которые не освобождаются системой автоматически (файлы-замки, жетоны, семафоры и пр.). Помимо этого программа ещё должна установить обработчики некоторых сигналов (о сигналах — на следующем занятии).

Если программа завершается системным вызовом `_exit`, обработчики завершения программы не вызываются.

3 Принудительное завершение работы процесса

Чтобы завершить процесс с заданным идентификатором процесса или группу процессов с заданным идентификатором группы процессов, нужно послать процессу или группе сигнал `SIGTERM`, как показано в следующем фрагменте программы:

```
#include <sys/types.h>
#include <signal.h>
kill(pid, SIGTERM);
```

В некоторых случаях, когда, например, сигнал `SIGTERM` игнорируется или обрабатывается неправильно, этот сигнал может не завершить процесс. Тогда процессу нужно послать сигнал `SIGKILL`:

```
kill(pid, SIGKILL);
```

Посылать сразу сигнал `SIGKILL` ни в коем случае нельзя, так как этот сигнал не даёт возможности прерываемому процессу выполнить завершающие действия, например, освободить ресурс. Между посылкой `SIGTERM` и `SIGKILL` должно пройти какое-то время (зависит от ситуации, например 1 секунда), которое отводится процессу на «добровольное» завершение.

После того, как сигнал был послан, необходимо прочитать статус завершения процесса с помощью какой-либо функции семейства `wait`.

1 Сигналы

Сигнал можно рассматривать как программное прерывание нормальной работы процесса. Ядро использует сигналы чтобы сообщить процессу об исключительных ситуациях, которые могут возникнуть в его работе. Некоторые сигналы сообщают об ошибках, таких как недопустимое обращение к памяти, другие сигналы сообщают об асинхронных событиях, таких как потеря связи с терминалом.

В операционной системе определено большое количество типов сигналов, каждый для своего типа события. Некоторые события делают нежелательным или невозможным продолжение нормальной работы процесса, такие сигналы по умолчанию завершают работу процесса. Другие сигналы, сообщающие о «безобидных» событиях, по умолчанию игнорируются.

Процесс может послать сигнал другому процессу. Это позволяет, например, родительскому процессу завершить выполнение сыновнего процесса. В другой ситуации два процесса могут таким образом синхронизовать своё выполнение.

Обработка сигналов — это область, где работа операционных систем-наследниц System V (сейчас наиболее распространены SCO Unix и Solaris) наиболее сильно отличается от работы систем семейства BSD. В дальнейшем изложении все системы первого типа будут обобщённо названы System V, а системы второго типа — BSD. Хотя при разработке Linux за образец была взята System V, обработка сигналов по умолчанию ведётся как в BSD. Все примеры проверялись на Linux, то есть в них используется семантика сигналов BSD.

1.1 Генерация сигналов

События, которые генерируют сигналы, делятся на три основные группы: ошибки, внешние события и явные запросы.

Ошибка означает, что программа сделала что-то недопустимое и не может продолжить своё выполнение. Не все типы ошибок генерируют сигналы (на самом деле, большинство ошибок не генерирует). Например, попытка открытия несуществующего файла — ошибка, но она не генерирует сигнал, вместо этого системный вызов `open` возвращает `-1`. Вообще, ошибки, которые связаны с библиотечными функциями и системными вызовами, сообщаются программе с помощью специальных возвращаемых значений. Ошибки, которые могут встретиться в любом месте программы, а не только в библиотечных функциях и системных вызовах генерируют сигналы. К таким ошибкам относятся деление на 0 и неверное обращение к памяти.

Внешнее событие обычно относится к операциям ввода/вывода или другим процессам. Например, получение данных при асинхронных операциях, срабатывание таймера, завершение сыновнего процесса.

Явный запрос предполагает использование функций, таких как `kill`, задача которых — сгенерировать сигнал.

Сигналы могут генерироваться *синхронно* или *асинхронно*. Синхронный сигнал относится к некоторому действию в программе и доставляется (если только не заблокирован) во время этого действия. Большинство ошибок генерируют синхронные сигналы. Посылка процессом сигнала самому себе также синхронна.

Асинхронные сигналы генерируются событиями, неподконтрольными процессу, который их получает. Такие сигналы могут приходить в процесс в непредсказуемые моменты времени. Посылка сигнала одним процессом другому процессу также асинхронна.

Каждый тип сигнала как правило синхронен или асинхронен. Например, сигналы, соответствующие ошибкам выполнения программы, синхронны. Но любой сигнал может быть синхронным или асинхронным, когда посылается явно.

Системный вызов `kill` позволяет послать сигнал процессу или группе процессов.

```
int kill(pid_t pid, int sig);
```

Если `pid` больше нуля, заданный сигнал `sig` посылается процессу с заданным идентификатором процесса.

Если `pid` равен 0, сигнал `sig` посылается всем процессам в группе процессов, к которой относится данный процесс.

Если `pid` меньше -1, сигнал `sig` посылается всем процессам в группе процессов с идентификатором `-pid`.

Наконец, если `pid` равен -1, сигнал `sig` посылается всем процессам с тем же эффективным идентификатором пользователя, что у текущего процесса.

Если номер сигнала равен 0, сигнал не посылается, но все возможные ошибки проверяются. Таким образом можно проверить, например, существование процесса с заданным `pid`.

