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 выражение `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
```

`-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**

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 ключевых 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

`-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

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 будет возвращать значение 1.

Прототип функции, объявленный с классом памяти **`static`** на глобальном уровне, виден 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.