Процесс может послать сигнал самому себе с помощью вызова вида `kill(getpid(), sig)`. Если этот сигнал не блокируется процессом, `kill` доставит хотя бы один сигнал из ожидающих доставки текущему процессу (это может быть другой сигнал, ожидающий доставки, вместо `sig`) перед возвратом из системного вызова `kill`.

В случае успеха системный вызов возвращает 0. Если сигнал посылается группе процессов, вызов `kill` заканчивается успешно, если сигнал был послан хотя бы одному процессу в группе. Нет способа узнать, какой из процессов получил сигнал, или получили ли его они все. В случае неудачи системный вызов возвращает значение `-1`, а переменная `errno` устанавливается в код ошибки, который может быть одним из следующих:

`EINVAL` Аргумент `sig` является недопустимым или неподдерживаемым номером сигналом.

`EPERM` Недостаточно прав послать сигнал процессу или группе процессов.

`ESCRH` Аргумент `pid` не является правильным номером процесса или группы процессов. Эта ошибка возвращается и в случае, когда заданному номеру соответствует процесс-зомби.

Действие функции `raise`, определённой следующим образом:

```
int raise(int sig);
```

эквивалентно вызову `kill(getpid(), sig)`.

1.2 Получение сигналов

После генерации сигнала он становится *ожидающим доставки*. Обычно он таковым остаётся в течение непродолжительного времени, после чего доставляется процессу-адресату. Однако если данный тип сигнала в текущий момент времени заблокирован процессом, сигнал может оставаться ожидающим доставки неограниченно долгое время до тех пор, пока сигналы этого типа не будут разблокированы. После разблокирования сигнал будет доставлен немедленно.

Доставка сигнала означает выполнение определённого действия. Для некоторых сигналов (`SIGKILL` и `SIGSTOP`) это действие фиксировано, но для большинства сигналов у программы есть выбор: игнорировать сигнал, обрабатывать сигнал или выполнять действие по

умолчанию для данного сигнала. Программа может задавать реакцию на сигнал с помощью вызовов `signal` или `sigaction`. Во время работы обработчика сигнала этот тип сигналов заблокирован.

Если для некоторого типа сигнала установлено игнорирование сигнала, любой сигнал такого типа будет сброшен сразу же после генерации, даже если этот тип сигналов в тот момент был заблокирован. Такой сигнал никогда не будет доставлен, даже если программа затем установит обработчик сигнала и разблокирует его.

Если процесс получает сигнал, который ни обрабатывается, ни игнорируется, выполняется действие по умолчанию. Каждый тип сигналов имеет своё действие по умолчанию. Для большинства сигналов действием по умолчанию является завершение процесса. Для некоторых сигналов, обозначающих «безобидные» события, действие по умолчанию состоит в том, чтобы ничего не делать. Обратите внимание, что такое действие по умолчанию не является игнорированием сигнала.

Когда сигнал приводит к завершению работы процесса, родитель процесса может определить причину завершения, проверив статус завершения процесса, возвращаемый функциями семейства `wait`. Сигналы, по умолчанию завершающие процесс, вызывают функцию ядра, аналогичную `_exit`, то есть не вызываются обработчики завершения, зарегистрированные по `atexit`, не записываются буферы дескрипторов потока `FILE`.

Сигналы, обозначающие ошибки выполнения процесса, имеют специальное свойство: когда процесс завершается по одному из таких сигналов, ядро записывает на диск дамп памяти (core dump), который хранит состояние процесса в момент остановки. Этот дамп можно просматривать с помощью отладчика, чтобы определить причину ошибки.

Дамп памяти имеет по умолчанию имя `core` и создаётся в текущем каталоге. Ядро не записывает дамп памяти в случае, когда выполняется хотя бы одно из следующих условий.

- Эффективный идентификатор пользователя или группы пользователей процесса не совпадает с реальным идентификатором пользователя или группы пользователей.
- У процесса отсутствуют права записи в текущий каталог процесса, если файл `core` не существует.
- У процесса отсутствуют права записи в файл `core` в текущем каталоге, если такой файл существует.
- Пользователь запретил создание файла дампа памяти с помощью команды `ulimit -c 0`.

Если сигнал, обозначающий ошибку выполнения процесса, посылается явным запросом и завершает процесс, ядро точно так же сохраняет дамп памяти, как будто бы сигнал был вызван непосредственно ошибкой.

1.3 Стандартные сигналы

В этом разделе перечислены имена стандартных типов сигналов с описанием, какое событие они обозначают. Каждое имя сигнала — это макрос, соответствующий положительному числу — номеру сигнала. Программа не должна делать никаких предположений о номерах сигналов и всегда ссылаться на сигналы, используя символические константы. Номера сигналов могут меняться от системы к системе, а имена сигналов стандартизованы.

Имена сигналов определены в заголовочном файле <signal.h>. Макрос NSIG даёт общее количество сигналов, определённых в системе. Поскольку сигналы нумеруются последовательно, его значение на 1 больше максимального номера сигнала.

Чтобы напечатать строку, описывающую сигнал (например, для SIGSEGV — "Segmentation fault"), можно использовать функцию strsignal, описанную следующим образом:

```
char *strsignal(int signum);
```

Здесь signum — номер сигнала.

Далее приведены таблицы сигналов, разбитых по группам. В столбце «реакция» определена реакция на данный сигнал по умолчанию. Реакция может описываться комбинацией букв, каждая из которых означает следующее:

- C Записать дамп памяти (core dump).
- T Завершить процесс.
- B Сигнал не может быть обработан, заблокирован или проигнорирован.
- S Остановить процесс.
- N Ничего не делать.
- R Продолжить процесс после остановки.

Имя	Реакция	Описание
		Программные ошибки
SIGFPE	CT	Фатальная арифметическая ошибка. Для целочисленных операций это может быть деление на 0, умножение минимального целого числа на -1. Для вещественных чисел существует много возможных ошибок. Например, переполнение, антипереполнение, деление на 0 и т. д.
SIGILL	CT	Недопустимая инструкция. Процесс пытался выполнить код, не соответствующий никакой инструкции процессора, или привилегированную инструкцию процессора. Обычно это означает, что процесс пытался выполнить какие-то данные. Это может произойти из-за переполнения массива, размещённого в стеке, или из-за неинициализированного указателя на функцию.
SIGSEGV	CT	Попытка чтения или записи по адресу, на который не отображается память, попытка записи в память, открытую только для чтения, попытка выполнения невыполняемой памяти. Чтение или запись по адресу 0 (NULL) может вызывать эту ошибку.
SIGBUS	CT	Неверное обращение к памяти, например, обращение по невыровненному адресу. Обращение по адресу 0 также может вызывать этот сигнал. Типы сигналов SIGSEGV и SIGBUS близки по смыслу, и точное деление между ними зависит от операционной системы и типа процессора.
SIGABRT	CT	Ошибка, выявленная самой программой, которая в этом случае вызвала функцию abort().
SIGTRAP	CT	Сигнал трассировки. Генерируется специальной инструкцией трассировки процессора и, возможно, другими инструкциями. Этот сигнал используется отладчиками.
		Завершение процесса

Продолжение на следующей странице

Имя	Реакция	Описание
SIGTERM	T	Завершить выполнение процесса. В отличие от SIGKILL этот сигнал может быть заблокирован, проигнорирован или обработан процессом. Посылка этого сигнала — стандартный способ «вежливо» попросить программу завершиться. Команда kill командного интерпретатора посылает по умолчанию этот сигнал.
SIGINT	T	Завершение процесса. Сигнал обычно посылается процессу, когда пользователь нажимает клавиши Ctrl-C (символ INTR).
SIGQUIT	CT	Аварийное завершение процесса. Сигнал обычно посылается процессу при нажатии клавиш Ctrl-\ на клавиатуре (символ QUIT). По умолчанию этот сигнал вызывает завершение процесса с дампом памяти. Его можно рассматривать как ошибку выполнения программы, распознанную пользователем.
SIGKILL	BT	Немедленное завершение процесса. Этот сигнал не может быть обработан, заблокирован или проигнорирован и, следовательно, всегда фатален. Сигнал SIGKILL нужно рассматривать как последнее средство завершить работу процесса, после того, как процесс не завершился по SIGTERM или SIGINT. Если сигнал SIGKILL не завершил процесс, это — ошибка ядра операционной системы.
SIGHUP	T	Сигнал посылается процессу, когда отсоединяется управляющий терминал данного процесса (например, разорванное сетевое соединение). Кроме того, сигнал посылается всем процессам в сессии, когда завершается лидер сессии. Завершение процесса-лидера сессии означает отсоединение всех процессов в сессии от управляющего терминала.
Срабатывание таймеров		
SIGALRM	T	Сработал таймер, измеряющий реальные (календарные) интервалы времени.
SIGVTALRM	T	Сработал таймер, измеряющий виртуальные интервалы времени (то есть время работы процесса в режиме пользователя).
SIGPROF	T	Сработал таймер, измеряющий время процессора, потраченное на данный процесс и в режиме пользователя, и в режиме ядра. Такой таймер используется для профилирования программы, отсюда и название.
Работа с процессами и управление заданиями (job control)		

Продолжение на следующей странице

Имя	Реакция	Описание
SIGCHLD	N	Этот сигнал посылается родительскому процессу, когда один из его сыновних процессов завершается или останавливается. Обработчик по умолчанию этого сигнала ничего не делает. Если пользовательский обработчик сигнала устанавливается в то время, когда есть сыновние процессы-зомби, будет ли этот обработчик вызван для процессов-зомби, зависит от конкретной операционной системы (Linux — нет). Для систем BSD, если сигнал SIGCHLD явно установлен как игнорируемый процессом, система не создаёт процессов-зомби, а сразу уничтожает их по завершению. Однако стандарт POSIX запрещает процессам явно игнорировать сигнал SIGCHLD. Программы, написанные для BSD, в этом случае не будут работать в других операционных системах. Новейший стандарт Unix98 снова разрешает игнорирование сигнала SIGCHLD в стиле BSD.
SIGSTOP	BS	Сигнал останавливает выполнение процесса. Он не может быть обработан, проигнорирован или заблокирован.
SIGCONT	R	Сигнал вызывает продолжение работы процесса, если он был остановлен. Этот сигнал не может быть заблокирован. Для него можно определить обработчик, но перед вызовом обработчика процесс всё равно будет продолжен. Для большинства программ не имеет смысла обрабатывать SIGCONT. Такие программы просто продолжают свою работу, не заметив, что они были приостановлены.
SIGTSTP	S	Интерактивный сигнал остановки выполнения процесса. В отличие от SIGSTOP этот сигнал может обрабатываться, блокироваться или игнорироваться. Сигнал генерируется, когда пользователь нажимает на клавиатуре Ctrl-Z (символ SUSP). Процесс должен обрабатывать этот сигнал, если требуется оставлять системные данные или файлы в целостном состоянии при остановке. Например, программа, которая отключает канонический режим ввода символов, может его снова включить при остановке.
SIGTTIN	S	Процесс не может считывать данные с терминала, когда он запущен как фоновое задание. Когда какой-либо процесс в фоновом задании пытается это сделать, все процессы в задании получают сигнал SIGTTIN. По умолчанию этот сигнал вызывает остановку выполнения процесса.
SIGTTOU	S	Аналогично SIGTTIN, но этот сигнал генерируется, когда процесс из фонового задания пытается записать на терминал или установить его режимы работы. По умолчанию запись на терминал для фоновых процессов разрешена, чтобы запретить её должен быть установлен режим TOSTOP терминала.

Ошибки операций ввода/вывода

Продолжение на следующей странице

Имя	Реакция	Описание
SIGPIPE	T	Попытка записи в канал (анонимный или именованный), у которого закрыт выходной конец. Если этот сигнал блокируется, игнорируется или обрабатывается, операция, вызвавшая ошибку, завершается с кодом ошибки EPIPE.
		Прочие сигналы
SIGUSR1	T	
SIGUSR2	T	Сигналы SIGUSR1 и SIGUSR2 предназначены для использования в прикладных программах произвольным образом.

Когда процесс остановлен, он не может получать сигналы, кроме SIGKILL и SIGCONT. Все посланные процессу сигналы будут сделаны ожидающими доставки. Как только процесс продолжит работу, сигналы будут ему доставлены. Когда процесс получает сигнал SIGCONT, все сигналы остановки, ожидающие доставки, будут сброшены. Аналогично, когда процесс получает сигнал остановки, все сигналы SIGCONT, ожидающие доставки, будут сброшены.

1.4 Работа с множествами сигналов

Аргументами многих функций, работающих с сигналами, могут быть множества сигналов. Например, sigprocmask позволяет изменить множество блокируемых сигналов. Для удобства работы с множествами сигналов стандартная библиотека предоставляет типы и функции. Они определены в заголовочном файле <signal.h>.

Тип sigset_t должен использоваться для хранения множества сигналов. Программа не должна предполагать, что этот тип эквивалентен некоторому целому типу, как изначально было в системах BSD.

```
int sigemptyset(sigset_t *pset);
int sigfillset(sigset_t *pset);
int sigaddset(sigset_t *pset, int signum);
int sigdelset(sigset_t *pset, int signum);
int sigismember(const sigset_t *pset, int signum);
```

Функция sigemptyset очищает множество сигналов, на которое указывает pset. Пустое множество не содержит ни одного сигнала.

Функция sigfillset полностью заполняет множество сигналов, на которое указывает pset. В получившееся множество включены все сигналы.

Функция sigaddset добавляет сигнал signum в множество сигналов, на которое указывает pset.

Функция sigdelset удаляет из множества сигналов, на которое указывает pset, сигнал signum.

Функция sigismember проверяет, присутствует ли в множестве сигналов, на которое указывает pset, сигнал с номером signum.

1.5 Установка обработчика сигнала

1.5.1 Функция signal

Простейшая функция, с помощью которой можно изменить обработку сигнала, это функция signal, описанная следующим образом:

```
#include <signal.h>
void (*signal(int, void (*handler)(int)))(int);
```

Если ввести специальный тип для указателя на функцию-обработчик сигнала, определение функции упростится:

```
typedef void (*sighandler_t)(int signum);
sighandler_t signal(int signum, sighandler_t handler);
```

Первый аргумент `signum` задает номер сигнала, обработку которого нужно изменить. Вместо номера сигнала предпочтительнее использовать символическое имя сигнала, как определено выше.

Второй аргумент `handler` определяет, как будет обрабатываться сигнал. Он может принимать следующие значения:

`SIG_DFL` устанавливает обработку сигнала на стандартную обработку по умолчанию (см. таблицы выше).

`SIG_IGN` задает, что сигнал должен игнорироваться. Программа не должна игнорировать сигналы, которые обозначают серьёзные программные ошибки или используются для завершения процесса. Если процесс игнорирует сигнал `SIGSEGV` и другие аналогичные сигналы, его поведение после ошибки неопределено (например, он может зациклиться на месте ошибки). Игнорировать запросы пользователя, такие как `SIGINT` и пр. — недружественно по отношению к пользователю.

Третья возможность — это задать функцию обработки сигнала. Эта функция будет вызвана, когда процесс получит сигнал.

Если обработка сигнала устанавливается в `SIG_IGN`, или когда обработка сигнала устанавливается в `SIG_DFL`, а обработка по умолчанию игнорирует сигнал, все сигналы этого типа, ожидающие доставки, будут сброшены, даже если они заблокированы. Такие сигналы никогда не будут доставлены, даже если впоследствии обработчик сигнала будет переустановлен, и сигнал будет разблокирован.

Функция `signal` возвращает предыдущий обработчик сигнала. Это значение может использоваться для того, чтобы восстановить старый обработчик, если это необходимо.

Функция `signal` присутствует в стандарте ANSI C, тем не менее её использование не рекомендуется. Исторически существовало два подхода к обработке сигналов: подход, реализованный в System V, и подход BSD, различия между которыми приведены в таблице ниже. Поэтому рекомендуется использовать более универсальную функцию `sigaction`, описанную ниже.

Свойство	System V	BSD
Блокировка сигнала	Текущий обрабатываемый сигнал не блокируется на время выполнения обработчика.	Текущий обрабатываемый сигнал блокируется на время выполнения обработчика.
Сброс обработчика	Обработчик переустанавливается на обработчик по умолчанию.	Обработчик не переустанавливается на обработчик по умолчанию.
Системные вызовы	Прерываются с кодом ошибки `EINTR`.	Перезапускаются.

1.5.2 Пример программы

В следующем примере программа завершит работу, когда три раза будет нажата комбинация Ctrl-C. Предполагается, что функция `signal` поддерживает семантику BSD, то есть

программа будет работать без изменений на операционных системах семейства BSD и на Linux.

```
#include <stdio.h>
#include <signal.h>

int cnt = 0;

void sigint_handler(int signo)
{
  printf("Ctrl-C pressed\n");
  if (++cnt == 3) {
    signal(SIGINT, SIG_DFL);
    raise(SIGINT);
  }
}

int main(void)
{
  signal(SIGINT, sigint_handler);
  while (1) {
    printf("Some string to print\n");
  }
  return 0;
}
```

Обратите внимание, как происходит выход из программы в случае, когда три раза подряд нажата комбинация Ctrl-C. Обработчик сигнала `SIGINT` устанавливается в стандартное значение по умолчанию (то есть выход из программы), затем сигнал `SIGINT` посылается самому себе с помощью функции `raise`. Это гарантирует, что родительскому процессу будет сообщена истинная причина завершения программы: получение сигнала `SIGINT`. Если бы программа завершала свою работу по вызову `exit(0)`, родительский процесс получил бы информацию, что наш процесс завершился нормально с кодом завершения `0`. Какой из двух вариантов завершения программы: посылка сигнала самой себе еще раз или выход по `exit` — предпочтительнее, зависит от конкретной ситуации.

Приведённая выше программа имеет серьёзный дефект, связанный с тем, что сигнал может поступить в программу и начать обрабатываться в любой момент времени. Если сигнал поступит, например, в середине работы функции `printf`, выполнение функции `printf` будет приостановлено, и начнется выполнение обработчика сигнала, который вновь вызовет `printf`. Получается, что функция будет вызвана вновь из середины самой себя. Далеко не все функции стандартной библиотеки будут корректно работать в этой ситуации (на самом деле, большинство не будет). Функция, которая может безопасно вызываться из обработчика сигнала, называется *асинхронно-безопасной*.

Чтобы устранить этот дефект можно поступить двумя способами: во-первых, на время выполнения функции `printf` можно заблокировать сигнал `SIGINT`. Блокирование сигналов будет рассмотрено ниже. Во-вторых, можно переписать программу так, чтобы обработчик сигнала просто устанавливал некоторую глобальную переменную, которую будет проверять основная программа. Основная программа будет печатать сообщение о нажатой клавише.

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

```c
#include <signal.h>

int cnt = 0;
volatile int flag = 0;

void sigint_handler(int signo)
{
  flag = 1;
  if (++cnt == 3) {
    signal(SIGINT, SIG_DFL);
    raise(SIGINT);
  }
}

int main(void)
{
  signal(SIGINT, sigint_handler);
  while (1) {
    if (flag) {
      printf("Ctrl-C␣pressed\n");
      flag = 0;
    } else {
      printf("Some␣string␣to␣print\n");
    }
  }
  return 0;
}
```

Ключевое слово **volatile** в определении переменной `flag` говорит компилятору, что значение переменной может быть изменено асинхронно, и поэтому компилятор не должен пытаться оптимизировать обращения к этой переменной, например, сохраняя её на регистрах процессора.

Новый вариант программы будет работать, как ожидается, при выводе в файл или на консоль, но всё равно не будет работать, когда вывод происходит в окно эмулятора терминала `xterm`. В этом случае проблема уже не в самой программе, а в том, как взаимодействуют X-сервер, эмулятор терминала `xterm` и программа. Сигнал `SIGINT` посылает программе не ядро операционной системы, а программа `xterm`. Перед тем, как послать сигнал, `xterm` сбрасывает считанные, но еще не выведенные на экран символы выходного потока. По всей видимости, на это свойство `xterm` из программы никак повлиять нельзя. Поэтому, если программа получила сигнал `SIGTERM` в момент интенсивной записи на экран, нельзя гарантировать, что запись будет целостна, то есть не будут пропущены отдельные символы. Единственный способ обойти эту проблему — отключить режим канонического ввода с терминала.

1.5.3 Функция `sigaction`

Системный вызов `sigaction` позволяет установить обработчик сигнала. Функция описана следующим образом:

```c
#include <signal.h>
```

```
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

Структура sigaction описана следующим образом:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

Функция sigaction устанавливает обработчик для сигнала signum, который задается в аргументе act, если этот аргумент не равен NULL. Если аргумент oldact не равен NULL, информация о предыдущем обработчике возвращается в структуру, на которую указывает oldact.

Поле sa_restorer сохранено для совместимости со старым программным обеспечением и не должно использоваться. Поле sa_sigaction позволяет задать обработчик сигнала, которому будет передаваться больше информации, чем обычно. Описание работы расширенного обработчика сигналов выходит за рамки данного документа.

Поле sa_handler задаёт обработчик сигнала. Он может быть равен SIG_IGN или SIG_DFL, что означает игнорирование сигнала или установку обработчика по умолчанию для сигнала.

Поле sa_mask задаёт множество сигналов, которые будут заблокированы на время работы обработчика сигнала.

Поле sa_flags позволяет определить режим, в котором будет обрабатываться сигнал. Значение этого поля получается объединением значений флагов, перечисленных ниже.

SA_NOCLDSTOP		Может задаваться только для сигнала SIGCHLD. В этом случае процесс не будет получать сигнал SIGCHLD, когда какой-либо из его сыновних процессов будет остановлен, то есть получит сигнал SIGSTOP, SIGTSTP, SIGTTIN или SIGTTOU.
SA_ONESHOT	или	Устанавливает обработку сигнала по умолчанию перед тем, как
SA_RESETHAND		будет вызван обработчик сигнала (семантика System V функции signal).
SA_RESTART		Устанавливает перезапуск системных вызовов после возврата из обработчика сигнала (семантика BSD функции signal).
SA_NOMASK	или	Не блокирует получение сигнала в обработчике этого сигнала
SA_NODEFER		(семантика System V функции signal).
SA_SIGINFO		Если установлен этот флаг, функция обработки сигнала будет получать 3 аргумента вместо одного. В этом случае адрес функции-обработчика сигнала должен содержаться в sa_sigaction, а не в sa_handler. Обсуждение расширенной обработки сигналов выходит за пределы данного документа.

Из всех этих флагов стандарт POSIX определяет только SA_NOCLDSTOP. Остальные флаги являются расширениями стандарта. В системах BSD и Linux поддерживаются все флаги.

1.5.4 Пример программы

Следующий пример делает то же самое, что и предыдущий пример, но использует функцию `sigaction`.

```
#include <stdio.h>
#include <signal.h>

int cnt = 0;
volatile int flag = 0;

struct sigaction orig_sigint_handler;

void sigint_handler(int signo)
{
  flag = 1;
  if (++cnt == 3) {
    sigaction(SIGINT, &orig_sigint_handler);
    raise(SIGINT);
  }
}

int main(void)
{
  struct sigaction sa;

  sa.sa_handler = sigint_handler;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;
  sigaction(SIGINT, &sa, &orig_sigint_handler);

  while (1) {
    if (flag) {
      printf("Ctrl-C␣pressed\n");
      flag = 0;
    } else {
      printf("Some␣string␣to␣print\n");
    }
  }
  return 0;
}
```

1.6 Сигналы и системные вызовы

Процесс может получить и обработать сигнал в тот момент, когда процесс ждёт или выполняет какую-либо операцию ввода-вывода, например `read` или `open`. Если обработчик сигнала возвращает управление процессу (не завершает его), возникает вопрос, что должна сделать система с идущей операцией ввода/вывода?

Если функция передачи данных, такая как `read` или `write`, прервана сигналом после того, как была считана или записана часть данных, ядро завершит выполнение соответствующего системного вызова и вернёт в качестве результата то количество байт, которое было

обработано к моменту прихода сигнала, обозначая частичный успех операции. Это не касается атомарных операций, таких как чтение из канала или запись в канал, чтение/запись очереди сообщений и пр.

Если сигнал пришёл в момент, когда операция ввода/вывода еще не начала собственно обмен данными, например, когда системный вызов `read` ждёт ввода очередного символа с терминала, поведение системы отличается в зависимости от того, реализует ли она семантику System V, стандартизованную впоследствии POSIX, или семантику BSD.

Стандарт POSIX определяет, что в этом случае системный вызов должен завершиться с ошибкой `EINTR`. Этот подход даёт гибкость, но обычно достаточно неудобен для программиста. Приложение в этом случае должно проверять код ошибки `EINTR` после каждого системного вызова, который может вернуть эту ошибку, и перезапускать системный вызов при необходимости. Если программист забудет это сделать, программа будет работать неправильно.

Семантика BSD предполагает, что прерванный системный вызов будет автоматически перезапущен. В этом случае системный вызов никогда не вернёт ошибку `EINTR`.

Функция `sigaction` позволяет выбрать, какая семантика предпочтительнее для обработчика данного сигнала. Если указан флаг `SA_RESTART`, прерванный системный вызов будет автоматически перезапущен, а если флаг не указан, системный вызов завершится с ошибкой `EINTR`.

1.7 Блокирование сигналов

Временная блокировка сигнала с помощью `sigprocmask` позволяет предотвратить прерывание нормальной работы процесса в критической секции кода. Если сигнал поступит процессу в это время, он будет доставлен позднее, когда процесс его разблокирует.

Временная блокировка может быть полезна, когда и обработчик сигнала, и основная программа работают с некоторой разделяемой структурой данных (в примере выше — это функция `printf`, которая работает с дескриптором потока `stdout`). Если работа с этой структурой не атомарна, обработчик сигнала может быть запущен, когда структура находится в нецелостном состоянии, что может приводить к самым неприятным последствиям. Чтобы предотвратить прерывание программы в момент модификации разделяемой структуры, критическая секция кода должна быть защищена командами блокирования опасного сигнала.

Кроме того, сигнал должен блокироваться, когда программа должна выполнить некоторое действие только тогда, когда сигнал не пришёл. В противном случае программа будет содержать временну́ю ошибку (timing error). Пример такой программы будет разобран ниже в разделе, посвящённом ожиданию прихода сигнала.

Множество сигналов, которые в текущий момент заблокированы процессом, называется маской сигналов процесса. Каждый процесс имеет свою маску сигналов. Когда вызовом `fork()` создаётся новый процесс, он наследует маску сигналов родительского процесса.

Функция `sigprocmask` позволяет изменить маску сигналов процесса.

```
int sigprocmask(int mode,
                const sigset_t *pset,
                sigset_t *poldset);
```

Аргумент `mode` задаёт, какая операция будет выполнена с маской сигналов. Он должен быть равен одному из следующих значений:

SIG_BLOCK	Множество сигналов, на которое указывает pset, добавляется к маске сигналов процесса.
SIG_UNBLOCK	Множество сигналов, на которое указывает pset, удаляется из маски сигналов процесса. Допускается удалить из маски неблокируемый сигнал.
SIG_SETMASK	Маска сигналов копируется из множества сигналов, на которое указывает pset.

Если указатель poldset не равен NULL, то по этому адресу копируется старое значение маски сигналов процесса. Если нужно только узнать текущую маску процесса, но не изменять её, аргумент pset можно задать равным NULL.

Следующий пример блокирует сигнал SIGINT на время выполнения функции printf.

```
#include <stdio.h>
#include <signal.h>

int cnt = 0;

void sigint_handler(int signo)
{
  printf("Ctrl-C␣pressed\n");
  if (++cnt == 3) {
    signal(SIGINT, SIG_DFL);
    raise(SIGINT);
  }
}

int main(void)
{
  sigset_t blockset;

  sigemptyset(&blockset);
  sigaddset(&blockset, SIGINT);
  signal(SIGINT, sigint_handler);
  while (1) {
    sigprocmask(SIG_BLOCK, &blockset, 0);
    printf("Some␣string␣to␣print\n");
    sigprocmask(SIG_UNBLOCK, &blockset, 0);
  }
  return 0;
}
```

1.8 Ожидание прихода сигналов

Простейший способ приостановить выполнение процесса до поступления сигнала заключается в использовании функции pause.

```
#include <unistd.h>
int pause(void);
```

Функция pause приостанавливает выполнение процесса до поступления сигнала, который не блокируется и не игнорируется. Если сигнал обрабатывается по умолчанию, он дол-

жен вызывать завершение работы процесса. Если поступление сигнала запускает функцию обработки сигнала, которая возвращает управление в процесс, функция `pause` возвращается с кодом завершения -1 и кодом ошибки `EINTR` даже в случае, когда включена семантика перезапускаемых системных вызовов.

Однако кажущаяся простота этой функции может приводить к серьёзным ошибкам. Рассмотрим программу, которая при поступлении сигнала `SIGINT` печатает сообщение. Возможный вариант этой программы представлен ниже:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sigint_handler(int signo)
{
}

int main(void)
{
  signal(SIGINT, sigint_handler);
  while (1) {
    pause();
    printf("Ctrl-C␣pressed\n");
  }
  return 0;
}
```

Эта программа содержит *временну́ю ошибку* (timing error). Предположим, что сигнал поступит в момент, когда программа выполняет оператор `printf`. В этом случае, когда управление вернётся на вызов `pause`, сигнал будет уже обработан, и функция `pause` «повиснет» на неопределённое время. Сигнал был потерян.

Программа может быть модифицирована так, что обработчик сигнала устанавливает некоторую переменную, которая потом проверяется в основном цикле.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

volatile int flag;
int cnt;

void sigint_handler(int signo)
{
  flag = 1;
}

int main(void)
{
  signal(SIGINT, sigint_handler);
  while (1) {
    if (!flag) pause();
    flag = 0;
    printf("Ctrl-C␣pressed\n");
```

```
  }
  return 0;
}
```

Модифицированная программа на самом деле содержит ту же самую ошибку, только в более завуалированной форме. Предположим, что сигнал поступил в тот момент, когда значение переменной `flag` уже было проверено, но функция `pause` ещё не была вызвана. В этом случае сигнал будет потерян, и программа опять зависнет на неопределённое время. Что самое неприятное, такая ошибка может проявлять себя крайне редко и практически невоспроизводима.

Корректный способ ждать прихода сигнала реализуется с использованием функции `sigsuspend`.

```
int sigsuspend(const sigset_t *pset);
```

Функция заменяет маску сигналов процесса маской, на которую указывает аргумент `pset`, затем приостанавливает выполнение процесса до поступления сигнала, который либо вызывает завершение процесса, либо обрабатывается процессом.

Если сигнал обрабатывается процессом, и обработчик возвращает управление в процесс, функция `sigsuspend` также возвращается, при этом восстанавливается маска сигналов процесса, которая была на момент вызова функции.

Программа, корректно ожидающая поступления сигнала `SIGINT`, приведена ниже.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

volatile int flag;
int cnt;

void sigint_handler(int signo)
{
  flag = 1;
}

int main(void)
{
  sigset_t mask, oldmask;

  sigemptyset(&mask);
  sigaddset(&mask, SIGINT);
  signal(SIGINT, sigint_handler);

  while (1) {
    sigprocmask(SIG_BLOCK, &mask, &oldmask);
    while (!flag)
      sigsuspend(&oldmask);
    flag = 0;
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    printf("Ctrl-C␣pressed\n");
  }
  return 0;
```

```
}
```

Обратите внимание, что значение переменной `flag` сбрасывается в 0 при заблокированном сигнале, так как в противном случае сигнал может поступить в процесс в момент, когда сигнал уже разблокирован, но переменная `flag` ещё не сброшена. Сигнал в этом случае будет потерян.

1.9 Слияние сигналов

Рассмотрим следующий пример:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sigint_handler(int signo)
{
  printf("Ctrl-C␣pressed\n");
}

int main(void)
{
  sigset_t mask;

  sigemptyset(&mask);
  sigaddset(&mask, SIGINT);
  signal(SIGINT, sigint_handler);

  while (1) {
    sigprocmask(SIG_BLOCK, &mask, NULL);
    sleep(2);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
  }
  return 0;
}
```

Вызов `sleep` здесь заменяет некоторую операцию, требующую значительного времени.

Сколько бы раз в процесс не поступил сигнал `SIGINT` в то время, пока он заблокирован, обработчик сигнала будет вызван после разблокировки не более одного раза. Ядро не поддерживает очередь сигналов, ожидающих доставки, а хранит для каждого типа сигнала единственный бит, означающий наличие недоставленного сигнала соответствующего типа. Эти биты все вместе образуют маску сигналов процесса, ожидающих доставки. Поэтому сигналы не могут использоваться там, где необходимо считать количество их поступлений. В этом случае необходимо использовать другой механизм межпроцессного взаимодействия.

В современных системах введён новый тип сигналов — сигналы реального времени, для которых ядро поддерживает очередь ожидающих доставки сигналов, но традиционные сигналы, рассмотренные в этом документе к ним не относятся.

1.10 Пример программы

Рассмотрим следующую программу. Два процесса обмениваются друг с другом сообщениями в стиле пинг-понг, то есть первый посылает второму число 1, на что второй отвечает первому числом 2, и так далее. Для обмена данными используется единственный канал, а процессы синхронизуются посылкой друг другу сигнала SIGUSR1. Главный процесс породит два подпроцесса, которые и будут обмениваться данными, а он сам будет просто ожидать завершения обоих процессов.

Первая проблема, которая возникает, как передать каждому процессу идентификатор другого процесса. Мы обойдем ее поместив оба процесса в одну группу процессов. Каждый процесс будет посылать сигнал SIGUSR1 всей группе процессов.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

volatile int flag;
void handler(int signo)
{
  flag = 1;
}

void do_work(int pgid, int sig, int *p)
{
  sigset_t bs, os, b2;
  int      val;

  sigemptyset(&bs);
  sigaddset(&bs, SIGUSR1);
  sigaddset(&bs, SIGUSR2);
  sigemptyset(&b2);
  sigaddset(&b2, SIGINT);
  sigaddset(&b2, SIGTERM);
  sigprocmask(0, NULL, &os);
  while (1) {
    /* ожидаем прихода либо SIGUSR1, либо SIGUSR2 */
    /* так как один из них игнорируется, мы всегда получим нужный */
    sigprocmask(SIG_BLOCK, &bs, NULL);
    while (!flag)
      sigsuspend(&os);
    flag = 0;
    sigprocmask(SIG_UNBLOCK, &bs, NULL);
    read(p[0], &val, sizeof(val));
    /* мы не хотим, чтобы нас прервали на середине печати */
    sigprocmask(SIG_BLOCK, &b2, NULL);
    printf("Process␣%d␣got␣value␣%d\n", getpid(), val);
    fflush(stdout);
    sigprocmask(SIG_UNBLOCK, &b2, NULL);
    val++;
```

```
    write(p[1], &val, sizeof(val));
    kill(-pgid, sig);
  }
}

int main(void)
{
  int pid1, pid2;
  int pgid;
  int p[2];
  int val = 0;

  pgid = getpid();
  signal(SIGUSR1, handler);
  signal(SIGUSR2, SIG_IGN);
  if (pipe(p) < 0) { perror("pipe"); exit(1); }
  if ((pid1 = fork()) < 0) { perror("fork"); exit(1); }
  if (!pid1) {
    setpgid(0, pgid);
    do_work(pgid, SIGUSR2, p);
    _exit(0);
  }
  setpgid(pid1, pgid);
  signal(SIGUSR1, SIG_IGN);
  signal(SIGUSR2, handler);
  if ((pid2 = fork()) < 0) { perror("fork"); exit(1); }
  if (!pid2) {
    setpgid(0, pgid);
    do_work(pgid, SIGUSR1, p);
    _exit(0);
  }
  setpgid(pid2, pgid);
  write(p[1], &val, sizeof(val));
  close(p[0]); close(p[1]);
  signal(SIGUSR2, SIG_IGN);
  kill(-pgid, SIGUSR1);
  sleep(1);
  signal(SIGTERM, SIG_IGN);
  kill(-pgid, SIGTERM);
  wait(0); wait(0);

  return 0;
}
```

Обратите внимание, как основная программа последовательно переустанавливает обработчики сигналов `SIGUSR1` и `SIGUSR2`. Это необходимо делать, чтобы избежать временны́х ошибок. В противном случае главный процесс мог бы послать сигнал `SIGUSR1` первому процессу, который бы еще не успел установить обработчик, и сигнал был бы либо потерян, либо проигнорирован, что в любом случае привело бы к зависанию программы. Вызов `setpgid` делается и в отце, и в каждом из сыновних процессов опять-таки чтобы избежать временны́х ошибок.